



Visualização Volumétrica

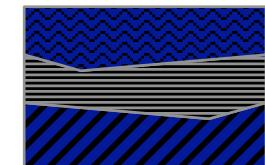
Waldemar Celes

DI/PUC-Rio

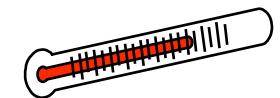


Tipos de Dados

- Enumeráveis (material, litologia, ...)



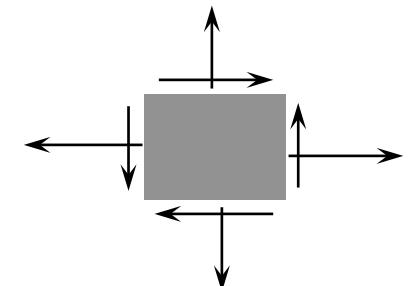
- Escalares (temperatura, pressão, ...)



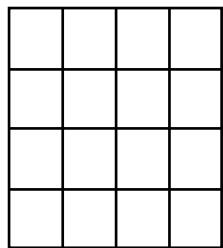
- Vetoriais (velocidade, aceleração, ...)



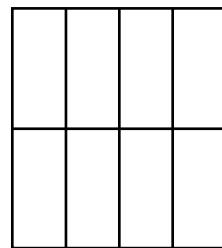
- Tensoriais (tensão, deformação, ...)



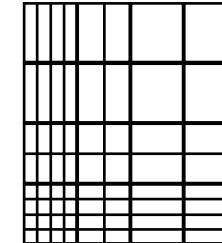
Estrutura dos Dados



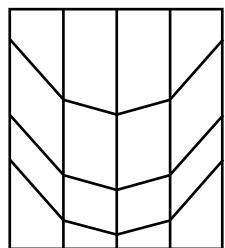
Grade Cartesiana
(i, j, k)



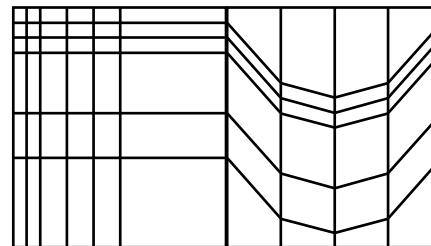
Grade Regular
($i*dx, j*dy, k*dz$)



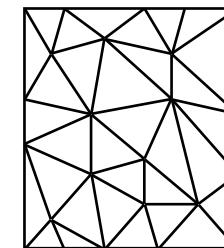
Grade Retilínea
($x[i], y[j], z[k]$)



Grade Estruturada
($x[i,j,k], y[i,j,k], z[i,j,k]$)



Grade Estruturada
por blocos

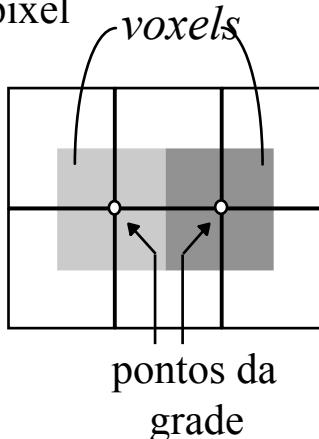


Grade Não Estruturada
 $\{(x[i], y[i], z[i]), e = (v_1, v_2, v_3)\}$

Interpolação

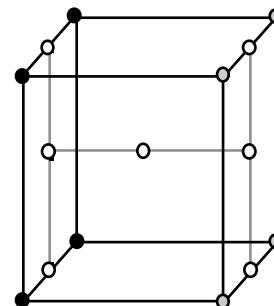
■ Matriz de voxels

- análogo 3D do pixel
- (i, j, k)

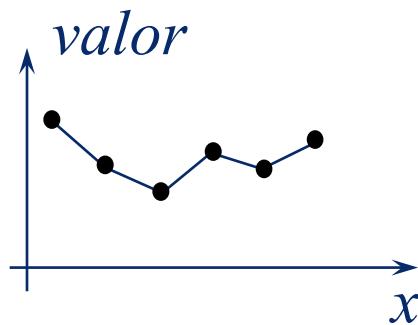
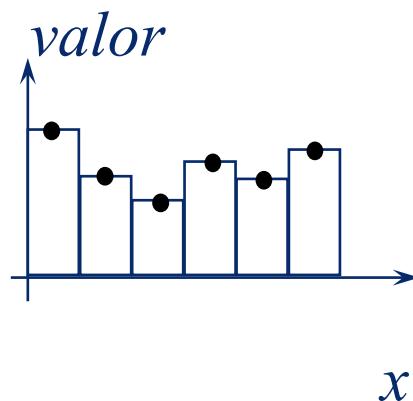


— Matriz de células

- interpolação trilinear
- imagens mais suaves



- pontos da grade
- valores interpolados

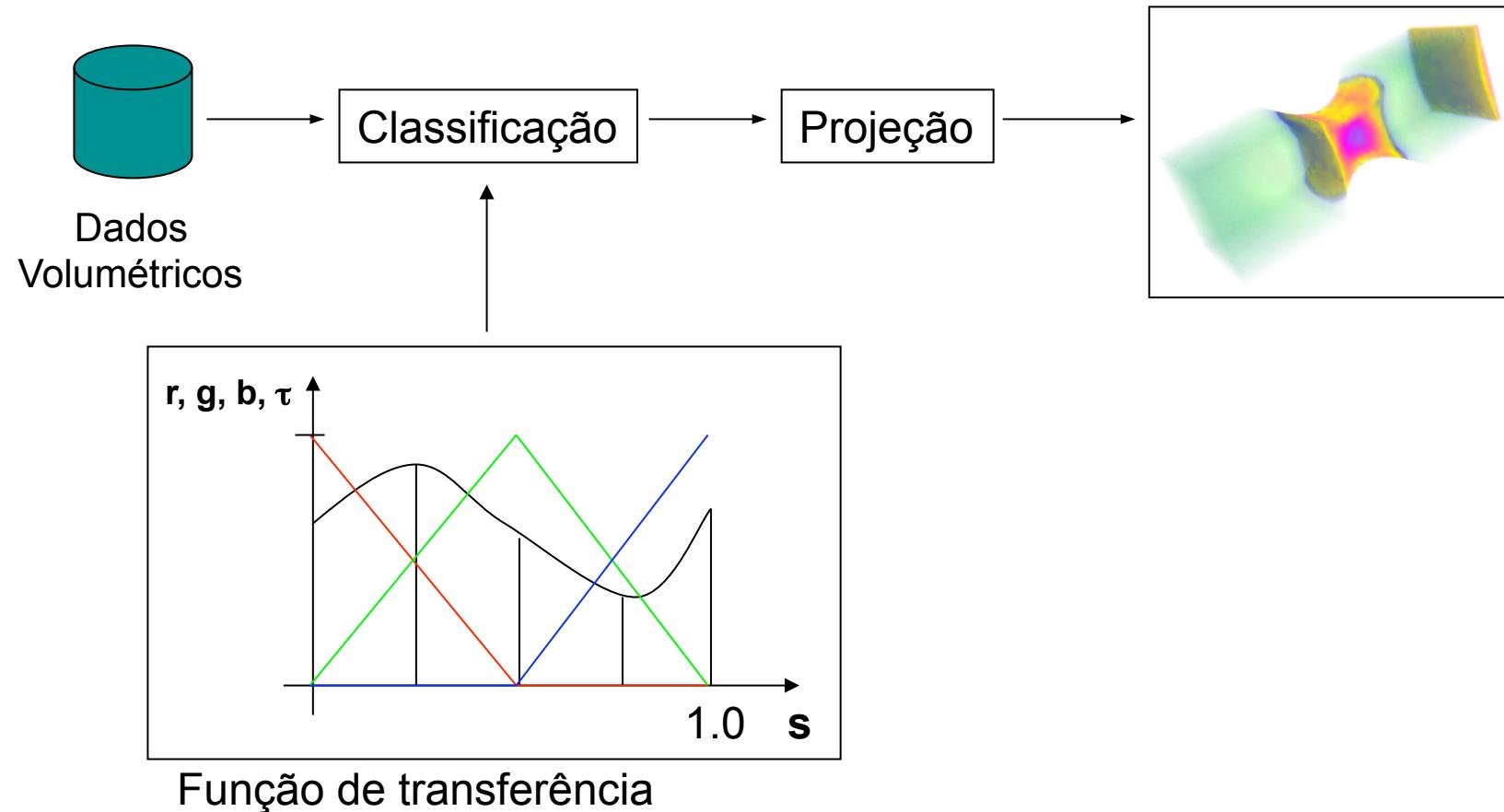


Métodos de Visualização

- **Indiretos: por extração de superfícies implícitas**
 - + representação por polígonos [Zbuffer]
 - + dados menores
 - ⇒ precisa ser refeito quando muda a classificação
 - ⇒ dificuldade de modelar objetos amorfos

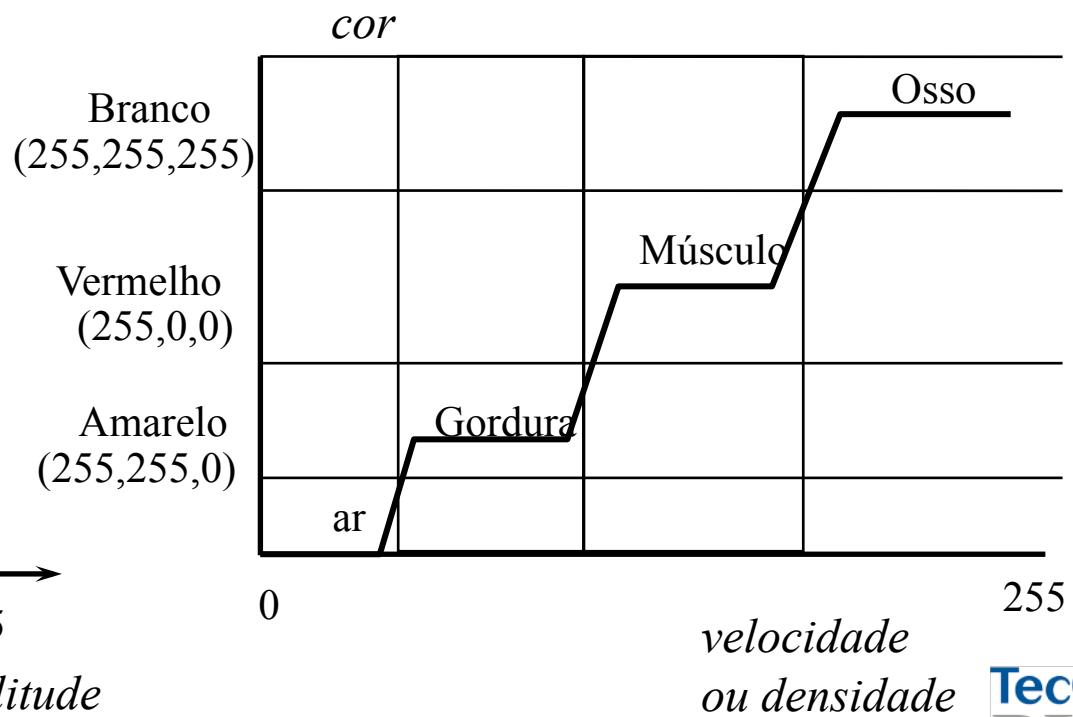
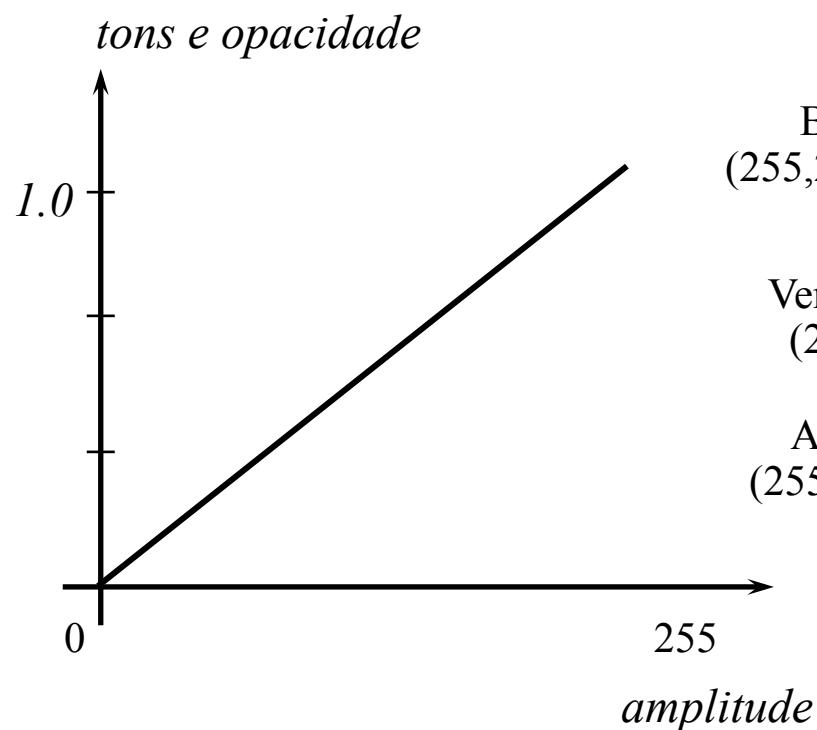
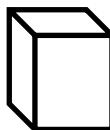
- **Diretos: por modelos de iluminação volumétrica**
 - + geração de imagens diretamente a partir dos dados volumétricos
 - + visualização de múltiplas características, inclusive de dados amorfos
 - ⇒ grande volume de dados

