Criando Efeitos Fotorealistas e Não-Fotorealistas Para Jogos

Bruno Evangelista[1]
bpevangelista@gmail.com

Alessandro Silva[1]
alessandro.ribeiro.silva@gmail.com

Universidade Federal de Minas Gerais, Brasil [1]

1.Introdução

Nós últimos anos a indústria têm buscado criar jogos com um alto nível de realismo. O objetivo muitas vezes é criar um nível de realismo tão grande, que os jogos não possam ser diferenciados do mundo real, proporcionando assim uma maior imersão do jogador. Um dos grandes fatores que contribui para o aumento no realismo nos jogos foi à introdução dos hardwares gráficos programáveis (GPU – Graphics Processing Unit), que tornou possível programar o fluxo de renderização dos jogos. A flexibilidade de se programar a GPU tornou possível adicionar aos jogos efeitos visuais que antes só eram vistos em filmes. Por outro lado, devido ao grande aumento no realismo nos jogos algumas vertentes da indústria começaram a investir no não-fotorealismo, utilizando técnicas como renderização de cartoons em ambientes tridimensionais.

Neste trabalho serão apresentadas algumas das principais técnicas foto-realistas e não-fotorealistas que têm sido amplamente utilizadas nos jogos nos últimos anos. Para cada uma das técnicas abordadas serão apresentados a sua base teórica, aplicação e resultados obtidos. Na Seção 2 é apresentada uma visão geral do pipeline de renderização programável. Na Seção 3 são apresentadas algumas linguagens de shaders existentes e uma visão geral sobre efeitos. A Seção 4 mostra o processo de construção de um efeito para realizar o cálculo da iluminação de Phong por pixel. A Seção 5 mostra como utilizar texturas dentro de um shader e como gerar essas texturas proceduralmente. Na Seção 6 são apresentadas técnicas que podem ser utilizadas para renderizar superfícies detalhadas em tempo real. A Seção 7 mostra o processo de construção de efeitos de pós-processamento utilizando shaders, esses efeitos são divididos entre transformações radiométricas e filtros. Por fim, a Seção 8 apresenta efeitos mais elaborados.

2.Shaders e Pipeline de Renderização

Shaders são pequenos programas que são executados no hardware gráfico programável (GPU: Graphics Processing Unit), e permitem modificar alguns estágios do pipeline de renderização. Antes da introdução dos shaders, as APIs gráficas como DirectX e OpenGL, possuíam um pipeline de renderização completamente fixo. Isso fazia com que todos os jogos utilizassem um mesmo processo de renderização, sendo possível apenas modificar parâmetros do mesmo. Por exemplo, para iluminar uma cena, era possível escolher o tipo de fontes de luz (direcional, holofote e omnidirecional), e configurar os seus parâmetros. No entanto, o número de fontes de luz disponível era bastante limitado e definido pela API, além disso, não era possível criar novas fontes de luzes, nem modificar o algoritmo de iluminação que era utilizado. A flexibilidade de se programar a GPU tornou possível adicionar aos jogos efeitos visuais que

antes só eram vistos em filmes. Esses efeitos geralmente eram criados utilizando ferramentas como RenderMan [Hanrahan 90], que apesar de possuir uma linguagem de shaders não tira proveito dos hardwares gráficos programáveis e não pode ser utilizada para aplicações de tempo real. A Figura 1 exibe os vários estágios de um pipeline de renderização.

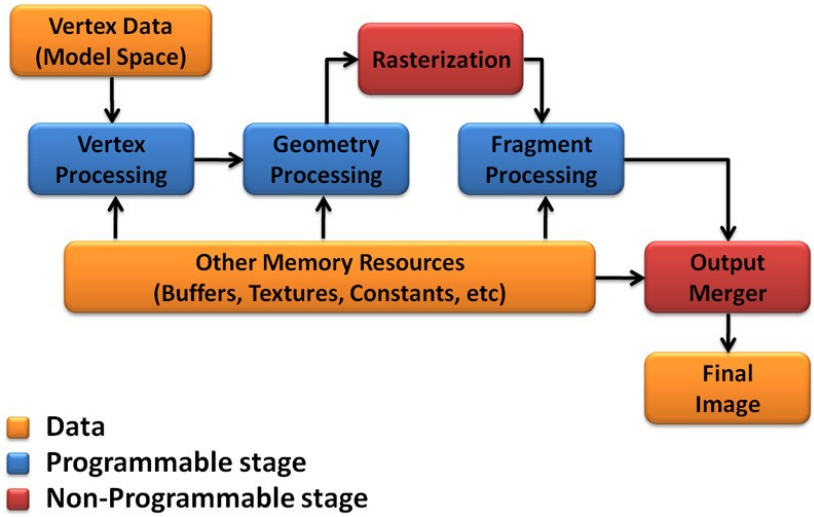


Figura 1: Pipeline de renderização.

A entrada principal do pipeline da Figura 1 são os vértices da malha do modelo, onde cada vértice pode conter informações de posição, cor, normal, coordenada de textura, dentre outros. Vários estágios do pipeline têm acesso a outros recursos de memória, como constantes passadas pela aplicação e texturas. A saída final do pipeline é uma imagem na forma de uma matriz de pixels, que é exibida ao usuário. Os estágios "Vertex Processing", "Geometry Processing" e "Fragment Processing" são os estágios do pipeline de renderização que são atualmente programáveis.

2.1.Processamento de Vértices

O estágio de processamento de vértices é responsável em transformar os vértices da malha do modelo para que eles possam ser utilizados no estágio de rasterização [Foley 97], e posteriormente no processamento de pixels. A Figura 2 exibe o estágio de processamento de vértices, e algumas das etapas que eram realizadas pelo pipeline de renderização fixo. Com a introdução do pipeline programável fica a critério do usuário decidir qual processamento deve ser feito para cada vértice.

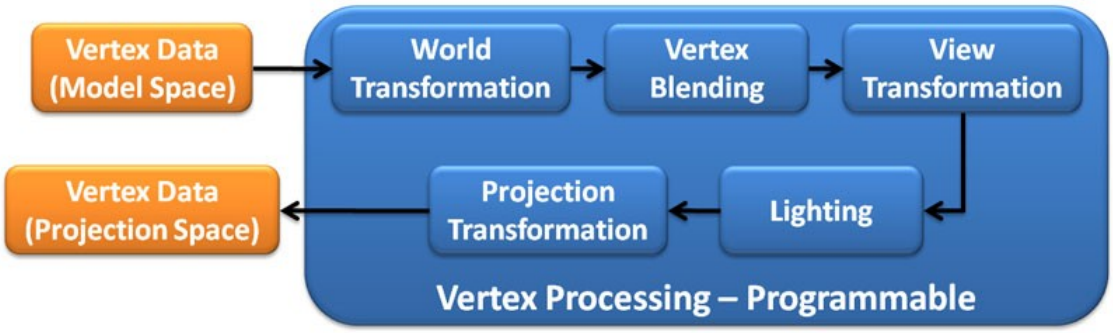


Figura 2: Estágio de processamento de vértices.

No pipeline fixo de renderização este estágio era utilizado para transformar os vértices e calcular a iluminação dos mesmos. Utilizando o pipeline programável é possível realizar novas tarefas, como: deformação de sólidos, animação esqueletal, movimentação de partículas, dentre várias outras. A entrada do processamento de vértices são os dados dos vértices da malha do modelo, e a saída é a posição final do vértice (no espaço de projeção) e outros dados definidos pelo usuário. Os dados de saída serão utilizados posteriormente como entrada para o processamento de pixels. Por exemplo, é possível adicionar à saída de cada vértice o seu vetor normal. Esta normal será interpolada e utilizada como entrada para o processamento de pixels, que poderá utilizá-la para fazer um cálculo de iluminação diferente para cada pixel.

A saída do processamento de vértices é utilizada como entrada no processamento de geometrias [Blythe 06]. O processamento de geometrias, não será abordado neste trabalho, mas em termos gerais ele pode ser utilizado para modificar as geometrias existentes, ou criar novas geometrias em tempo real. A saída do processamento de geometrias é utilizada como entrada no estágio de rasterização.

2.2.Rasterização

A rasterização é responsável em transformar os dados dos vértices em fragmentos. Um fragmento é um conjunto de informações relacionadas a um pixel, como: cor, profundidade, coordenada de textura e outros. Apesar de fragmento ser um termo mais correto, geralmente o termo pixel é utilizado para se referir a um fragmento. A Figura 3 exibe o estágio de rasterização, este é um estágio fixo do pipeline que não pode ser programado.

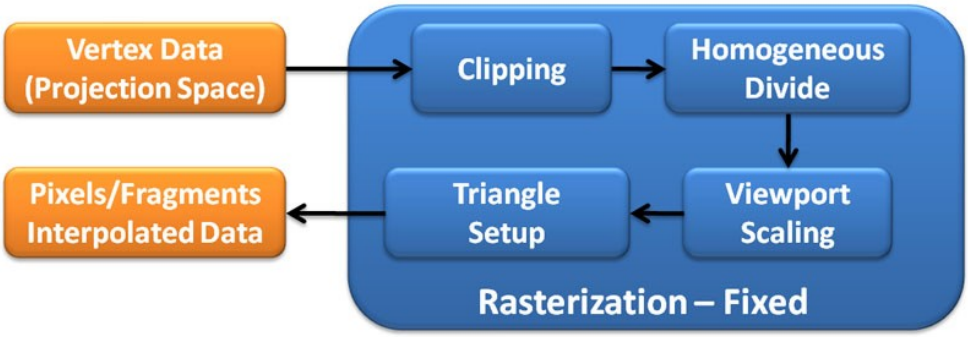


Figura3: Estágio de rasterização.

Na etapa de rasterização as primitivas que possuem vértices fora do volume de visão são recortadas, e apenas a parte dentro do volume de visão é mantida. Em seguida as primitivas, geralmente triângulos, são rasterizadas e transformadas em um conjunto de fragmentos, que são escalados para a dimensão da tela. Os dados dos vértices das primitivas são interpolados para cada fragmento gerado. Os fragmentos gerados pela rasterização são utilizados como entrada para o processamento de pixels.

2.3.Processamento de Pixels/Fragmentos

O estágio de processamento de pixels é responsável em gerar a cor final de cada pixel, a partir dos dados recebidos do estágio de rasterização.

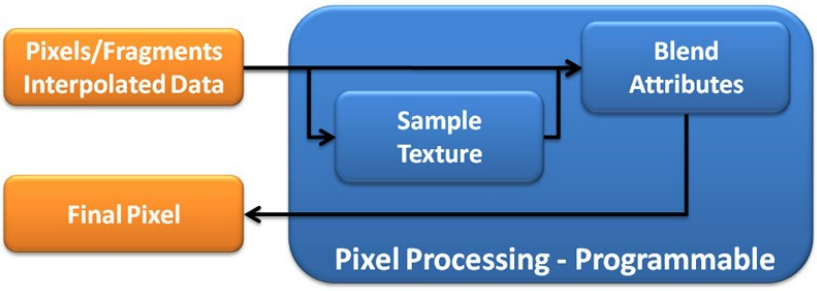


Figura 4: Estágio de processamento de pixels.

No pipeline fixo de renderização este estágio era utilizado para amostrar texturas e gerar a cor final de cada pixel. Utilizando o pipeline programável é possível realizar diversas novas tarefas, como: iluminação por pixel, iluminação global (HDR), geração de texturas procedurais, normal mapping, além de efeitos de pós-processamento sobre a imagem gerada, como: motion blur, bloom e outros. Os pixels de saída do processamento de pixels estão prontos para serem gravados no framebuffer (matriz de pixels).

