Desmistificando Shaders em XNA 4.0

Bruno Duarte Correa
Thiago Dias Pastor
Department of Computer and Digital Systems Engineering, Escola Politécnica da Universidade de São Paulo, Brazil

1 Introdução

O uso de efeitos especiais em jogos está cada vez mais recorrente, entretanto ainda existe uma grande relutância das pessoas em usá-lo ou mesmo ainda existe a idéia de que exige um grau de conhecimento muito alto para desenvolver shaders.O objetivo desse tutorial é desmistificar essa idéia e mostrar que com pouco esforço é possível atingir resultados muito satisfatórios dando ao jogo um viés mais profissional. Mostraremos a parte teórica e como utilizar shaders com XNA além de mostrar alguns exemplos muito utilizados em jogos atuais.

Keywords:: Computer graphics,Shaders,Electronic games.

Author's Contact:

{bruno.duartec,thiagodiaspastor}@gmail.com

2 Motivação

Os efeitos visuais são cada vez mais recorrentes nos jogos atuais, sendo ainda assim um grande diferencial e por que não dizer um requisito básico para um jogo de sucesso. O tutorial visa demonstrar conceitos básicos de post effects tentando desmistificar a imagem comum sobre o assunto e demonstrar alguns exemplos.

3 Conceitos Básico

Antes de entendermos de fato o que é um shader alguns conceitos são de suma importância pois são bases para todo o conhecimento subsequente

3.1 Modelos

Um modelo é armazenado como uma lista de índices e uma lista de vértices conforme a figura a seguir: (em jogos normalmente usamos esta representação por ser compacta):

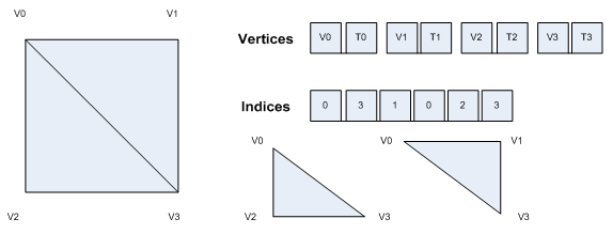


Figure 1: *Representação de um modelo 3D*

O modelo (quadrado a esquerda) pode ser descrito por meio de duas listas, a primeira contendo atributos de vértice (como Posições V e Texturas T) e a segunda lista contendo os índices (posição na lista dos vértices) que formarão cada triângulo do modelo (0,3,1 e 0,2,3).

Notar que vértice não significa posição, vértice contém posição e mais outros atributos, neste modelo (quadrado acima), cada vértice tem uma coordenada de textura e uma posição. Atributos comuns são: normais, cor, tangentes, índices de ossos e peso de ossos, sendo os dois últimos usados em Bone Animation.

3.2 Enviando Modelos para a GPU

No XNA e no DirectX, as informações de vértices são armazenadas em um container chamado VertexBuffer. Esta classe representa um elo entre a CPU e a GPU. Ao criar um vertexBuffer estamos na verdade criando uma espécie de proxy do lado da CPU para dados que serão armazenados na GPU. (É comum dizer "lado da CPU" e "lado da GPU", mas na verdade estamos querendo dizer, RAM da CPU e RAM da GPU).

Em um jogo, é bastante comum termos objetos (caixa, casa, parede ...) com formatos de VertexBuffers distintos (por exemplo, um vértice de um sistema de partículas tem atributos como tempo de vida da partícula e um vértice de um modelo simples não tem este atributo), desta forma precisamos informar para a GPU o FORMATO correto do dado contido no VertexBuffer.

Para esta finalidade, existe uma estrutura chamada VertexDeclaration (XNA e Directx) que descreve cada um dos atributos dos vértices de um VertexBuffer.

Antes de criar o VertexDeclaration, devemos criar o nosso vértice de fato. Como exemplo, criarei um tipo de vértice chamado CUSTOMVertexPositionNormalTexture que conterá informações sobre Posição, Normal e Textura do modelo. O XNA oferece uma versão chamada VertexPositionNormalTexture praticamente idêntica a esta.

