



# Introdução à Computação Gráfica

Marcel P. Jackowski  
[mjack@ime.usp.br](mailto:mjack@ime.usp.br)

Aula #12



# Objetivos

---

- Buffers
- Misturas ("Blending")
  - Translucidez
    - Neblina ("Fog")
  - Antiserrilhamento
- Sistemas de partículas

# Buffers de OpenGL

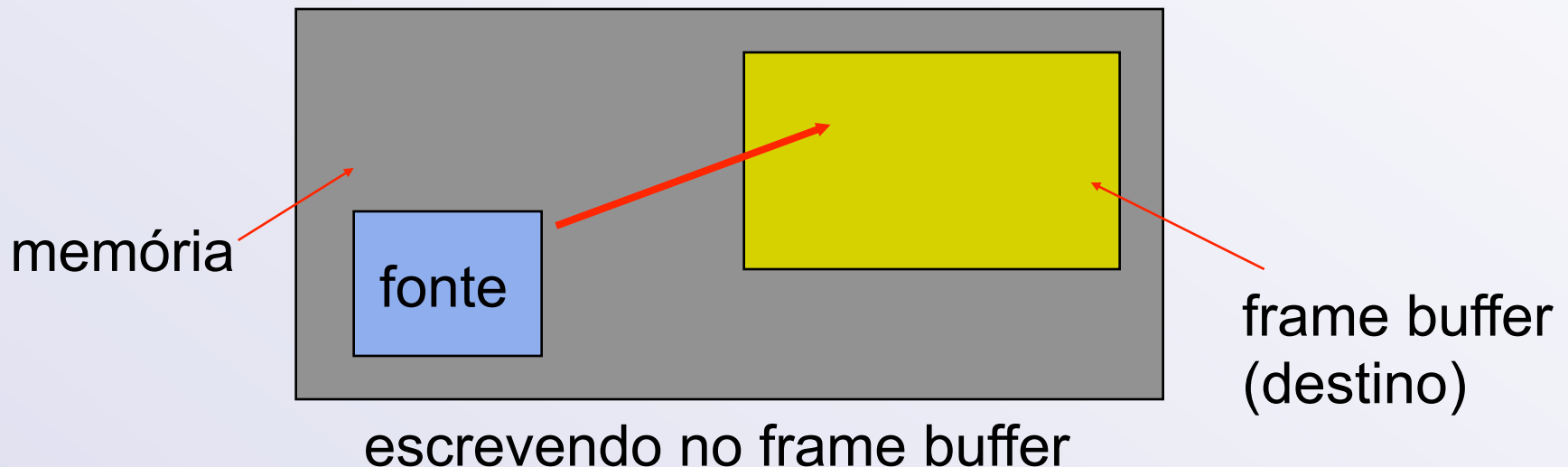
---

- Buffers coloridos que podem ser visualizados:
  - Front
  - Back
  - Auxiliary
  - Overlay
- **Depth**
- **Accumulation**
  - Buffer de alta resolução
- **Stencil**
  - Armazena máscaras

# Escrevendo nos buffers

---

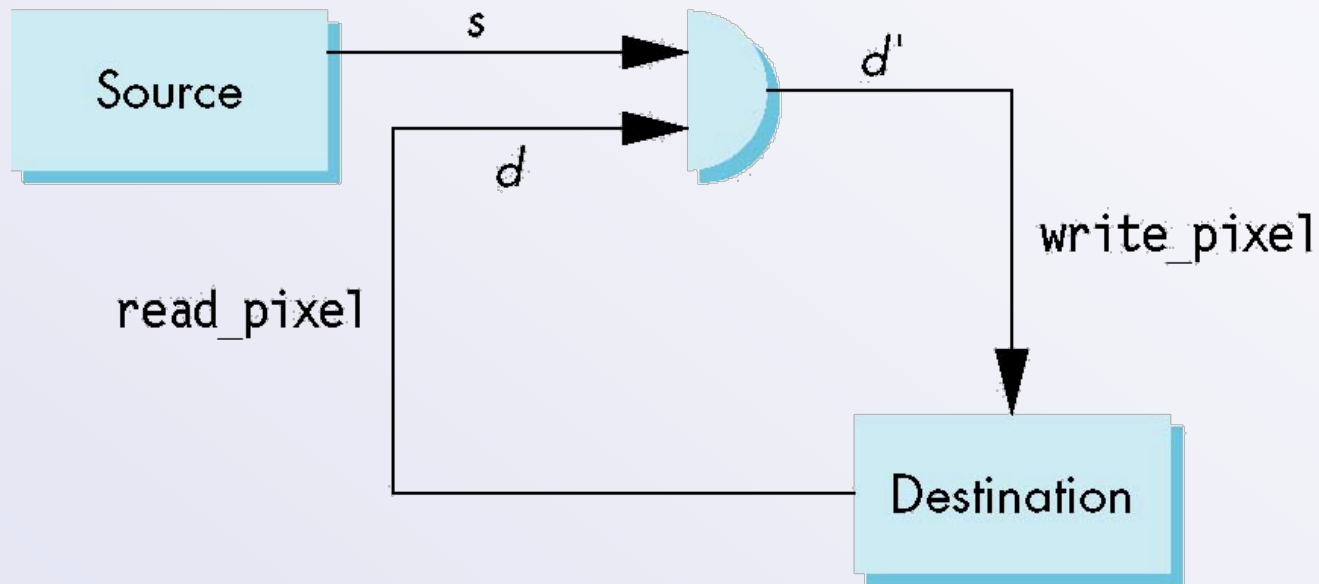
- Podemos considerar a memória como uma matriz bidimensional de pixels;
- Podemos então ler e escrever blocos retangulares de pixels
  - Operações de “*Bit block transfer*” (*bitblt*)
- O frame buffer é parte desta memória



# Modelo de escrita

---

Ler pixel de destino antes de re-escrevê-lo com o pixel origem:



# Modos de escrita

- Pixels origem e destino são combinados bit a bit
- 16 diferentes funções são possíveis

<i>s</i>	<i>d</i>		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0		0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1		0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0		0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1		0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

REPLACE      XOR      OR

# Modo XOR

---

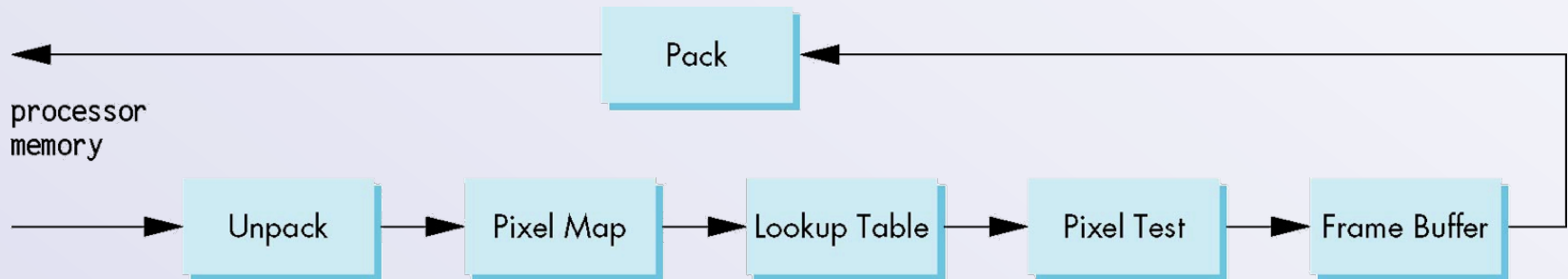
- Em OpenGL:
  - `glEnable(GL_COLOR_LOGIC_OP);`
  - `glLogicOp(GL_XOR);`
- XOR é especialmente útil para a troca de blocos de memória (e.g. menus) que são armazenados fora da tela
  - $S = S \text{ XOR } M$
  - $M = S \text{ XOR } M$
  - $S = S \text{ XOR } M$

Opcode	Resulting Operation
GL_CLEAR	0
GL_SET	1
GL_COPY	s
GL_COPY_INVERTED	~s
GL_NOOP	d
GL_INVERT	~d
GL_AND	s & d
GL_NAND	~(s & d)
GL_OR	s   d
GL_NOR	~(s   d)
GL_XOR	s ^ d
GL_EQUIV	~(s ^ d)
GL_AND_REVERSE	s & ~d
GL_AND_INVERTED	~s & d
GL_OR_REVERSE	s   ~d
GL_OR_INVERTED	~s   d

# O pipeline de pixels

---

- O OpenGL possui um pipeline separado para pixels
  - Na escrita
    - Mover pixels da memória para o frame buffer
    - Conversão entre formatos
    - Mapeamentos, Buscas e Testes
  - Na leitura
    - Conversão entre formatos





# Posição de rasterização

---

- O OpenGL mantém uma *posição de rasterização* como parte do estado
- Especificado através de `glRasterPos*()`
  - `glRasterPos3f(x, y, z);`
- A posição de rasterização é uma entidade geométrica
  - Passa através do pipeline de geometria
  - Eventualmente usado como posição 2D em coordenadas da tela
  - Esta posição representa onde a próxima primitiva do tipo raster será desenhada dentro do frame buffer

# Seleção de buffers

---

- O OpenGL pode desenhar (escrever) ou ler de qualquer um dos buffers coloridos (front, back, auxiliary)
- O buffer default de escrita é o back buffer
- Podemos alterar isto como os comandos:
  - `glDrawBuffer` and `glReadBuffer`
- O formato dos pixels no frame buffer é diferente do formato na memória principal;
  - Estes dois tipos de memória residem em lugares diferentes
- Precisamos então realizar um “packing” e “unpacking”
  - O desenho ou leitura pode se tornar mais lentos

# Bitmaps

---

- O OpenGL trata pixels de 1-bit (*bitmaps*) de um modo diferente de pixels multi-bit (*pixelmaps*)
- Bitmaps são máscaras que determinam se o pixel correspondente no frame buffer será desenhado com a *cor de raster corrente*
  - 0 → cor é inafetada
  - 1 → cor é alterada de acordo com o modo de escrita
- Bitmaps são úteis para textos raster
  - GLUT font: `GLUT_BITMAP_8_BY_13`

# Cor de rasterização

---

- Mesma cor especificada por `glColor* ()`
- Fixada pela última chamada à `glRasterPos* ()`

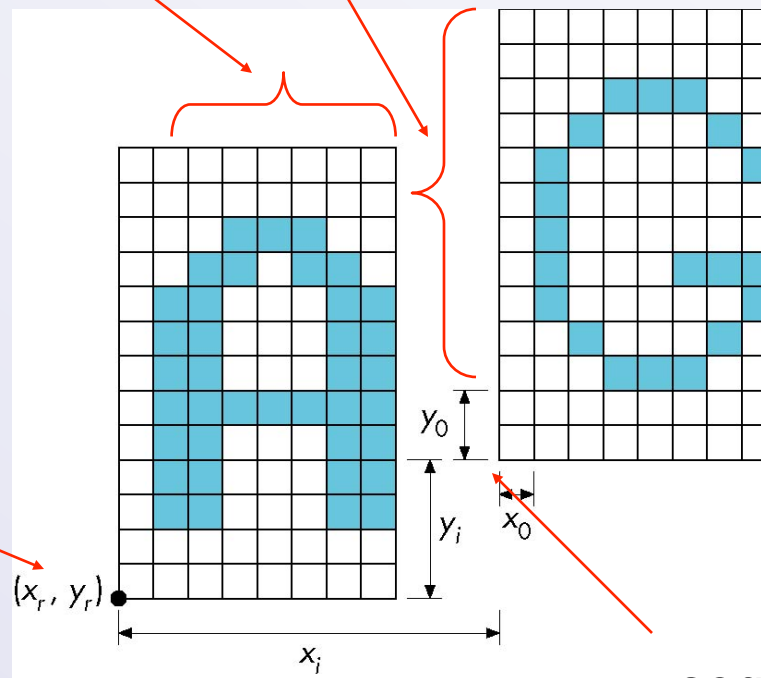
```
glColor3f(1.0, 0.0, 0.0) ;  
glRasterPos3f(x, y, z) ;  
glColor3f(0.0, 0.0, 1.0) ;  
glBitmap(.....) .  
glBegin(GL_LINES) ;  
    glVertex3f(... .)
```

