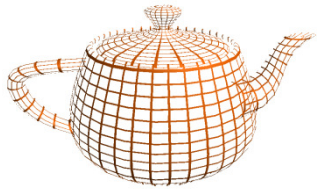
sim

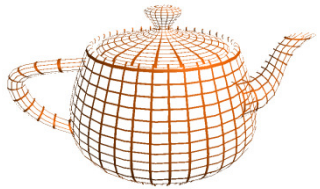
Computação Gráfica

GLSL
Programação de Shaders



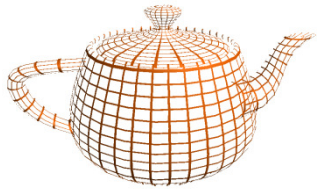
GLSL Sumário

- Evolução do Hardware Gráfico PCs
- Pipeline Gráfico Fixo
- Pipeline Gráfico Programável
  - Processador de Vértices
  - Processador de Fragmentos
- GLSL
  - Aplicação OpenGL
  - Shader I/O
  - Sintaxe
  - Exemplos
- GLSL vs. Cg vs. HLSL vs. Renderman



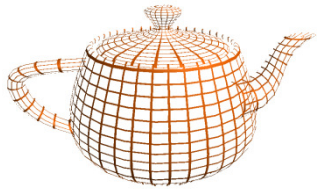
GLSL Sumário

- Evolução do Hardware Gráfico PCs
- Pipeline Gráfico Fixo
- Pipeline Gráfico Programável
  - Processador de Vértices
  - Processador de Fragmentos
- GLSL
  - Aplicação OpenGL
  - Shader I/O
  - Sintaxe
  - Exemplos
- GLSL vs. Cg vs. HLSL vs. Renderman



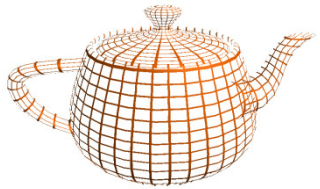
Evolução do Hardware Gráfico PCs

- Controladores VGA (Video Graphics Array)
  - Introduzido em 1987 pela IBM
  - Todo o esforço computacional é realizado pelo CPU.
  - O VGA é utilizado somente como frame buffer.



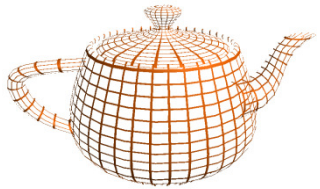
Evolução do Hardware Gráfico PCs

- GPU (Graphics Processing Unit)
  - Termo introduzido pela nVidia em 1990 quando o termo VGA já não descrevia correctamente o hardware gráfico.



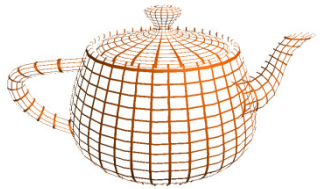
Evolução do Hardware Gráfico PCs

- 1ª Geração de GPUs
  - até 1998 (nVidia TNT2, ATI Rage)
  - Interpolação: capacidade para calcular os pixels a partir dos vértices de um triângulo e aplicar texturas.



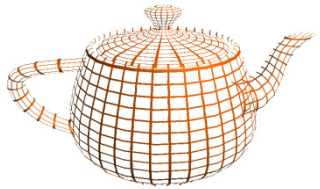
Evolução do Hardware Gráfico PCs

- 2ª Geração de GPUs
  - 1999-2000 (GeForce 256, GF 2, Radeon 7500)
  - Transformação de Vértices
  - Cálculo de Iluminação (por vértice)



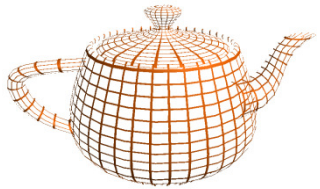
Evolução do Hardware Gráfico PCs

- 3ª Geração de GPUs
  - 2001 (GeForce 3 e 4, Radeon 8500)
  - Programação ao nível dos vértices



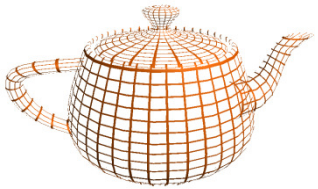
Evolução do Hardware Gráfico PCs

- 4ª Geração de GPUs
  - 2002-... (GeForce FX, ATI Radeon 9700)
  - Programação ao nível do pixel.