A definição do vértice é a seguinte:

```
struct CUSTOMVertexPositionNormalTexture
{
        public Vector3 position; ///
            informacoes sobre posicao
        public Vector3 normal;   ///
            informacoes sobre normal
        public Vector2 texcoord; ///
            informacoes sobre coordenadas de
             textura
}
```

A seguir devemos criar o VertexDeclaration para este tipo de vértice. O construtor do VertexDeclaration recebe um array de VertexElement. Cada VertexElement descreve um dos atributos do vértice. A seguir temos a definição dos VertexElements para o vértice criado:

```
VertexElement positionDeclaration = new
    VertexElement(0, VertexElementFormat.
    Vector3, VertexElementUsage.Position,0);
VertexElement normalDeclaration = new
    VertexElement(sizeof(float) * 3,
    VertexElementFormat.Vector3,
    VertexElementUsage.Normal,0);
VertexElement textCoordDeclaration = new
    VertexElement(sizeof(float) * 6,
    VertexElementFormat.Vector2,
    VertexElementUsage.TextureCoordinate,0);
VertexElement[] vertexElements = new
    VertexElement[] {positionDeclaration,
    normalDeclaration, textCoordDeclaration};
VertexDeclaration vd = new VertexDeclaration
    (vertexElements);
```

Vamos por partes. O construtor do VertexElement recebe 4 parâmetros cuja descrição é a seguinte:

Offset: Distancia em bytes entre o começo do vértice e o atributo em questão elementFormat: Tamanho do atributo em questão (XNA tem um enum que facilita as coisas, o DirectX não ...) elementUsage: Uso pretendido do element, neste campo colocamos a semântica que será utilizada pelos Shaders para acessar este dado. usageIndex: Índice utilizado para acessar o elemento no Shader (Falarei deles mais a frente) Como exemplo descreverei a criação do VertexElement para o atributo coordenada de textura:

```
VertexElement textCoordDeclaration = new
    VertexElement(sizeof(float) * 6,
    VertexElementFormat.Vector2,
    VertexElementUsage.TextureCoordinate,0);
```

O primeiro parâmetro é o offset, neste caso usamos sizeof(float) * 6, pois existem 2 atributos Vector3 (Posição e Normal) antes da coordenada de textuta (veja definição do CUSTOMVertexPositionNormalTexture) e cada Vector3 é constituído de 3 floats.

O segundo parâmetro diz qual o tamanho do dado usado (Vector2), o XNA fornece um Enum que ajuda a definir este parâmetro. Em DirectX, não existe esta facilidade, colocaríamos algo do tipo sizeof(float) * 2

O terceiro e o quarto parâmetro descrevem o uso que daremos a este elemento no shader. Esta parte será explicada mais a frente.

Em seguida os VertexElements são empacotados em um array e usados para construir o VertexDeclaration. Com ele em mãos, a GPU poderá "entender" os dados dos vértices que estão sendo enviados.

No DirectX e no XNA 3.1, o VertexBuffer e o VertexDeclaration são entidades completamente separadas, porém no XNA 4.0 o VertexDeclaration virou um membro do VertexBuffer (isto facilita um pouquinho na hora de desenhar um modelo).

Para criar o vertexBuffer usamos:

```
VertexBuffer vb = new VertexBuffer(
    GraphicsDevice, vertexDeclaration,
    NUMERO_DE_VERTICES, BufferUsage.None);
vb.SetData<customvertexpositionnormaltexture
    >(ARRAY_DE_VERTICES, 0,
    NUMERO_DE_VERTICES);
```

A linha vb.SetData(...) corresponde ao momento em que os vértices do modelo estão sendo enviados para a RAM da GPU. (Este processo não é síncrono, ou seja, a transferência não acontece necessariamente no momento que chamamos esta função).

Após configurar o VertexBuffer, devemos enviar as informações de índices para que a GPU consiga criar cada um dos triângulos do modelo. Para isto criamos um IndexBuffer.

```
short[] indices = new short[
    NUMERO_DE_INDICES];
///inicia o array de indices
IndexBuffer ib = new IndexBuffer(
    GraphicsDevice, IndexElementSize.
    SixteenBits, NUMERO_DE_INDICES,
    BufferUsage.None);
ib.SetData<short>(INDICES);
```

INDICES é um array de shorts contendo os índices do modelo. (para modelos com muitos vértices usamos int ao invés de short, neste caso deveríamos usar IndexElementSize.ThirtyTwoBits no construtor do IndexBuffer)

4 Pipeline

A GPU é um processador SIMD (Single instruction Multiple Data) cuja função (entre outras) é auxiliar no processo de renderização (ver também GPGPU). Para isto, a GPU usa uma arquitetura extremamente especializada baseada massivamente em pipeline (um pipeline de um processador Pentium 4 tem algo em torno de 10 estágios, uma GPU costuma ter algo próximo de 1000).