- Geometria é desenhada em azul
- Bitmap é desenhado na cor vermelha

# Desenhando bitmaps

`glBitmap(width, height, x0, y0, xi, yi, bitmap)`

primeira posição



distância da  
posição raster

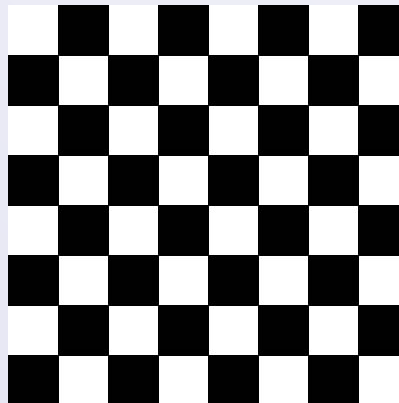
incrementos na  
posição raster  
depois do  
desenho do  
bitmap

segunda posição raster

# Desenhando um tabuleiro de xadrez

---

```
GLubyte wb[2] = {0x00, 0xff};  
GLubyte check[512];  
int i, j;  
for(i=0; i<64; i++)  
    for (j=0; j<64; j++)  
        check[i*8+j] = wb[(i/8+j)%2];  
  
glBitmap(64, 64, 0.0, 0.0, 0.0, 0.0, check);
```



# Mapas de pixels (pixmap)

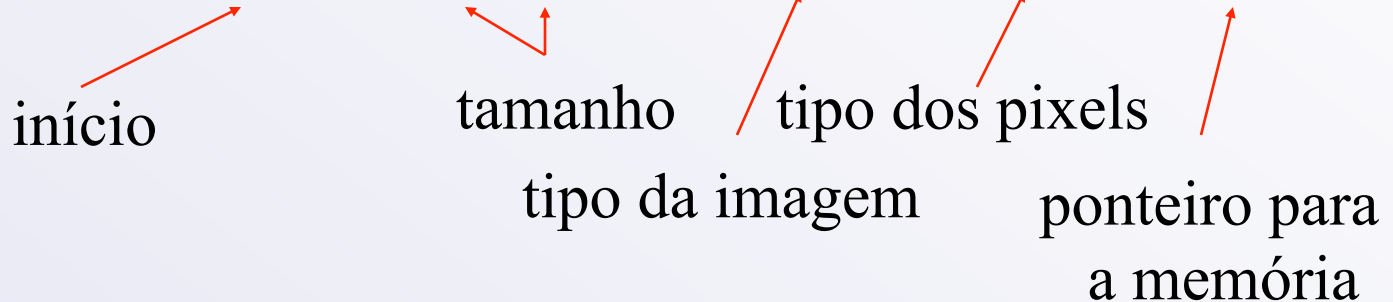
---

- O OpenGL trabalha com arrays retangulares de pixels chamados de pixmaps ou imagens
- Pixels são armazenados em bytes (8 bit)
  - Luminância (tons de cinza) ocupa 1 byte/pixel
  - RGB 3 bytes/pixel
- Três funções principais:
  - Desenho: memória do sistema para o frame buffer
  - Leitura: frame buffer para a memória do sistema
  - Cópia: frame buffer para frame buffer

# Leitura

---

`glReadPixels(x,y,width,height,format,type,myimage)`

  
início                      tamanho      tipo da imagem      tipo dos pixels      ponteiro para  
a memória

```
GLubyte myimage[512][512][3];  
glReadPixels(0,0, 512, 512, GL_RGB,  
             GL_UNSIGNED_BYTE, myimage);
```

`glDrawPixels(width,height,format,type,myimage)`

desenha pixmap na posição raster atual



# Formatos

---

- Normalmente trabalhamos com imagens em formatos padrões (JPEG, TIFF, PNG)
- Como ler/escrever tais imagens em OpenGL?
- Infelizmente, não existe suporte a estes formatos em OpenGL
  - Precisamos escrever funções para leitura e escrita de tais formatos

# Formato PPM (Portable Pixel Format)

---

- Formato simples: ASCII ou binário
- Cada arquivo de imagem consiste de um header seguido dos pixels
- Cabeçalho:

```
P3
# comentário 1
# comentário 2
...
# comentário n
linhas colunas valormáximo
pixels
```

# Lendo o cabeçalho

---

```
FILE *fd;
int i, k, nm;
char c, nome[100];
float s;
int red, green, blue;

printf("Entre com o nome do arquivo\n");
scanf("%s", nome);

fd = fopen(nome, "r");
fscanf(fd, "%[^\\n] ", nome);
if (nome[0] != 'P' || nome[1] != '3') {
    printf("%s não é um arquivo PPM!\n", nome);
    exit(0);
}
```

# Lendo o cabeçalho

---

```
fscanf(fd, "%c", &c) ;  
while(c == '#')  
{  
    fscanf(fd, "%[^\n]", nome) ;  
    printf("%s\n", nome) ;  
    fscanf(fd, "%c", &c) ;  
}  
ungetc(c, fd) ;
```

pula os comentários

# Lendo os pixels

---

```
fscanf(fd, "%d %d %d", &n, &m, &k);  
printf("linhas=%d, colunas=%d, valmax=%d\n", n, m, k);
```

```
nm = n*m;  
image=malloc(3*sizeof(GLuint)*nm);  
s=255./k; ← fator de escala
```

```
for (i=0; i<nm; i++)  
{  
    fscanf(fd, "%d %d %d", &red, &green, &blue);  
    image[3*nm-3*i-3]=red;  
    image[3*nm-3*i-2]=green;  
    image[3*nm-3*i-1]=blue;  
}
```

# Escalando os pixels

---

Podemos escalar a imagem no pipeline:

```
glPixelTransferf(GL_RED_SCALE, s);  
glPixelTransferf(GL_GREEN_SCALE, s);  
glPixelTransferf(GL_BLUE_SCALE, s);
```

Dependendo do processador, talvez precisemos trocar a ordem dos bytes quando transferimos a imagem da memória para o frame buffer. Para isso, usamos

```
glPixelStorei(GL_UNPACK_SWAP_BYTES, GL_TRUE);
```

# Callback de display

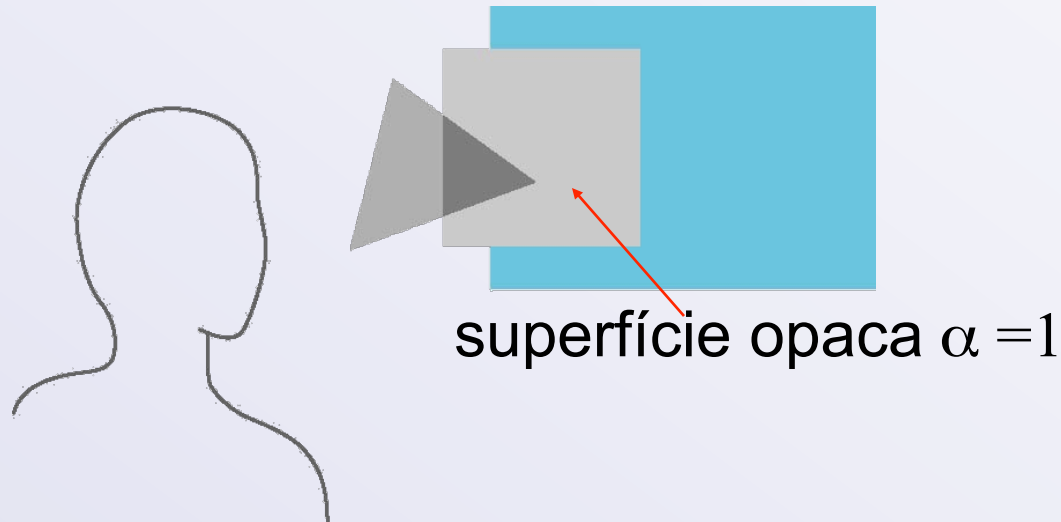
---

```
void display()  
{  
    glClear(GL_COLOR_BUFFER_BIT);  
    glRasterPos2i(0,0);  
    glDrawPixels(n, m, GL_RGB, GL_UNSIGNED_INT, image);  
    glFlush();  
}
```

# Opacidade e transparência

---

- Superfícies opacas não permitem a passagem da luz (não existe refração)
- Superfícies transparentes deixam toda a luz passar
- Superfícies translúcidas deixam passar alguma luz  
translucência =  $1 - \text{opacidade } (\alpha)$

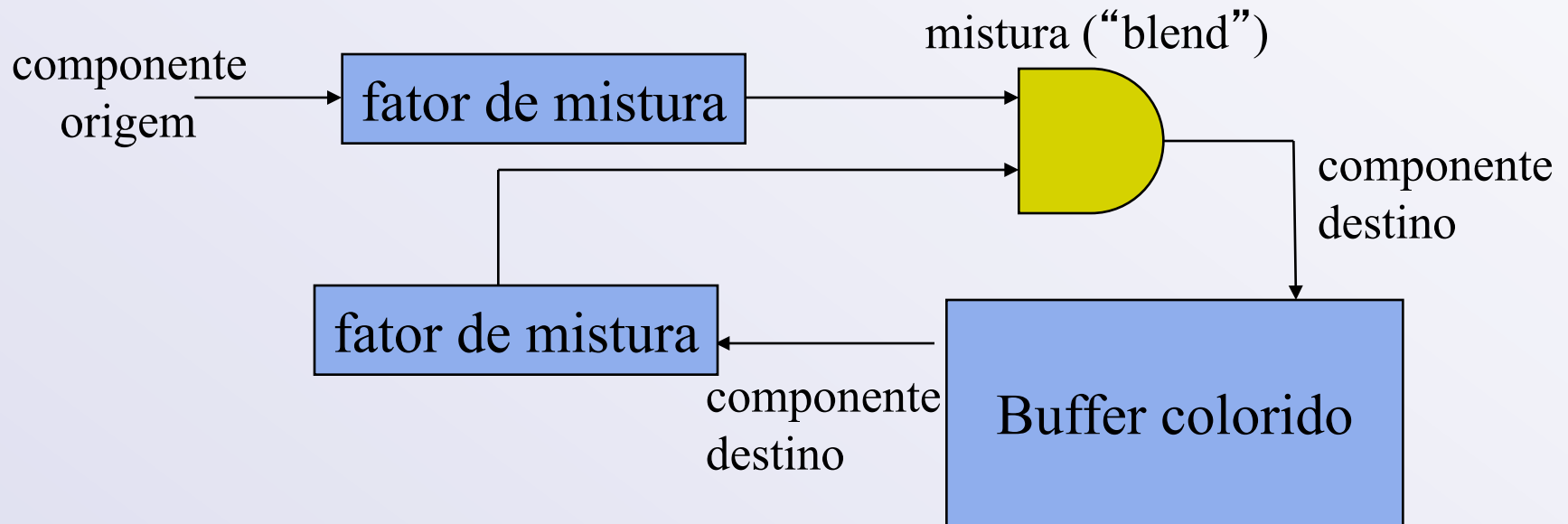




# Modelo de escrita

---

- Usar o componente A do RGBA (or  $\text{RGB}\alpha$ ) para armazenar a opacidade
- Durante a renderização, podemos expandir o nosso modelo de desenho para usar valores RGBA



# Equação de mistura

---

- Podemos definir fatores de mistura para a origem e destino para cada componente RGBA

$$\mathbf{s} = [s_r, s_g, s_b, s_a]$$

$$\mathbf{d} = [d_r, d_g, d_b, d_a]$$

Suponha que as cores origem e destino sejam

$$\mathbf{b} = [b_r, b_g, b_b, b_\alpha]$$

$$\mathbf{c} = [c_r, c_g, c_b, c_\alpha]$$

Combine-as da seguinte forma:

$$\mathbf{c}' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_\alpha s_\alpha + c_\alpha d_\alpha]$$

# Composição e Mistura em OpenGL

---

- Devemos habilitar e escolher os fatores de mistura para origem e destino

```
glEnable(GL_BLEND)
```

```
glBlendFunc(fator origem, fator_destino)
```

- Somente um subconjunto de fatores são suportados:
  - GL\_ZERO, GL\_ONE
  - GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA
  - GL\_DST\_ALPHA, GL\_ONE\_MINUS\_DST\_ALPHA