Renderização direta de volumes

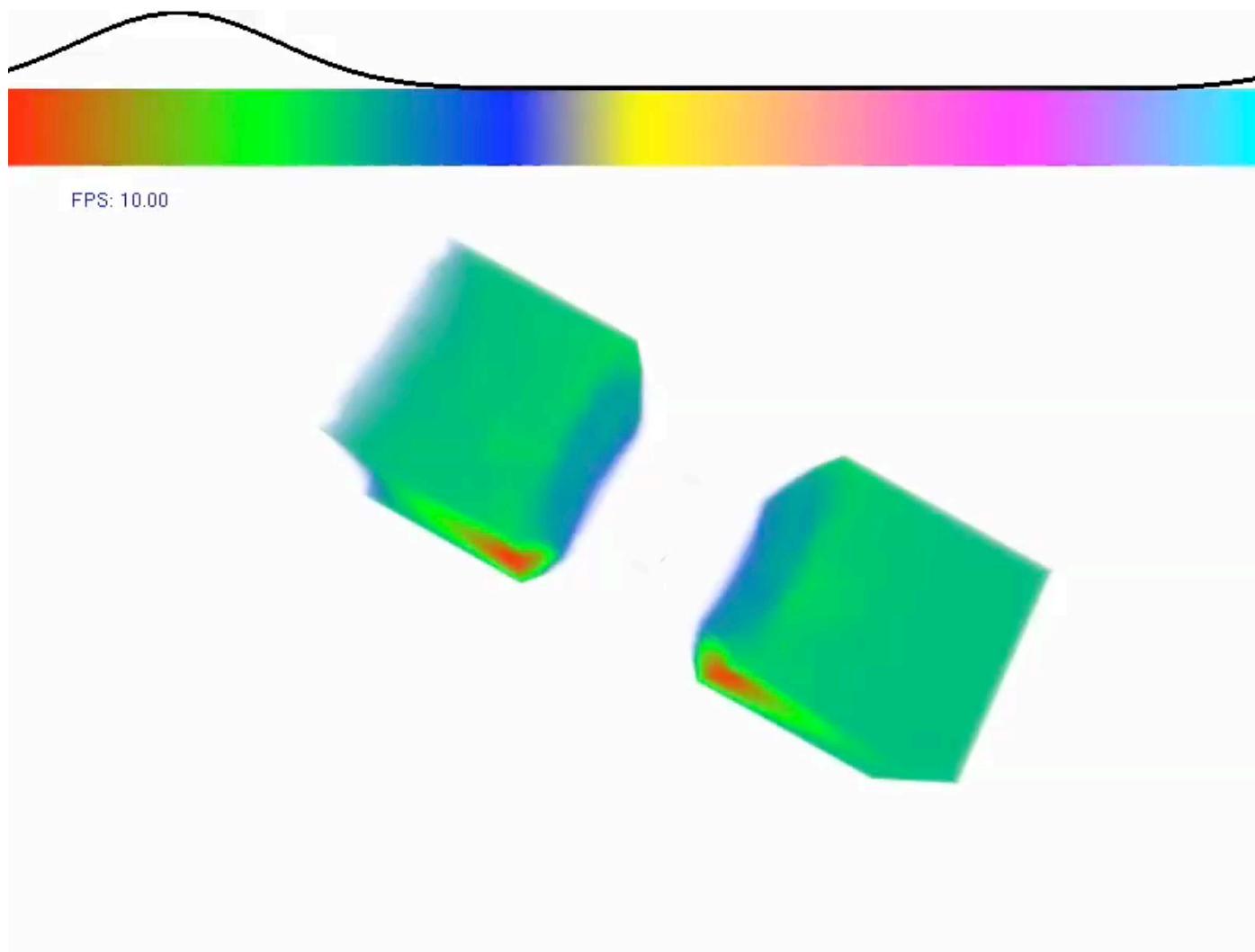


Classificação do Voxel

Voxel

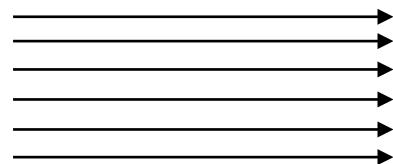


Visualização volumétrica

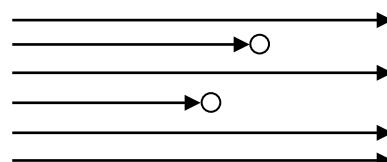


Modelos Ópticos

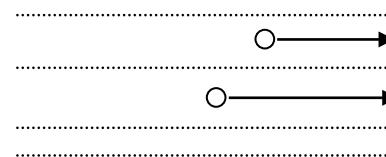
- Interação raio de luz com partículas



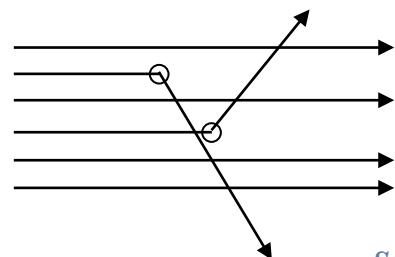
transmission



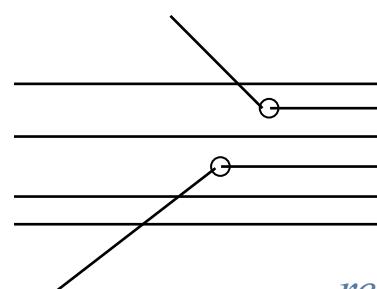
absorption



emission



scattering



redirection

Modelos Ópticos

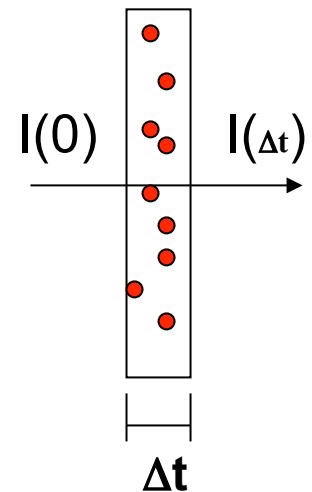
□ Modelo de absorção

⇒ Fatia finita da região

- ✧ Fina o suficiente para não haver sobreposição
- ✧ $\text{Vol} = E \cdot \Delta t$
 - E = área frontal da fatia
 - Δt = espessura da fatia
- ✧ ρ = densidade de partículas
 - Número de partículas por unidade de volume
- ✧ A = área frontal de cada partícula

⇒ Cálculo de absorção

- ✧ Número total de partículas: $N = \rho E \Delta t$
- ✧ Área frontal obstruída pelas partículas: $N A = \rho A E \Delta t$
- ✧ Percentagem de luz absorvida: $\rho A E \Delta t / E = \rho A \Delta t$



Modelos Ópticos

□ Modelo de absorção

⇒ Diferenciando

$$\Delta I = -\rho A \Delta t I$$

$$\frac{\partial I}{\partial t} = -\rho A I(t) = -\tau(s(t)) I(t)$$

onde: $\tau(s(t))$ = coeficiente de extinção

⇒ Integrando

$$I(D) = I(0) e^{-\int_0^D \tau(t) dt}$$

$$T(D) = e^{-\int_0^D \tau(t) dt} \longrightarrow \text{transparência do meio}$$

Opacidade:

$$\alpha = 1 - T(D)$$

Modelos Ópticos

□ Mapeamento de τ para função escalar

⇒ Função de transferência

✧ Propriedade escalar

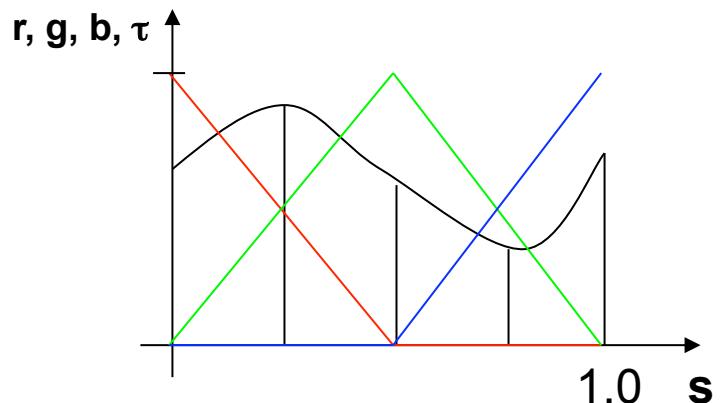
$$s : (x, y, z) \mapsto F(x, y, z)$$

✧ Função de transferência 1D

$$FT : s \mapsto (r(s), g(s), b(s), \tau(s))$$

✧ Função de transferência 2D

$$FT : (s, \vec{\nabla} s) \mapsto (r(s, \vec{\nabla} s), g(s, \vec{\nabla} s), b(s, \vec{\nabla} s), \tau(s, \vec{\nabla} s))$$



Modelos Ópticos

□ Modelo de emissão

- ⇒ Partículas transparentes, mas brilhantes
- ⇒ $C = \text{emissão de luz por unidade de área}$
- ⇒ Fluxo de emissão
 - ✧ $C N A = C\rho A E \Delta s$
 - ✧ Variação do fluxo: $C\rho A \Delta s$

⇒ Derivando e integrando

$$\frac{\partial I}{\partial s} = C(s)\rho(s)A = C(s)\tau(s)$$

$$I(s) = I_0 + \int_0^s C(t)\tau(t)dt$$

Modelos Ópticos

□ Absorção + emissão

- Absorção

$$\frac{dI}{dt} = -\tau(t)I(t), \quad \tau \in [0, \infty]$$

- Emissão

$$\frac{dI}{dt} = c(t)\tau(t), \quad c \in [0, \infty]$$

- Absorção + Emissão

$$\frac{dI}{dt} = c(t)\tau(t) - \tau(t)I(t), \quad \tau, c \in [0, \infty]$$

Modelos Ópticos

□ Absorção + emissão

$$I(D) = I_0 e^{-\int_0^D \tau(s(\vec{x}(t'))) dt'} + \int_0^D c(s(\vec{x}(t))) \tau(s(\vec{x}(t))) e^{-\int_t^D \tau(s(\vec{x}(t'))) dt'} dt$$

$$\alpha = 1 - e^{-\int_0^l \tau(\vec{x}(t')) dt'}$$

$c, \tau = cte :$

$$I(D) = I_0(1 - \alpha) + c \int_D^0 -\tau e^{-\tau t} dt = I_0(1 - \alpha) + c\alpha$$

$C = c\alpha = \text{opacity-weighted color}$ [Blinn1994, Wittenbrink1998]

