Criando Efeitos Fotorealistas e Não-Fotoreaslitas para Jogos

Bruno P. Evangelista **(UFMG)**
Alessandro R. Silva **(UFMG)**

Revisão 2

Introdução

- **Com a rápida evolução das GPUs, os jogos estão cada vez mais próximos do mundo real !!!**
  - **É possível diferenciar uma cena de jogo de uma fotografia?**

Introdução

Introdução

- Utilizando as GPUs modernas podemos criar inúmeros efeitos para a renderização das cenas
  - Esses efeitos podem ser usados para criar ambientes extremamente realísticos
  - Ou mesmo ambientes não realístico!
- Nesta palestra apresentaremos alguns dos principais efeitos utilizados pelos jogos atuais

Agenda

- Pipeline de Renderização e Shaders (Revisão)
- Linguagens de shaders
- Efeitos
  - Iluminação por pixel
  - Reflexão/Refração de ambientes
  - Texturização/Multi-Texturização
  - Geração de texturas procedurais
  - Simulação de superfícies detalhadas
- Pós-Processamento
  - Radiometria
  - Bloom
  - Cartoon Rendering

Pipeline de Renderização

- Durante muitos anos as APIs gráficas como o DirectX ou o OpenGL utilizavam um fluxo fixo de renderização
  - Os processos executados em cada estágio do pipeline de renderização eram pré-programados em hardware e não podiam ser modificados.
    - Exemplo: Transformação, iluminação, etc...
  - Era possível apenas alterar um pequeno número de parâmetros no pipeline
- Resultado: Jogos com gráficos semelhantes!!!

Pipeline de Renderização

- Enquanto isso...
  - A industria do cinema já possuía ferramentas capazes de programar a renderização das cenas
- RenderMan
  - Especificação de uma linguagem de shaders, criada pela Pixar em 1988
  - Atualmente existem várias implementações open-source e comerciais
- No entanto, essas ferramentas eram utilizadas apenas para renderização off-line =(

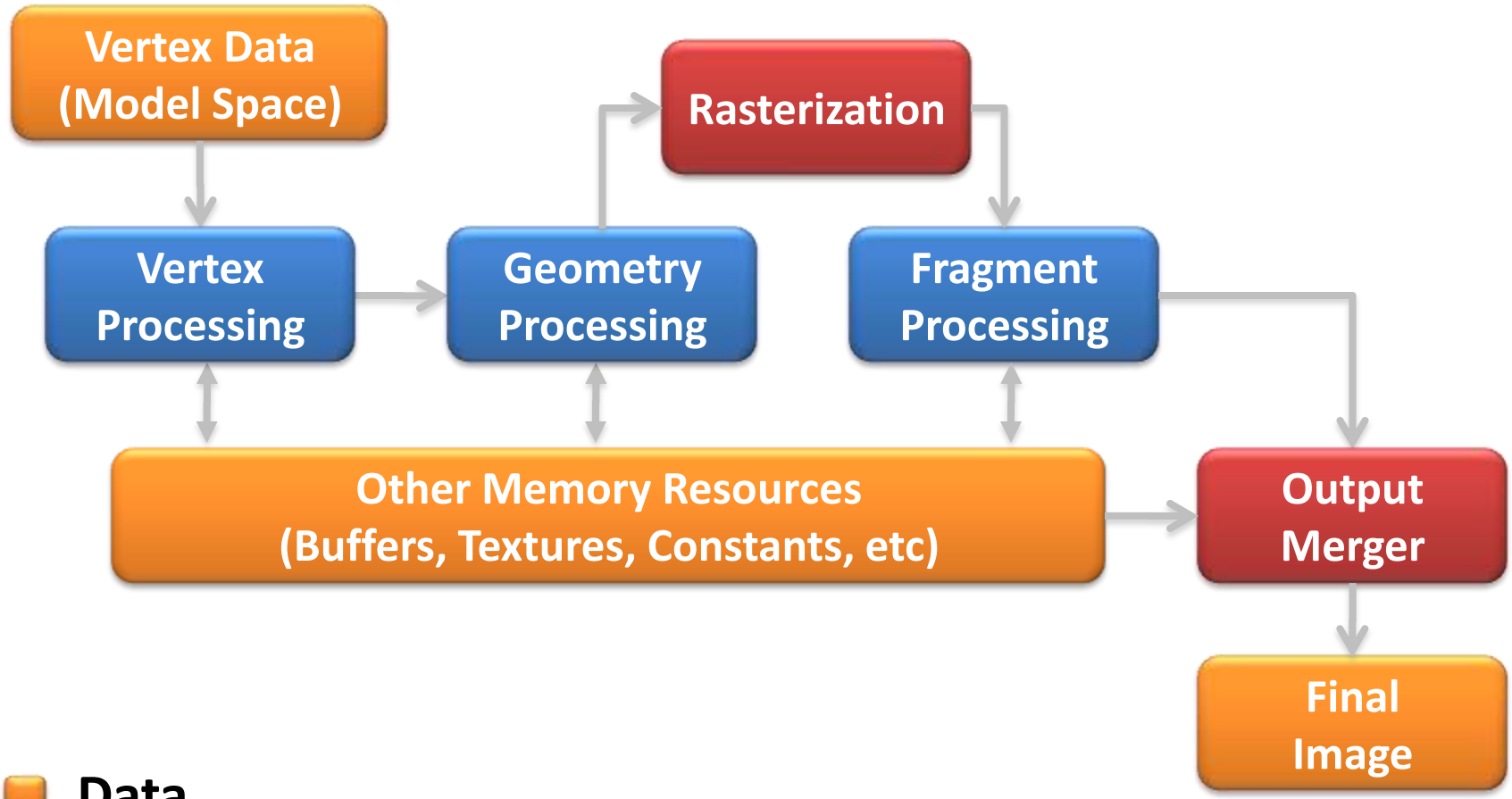
Shaders

- Pequenos programas executados na GPU
- Permitem programar alguns estágios do pipeline de renderização
- Novas possibilidades
  - Modelos de iluminação reais
  - Renderização de superfícies com muitos detalhes
  - Efeitos de pós-processamento
- Podemos utilizar tudo isso em tempo real !!!

Shaders

- **Shaders que rodam em diferentes estágios da GPU recebem diferentes nomes**
  - **Vertex Shader – Estágio de processamento de vértices**
  - **Pixel Shader – Estágio de processamento de pixels**
  - **Geometry Shader – Estágio de Processamento de geometrias**

Pipeline de Renderização



A Evolução dos Shaders

Vídeos e Shaders

Linguagens de Shaders

- **Renderização Offline**
  - **RenderMan – PRMan Pixar/Outras implementações**
  - **Gelato – nVidia**
- **Renderização em tempo real**
  - **HLSL (High Level Shading Language) – Microsoft**
    Utilizada no DirectX e XNA
  - **GLSL (OpenGL Shading Language) – 3D Labs**
    Utilizado no OpenGL
  - **Cg (C for Graphics) – nVidia**
    Pode ser utilizada no DirectX e OpenGL

HLSL

- Possui um pequeno número de funções
  - Operações matemáticas, acessos a textura e controle de fluxo
- Possui alguns tipos de dados padrões do C++, além de vetores e matrizes
  - bool, int, half, float, double
  - vetores (floatN, boolN, ...), matrizes (floatNxM, ...)
  - texture, sampler, struct
- O código escrito no shader é similar a uma equação matemática

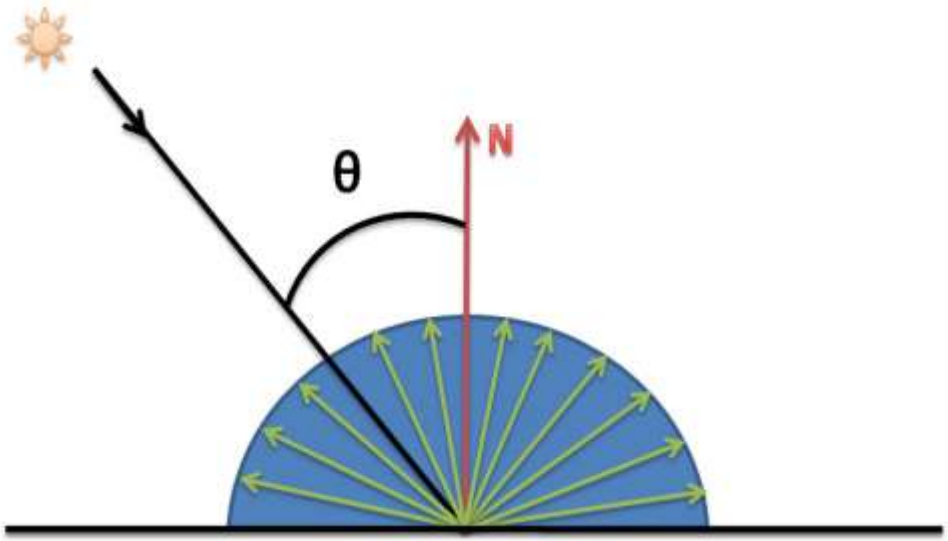
Shader: Iluminação por Pixel

- O algoritmo de Blinn-Phong é um dos mais utilizados nas APIs gráficas para iluminação
  - Modelo empírico
  - A luz é representada por três componentes: ambiente, difuso e especular
- Componentes de luz
  - Ambiente: Luz espalhada uniformemente pela cena
  - Difuso:Luz que interage somente com as superfícies
  - Especular: Luz refletida perfeitamente pelas superfícies

$$I_{total} = I_{ambient} + \sum_{LIGHTS} I_{diffuse} + I_{specular}$$

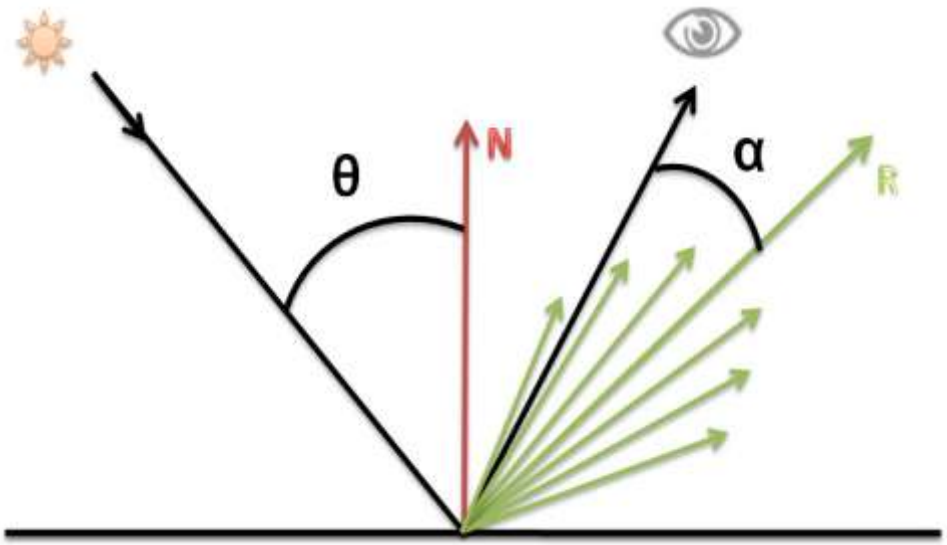
Componentes Difuso e Especular

- **Difuso: Luz que incide na superfície e reflete em todas as direções com mesma intensidade (isotrópico)**
- **Calculada através da lei de Lambert**