# Exemplo

---

- Começamos com um fundo opaco ( $R_0, G_0, B_0, 1$ )
  - Esta cor é a cor destino inicial
- Agora queremos desenhar um polígono translúcido com cor ( $R_1, G_1, B_1, \alpha_1$ )
- Então precisaremos selecionar `GL_SRC_ALPHA` e `GL_ONE_MINUS_SRC_ALPHA` como fatores origem e destino

$$R'_1 = \alpha_1 R_1 + (1 - \alpha_1) R_0, \dots\dots$$

- Esta fórmula funciona quando tanto quando o polígono é opaco ou transparente

# Exemplo 2

---

- Gostaríamos de combinar 3 imagens, com as frações de mistura  $1/3$ ,  $1/3$ ,  $1/3$ :

```
glEnable(GL_BLEND)
```

```
// primeira primitiva: mantém 1/3 da sua cor
```

```
glColor4f(1.0, 1.0, 1.0, 0.333f);    // define valor de alfa ~= 1/3  
glBlendFunc(GL_SRC_ALPHA, GL_ZERO);
```

```
// desenha primeira primitiva
```

```
// segunda primitiva: adiciona 1/3 da sua à cor existente  
glColor4f(1.0,1.0,1.0, 0.333f);  
glBlendFunc(GL_SRC_ALPHA, GL_ONE);    1/3 + 1/3
```

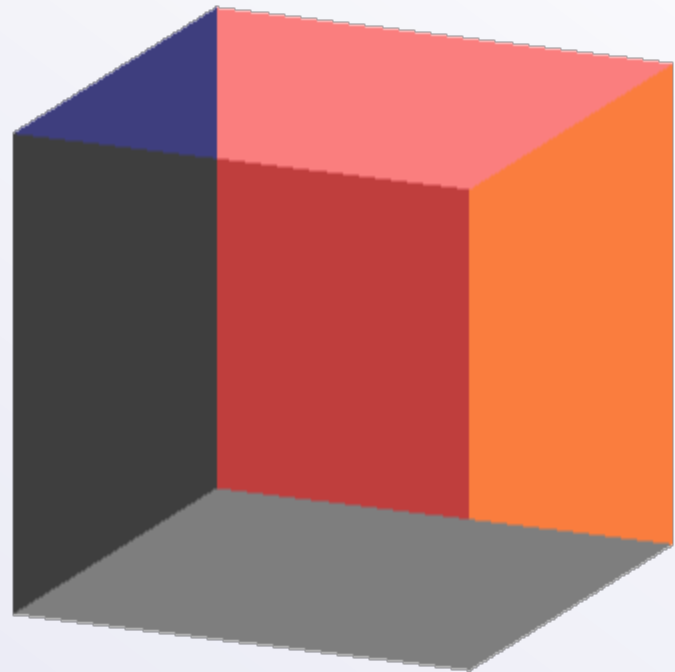
```
// desenha segunda primitiva
```

```
...
```

# Translucidez

---

- A imagem ao lado é uma imagem correta?
  - Provavelmente não...
  - Polígonos são desenhados na ordem que são passados no pipeline
  - As funções de mistura são dependentes da ordem de renderização



# Polígonos opacos e translúcidos

---

- Vamos supor que temos um grupo de polígonos opacos e alguns translúcidos
- Como usar a remoção de superfícies escondidas?
- Ordenar a sequência de desenho:
  - Desenhar todos os polígonos opacos primeiro
  - Desenhar os objetos translúcidos (do mais distante ao mais perto)
- Polígonos translúcidos não deveriam afetar o buffer de profundidade:
  - Renderizar objetos opacos primeiro
  - Renderizar objetos translúcidos com `glDepthMask(GL_FALSE)`, que faz com que o depth buffer não possa ser alterado.

# Demo FirstTransp.c

---



# Neblina

---

- Podemos compor uma cena com uma cor fixa e variar o fator de mistura para que a cor final dependa da profundidade
  - Simular neblina
- Dados a cor origem  $C_s$  e a cor de neblina  $C_f$ , a cor final é dado pela equação:

$$C_s' = f C_s + (1-f) C_f$$

- Onde  $f$  é o fator de atenuação de neblina:
  - Exponencial
  - Gaussiano
  - Linear

# Funções de neblina em OpenGL

---

```
GLfloat fcolor[4] = {.....};
```

```
glEnable(GL_FOG);  
glFogf(GL_FOG_MODE, GL_EXP);  
glFogf(GL_FOG_DENSITY, 0.5);  
glFogv(GL_FOG, fcolor);
```

$$f = e^{-(density \cdot z)} \quad (GL\_EXP)$$

$$f = e^{-(density \cdot z)^2} \quad (GL\_EXP2)$$

$$f = \frac{end - z}{end - start} \quad (GL\_LINEAR)$$

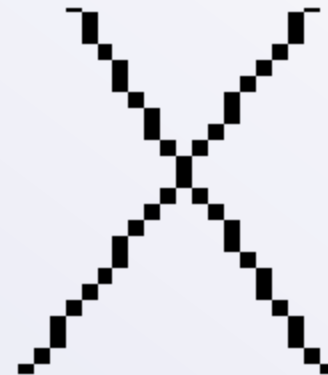
# Demo fog.c

---

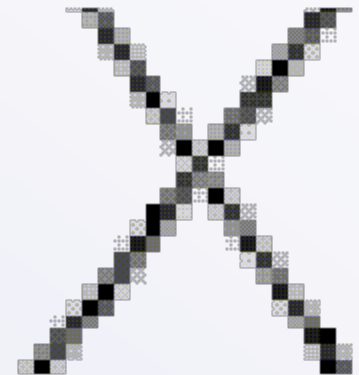
# Serrilhamento

---

- Todos os segmentos de linha, exceto os verticais e horizontais, cobrem parcialmente os pixels
- Algoritmos simples de rasterização produzem efeitos indesejáveis:
  - serrilhamento (“aliasing”);
- Também é problemático na renderização de polígonos



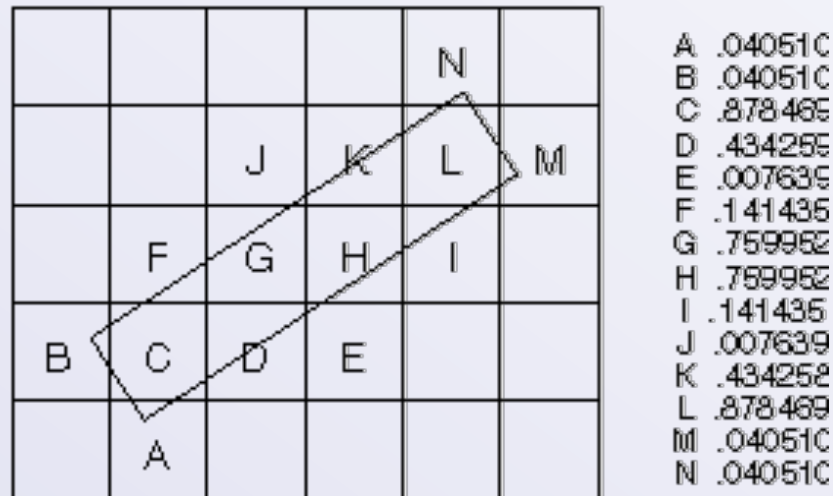
Aliased



Antialiased

# Antiserrilhamento

- Podemos colorir um pixel adicionando somente uma fração da sua cor no frame buffer
- Esta fração depende da porcentagem do pixel coberto pelo fragmento
- Fração também depende da existência de sobreposição ou não



# Antiserrilhamento em OpenGL

---

- Podemos habilitar a técnica de antiserrilhamento separadamente para pontos, linhas e polígonos

```
glEnable(GL_POINT_SMOOTH);  
glEnable(GL_LINE_SMOOTH);  
glEnable(GL_POLYGON_SMOOTH);
```

```
glEnable(GL_BLEND);  
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

# Demo antiserrilhamento

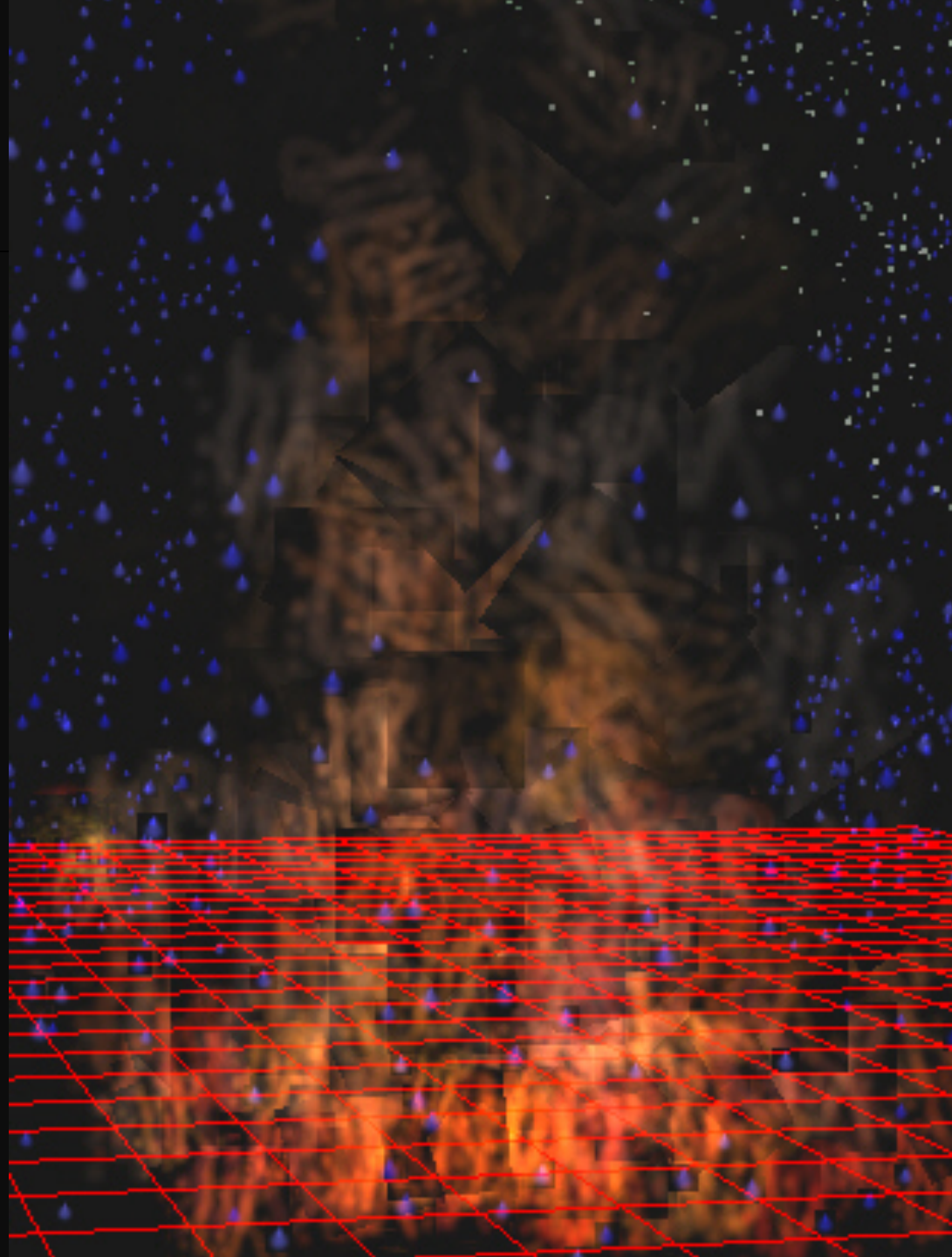
---

# Sistemas de partículas

---

- Um dos mais importantes métodos procedurais
- Usado na modelagem de:
  - Fenômenos naturais
    - Nuvens
    - Terrenos
    - Plantas
  - Multidões
  - Processos físicos reais









1,800,000

COMBO

CAVE

BALL-LOCK

BUNKER

LAUNCH!

MARS

DESERT

ARCTIC

JUNGLE

HELL

RESCUE  
ACTIVE

WORM

MIN

S. BOBERG  
SENGHORE  
LYNNE



My Name

My Friend

Aiming, press and hold 'Ctrl' key to lock the cue for shot

Space: Get Cursor ESC: Get Menu Ft. Wall

0:0



# Leis de Newton

---

- Primeira lei de Newton ou princípio da inércia:  
Um corpo que esteja em movimento ou em repouso, tende a manter seu estado inicial.
- Segunda lei de Newton ou princípio fundamental da mecânica:  
A resultante das forças de agem num corpo é igual ao produto de sua massa pela aceleração adquirida.
- Terceira lei de Newton ou lei de ação e reação:  
Para toda força aplicada, existe outra de mesmo módulo, mesma direção e sentido oposto.

# Partículas Newtonianas

---

- Um sistema de partículas representa um conjunto de partículas
- Cada partícula é ponto ideal de massa
- Seis graus de liberdade
  - Posição  $x = vt$
  - Velocidade
- Cada partícula obedece a segunda lei de Newton

$$f = ma$$

# Equações

---

$$\mathbf{p}_i = (x_i, y_i, z_i)$$

$$\mathbf{v}_i = d\mathbf{p}_i / dt = \mathbf{p}_i' = (dx_i / dt, dy_i / dt, dz_i / dt)$$

$$\mathbf{a}_i = \mathbf{v}_i' = d\mathbf{v}_i / dt$$

$$m \mathbf{v}_i' = \mathbf{f}_i$$

A parte complicada é a definição do vetor força !

# Vetor Força

---

- Partículas independentes
  - Gravidade
  - Forças do vento
  - Cálculo em  $O(n)$
- Partículas interligadas  $O(n)$ 
  - Malhas
  - Sistemas de molas
- Partículas interligadas  $O(n^2)$ 
  - Forças de atração e repulsão

# Algoritmo

---

