



# MAC420/5744: Introdução à Computação Gráfica

---

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

## Aula #15: Mapeamento de textura II

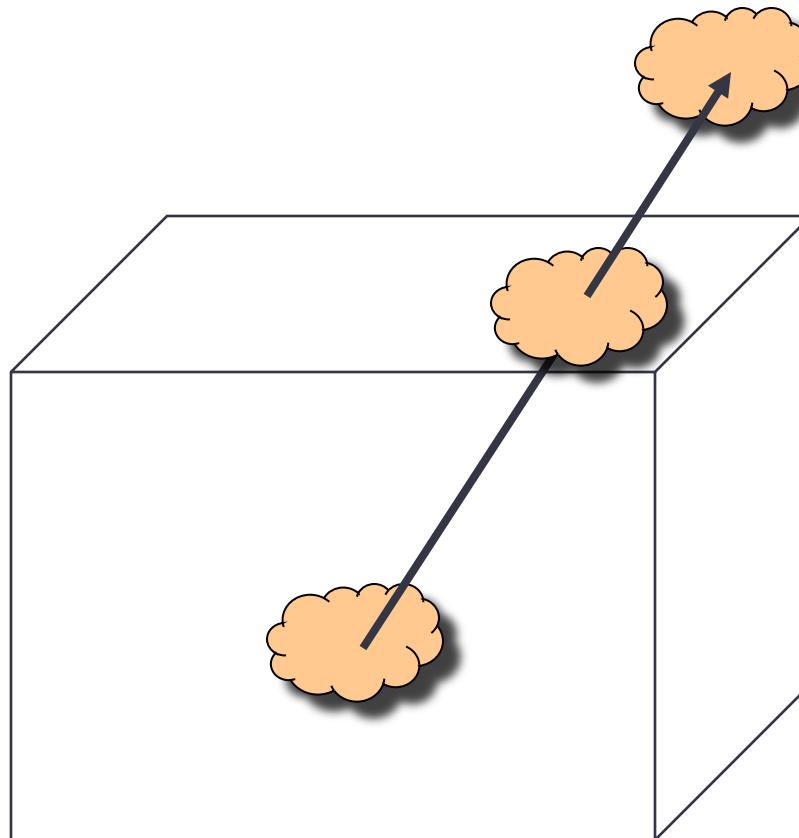
# Mapeamento de ambiente

---

- Mapeamento que simula a aparência altamente especular de certos objetos sem usar técnicas globais de renderização
  - Comum em filmes e video games
  - É uma forma de mapeamento de textura
- Também podemos simular efeitos da refração

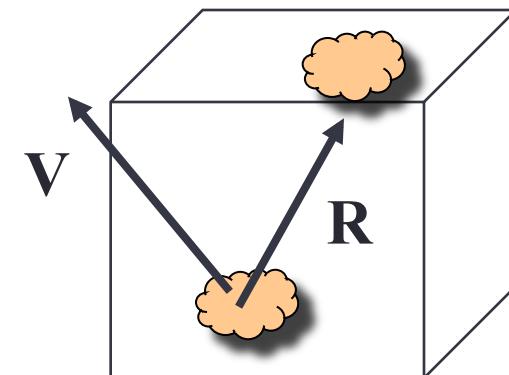
# *Cube mapping*

---

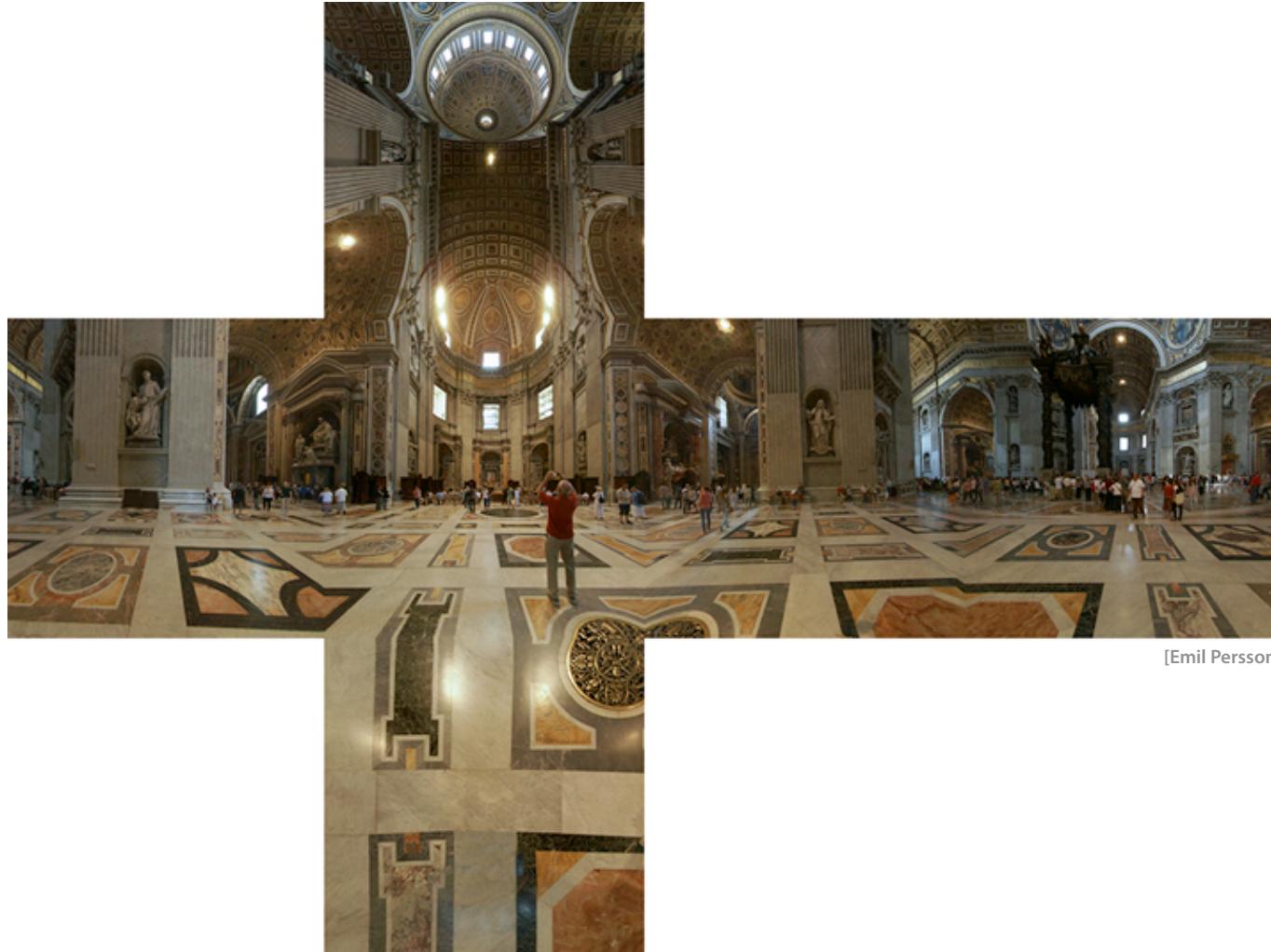


# Determinação das faces

- Calcular o vetor reflexão
  - $R = 2(N \cdot V)N - V$
- Usar a maior magnitude dos componentes de  $R$  para determinar a face do cubo
- Outros dois componentes resultam em coordenadas de textura

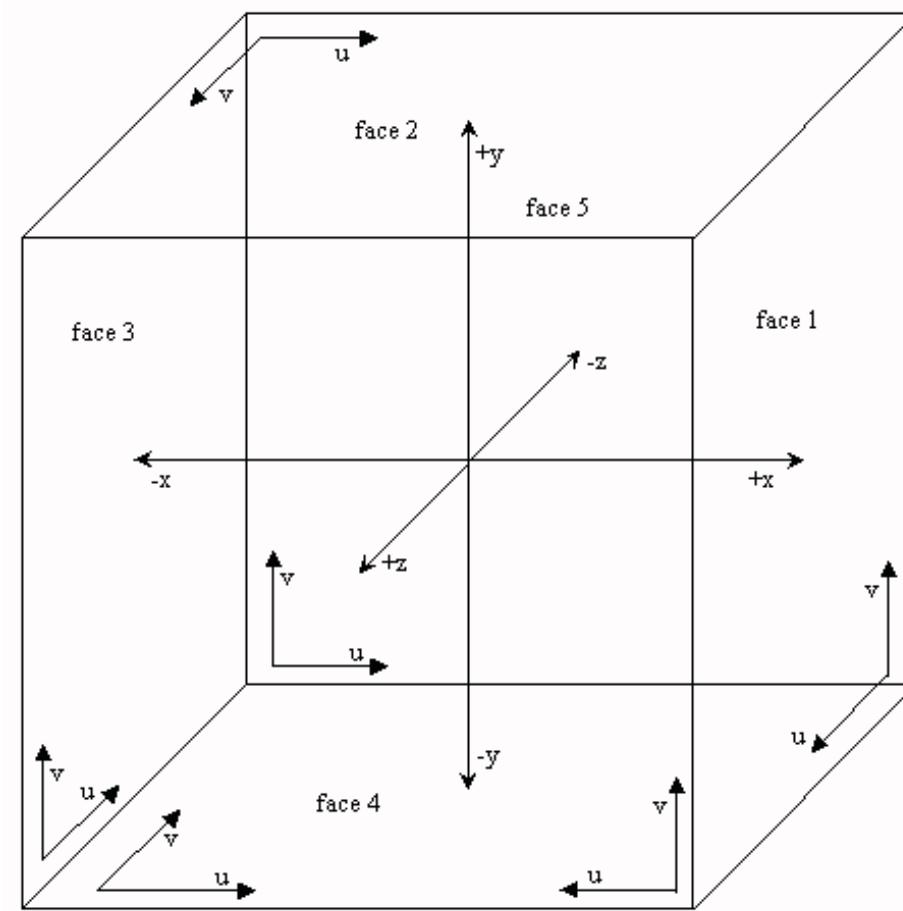


# Mapeamento cúbico



[Emil Persson]

