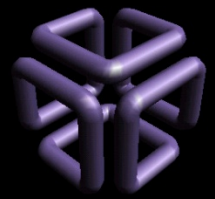
The Cyclops Project
German-Brazilian Cooperation Programme on IT
CNPq GMD DLR

Computação Gráfica:

Pixel Shading em GPU

Prof. Dr. rer.nat. Aldo von Wangenheim

Departamento de Informática e Estatística - INE/CTC/UFSC



Conteúdo desta Aula

- Unidade de Processamento Gráfico (GPU)
  - Pipeline gráfico acelerado
  - Iluminação padrão do pipeline
- Programação de shaders
  - Shaders no pipeline gráfico
  - OpenGL Shading Language (GLSL)
  - Implementação de Toon Shading

Unidades de Processamento Gráfico (GPU)

- GPUs são processadores especializados para acelerar a renderização, de forma a gerar imagens de cenas complexas várias vezes por segundo para aplicações interativas.
  - Muitas das etapas de renderização possuem processamento independente de grande quantidades de informação. As GPUs possuem uma arquitetura massivamente paralela para aproveitar esta característica.
  - Cálculos de transformações e iluminação envolvem muitas matrizes e vetores. O conjunto de instruções da GPU foi desenvolvido para acelerar essas operações.

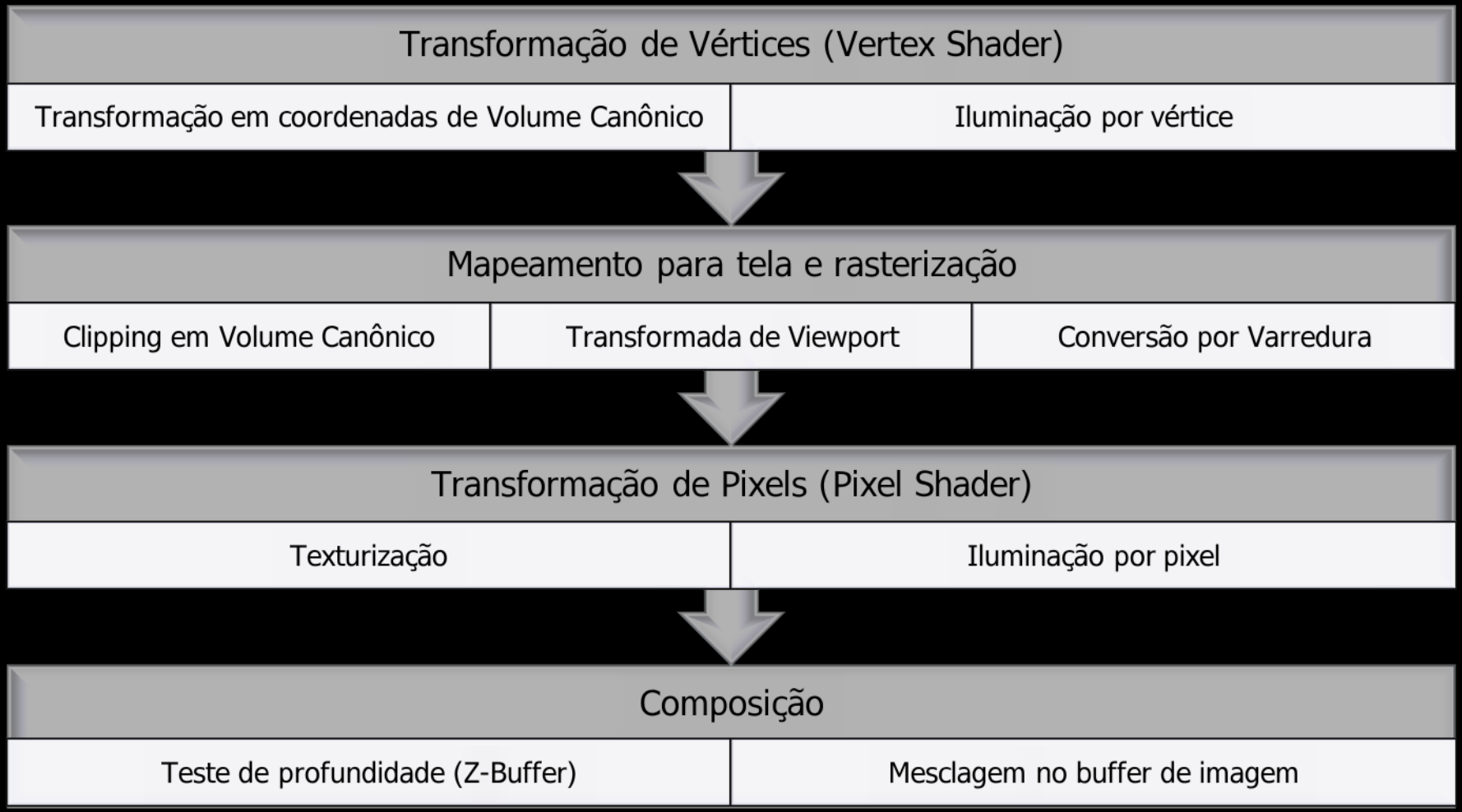
Unidades de Processamento Gráfico (GPU)