```
float tempo, dt, estado[6*n], força[3*n];  
  
estado = estado_inicial();  
  
for (tempo=t0;tempo<tempo_final,tempo+=dt)  
{  
    força = calcula_força(estado, tempo);  
    estado = edo(força, estado, tempo, dt);  
    desenha_partículas(estado, tempo);  
}
```



# Forças simples

---

- Considere a força aplicada na partícula  $i$

$$\mathbf{f}_i = \mathbf{f}_i(\mathbf{p}_i, \mathbf{v}_i)$$

- Gravidade  $\mathbf{f}_i = \mathbf{g}$

$$\mathbf{g}_i = (0, -g, 0)$$

- Força do vento

- Fricção

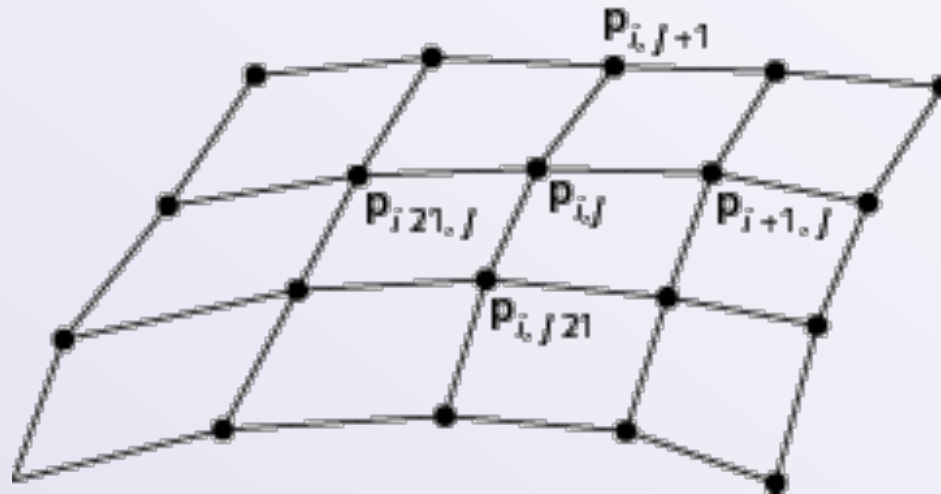


$$\mathbf{p}_i(t_0), \mathbf{v}_i(t_0)$$

# Malhas de partículas

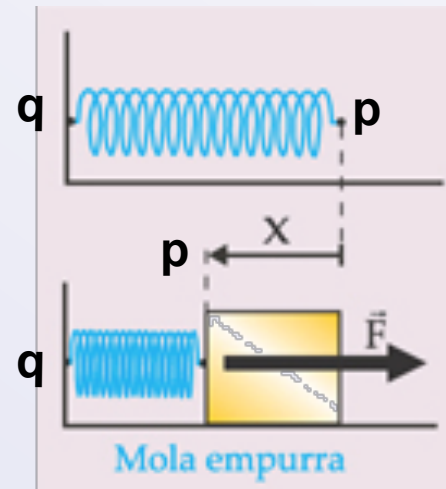
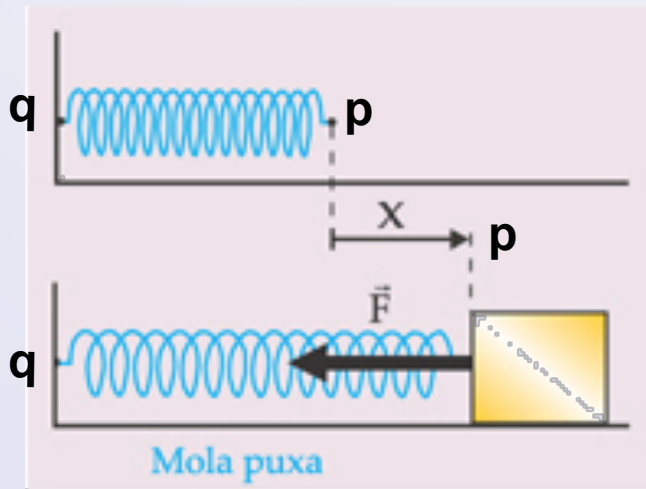
---

- Cada partícula é conectada aos seus vizinhos
- Partículas conectadas através de molas



# Força elástica

- Assume-se que cada partícula possui massa unitária e está conectada por uma mola
- Lei de Hooke: a força é proporcional a distância ( $d = \|\mathbf{p} - \mathbf{q}\|$ ) entre os pontos



# Lei de Hooke

---

- Seja  $s$  a distância entre  $\mathbf{p}$  e  $\mathbf{q}$  quando em equilíbrio:

$$\mathbf{f} = -k_s(|\mathbf{d}| - s) \mathbf{d}/|\mathbf{d}|$$

onde  $k_s$  representa a constante elástica e

$\mathbf{d}/|\mathbf{d}|$  é o vetor unitário apontando de  $\mathbf{p}$  para  $\mathbf{q}$

- Podemos modelar a interação de cada partícula em uma malha através de 4 forças aplicadas à elas.

# Resistência

---

- Um sistema elástico puro oscila eternamente
- Deve-se adicionar um termo de resistência (“damping”)

$$\mathbf{f} = -(k_s(|\mathbf{d}| - r) + k_d \frac{\mathbf{d} \cdot \mathbf{d}}{|\mathbf{d}|}) \frac{\mathbf{d}}{|\mathbf{d}|}$$

onde  $\mathbf{d} = \mathbf{sp} - \mathbf{q}$

- O coeficiente de damping ou resistência depende da velocidade entre as partículas

# Forças de atração e repulsão

---

- Inversamente proporcional a distância  $d$  entre duas partículas

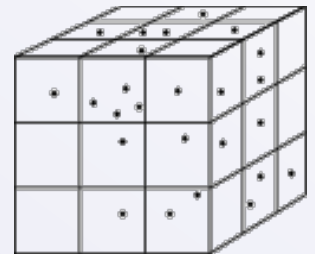
$$\mathbf{f} = -k_r \mathbf{d} / |\mathbf{d}|^3$$

- O caso genérico requer cálculo da ordem  $O(n^2)$
- Na maioria dos problemas, nem todas as partículas irão contribuir para a força de repulsão ou atração em uma determinada partícula
- A fim de simplificar o problema, podemos dividir o espaço em células;

# Caixas

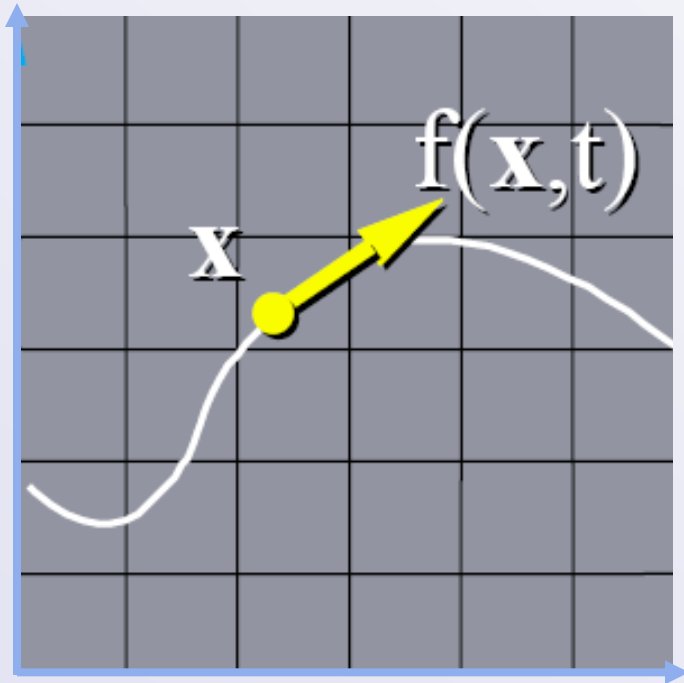
---

- Técnica de subdivisão espacial
- Dividir o espaço em caixas
- Uma partícula somente dependerá de forças de partículas da sua própria caixa ou da sua caixa adjacente
- A cada passo de tempo, precisamos determinar a caixa à qual a partícula pertence



# Uma Equação Diferencial Canônica

---



$$\dot{\mathbf{x}} = f(\mathbf{x}, t)$$

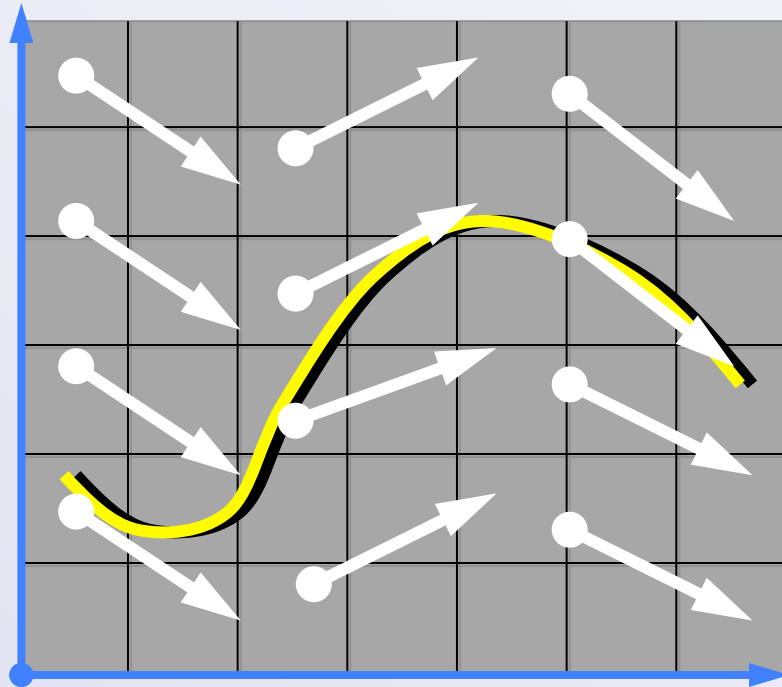
$\mathbf{x}(t)$  : um ponto em movimento

$f(\mathbf{x}, t)$  : velocidade de  $\mathbf{x}$



# Campo vetorial

---



**A equação diferencial**

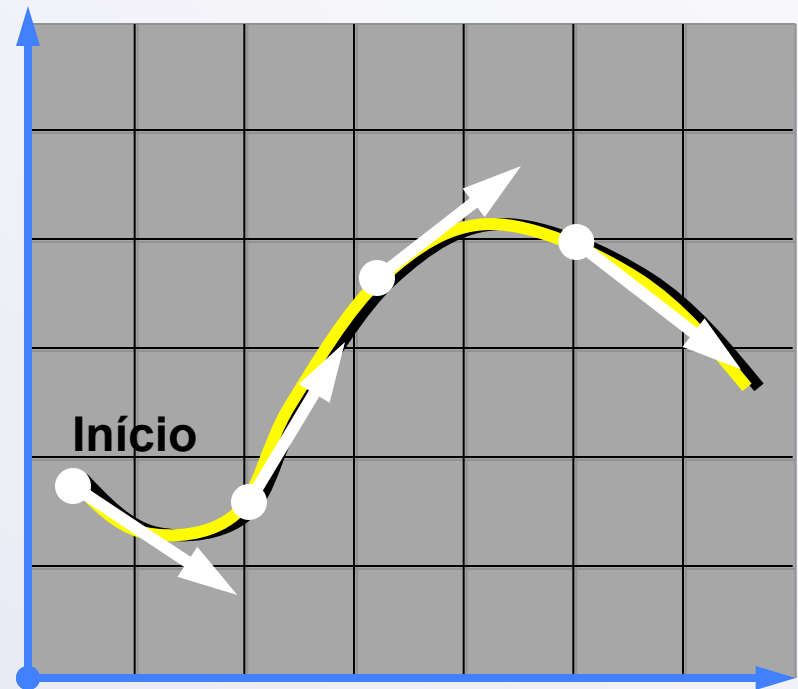
$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$$

**define um campo  
vetorial em  $\mathbf{x}$ .**

# Curvas integrais

---

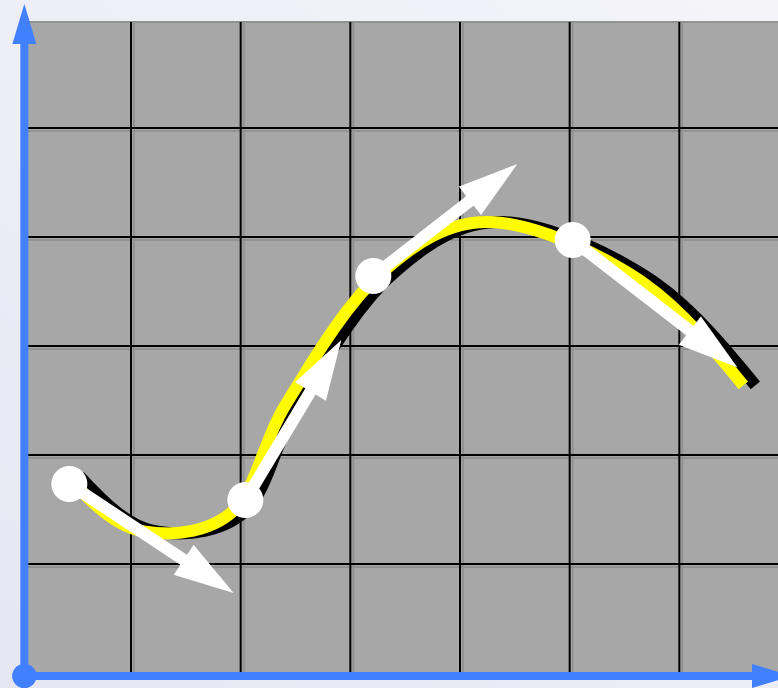
- Escolha qualquer ponto de início e siga a direção dos vetores.



# Problema de valor inicial

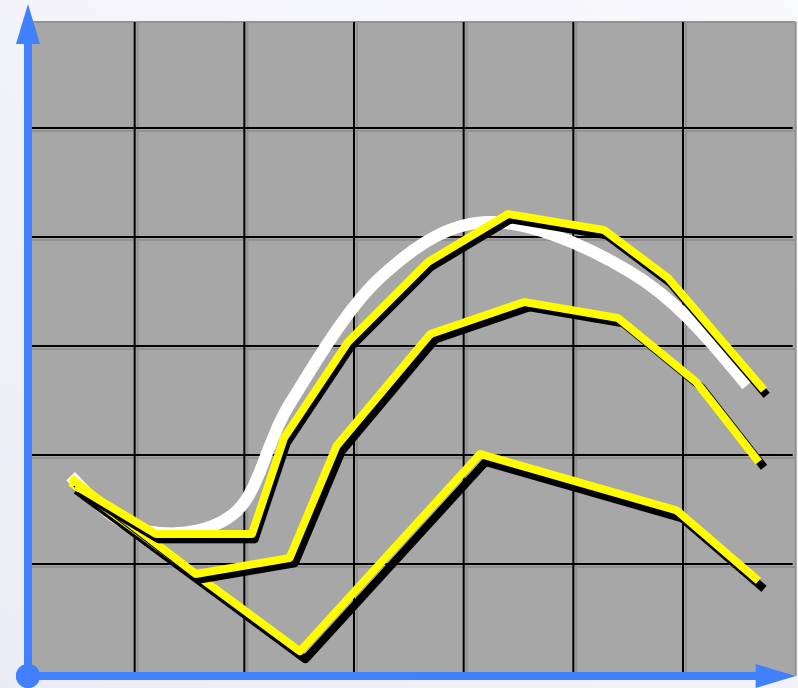
---

- Dado um ponto inicial, seguir a curva integral.



# Método de Euler

- Método numérico mais simples.
- Utiliza passos discretos de tempo.