# Indexação das faces



96.82



# Mapeamentos em WebGL/OpenGL

---

- WebGL suporta somente mapeamentos em cubos
  - OpenGL desktop também oferece mapeamento esférico
- Primeiramente criam-se os mapas
  - Imagens a partir de câmeras
  - Criação de imagens com WebGL/OpenGL
- Efetua-se o mapeamento de textura

# Mapeamentos cúbicos

---

- Podemos formar uma mapeamento em cubo definindo 6 texturas 2D que correspondem aos lados de um cubo
- O WebGL oferece um *sampler* do tipo *cubemap*  
`vec4 texColor = textureCube(mycube, texcoord);`
- Coordenadas de textura em 3D
  - Normalmente especificados juntamente com os vértices

# Considerações

---

- Devemos assumir que o ambiente está bem longe do objeto
- Objetos não devem ser côncavos (ausência de auto-reflexões)
- Sem reflexões entre objetos
- Precisamos de uma mapa de reflexão para cada objeto

# Em WebGL

---

```
gl.textureMap2D(  
    gl.TEXTURE_CUBE_MAP_POSITIVE_X,  
    level, rows, columns, border, gl.RGBA,  
    gl.UNSIGNED_BYTE, image1)
```

- A chamada é semelhante para as outras 5 imagens
- Cria-se então um *texture object* com as 6 imagens

# Exemplo

---

- Considere um cubo rotacionando que reflete cores das paredes de um quarto
- Cada parede é feita de uma cor (vermelho, verde, azul, ciano, magenta, amarelo)
  - Cada face pode ser uma textura contendo um texel

```
var red = new Uint8Array([255, 0, 0, 255]);
var green = new Uint8Array([0, 255, 0, 255]);
var blue = new Uint8Array([0, 0, 255, 255]);
var cyan = new Uint8Array([0, 255, 255, 255]);
var magenta = new Uint8Array([255, 0, 255, 255]);
var yellow = new Uint8Array([255, 255, 0, 255]);
```

# Objeto de textura

---

```
cubeMap = gl.createTexture();
gl.bindTexture(gl.TEXTURE_CUBE_MAP, cubeMap);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_X, 0, gl.RGBA,
    1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE, red);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_X, 0, gl.RGBA,
    1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE, green);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_Y, 0, gl.RGBA,
    1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE, blue);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, gl.RGBA,
    1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE, cyan);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_Z, 0, gl.RGBA,
    1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE, yellow);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, gl.RGBA,
    1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE, magenta);
gl.activeTexture( gl.TEXTURE0 );
gl.uniform1i(gl.getUniformLocation(program, "texMap"), 0);
```

# Vertex shader

---

```
varying vec3 R;
attribute vec4 vPosition;
attribute vec4 vNormal;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec3 theta;

void main(){
    vec3 angles = radians( theta );
    // compute rotation matrices rx, ry, rz here
    mat4 ModelViewMatrix = modelViewMatrix*rz*ry*rx;
    gl_Position = projectionMatrix*ModelViewMatrix*vPosition;
    vec4 eyePos = ModelViewMatrix*vPosition;
    vec4 N = ModelViewMatrix*vNormal;
    R = reflect(eyePos.xyz, N.xyz);
}
```

# *Fragment shader*

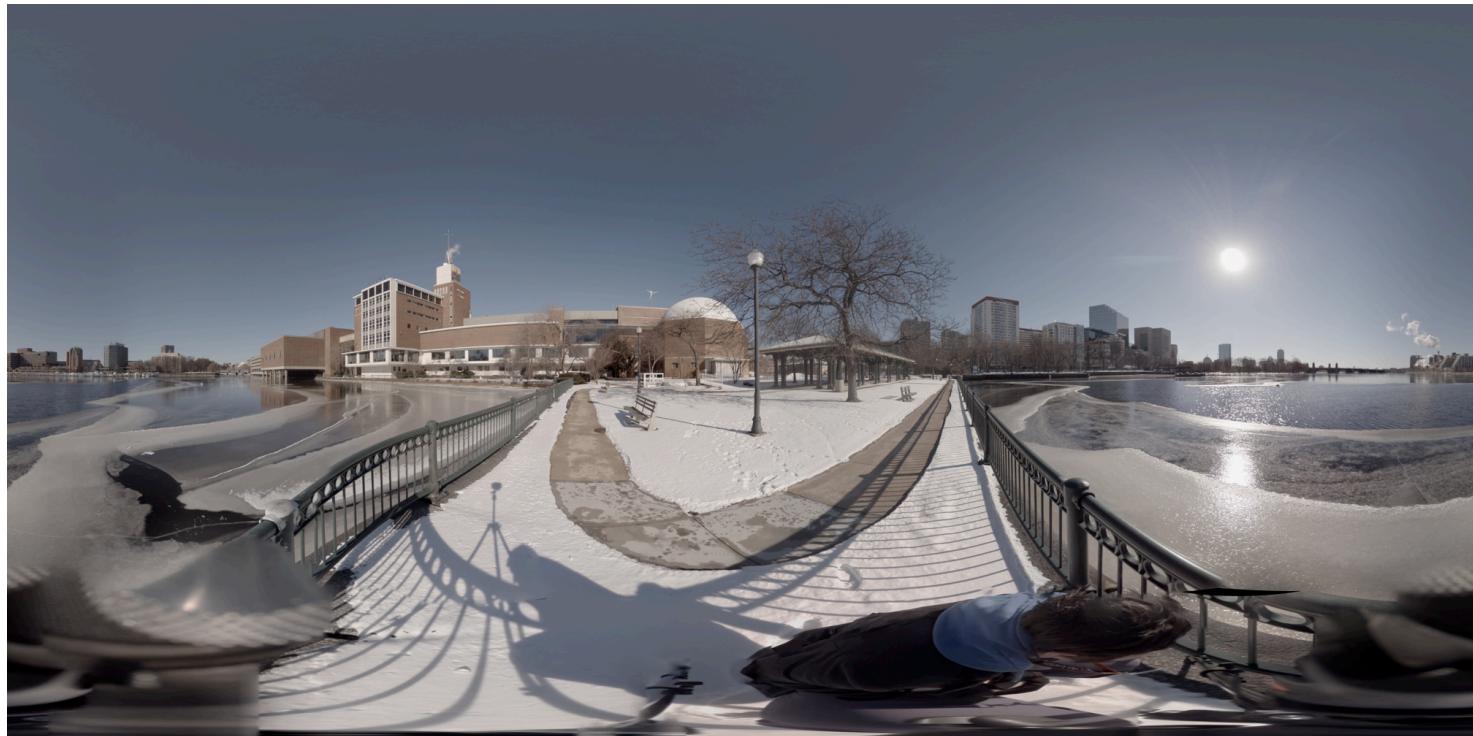
---

```
precision mediump float;  
  
varying vec3 R;  
uniform samplerCube texMap;  
  
void main()  
{  
    vec4 texColor = textureCube(texMap, R);  
    gl_FragColor = texColor;  
}
```

# Mapeamento esférico

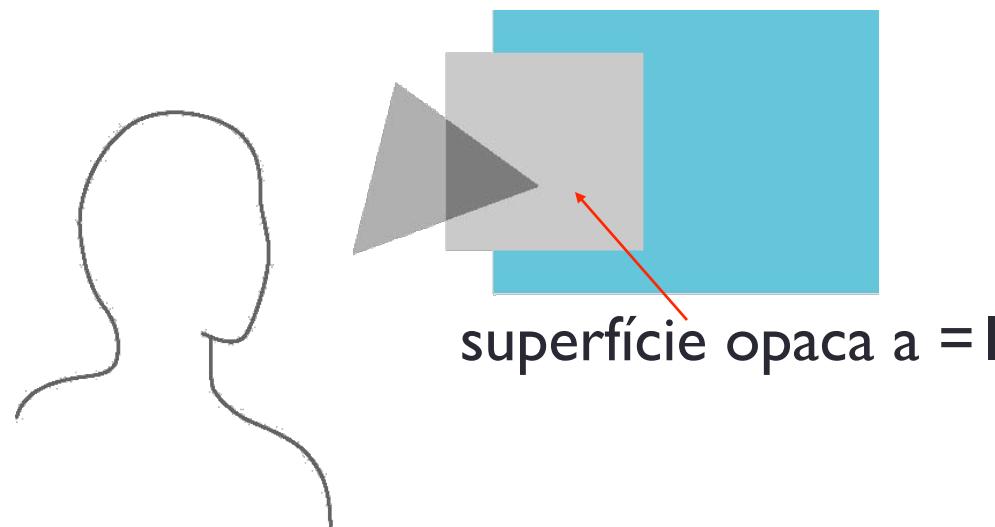
---

- O OpenGL suporta mapeamentos esféricos porém requer que uma textura circular equivalente a uma imagem adquirida com uma lente olho de peixe.