- Aplicações se comunicam com a GPU através de uma biblioteca gráfica, que por sua vez se comunica com o driver de vídeo.
- Bibliotecas gráficas mais usadas são o Direct3D e o OpenGL.
- Uma vez definida a cena, a maior parte das etapas de renderização ocorre dentro da GPU.
- Algumas dessas etapas podem ser controladas por parâmetros ou até mesmo totalmente reprogramadas.

Pipeline gráfico acelerado

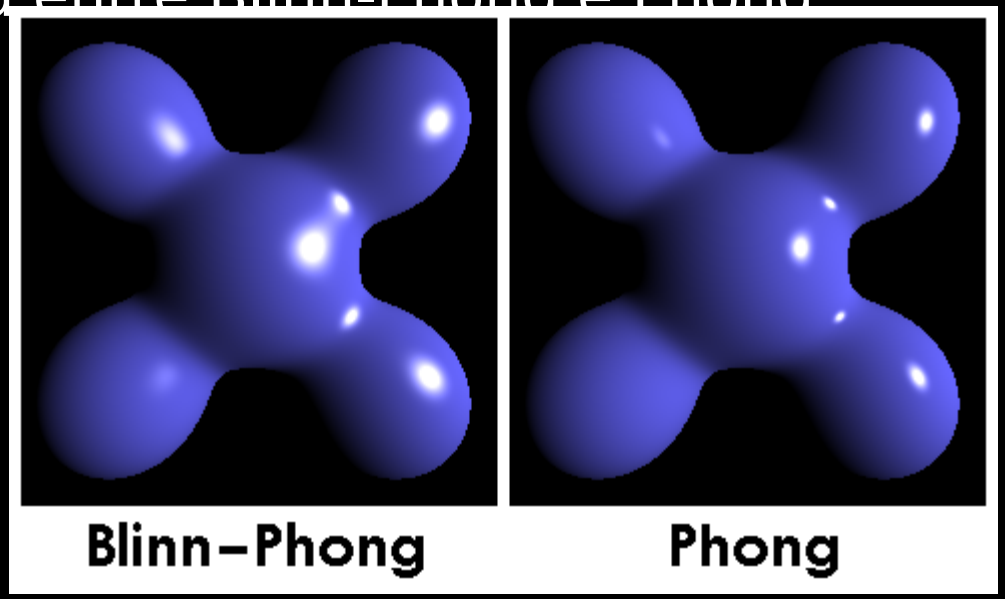
- As etapas do pipeline podem ser divididas simplificadamente em:
  1. Transformação de vértices: as coordenadas são normalizadas para o Volume Canônico e são feitos cálculos de iluminação por vértice.
  2. Mapeamento para tela e rasterização: ocorre o Clipping, a transformada de Viewport e a Conversão por Varredura.
  3. Transformação de pixels: nos pixels gerados é feita a aplicação de texturas e iluminação por pixel.
  4. Composição: os pixels passam pelo teste de profundidade e são mesclados no buffer de imagem.

Pipeline gráfico acelerado



Iluminação padrão do pipeline

- As bibliotecas gráficas Direct3D e OpenGL utilizam o modelo de iluminação Blinn-Phong. A iluminação é calculada por vértice, ou seja, é usado Gouraud shading.
- Diferença entre Blinn-Phong e Phong:

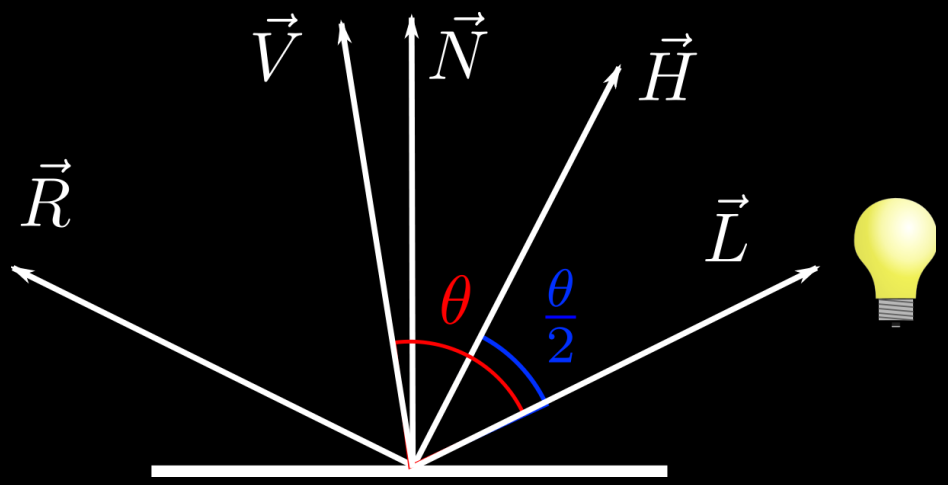


Iluminação padrão do pipeline

$$I_{Phong} = I_a r_a + I_d r_d (\vec{L} \cdot \vec{N}) + I_s r_s (\vec{R} \cdot \vec{V})^s$$

$$I_{Blinn-Phong} = I_a r_a + I_d r_d (\vec{L} \cdot \vec{N}) + I_s r_s (\vec{N} \cdot \vec{H})^s$$

- O vetor H é o vetor entre o vetor da luz e do observador. Blinn mostrou que essa aproximação é computacionalmente mais eficiente e gera resultados próximos a

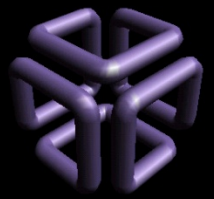


Iluminação padrão do pipeline

- Existem outros modelos de iluminação para diferentes tipos de aplicação. Com a evolução dos processadores gráficos, veio a possibilidade de substituir o modelo de iluminação.
- Exemplo d [illegible] realista:

Programação de shaders

ine5341 - Computação Gráfica

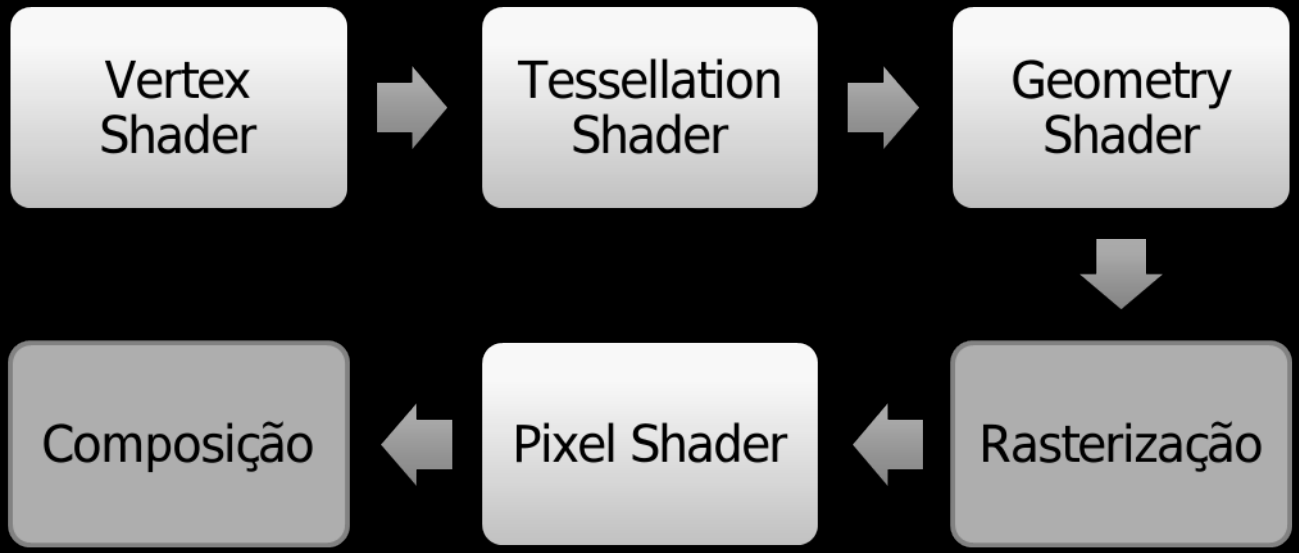


Programação de shaders

- As placas de vídeo modernas permitem que algumas das etapas do pipeline possam ser substituídas.
  - As primeiras etapas reprogramáveis foram o Vertex Shader e o Pixel Shader, em 2001.
  - Em 2006, algumas GPUs passaram a suportar o Geometry Shader, uma etapa após o Vertex Shader onde é possível criar ou remover vértices e facetas antes do Clipping.
  - Em 2009 surgiu o Tessellation Shader, para programação do hardware específico de subdivisão de facetas em GPUs. Técnicas de Tessellation servem especificamente para refinar ou simplificar conjuntos de polígonos. Isso é possível com Geometry Shader, porém de forma ineficiente.