- Quanto maiores forem os passos, maiores serão os erros.
- Magnitude do erro depende da curvatura da solução.

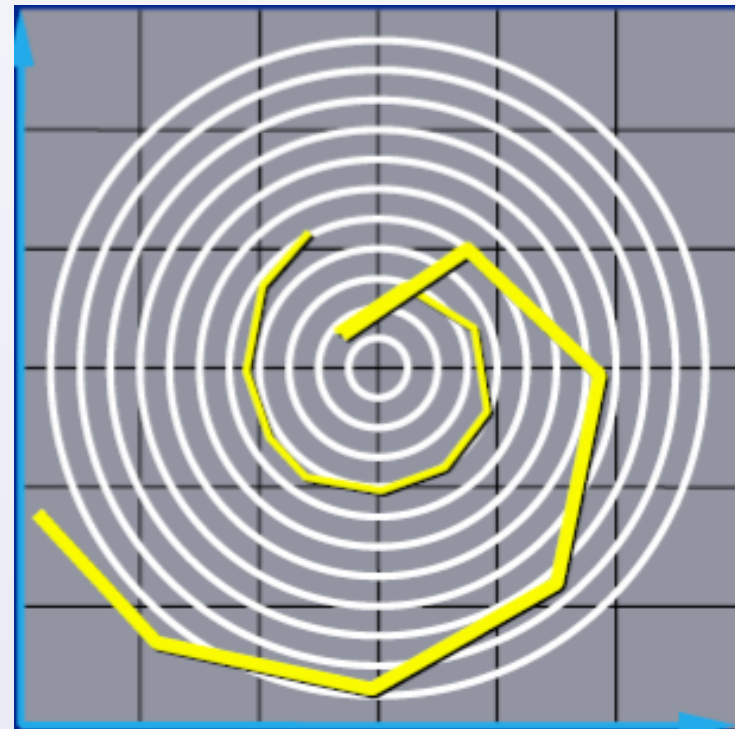


$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \mathbf{f}(\mathbf{x}, t)$$

# Problema 1: Inacurácia

---

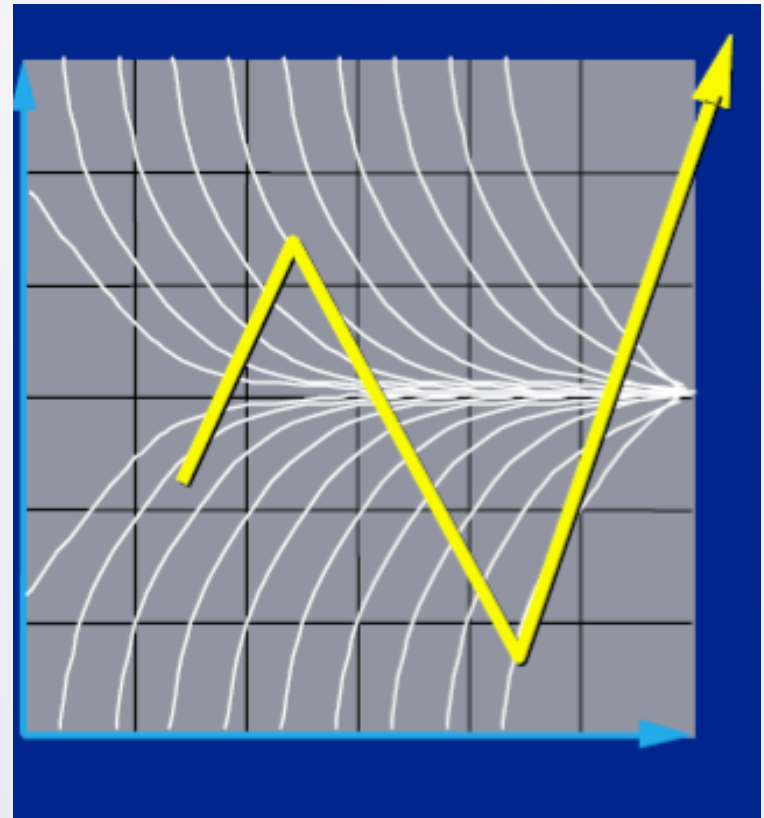
- O erro faz com que  $\mathbf{x}(t)$  saia de um círculo e entre em uma espiral de sua escolha.



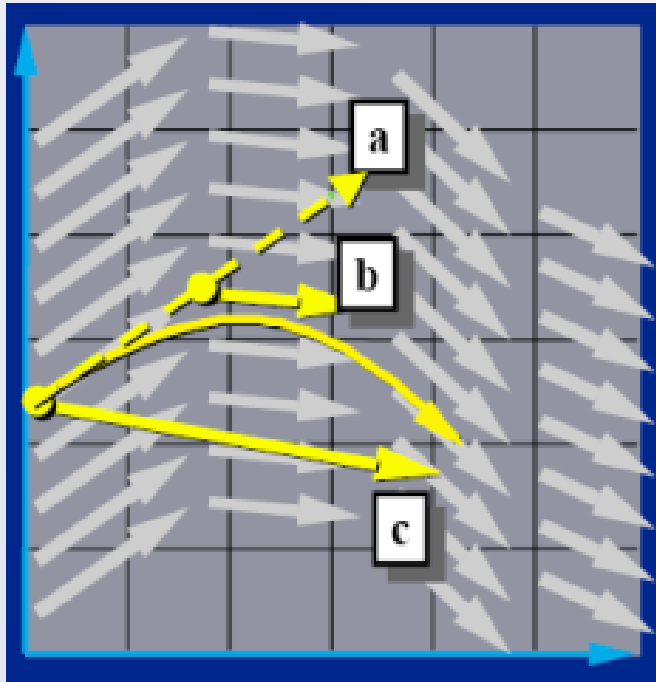
# Problema 2: Instabilidade

---

- O método de Euler pode se tornar instável dependendo do campo vetorial.



# Método do Ponto Médio



- a. Calcule um passo de Euler

$$\Delta \mathbf{x} = \Delta t \mathbf{f}(\mathbf{x}, t)$$

- b. Avalie  $\mathbf{f}$  no ponto médio

$$\mathbf{f}_{\text{med}} = \mathbf{f}\left(\frac{\mathbf{x} + \Delta \mathbf{x}}{2}, \frac{t + \Delta t}{2}\right)$$

- c. Dê um passo usando o valor do ponto médio

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \Delta t \mathbf{f}_{\text{med}}$$

# Outros Métodos

---

- O método de Euler é de 1ª. ordem
- O método do Ponto Médio é de 2ª. ordem
  - Runge-Kutta 2
- Outros métodos:
  - Runge-Kutta 3ª. e 4ª. ordem
  - Passos adaptativos
- Dicas básicas:
  - Não use o método de Euler (embora você acabará usando);
  - Use passo de tempo adaptativo;



# Runge-Kutta 4<sup>a</sup>. Ordem (RK4)

---

$$\mathbf{F}_1 = \mathbf{f}(\mathbf{x}_i, t)$$

$$h = \Delta t / 2$$

$$\mathbf{F}_2 = \mathbf{f}(\mathbf{x}_i + h\mathbf{F}_1, t + h)$$

$$\mathbf{F}_3 = \mathbf{f}(\mathbf{x}_i + h\mathbf{F}_2, t + h)$$

$$\mathbf{F}_4 = \mathbf{f}(\mathbf{x}_i + \Delta t\mathbf{F}_3, t + \Delta t)$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \frac{\Delta t}{6} (\mathbf{F}_1 + 2\mathbf{F}_2 + 2\mathbf{F}_3 + \mathbf{F}_4)$$

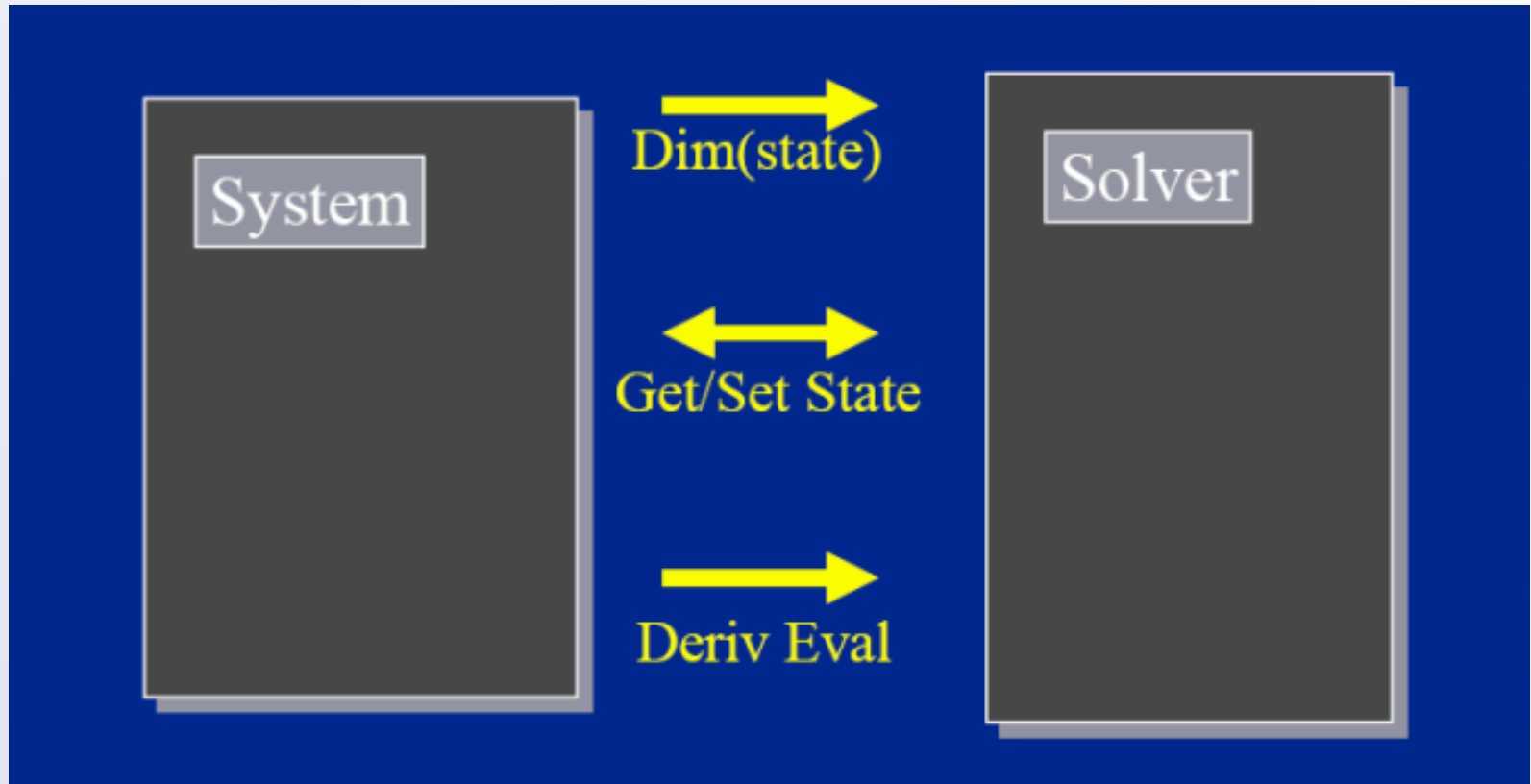
# Implementação modular

---

- Operações genéricas:
  - Get Dim( $\mathbf{x}$ )
  - Get/Set  $\mathbf{x}$  e  $t$
  - Avaliar derivada na posição  $(\mathbf{x}, t)$
- Escrever resolução de EDOs em termos destas funções:
  - Código reutilizável
  - Simplificação do modelo de implementação

# Interface do Resolvedor

---



# Partícula Newtoniana

---

- Equação diferencial

$$\mathbf{f} = m \mathbf{a}$$

- Forças podem depender de:
  - Posição
  - Velocidade
  - Tempo

$$\ddot{\mathbf{x}} = \frac{\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t)}{m}$$

# Equações de segunda ordem

---

$$\ddot{\mathbf{x}} = \frac{\mathbf{f}(\mathbf{x}, \dot{\mathbf{x}}, t)}{m}$$

$$\begin{cases} \dot{\mathbf{x}} = \mathbf{v} \\ \dot{\mathbf{v}} = \mathbf{f} / m \end{cases}$$

- Não está na forma canônica porque contém derivadas de 2ª ordem
- Adicionar uma nova variável  $\mathbf{v}$ , para conseguir um par de equações de 1ª. ordem

# Espaço de Fase

---

$$\begin{bmatrix} \mathbf{x} \\ \mathbf{v} \end{bmatrix}$$

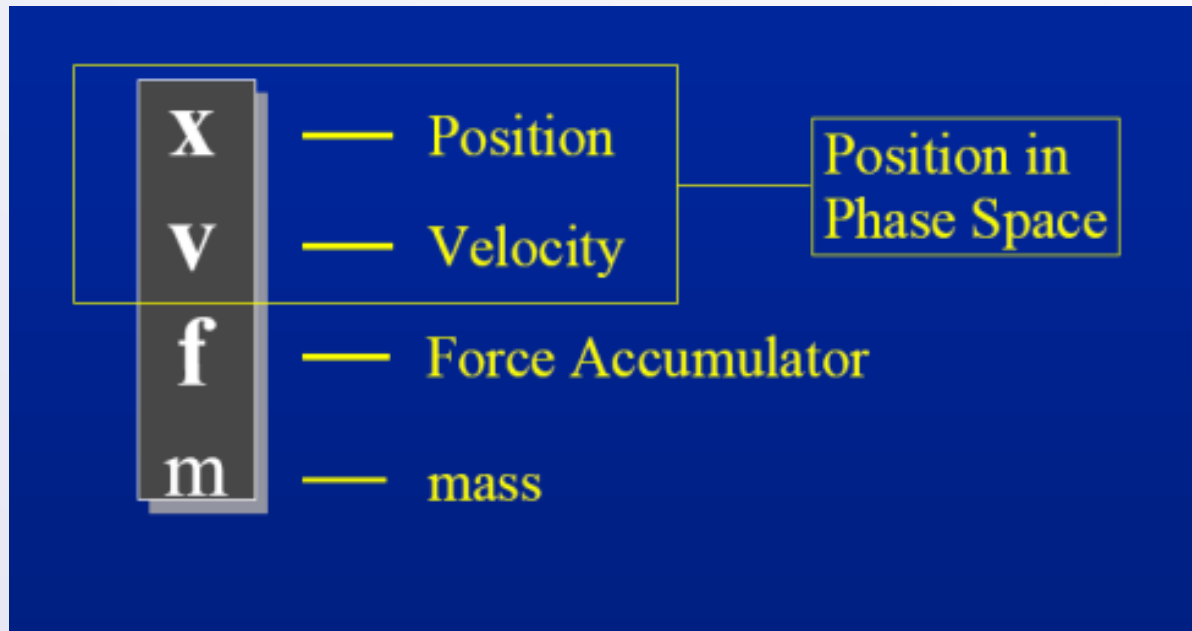
$$\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{v}} \end{bmatrix}$$

$$\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{v}} \end{bmatrix} = \begin{bmatrix} \mathbf{v} \\ \mathbf{f} / m \end{bmatrix}$$

- Concatenar  $\mathbf{x}$  e  $\mathbf{v}$  para compor um vetor de 6 elementos: Posição em Espaço de Fase.
- Velocidade em Espaço de Fase (vetor-6)
- Equação diferencial de 1ª. ordem

# Estrutura da Partícula

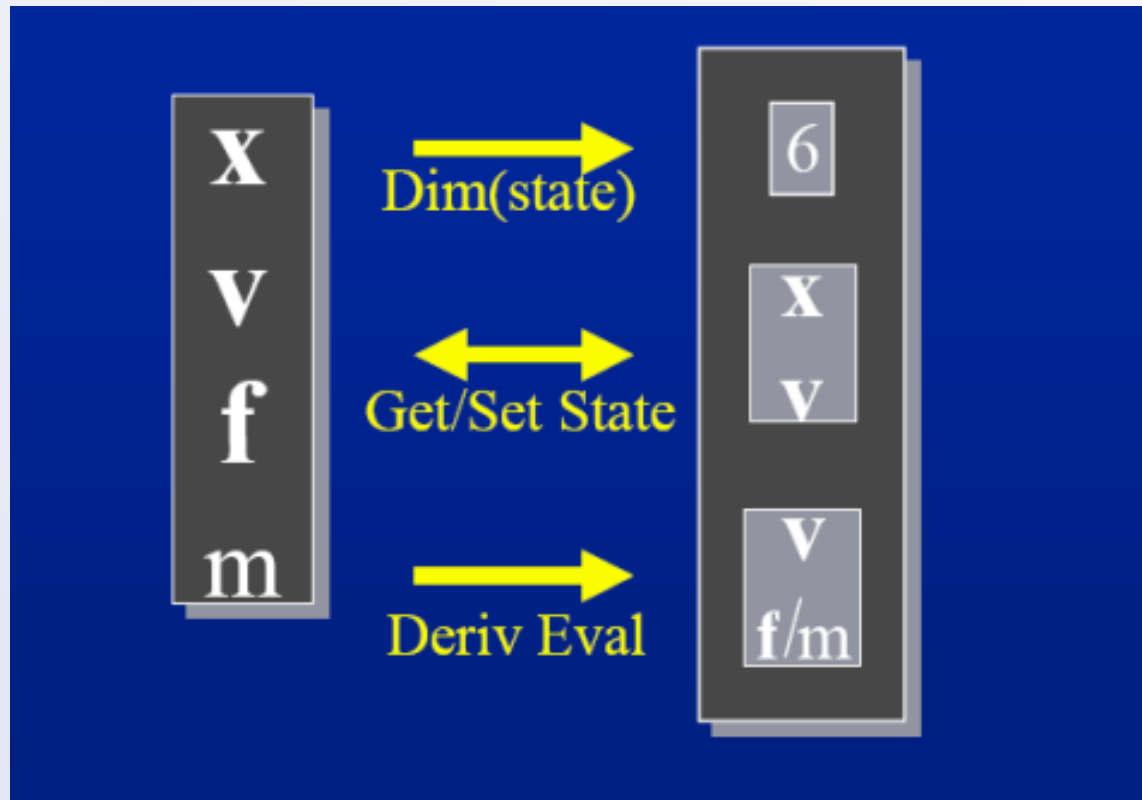
---