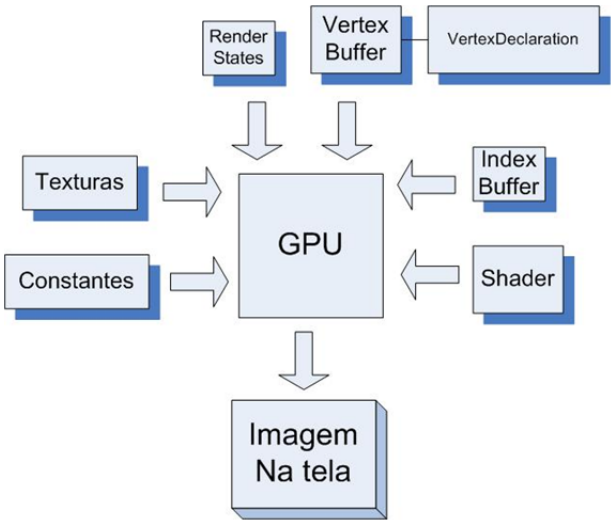


Figure 2: *troca de informações entre a CPU e a GPU*

Onde:

VertexBuffer,IndexBuffer e VertexDeclaration: Dados geométricos do modelo Render States: São parâmetros que determinam como que as etapas configuráveis da GPU funcionarão. Shader: Código que comandará o funcionamento das etapas programáveis da GPU. Texturas e Constantes: São parâmetros constantes que o shader atual acessará. Eles são de somente leitura.

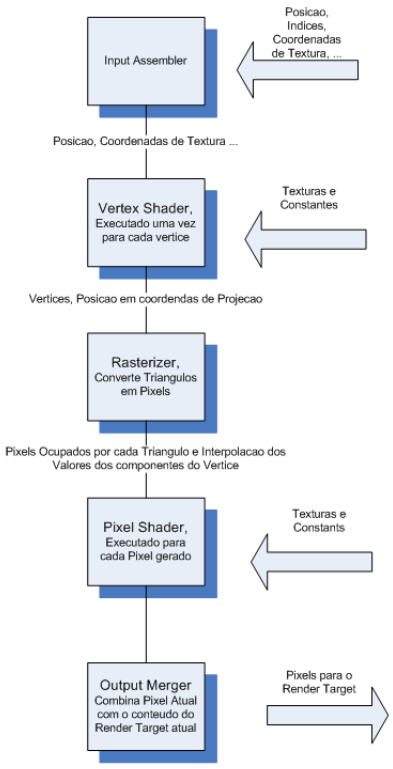


Figure 3: *Processos de troca de informaçoes*

Descrição:

- Input Assembler: Ler e interpretar os vértices e atributos de vértices do Vertex Buffer (por meio do VertexDeclaration e do IndexBuffer) e enviá-los para o VertexShader.
- Vertex Shader: Executado uma vez para cada vértice de cada triângulo. Sua função principal é converter os vértices para espaço de projeção.
- Rasterizer: Converte os triângulos (A GPU suporta diversas outras primitivas) em pixels e envia-os para o Pixel Shader.

O rasterizador também realiza outras tarefas como clipping e interpolação dos atributos do vértice para cada pixel.

- Pixel Shader : Determina a cor final do pixel a ser escrito no framebuffer (num primeiro momento pode ser entendido como a tela do monitor), é executado uma vez para cada pixel rasterizado de cada primitiva.
- Output Merger: Combina a saída do Pixel Shader com os valores do Render Target atual. Pode efetuar algumas operações como alpha blending e depth/stencil/alpha testings. (Neste tutorial usarei Render Target como um sinônimo para framebuffer, mas na verdade framebuffer é um tipo de Render Target)

5 Shaders

Como mostrado no item anterior, as GPU são processadores especializados em cálculos vetoriais, e como todo processador possui uma linguagem que nos permita programá-lo. Os shader são instruções de máquina (zeros e uns) que rodam na GPU. Nó inicio (Shader Model 1.0 – DirectX 8) o era necessário programar os shaders em ASSEMBLY o que demandava uma expecialização muito alta e um tempo de desenvolvimento muito alto. Devido a esses problemas de desenvolvimento em pouco tempo surgiram linguagens de mais alto nível como por exemplo o HLSL e a GLSL.

A Microsoft e o XNA usam a HLSL (High Level Shader Language), portanto nos prenderemos a esta linguagem. A HLSL foi criada para ser familiar aos programadores acostumados com C. A sintaxe é bastante clara e fácil de aprender.

A GPU (Shader Model 3.0 ou inferior) tem apenas dois estágios programáveis

5.1 Vertex Shader

- **Entrada**: Vértices
- **Saída**: Vértices, cujo atributo posição deve estar em espaço de projeção.
- **Quando que é chamado**: Uma vez para cada vértice de cada triangulo.
- **Função**: Sua função (mínima) é receber os vértices, e converter o atributo posição para o espaço de projeção.