Programação de shaders com GLSL

- GL Shading Language é a linguagem específica para programação de shaders.
- Do ponto de vista do programador shader, o pipeline gráfico pode ser visto da seguinte forma:

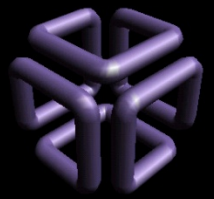


Programação de shaders com GLSL

- Para realizar suas funções, os shaders tem acesso de leitura às texturas carregadas na memória da placa de vídeo. Desta forma é possível aplicar texturas na etapa de Pixel Shading, por exemplo.
- Os shaders também têm acesso de leitura à parâmetros especificados pela aplicação na forma de variáveis “uniform”.
- Além disso, existe um tipo especial de variável que são as “varying”. Elas são usadas para passar informações entre shaders consecutivos.

Vertex Shader

- O Vertex Shader é responsável pelo processamento dos vértices. O mesmo algorítmo é executado sobre cada um dos vértices a serem renderizados.
  - Como entrada tem-se a posição do vértice e alguns atributos básicos especificados pela aplicação, como normal e coeficientes do material.
  - O shader é responsável por transformar a posição do vértice para coordenadas do Volume Canônico.
  - Adicionalmente podem ser calculados valores a serem transmitidos para as etapas seguintes, através das variáveis “varying”.

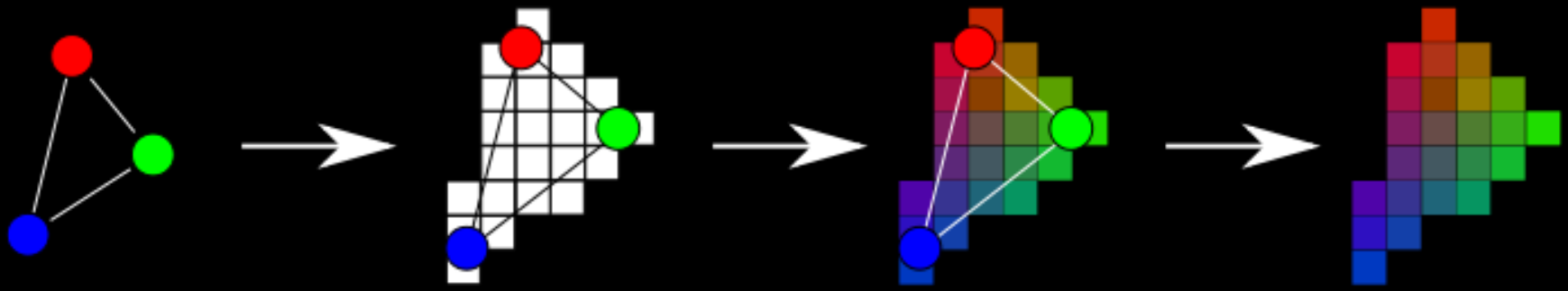


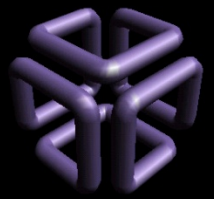
Tessellation Shader e Geometry Shader

- Essas etapas são opcionais e não são relevantes para iluminação. O propósito delas é fazer processamento na geometria dos objetos.
- Enquanto o Vertex Shader apenas altera atributos dos vértices, essas etapas são capazes de remover ou criar novos vértices ou até mesmo novos polígonos.
- Esses shaders são interessantes para geração procedural de geometria, a partir da descrição de curvas paramétricas por exemplo, e detalhamento de superfícies.

Rasterização

- Até aqui, os shaders estavam manipulando vértices. Esta etapa transforma os vértices em pixels e não é reprogramável.
- As variáveis “varying” e alguns outros atributos, como cor dos vértices, são definidas para cada pixel através de interpolação linear.





Pixel Shader

- Este é o ultimo shader a ser executado e ele processa cada pixel individualmente. O resultado desta etapa é a cor do pixel.
  - É possível também alterar a profundidade do pixel, pois o OpenGL especifica que o teste de profundidade (Z-Buffer) será feito somente depois.
  - Porém se a profundidade não for alterada o teste pode ser executado antes, de forma a processar menos pixels nesta etapa.
- Nesta etapa o GLSL só fornece as coordenadas de tela do pixel. Os vetores necessários para iluminação como normal, direção da luz e do observador devem ter sido passados como variáveis "varying". Note que vetores unitários após serem interpolados deixam de ser unitários.