```
typedef struct{  
    float m;           /* mass */  
    float *x;          /* position vector */  
    float *v;          /* velocity vector */  
    float *f;          /* force accumulator */  
} *Particle;
```

# Interface com o Solver

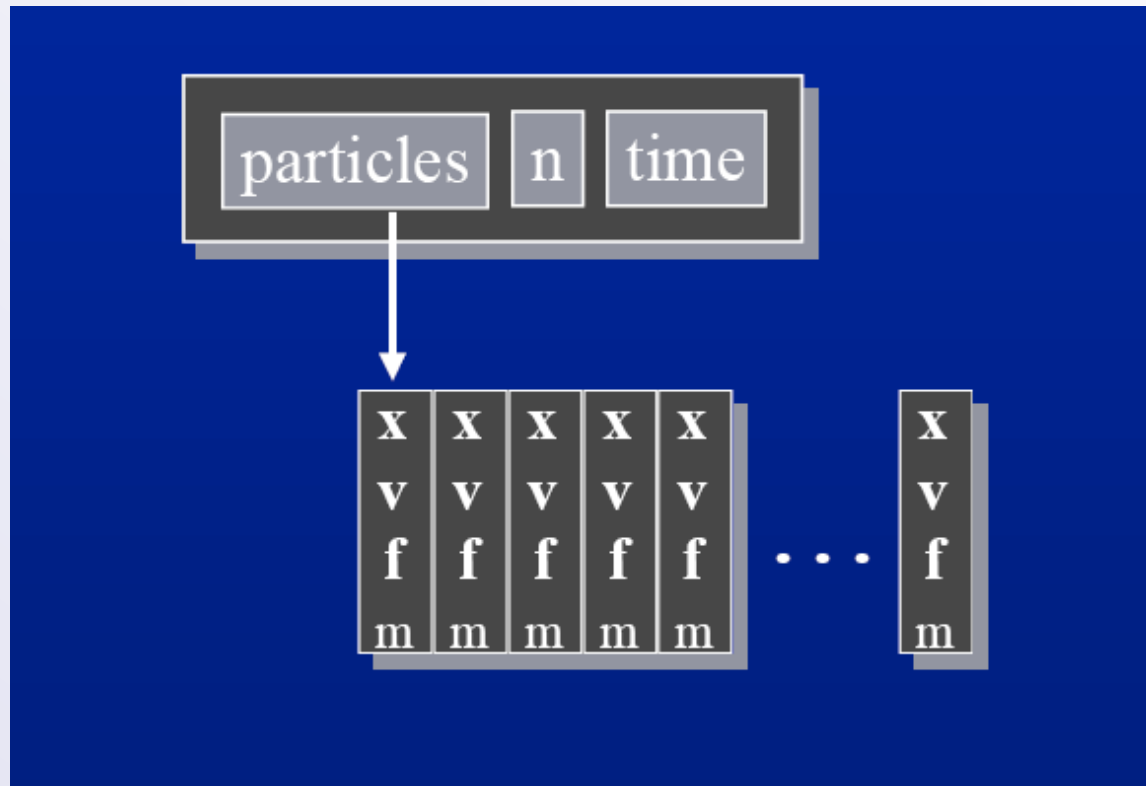
---





# Sistema de Partículas

---