Em HLSL, a entrada do Vertex Shader é especificada por uma struct:

```
struct VertexShaderInput
{
    float4 Position : POSITION0;
    float2 TexCoord : TEXCOORD0;
};
```

Em HLSL a saída do Vertex Shader também é uma struct conforme declarado a seguir:

```
struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float2 TexCoord : TEXCOORD0;
};
```

Assim como no struct de entrada, temos que declara a semântica que será transmitida pela GPU, entretanto a única restrição é que sempre deve existir um POSITION com índice 0. No caso do exemplo acima, teremos as informações de posição e textura para a próxima etapa. Com uma análise mais profunda podemos encarar as semânticas como registradores internos da GPU. Agora vamos ao código do Vertex Shader de fato:

```
VertexShaderOutput VertexShaderFunction(
    VertexShaderInput input)
{
    VertexShaderOutput output;
    ///Transforma o vertice em questao de
        coordenadas locais para de Mundo
    float4 worldPosition = mul(input.
        Position, World);
    ///Coordenadas de Mundo para de Camera
    float4 viewPosition = mul(worldPosition,
        View);
    ///Coordenadas de camera para de
        projecao
    output.Position = mul(viewPosition,
        Projection);
    ///repassa a coordenada de textura para
        frente
    output.TexCoord = input.TexCoord;
    return output;
}
```

Em HSLS o vertex shader é uma função (que pode ter qualquer nome, mais a frente diremos para a GPU quem é quem) que recebe uma estrutura como entrada e outra como saída.

Neste exemplo estamos simplesmente convertendo a posição dos vértices para espaço de projeção e repassando as coordenadas de textura sem alterar seu valor.

As matrizes World, View e Projection que aparecem no código já foram apresentadas e são chamadas de constantes, parâmetros imutáveis somente leitura que podem ser acessados do Vertex e do Pixel Shader.

As constantes são parecidas com variáveis globais em C, elas são declaradas em hlsl da seguinte maneira:

```
float4x4 World;
float4x4 View;
float4x4 Projection;
```

Float4x4 é uma matrix de transformação genérica com 16 elementos do tipo float (column-major dentro do shader, apesar da classe MATRIX do xna ser row-major). Podemos ter constantes de qualquer tipo base do hlsl. Mostraremos como definir o valor destas constantes mais a frente.

O Vertex Shader não tem acesso a nenhum outro vértice do modelo (nem os que compartilham o triângulo que está sendo processado no momento). O próximo passo, após ser executado o VertexShader quem toma vez é o Rasterizador, que remonta os triângulos a partir dos vertices processados pelo VertexShader

5.2 Rasterizador

O Rasterizador irá converter os triângulos que saíram do Vertex Shader em pixels na tela, além disto, ele irá interpolar todos os atributos do VertexShaderOutput para todos os pixel gerados. Conforme mostra a figura a seguir:

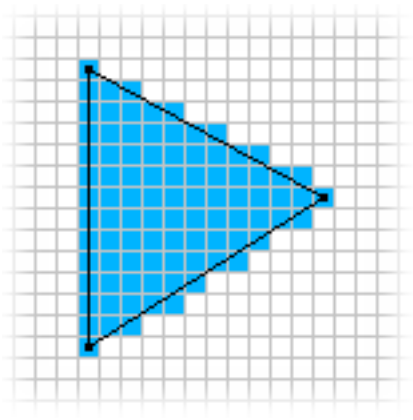


Figure 4: *Interpolação criada pelo rasterizador*

A partir de 3 vértices de um triângulo são gerados todos pixels que o compõe (quadrados azuis). Os atributos de cada pixel gerado serão um interpolação entre os atributos dos 3 vértices originais do triângulo. Por exemplo a coordenada de textura de um pixel que esta localizado exatamente no baricentro do triângulo será uma media simples entre as coordenadas de textura dos três vértices do triângulo. (A media simples é válida neste caso, pois supus que a rasterização não usa perspective correction, algo bastante irreal, porém facilita na hora de entender esta interpolação)

Para cada pixel gerado de cada triângulo, o Pixel Shader é chamado.

O Xna expõe alguns parâmetros configurávies para o Rasterizador como o Culling Mode e o FillMode

5.3 Pixel Shader

- **Entrada**: A mesma estrutura saída do Vertex Shader, porém o atributo com a semântica Position NÃO será acessível no Pixel Shader (ele é obrigatório na saída do Vertex Shader e invisível no Pixel Shader). No nosso exemplo poderemos acessar apenas a coordenada de textura.
- **Saída**: Cor do pixel como um float4 (RGBA) em caso de render target único (nosso caso)
- **Quando é chamado**: Uma vez para cada pixel gerado de cada triângulo rasterizado
-

Nosso pixel shader será definido da seguinte maneira:

```
float4 PixelShaderFunction(
    VertexShaderOutput input) : COLOR0
{
   return tex2D(diffuseSampler,input.
       TexCoord);
}
```

Uma função que recebe como entrada a saída do Vertex Shader (agora como atributo de pixel e não de vértice) e tem como saída um float4 (que tem a semântica Color0, em um uso mais avançado poderemos ter uma estrutura como saída também).