$$I_{diffuse} = K_d L_d (N.L)$$

- **Especular: Luz que incide na superfície e reflete com mesmo ângulo de incidência (espelho)**
- **Calculada através da lei de Snell, intensidade dependente da posição do observador**



$$I_{specular} = K_s I_s (R.V)^{shininess}$$

Implementação – Shader

$$I_{total} = I_{ambient} + \sum_{LIGHTS} I_{diffuse} + I_{specular}$$

$$I_{diffuse} = K_d L_d (N.L)$$

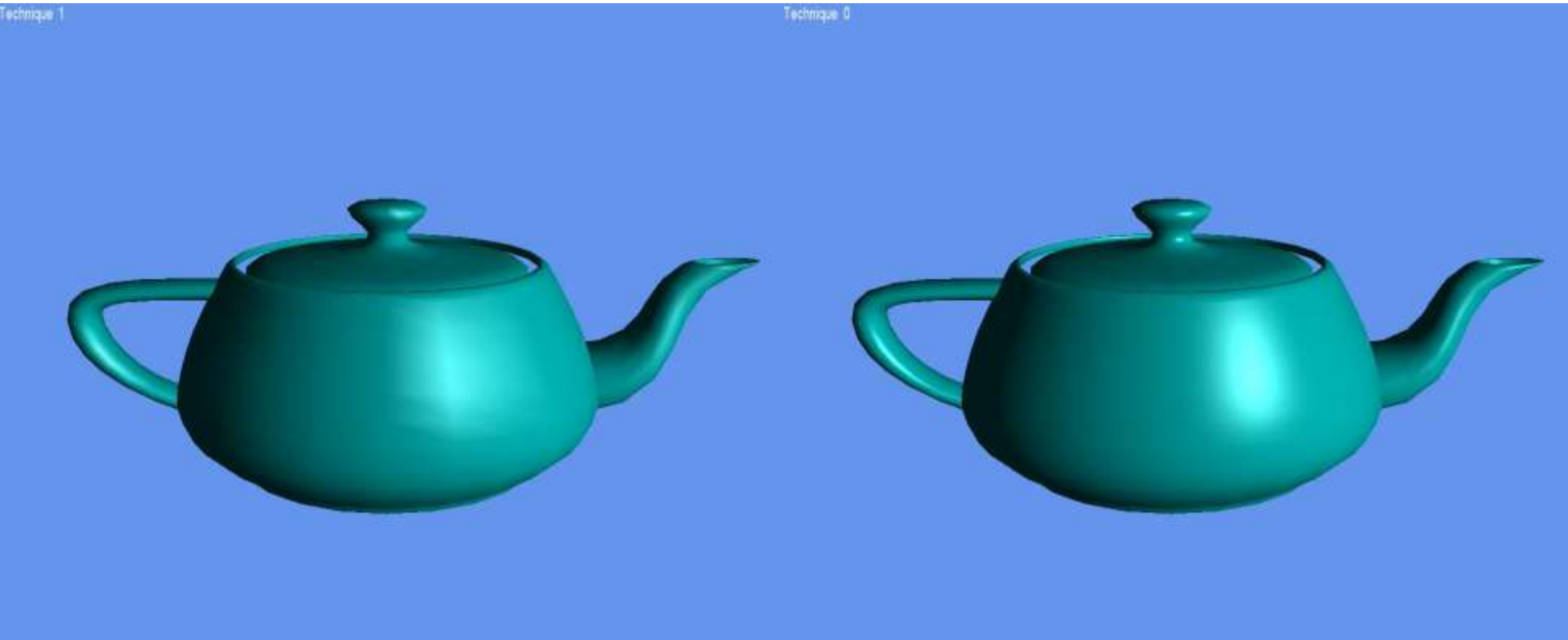
$$I_{specular} = K_s I_s (R.V)^{shininess}$$

```
void phongLighting(in float3 normal, in float3 lightVec,
  in float3 eyeVec, in float3 lightColor,
  out float3 diffuseColor, out float3 specularColor)
{
  float diffuseInt = saturate(dot(normal, lightVec));
  diffuseColor = diffuseInt * materialKd * lightColor;

  float3 reflectVec = reflect(-lightVec, normal);
  float specularInt = saturate(dot(reflectVec, eyeVec));
  specularInt = pow(specularInt, materialShininess);
  specularColor = specularInt * materialKs * lightColor;
}
```

Iluminação por Pixel

- **Demo – XNA**



Atenuação

- Podemos definir um raio de alcance para luzes pontuais e holofote, e utilizar isto para atenuar a intensidade de luz
  - A atenuação determina o quão rápido a luz perde intensidade
  - A atenuação pode ser constante, linear, quadrática, etc...
  - O resultado será uma luz suave sobre a superfície

Implementação – Shader

Função de atenuação de luz quadrática

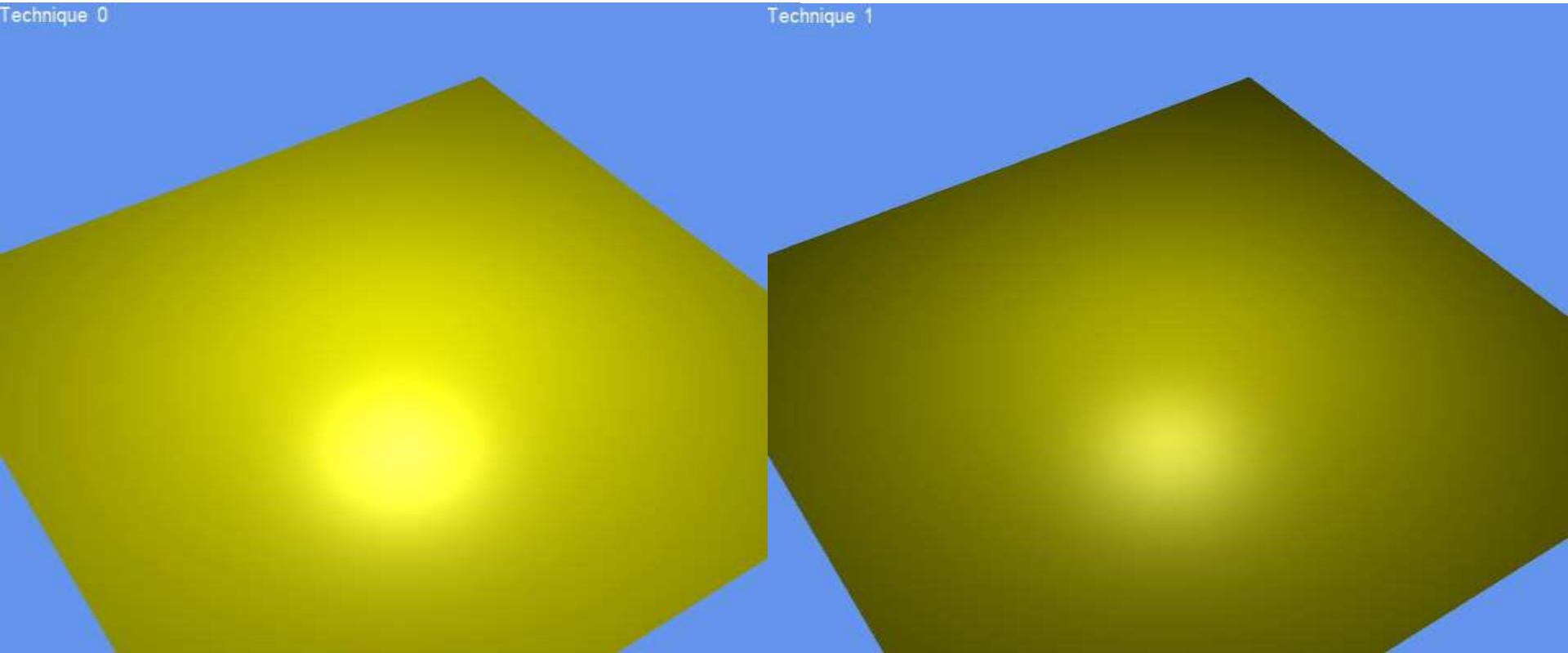
$$L_{intensity} = Max\left(\frac{L_{range} - distance}{L_{range}}, 0\right)^2$$

Implementação no Pixel Shader

```
float lightDistance = length(IN.lightVec);
float attenuation = (lightRadius - lightDistance) / lightRadius;
attenuation = pow(saturate(attenuation), 2);
```