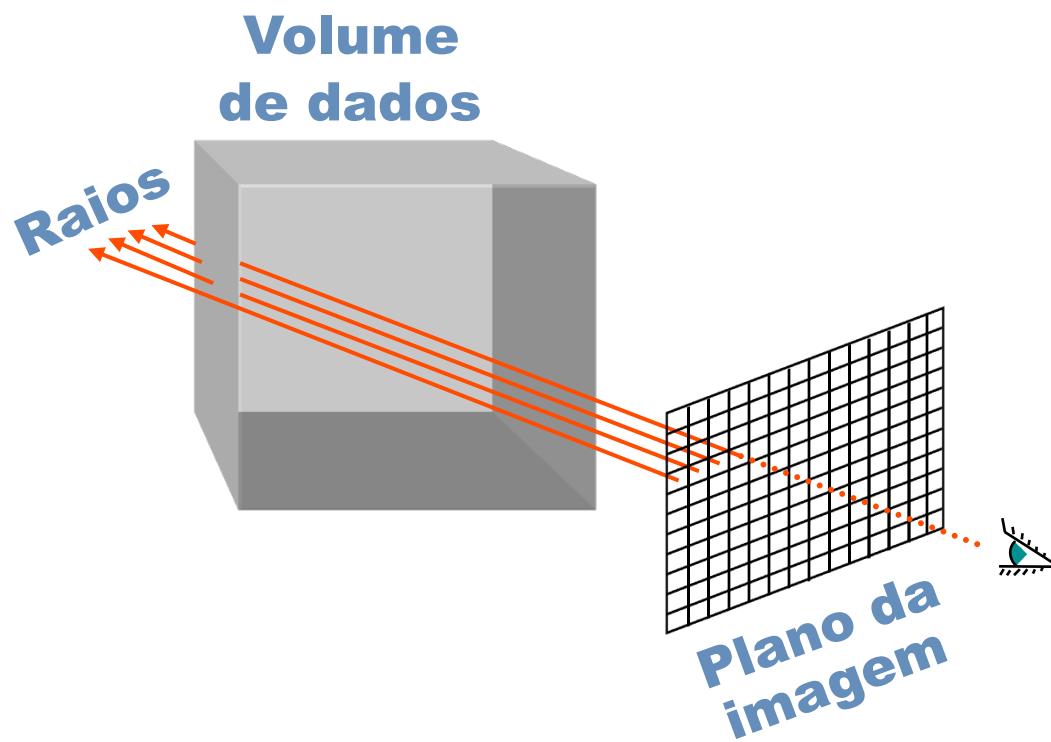
Integração na prática

- Se $FT = \text{cte}$
 - ⇒ Resultado analítico
- Se $FT = \text{linear}$ e $S = \text{linear}$
 - ⇒ Solução analítica complexa
- Variação qq
 - ⇒ Integração numérica
 - ⇒ Pré-integração
 - ✧ Armazenamento em textura
 - Problemas de discretização
 - ✧ Integração adaptativa

Métodos de visualização

- *Ray-casting*
- *Cell projection*
- *Shear-Warp*
- *Splatting*
- *Slicing*

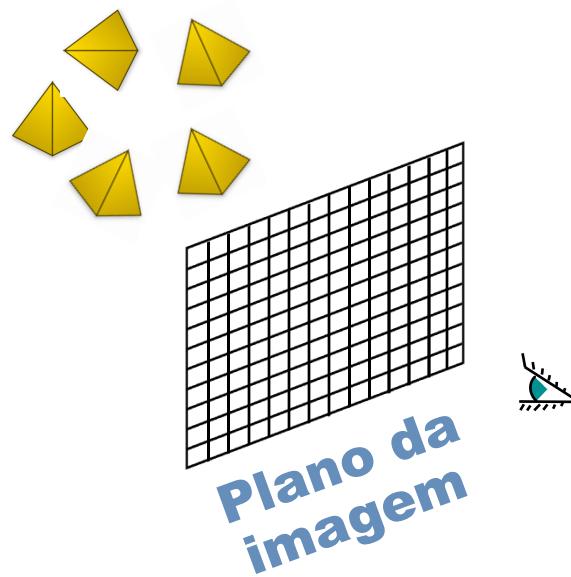
Algoritmo ray-casting



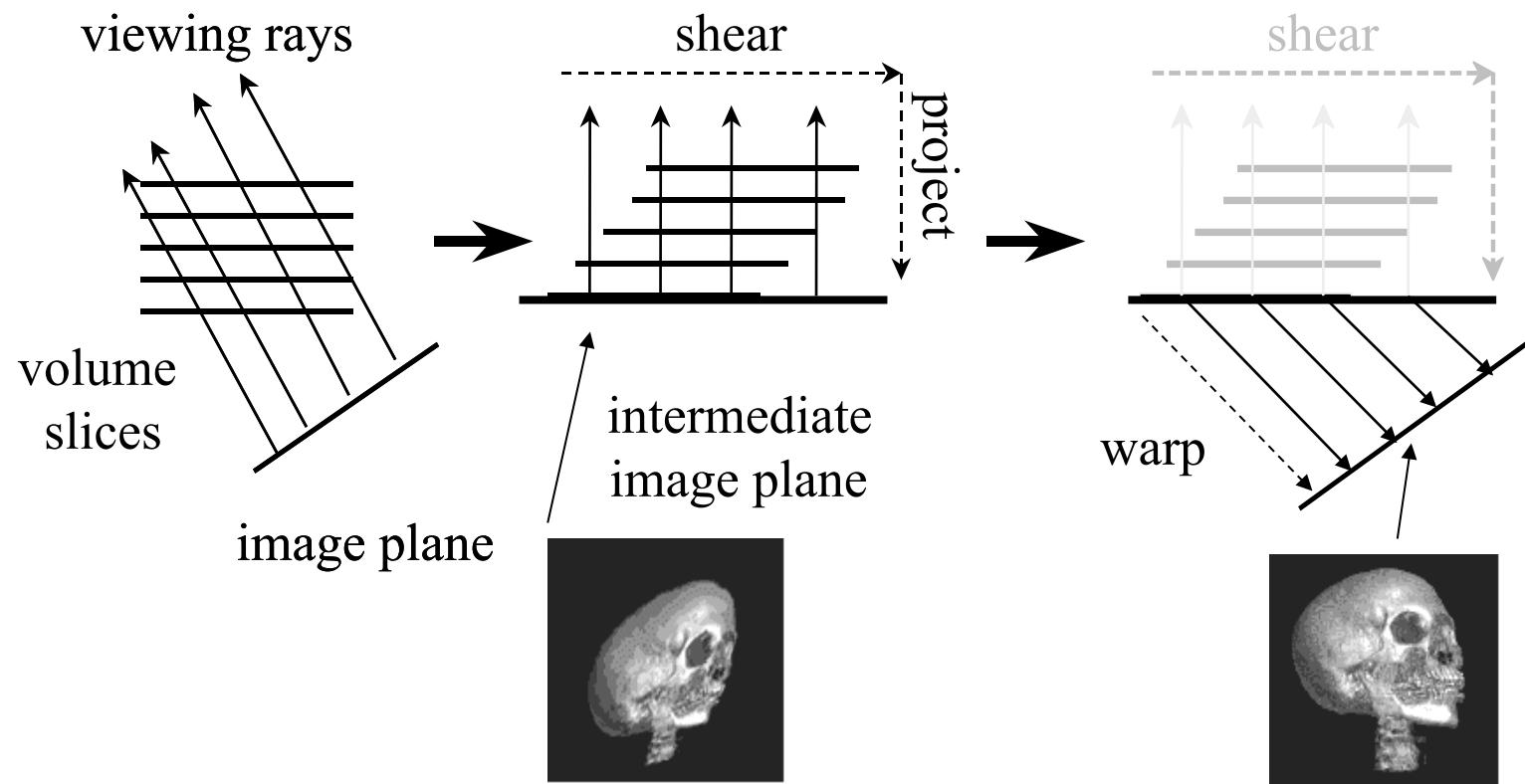
- Ordem da imagem.
 - para cada pixel
 - lance um raio e encontre os *voxels* que são interceptados
 - fim para

Algoritmo *cell projection*

- Célula (tetraedro) como primitiva gráfica
 - Desenho em ordem-z para composição

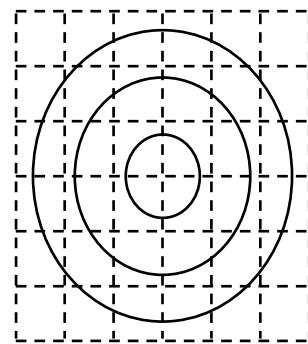


Shear-Warp



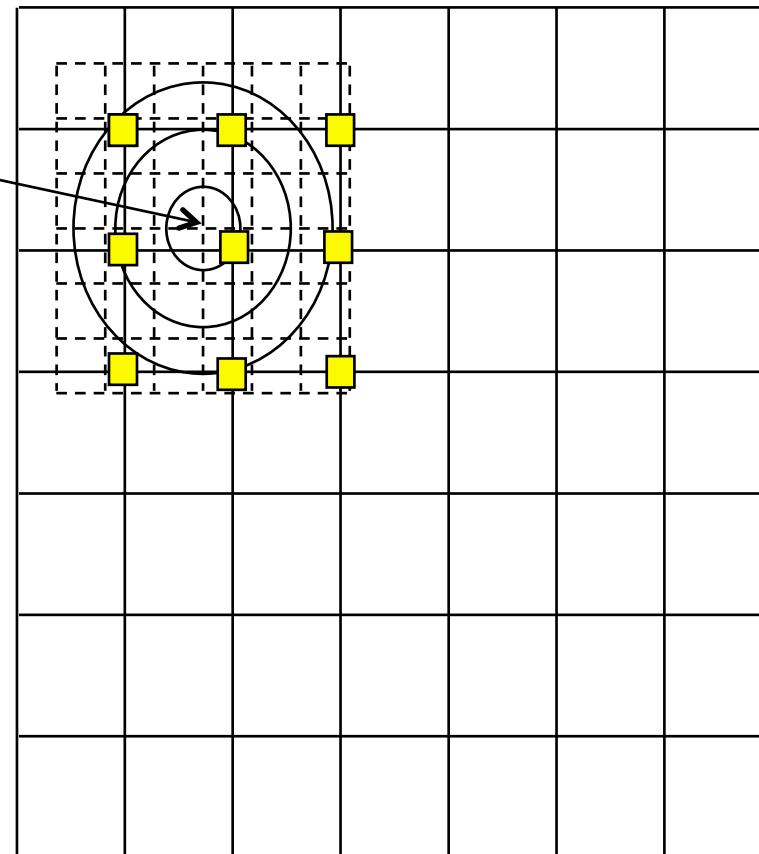
$$M_{\text{view}} = M_{\text{warp2D}} * M_{\text{shear3D}}$$