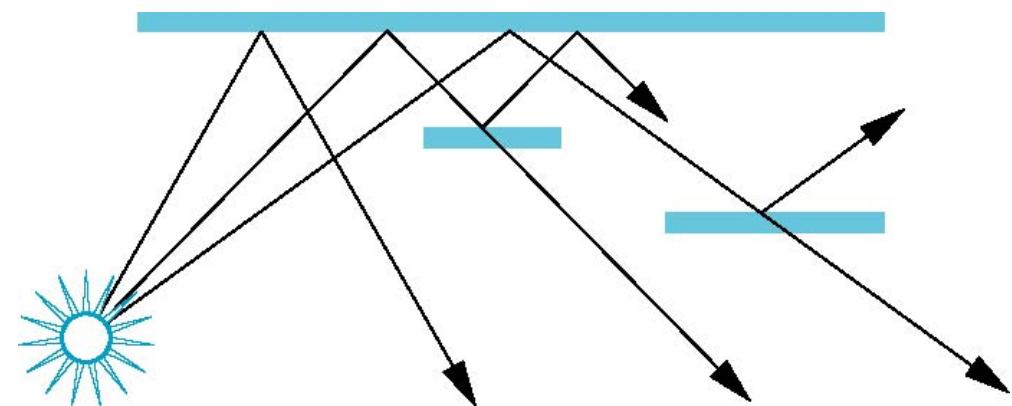
# Opacidade e transparência

- Superfícies opacas não permitem a passagem da luz (ausência de refração)
- Superfícies transparentes deixam toda a luz passar
- Superfícies translúcidas deixam passar alguma luz  
translucidez =  $I - \text{opacidade} (a)$



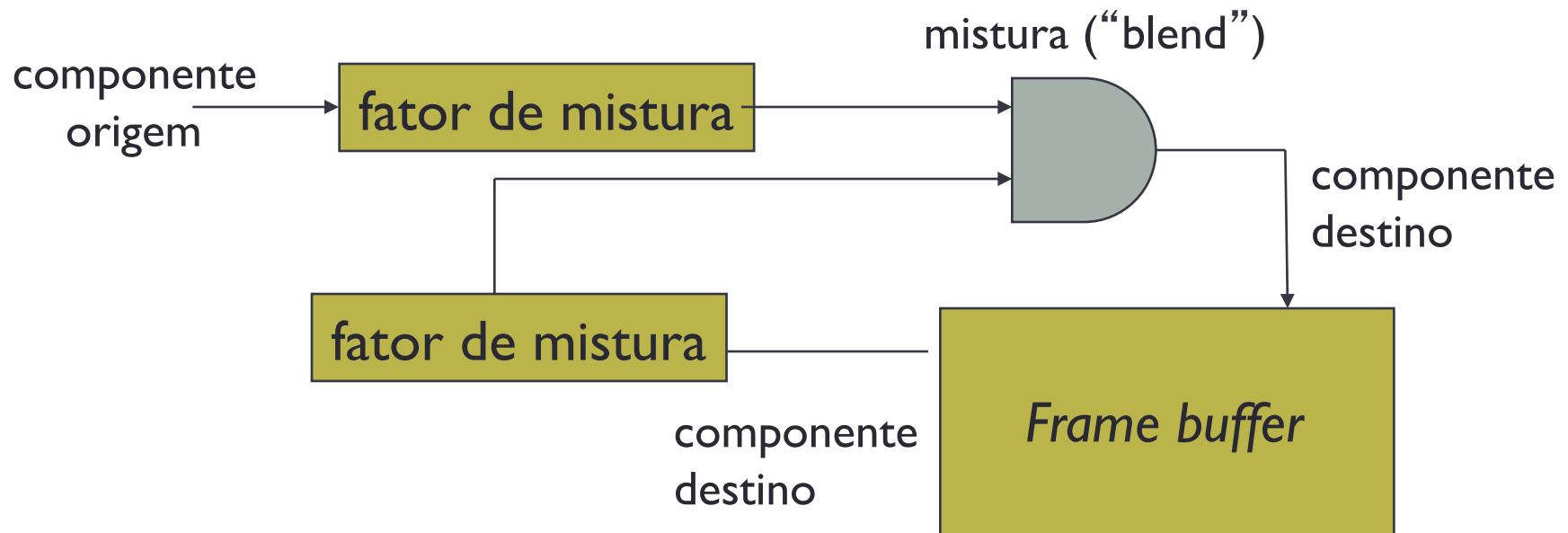
# Modelos físicos

- Modelar os efeitos de reflexão e refração de forma correta (fisicamente) é difícil devido:
  - À complexidade das interações entre luz e matéria
  - Ao uso de um renderizador baseado em pipeline: modelo local de iluminação



# Modelo de escrita

- Usar o componente A do RGBA (or RGBa) 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
  - $s = [s_r, s_g, s_b, s_a]$
  - $d = [d_r, d_g, d_b, d_a]$
- Suponha que as cores origem e destino sejam
  - $b = [R_b, G_b, B_b, A_b]$
  - $c = [R_c, G_c, B_c, A_c]$
- Combine-as da seguinte forma:
  - $c' = [s_r R_b + d_r R_c, s_g G_b + d_g G_c, s_b B_b + d_b B_c, s_a A_b + d_a A_c]$

# Composição e mistura em OpenGL

---

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

```
gl.enable(GL_BLEND);  
gl.blendFunc(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 #1

---

- Começamos com um fundo opaco ( $R_0, G_0, B_0, I.0$ )
  - Esta cor é a cor destino inicial
- Agora queremos desenhar um polígono translúcido com cor ( $R_1, G_1, B_1, A_1$ )
- Então precisaremos selecionar
  - `GL_SRC_ALPHA` e `GL_ONE_MINUS_SRC_ALPHA` como fatores origem e destino
    - $R'_1 = A_1 R_1 + (I.0 - A_1) R_0$ , etc
- Esta fórmula funciona para polígono opaco ou transparente

## Exemplo #2

---

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

```
gl.Enable(GL_BLEND);
```

```
gl.BlendFunc(GL_SRC_ALPHA, GL_ZERO);
```

```
// desenha primeira primitiva com valor de alfa ~ 1/3
```

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);    1/3 + 1/3
```

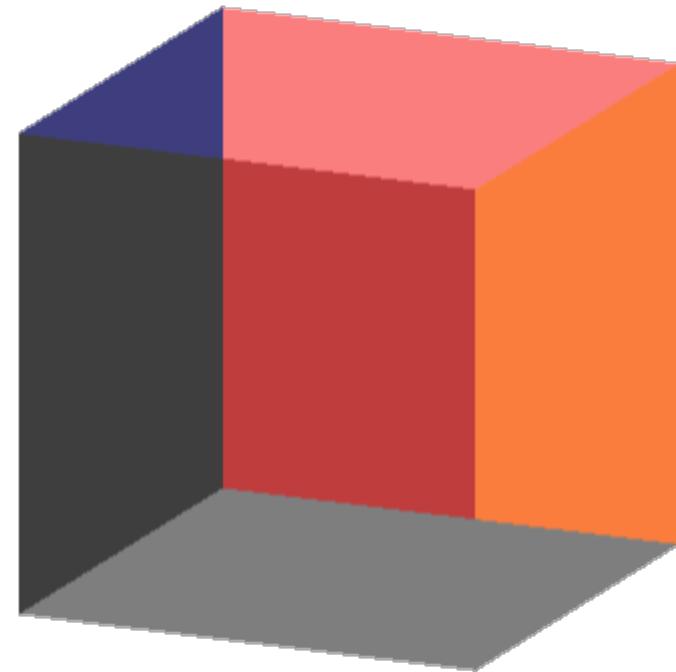
```
// desenha segunda primitiva: adiciona 1/3 da sua à cor  
// existente
```

```
...
```