Atenuação

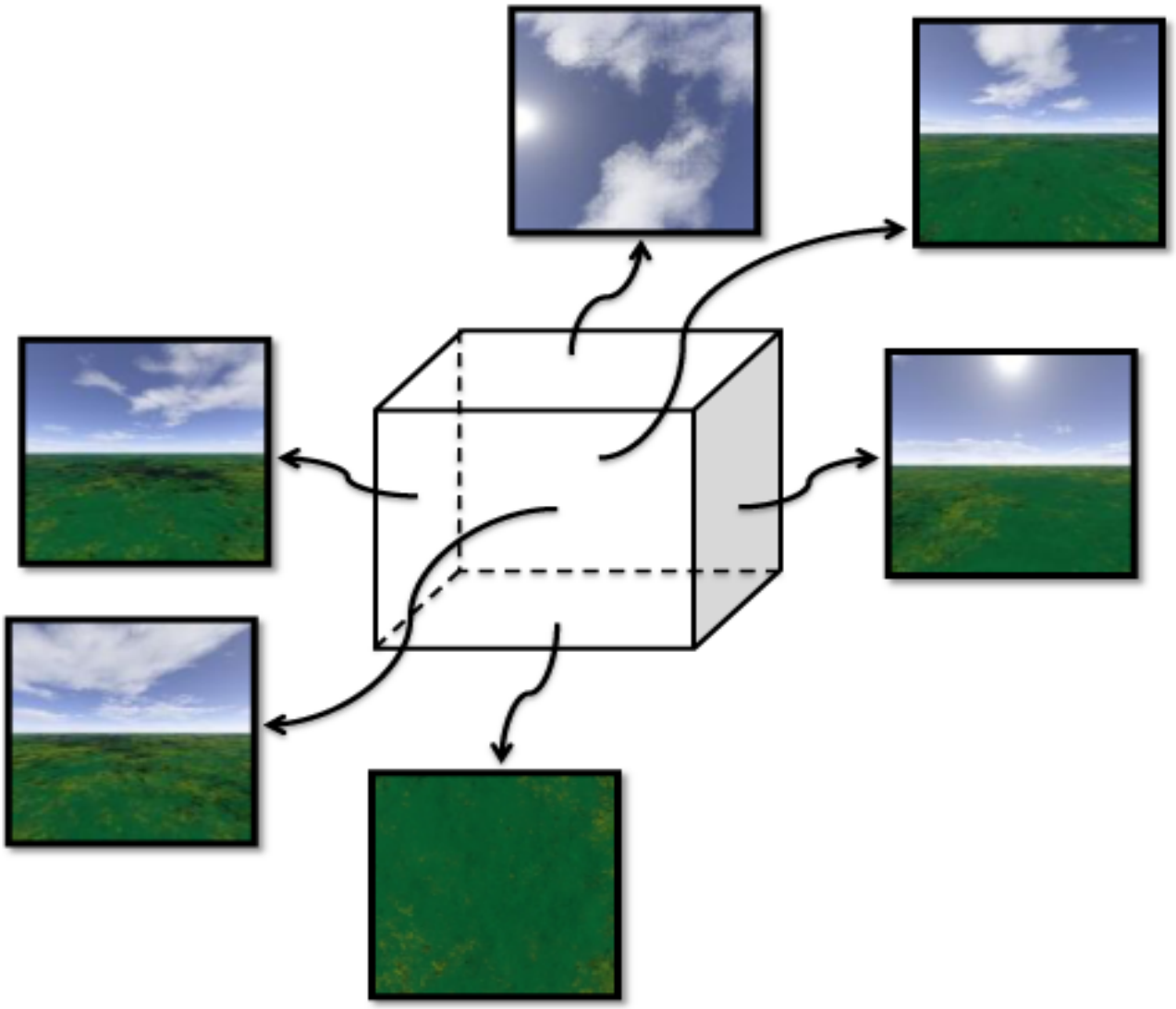
- Demo – XNA Multiple Lights



Reflexão/Refração de Ambientes

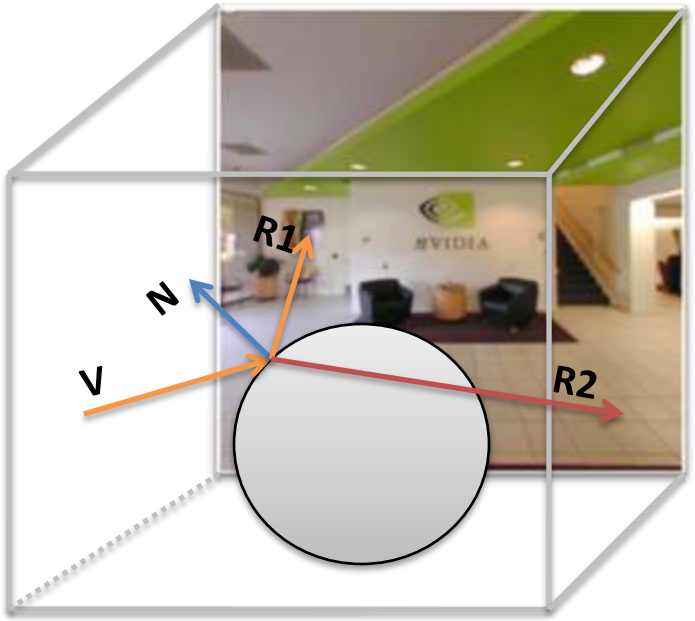
- Geralmente implementadas através de mapas pré-computados ou mapas dinâmicos de reflexão e refração
  - O ambiente ao redor do objeto é renderizado para uma textura
  - Estas texturas são mapeadas em um sólido que cobre toda a cena (geralmente um cubo ou esfera)
  - Os vetores de reflexão e refração são utilizados para indexar esta textura
- É possível estender a técnica para ambientes dinâmicos
  - Dynamic Cube Mapping

Cube Mapping



Cube Mapping

- O Cube Map é acessado como uma textura 3D
  - Pode-se utilizar os vetores de reflexão e refração para acessar a textura



Implementação - Shader

```
float4 PS_CubeMapReflect(vertexOutput IN) : color0 {

    float3 n = normalize(IN.normal);
    float3 e = normalize(IN.eyeVec);

    // Reflection
    float3 reflectCoord = reflect(-e, n);
    float3 reflectColor = texCUBE(cubemap_sampler, reflectCoord);
    // Refraction
    float3 refractCoord = refract(-e, n, 0.87);
    float3 refractColor = texCUBE(cubemap_sampler, refractCoord);

    return float4(reflectColor * 0.9f + 0.1f * refractColor, 1.0f);
}
```

Cor Final = 90% de reflexão + 10% de refração

Podemos utilizar o termo de Fresnel

Cube Mapping

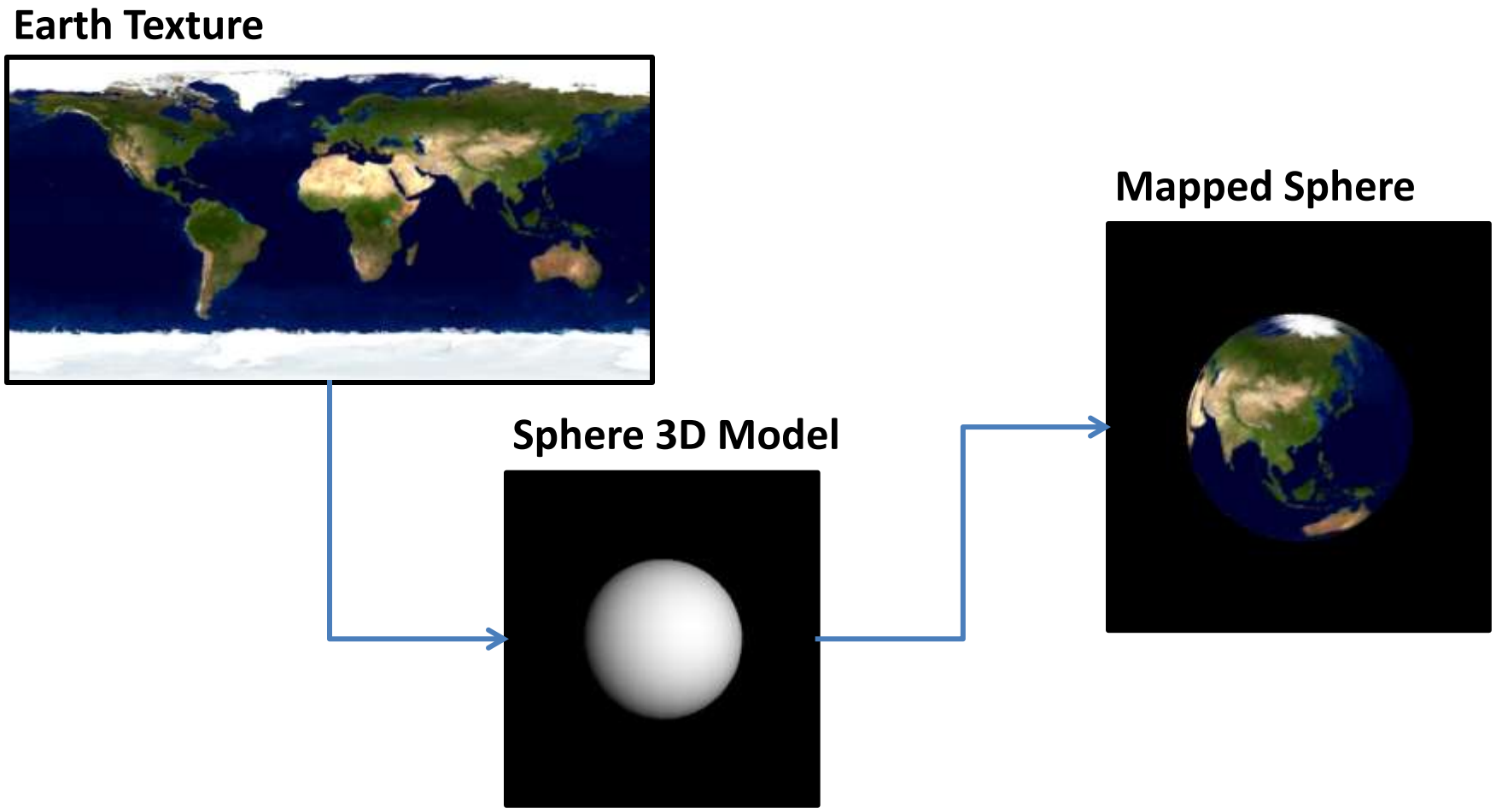
- Demo – FX Composer

Texturas

- Pode ser vista como uma função que mapeia uma coordenada a uma cor
- Sua implementação pode ser
  - A partir de uma imagem (acesso a textura)
  - Proceduralmente (algoritmo)