Splatting

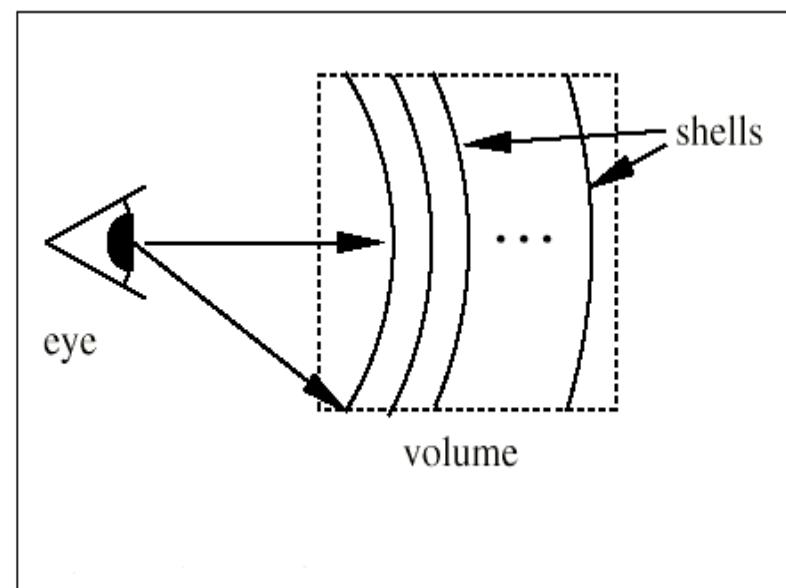
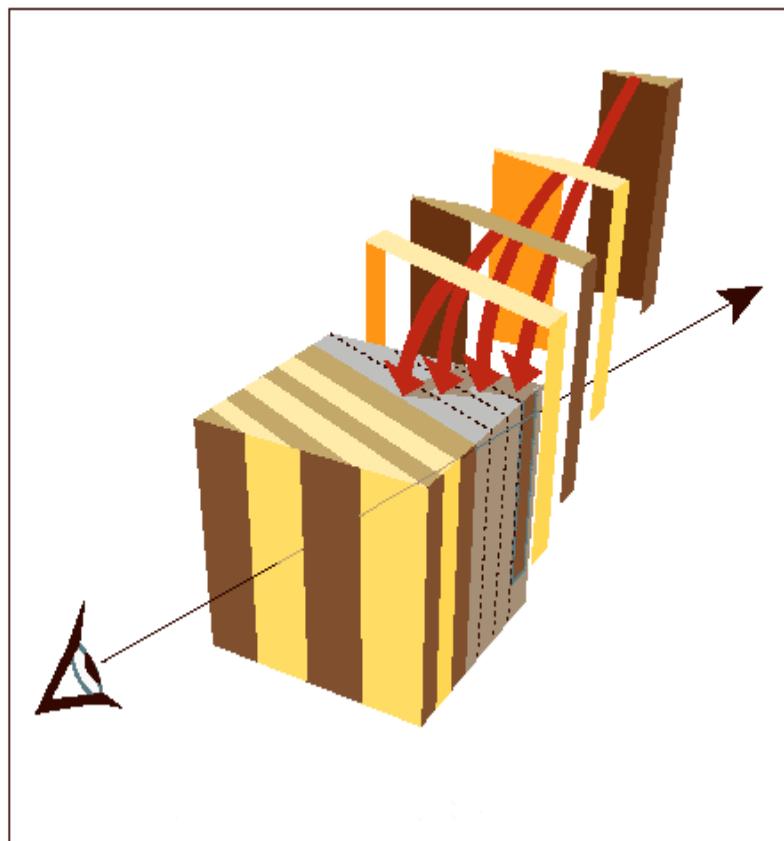


footprint

centro da projeção



Slicing (textura 3D)

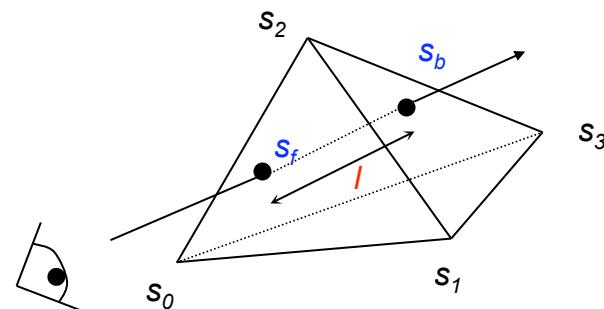


3D texture volume rendering

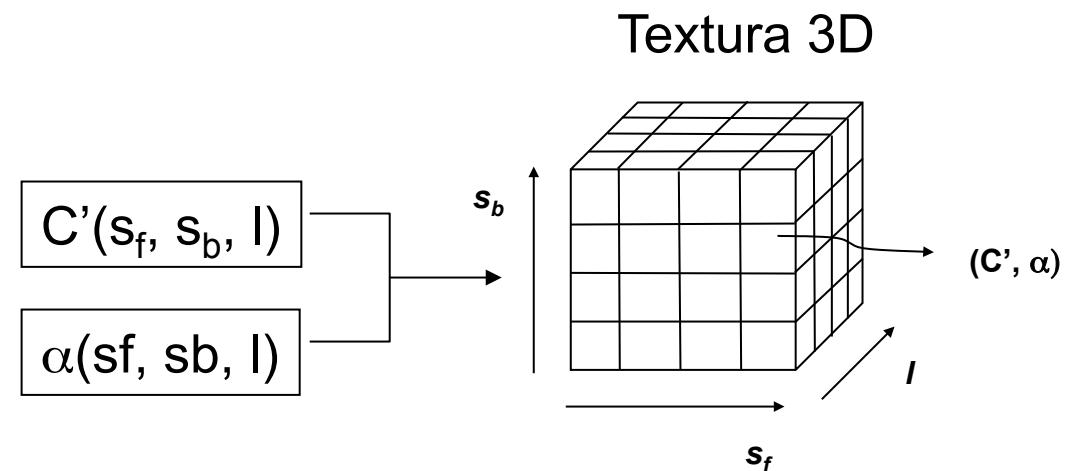
- Load the volume data into a 3D texture. This processing is done once for a particular data volume.
- Choose the number of slices. Usually this matches the texel dimensions of the volume data cube.
- Find the desired viewpoint and view direction
- Compute a series of polygons that cut through the data perpendicular to the direction of view. Use texture coordinate generation to texture the slice properly with respect to the 3D texture data.
- Use the texture transform matrix to set the desired orientation the textured images on the slices.
- Render each slice as a textured polygon, from back to front.
- As the viewpoint and direction of view changes, recompute the data slice positions and update the texture transformation matrix as necessary.

Aplicação do modelo óptico

- Pré-integração (Roettger et al., 2000)
 - ⇒ Integração numérica (ao modificar FT)
 - ⇒ Textura 3D

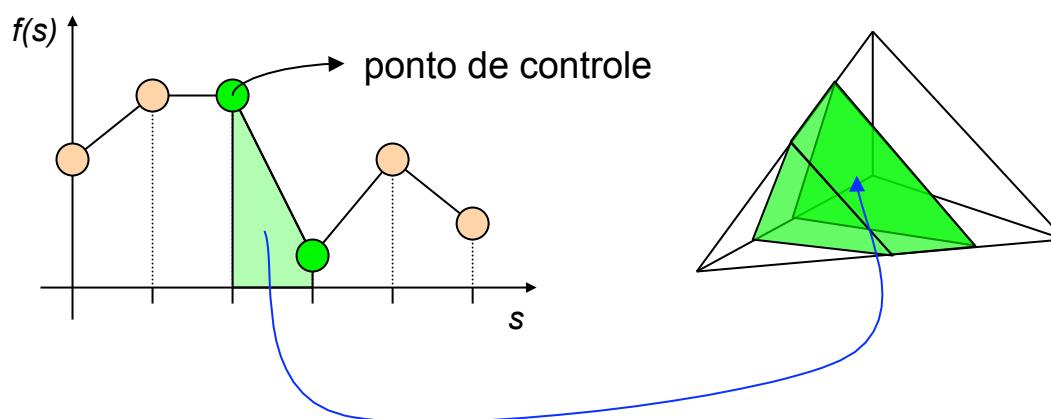


$$\alpha(s_f, s_b, l) = 1 - e^{-\int_0^1 \tau((1-t)s_f + t.s_b)l.dt}$$
$$C'(s_f, s_b, l) = \int_0^1 C((1-t)s_f + t.s_b) \tau((1-t)s_f + t.s_b) e^{-\int_0^{t'} \tau((1-t')s_f + t'.s_b)l.dt'} l.dt'$$



Aplicação do modelo óptico

- Segmento linear da FT (Williams et al., 1998)
 - ⇒ Solução analítica complexa
 - ⇒ Custo computacional



Aplicação do modelo óptico

- Formulação de Moreland & Angel (2004)
 - ⇒ Integração de um segmento linear da FT
 - ⇒ ψ independente da FT => pré-processamento
 - ◊ textura 2D
 - ⇒ ζ computado na GPU

$$I(D) = I_0 \cdot e^{-\int_0^D \tau(s(t')) dt'} + \int_0^D C(s(t)) \cdot \tau(s(t)) \cdot e^{-\int_t^D \tau(s(t')) dt'} dt$$

$$\begin{cases} \zeta(D, \tau_f, \tau_b) \equiv e^{-\frac{D}{2}(\tau_b + \tau_f)} \\ \psi(\gamma_f, \gamma_b) = \int_0^1 e^{-\int_t^1 \left(\frac{\gamma_b}{1-\gamma_b}(1-t') + \frac{\gamma_f}{1-\gamma_f}t' \right) dt'} dt \end{cases}$$
$$\gamma \equiv \frac{\tau \cdot D}{\tau \cdot D + 1}$$

$$I(D) = I_0 \cdot \zeta + C_b (\psi - \zeta) + C_f (1 - \psi)$$

Independente de FT!

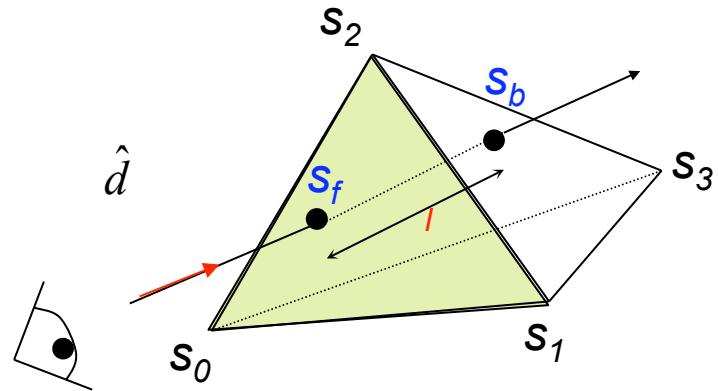
Algoritmos baseados na GPU

- Projeção de células
 - ⇒ View-independent cell projection (VICP) (Weiler et al., 2003b)
- Traçado de raios (Weiler et al., 2003a)

Cell-projection na GPU

□ Tetraedro linear: $s = ax + by + cz + d$

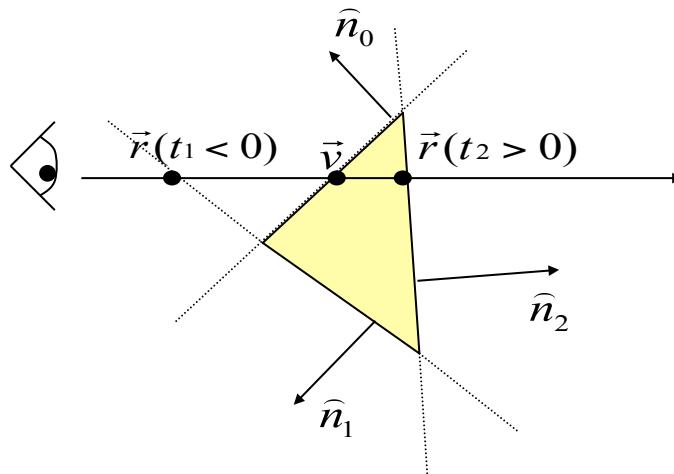
$$\vec{g} = \{a, b, c\}$$



Computar (s_f , s_b , l)

s_f = rasterização da face frontal

$$s_b = s_f + (\vec{g} \cdot \hat{d})l$$

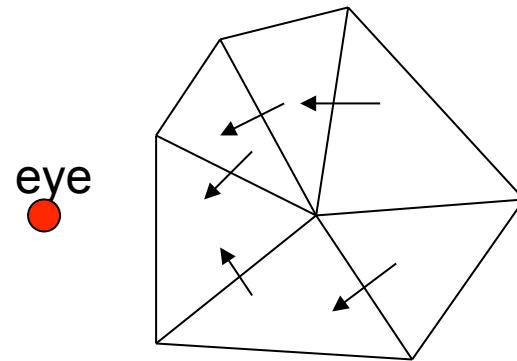


$$t_i = \frac{(\vec{p}_i - \vec{v}) \cdot \hat{n}_i}{\hat{d} \cdot \hat{n}_i}$$

$$l = \min(\max(0, t_i)), \quad i \in [0, 3]$$

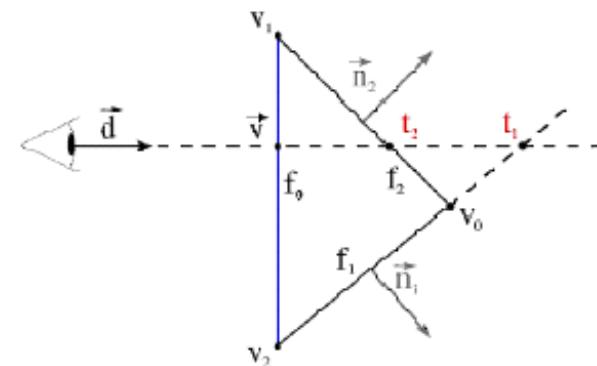
Ordenação de visibilidade

- **MPVO** [Williams1992]
 - ⇒ Construção de um *DAG*.
 - ⇒ Ordenação topológica do grafo.
 - ⇒ Malhas convexas e acíclicas.
 - ⇒ $O(n)$.
- Melhorias sobre o **MPVO**
 - ⇒ **MPVONC** [Williams1992]
 - ⇒ Heurística para malhas não-convexas.
 - ⇒ **XMPVO** [Silva et al. 1998]
 - ◊ Plano de varredura
 - ⇒ **BSP-XMPVO** [Comba et al. 1999]
 - ⇒ **Hardware assisted** [Krishnan et al. 2001]
 - ◊ Múltiplas renderizações
 - ⇒ **Cyclic meshes** [Kraus and Ertl 2001]
- Outros métodos
 - ⇒ **A-buffer** [Carpenter 1984]
 - ◊ Ordenação por fragmento da imagem
 - ⇒ **SXMPVO** [Cook et al. 2004]
 - ⇒ **Hardware assisted...** [Callahan et al. 2004]