OpenGL Shading Language (GLSL)

- Linguagem desenvolvida para escrever programas que substituam as etapas de shading.
- Sintaxe muito similar a linguagem C.

```
void main() {

   /* Define a cor do pixel como vermelho */

   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);

}
```

Tipos primitivos

- float
  - Não segue o padrão IEEE obrigatoriamente.
  - Permite hardware mais barato, com menos precisão numérica.
- int
  - Na versão do GLSL correspondente ao OpenGL 2.x, a especificação não espera que o hardware tenha suporte a inteiros no conjunto de instruções.
  - A precisão do inteiro nesta versão do GLSL é de 16 bits.
- bool
  - Assim como o int, não se espera que a GPU tenha esse tipo nativamente. Porém não há problemas de precisão.

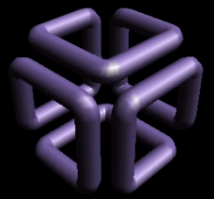
Tipos primitivos

- vec2, vec3, vec4
  - Tipos vetoriais de até 4 componentes ponto-flutuante. As operações aritméticas, tais como soma, subtração, multiplicação e divisão, são definidas componente a componente.
  - Os valores do vetor podem ser acessados de diversas formas:
    - “x, y, z, w” ou “r, g, b, a” ou “s, t, u, v”: acessa os valores como se fossem propriedades de um objeto.
    - As 3 nomenclaturas existem apenas por conveniência. Ao usar um vec4 para cores pode-se usar “rgba”, ao invés de “xyzw” por exemplo, para clareza do código.

Tipos primitivos

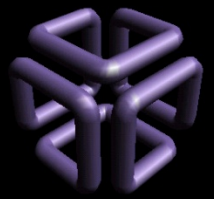
- vec2, vec3, vec4
  - Outra forma de acessar os componentes são através de indexação, por exemplo v[2] para obter a terceira componente (equivalente a v.z, v.b e v.t).
  - Além disso, as propriedades podem ser especificadas em conjunto. Considere o seguinte exemplo:

```
vec4 v = vec4(1.0, 2.0, 3.0, 4.0);
v[0] = 10.0;        /* v == (10, 2, 3, 4) */
v.zw += v.xy;       /* v == (10, 2, 13, 6) */
v.rgb = v.wwx;      /* v == (6, 6, 10, 6) */
```



Tipos primitivos

- vec2, vec3, vec4
  - Além disso, GLSL define operações vetoriais tais como produto escalar e vetorial, normalização, etc:
    - dot(u, v): retorna o produto escalar.
    - cross(u, v): retorna o produto vetorial.
    - normalize(u): retorna o vetor normalizado.
    - length(u): retorna a norma (comprimento) do vetor.



Tipos primitivos

- mat2, mat3, mat4
  - Matrizes quadradas. Assim como vetores, possuem as operações aritméticas definidas elemento a elemento, com exceção da multiplicação. A multiplicação entre vetores e matrizes ou matrizes e matrizes realiza a multiplicação definida pela álgebra linear.
  - As matrizes podem ser inicializadas por 4 vetores ou elemento a elemento. Note que no OpenGL as matrizes são definidas coluna a coluna.

```
mat3 a = mat3(11.0, 21.0, 31.0, 12.0, 22.0, 32.0, 13.0, 23.0,
33.0);
mat2 b = mat2(vec2(8.1, 8.2), vec2(9.1, 9.2));
```

Tipos primitivos

- Samplers (sampler2D, sampler3D, ...)
  - Samplers são tipos especiais para acessar texturas. As texturas devem ser previamente carregadas pela aplicação através do OpenGL e os identificadores delas passadas para os shaders através desse tipo de variável.

Structs

```
struct cube {

   vec3 position;

   float length;

};

struct cube myNewCube;
```

- Como structs C, porém simplificadas.
- Podem ser aninhadas.
- Muitas variáveis fornecidas pelo GLSL são organizadas como structs.

Arrays

- A sintaxe é como em C, porém não há ponteiros e nem espaço de memória para armazenar os arrays.
- O tamanho do array precisa ser conhecido em tempo de compilação, ou seja, devem ser inicializados com tamanho fixo.
- Indexação de arrays fora de seus limites produz resultados indefinidos.
- É desencorajado a transferências de arrays grandes entre a aplicação e os shaders através de variáveis “uniform” por questões de performance. Como alternativa pode-se usar texturas de uma dimensão.