Texturas

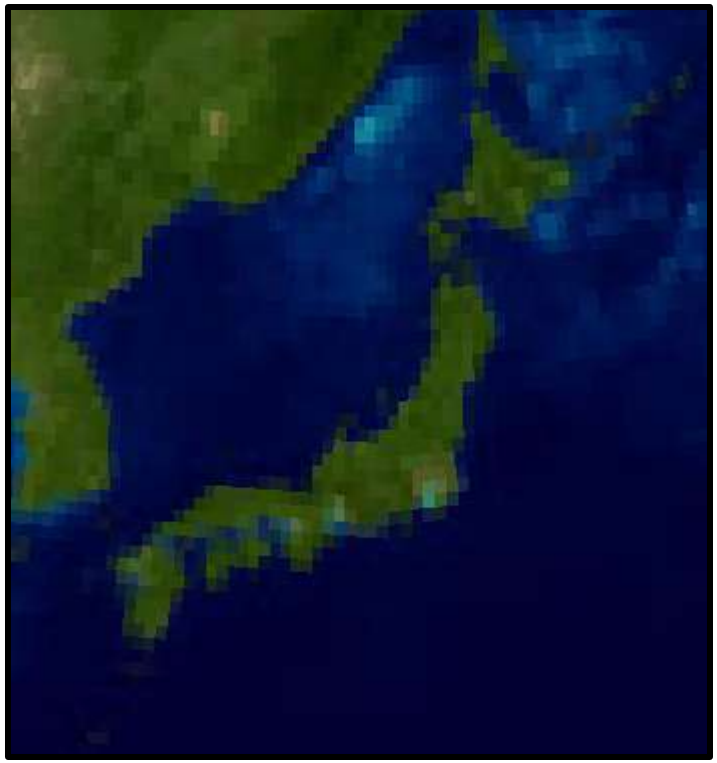
- Mapeando uma textura a uma esfera



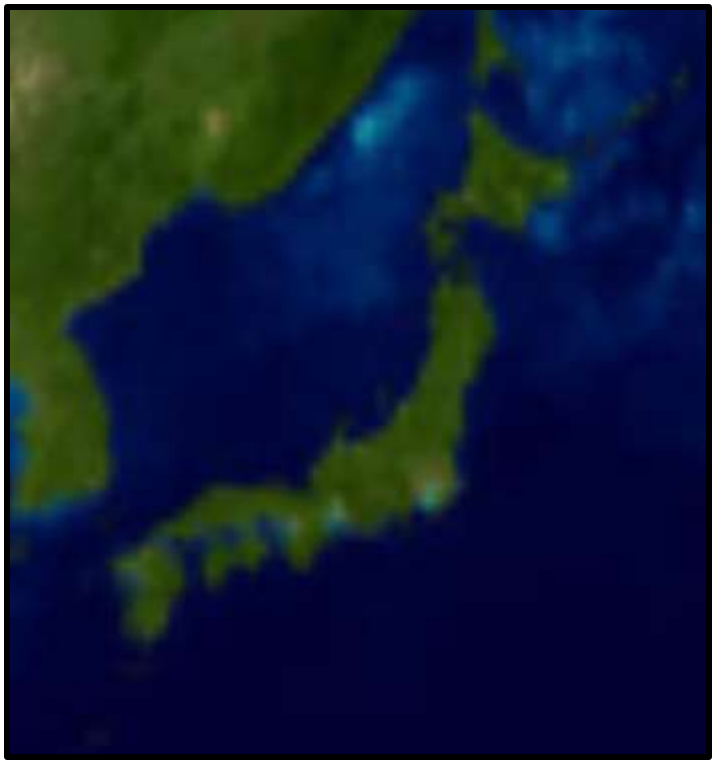
Textura da Terra de: http://www.oera.net/How2/TextureMaps2.htm

Texturas

- Configuração de amostragem para os texels



Nearest Filter



Linear Filter

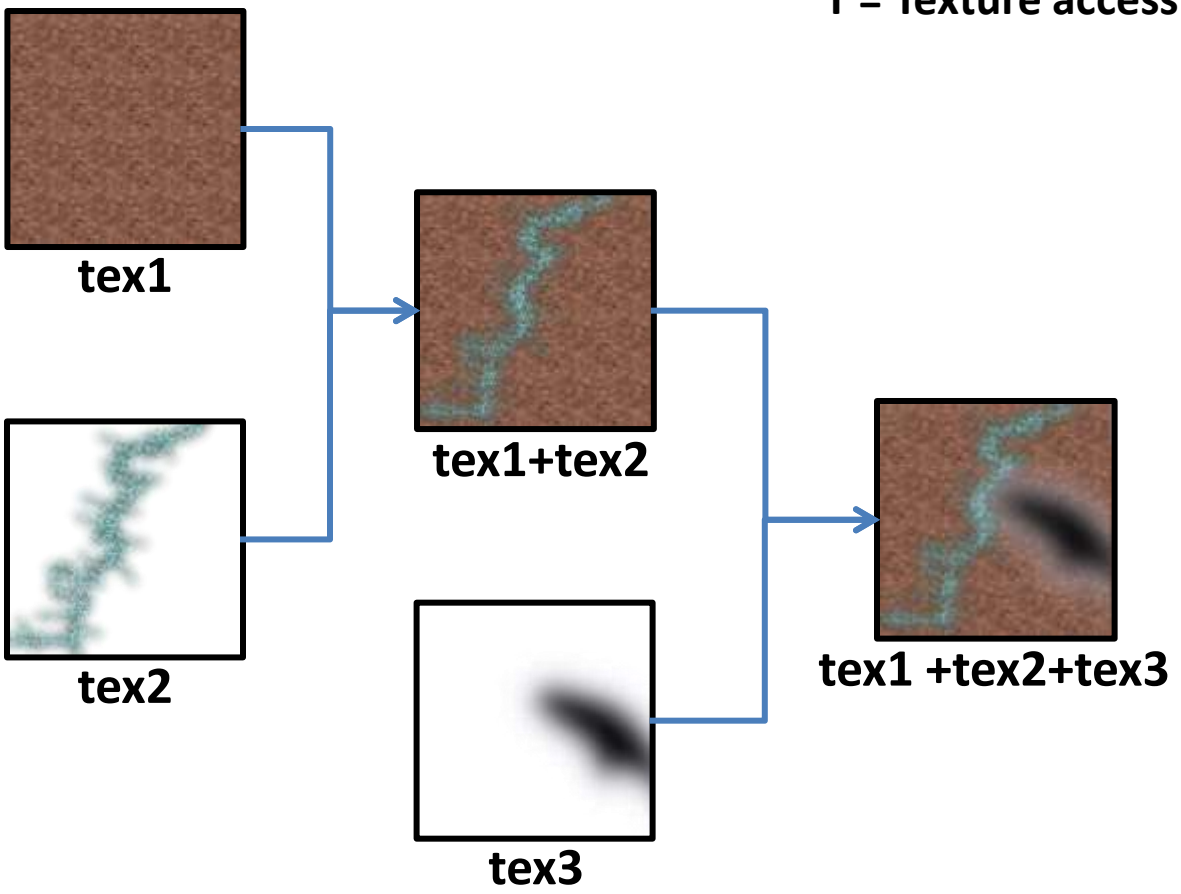
Multi-Texturização

- **Combinando texturas**

$$MT_{[n]}(coord) = \sum_{i=1}^{n} T_i(coord) * \propto_i$$

MT = Multi-texture access

T = Texture access



Texturas

- Demo – FX Composer
  - A_GlobalSingle
  - B_Multitexture

Geração Procedural de Texturas

- São texturas geradas por um algoritmo
- Geralmente é utilizado um modelo de geração
  - Ruído (Perlin, 85)
  - Celular (Worley, 96)
  - Função analítica
  - Etc...

Geração Procedural de Texturas

- Características interessantes
  - Controle paramétrico
  - Representação compacta (armazena poucos parâmetros)
  - Alguns algoritmos são fáceis de implementar
- Problemas
  - Serrilhado
  - Controle sobre os detalhes
  - Desempenho

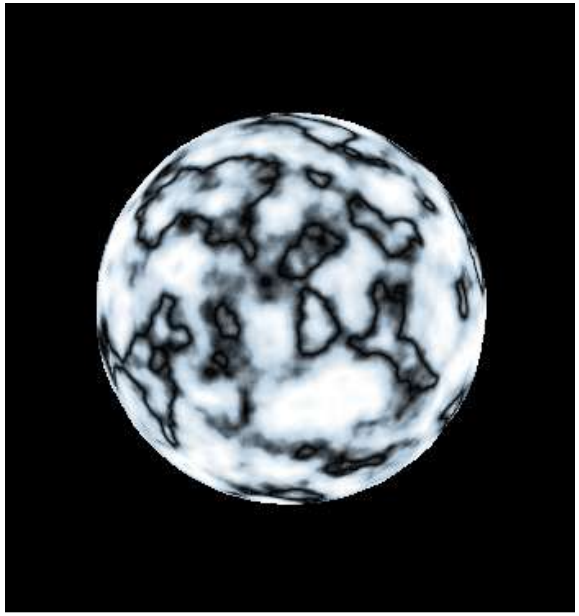
Geração Procedural de Texturas

- **Ruído (Perlin, 85)**
  - **Procura modelar o comportamento aleatório observado na natureza**
- **Propriedades desejadas**
  - **Invariância estatística em relação a orientação e translação**
    - **Manter a aparência do ruído no espaço**
  - **Range dinâmico limitado**
    - **Permitir a amostragem do ruído em várias escalas sem aliasing**

Geração Procedural de Texturas

- **Mármore modelado a partir da função de ruido**
  - **A função de ruido 3D mapeia o mármore em todo o espaço**

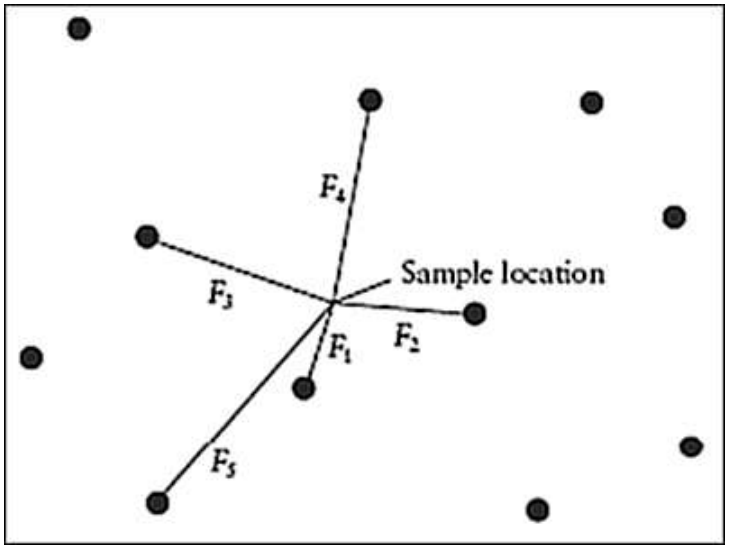
```
float3 procedural_marble3D(float3 pnt){
    float y;
    y = pnt.y*p4+p3 + p2*noise(pnt, p1);
    y = sin(y*M_PI);
    return (marble_color(y));
}
```