Antes dos pixels serem gravados no framebuffer, eles devem passar por um último estágio, chamado Output Merger. Neste estágio os pixels recebidos podem ser descartados por serem oclusos por outros pixels, por serem transparentes ou por não colaborarem para a cor final do pixel da imagem. Com isso, apenas alguns pixels irão ser efetivamente gravados no framebuffer. Ao final de todo esse processamento o framebuffer contém a imagem final resultante do processamento de renderização.

3.Linguagens de Shaders

Para desenvolvermos shaders precisamos utilizar uma linguagem própria para a programação dos shaders. A escolha da linguagem de shaders a ser utilizada está muitas vezes ligada à escolha da API gráfica utilizada. Por exemplo, utilizando o DirectX podemos utilizar as linguagens de shader HLSL (Microsoft) ou Cg (nVidia). Existem ainda outras linguagens de shaders que podem ser utilizadas para renderização offline, como o RenderMan.

Atualmente as principais linguagens de shaders utilizadas para renderização em tempo real, são:

- HLSL (High Level Shading Language) – Microsoft. Utilizada no DirectX e XNA
- GLSL (OpenGL Shading Language) – 3D Labs. Utilizada no OpenGL
- Cg (C for Graphics) – nVidia. Pode ser utilizada no DirectX e OpenGL

De maneira geral, as linguagens de shaders utilizadas no DirectX e OpenGL são bem similares. Essas linguagens são compostas por um pequeno conjunto de funções, como operações matemáticas, acessos a texturas, controle de fluxo e outros. Os tipos de dados utilizados são similares aos tipos padrões existentes no C++, além de vetores e matrizes. Por exemplo, é possível criar um vetor de variáveis do tipo float de vários tamanhos, como: float2, float3 e float4. E também é possível criar matrizes, como: float2x2, float3x3 e float4x4. O código final do shader gerado geralmente é similar a uma equação matemática, construída utilizando as funções presentes na linguagem de shaders utilizada.

3.1.Efeitos

Efeitos são entidades que podem encapsular vários shaders diferentes, além de configurações dos estados do pipeline fixos necessárias para a execução do mesmo. Em cada efeito é possível criar uma ou mais técnicas, que são utilizadas na renderização das malhas. Cada técnica criada deve definir quais shaders serão utilizados para o processamento de vértices, geometria e pixels. Isso permite, por exemplo, que duas técnicas diferentes possam utilizar um mesmo processamento de vértices, mas possuam diferentes processamentos de pixels. A técnica a ser utilizada na renderização de um objeto pode ser escolhida em tempo de execução de acordo, por exemplo, com os recursos do hardware gráfico do usuário.

4.Equação de iluminação de Phong

Existem vários equações de iluminação na literatura [Möller 99], que permitem calcular a intensidade final de luz em um ponto de uma superfície. A equação de iluminação Phong [Möller 99], foi uma das equações mais utilizadas para esse propósito, tendo sido utilizada no pipeline fixo de várias APIs gráficas como DirectX e OpenGL. Sendo que geralmente a iluminação era calculada para cada vértice da malha de um objeto, e o resultado era interpolado por toda a superfície do objeto. A equação de Phong não considera as propriedades físicas das superfícies, ou trata de maneira real a iteração entre a luz e as superfícies, gerando um resultado aproximado com pouco realismo, mas rápido de ser calculado. A equação proposta por Phong é apresentada na Equação 1.

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

Equação 1: Equação de iluminação de Phong [Möller 99].

Segundo a Equação 1, a intensidade total de luz refletida em um ponto da superfície é igual à intensidade da componente de luz ambiente da cena, somada as intensidades das componentes de luz difusa e especular para cada fonte de luz. A luz ambiente é a luz que não é emitida diretamente de uma fonte de luz, mas está espalhada no ambiente. Por isso, considera-se que sua intensidade contribui igualmente para todos os objetos da cena, e pode ser representada através de uma constante. A componente de luz difusa é calculada para cada fonte de luz, e representada pela luz que incide em um ponto na superfície e é refletida igualmente em todas as direções sobre a superfície. A Figura 5 ilustra a reflexão difusa da luz.

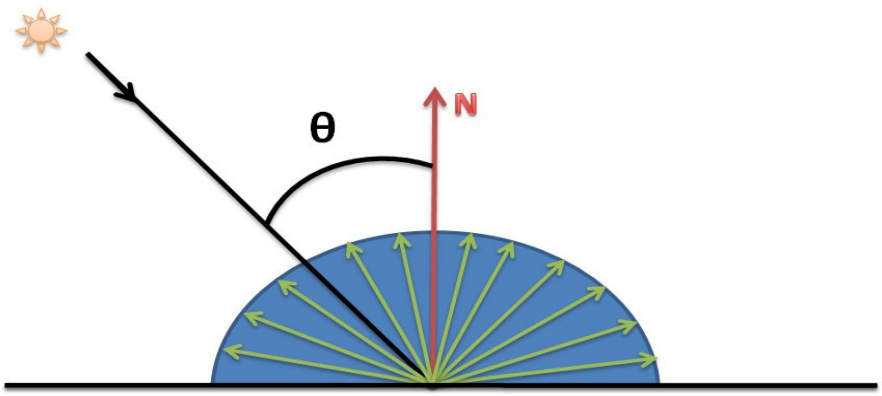


Figura 5: Reflexão difusa da luz.

A Intensidade da reflexão difusa da luz em um ponto da superfície é calculada utilizando a lei de reflexão de Lambert, e é apresentada na Equação 2.

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

Equação 2: Cálculo da intensidade de reflexão difusa em um ponto da superfície.

Na Equação 2, a intensidade de luz difusa refletida em um ponto é dada pelo coeficiente de reflexão difusa da superfície K_d, pela intensidade da componente difusa da luz L_d, e pelo ângulo entre o vetor de luz incidente e a normal da superfície.

A última componente da equação de Phong é a componente especular. A reflexão especular é aquela que incide em um ponto da superfície e reflete como um espelho perfeito, onde o ângulo de incidência e reflexão da luz são iguais. Desta maneira a intensidade de luz especular, depende da posição pela qual o ponto da superfície é observado. A Figura 6 ilustra a reflexão especular da luz.

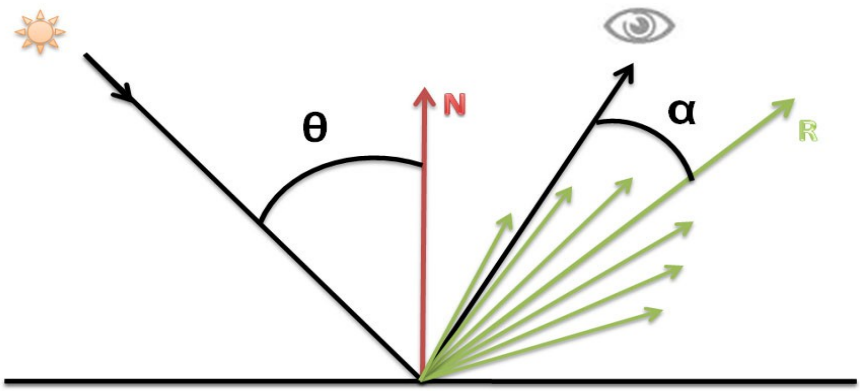


Figura 6: Reflexão especular da luz.

A Intensidade da reflexão especular da luz em um ponto da superfície é calculada utilizando a lei de reflexão de Snell, e é apresentada na Equação 3.

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

Equação 3: Cálculo da intensidade de reflexão especular em um ponto da superfície.

Na Equação 3, a intensidade de luz especular refletida em um ponto é dada pelo coeficiente de reflexão especular da superfície K_s, pela intensidade da componente especular da luz L_s, e pelo ângulo entre o vetor de luz refletida e o vetor de visão do ponto. Note que nas APIs gráficas o cálculo de iluminação especular geralmente utiliza o halfway vector [Blinn 77], ao invés de calcular o vetor real de reflexão da luz. A Figura 7 apresenta o cálculo das componentes de luz ambiente, difusa e especular para uma esfera. E a Figura 8 apresenta o resultado final da combinação das três componentes.

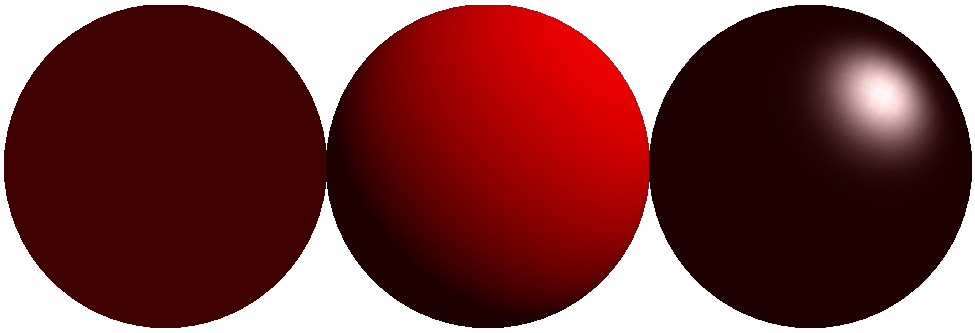


Figura 7: (a) Cálculo da componente ambiente da luz. (b) Cálculo da componente difusa da luz. (c) Cálculo da componente especular da luz.

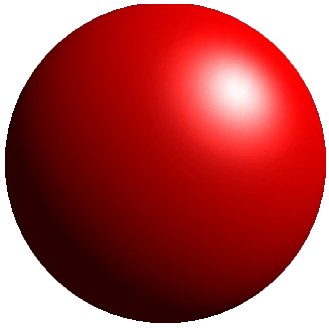


Figura 8: Resultado final da equação de iluminação de Phong. Combinação das três componentes apresentadas na Figura 7.

4.1.Criando efeito

Nesta seção criaremos um efeito de iluminação por pixel utilizando a equação de Phong apresentada na Seção 2.1, que será utilizado na renderização dos objetos de uma cena. A Tabela 1 apresenta algumas das funções de linguagem HLSL, utilizadas na construção do efeito.

dot	Calcula o produto escalar entre dois vetores.
mul	Realiza multiplicação entre vetores e matrizes.
normalize	Normaliza um vetor.
reflect	Reflete um vetor incidente a uma normal.
saturate	Satura os valores entre 0.0 e 1.0

Tabela 1: Funções do HLSL utilizadas no efeito criado.

Quando criamos um efeito à primeira coisa que precisamos declarar são os dados que serão recebidos da aplicação, esses dados são declarados como variáveis globais no efeito. O código a seguir apresenta as variáveis utilizadas pelo efeito.

```
// Matrix
// ------------------------------------------------
float4x4 matW : World;
float4x4 matV : View;
float4x4 matVI : ViewInverse;
float4x4 matWV : WorldView;
float4x4 matWVP : WorldViewProjection;

// Ambient light
// ------------------------------------------------
float3 ambientLightColor;

// Light 0
// ------------------------------------------------
float3 lightPosition;
float3 lightColor;

// Material
// ------------------------------------------------
float3 materialColor;
float materialKd;
float materialKs;
float materialShininess;
```

As matrizes utilizadas pelo efeito são as matrizes de mundo, visão e projeção. Note que algumas combinações dessas matrizes, como matWV (Mundo * Visão) e matWVP (Mundo * Visão * Projeção), também são declaradas para não precisarem ser calculadas no efeito. Em seguida declaramos a cor da componente ambiente de luz da cena, a posição e cor de uma fonte de luz omnidirecional e as propriedades do material da superfície. Todas essas informações são utilizadas no cálculo de iluminação.