Limitações

- O único acesso a memória é o acesso a texturas.
- Não há recursão. Não há ponteiros.
- Arrays devem ter seu tamanho conhecido em tempo de compilação. Isso porque como não há memória para ser indexada, o compilador aloca registradores para acesso ao array em tempo de compilação.
- Ponto flutuante não segue padrão IEEE-754, portanto as regras de arredondamento também variam.

Variáveis builtins – qualquer shader

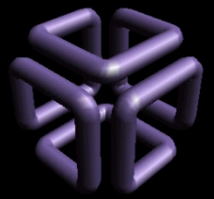
- gl_FrontMaterial: propriedades do material (ambient, diffuse, specular, shininess).
- gl_LightSource[n]: n-ésima fonte de luz (valores de ambient, diffuse, specular, position).
- gl_LightModel: propriedades de iluminação global (ambient).

Variáveis de shader – Vertex Shader

- Entradas (fornecidas pela aplicação)
  - gl_Normal: normal em coordenadas de mundo.
  - gl_Vertex: posição em coordenadas de mundo.
  - gl_Color: cor dada pela aplicação.
  - gl_TexCoord: coordenada de textura dada pela aplicação.
- Saídas
  - gl_Position: posição do vértice em coordenadas do Volume Canônico. É obrigatório definir essa variável.
  - gl_FrontColor: cor a ser dada para o vértice

Variáveis de shader – Vertex Shader

- Variáveis adicionais:
  - gl_ModelViewMatrix: matriz de transformação de coordenadas de mundo para coordenadas de window. Lembre-se que após esta transformação o observador está na origem, portanto é fácil obter os vetores de direção dos vértices até o observador.
  - gl_ProjectionMatrix: matriz de transformação de coordenadas de window para coordenadas de Volume Canônico.
  - gl_ModelViewProjectionMatrix: resultado da multiplicação das matrizes anteriores.
    - Para a aplicação especificamente no gl_Vertex, existe a função “ftransform()” que já retorna as coordenadas do vértice em espaço de Volume Canônico.



Variáveis de shader – Vertex Shader

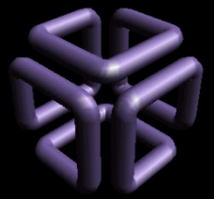
- Variáveis adicionais:
  - gl_NormalMatrix: matriz análoga à gl_ModelViewMatrix, mas para transformar as normais para coordenadas de window. É usada uma matriz separada pois esta matriz não faz transformação de escala, o que alteraria a direção das normais.

Variáveis de shader – Pixel Shader

- Entradas (definidas pela rasterização):
  - gl_Color: cor do pixel interpolada pela rasterização.
  - gl_FragCoord: posição do pixel em coordenadas de viewport.
- Saídas
  - gl_FragColor: cor a ser definida para o pixel.

Funções fornecidas

- Funções trigonométricas e exponenciais: sin, cos, acos, asin, tan, atan, pow, sqrt, log...
- Vetoriais: além das citadas anteriormente (dot, cross, normalize) existe o reflect, especialmente útil em alguns algoritmos de shading.
  - reflect(v, n): calcula o vetor “v” refletido em uma superfície com normal “n”. O vetor “n” deve estar normalizado.
- max(a, b) e min(a, b) para retornar o maior e menor dos valores.



Funções fornecidas

- Acesso a textura: para acessar as texturas, deve-se usar essa classe de funções.
  - texture2D(s, pos): “s” é a varíavel do tipo sampler2D que referencia a textura e “pos” é um vec2 com valores no intervalo entre 0 e 1 que indica qual pixel da textura deve ser lido. O valor retornado é a cor deste pixel.
  - Isso pode ser generalizado para texture1D e texture3D, mas usando float e vec3, respectivamente, para especificar a coordenada dentro da textura.

Implementação de Toon Shading

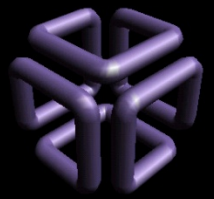
- Técnica de renderização cartoonizada.
- Modifica a componente difusa de forma a criar transições bruscas na iluminação.
- A seguir o passo a passo de como realizar a implementação desta técnica.

Implementação de Toon Shading

- A idéia é utilizar a componente difusa de Phong e reduzir o resultado a poucas cores básicas.
- Note que o efeito de suavização que a luz causa na superfície é baseada no ângulo entre a luz e a normal da superfície. É esta suavização que queremos simplificar em intensidades fixas.
- Mas primeiro precisamos calcular os vetores básicos.