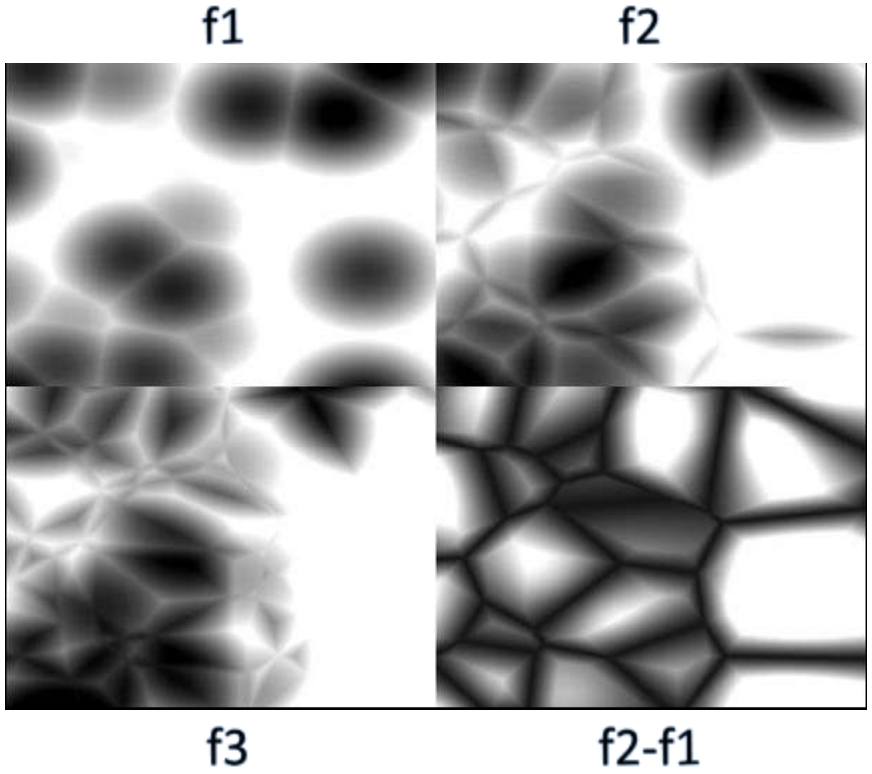


P1, p2, p3 and p4 são os parametros da geração da textura de mármore baseados na função ruído

Geração Procedural de Texturas

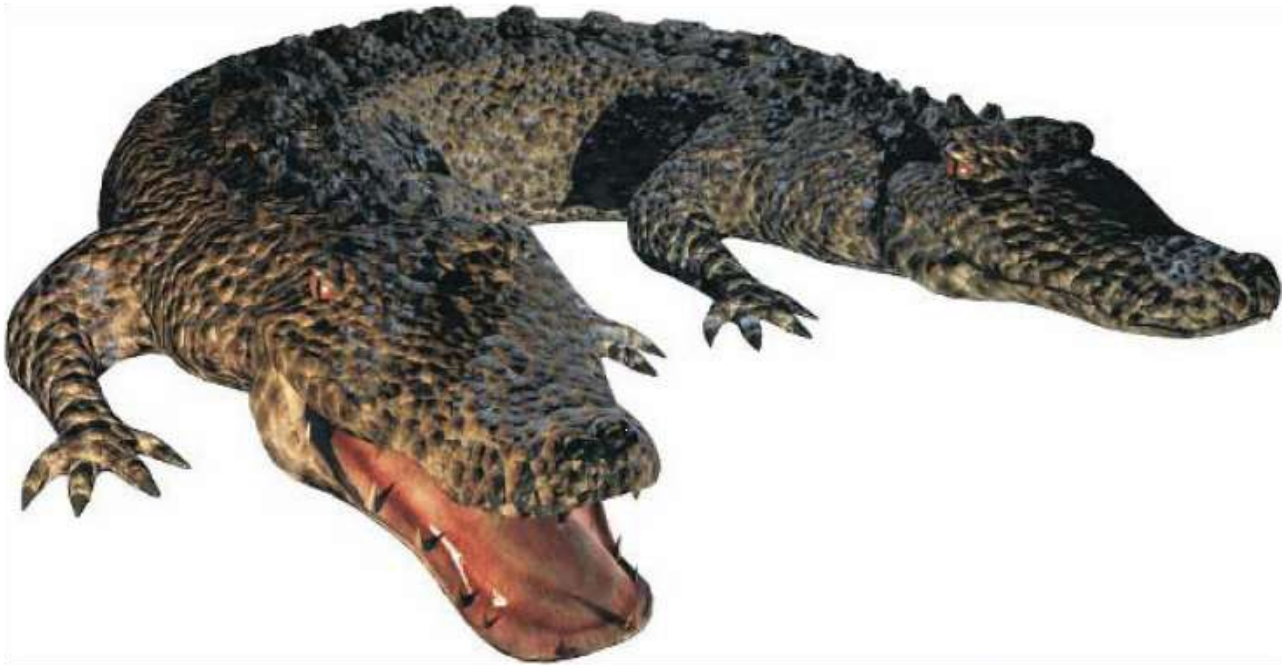
- **Textura celular (Worley, 96)**
  - $F_n(x)$ = nªgésima distancia até o ponto mais próximo
  - $F_n(x) <= F_{n+1}(x)$





Imagens do livro Texturing & Modelling

Geração Procedural de Texturas



Imagens do livro Texturing & Modelling

Geração Procedural de Texturas

- Demo – FX Composer
  - C_ProceduralSquare
  - D_ProceduralMarble

Simulação de Malhas Detalhadas

- Objetos do mundo real geralmente possuem superfícies com um alto nível de detalhe
  - Mesoestrutura: Rugosidade, relevo, pequenas deformações
  - Microestrutura: Detalhes não visíveis ao olho humano
- Problemas
  - Requerem milhões de triângulos para serem representadas (B-rep)
  - Necessita do armazenamento e processamento de grandes volumes de dados

Lucy model

Stanford University

Scanned model

- 116 million triangles

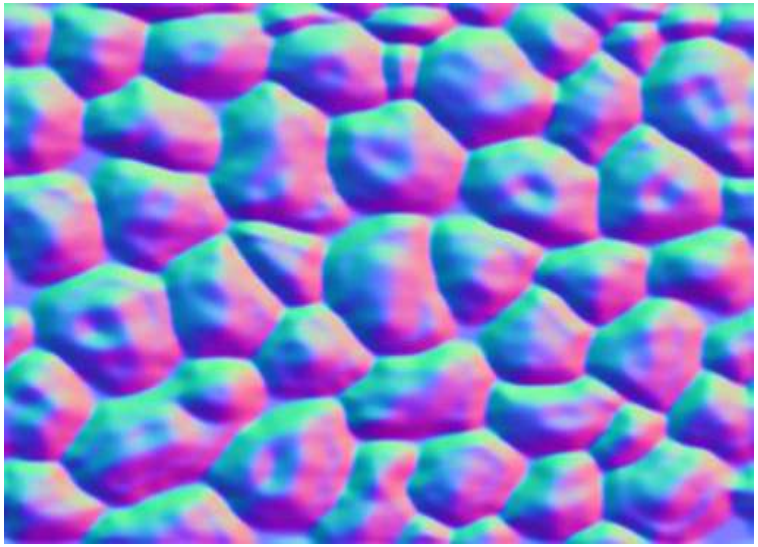
- 325 MB uncompressed

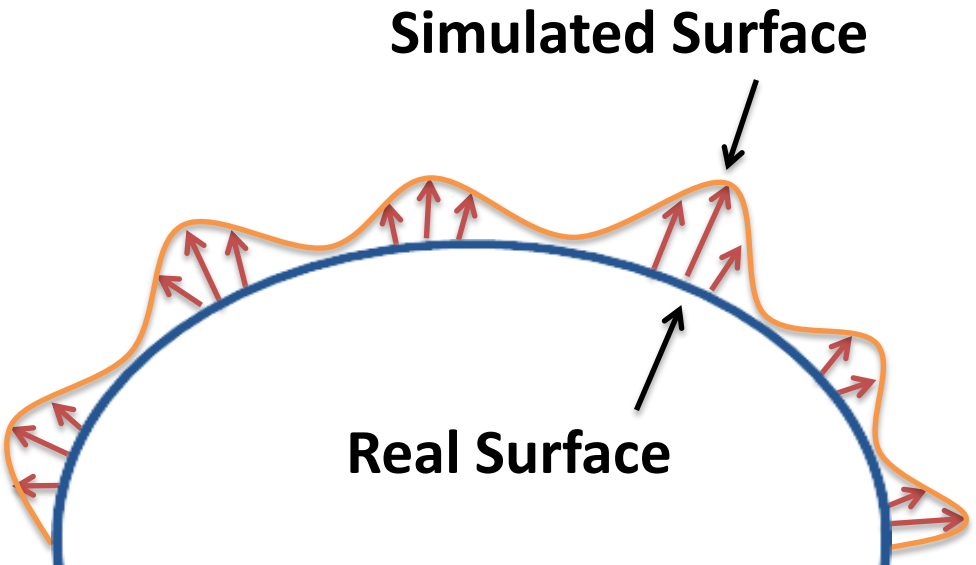
Superfícies Detalhadas

- Solução
  - Simular os detalhes das superfícies sem aumentar a complexidade da mesma
- Algumas técnicas
  - Bump Mapping
  - Normal Mapping
  - Offset Parallax Mapping
  - Relief Mapping
  - Parallax Offset Mapping
  - Cone Step Mapping

Normal Mapping

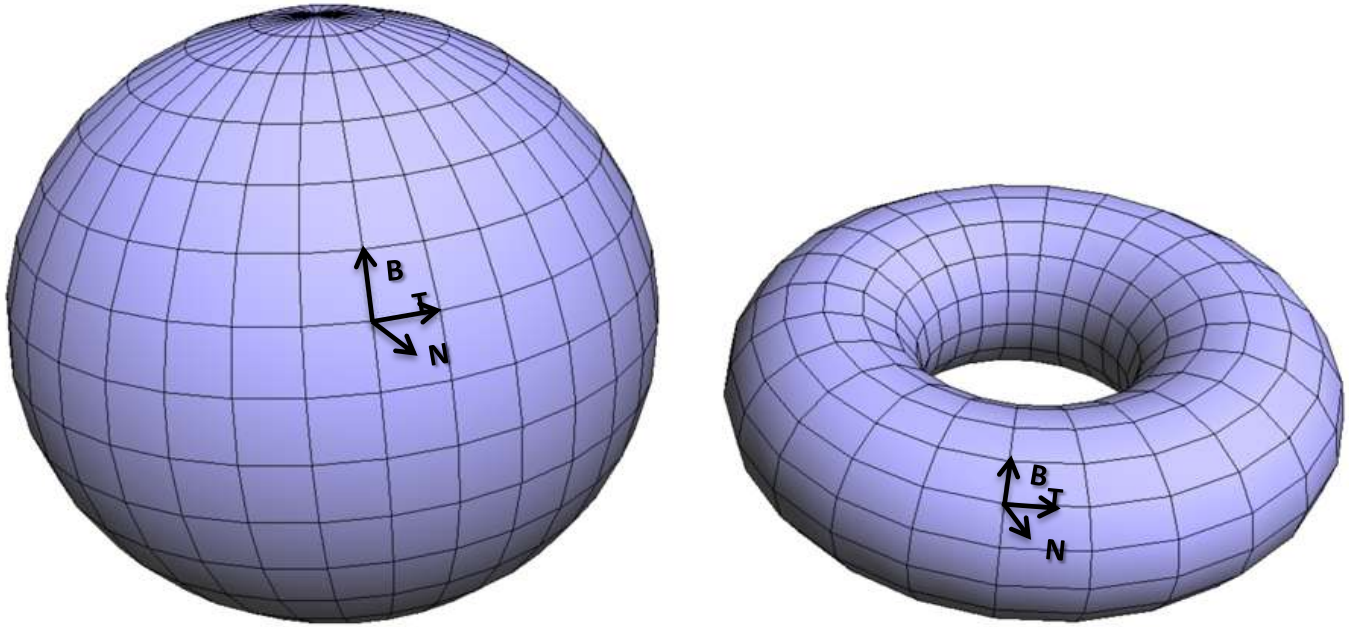
- As normais da superfícies são armazenadas em uma textura que é mapeada em um objeto
  - Os eixos XYZ da normal são mapeados aos canais RGB
  - Os cálculos de iluminação são feitos utilizando o mapa de normais