# 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

---

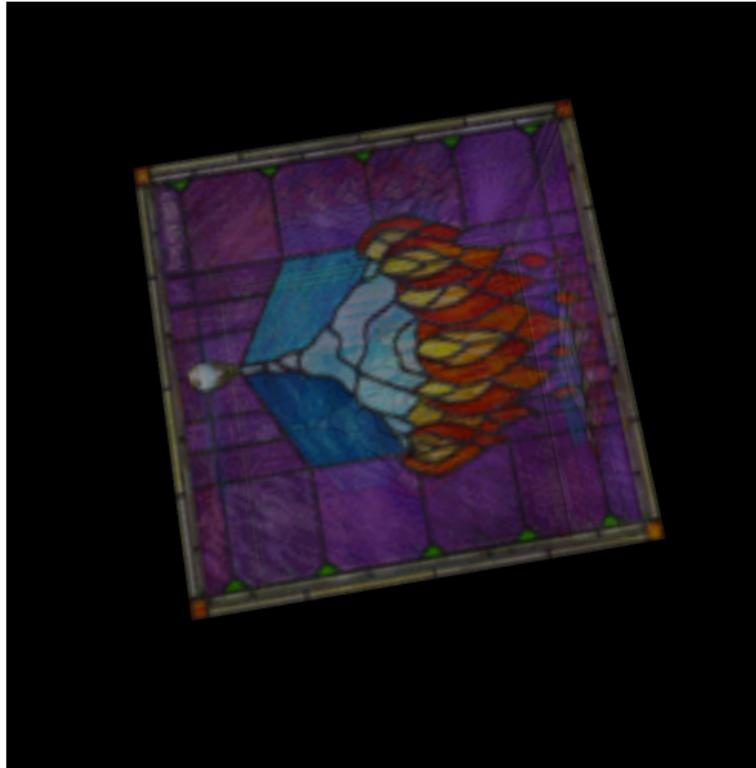
- Vamos supor que temos um grupo de polígonos opacos e alguns translúcidos
  - Desenhar todos os polígonos opacos primeiro
  - Desenhar os objetos translúcidos (do mais distante ao mais perto)
- Polígonos translúcidos não devem afetar o buffer de profundidade:
  - Renderizar objetos opacos primeiro
  - Renderizar objetos translúcidos com `gl.depthMask(GL_FALSE)`, que faz com que o depth buffer não possa ser alterado.

# Cubo transparente

---

<http://learningwebgl.com/lessons/lesson08/index.html>

[<< Back to Lesson 8](#)



Use blending

Alpha level

# Efeitos especiais: neblina

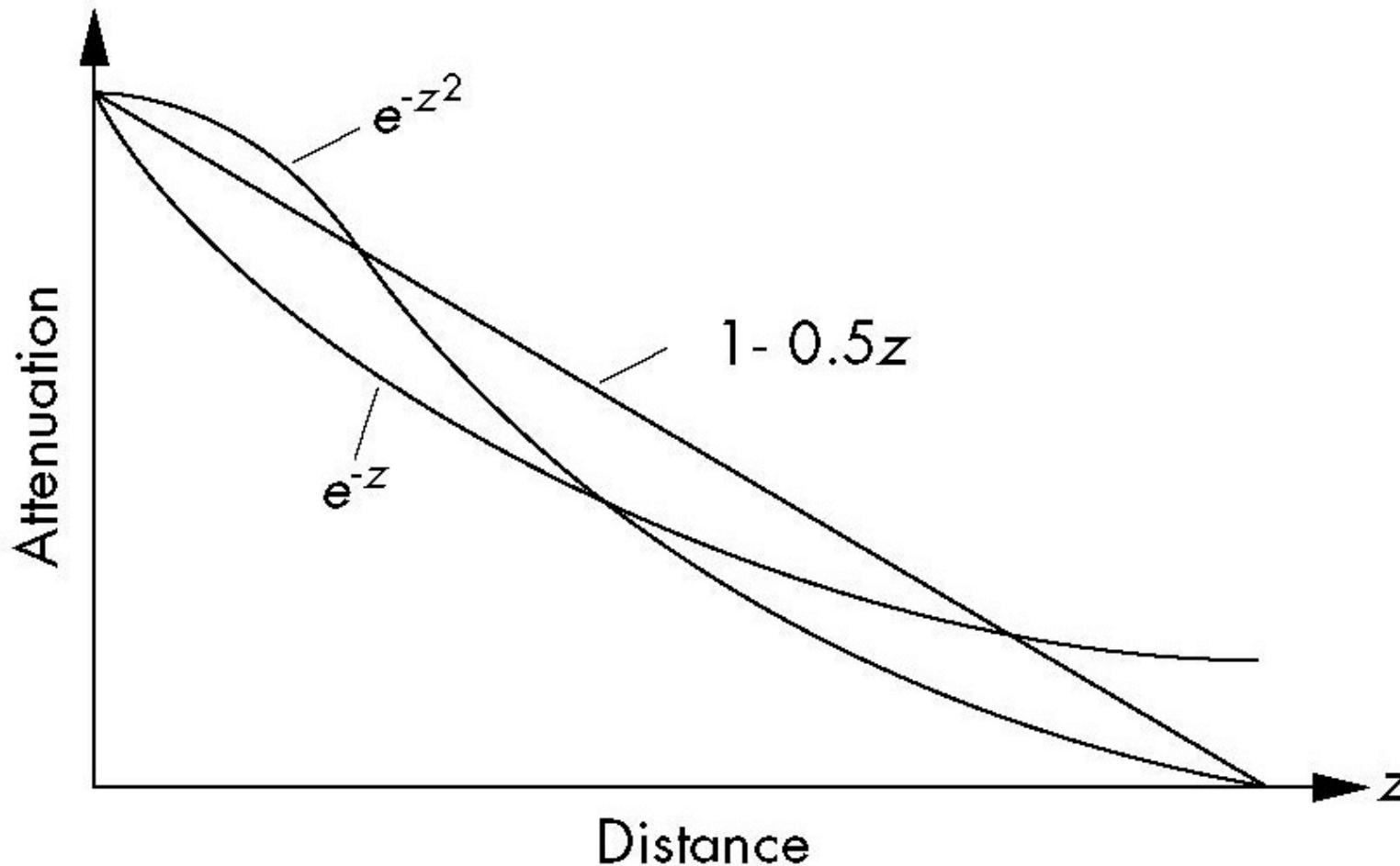
---

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

$$C'_s = fC_s + (1-f)C_f$$

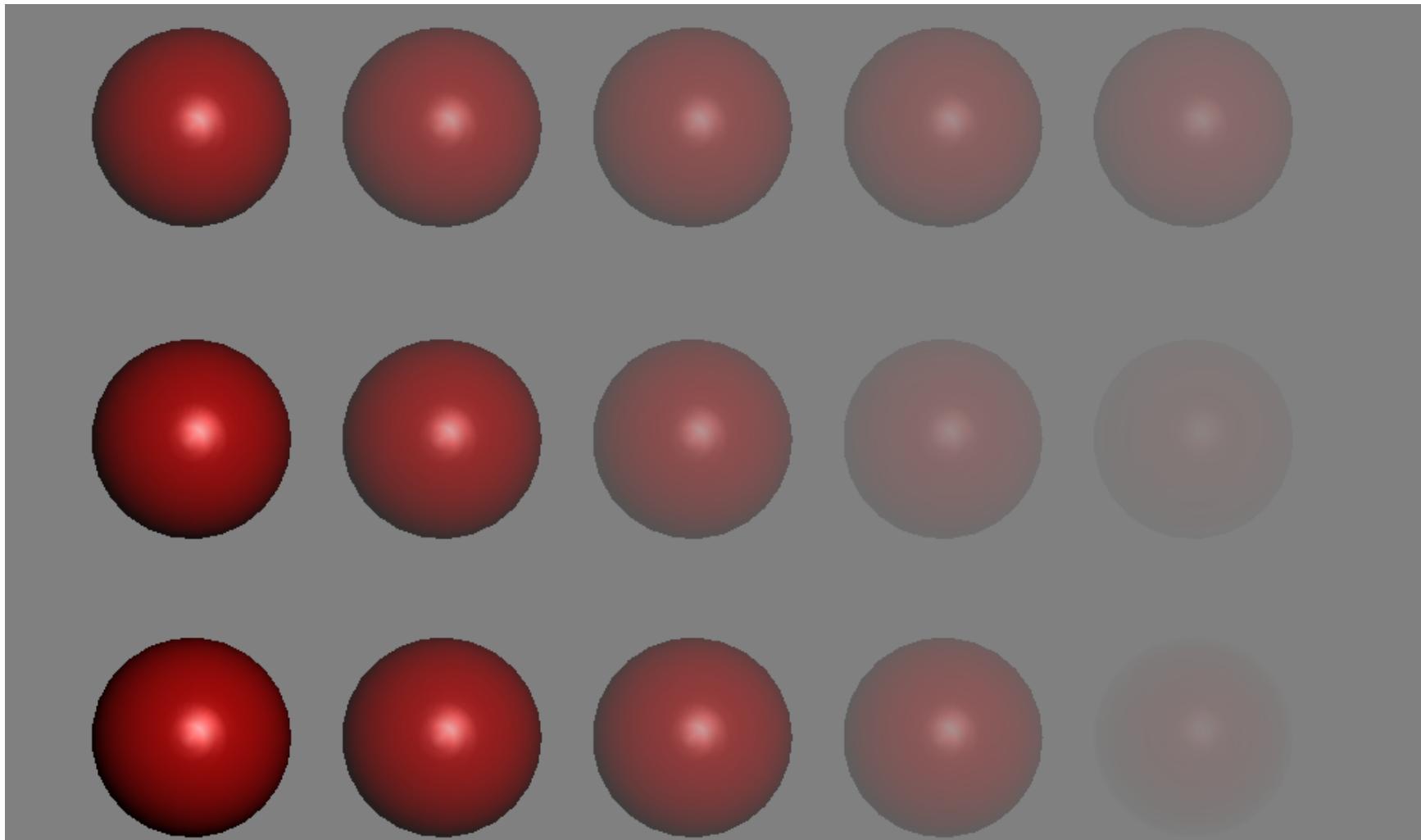
- Onde  $f$  é o fator de atenuação de neblina:
  - Exponencial
  - Gaussiano
  - Linear
  - etc