Este exemplo usa uma função do próprio HLSL chamada text2D para gerar a cor do pixel. Esta função recebe um Sampler e um coordenada de textura como parâmetro e devolve o texel correspondente.

Um Sampler é uma espécie de container com informações sobre como acessar uma textura. Nosso Sampler foi definido da seguinte maneira. (global como as constantes)

```
texture diffuseTexture;
sampler diffuseSampler = sampler_state
{
    Texture = (diffuseTexture); //textura
    AddressU = CLAMP;
    AddressV = CLAMP;
    MagFilter = LINEAR;
    MinFilter = LINEAR;
    Mipfilter = LINEAR;
};
```

A linha

```
texture diffuseTexture;
```

declara a textura. Este campo é tratado como uma Constante (é acessível pela CPU, usado para enviar a textura).

Em seguida temos a definição do Sampler. O primeiro parâmetro é o nome da textura, os outros especificam como que a leitura da textura deve ser feita.

5.4 OutputMerger

Sua função é combinar os pixel que saem do Pixel Shader com aqueles que estão no frameBuffer. Este módulo é bastante configurável (através dos Render states). Os principais parâmetros que podem ser alterados são:

- **Depth Test**: Teste de profundidade. A placa de vídeo mantém um buffer chamado DepthBuffer em que são guardados as distancias (Z Depth) entre a câmera e o "ponto 3D" que originou o pixel desenhado(essa distância é a coordenada Z em espaço de projeção interpolada). Quando um novo pixel chega do Pixel Shader, antes de escrevê-lo no framebuffer, a GPU verifica a distância (Z Depth) deste pixel com a armazenada no depthbuffer, se ela for maior, o pixel é descartado. (O funcionamento descrito é o padrão, existem diversos outros modos que podem ser usados para produzir efeitos especiais)
- **Alpha Test**: Podemos descartar pixels de acordo com o alpha de sua cor.
- **Blending**: Podemos combinar (de diversas maneiras, Ex: usando o alpha) o valor do pixel atual com o seu corresponde que esta no framebuffer.

5.5 Finalizando o Shader

Anteriormente criamos o Vertex e o Pixel Shader, agora vamos dizer o para a GPU quais funções fazem cada passo

```
technique Tut0
{
    pass Pass1
    {
        VertexShader = compile vs_2_0
            VertexShaderFunction();
        PixelShader = compile ps_2_0
            PixelShaderFunction();
    }
}
```

Um arquivo pode ter diversas funções de Pixel e Vertex Shader, uma técnica é uma combinação especifica de um vertex shader com um pixel shader. Cada técnica pode ter diversos passos. As funções de Pixel e Vertex Shader assim como as definições de estruturas de entrada e saída devem estar no mesmo arquivo onde foi definida a técnica (pode-se usar diretivas como include igual ao C).

5.6 Criando e aplicando o Shader no XNA

Em XNA,os shaders são representados pela classe Effect, abaixo temos a definição do nosso effect. É necessário carregá-lo com o ContentPipeline da mesma maneira que fazemos com texturas e depois setarmos a técnica que será utilizada.

```
Effect Tutorial0Effect;
Tutorial0Effect = this.Content.Load<effect>(
    "Effects//Tutorial0Effect0");
Tutorial0Effect.CurrentTechnique =
    Tutorial0Effect.Techniques["Tut0"];
```

5.7 Desenhando um Modelo usando nosso Shader

Para desenhar um modelo devemos definir o VertexBuffer (junto com o VertexDeclaration), o IndexBuffer, o shader, as constantes e o render state que queremos usar. O código a seguir realiza estas tarefas. (neste momento não estamos enviando, por exemplo, o vertexbuffer completo, estamos apenas passando um ponteiro para a região da RAM da GPU onde ele se encontra).

```
///define constante view
this.Tutorial0Effect.Parameters["View"].
    SetValue(cameraSimples.View);
///define constante projection
```

```
this.Tutorial0Effect.Parameters["Projection"
    ].SetValue(cameraSimples.Projection);
///define a textura
this.Tutorial0Effect.Parameters["
    diffuseTexture"].SetValue(diffuse);
///define a matrix world            this.
    Tutorial0Effect.Parameters["World"].
    SetValue(trandformation);
///define o Index Buffer
this.GraphicsDevice.Indices = INDEXBUFFER;
///define o VertexBuffer
this.GraphicsDevice.SetVertexBuffer(
    VERTEXBUFFER);
///define o Shader
this.Tutorial0Effect.CurrentTechnique.Passes
    [0].Apply();
///Desenha o modelo (Ativa a pipeline)
this.GraphicsDevice.DrawIndexedPrimitives(
    PrimitiveType.TriangleList, 0, 0,
    vertexCount,0, primitiveCount);
```

6 Phong Shading

Esta técnica para o cálculo de iluminação foi desenvolvida por Bui Tuong Phong na universidade de Utah em 1973. Na época de sua concepção ela foi considerada radical (pois era simples demais comparado com os modelos da época), entretanto hoje é o mais utilizado em aplicações de tempo real (o modelo originalmente proposto foi sendo alterado no decorrer dos anos).