Normal Mapping

- Os cálculos são feitos no espaço da tangente
  - Base formada pelos vetores: Tangente, Binormal e Normal
- Por que?
  - Normal map se torna independente da superfície



Normal Map Shader

- **Vertex Shader – Passos**
  - **Transformar os vetores de visão e iluminação para o espaço da tangente**
- **Fragment Shader – Passos:**
  - **Ler a normal do mapa de normais**
  - **Iluminar o pixel utilizando a nova normal, o vetor de visão e os vetores de luzes**

Implementação – Shader

Vertex Shader

```
...

// Tangent space (Tangent, binormal, normal)
float3x3 tangentMap = float3x3(IN.tangent, IN.binormal, IN.normal);
tangentMap = transpose(mul(tangentMap, matW));

// View and Light vector
float3 eyeVec = vertexPos - matVI[3].xyz;
OUT.eyeVec = mul(eyeVec, tangentMap);
float3 lightVec = lightPos - vertexPos;
OUT.lightVec = mul(lightVec, tangentMap);
return OUT;
```

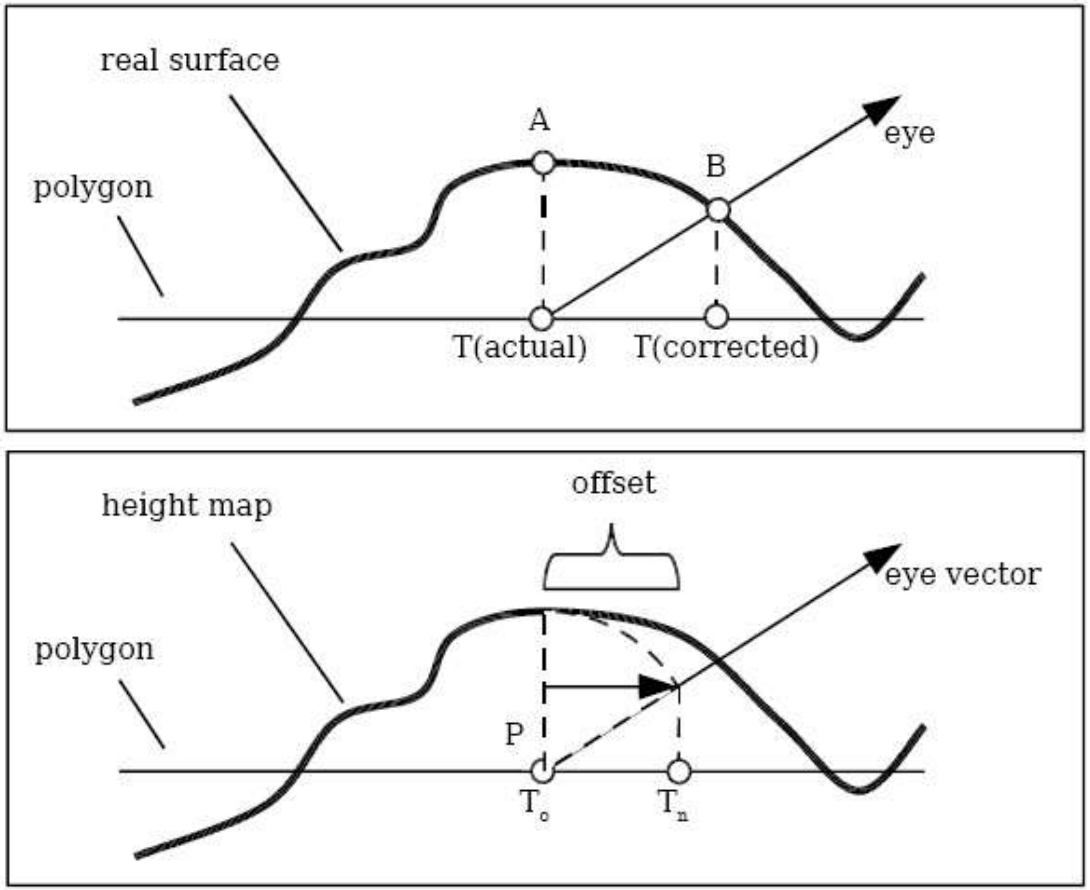
Pixel Shader

```
// View and Light vector
float3 v = normalize(eyeVec);
float3 l1 = normalize(lightVec);

// Diffuse and Normal texture
float3 color = tex2D(color_map, uv0).xyz;
float3 n = tex2D(cone_map, uv0);
n.xy = n.xy * 2.0 - 1.0;
return phongShading(n, l1, -v, color);
```

Offset Parallax Mapping

- **Uma heurística para simular o efeito de Motion Parallax**
  - **Melhora o resultado do normal mapping**



Imagens de Parallax mapping with offset limiting [Welsh 04]

Implementação Shader

- **Calcula a coordenada da textura correta baseada no deslocamento da paralaxe**

```
float offset = tex2D(cone_map, IN.uv0).w *
    parallaxScale - parallaxBias;
IN.uv0 += normalize(IN.eyeVec) * offset;
```

Relief Mapping/POM Mapping

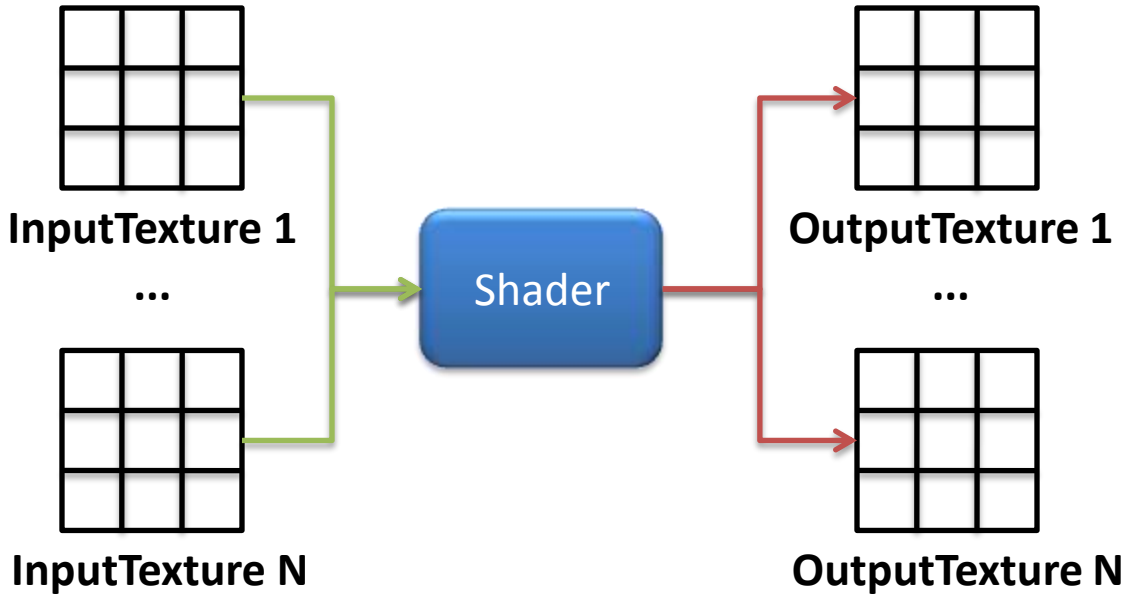
- **Técnica poderosa para renderizar superficies detalhadas com precisão**
  - **Utiliza um algoritmo de raytracing para o cálculo de colisão sobre um campo de alturas**
  - **Demanda muitas iterações para encontrar a posição correta na superfície**

Superfícies Detalhadas

- **Demo – Detailed Surfaces**

Pós-Processamento

- Efeitos que são aplicados sobre a imagem renderizada
  - Pode possuir várias texturas como entrada e escrever em várias texturas

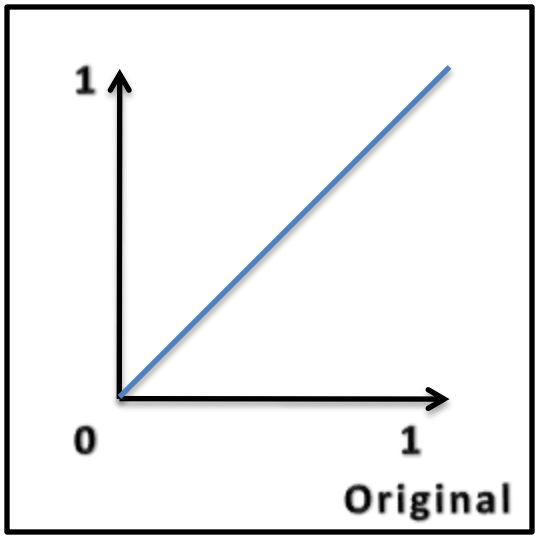


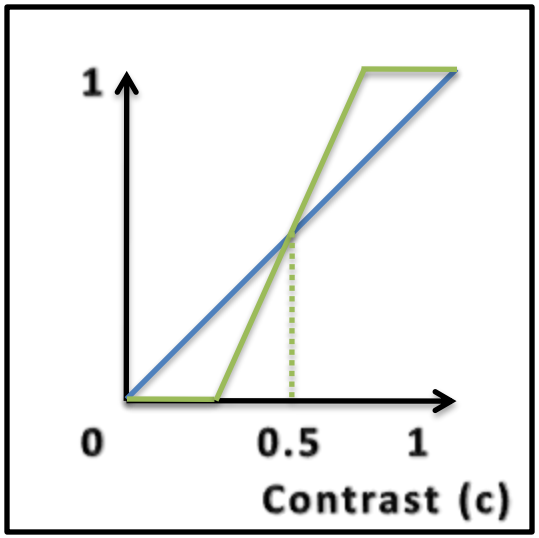
Pós-Processamento

- Algoritmos de processamento digital de imagens (PDI) que podem ser utilizados
  - Transformações radiométricas
    - Contraste, brilho e conversão para tons de cinza
  - Filtros
    - Suavização (blur), detecção de bordas
  - Composição de imagens
    - Radial motion blur