# Funções de atenuação



# Exemplos

---



# Composição com HTML5

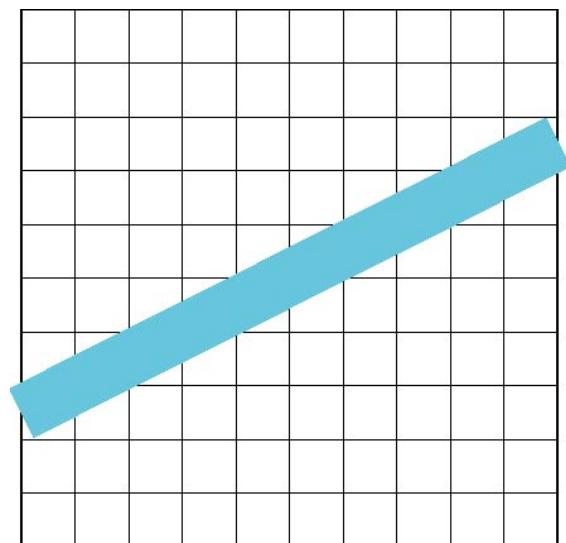
---

- No desktop OpenGL, o componente A não causa nenhum efeito se blending não estiver habilitado
- No WebGL, um A com qualquer valor diferente de 1.0 causa um efeito porque o WebGL utiliza o Canvas do HTML5
  - $A = 0.5$  reduzirá os valores RGB pela  $\frac{1}{2}$  quando um pixel for pintado
- Isto faz com que o visual de aplicações possam ser compostos com a parte gráfica.

# Aliasing

---

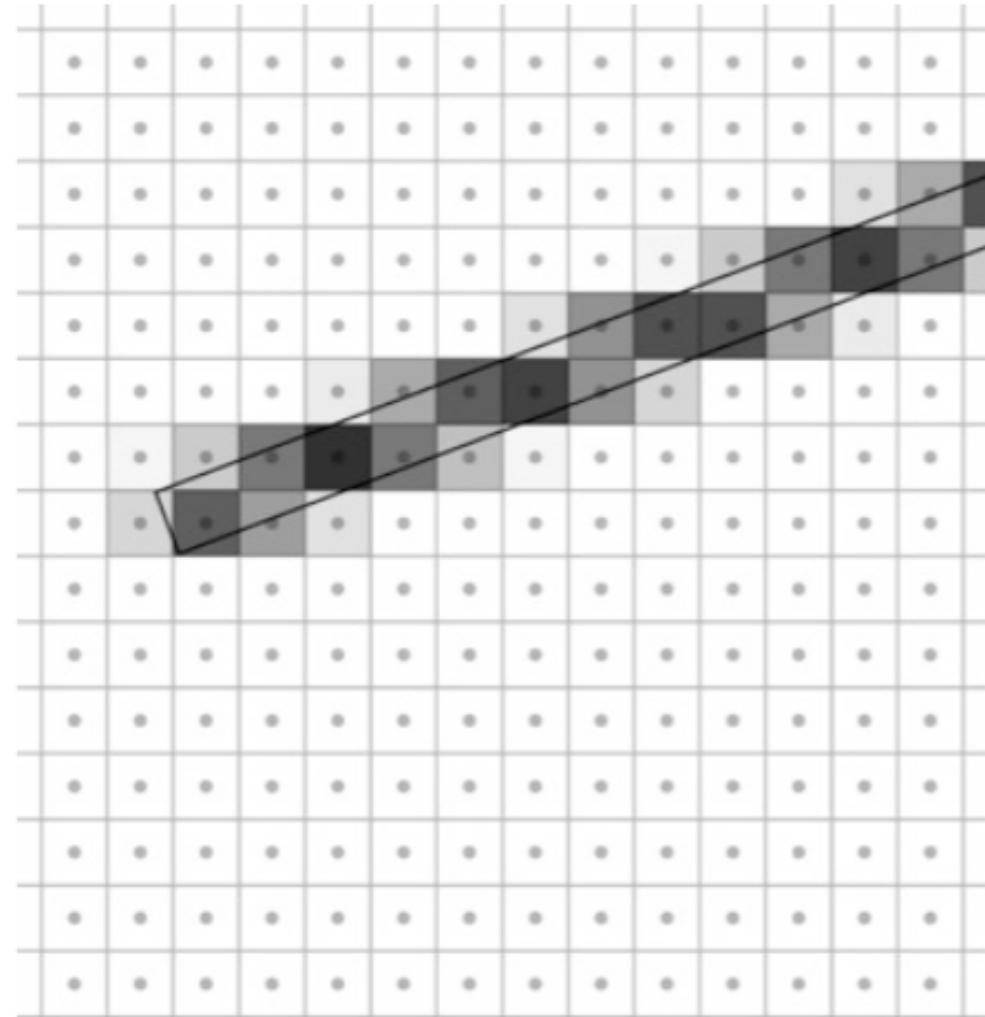
- Todos os segmentos de reta, exceto os verticais e horizontais, cobrem parcialmente os pixels
- Algoritmos simples de rasterização produzem efeitos indesejáveis:
  - serrilhamento (“aliasing”);
- Também é problemático na renderização de polígonos



# Aliasing

---

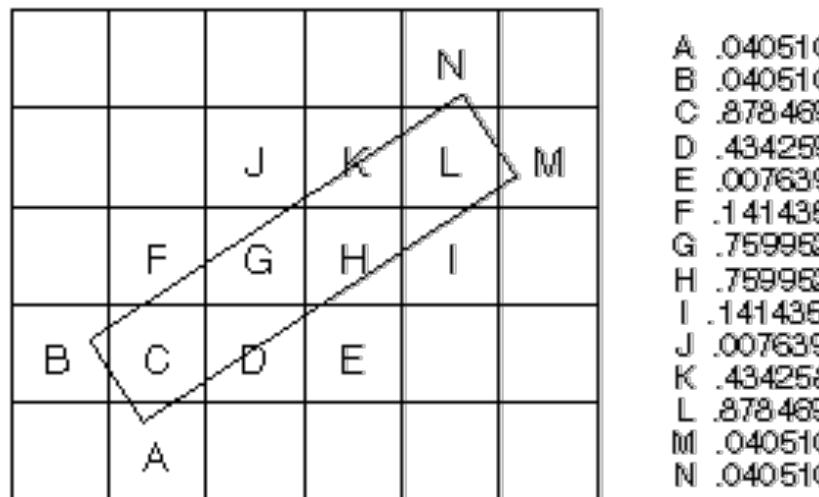
- Basic idea: replace “is the image black at the pixel center?” with “how much is pixel covered by black?”
- Replace yes/no question with quantitative question.