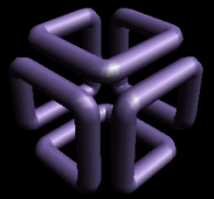
Implementação de Toon Shading

- Precisamos da normal da superfície e do vetor que parte do vértice até a luz. Os vetores são trabalhados em coordenadas de window por simplificar alguns cálculos.
- Essas informações podem ser calculadas no Vertex Shader e transmitidas ao Pixel Shader através de variáveis “varying”.
- Cálculo da normal da superfície:
  - Basta aplicar a matriz de transformação da normal para coordenadas de window.
- Cálculo da posição da fonte de luz (usaremos apenas uma):
  - A propriedade gl_LightSource[0].position.xyz nos dá as



Implementação de Toon Shading

- Cálculo do vetor do vértice à fonte de luz:
  - Primeiro deve ser obtida a posição do vértice em coordenadas de window aplicando a matriz de transformação correspondente.
  - Em seguida, basta calcular o vetor entre a posição da fonte de luz e a posição do vértice obtida.
- Todos os vetores indicam direção, portanto devem ter comprimento unitário. Porém a interpolação desses vetores nem sempre terá vetores unitários como resultado. Para corrigir isso, esses vetores devem ser normalizados no Pixel Shader.



Implementação de Toon Shading

- Com esses vetores acessíveis no Pixel Shader, é possível agora calcular a componente difusa.
- Para criar as transições bruscas, basta fazer uma função que reduza os intervalos de intensidade:

```
float toonShading(float intensity) {
    if (intensity > 0.95) return 1.0;
    else if (intensity > 0.5) return 0.6;
    else if (intensity > 0.25) return 0.4;
    else return 0.0;
}
```

Implementação de Toon Shading

- Por fim, multiplicamos a intensidade alterada com as propriedades difusas do material e da luz e completamos o Pixel Shader:

```
varying vec3 normal;

varying vec3 luz;


void main() {

    vec3 N = normalize(normal);

    vec3 L = normalize(luz);

    float NdotL = dot(N, L);
```

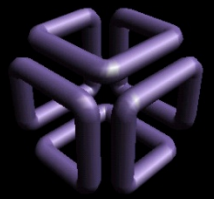
```
vec4 Id =
gl_LightSource[0].diffuse;

vec4 rd =
gl_FrontMaterial.diffuse;


float toon = toonShading(dot(N,
L));


gl_FragColor = Id * rd * toon;
}
```

Outros cálculos de vetores

- Vetor da luz refletido:
  - Basta usar a função reflect com o vetor direção da luz com o vetor normal da superfície.
- Vetor do observador:
  - Como os cálculos são feitos em coordenadas de window, o observador está na coordenada (0,0,0). Basta subtrair a posição do vértice da origem.
- Vetor bissetriz entre luz e observador (H):
  - Soma-se o vetor da luz com o do observador (ambos normalizados) e normaliza-se o resultado.

Trabalho: Pixel Shading em GLSL

- O programa Shader Maker será usado para desenvolvimento e teste. Nele é possível especificar parâmetros de material, iluminação, variáveis “uniform”, carregar objetos, etc.
- Existem vários algoritmos para calcular as componentes difusas e especulares, além daquelas que vimos nos modelos Phong e Blinn-Phong. Nos slides seguintes esses algoritmos serão descritos.
- Os algoritmos devem ser implementados no mesmo código de shader. Se houver mais de um algoritmo para uma componente, deve haver uma variável “uniform” para selecionar qual dos algoritmos será ativado.



Trabalho: Pixel Shading em GLSL

- A convenção para as fórmulas de iluminação a seguir são:
  - I : a intensidade da componente em um canal de cor.
  - ρ : refletância do material (d para difusa, s para especular).
  - : transmitância da luz (d para difusa, s para especular).
  - N : vetor normal da superfície
  - L : vetor da direção da luz a partir do vértice
  - R : vetor L refletido por N
  - V : vetor da direção do observador a partir do vértice
  - H : vetor bissetriz entre L e V
  - r: rugosidade do material

Trabalho: Pixel Shading em GLSL

- Alguns modelos utilizam o termo de Fresnel para um resultado mais realista no cálculo da componente especular.
- Embora fisicamente acurada, as equações de Fresnel são complexas e por isso normalmente utiliza-se a aproximação de Schlick:

$$fresnel(x) = M_s + (1 - M_s)(1 - x)^5$$

Lambert

$$I = M_d \rho_d (N \cdot L)$$

- É a componente difusa no modelo de Phong.
- Cuidado ao implementar a fórmula. Para evitar que o produto escalar resulte em $(N \cdot L)$ ção negativa, substitua por $\max(0, N \cdot L)$