O modelo Phong é essencialmente uma aproximação empírica de um sistema de iluminação local (sistemas que consideram apenas a luz que incide diretamente nos objetos). Ele descreve como uma superfície reflete a luz, sendo uma combinação da reflexão difusa, da reflexão especular e da reflexão ambiente. Para uma explicação mais física (incluindo conceitos de Irradiancia, radiancia ...)

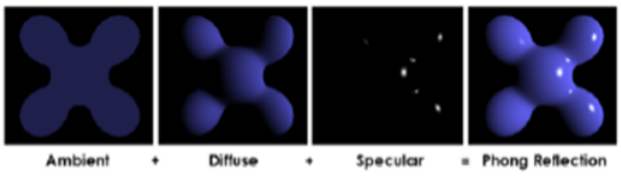


Figure 5: *combinação das reflexões para a geração da imagem final*

Matematicamente, o modelo de Phong é quebrado nas três componentes aditivas citadas acima conforme mostra a equação a seguir:

Intensidade da Cor Final = Ia + Id + Is

- Ia = Componente ambiente
- Id = Componente difusa
- Is = Componente especular

6.0.1 Componente ambiente

O primeiro fator corresponde ao que chamamos de iluminação indireta (luz que é refletida diversas vezes no ambiente e chega até o objeto em questão). Ela é necessária, pois alguns objetos não são iluminados diretamente por nenhuma fonte de luz e mesmo assim são vistos.

Modelos de iluminação de tempo real não conseguem simular este fator de maneira apropriada (alta complexidade / alto custo computacional), então se usa uma constante Ka para representá-lo.

$$I_a = K_a + I_{ambiente}$$

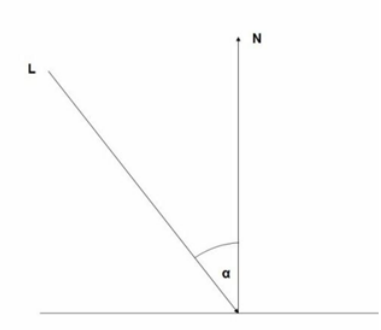
Sendo:

- Ka constante de luz ambiente (Definido pelo usuário, Propriedade do material/luz)
- Iambiente é a cor do objeto (Definido pelo usuário, Propriedade do material)

6.0.2 Componente Difuso

Corresponde à luz que o objeto reflete igualmente em todas as direções.

A reflexão é calculada segundo a lei de Lambert que infere que a energia que reflete de uma superfície é proporcional ao cosseno do ângulo entre a direção da luz e à normal da superfície. Equacionando temos:



$$I_d = K_d * I_{diffuse} * cos\alpha$$

Em que:

- Idiffuse representa a intensidade da cor difusa do objeto (definida pelo usuário,propriedade do material - existem alguns livros que consideram apenas a intensidade da luz (Ilight, apresentado a seguir) e uma constante Kd (apresentado a seguir também) na equação da luz difusa, eu preferi uma abordagem que separa a cor do objeto da constante Kd por ser mais fácil de compreender).
- Kd representa o coeficiente de reflexão difusa da superfície. (definida pelo usuário, propriedade do material/luz)
- Alpha é o ângulo entre a normal do objeto no ponto em questão e a direção da luz. (Propriedade da cena). Notar que o vetor Direção da Luz aponta para o objeto e a Normal aponta para fora.

Costumamos reduzir o cosseno acima a um produto escalar (já que a normal e o vetor de direção da luz são unitários), desta forma a equação resultante é:

$$I_{Diffuse} = K_d * I_{light} * (\bar{n} \cdot \bar{L})$$

Com:

- N sendo o vetor normal no ponto em questão da superfície
- L o vetor de incidência da luz (invertido, ou seja, apontando para a luz ao invés de apontar para o objeto)

Normalmente Adicionamos um fator a mais nesta equação chamado de ILuz que representa a cor da luz que esta iluminando o objeto (as vezes não representamos ILuz na equação pois consideramos a cor da luz como branca (r=1,g=1,b=1)).

$$I_{Diffuse} = K_d * I_{diffuse} * I_{light} * (\bar{n} \cdot \bar{L})$$

Figure 6: *objeto sendo iluminado por uma fonte de luz com apenas componente difuso*

6.0.3 Componente especular

O terceiro componente da equação do modelo Phong é resultado de uma característica comum a todos os objetos reflexivos, que são pequenas regiões de brilho mais intenso com tamanho maior ou menor dependendo do tipo de material. Esse efeito é dependente da posição do observador.