Cell-projection na GPU

- Interseção raio com tetraedro
 - ⇒ Raio de visão intercepta face frontal em v
 - ✧ Corresponde a fragmento da face em azul
 - ⇒ No *fragment shader*
 - ✧ Calcula interseção com as outras faces
 - Dados passados como atributos (varying)
 - $t = ((v_o - v).n_i) / (d.n_i)$
 - ✧ Escolhe menor t com $t > 0$
 - Determina comprimento I
 - Se $t < 0$, interseção atrás



Cell-projection na GPU

□ Vertex shader

```
struct vert_in {  
    float4 vertex : POSITION;  
    float4 scalar : COLOR0; // Escalar normalizado (assumindo correcao da perspectiva)  
    float4 point : NORMAL; // Vértice oposto à face.  
    float3 normal0 : TEXCOORD0; // Normais das outras faces.  
    float3 normal1 : TEXCOORD1;  
    float3 normal2 : TEXCOORD2;  
    float3 gradient : TEXCOORD3; // Gradiente do escalar no tetraedro.  
};  
  
struct vert2frag {  
    float4 vertex : POSITION;  
    float4 sf : COLOR0;  
    float3 normal0 : TEXCOORD0;  
    float3 normal1 : TEXCOORD1;  
    float3 normal2 : TEXCOORD2;  
    float3 gradient : TEXCOORD3;  
    float3 dir : TEXCOORD4;  
    float3 position : TEXCOORD5; // Posição do vértice não transformada  
    float3 point : TEXCOORD6;  
};
```

Cell-projection na GPU

□ Vertex shader

```
vert2frag main(vert_in IN, uniform float4x4 ModelViewProj, uniform float3 EyePos)
{
    vert2frag OUT;
    OUT.vertex = mul(ModelViewProj, IN.vertex);
    OUT.sf      = IN.scalar;
    OUT.point   = IN.point.xyz;
    OUT.normal0 = IN.normal0;
    OUT.normal1 = IN.normal1;
    OUT.normal2 = IN.normal2;
    OUT.gradient = IN.gradient;
    OUT.dir     = IN.vertex.xyz - EyePos;
    OUT.position = IN.vertex.xyz;
    return OUT;
}
```

Cell-projection na GPU

□ Fragment shader

```
frag_out main(vert2frag IN, uniform float MaxLength, uniform sampler3D TransferFuncTex)
{
    frag_out OUT;
    float t = 10000.0; // Inicializa com valor alto.
    float f_reg1;
    float3 v_reg1;

    // Acha menor t : t = (P0 - V).Ni / D.Ni
    v_reg1 = IN.point - IN.position;
    f_reg1 = dot(IN.dir, IN.normal0);
    if (f_reg1 > 0.0f) t = dot(v_reg1, IN.normal0) / f_reg1;
    f_reg1 = dot(IN.dir, IN.normal1);
    if (f_reg1 > 0.0f) t = min(dot(v_reg1, IN.normal1)/f_reg1, t);
    f_reg1 = dot(IN.dir, IN.normal2);
    if (f_reg1 > 0.0f) t = min(dot(v_reg1, IN.normal2)/f_reg1, t);

    ...
}
```

Cell-projection na GPU

□ Fragment shader

...

```
// Calcula Sb = Sf + I(G.D), onde I==t, se D está normalizado.  
// Coordenadas de textura: Direct Volume Rendering: (sf, sb, I)  
v_reg1.x = IN.sf.x;  
v_reg1.y = IN.sf.x + t*dot(IN.gradient, IN.dir);  
v_reg1.z = length(t*IN.dir) / MaxLength;           // dir nao esta' normalizado  
  
OUT.color = tex3D(TransferFuncTex, v_reg1);  
return OUT;  
}
```

$$s_b = s_f + (\vec{g} \cdot \hat{d})l$$

View-independent cell projection (VICP)

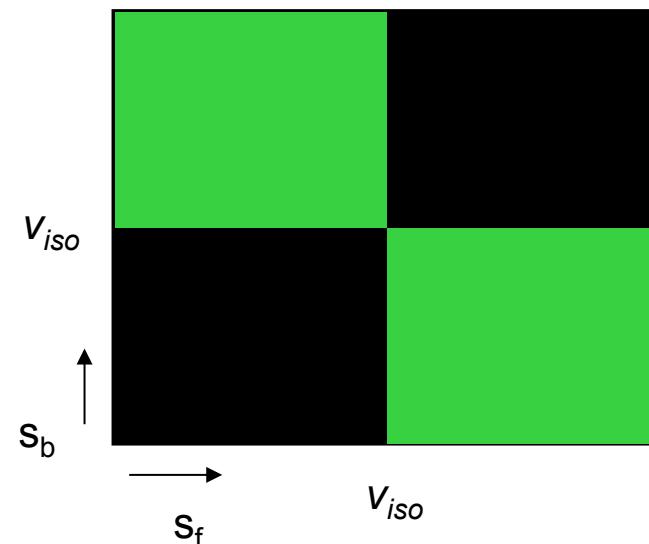
- Ordenação de células
 - ⇒ Estrutura de dados na CPU
- Informações dependentes da face
 - ⇒ 12 vértices+atributos enviados por tetraedro
- Gargalo: transferência de dados para a GPU
- Otimizações para projeção ortográfica

Cell-projection na GPU

□ Extração de iso-superfícies

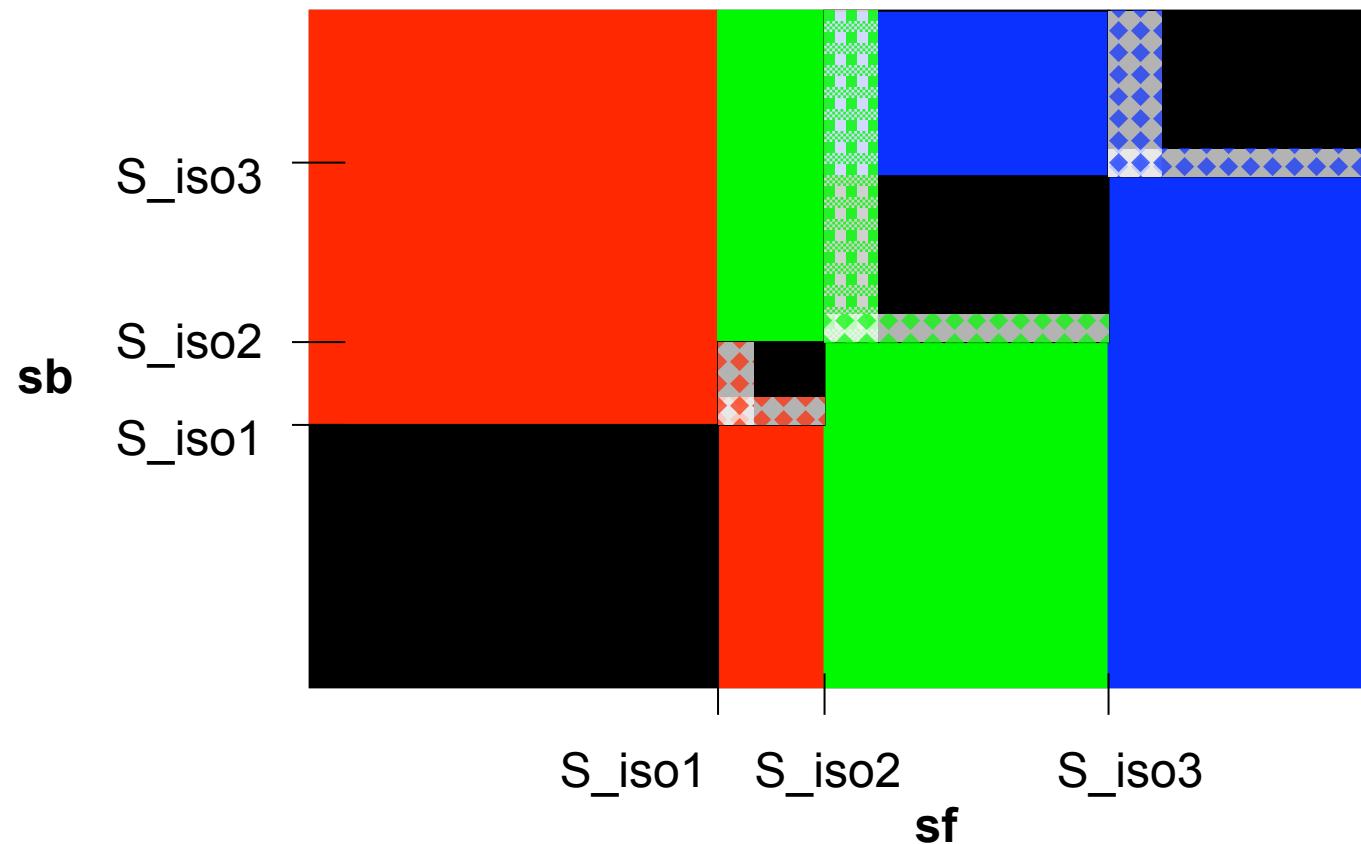
⇒ Textura 2D

- ✧ Sem atualizar valor de z-buffer
 - z do fragmento fica com o valor da face do tetraedro



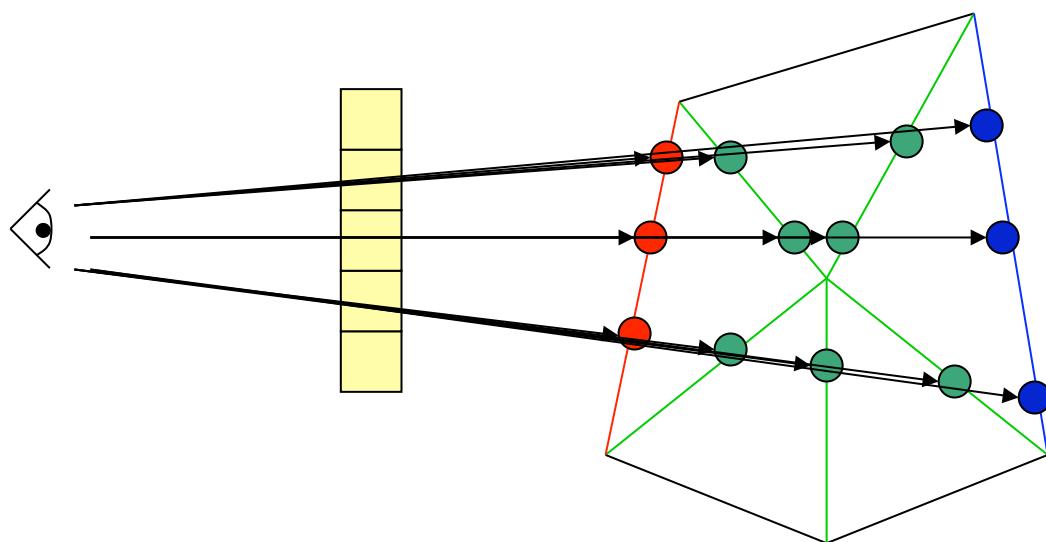
Cell-projection na GPU

- Extração de várias iso-superfícies
 - ⇒ E imprecisão numérica na fronteira (8 bits)



- Pode ser estendido para superfícies transparentes

Traçado de Raios



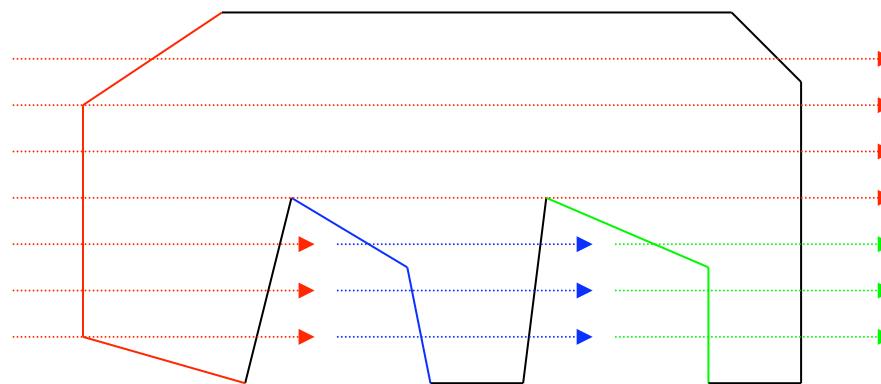
- 1.
- 2.