Oren-Nayar

$$I = M_d \rho_d (N \cdot L)(C_1 + \chi + \varepsilon)$$

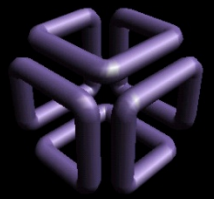
$$\chi = \gamma C_2 \tan \beta$$

$$\varepsilon = (1 - |\gamma|) C_3 \tan(\tfrac{\alpha+\beta}{2})$$

$$\gamma = ((V - N(V \cdot N)) \cdot (L - N(L \cdot N)))$$

$$\alpha = \max[\arccos(V \cdot N), \arccos(L \cdot N)]$$

$$\beta = \min[\arccos(V \cdot N), \arccos(L \cdot N)]$$



Oren-Nayar

$$C_1 = 1 - 0.5\frac{r^2}{r^2+0.33}$$

$$C_2 = \begin{cases} 0.45\frac{r^2}{r^2+0.09}\sin\alpha & \text{se } \gamma \geq 0 \\ 0.45\frac{r^2}{r^2+0.09}\left(\sin\alpha - \left(\frac{2\beta}{\pi}\right)^3\right) & \text{se } \gamma < 0 \end{cases}$$

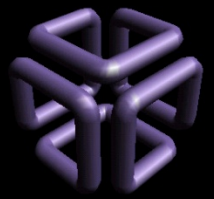
$$C_3 = \frac{1}{8}\left(\frac{r^2}{r^2+0.09}\right)\left(\frac{4\alpha\beta}{\pi^2}\right)^2$$

- Interessante para iluminação de tijolo, concreto e veludo.
- O terceiro fator de I deve receber o mesmo tratamento feito em Lambert.

Oren-Nayar

rugosidade = 0.0

rugosidade = 0.3



Ashikhmin-Shirley

- Difusa:

$$I_d = \frac{28 M_d \rho_d}{23\pi}(1 - M_s)(1 - (1 - \frac{N \cdot L}{2})^5)(1 - (1 - \frac{N \cdot V}{2})^5)$$

- Especular: $I_s = M_s \rho_s \frac{\sqrt{(n_u+1)(n_v+1)}}{8\pi(H \cdot L) max(N \cdot L, N \cdot V)}(N \cdot H)^E F$

$$E = \frac{n_u (H \cdot t_1)^2 + n_v (H \cdot t_2)^2}{1 - (H \cdot N)^2}$$

$$F = fresnel(H \cdot L)$$

- n_u e n_v são valores de anisotropia, entre 0 e 10.000.
- Anisotropia é a propriedade que torna o comportamento do material dependente de direção. Aço escovado, por exemplo, possui um padrão de reflexão maior em uma direção.

Ashikhmin-Shirley

- São usados também dois vetores para definir em espaço tangente. Esses vetores podem ser calculados através de produto cruzado (não esquecendo de

$$t_1 = N \times (1, 0, 0)$$

$$t_2 = N \times t_1$$

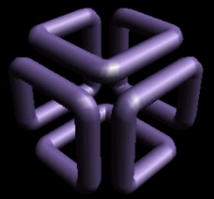
- Note que se os valores de anisotropia forem iguais, o material se torna isotrópico.

Ashikhmin-Shirley

- U = 100
- V = 100

- U = 100
- V = 5000

- U = 5000
- V = 10



Ward (isotrópico)

$$I = M_s(N \cdot L)\frac{e^{-(\frac{\tan\alpha}{r})^2}}{4\pi r^2\sqrt{(N\cdot L)(N\cdot V)}}$$

- O segundo fator também deve ser tratado como em Lambert.
- O valor α no expoente é o ângulo entre N e H. A tangente pode ser calculada através de produto escalar e identidades trigonométricas.
- Essa fórmula possui uma extensão para modelar superfícies anisotrópicas.

Ward (isotrópico)

rugosidade = 0.1

rugosidade = 0.5

Cook-Torrance

$$I = \rho_s (N \cdot L) \frac{FGR}{(N \cdot V)(N \cdot L)}$$

$$F = fresnel(H \cdot V)$$

$$G = \min\left(1, 2\frac{(N \cdot H)(N \cdot V)}{H \cdot V}, 2\frac{(N \cdot H)(N \cdot L)}{H \cdot V}\right)$$

$$R = \frac{1}{r^2 (N \cdot H)^4} \times e^{-\left(\frac{\tan \alpha}{r}\right)^2}$$

- O segundo fator da fórmula deve ser tratado com em Lambert.
- α , na fórmula de R, é o ângulo entre N e H. A tangente desse ângulo pode ser calculada a partir de produto escalar e identidades trigonométricas.

Cook-Torrance

- Usado principalmente para representar metal e

rugosidade = 0.3

rugosidade = 0.7