O fator especular é meramente a reflexão da luz projetada na superfície do objeto. O modelo Phong não reflete fielmente o formato da luz, pois considera que todos os pontos de luz são representadas por formas cônicas. Apesar da simplificação nota-se que o efeito gerado é suficiente para "enganar" a percepção humana.

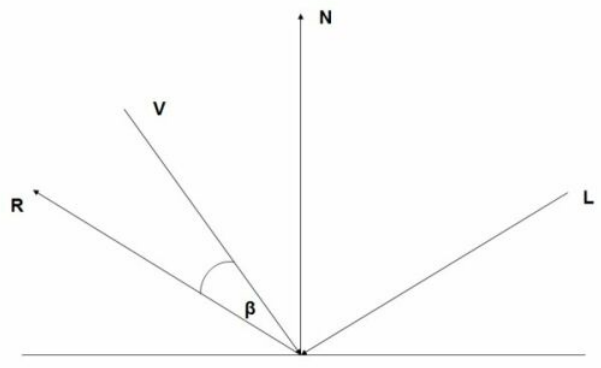


Figure 7: *Componentes da equação especular*

$$I_{Specular} = K_s * I_{Light} * cos\beta^n$$

Onde:

- Ks é uma constante especular do material (Definida pelo usuario, propriedade do material/luz)
- ILight é a intensidade do raio especular que atinge o objeto (definido pelo usuario, propriedade do material)
- n (que esta elevando o cos)é chamado de Specular Power e controla o "tamanho" do brilho especular. (definido pelo usuario, propriedade do material). Quanto maior o valor deste parametro menor será a região de brilho gerada.
- Beta é o ângulo entre o raio refletido R e o Vetor Eye V (vetor entre observador e o ponto iluminado, apontando para o observador) (Propriedade da cena)

6.0.4 Juntando as partes

Como vimos, a equação de Phong é dada por:

Intensidade da Cor Final = Ia + Id + Is

Substituindo cada um dos componentes vistos temos:

$$Intensidade = K_a * I_{ambiente} + K_d * I_{diffuse} * I_{light} * cosN * L + K_s * I_{specular} * cos(R \cdot V)^n$$

Os parâmetros N,L,R e Eye provêem da Camera, Da Luz e da Geometria do Objeto, não são definidos diretamente pelo usuário.

Os parâmetros Ka,Kd,Ks,Iambiente,Idiffuse,Ispecular, Ilight e n são definidos pelo usuário (propriedades do material e da luz) e nos shaders são chamados de:

- Ka: Ambient Intensity
- Kd: Diffuse Intensity
- Ks: Specular Intensity
- n: Specular Power
- Iambiente: Ambient Color
- Idiffuse: DiffuseColor
- Ispecular: Specular Color
- Ilight: Light Color

Usarei esta nomenclatura acima no shader desenvolvido.

6.0.5 Entendendo e usando a Equação

Para iluminar uma cena devemos, para cada ponto da superfície de cada uns dos objetos, calcular esta equação. (O valor que aparecera no monitor é a saída desta equação)

Em uma cena normalmente temos diversas luzes, a cor final de um objeto é obtida através da soma dos efeitos de cada uma das luzes individualmente.

Conforme mostrado, o modelo de Phong é um sistema de iluminação local, pois a intensidade de um ponto depende apenas dele, não precisamos de informações de pontos vizinhos (modelos de iluminação como Radiosity e Raytracing precisam).

Existem diversos tipos de luzes, as mais comuns são: Point, Directional e Spot. Elas se diferenciam principalmente quanto ao alcance (por exemplo: uma point light tem um raio de alcance igual ao de uma esfera) e quanto à atenuação (Ka,Kd,Ks). (Por exemplo, uma point light é mais intensa no centro e menos nas bordas).

Para este tutorial, implementaremos uma Directional Light apenas, sua atenuação é constante em toda a cena e o seu raio de alcance é global (afeta todos os objetos da cena). Matematicamente, Ka,Kd e Ks não variam (apesar de serem chamadas de constantes, elas variam no caso de Point Lights ... muitas pessoas preferem criar uma nova variável chamada LightAttenuation para este propósito) e todos os objetos são afetados pela equação.

6.0.6 Implementação - Shader

A conversão entre um modelo matemático para shaders não é sempre uma tarefa trivial. Devemos entender como que as equações funcionam e encaixá-las na arquitetura da GPU. No caso de Phong Shading este processo é bastante simples.