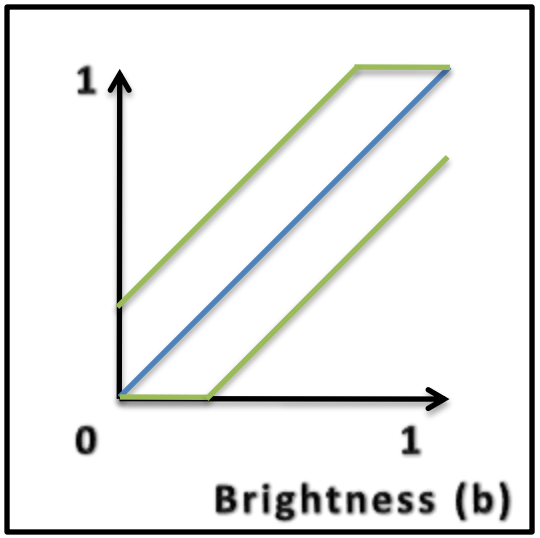
Transformações radiométricas

- **Contraste e Brilho**





$F(x,y) = [F(x,y)-0.5]*c + 0.5$



$F(x,y) = F(x,y) + b$

Transformações radiométricas

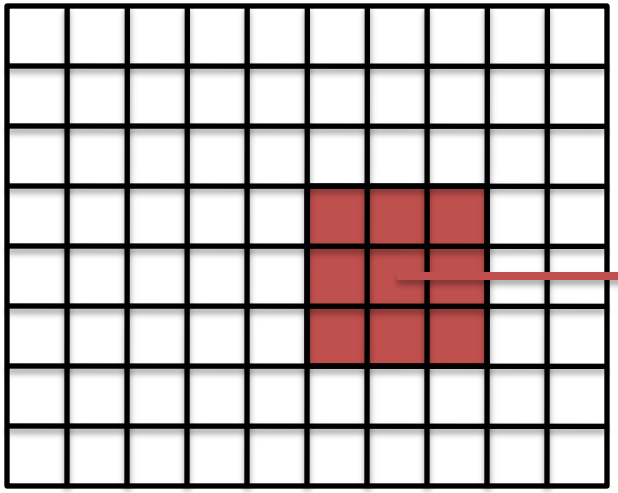
- **Extração de tons de cinza**
  - **Considerando o sistema HSV**
    - **Cinza = V = (r + g + b)/3**
    - **Cinza = V = máximo (r, g, b)**
  - **Considerando a percepção humana (YIQ)**
    - **Sistema de cor YIQ criado para transmissão de sinal de TV**
    - **Cinza = Y = r*0.299 + g*0.587 + b*0.114**

Transformações radiométricas

- Demo – FX Composer
  - E_ppBrilhoContraste
  - F_ppCinza

Filtros

Original Image



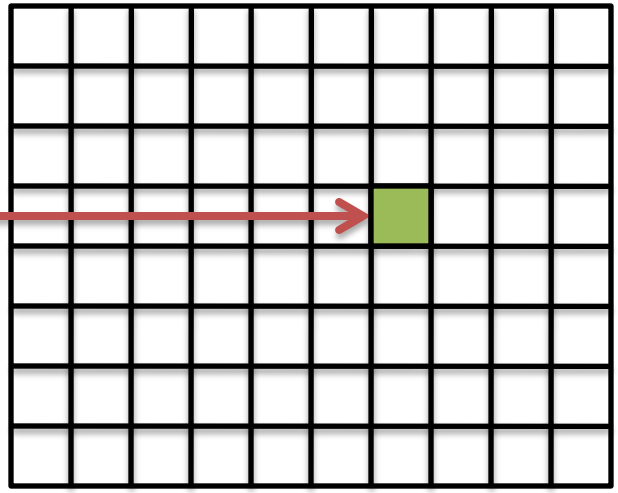
Blur

.11	.11	.11
.11	.11	.11
.11	.11	.11

Edge Detection

0	-1	0
-1	4	-1
0	-1	0

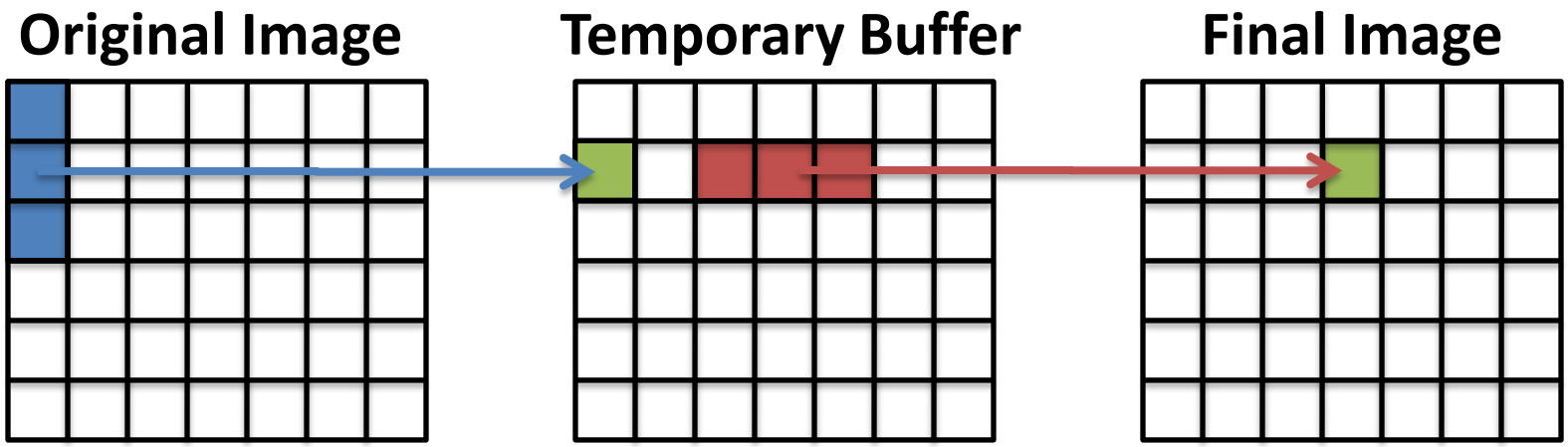
Resulting Image



Complexity: $O(n^2)$

Blur

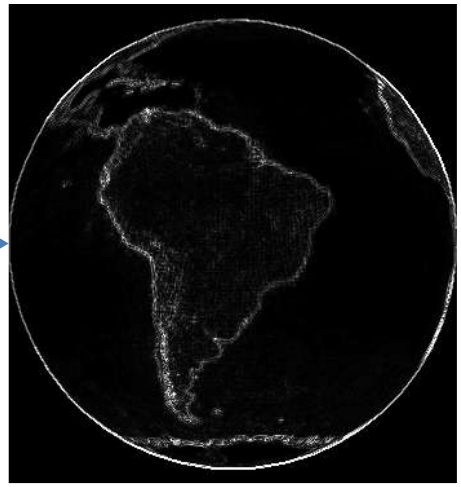
- Este filtro pode ser otimizado utilizando apenas dois passos



Complexity: O(n)

Filtros

Original Image



Edge Detetion

Horizontal Blur

Vertical Blur

Filtros

- Demo – FX Composer
  - G_ppBlur
  - H_ppLaplace

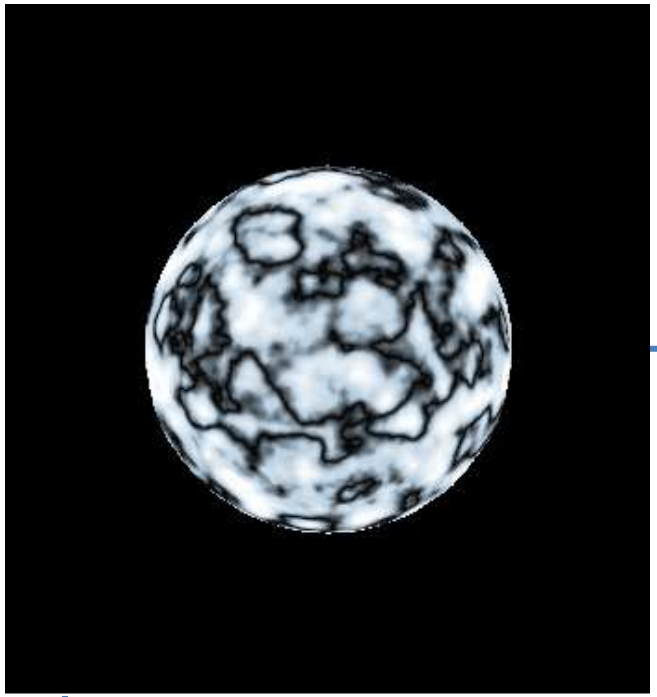
Pós-Processamento

- Efeitos compostos
  - Alguns efeitos necessitam de vários passos de renderização, onde é necessário utilizar alguns buffers auxiliares (ou texturas)
  - Neste caso é necessário implementar o controle de fluxo de renderização (geralmente em software)
    - O controle do fluxo de renderização deve minimizar a utilização de recursos (memória de video) e gerenciar as texturas de saida (render targets)
- Exemplo
  - Bloom, Cartoon Rendering, outros...