```
typedef struct{  
    Particle *p;    /* array of pointers to particles */  
    int n;          /* number of particles */  
    float t;        /* simulation clock */  
} *ParticleSystem;
```

# Loop de Derivação

---

- Limpa forças
  - Para todas as partículas, zera acumuladores de força;
- Calcula forças
  - Soma todas as forças para cada acumulador
- Coleta resultados
  - Para todas as partículas, copia  $\mathbf{v}$  e  $\mathbf{f}/m$  no array de destino

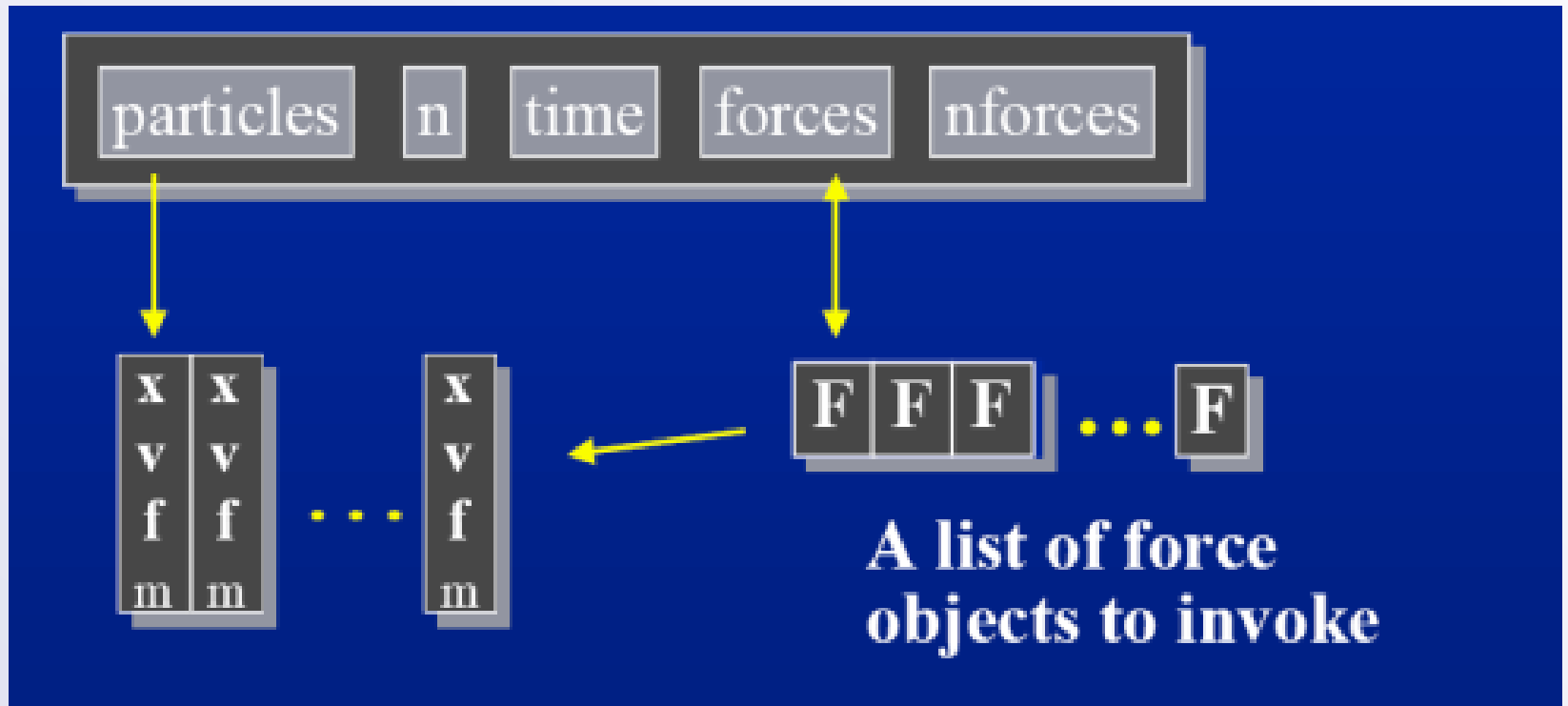
# Estruturas de Força

---

- Diferentemente das partículas, forças são heterogêneas
- Objetos de força:
  - caixas-pretas
  - apontam para as partículas que elas influenciam
  - adicionar suas próprias forças
- Cálculo global das forças
  - Loop, invocando os objetos de força

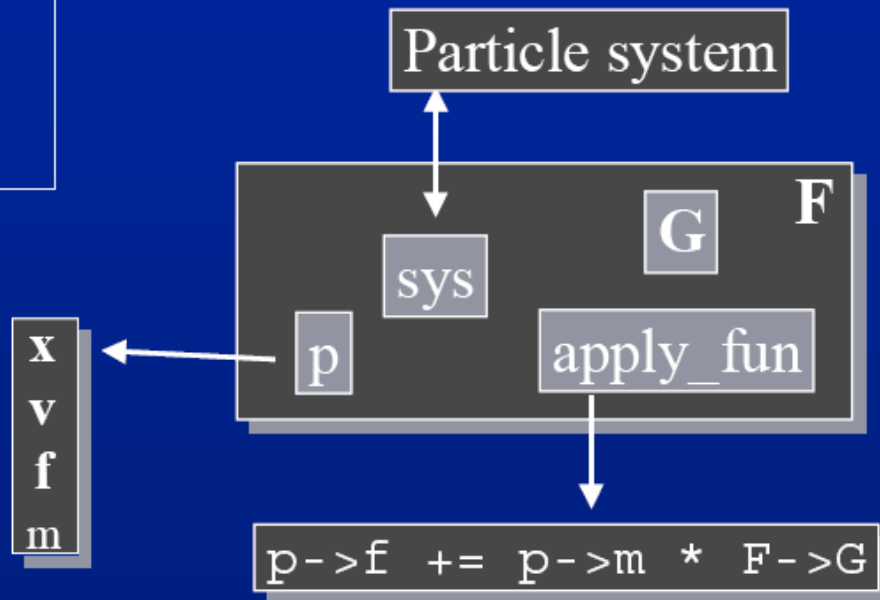
# Sistemas de Partículas, com Forças

---

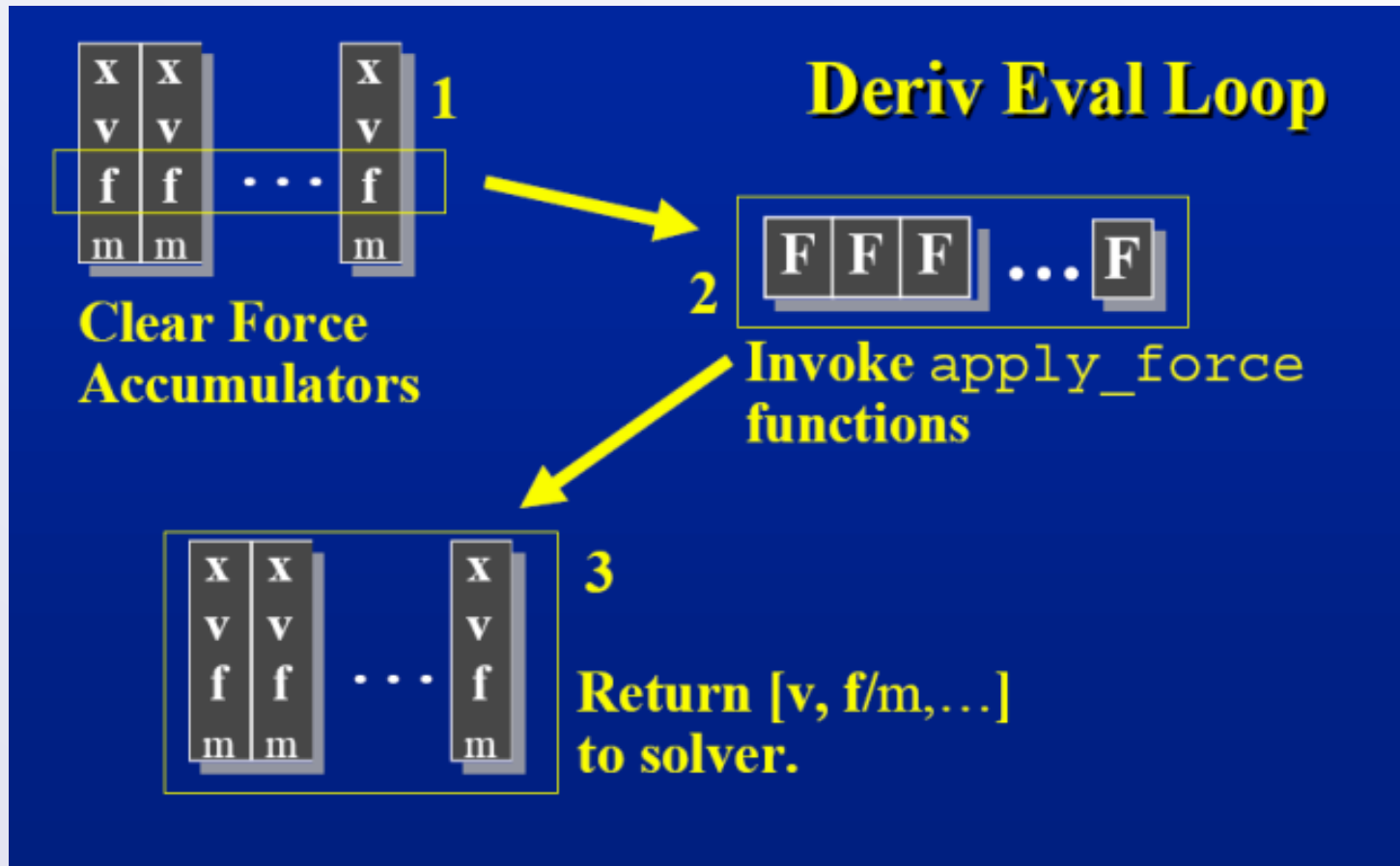


# Força da Gravidade

*Force Law:*  
 $\mathbf{f}_{\text{grav}} = m\mathbf{G}$



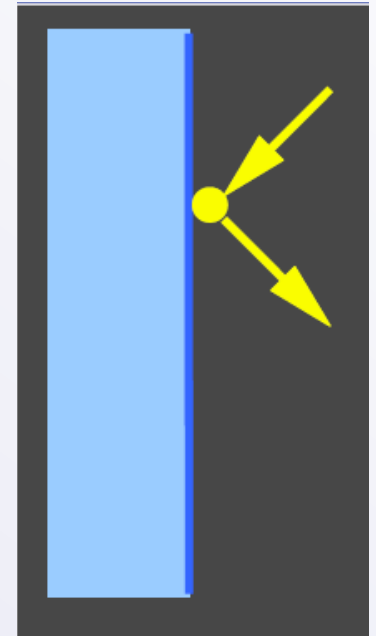
# Esquema Global



# Colisões

---

- O problema geral de contato e colisão é difícil;
- Dois aspectos no tratamento de colisões:
  - Detecção
  - Reação
- Caso após um passo da resolução da EDO, uma colisão for detectada, devemos voltar ao “passado”, resolvendo para o momento da colisão.