Devemos declarar também uma estrutura com os dados dos vértices da malha do modelo que serão enviados ao processamento de vértices. E uma outra estrutura, que define os dados de saída do processamento de vértice, que serão utilizados como entrada do processamento de pixels. O código a seguir apresenta as estruturas criadas.

```
// Application to Vertex
// ------------------------------------------------
struct a2v
{
    float4 position   : POSITION;
    float3 normal     : NORMAL;
};

// Vertex to Fragment
// ------------------------------------------------
struct v2f
{
    float4 hposition  : POSITION;
    float3 normal     : TEXCOORD1;
    float3 lightVec   : TEXCOORD2;
    float3 eyeVec     : TEXCOORD4;
};
```

A estrutura a2v define os dados de entrada do processamento de vértices, e a estrutura v2f define os dados de saída do processamento de vértice. Os dados de entrada são a posição do vértice e seu vetor normal, e os dados de saída são a posição final do vértice e os vetores de normal, luz e visão.

Após declararmos as variáveis e estruturas utilizadas pelo efeito podemos criar o código do processamento de vértices e pixels. A seguir é apresentado o código da função de processamento de vértices criada. Observe que a entrada da função é a estrutura a2v, e a saída à estrutura v2f.

```
v2f LightingVS(a2v IN)
{
    v2f OUT;

    // Transform vertex and normal
    OUT.hposition = mul(IN.position, matWVP);
    OUT.normal = mul(IN.normal, matWV);

    // Calculate light and eye vectors
    float4 worldPosition = mul(IN.position, matW);

    float3 eyePosition = matVI[3].xyz;
    OUT.eyeVec = mul(eyePosition - worldPosition, matV);
    OUT.lightVec = mul(lightPosition - worldPosition, matV);

    return OUT;
}
```

No processamento de vértices a posição final do vértice é calculada, e a matriz transformada de acordo com as matrizes de mundo e visão. Em seguida são calculados os vetores que apontam do vértice para a luz e para o observador da cena. Repare que no final do processamento todos esses vetores se encontram no espaço de visão. O código a seguir apresenta a função utilizada para o processamento de pixels. A seguir é apresentado o código da função de processamento de pixels.

```
float4 LightingPS(v2f IN): COLOR0
{
    // Normalize all input vectors
    float3 normal = normalize(IN.normal);
    float3 lightVec = normalize(IN.lightVec);
    float3 eyeVec = normalize(IN.eyeVec);

    // Calculate reflected light
    float3 reflectLight = reflect(-lightVec, normal);

    // Calculate diffuse and specular intensity
    float diffuseInt = saturate(dot(normal, lightVec));
    float specularInt = saturate(dot(eyeVec, reflectLight));
    specularInt = pow(specularInt, materialShininess);

    // Modulate by the light color
    float3 diffuseColor = diffuseInt * materialKd * lightColor;
    float3 specularColor = specularInt * materialKs * lightColor;

    // Final output color
    float4 finalColor;
        finalColor.rgb = specularColor + materialColor *
        (ambientLightColor + diffuseColor);
    finalColor.a = 1.0f;

    return finalColor;
}
```

No processamento de pixels, a equação de Phong apresentada na Seção 2.1 é calculada para cada pixel e a cor final do pixel é gerada. Por fim, precisamos criar uma técnica que defina qual

processamento de vértice e pixel iremos utilizar. A seguir é apresentado o código da técnica para iluminação criada.

```
technique Lighting
{
    pass p0
    {
        VertexShader = compile vs_2_0 LightingVS();
        PixelShader = compile ps_2_0 LightingPS();
    }
}
```

5.Texturas

Uma das primeiras formas de acrescentar detalhes em uma renderização sem acrescentar geometrias é através da utilização de texturas. Geralmente uma textura é uma imagem 2D composta por texels (texture elements) que é aplicada sobre um triângulo. Esta definição restringe uma gama de aplicações interessantes que podemos fazer, por este motivo textura para este tutorial deve ser vista por uma visão mais geral: É uma função que mapeia uma coordenada n-dimensional a uma informação de 4 componentes que pode representar uma cor. Podemos ver na Equação 4 um exemplo da função textura 'T'.

$$T(coordenada) = informação_{rgba}$$

Equação 4: Função textura.

Uma textura no shader é acessada através de um sampler. Mas somente funções específicas como tex1D, tex2D ou tex3D podem ser utilizadas. O trecho de código abaixo é um exemplo de como configurar um sampler e como acessa-lo.

```
//definição da textura
texture tex1 : DIFFUSE;
...
//configuração do sampler para a textura utilizada
sampler2D tex1sampler = sampler_state {
    Texture = <tex1>;
    MinFilter = Linear;
    MipFilter = Linear;
    MagFilter = Linear;
    AddressU = Wrap;
    AddressV = Wrap;
};
...
struct app2vertex {
    float4 UV   : TEXCOORD0;
    ...
};
struct vertex2fragment {
    float2 UV   : TEXCOORD0;
    ...
};
vertex2fragment std_VS(app2vertex IN) {
    ...
    OUT.UV = IN.UV;
    return OUT;
}
float4 std_PS(vertex2fragment IN) : COLOR {
    //utilizando função de acesso ao sampler já configurado
    float4 texel1 = tex2D(tex1sampler,IN.UV);
    ...
    return float4(result, texel1.a);
}
```

A textura é definida pelo tipo ‘texture’, mas para utilizar a mesma deve-se configurar um sampler para a mesma a fim de configurar o modo como os texels serão acessados e filtrados.

O espaço de coordenadas de textura geralmente é desvinculado do espaço do objeto ou modelo 3D utilizado na aplicação, por isso é necessário definir uma forma de mapeamento. Este mapeamento pode ser feito de forma analítica (mapear esferas, cubos, cilindros, etc...) ou pode ser feito por um artista com o auxilio de uma ferramenta de modelagem. Na Figura 9 é possível observar um exemplo de textura 2D com o espaço de endereçamento a partir das coordenadas ‘s’ e ‘t’. Este endereço é que deve ser passado como atributo adicional ao vértice do modelo 3D para que o mesmo possa “colar” a imagem sobre a superfície do objeto 3D a ser renderizado. Por fim, quando as cores dos pixels são calculadas é que se faz efetivamente o acesso à textura.

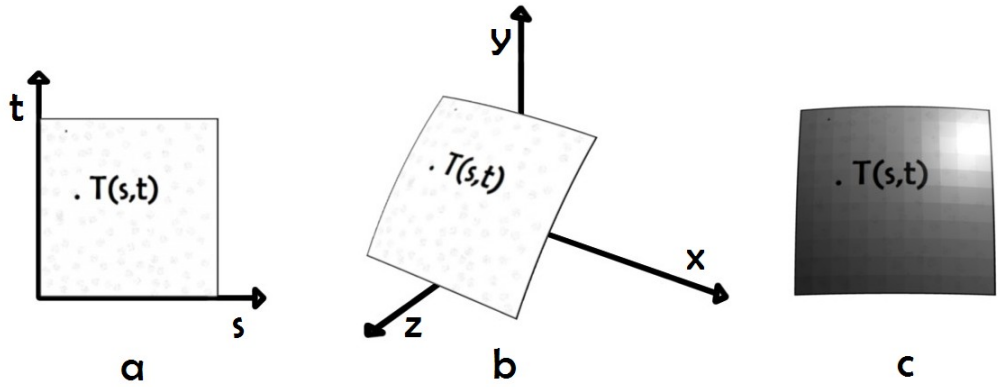
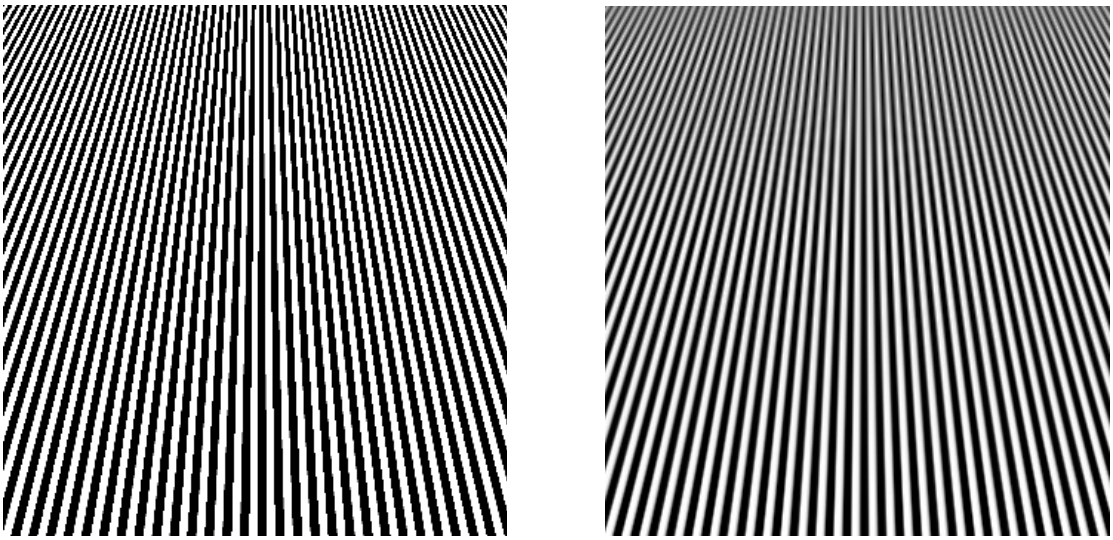


Figura 9: Mapeamento de textura. a) função textura ‘T’ que recebe as coordenadas ‘s’ e ‘t’ e endereçam uma determinada cor; b) objeto3D que possui um dado mapeamento para o espaço de ‘T’; c) imagem final que utiliza os texels de ‘a’ para atribuir cor aos pixels.

A Figura 10 mostra um exemplo de 2 configurações de sampler para a mesma textura sobre uma imagem de alta frequência (variação brusca de branco para preto). Note o filtro age de maneira diferente sobre os texels, é importante saber escolher a configuração certa destes filtros, pois a mesma afetar o desempenho da renderização.



a b

Figura 10: Exemplo da configuração de samplers. a) não utilizando nenhum filtro; b) utilizando filtro linear.

5.1.Multi-texturização

Várias texturas diferentes podem ser utilizadas dentro do shader e também vários elementos de uma mesma textura podem ser utilizados. Com base nisto é possível implementar a técnica de multitexturização.

Para implementar multitexturização com shaders é necessário definir uma função de combinação das cores das várias texturas utilizadas. Como pode ser visto na Equação 5, a multitexturização pode ser definida como uma combinação das cores de várias texturas.

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

Equação 5: Equação multitexturização.

A Equação 6 mostra uma relação de recorrência que pode representar um tipo de combinação de multitexturização que utiliza o canal alpha da textura como fator de combinação.

$$a(cor_{rgba}) = \ valor\ do\ canal\ alpha\ de\ uma\ cor_{rgba}$$
$$MT_{[n]} = T_{[n]} * a(T_{[n]}) + \ MT_{[n-1]} * \left(1 - a(T_{[n]})\right), n \geq 2$$
$$MT_{[1]} = T_{[1]}$$

Equação 6: Relação de recorrência para descrição de uma combinação de multitexturização. MT é a multi-textura, T é a textura e as operações são realizadas sobre valores em RGB em uma dada coordenada de acesso.