# Antiserrilhamento

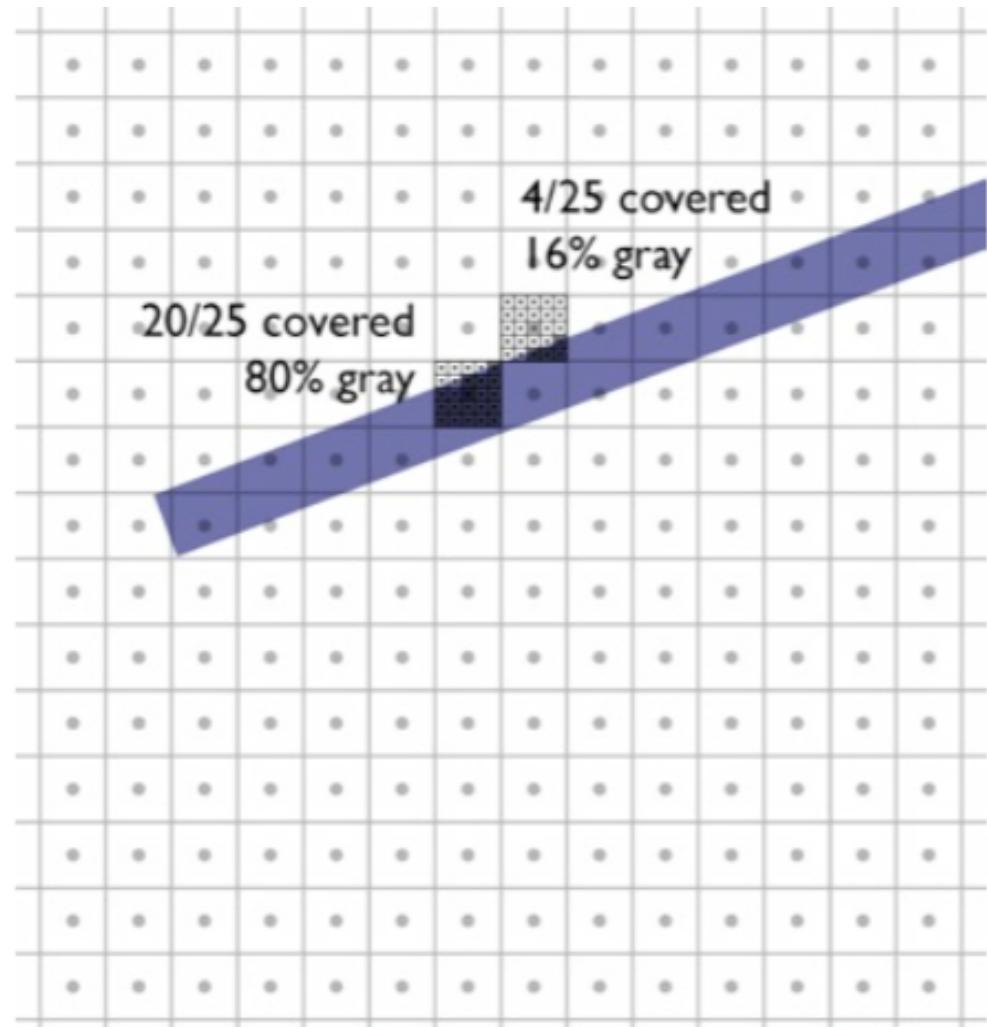
---

- 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



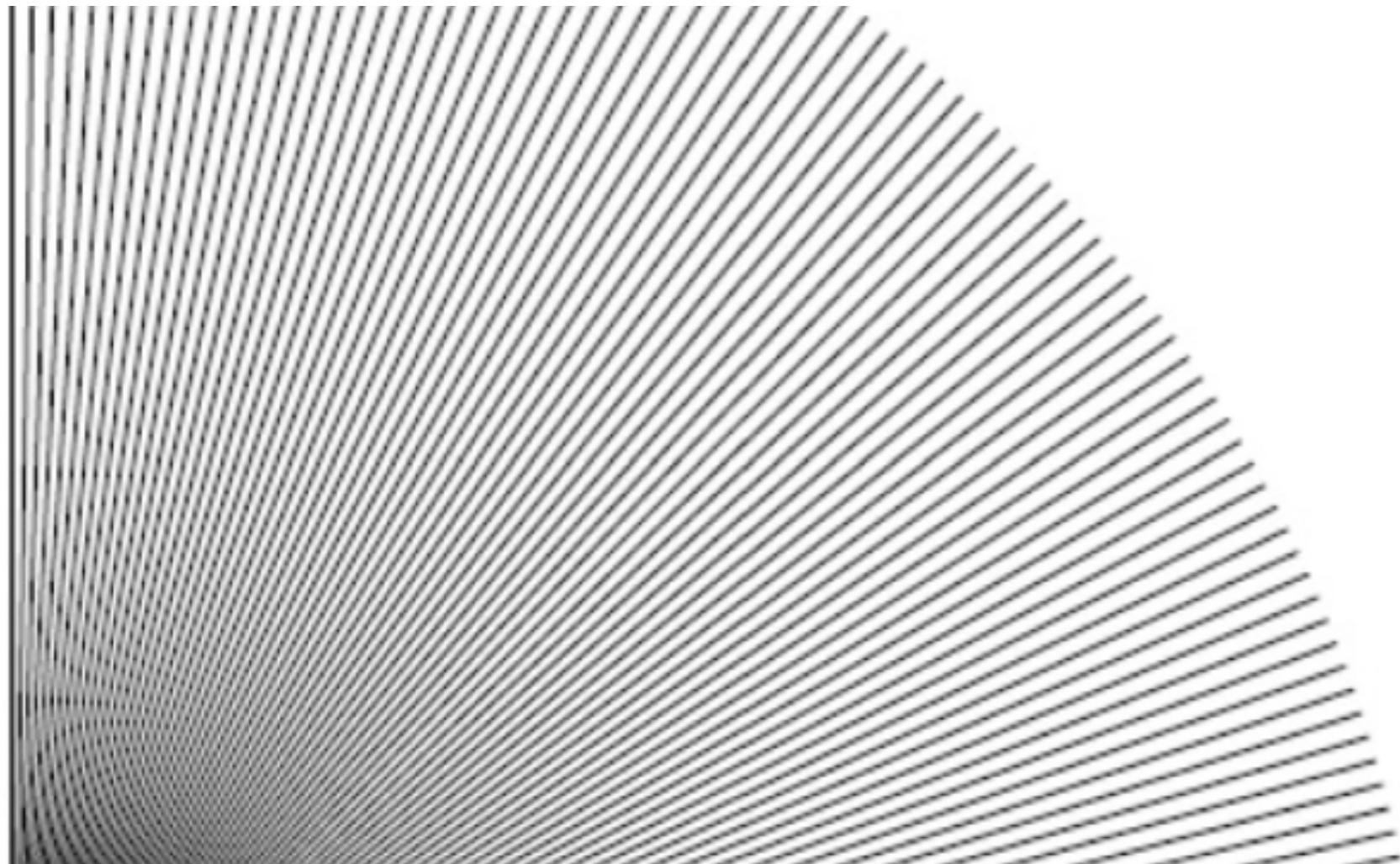
# Filtragem por área

- Compute coverage fraction by counting subpixels
- Simple, accurate
- But slow



# Exemplo de antiserrilhamento

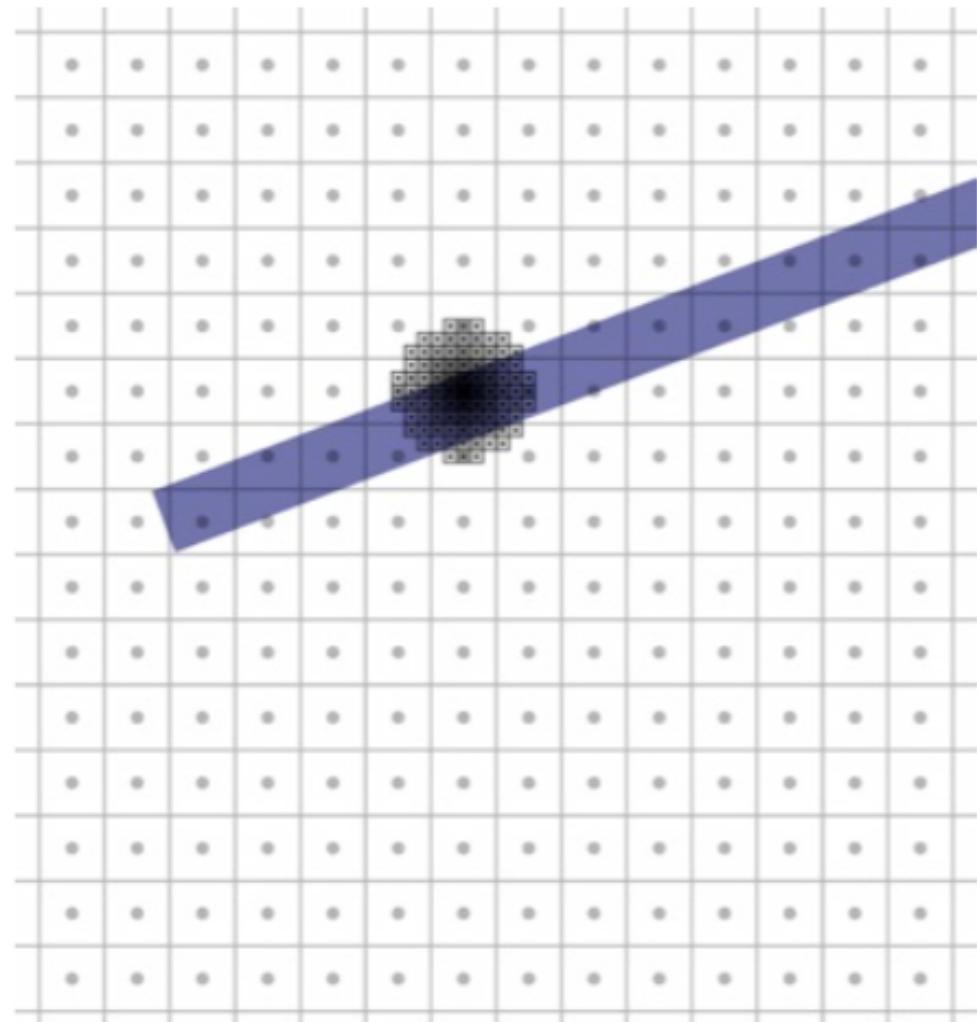
---



# Filtragem por funções radiais

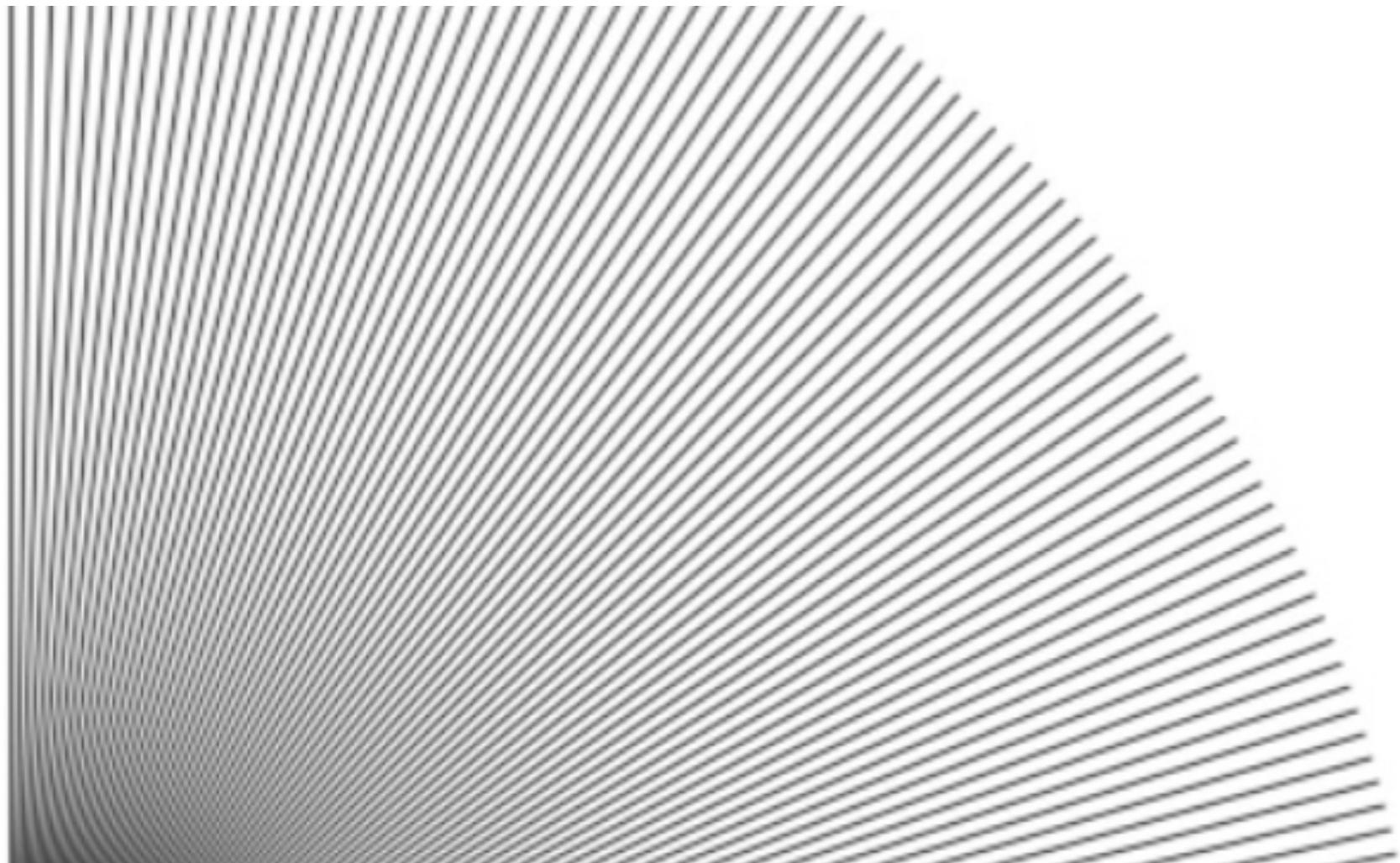
---

- Compute filtering integral by summing filter values for covered subpixels