Conforme visto, temos uma equação que calcula a cor final de um ponto de um objeto. Em shaders "ponto" significa pixel gerado na rasterização do modelo. Desta forma, calcularemos a equação de Phong para cada pixel de cada objeto do mundo 3D.

A seguir descreverei o shader responsável por implementar o Phong Shading.

As constantes necessárias para este shader são:

```
float4x4    World;
float4x4    View;
float4x4    Projection;
float3       LightDirection;
float4       LightColor;
float3       camPosition;
float4       SpecularColor;
float        SpecularPower;
float        SpecularIntensity;
float4       AmbientColor;
float        AmbientIntensity;
float        DiffuseIntensity;
texture      DiffuseTexture;
```

Estes parâmetros são definidos pelo usuário, alguns diretamente como a SpecularPower e outros indiretamente como a camPosition.

A entrada do vertex shader é bastante simples e recebe apenas a posição, coordenada de textura e normal de cada vértice, conforme mostrado a seguir:

```
struct VertexShaderInput
{
    float3 Position : POSITION0;
    float3 Normal   : Normal0;
    float2 TexCoord : TexCoord0;
};
```

A saída contém os dados da entrada transformados mais o nosso Eye Vector:

```
struct VertexShaderOutput
{
    float4 Position         : POSITION0;
    float3 N                : TEXCOORD0;
    float2 TextureCoord     : TEXCOORD1;
    float3 V                : TEXCOORD2;
};
```

O vertex Shader transforma os dados do vértice e calcula o Eye Vector:

```
VertexShaderOutput VertexShaderFunction(
    VertexShaderInput input)
{
    VertexShaderOutput output;

    float4 worldPosition = mul(float4(input.
        Position,1), World);
    float4 viewPosition = mul(worldPosition,
         View);
    output.Position = mul(viewPosition,
        Projection);
    output.N = mul(float4(input.Normal,0),
        World);
    output.TextureCoord = input.TexCoord;
    output.V = camPosition - worldPosition;
    return output;
}
```

O Eye Vector é o vetor entre a posição da camera e o vértice atual que estamos processando.

Todo o cálculo da equação de Phong é feito no pixel shader (pode-se fazer no Vertex Shader, mas a aparência não fica legal – ver Flat Shading):

```
float4 PixelShaderFunction(
    VertexShaderOutput input) : COLOR0
{
    float3 Normal = normalize(input.N);
    float3 LightDir = -normalize(
        LightDirection);
    float3 ViewDir = normalize(input.V);
    float4 diffuseColor = tex2D(
        DiffuseSampler,input.TextureCoord);

    float Diff = saturate(dot(Normal,
        LightDir));
    // R = 2 * (N.L) * N - L
    float3 Reflect = normalize(2 * Diff *
        Normal - LightDir);
    float Specular = pow(saturate(dot(
        Reflect, ViewDir)), SpecularPower);
    // I = Dc*A + Dcolor * Dintensity * N.L
        + Sintensity * Scolor * (R.V)n
    return AmbientColor*AmbientIntensity +
        LightColor * DiffuseIntensity *
        diffuseColor * Diff +
        SpecularIntensity * SpecularColor *
        Specular;

}
```

O cálculo da equação de phong começa em:

float Diff = saturate(dot(Normal, LightDir)); A operação dot efetua um produto escalar, operação básica em computação gráfica que consiste em multiplicar cada elemento do primeiro vetor pelo elemento correspondente do segundo vetor e depois somar os valores obtidos. No nosso caso esta operação esta calculando o Cosseno entre os vetores Normal e LightDir.

A operação saturate garante que o valor passado como argumento esteja entre 0 e 1.

A linha

float3 Reflect = normalize(2 * Diff * Normal - LightDir);

Calcula o reflection vetor. Para entender o porquê de esta equação funcionar sugiro este livro (Algebra Linear).

Em seguida temos

float Specular = pow(saturate(dot(Reflect, ViewDir)), SpecularPower);

Que implementa a parte . (saturate usado para garantir que o Cos Teta permanece no intervalo 0-1)

Por fim temos a implementação da equação completa:

return AmbientColor*AmbientIntensity + LightColor * DiffuseIntensity * diffuseColor * Diff + SpecularIntensity * SpecularColor * Specular;

Como vimos, este shader é bastante simples e o resultado obtido é interessante.

A implementação vista é didática porém pouco prática. Em um universo mais real temos várias luzes de vários tipos (spot, directional ...) afetando cada uns dos objetos.

Para atender estas necessidades existem algumas “arquiteturas” de sistema de iluminação. As mais usadas são: Single Pass Multi-Lightning e Multiple Pass Multi-Lightning para o caso de Forward Rendering e Deferred Shading e Inferred Shading para o caso de Deferred Render. (A PloobsEngine suporta nativamente Deferred Shading e Single Pass Multi-Lightning).