Este tipo de combinação pode ser utilizado para renderização de terrenos, onde podem existir várias texturas aplicadas simultaneamente a uma determinada região do mapa. Como exemplo, na Figura 11, uma textura de terreno(tex1) é combinada com a textura da estrada(tex2) e logo depois este resultado é combinado com a área não acessível do mapa(tex3).

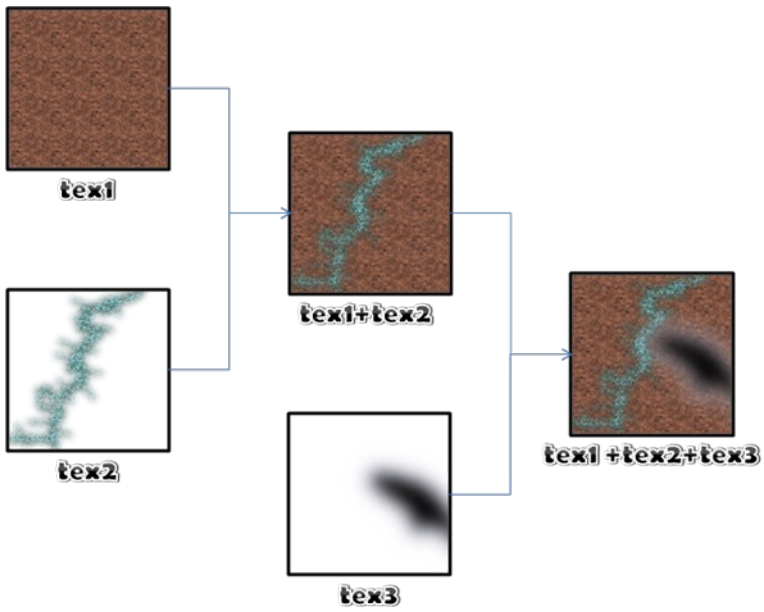


Figura 11: Diagrama da aplicação da recorrência de multitexturização sobre 3 texturas para composição de uma imagem de terreno.

Ao voltarmos na Equação 6, é possível notar que a operação de recorrência é uma combinação convexa da textura ‘n’ com a textura ‘n-1’ ponderada pelo fator do canal alpha, por isto é possível implementar esta relação no shader utilizando a função de interpolação linear (lerp). O código do processador de pixels é mostrado abaixo.

```
float4 std_PS(vertex2fragment IN) : COLOR {
    //leitura dos 3 texels
    float4 texel1 = tex2D(tex1sampler,IN.UV);
    float4 texel2 = tex2D(tex2sampler,IN.UV);
    float4 texel3 = tex2D(tex3sampler,IN.UV);
    //combinação das 3 texturas com base no canal alpha
    float3 result = lerp( texel1, texel2, texel2.a );
           result = lerp( result, texel3, texel3.a );
    return float4(result, texel1.a);
}
```

5.2.Geração Procedural

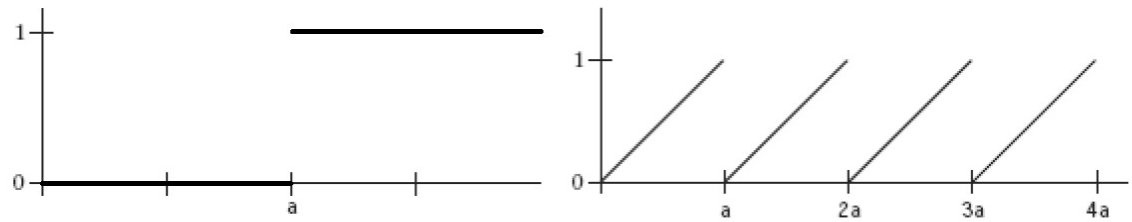
As texturas podem ser obtidas a partir de fotografias, desenhos ou de forma sintética. Estas geradas de forma sintéticas são conhecidas como texturas procedurais.

As texturas procedurais são aquelas que o processo de síntese da textura pode ser descrito de forma algorítmica. Ao utilizar shaders é possível implementar estas texturas de forma dinâmica, ou seja, não é necessário armazenar uma imagem, pois as cores da mesma podem ser geradas on-the-fly.

Um exemplo simples é a geração de uma textura que segue o padrão de uma onda quadrada, onde são utilizadas funções do shader para compor esta onda. No algoritmo de pixel abaixo é possível notar que é utilizada a função step e fmod.

```
float4 std_PS(vertexOutput IN) : COLOR {
    float result = IN.UV.x;
    result = result * freq ;
    result = fmod ( result , 1 );
    result = step ( result , 0.5 );
    return float4(result.xxx,1);
}
```

O comportamento das funções step e fmod é exemplificado na Figura 12.



step(s, a) fmod(s , a)

Figura 12: Comportamento das funções step e fmod para um dado parâmetro ‘a’.

Como pode ser visto no algoritmo acima, existe um parâmetro externo chamado ‘freq’ que é multiplicado ao valor da coordenada no eixo ‘s’ da textura, por esta razão é possível controlar a frequência de repetições da transição da onda quadrada apenas alterando este valor. A

Figura 13 mostra o exemplo da renderização de uma imagem com este padrão para os parâmetros: 1, 5, 15 e 30.

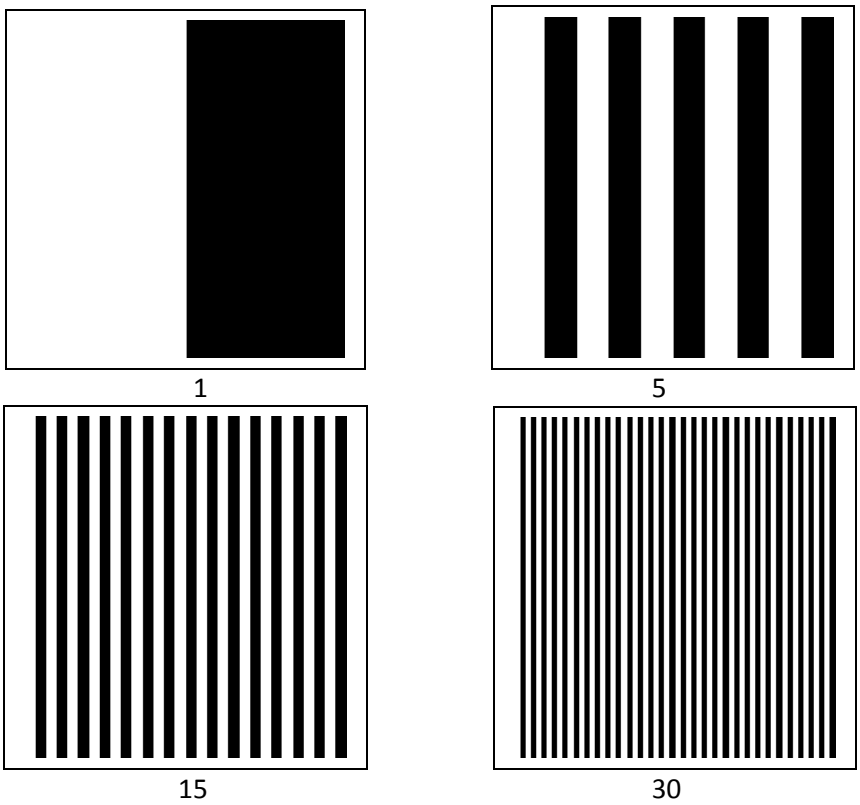


Figura 13: Renderização de textura de onda quadrada com frequências variadas.

Utilizar texturas procedurais é interessante, pois permite a implementação de controle paramétrico, o que faz um mesmo algoritmo ser capaz de gerar diversas texturas diferentes e como é necessário armazenar somente a configuração de parâmetros utilizados para sintetizar uma determinada textura, fornece também uma forma eficiente de compactação.

Porem, ao utilizar este tipo de textura diretamente sobre o shader, pode ocorrer os serrilhados na renderização, pois os filtros que antes eram aplicados nas texturas normais agora devem ser implementados também dentro da função de geração procedural. Outro ponto importante é o desempenho do shader que pode ser degradado com a implementação de algoritmos complexos. Por fim, o último problema é a falta de controle sobre os detalhes gerados, que apesar de não estarem presentes no exemplo simples que foi dado, ocorre em diversos modelos de geração procedural de texturas.

6.Renderização de superfícies detalhadas

Um dos fatores que determina o seu nível de realismo de um ambiente virtual é o nível de detalhe dos objetos presentes nesse ambiente. Objetos do mundo real podem ser representados computacionalmente de várias formas diferentes, sendo que a representação mais utilizada por aplicativos de computação gráfica é a representação B-rep (Boundary Representation). Nessa representação, um objeto é representado por sua borda (sua superfície mais externa). Essa superfície é dividida em um grande número de primitivas geométricas, geralmente triângulos, visando acelerar os cálculos necessários durante o processo de renderização. Dessa forma para armazenar um objeto do mundo real basta armazenar um arranjo de triângulos que formam a superfície desse objeto. Nesse tipo de

representação, o nível de detalhe dos objetos está diretamente relacionado ao número de triângulos utilizados para representar a superfície do mesmo. Quando maior o número de triângulos, mais subdividida é a superfície e mais detalhes podem ser armazenados.

Para representarmos com precisão a superfície de objetos do mundo real pode ser necessário trabalhar com um grande número de triângulos, que às vezes possui um número tão grande que não possa ser armazenado e processado em tempo real. A Figura 14 apresenta a renderização de um objeto do mundo real obtido a partir de um *scanner* 3D.

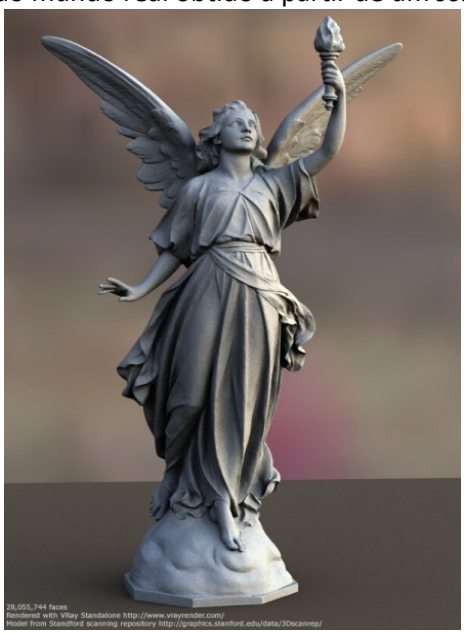


Figura 14: Lucy (Stanford University).

O modelo computacional utilizado na renderização da Figura 14 possui 116 milhões de triângulos e necessita de 325 MB de memória para ser armazenado. Para esse modelo poder ser renderizado em tempo real, seria necessário armazenar um grande volume de dados na GPU e processar todos esses dados pelo menos 24 vezes em um segundo.

Tentando solucionar o problema de desenhar superfícies extremamente detalhadas em tempo real, surgiram várias técnicas que tentam simular a renderização de uma superfície extremamente detalhada, sem a necessidade de se trabalhar com a superfície real dos objetos. Algumas dessas técnicas são: Bump Mapping [Blinn 78], Normal Mapping [Peercy 97], Parallax Mapping [Kaneko 01], Relief Mapping [Policarpo 05, Policarpo 06], Parallax Occlusion Mapping [Tatarckuk 06], Cone Step Mapping [Dummer 06] e Sphere Tracing [Donnelly 05]. A Figura 15 ilustra a aplicação de uma das primeiras técnicas propostas para simular detalhes na superfície, chamada Normal Mapping [Peercy 97].