Desenhe faces frontais da malha
Para cada passo de propagação

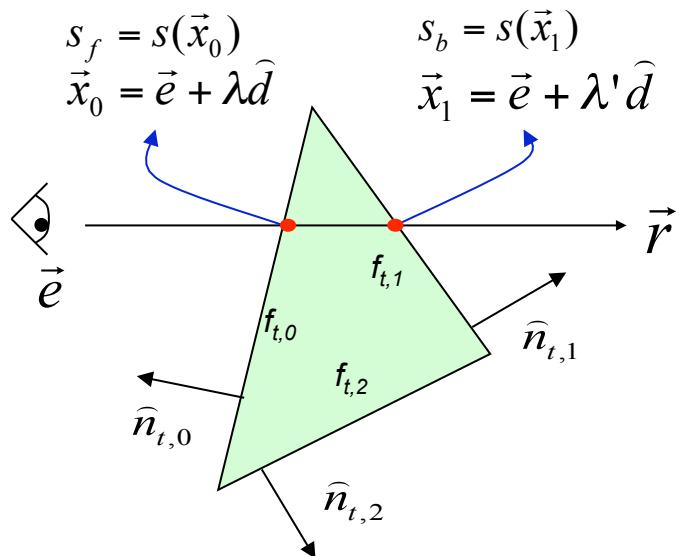
- ✧ Determine a posição de saída do tetraedro
- ✧ Calcule a contribuição do tetraedro
- ✧ Componha com o resultado anterior
- ✧ Avance para o tetraedro adjacente

Traçado de Raios

- Malhas não-convexas
 - ⇒ Descamação (*depth-peeling*)



Traçado de Raios



Computar (s_f , s_b , $|l|$)

$$s(\vec{x}) = \vec{g}_t \cdot (\vec{x} - \vec{x}_a) + s(\vec{x}_a) = \vec{g}_t \cdot \vec{x} + (s(\vec{x}_a) - \vec{g}_t \cdot \vec{x}_a) = \vec{g}_t \cdot \vec{x} + \hat{g}_t$$

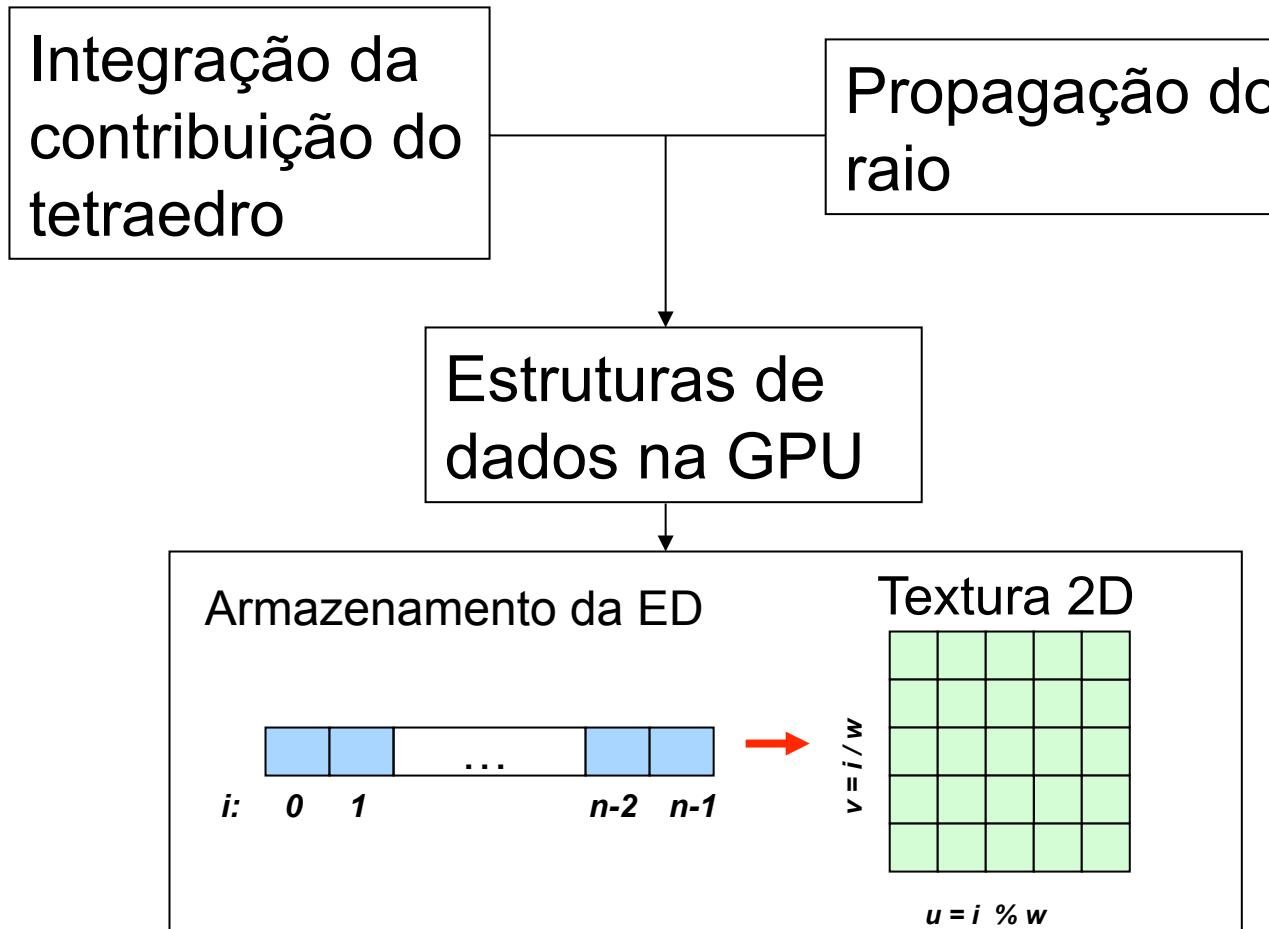
λ = computado no passo anterior

$$\lambda_i = \frac{(\vec{v}_{t,3-i} - \vec{e}) \cdot \hat{n}_{t,i}}{\hat{d} \cdot \hat{n}_{t,i}}$$

$$l = \min(\max(0, \lambda_i - \lambda)), \quad i \in [0,3]$$

Traçado de Raios

Estruturas de dados na GPU

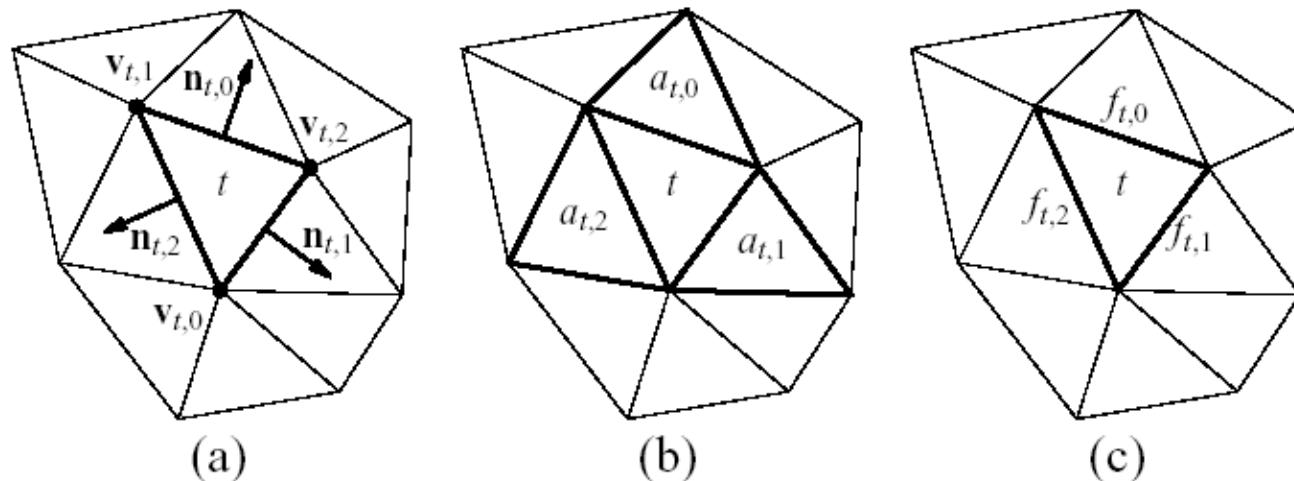


Ray-casting na GPU

□ Estrutura de dados topológica

⇒ Armazenamento em texturas (RGBA de 32 bits)

- ◊ Dados constantes
- ◊ Dados intermediários
 - Passados de quadro para quadro



Ray-casting na GPU

□ Estrutura de dados topológica

⇒ Dados da malha

- ✧ 3 texturas 3D: vértices, normais, adjacência
- ✧ 1 textura 2D: gradiente(\mathbf{g}), $\mathbf{g} = -\mathbf{g} \cdot \underline{\mathbf{x}_0} + \mathbf{s}(\underline{\mathbf{x}_0})$

data in texture	texture coords			texture data			
	u	v	w	r	g	b	α
vertices	t	i		$\mathbf{v}_{t,i}$			—
face normals	t	i		$\mathbf{n}_{t,i}$			$f_{t,i}$
neighbor data	t	i		$a_{t,i}$	—		—
scalar data	t	—		\mathbf{g}_t			\hat{g}_t

144 bytes / tetra

Ray-casting na GPU

□ Estrutura de dados topológica

⇒ Dados transientes (atualizados quadro a quadro)

✧ 3 texturas 2D: tetraedro corrente, posição e escalar do ponto de entrada corrente, cor acumulada

– Cor não é acumulada no frame buffer por causa de precisão

data in texture	texture coords			texture data			
	u	v	w	r	g	b	α
current cell	raster pos	—	—	t	j	—	—
intersect. pt.	raster pos	—	—	—	λ	$s(\mathbf{e} + \lambda\mathbf{r})$	—
color, opacity	raster pos	—	—	r	g	b	α

Traçado de Raios

Estruturas de dados na GPU

- Faixas (strips) de tetraedros (Weiler et al., 2004)
 - ⇒ Duas versões
 - ✧ Otimizada (desempenho) – 76 bytes/tetra
 - ✧ Compacta (memória) – 15 bytes/tetra
 - Apenas vértices, escalares e adjacências armazenados
 - ⇒ Importante maximizar
 - ✧ Comprimento das faixas – heurísticas
 - ✧ Utilização das linhas da textura 2D
 - ⇒ Compressão da malha
 - ✧ 56%, obtidos por Weiler et al. (2004)
 - ✧ Limite teórico de aproximadamente 63%
 - ⇒ Complexidade alta
 - ⇒ Muito difícil modificar a malha (ex. multi-resolução)