- Simple, accurate
- But really slow



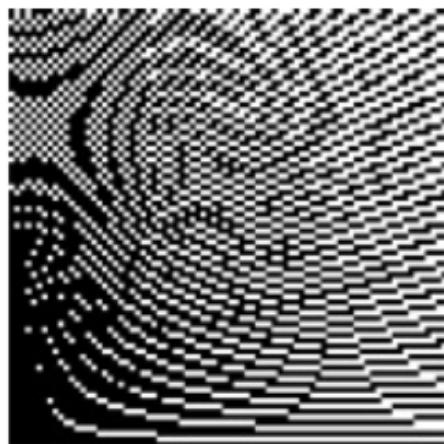
# Filtragem com Gaussiana

---

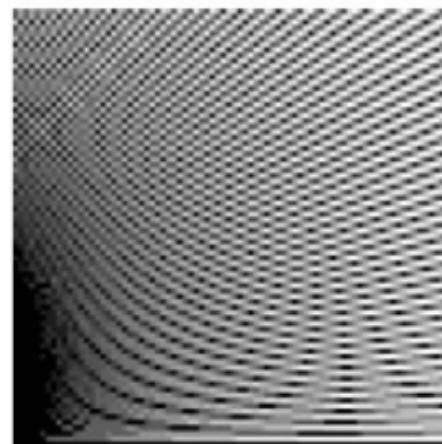


# Comparação

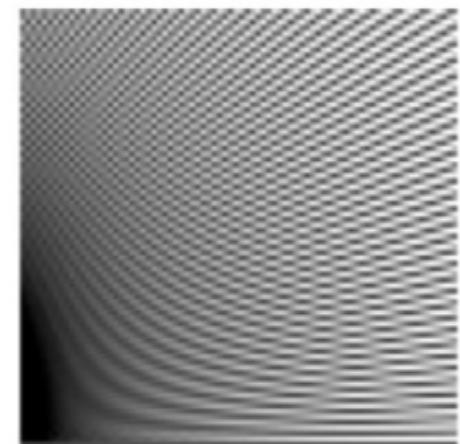
---



Point sampling



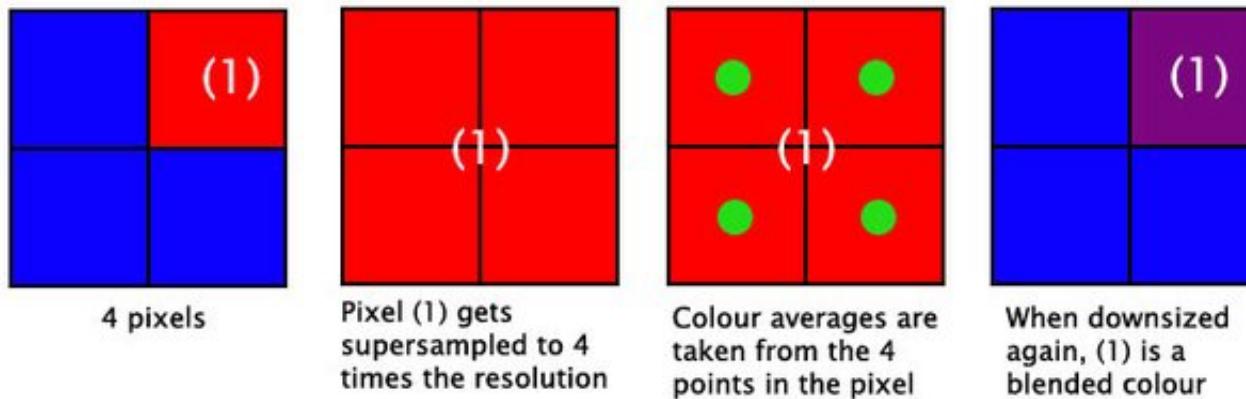
Box filtering



Gaussian filtering

# *Supersampling e multisampling*

- Full-Scene Antialiasing (FSAA)



- Multisampling Antialiasing (MSAA)
  - Utiliza deslocamentos do frame buffer para calcular o valor final do pixel