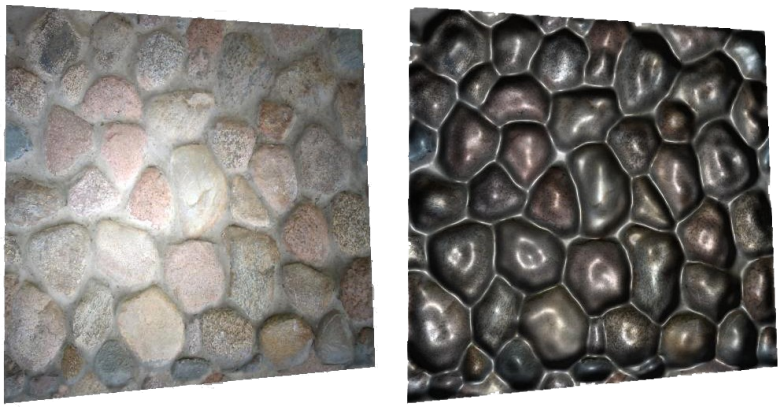


Figura 15: Comparação entre renderização de superfícies. (Esquerda) Mapeamento de textura. (Direita) Normal mapping.

Das técnicas existentes para simular a renderização de malhas detalhadas as mais recentes, como Relief Mapping, Parallax Occlusion Mapping, Cone Step Mapping e Sphere Tracing utilizam como base um algoritmo de ray-tracing. Nesses trabalhos o algoritmo de ray-tracing é executado no processamento de cada pixel gerado durante a renderização do objeto. Com isso é possível gerar imagens com uma alta qualidade visual, ao custo de um maior tempo de processamento. A Figura 16 ilustra a renderização de uma superfície utilizando a técnica de Cone Step Mapping (esquerda), e a renderização da superfície real (direita).

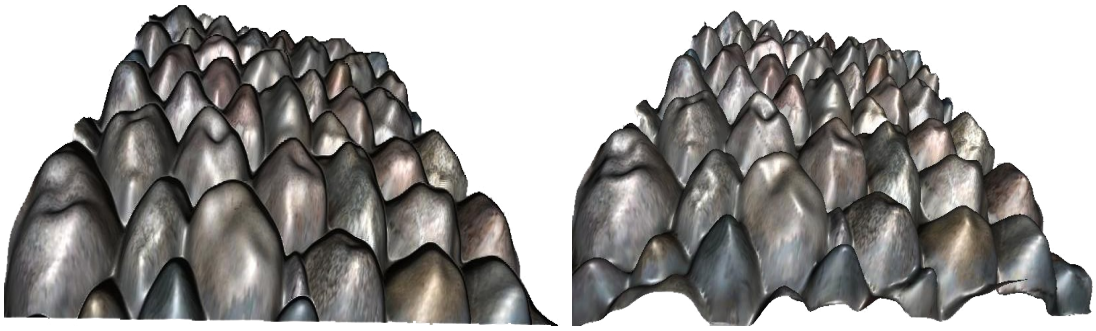


Figura 16: (Esquerda) Detalhes simulados utilizando a técnica de Cone Step Mapping. (Direita) Superfície Real.

6.1.Bump Mapping e Normal Mapping

As técnicas de Bump Mapping e Normal Mapping são utilizadas para simular detalhes nas superfícies dos objetos sem alterar a malha dos mesmos. Essas técnicas se baseiam na premissa que o calculo de iluminação em cada ponto da superfície é dependente do vetor normal da superfície naquele ponto. Com isso, para simular detalhes na superfície do objeto, basta aplicar perturbações sobre a normal do mesmo. A Figura 17 ilustra uma superfície lisa e suas normais sendo perturbadas por uma função de perturbação.

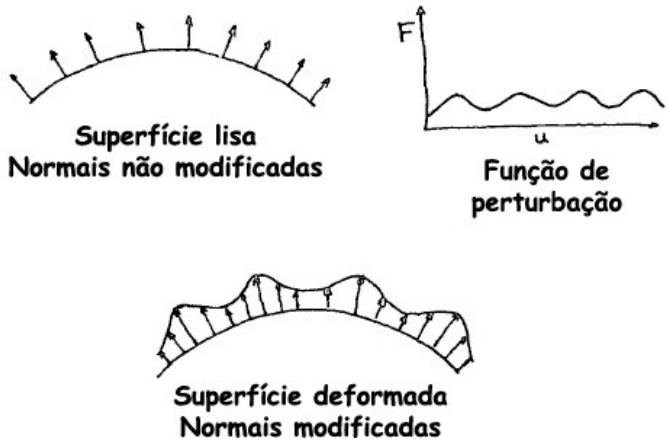


Figura 17: Perturbação das normais de uma superfície.

A técnica de Bump Mapping aplica perturbações nas normais do objeto baseado em uma função matemática ou em um mapa de curva de nível. Para aplicar a perturbação baseada em um mapa de curva de nível, para cada pixel é necessário ler a sua altura e a dos seus vizinhos do mapa de curva de nível e a partir disso calcular a nova normal para este ponto. Uma alternativa mais eficiente seria armazenar a normal perturbada para cada ponto da superfície, não precisando assim calcular a mesma, essa técnica é chamada de Normal Mapping. A Figura 18 mostra um mapa de curva de nível e um mapa de normais que podem ser utilizados respectivamente pelas técnicas de Bump Mapping e Normal Mapping.

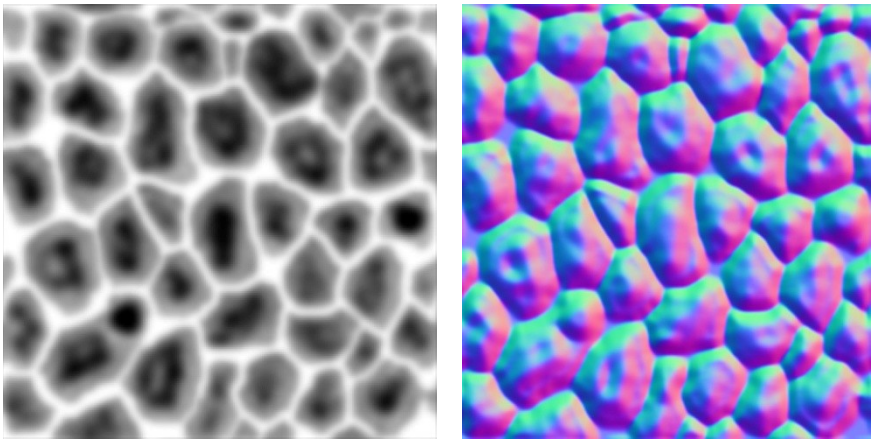


Figura 18: (Esquerda) Mapa de curva de nível. (Direita) Mapa de normais.

Quando utilizamos as técnicas de Bump Mapping ou Normal Mapping geralmente se opta por realizar todos os cálculos no espaço da tangente. O espaço da tangente é uma base formada pelos vetores tangente, bitangente e normal para cada ponto da superfície. Quando se trabalha no espaço da tangente não estamos mais considerando a curvatura da superfície do objeto ou sua orientação. Isso permite, por exemplo, que um mapa de normais gerado possa ser utilizado em qualquer parte da superfície do objeto. A Figura 19 exibe os vetores do espaço da tangente calculados para diferentes pontos em uma esfera e um torus.

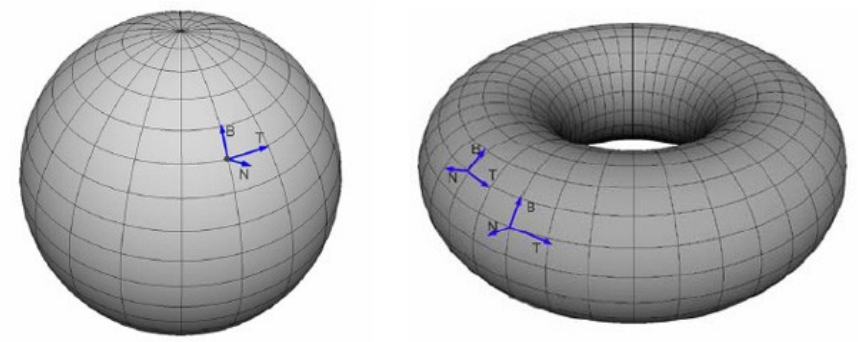


Figura 19: Calculo da base da tangente para diferentes pontos na superfície dos objetos.

Para se aplicar à técnica de Normal Mapping, primeiramente devemos calcular a base da tangente no processamento de vértice da superfície. E em seguida calcular a orientação dos vetores de visão e iluminação no sistema de coordenadas da tangente. No processamento de fragmentos, para cada fragmento devemos ler sua nova normal do mapa de normais da superfície, e em seguida, calcular a iluminação utilizando a normal lida e os vetores de visão e iluminação, calculados no processamento de vértices. Lembre que os atributos dos vértices calculados no processamento de vértices são interpolados para todos os fragmentos gerados durante o processo de rasterização. A seguir é apresentado o trecho do código de processamento de vértices, responsável em gerar a base da tangente e calcular as coordenadas dos vetores de visão e iluminação no espaço da tangente.

```
...

// 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;
```

A seguir é apresentado o código do processamento de pixels. O método phongShading executa o algoritmo de iluminação padrão de Phong. A Figura 20 apresenta o resultado da aplicação da técnica de Normal Mapping a um torus.

```
// 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);
```

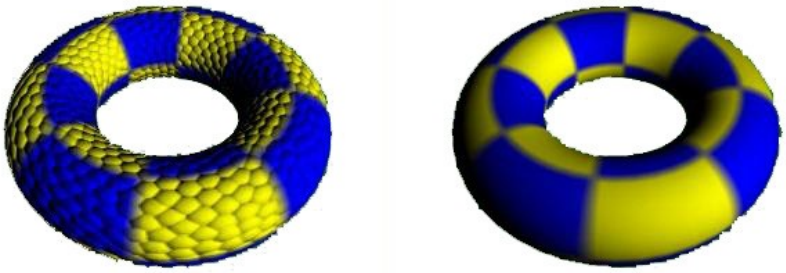


Figura 20: Aplicação da técnica de Normal Mapping. (Esquerda) Texturização e Normal Mapping. (Direita) Somente texturização.

7.Pós-processamento

As técnicas abordadas até agora são aplicadas diretamente sobre os triângulos dos modelos 3D renderizados. Existem outros tipos de efeitos especiais que podem ser aplicados sobre a imagem obtida a partir da renderização, eles são chamados de efeitos de pós-processamento.

Os shaders para pós-processamento geralmente utilizam uma ou mais texturas (imagens) como entrada para realçar ou extrair características das mesmas com o objetivo de gerar uma ou mais texturas. Esta definição está intimamente ligada com a área de processamento digital de imagens (PDI), só que as imagens são representadas por texturas. As técnicas apresentadas neste tutorial serão: transformações radiométricas, filtros e composições.