Ray-casting em GPU

- Implementação do algoritmo
 - ⇒ Criar pbuffers em formato FLOAT32
 - ✧ Suporte para *multiple render targets*
 - ⇒ Carregar estrutura de dados em texturas na GPU.
 - ⇒ Desenhar front-faces para obter dados transientes iniciais
 - ✧ Tetraedro
 - ✧ Escalar de entrada
 - ✧ Parâmetro do raio
 - ⇒ Loop
 - ✧ “Swap buffers”
 - Resultado vira textura
 - Textura vira target
 - ✧ Desenhar um retângulo para ativar fragment shader
 - Marcar o pbuffer com as mesmas informações do passo inicial
 - Acumular contribuição de cor
 - ✧ Repita enquanto houver alguma modificação em pbuffer[curr]
 - OCCLUSION_QUERY
 - ⇒ Copia cor resultante para color-buffer

Dissertação de Rodrigo Espinha

- Modificações propostas
 - ⇒ VICP (view-independent cell projection)
 - ✧ Estrutura de dados na GPU
 - ⇒ Traçado de Raios
 - ✧ Estrutura de dados utilizada
 - ✧ Integração de segmentos lineares na GPU

VICP – Estrutura de dados na GPU

- Motivação

- ⇒ Eliminar gargalo na transferência de dados

VICP – Estrutura de dados na GPU

Textura	Coordenadas		Dados			
	u	v	r	g	b	a
Vértices	k		\vec{v}_k			s_k
Normais0	t		$\hat{n}_{t,0}$			$o_{t,0}$
Normais1	t		$\hat{n}_{t,1}$			
Normais2	t		$\hat{n}_{t,2}$			
Normais3	t		$\hat{n}_{t,3}$			$o_{t,3}$
Gradientes	t		\vec{g}_t			

- Por vértice
 - ⇒ Índice do tetraedro (t)
 - ⇒ Índice do vértice (k)
- Por tetraedro
 - ⇒ Faixas de triângulos
 - ⇒ 6 vértices / tetraedro
 - ⇒ **48 bytes / tetra**

Total: $71 + 48 = 119$ bytes/tetra

71 bytes/tetra

Traçado de Raios – ED utilizada

□ Motivações

- ⇒ Eficiência x memória
- ⇒ Baixa complexidade
- ⇒ Modificações mais fáceis que as faixas de tetraedros

Traçado de Raios – ED utilizada

Textura	Coordenadas		Dados			
	u	v	r	g	b	a
Normais0	t		$\hat{n}_{t,0}$			$o_{t,0}$
Normais1	t		$\hat{n}_{t,1}$			$o_{t,1}$
Normais2	t		$\hat{n}_{t,2}$			$o_{t,2}$
Normais3	t		$\hat{n}_{t,3}$			$o_{t,3}$
Gradientes	t		\vec{g}_t			\hat{g}_t
Adjacências	t		$a_{t,0}$	$a_{t,1}$	$a_{t,2}$	$a_{t,3}$

96 bytes / tetra

Original: 144 bytes / tetra

Faixas de tetraedros: 15 a 76 bytes / tetra

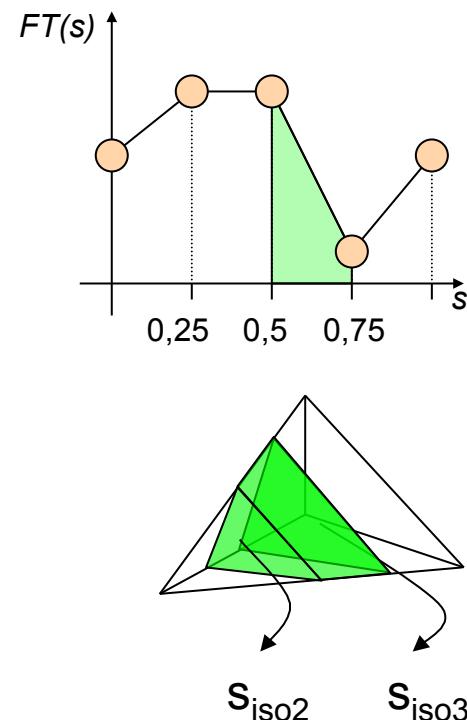
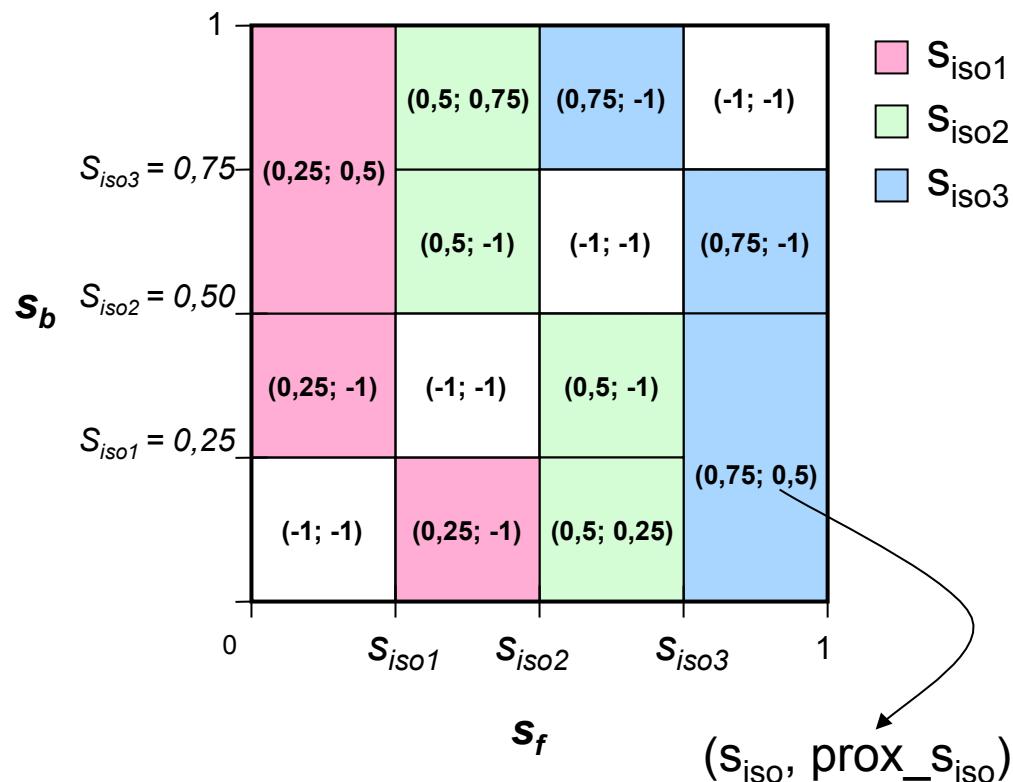
Integração na GPU

□ Motivações

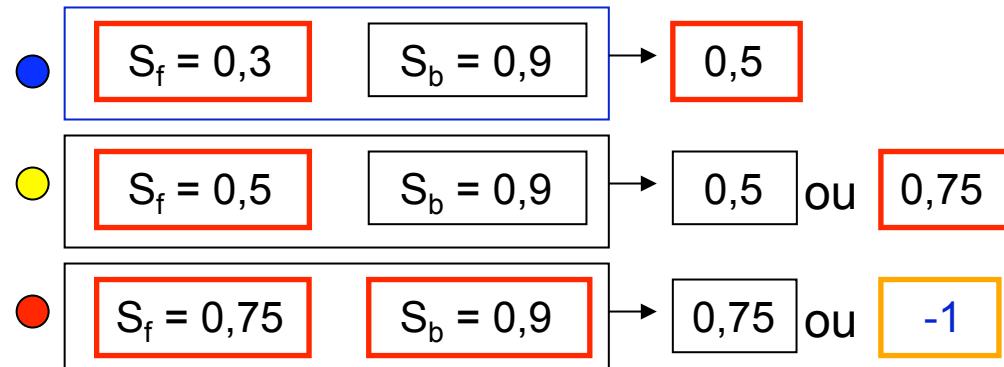
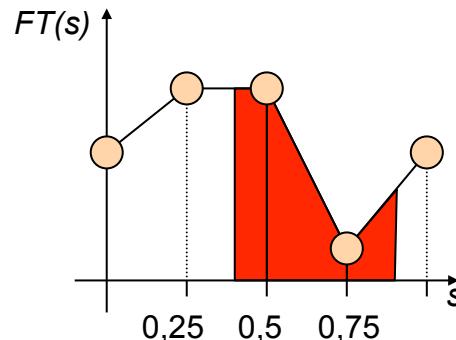
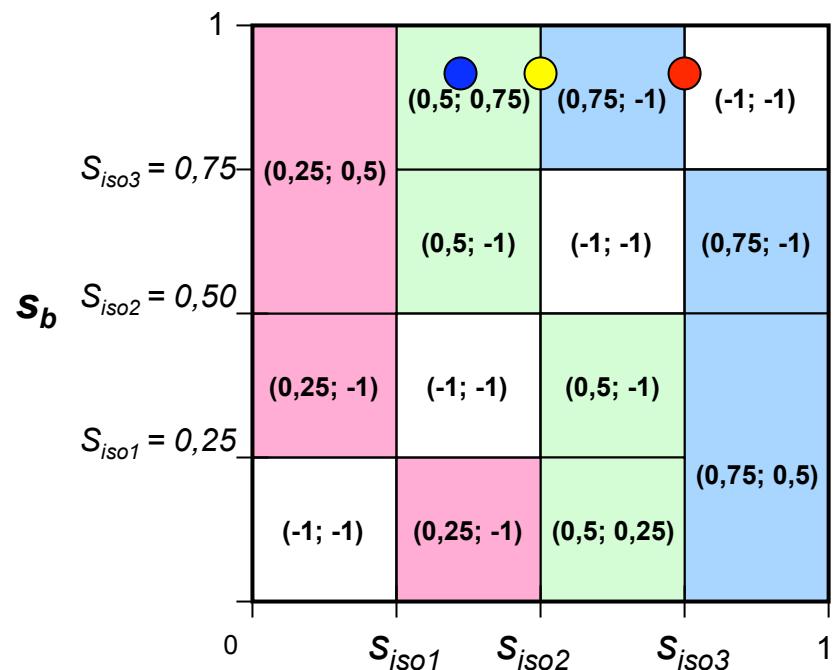
- ⇒ Melhor qualidade de imagem
- ⇒ Modificações interativas da FT

Integração na GPU

- Integração de segmentos lineares da FT
- Iso-superfícies codificadas em textura 2D



Integração na GPU

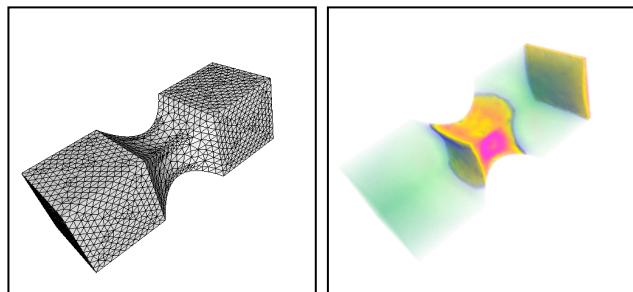


Resultados

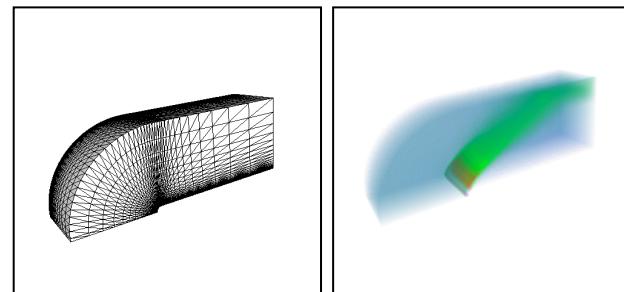
- VICP (CPU)
- VICP (GPU)
- Traçado de Raios (Pré-integração)
- Traçado de Raios (Integração na GPU)