Bloom

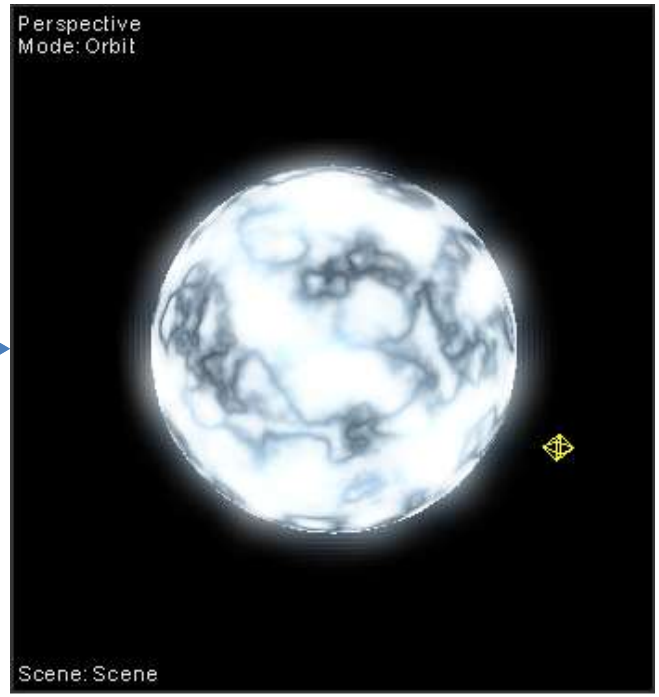
- Procura simular o efeito de expansão de regiões claras sobre superfícies
  - Efeito percebido no mundo real
  - Luzes, Reflexões, etc...
- Geralmente é utilizado juntamente à técnica de HDR (High Dynamic Range)

Bloom

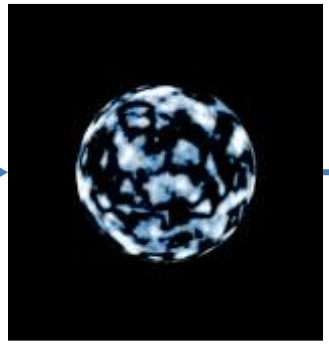


Original Image

Blending

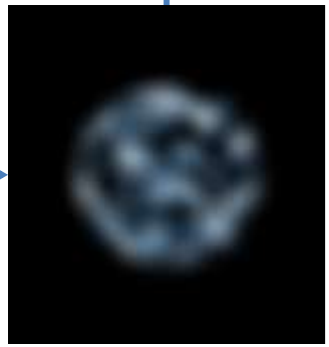


Final Result



Threshould

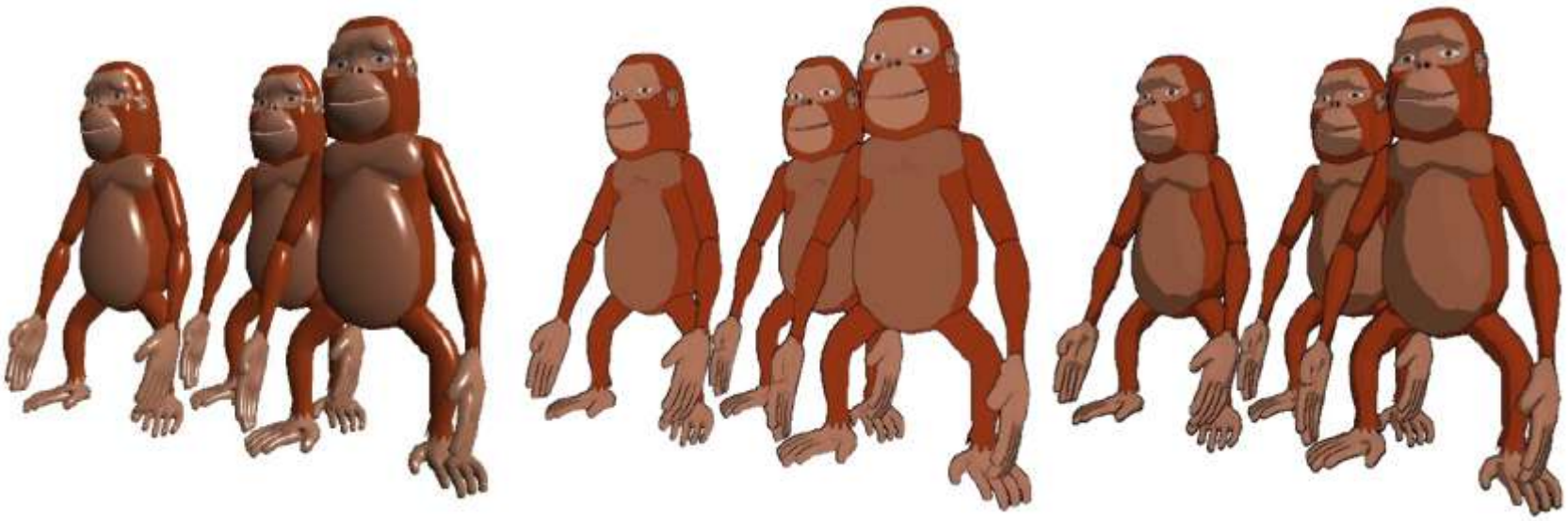
Horizontal Blur



Vertical Blur

Cartoon Rendering

- Renderiza uma cena com a aparência de um desenho feito a mão
  - Número fixo de tons
  - Detecção de bordas

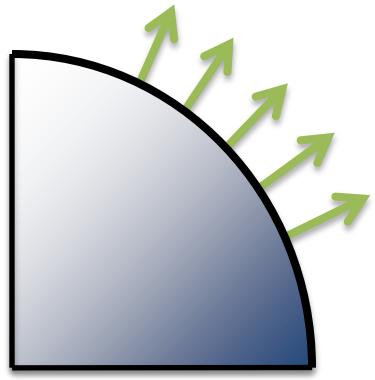


Phong 1 tom 3 tons

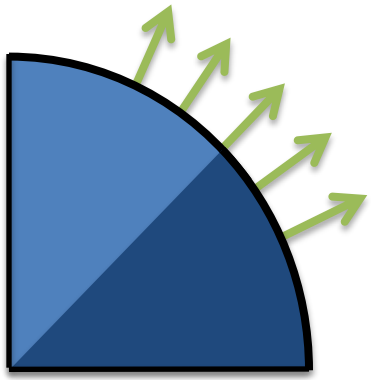
Cartoon Rendering

- **Mapeia o resultado da iluminação de phong para um número fixo de tons (geralmente dois ou três)**

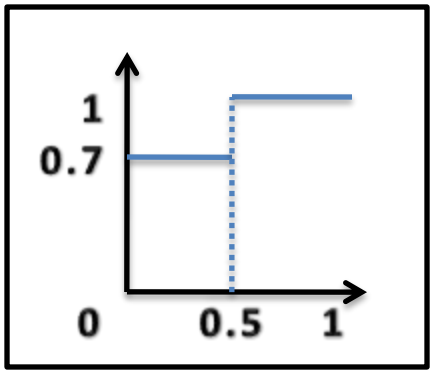
```
IF Phong(x, y, z) < 0.5
   Intensity = 0.7,
ELSE
   Intensity = 1
```



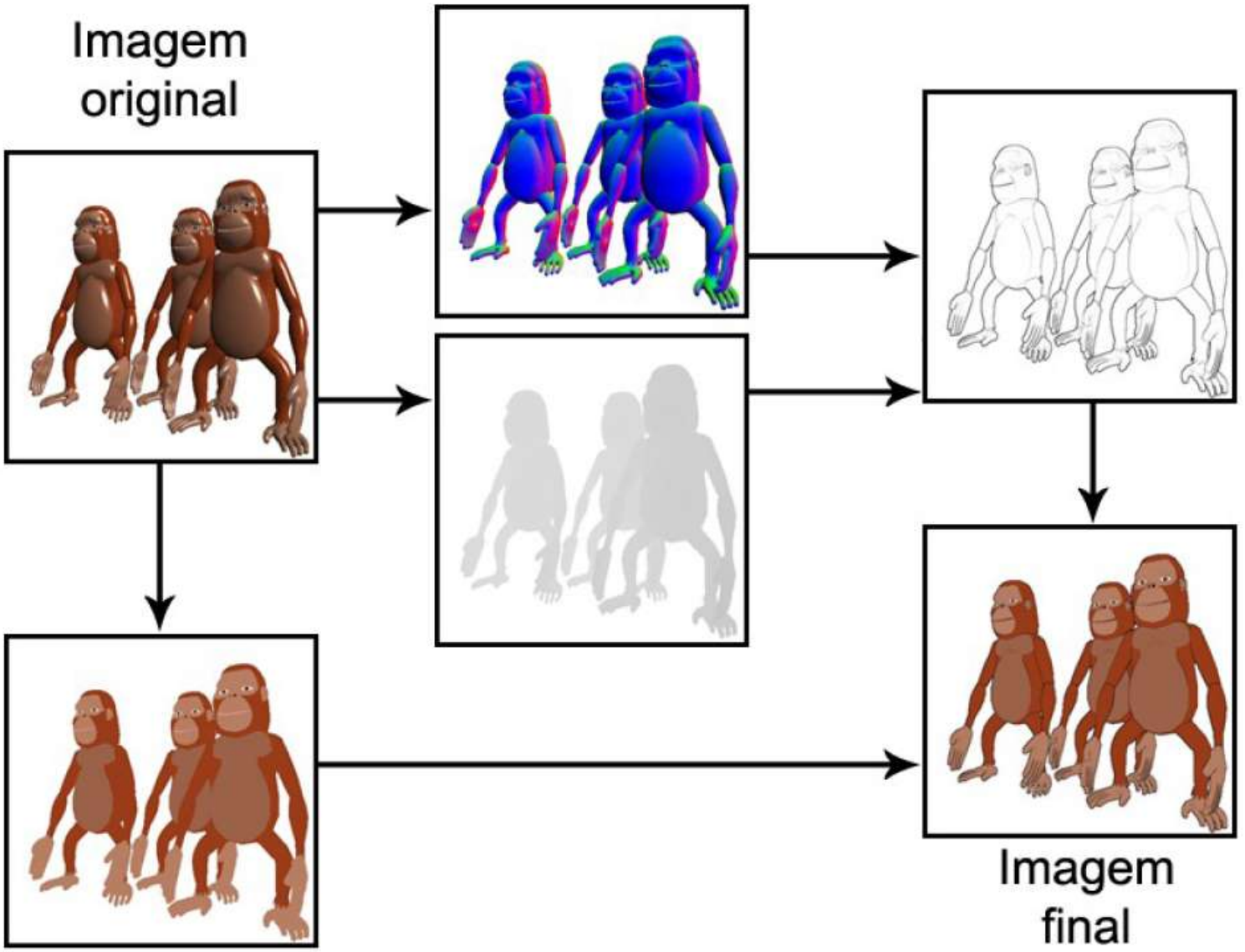
Original Lighting



Quantized Lighting
Using two tones



Cartoon Rendering



Pós-Processamento

- Demo – FX Composer
  - I_Bloom
  - Cartoon

Conclusões

- Nesta palestra apresentamos alguns efeitos que são utilizados em jogos comerciais
  - Todos estes efeitos são implementados em GPUs modernas
- Em um futuro próximo as GPUs serão totalmente programáveis e não existirá mais o pipeline fixo
  - APIs modernas não suportam o pipeline fixo (DirectX 10, XNA e OpenGL Mount Evans)
  - Hoje existem poucos estágios que permanecem fixos: Rasterização, Output Merger

Obrigado!

Bruno P. Evangelista

www.BrunoEvangelista.com

bpevangelista@gmail.com

Alessandro R. Silva

www.AlessandroSilva.com

alessandro.ribeiro.silva@gmail.com

"De fato, que aproveitará ao homem ganhar o mundo inteiro mas perder sua alma?" Mateus 16, 26