A Figura 21 mostra uma possível visão do funcionamento dos shaders de pós-processamento, considerando uma ou mais texturas de entrada e escrevendo o resultado em outras texturas. A gerência de texturas deve ser realizada dentro do jogo, pois é necessário utilizar a característica de renderização com escrita em textura da API em questão (OpenGL, DirectX ou XNA). Como estamos utilizando o FXComposer, o mesmo já implementa este controle, temos apenas determinar (no shader) quais texturas devem ser criadas ou escritas.

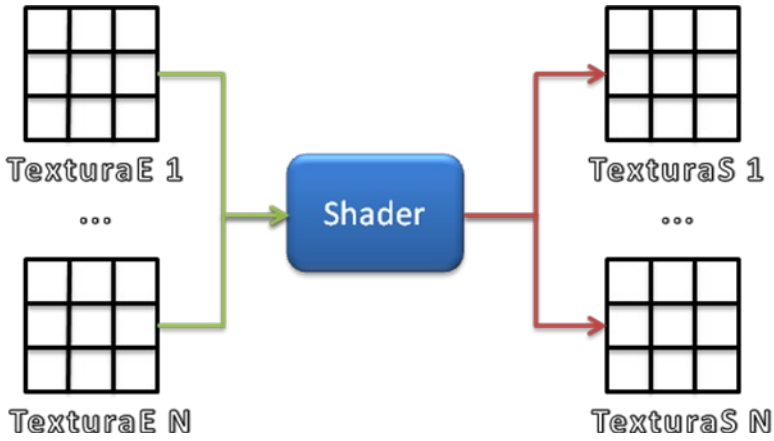


Figura 21: Funcionamento do shader de pós-processamento que recebe um conjunto de texturas como entrada e pode escrever em outro conjunto de texturas.

O código abaixo tem um exemplo da configuração para renderização com escrita em textura que é realizado.

```
float Script : STANDARDSGLOBAL <
    ...
    //indicar que este efeito é de pos-processamento
    string ScriptOrder = "postprocess";
> = 0.8;

//a técnica agora deve indicar pelo script qual
//            a ordem de execução de cada passo
technique Main < string Script =
                 "RenderColorTarget0=ScnMap;"
                 "RenderDepthStencilTarget=DepthBuffer;"
                     "ClearSetColor=ClearColor;"
                     "ClearSetDepth=ClearDepth;"
                     "Clear=Color;"
                     "Clear=Depth;"
                     "ScriptExternal=color;"
                 "Pass=Blur_Horz;"
                 "Pass=Blur_Vert;"
                 "Pass=BlendPass;";> {
    //para cada passo, deve se configurar as saidas de renderização
    pass Blur_Horz
        < string Script = "RenderColorTarget=texture1;"
                          "RenderDepthStencilTarget=DepthBuffer;"
                          "Draw=Buffer;"; > {
        ...
    }
    pass Blur_Vert
    ...
    //O último passo, atribui um valor vazio para a saida de
    //renderização, o que faz a imagem ser renderizada normalmente
    pass BlendPass
        < string Script= "RenderColorTarget0=;"
                         "RenderDepthStencilTarget=;"
                         "Draw=Buffer;";> {
        ...
    }
}
```

Na Figura 22 é mostrado um exemplo de renderização composta do efeito de bloom, onde existe apenas uma textura de entrada e esta é utilizada para compor a imagem final como efeito.

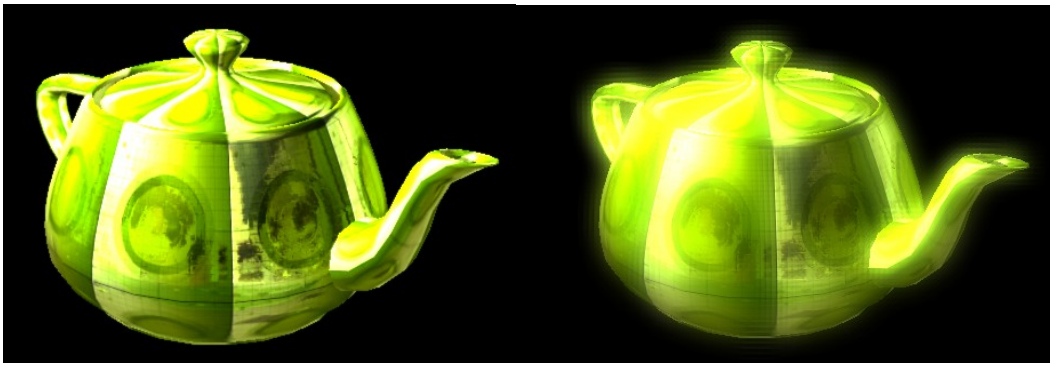


Figura 22: Esquerda: textura de entrada; Direita: textura que foi processada com o efeito de Bloom.

O desempenho do jogo pode ser fortemente influenciado pela utilização destas técnicas, pois os algoritmos são aplicados sobre os pixels da tela, ou seja, a carga de processamento gráfico está concentrada no fragment shader.

Muitos dos algoritmos que serão apresentados geralmente não são aplicados de forma isolada, pois os mesmos podem fazer parte de um processo elaborado de renderização de determinados efeitos.

7.1.Transformações radiométricas

As transformações radiométricas são operações realizadas sobre as cores da imagem através de um mapeamento direto, onde geralmente afetam o histograma da imagem. Estas transformações podem ser utilizadas para realçar características relacionadas com a distribuição de cores e podem ser implementadas através de operações aritméticas ou tabelas de mapeamento (LUT – LookUp Table). É possível implementar efeitos de transição da textura da renderização atual para uma determinada cor, controlar o seu brilho e contraste ou mesmo extrair os tons de cinza para utilizar como base para outro efeito.

Para realizar operações de brilho e contraste basta efetuar operações aritméticas como soma e multiplicação sobre os valores RGB da textura. A Equação 7 mostra um exemplo de operação de brilho e contraste.

$$f(x,y) = (f(x,y) - 0.5) * contraste + 0.5$$
$$f(x,y) = f(x,y) + brilho$$

Equação 7: Equações de contraste e brilho.

Na Figura 23 possui os gráficos de mapeamento de cores considerando apenas um canal de cor. A imagem original é uma reta, quando aplicamos a operação de contraste, a mesma é inclinada sobre o centro (0.5,0.5) e por fim no brilho a reta é apenas deslocada para cima ou para baixo.

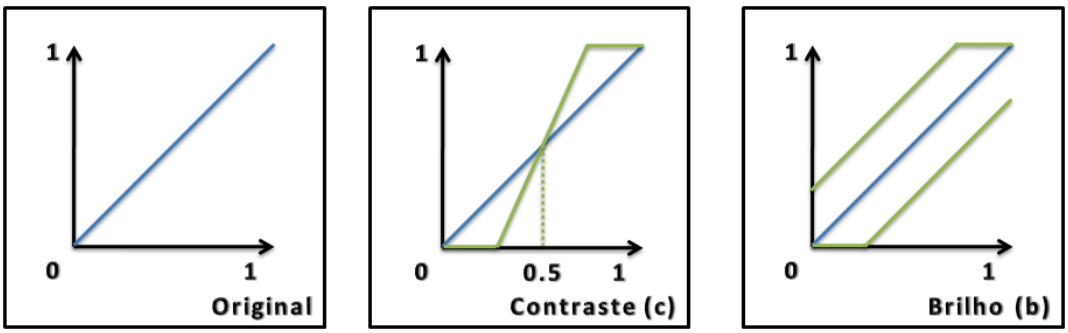


Figura 23: Exemplo das operações de contraste e brilho sobre o mapeamento de cores.

Os algoritmos para extração de tons de cinza estão relacionados ao espaço de cor utilizado, pois outros espaços de cor (não o RGB – Red Green Blue) podem representar a intensidade de cinza como um componente separado, que é o caso dos espaços HSV (Hue, Saturation, Value) e YIQ.

O espaço de cor HSV é definido como um cone invertido, onde o ângulo (Hue) em sua base define uma cor, a distancia do centro até a borda define a saturação (Saturation) e a profundidade é a intensidade (Value). O mesmo pode ser visualizado pela Figura 24.

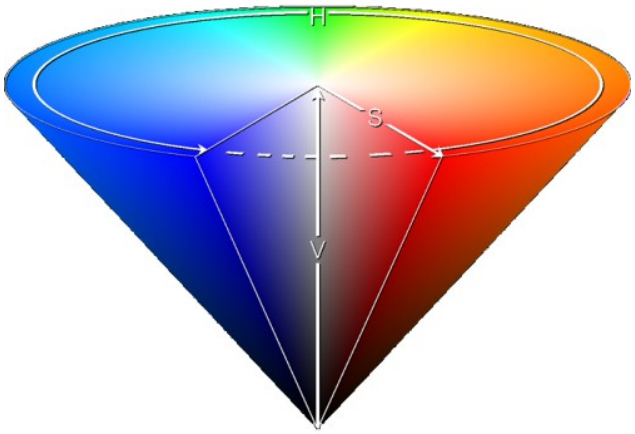


Figura 24: Representação do espaço de cor HSV em formato de cone.
Retirada de: http://en.wikipedia.org/wiki/Image:HSV_cone.png

O método de conversão do espaço RGB para o HSV é elaborado e pode ser encontrado em [Foley 97], mas como estamos interessados em obter somente o valor de 'V' para ser o nosso tom de cinza, é possível utilizar qualquer das duas equações da Equação 8.

$$Tom\ Cinza = \frac{r+g+b}{3}$$
$$Tom\ Cinza = (r,g,b)$$

Equação 8: Duas fórmulas para extração do tom de cinza.

Outro espaço de cor, o YIQ, é o mesmo utilizado no sistema de transmissão de televisão NSTC (National Television Systems Comittee). Onde o 'Y' representa a variação de intensidade do preto para o branco, o 'I' representa a variação do vermelho para o ciano e o 'Q' a variação de magenta para o verde. Este espaço de cor separa a informação de intensidade baseada na percepção do sistema visual humano, ou seja, os resultados visuais podem ser melhores que utilizar uma conversão direta para o espaço de cor HSV. A Figura 25 mostra o mapeamento de cores IQ em um espaço bidimensional.

Uma característica interessante do espaço de cor YIQ é que o mesmo separa a informação de cor da informação monocromática, o que permitiu a transmissão de um mesmo sinal para televisores coloridos ou não. A conversão do espaço RGB para o YIQ pode ser feita diretamente apenas aplicando uma multiplicação de matrizes, que pode ser observada na Equação 9.

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.595716 & -0.274453 & -0.321263 \\ 0.211456 & -0.522591 & 0.311135 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$
$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.9563 & 0.6210 \\ 1 & -0.2721 & -0.6474 \\ 1 & -1.1070 & +1.7046 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

Equação 9: Matrizes para conversão do espaço RBG para o YIQ e vice-versa.

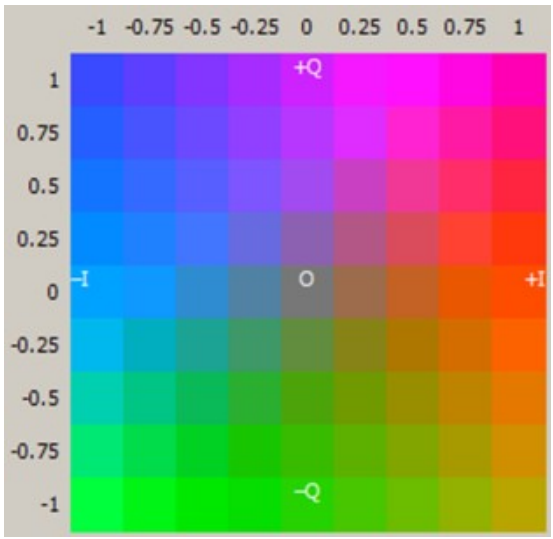


Figura 25: Representação dos componentes 'I' e 'Q' espaço de cor YIQ.
Retirada de: http://en.wikipedia.org/wiki/Image:YIQ_50%25Y_color_space.png

A extração de tons de cinza pode apresentar um bom resultado, mas é possível acrescentar uma melhora ao combinar uma cor a intensidade calculada. A Figura 26 mostra o resultado dos algoritmos de extração de tons de cinza.