Resultados

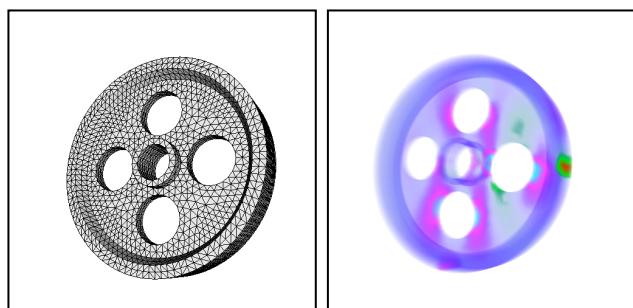
Barra Fixa - 24.576 tetra



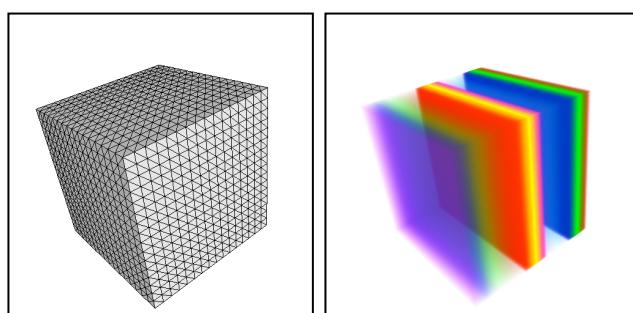
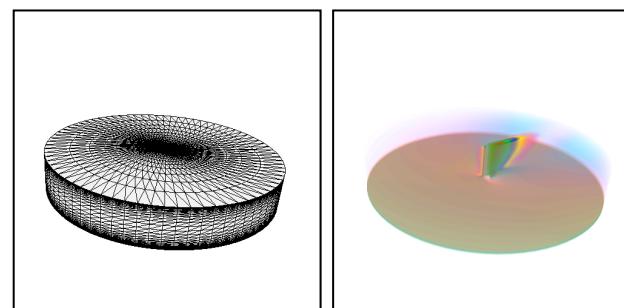
Bluntnfin - 224.874 tetra



Roda - 31.725 tetra



Oxygen - 616.050 tetra



Grade 16x16x16 - 24.576 tetra

Grade 32x32x32 - 196.608 tetra

Resultados Desempenho

□ VICP (CPU) x VICP (GPU)

Malhas	VICP (CPU)				VICP (GPU)			
	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s
Grade 16x16x16	11,85	12,55	291K	308K	14,53	21,32	357K	524K
Grade 32x32x32	1,43	1,51	281K	297K	3,42	3,74	672K	735K
Barra Fixa	9,84	10,67	272K	295K	24,63	26,67	682K	739K
Roda	8,76	9,28	278K	294K	22,08	22,88	700K	726K
Bluntnfin	1,26	1,36	283K	306K	3,09	3,50	695K	787K
Oxygen	0,46	0,47	283K	290K	0,98	1,21	604K	745K

Resultados

Desempenho

- ❑ Ordenação de células (VICP)
 - ⇒ Fator limitante para malhas grandes

Malhas	Tempo (s)	
	Min.	Máx.
Grade 16x16x16	0,015 (66,67 qd/s)	0,031
Grade 32x32x32	0,23	0,27
Bluntnfin	0,25	0,30
Oxygen	0,75 (1,33 qd/s)	0,82

Resultados Desempenho

- VICP (GPU) x Traçado de Raios (Pré-integração)

Malhas	VICP (GPU)				Traçado de Raios (Pré-integração)			
	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s
Grade 16x16x16	14,53	21,32	357K	524K	8,53	11,63	210K	286K
Grade 32x32x32	3,42	3,74	672K	735K	3,85	5,98	757K	1,18 M
Barra Fixa	24,63	26,67	682K	739K	8,76	14,88	243K	412K
Roda	22,08	22,88	700K	726K	7,80	14,53	247K	461K
Bluntnfin	3,09	3,50	695K	787K	6,09	8,76	1,37 M	1,97 M
Oxygen	0,98	1,21	604K	745K	3,40	5,91	2,09 M	3,64 M

Resultados Desempenho

□ Traçado de Raios

⇒ Pré-integração x Integração na GPU

Malhas	Traçado de Raios (Pré-integração)				Traçado de Raios (Integração na GPU) 10 seg.			
	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s
Grade 16x16x16	8,53	11,63	210K	286K	6,15	9,14	151K	225K
Grade 32x32x32	3,85	5,98	757K	1,2M	3,00	5,08	590K	999K
Barra Fixa	8,76	14,88	243K	412K	6,27	11,22	174K	311K
Roda	7,80	14,53	247K	461K	7,35	10,67	233K	339K
Bluntnfin	6,09	8,76	1,4M	2,0M	3,50	7,11	787K	1,6M
Oxygen	3,40	5,91	2,1M	3,6M	2,32	4,21	1,4M	2,6M

Resultados Desempenho

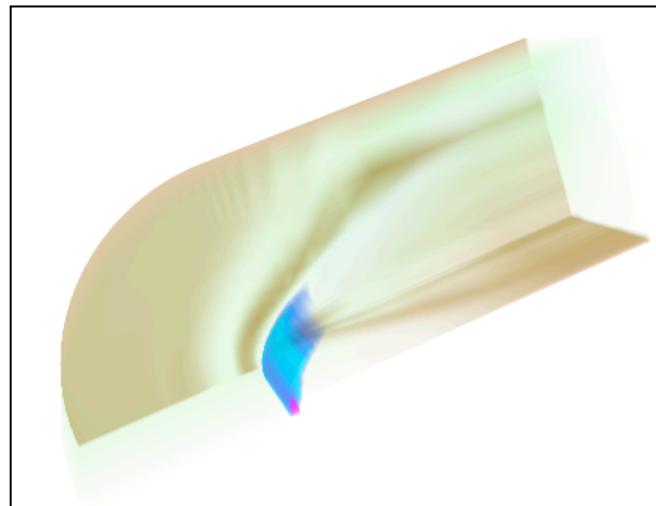
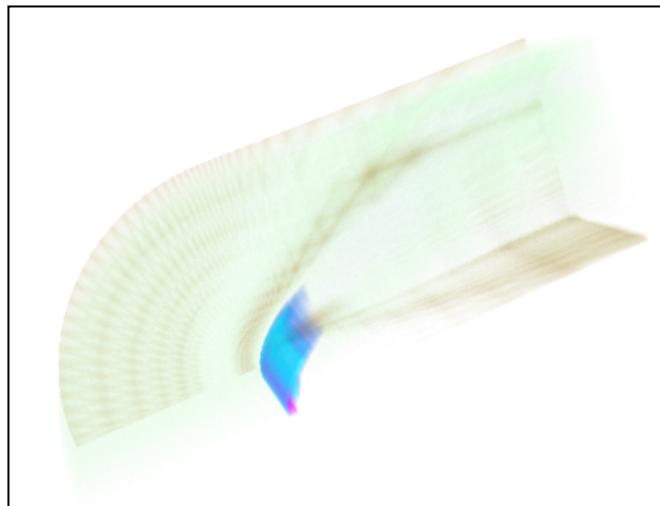
□ Traçado de Raios (Integração na GPU)

Malhas	Traçado de Raios (Integração na GPU) 10 seg.				Traçado de Raios (Integração na GPU) 255 seg.			
	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s	Min. qd/s	Máx. qd/s	Min. tet/s	Máx. tet/s
Grade16x16x16	6,15	9,14	151K	225K	2,02	5,76	50K	142K
Grade32x32x32	3,00	5,08	590K	999K	1,71	3,30	336K	649K
Barra Fixa	6,27	11,22	174K	311K	2,41	4,81	67K	133K
Roda	7,35	10,67	233K	339K	3,86	7,19	122K	228K
Bluntnfin	3,50	7,11	787K	1,60M	2,98	5,04	670K	1,13M
Oxygen	2,32	4,21	1,43M	2,59M	1,22	2,38	752K	1,47M

Resultados

Qualidade da imagem

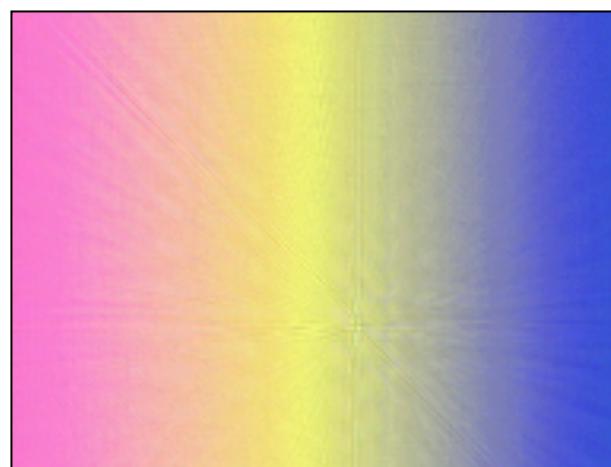
□ Pré-integração x Integração na GPU



Resultados

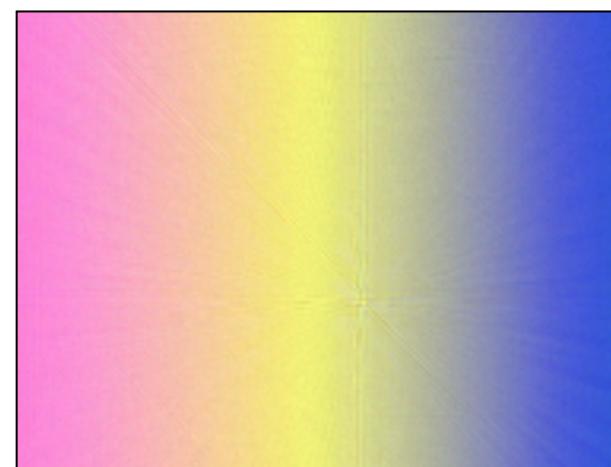
Qualidade da imagem – Precisão

VICP



Frame buffer convencional
8 bits/componente

Traçado de Raios
(Pré-integração)



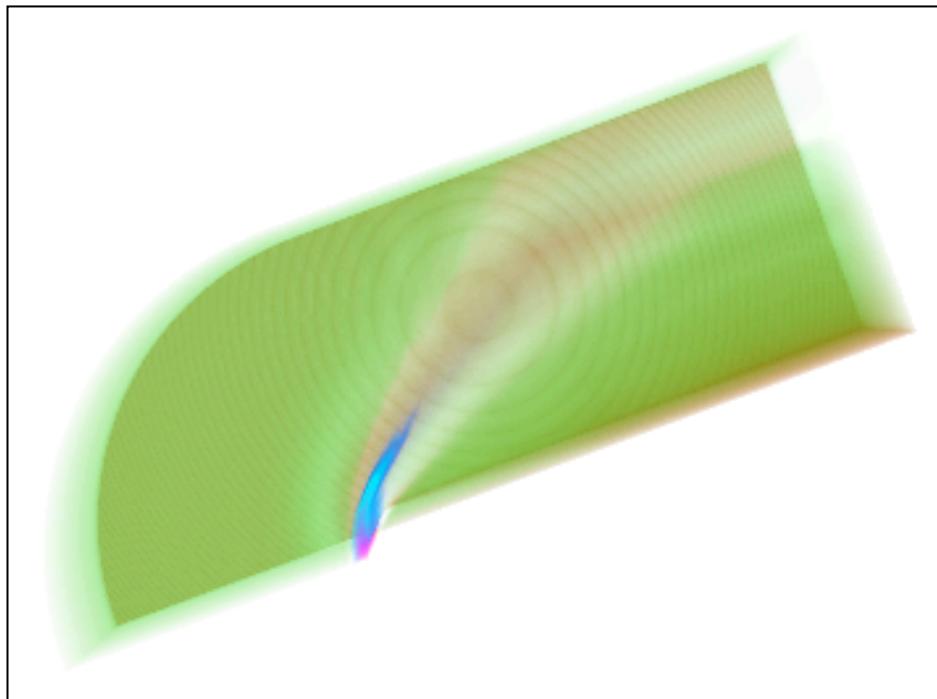
32 bits por componente

Resultados

Qualidade da imagem - Precisão

- Traçado de Raios

- ⇒ Com parâmetro do raio armazenado em tipo *half* (16 bits)



Resultados

□ Memória

⇒ VICP (GPU)

- ✧ Replicação de dados na CPU

⇒ Traçado de Raios

- ✧ Dados na GPU

⇒ Pode ser fator limitante para grandes malhas

Resultados

- Modificação interativa da FT
 - ⇒ Pré-integração com auxílio da GPU
 - ✧ 128x128x128, 32 segmentos: 1,3s
 - ⇒ Integração na GPU
 - ✧ Praticamente instantânea