# Detecção de Colisão

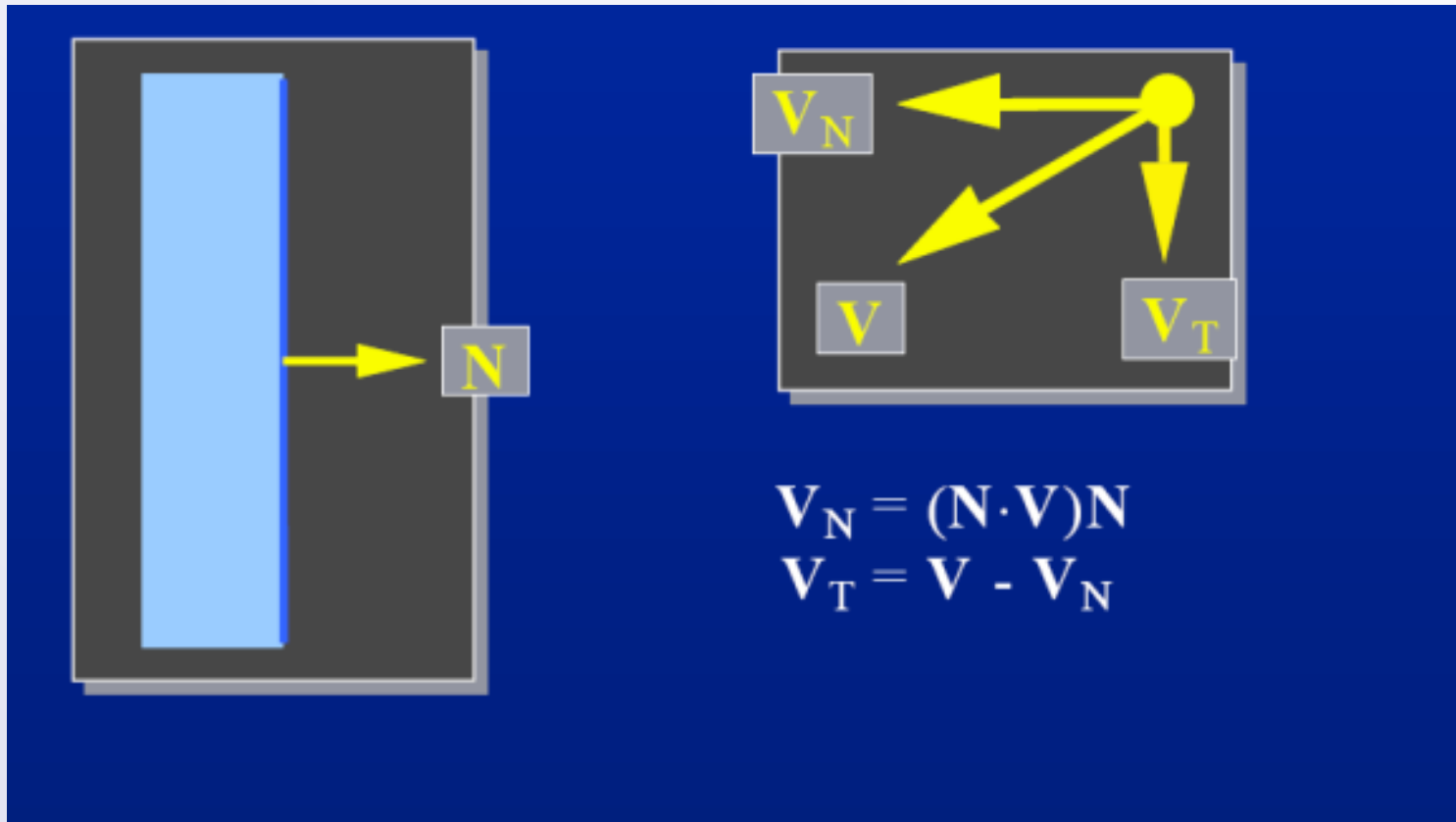
---

- Para descrever uma colisão, precisamos decompor os vetores de velocidade e posição nos seus componentes:
  - Normais
  - Tangenciais
- Seja o vetor de posição  $\mathbf{x}$ :
  - Componente normal  $\mathbf{x}_n = (\mathbf{N} \cdot \mathbf{x})\mathbf{x}$
  - Componente tangencial  $\mathbf{x}_t = \mathbf{x} - \mathbf{x}_n$



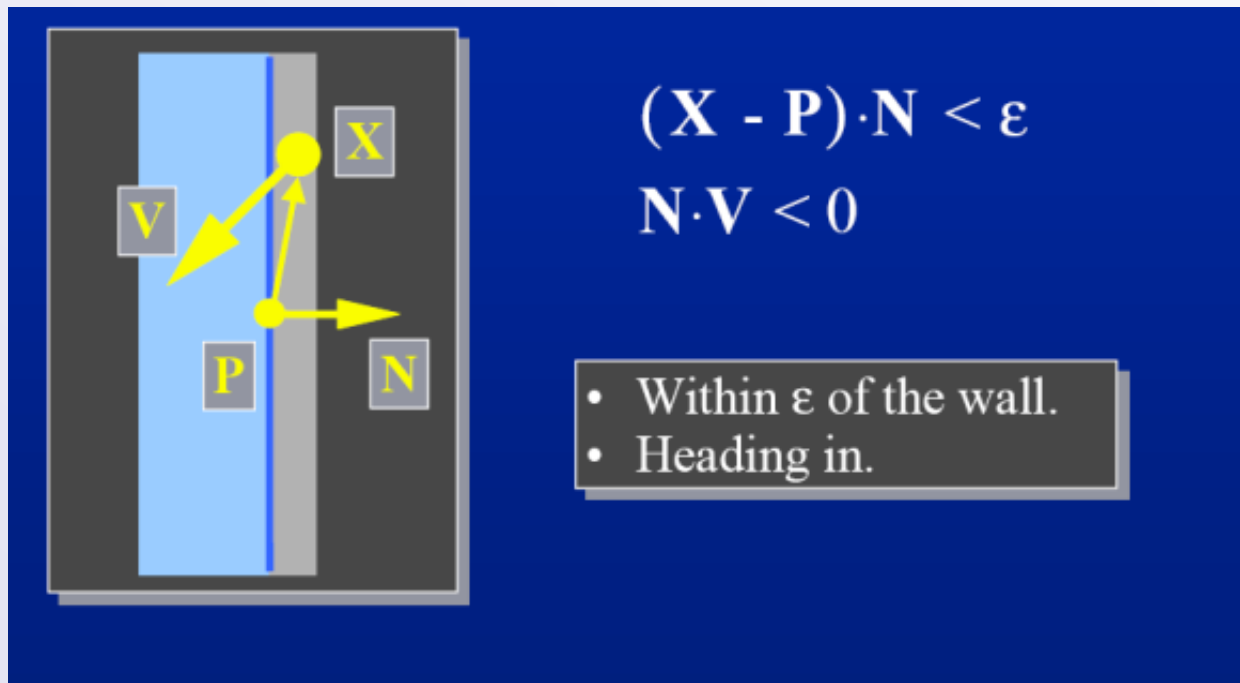
# Componentes Normais e Tangenciais

---



# Detecção de Colisão

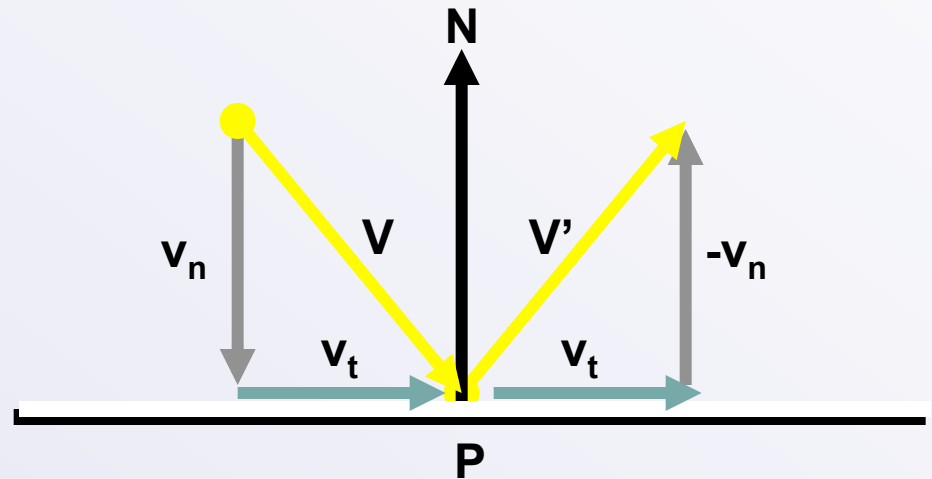
- Seja  $P$  um ponto no plano,  $N$  a normal, então  $(X-P) \cdot N =$ 
  - $=0$ :  $X$  está em contato com o plano
  - $<0$ :  $X$  colide com o plano
  - $>0$ :  $X$  não colide com o plano



# Colisão Elástica

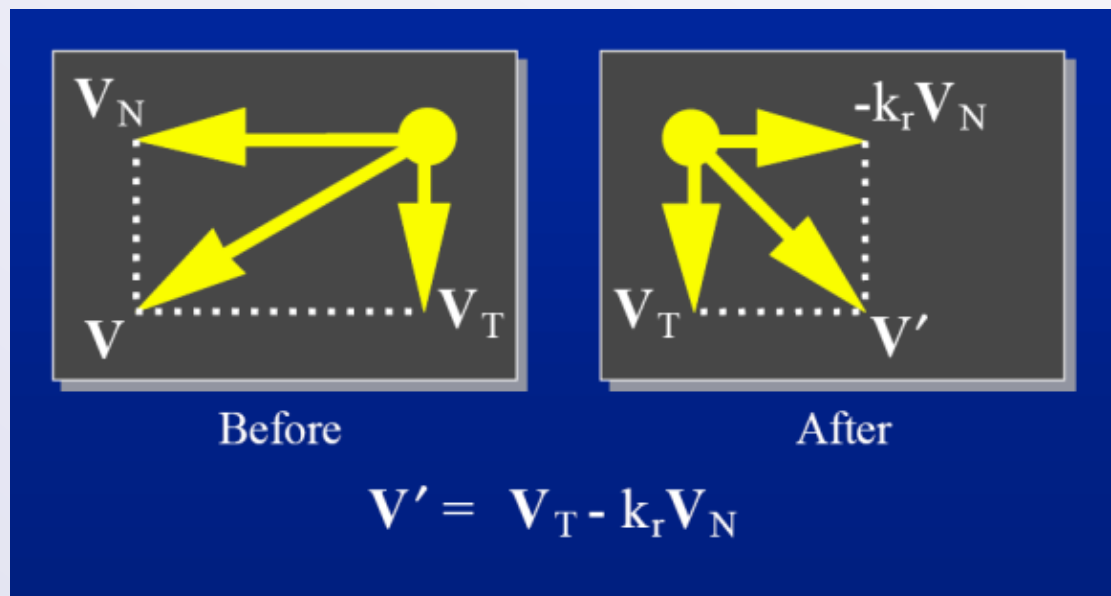
---

- O componente normal  $v_n$  da velocidade da partícula é simplesmente negado;



# Colisão Inelástica

- Em uma colisão inelástica, o componente  $v_n$  é multiplicado pelo fator  $-k_r$ , chamado de coeficiente de restituição;



# Tarefa de casa

---

- Modifique o programa "particula.c" de forma que a bola fique contida dentro de um cubo (invisível).
- Trate a colisão da bola com as 6 paredes do cubo. A colisão deverá ser inelástica, ou seja a bola perde energia mecânica.
- A bola deverá ter posição e velocidade inicial aleatória a cada rodada.