# Exemplo

---



# Antiserrilhamento em OpenGL

---

- Não suportado ainda pelo WebGL
- Podemos habilitar a técnica de antiserrilhamento separadamente para pontos, linhas e polígonos

```
gl.enable(GL_POINT_SMOOTH);  
gl.enable(GL_LINE_SMOOTH);  
gl.enable(GL_POLYGON_SMOOTH);
```

```
gl.enable(GL_BLEND);  
gl.blendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

- Note porém, que o hardware na maioria das vezes, realiza antiserrilhamento

# Multirenderização

---

- Composição de várias imagens
- Filtragem (convolução)
  - Adicionar versões transladadas e escaladas de uma imagem
- Antialiasing global de cenas
  - Transladar primitivas utilizando um pequeno fator em cada renderização
- "Depth of Field"
  - Mover o observador um pouco a cada renderização mantendo um plano imóvel
- Efeitos de movimentação ("Motion Blur")

# Imagens e *Fragment shaders*

---

- Desejamos enviar um retângulo (dois triângulos) para o *vertex shader* e renderizá-lo com uma textura  $n \times m$
- Suponha que o canvas possui também tamanho  $n \times m$
- Existe um correspondência 1-1 entre cada texel e cada fragmento
- Portanto, podemos considerar operações com fragmentos como operações no mapa de textura.

# GPGPU

---

- Notamos assim que o único propósito de utilizar geometria é iniciar as operações na imagem no *fragment shader*
- Consequentemente, podemos considerar tais operações mais como operações em matrizes do que operações gráficas
- Esta observação nos remete à área de General Purpose Computing with a GPU (GPGPU).

# Aplicações GPGPU

---

- Alguns exemplos:
  - Adição e multiplicação de matrizes
  - Solução de sistemas de equações
  - Transformadas (e.g. Fourier)
- Utiliza a rapidez e o paralelismo da GPU
- Como coletar os resultados?
  - Frame buffers de ponto flutuante
  - OpenCL (WebCL)
  - Compute shaders

# Acesso à múltiplos *texels*

---

- Se tivermos uma textura  $1024 \times 1024$  no objeto de textura chamado de “image”, então:  
**sampler2D(image, vec2(x, y))** retornará o valor da textura em  $(x, y)$   
**sampler2D(image, vec2(x+1.0/1024.0), y)** retornará o valor do texel à direita de  $(x, y)$
- Podemos utilizar qualquer combinação de texels na vizinhança de  $(x, y)$  no *fragment shader*

# Realce de contraste

---

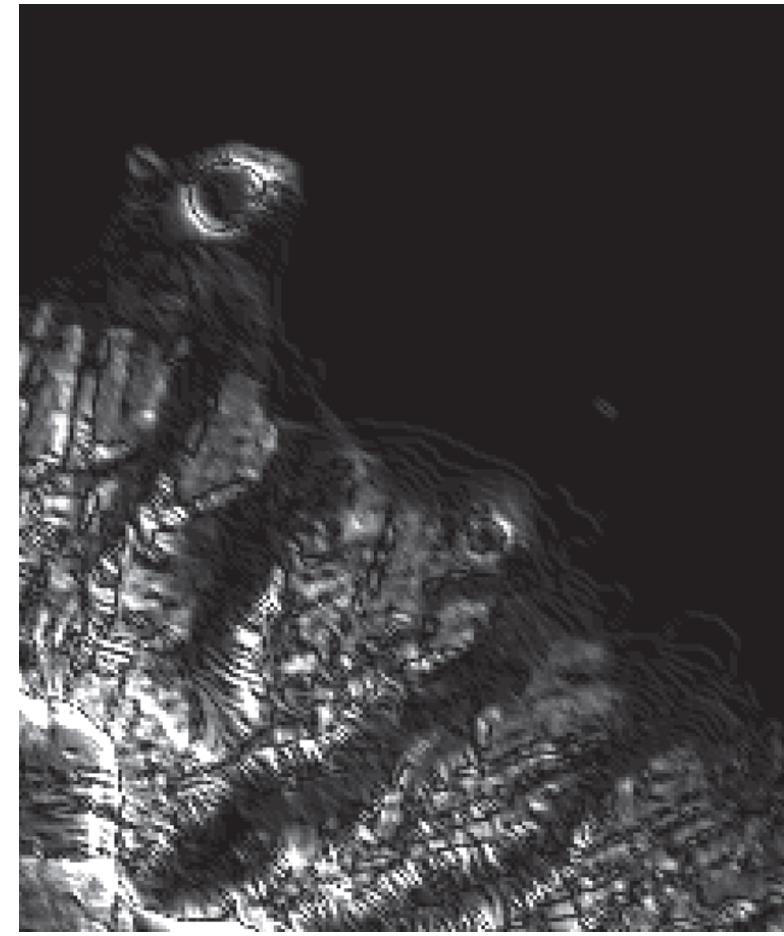
```
precision mediump float;
varying vec2 fTexCoord;
uniform sampler2D texture;
void main()
{
    float d = 1.0/256.0; //spacing between texels
    float x = fTexCoord.x;
    float y = fTexCoord.y;
    gl_FragColor =
        10.0*abs( texture2D( texture, vec2(x+d, y))
        -texture2D( texture, vec2(x-d, y)))
        +10.0*abs( texture2D( texture, vec2(x, y+d))
        -texture2D( texture, vec2(x, y-d)));
    gl_FragColor.w = 1.0;
}
```

# Exemplo de realce de contraste

---



original



enhanced

# Detecção de bordas

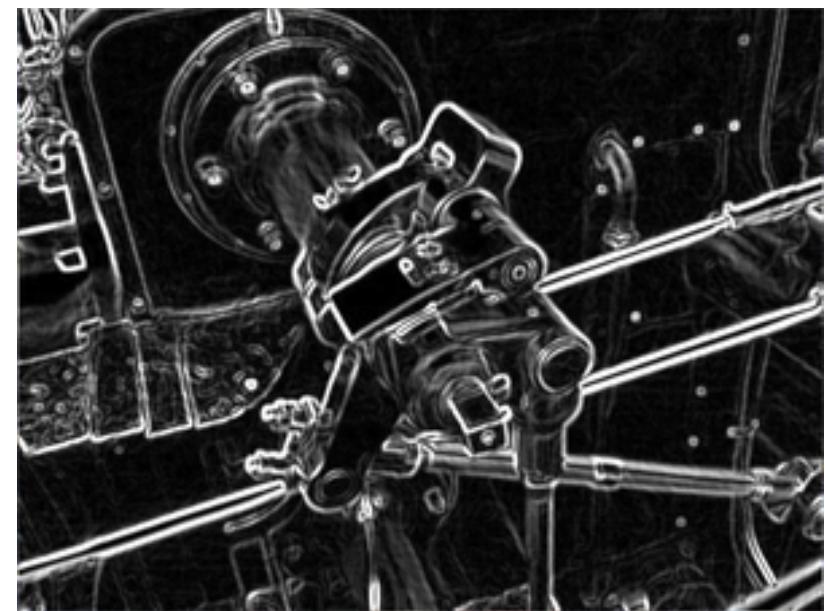
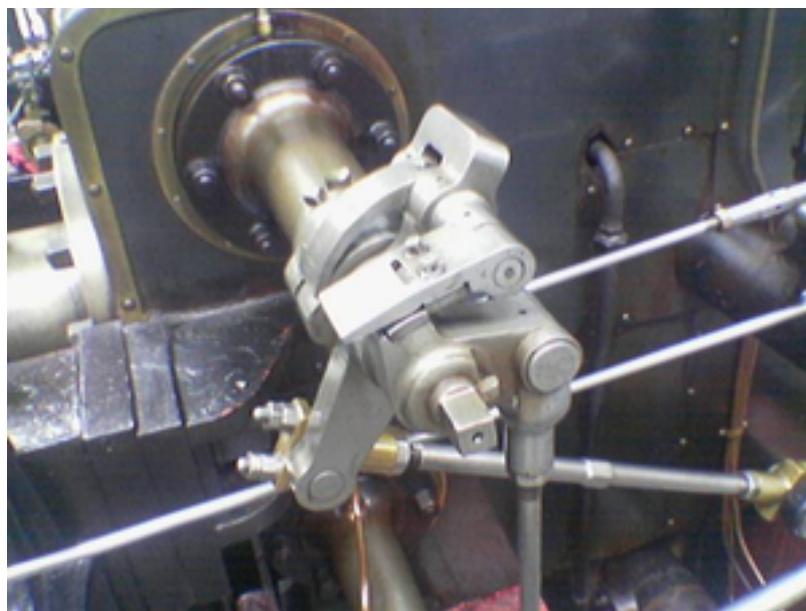
---

- Operador Sobel
  - Acha gradiente aproximado em cada ponto
  - Calcula aproximações suaves para as diferenças finitas em x e y separadamente
  - Mostra magnitude do gradiente aproximado
  - De simples implementação através de *fragment shaders*

$$\mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A}$$

# Operador Sobel

---



# Operador Sobel

---

```
vec4 gx = 2.0*texture2D( texture, vec2(x+d, y))
    + texture2D( texture, vec2(x+d, y+d))
    + texture2D( texture, vec2(x+d, y-d))
    - 2.0*texture2D( texture, vec2(x-d, y))
    - texture2D( texture, vec2(x-d, y+d))
    - texture2D( texture, vec2(x-d, y-d));

vec4 gy = 2.0*texture2D( texture, vec2(x, y+d))
    + texture2D( texture, vec2(x+d, y+d))
    + texture2D( texture, vec2(x-d, y+d))
    - 2.0*texture2D( texture, vec2(x, y-d))
    - texture2D( texture, vec2(x+d, y-d))
    - texture2D( texture, vec2(x-d, y-d));

gl_FragColor = vec4(sqrt(gx*gx + gy*gy), 1.0);
gl_FragColor.w = 1.0;
```

# Utilização de múltiplas texturas

---

- Exemplo: adição de matrizes
- Cria-se dois *samplers*, *texture1* e *texture2*, que contém os dados
- E assim no fragment shader:

```
gl_FragColor =  
    sampler2D(texture1, vec2(x,y)) +  
    sampler2D(texture2, vec2(x,y));
```

# Tarefa de casa

---

- Leitura livro-texto
  - Shirley and Marschner. Fundamentals of Computer Graphics, CRC Press, 3<sup>rd</sup> Ed. 2010
  - Capítulo 11