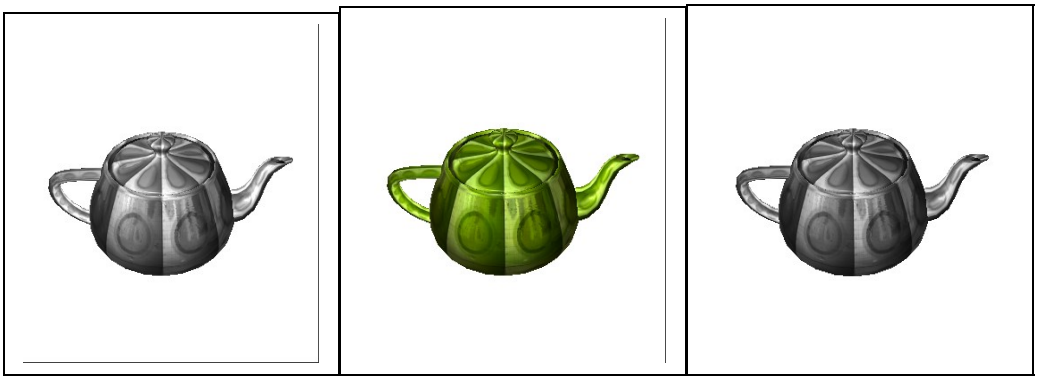


Figura 26: Tom de cinza utilizando o HSV; Imagem original; Tom de cinza utilizando YIQ.

.

7.2.Filtros

Os filtros são operações realizadas através da aplicação de máscaras (matriz de valores) sobre pixels. Geralmente são utilizados para extração ou realce de características relacionadas à informação que está presente na imagem com uma certa granularidade (definida pela área de cobertura da máscara). Diversos exemplos podem ser encontrados em "Digital Image Processing - Gonzalez, Woods".

A implementação de filtros utilizando shaders é possível porque os mesmos podem acessar vários texels de uma mesma textura. A Figura 27 mostra o exemplo da aplicação de um filtro de dimensão 3x3, onde para cada textel/pixel de destino são lidos 9 outros textels da entrada.

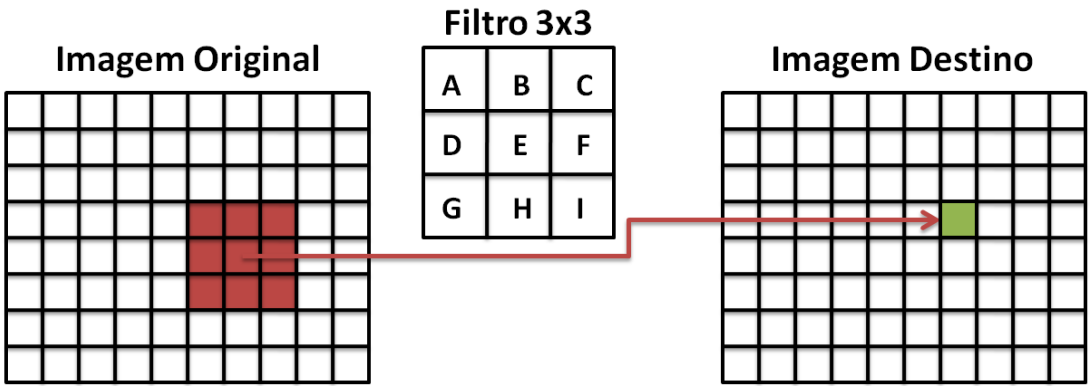


Figura 27: Exemplo de aplicação de um filtro de 3x3 genérico.

As máscaras não costumam serem grandes, mesmo porque a complexidade de aplicação de um filtro é determinada pela dimensão da máscara utilizada. Como o acesso à textura é uma operação cara, quanto menos à dimensão do filtro melhor será seu desempenho.

Os filtros podem ser utilizados para modificação de uma imagem, afetando a mesma no domínio da frequência, para implementar as operações de passa baixa e passa alta. Um exemplo de filtro de passa baixa bastante conhecido é o filtro da média, também conhecido como filtro de blur.

7.2.1.Blur

O filtro de blur é utilizado para suavizar a imagem, ou seja, remover as frequências altas entre cores vizinhas. Por este motivo também é conhecido como um filtro de passa baixa. Como a imagem após a aplicação deste aparenta estar embaçada, então esta pode ser aplicada quando a visão do personagem estiver prejudicada.

Uma aplicação simples deste filtro é realizar a soma de todos os elementos dentro da mascara e dar o mesmo peso para todos os elementos, um exemplo pode ser visto na Figura 28.

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

Figura 28: Filtro de blur em uma matriz de 3x3.

Outra forma de aplicar este filtro é utilizando pesos diferentes para os elementos dentro da matriz de forma a realçar a cor de uma dada região dentro do filtro. Geralmente é utilizada a distribuição Gaussiana para este fim.

No caso do filtro de blur, existe uma otimização possível para reduzir a complexidade de sua aplicação. Se a imagem for filtrada em um único eixo e sobre este resultado ser filtrada novamente no eixo perpendicular o resultado é semelhante à aplicação do filtro quadrado.

A Figura 29 mostra o processo de blur que funciona em dois passos. No primeiro o filtro é aplicado somente em um eixo e no segundo ele é reaplicado só que agora no eixo

perpendicular. Este método é interessante, pois gera um bom resultado além de diminuir a complexidade do algoritmo de $O(n^2)$ para $O(n)$.

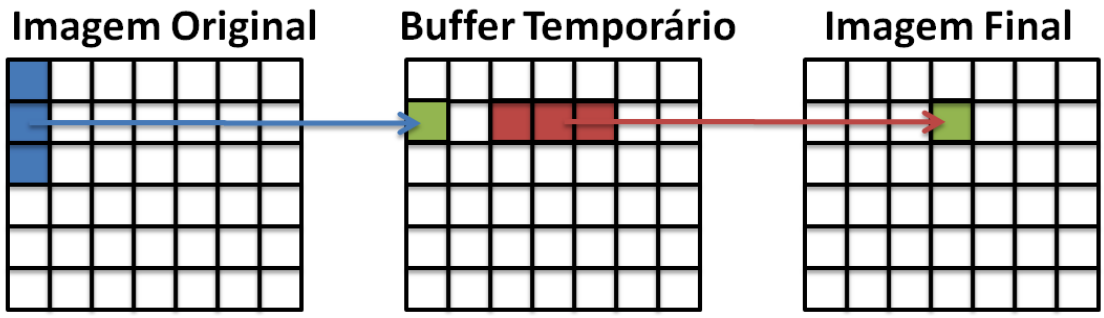


Figura 29: Diagrama do algoritmo otimizado de blur.

Uma característica interessante do blur é que a aplicação consecutiva deste filtro intensifica a suavização obtida, ou seja, ao invés de aumentar o número de acessos dentro do shader, pode se reaplicar o mesmo algoritmo com uma matriz pequena para atingir uma suavização considerável.

A Figura 30 mostra o fluxo de execução do shader de blur de dois passos a partir da textura do framebuffer, onde na área roxa o mesmo algoritmo de dois passos é executado várias vezes. No final a imagem gerada é mostrada como resultado da aplicação do filtro.

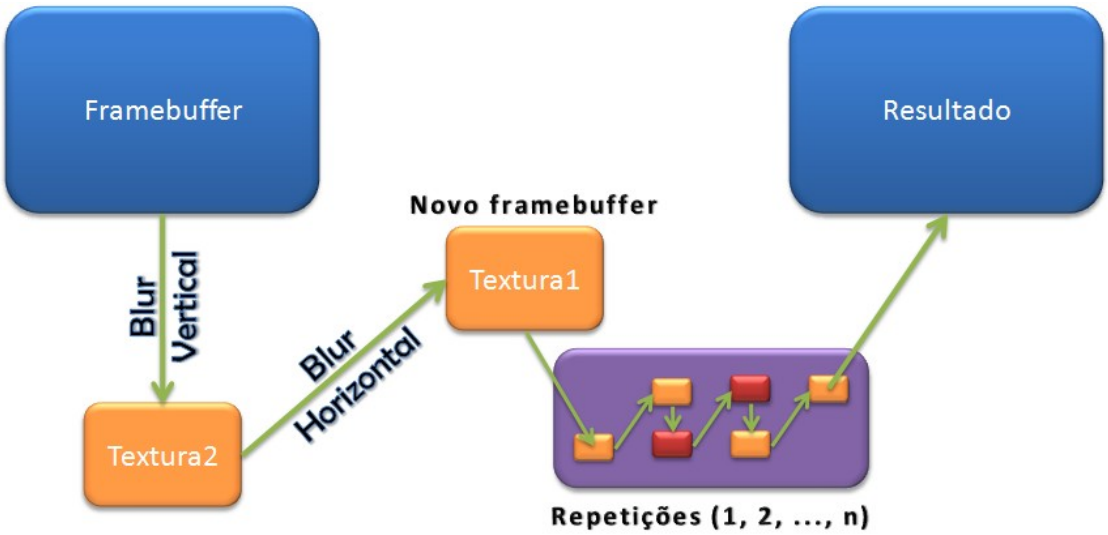


Figura 30: Algoritmo de blur com repetição.

8.Outros Efeitos

Vários efeitos que podem ser utilizados em jogos não se restringem somente à aplicação isolada de uma única técnica ou algoritmo. Alguns são combinações de diversos algoritmos, como exemplos têm o Bloom e o Cartoon Rendering.

É interessante ressaltar também que um efeito pode ser composto por uma parte de renderização da geometria (triângulos) com posterior acréscimo de alguma característica por pós-processamento. Nesta caso a estrutura de controle pode ficar complexa e necessitar de

várias texturas, por esta razão é necessário planejar a arquitetura de execução do efeito com cuidado.

8.1.Bloom

O efeito de bloom procura simular a expansão da luz ou reflexão que percebemos. O algoritmo é aplicado sobre uma imagem com ¼ da resolução da imagem e no final é realizada uma composição ponderada com o framebuffer.

O algoritmo pode ser dividido em três passos. No primeiro deve se gerar uma textura com ¼ do tamanho do framebuffer que possua informações sobre os locais que devem ser expandidos. Para isto pode se renderizar os objetos que emitem ou refletem luz ou pode se aplicar uma função de limiar sobre a renderização original. A textura com os tons claros deve ser suavizada no segundo passo, onde no nosso exemplo se utilizou filtro de blur. Por fim, no ultimo passo, a imagem suavizada deve ser combinada com o framebuffer. A Figura 31 mostra este fluxo de execução.

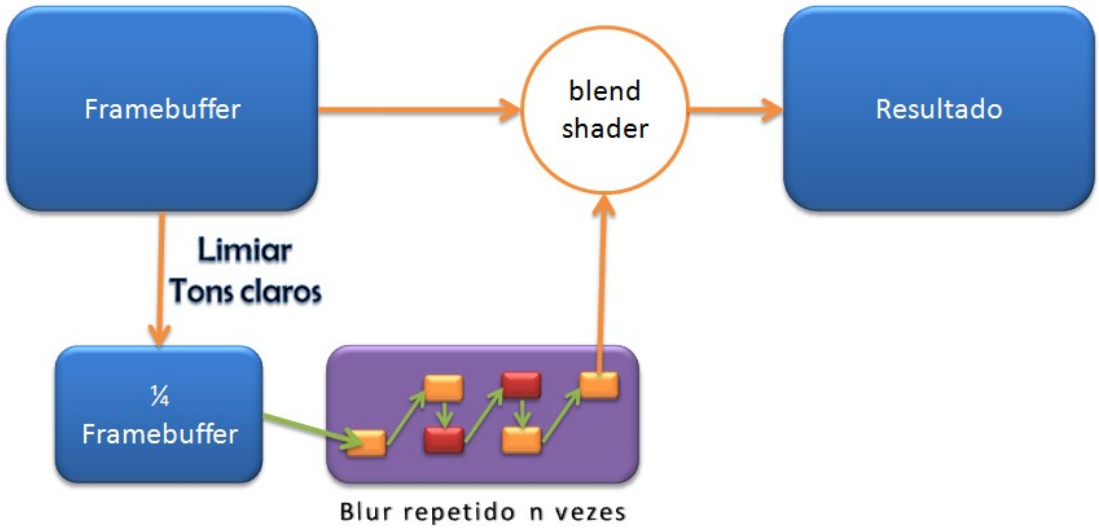


Figura 31: Algoritmo de Bloom.

8.2.Renderização de cartoon

Em oposição a técnicas que buscam o realismo estão as técnicas de NPR (Non-Photorealistic Rendering) como o sketch, gooch shading ou cartoon.

As técnicas de renderização de cartoon buscam dar a impressão da cena ser desenhada por um artista. Geralmente são utilizadas em desenhos técnicos ou jogos. A principal característica destas técnicas é a aparência uniforme das cores sobre a superfície dos objetos. A Figura 32 mostra um exemplo da renderização de um modelo 3D com cálculo de iluminação comum, logo após o resultado de uma técnica de cartoon com configuração de 1 tom e 3 tons.

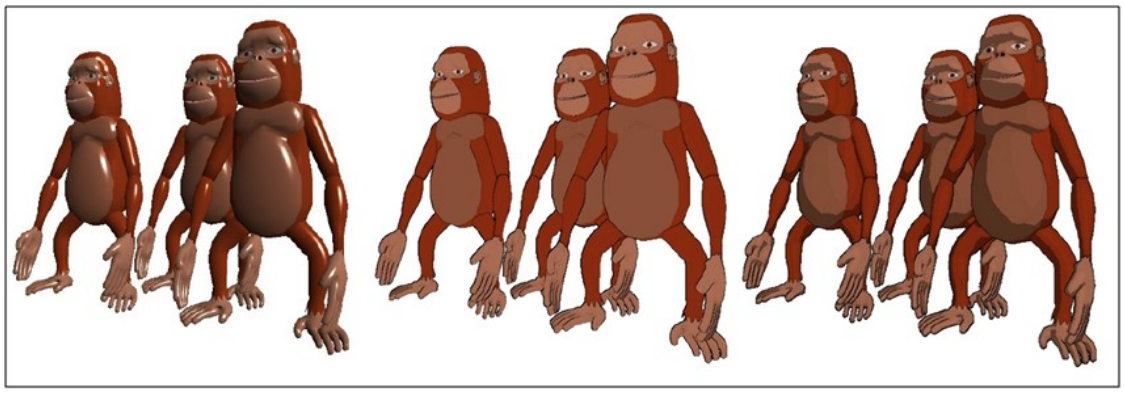


Figura 32: Exemplo de renderização de cartoon com aplicação de quantização de 1 tom e de 3 tons.

Para implementar a limitação de tons é realizada uma quantização sobre o resultado da aplicação da iluminação. Um exemplo de quantização pode ser observado na Figura 33, onde os tons do modelo de iluminação são mapeados para valores específicos, no caso do exemplo foram mapeados em 0.7 e 1.0.

Figura 33: Exemplo de quantização do modelo de iluminação e representação do mapeamento de cores.

Note que o na Figura 32 também são utilizadas bordas nos objetos, para realizar a detecção de bordas dos modelos pode-se utilizar um filtro ou realizar um processamento na geometria.

O algoritmo de renderização de cartoon pode ser dividido em três passos. No primeiro passo são extraídos os mapas de normais e profundidade e calculado o modelo de iluminação quantizado. No segundo passo são calculadas as bordas dos objetos utilizando os mapas de normais e profundidades com um filtro diferencial. No último passo a textura com os tons quantizados é combinada com a textura que contem as bordas dos objetos gerando a imagem final. Este processo pode ser visualizado na Figura 34.

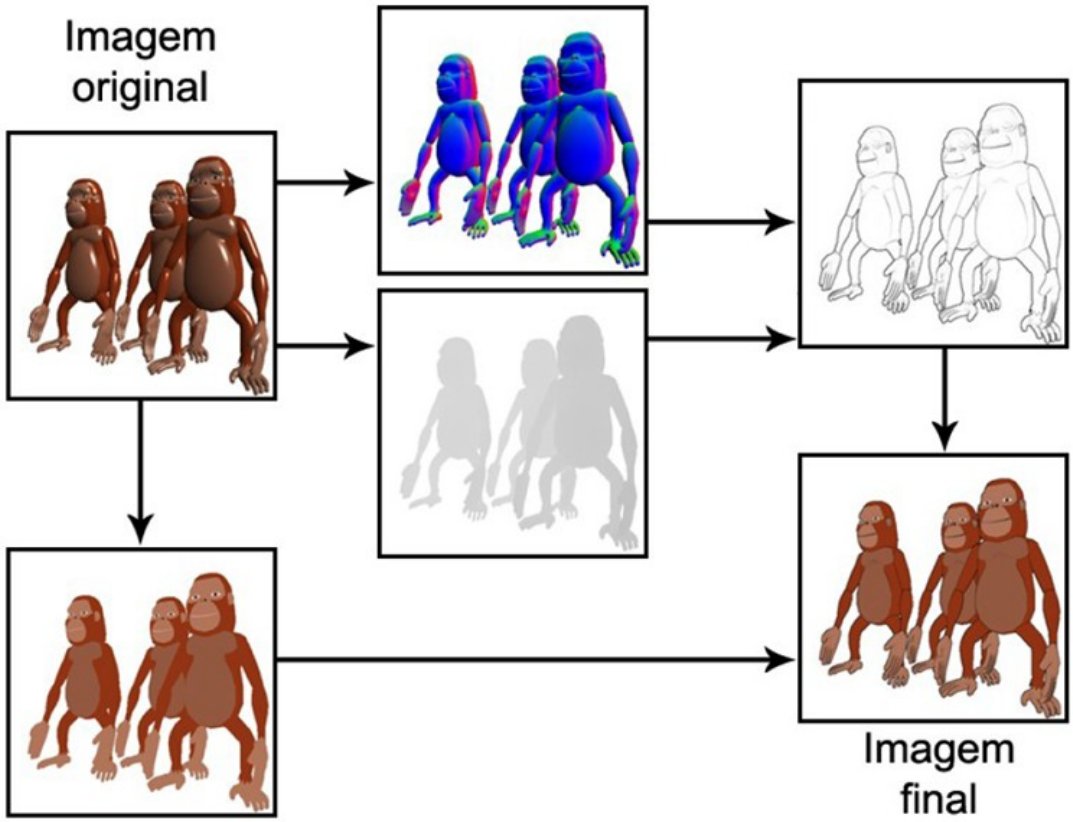


Figura 34: Diagrama do processo de cartoon rendering.

9.Conclusão

Neste trabalho foram apresentadas diversas técnicas para a criação de efeitos fotorealistas e não-fotorealistas pra jogos.

Dentre as várias técnicas apresentadas nesse trabalho, podemos destacar:

- Construção de um shader de iluminação por pixel (Seção 4)
- Multi-texturização e geração de texturas procedurais (Seção 5)
- Renderização de superfícies detalhadas (Seção 6)
- Efeitos de pós-processamento de cena (Seção 7)
- Bloom e Cartoon Rendering (Seção 8)

Todos os efeitos apresentados foram criados com shaders, e a linguagem de shaders utilizada foi a HLSL. Esses efeitos podem ser utilizados em aplicações de tempo real, como jogos, simuladores e ambientes virtuais. Além disso, como os efeitos são implementados na GPU a integração desses efeitos em ambientes virtuais existentes pode ser muito simples.

Com a rápida evolução das GPUs a mesma deve se tornar completamente programável nos próximos anos, o que tornará possível que mais efeitos sejam criados e utilizados em aplicações de tempo real, em especial jogos!

10.Referências

[Blinn 77] Blinn, J., Models of light reflection for computer synthesized pictures, Proceedings of the 4th annual conference on Computer graphics and interactive techniques, p.192-198, 1977.

[Blinn 78] BLINN, J. F. 1978. Simulation of wrinkled surfaces. In Proceedings of the 5th annual conference on Computer graphics and interactive techniques, ACM Press, 286–292.

[Blythe 06] Blythe, D., The Direct3D 10 system. In proceedings of SIGGRAPH 2006, 724-734.

[Cook 84] COOK, R. L. 1984. Shade trees. In Proceedings of the 11th annual conference on Computer graphics and interactive techniques, ACM Press, 223–231.

[Donnelly 05] Donnelly, W. 2005. GPU Gems 2. Addison-Wesley, March, ch. Per-Pixel Displacement Mapping with Distance Functions, 123--136.

[Dummer 06] DUMMER, J. Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm. [S.l.], 2006.

[Foley 97] Foley, J., van Dam, A., Feiner, S., and Hughes, J. Computer Graphics: Principles and Practice, 2nd edition. Addison Wesley, 1997.

[Hanrahan 90] HANRAHAN, P., and LAWSON, J. 1990. A language for shading and lighting calculations. In proceedings of SIGGRAPH 90, 289-298.

[Kaneko 01] KANEKO, T., TAKAHEI, T., INAMI, M., KAWAKAMI, N., YANAGIDA, Y., MAEDA, T., AND TACHI, S. 2001. Detailed shape representation with parallax mapping. In Proceedings of the ICAT 2001, 205–208.

[Möller 99] Möller, T., and Haines, E., Real-Time Rendering. AK Peters, 1999.

[Peercy 97] PEERCY, M., AIREY, J., AND CABRAL, B. 1997. Efficient bump mapping hardware. In SIGGRAPH ’97, 303–306.

[Policarpo 05] POLICARPO, F.; OLIVEIRA, M. M.; COMBA, J. L. D. Real-time relief mapping on arbitrary polygonal surfaces. I3D, 2005.

[Policarpo 06] Policarpo, F., Oliveira, M. M. 2006. Relief Mapping of Non-Height-Field Surface Details. In ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games Proceedings, ACM Press, pp. 55-52.

[Tatarckuk 06] TATARCHUK, N. Dynamic parallax occlusion mapping with approximate soft shadows. In: SI3D ’06: Proceedings of the 2006 symposium on Interactive 3D graphics and games. New York, NY, USA: ACM Press, 2006. p. 63–69. ISBN 1-59593-295-X.