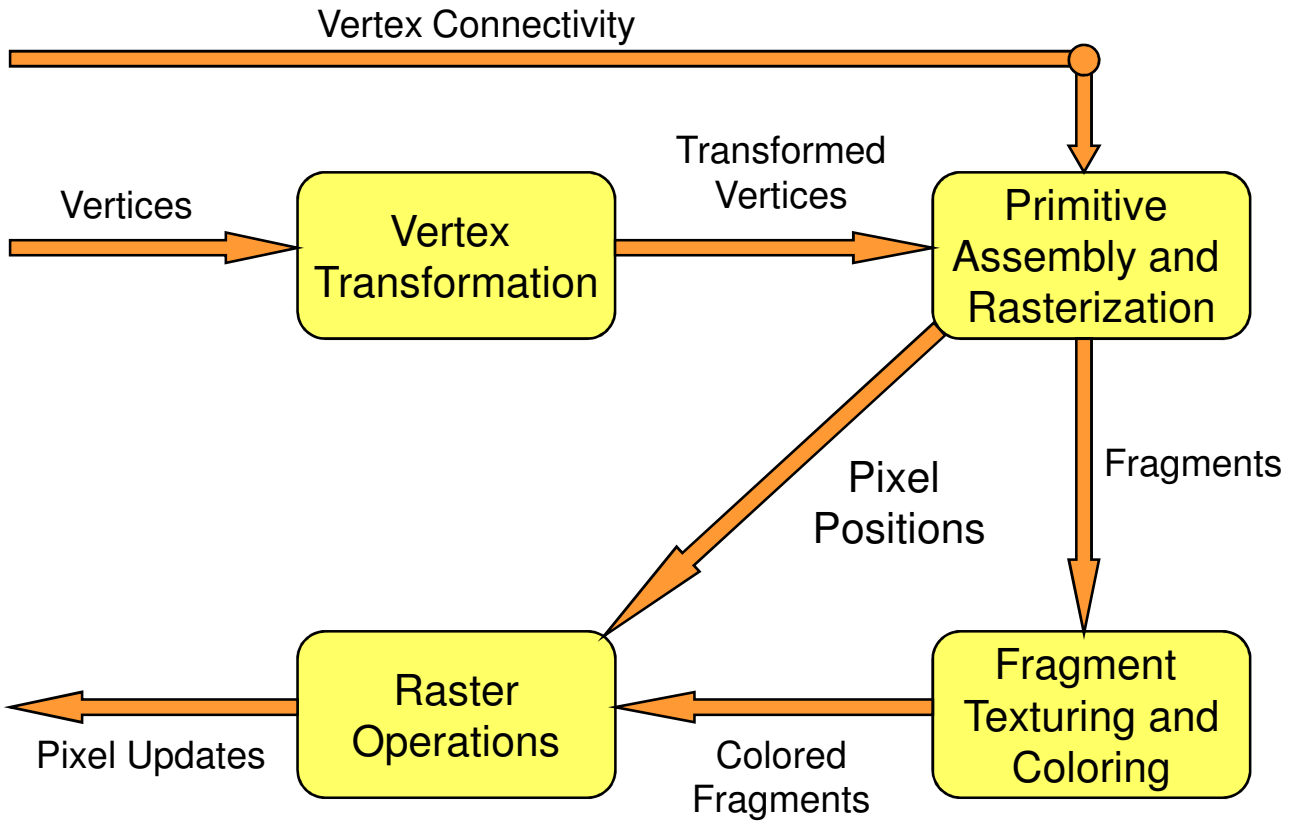


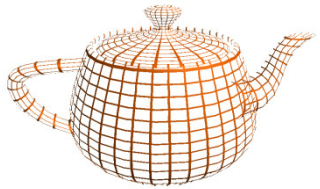
GLSL Sumário

- Evolução do Hardware Gráfico PCs
- Pipeline Gráfico Fixo
- Pipeline Gráfico Programável
  - Processador de Vértices
  - Processador de Fragmentos
- GLSL
  - Aplicação OpenGL
  - Shader I/O
  - Sintaxe
  - Exemplos
- GLSL vs. Cg vs. HLSL vs. Renderman



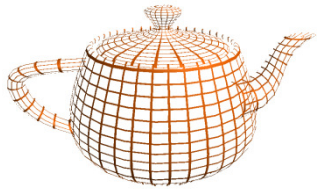
Pipeline Gráfico





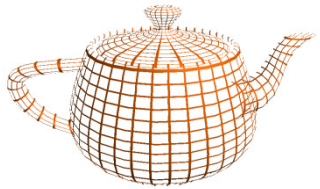
Pipeline Gráfico

- Vertex Transformation
  - Dados de entrada:
    - Vértices com as coordenadas tal como especificadas na aplicação, e outros atributos como normais, cores, coordenadas de textura.
    - Estado do OpenGL
  - Operações:
    - Transformação do vértice de acordo com as matrizes ModelView e Projection;
    - Transformação de normais (por exemplo: normalização);
    - Iluminação do vértice;
    - Geração/Transformação das coordenadas de textura.



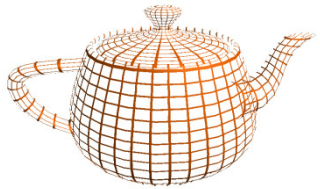
Pipeline Gráfico

- Primitive Assembly
  - Dados de entrada:
    - Vértices transformados
    - Informação de conectividade (GL_TRIANGLE, GL_QUAD,...)
  - Operações:
    - Construção das primitivas gráficas com os vértices já transformados
    - Clipping contra view frustum
    - Back face culling
    - Early Z culling



Pipeline Gráfico

- Rasterization
  - Dados de Entrada
    - Primitivas construídas na fase anterior
  - Operações
    - determina o conjunto de pixels cobertos por uma primitiva geométrica
    - Para cada pixel é calculado o seu conjunto de atributos por interpolação dos atributos dos vértices da primitiva
  - Resultado: conjunto de fragmentos



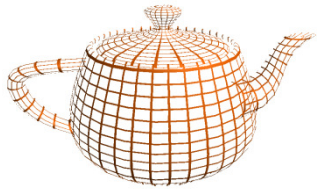
Pipeline Gráfico

- Rasterization (2)

  - Pixel vs. Fragment

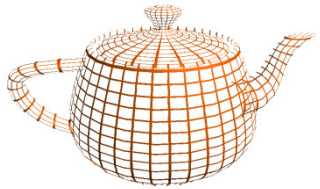
    - Pixel representa o conteúdo do frame buffer numa determinada posição: cor, profundidade, localização, ...

    - Fragmento: pixel potencial + atributos.



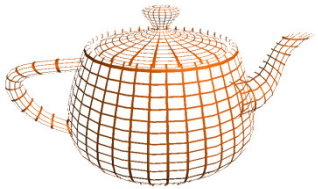
Pipeline Gráfico

- Texturização e Cor
  - Os dados de entrada desta fase são os valores interpolados na fase anterior para cada fragmento
  - Nesta fase aplicam-se as texturas e calcula-se a cor do fragmento.



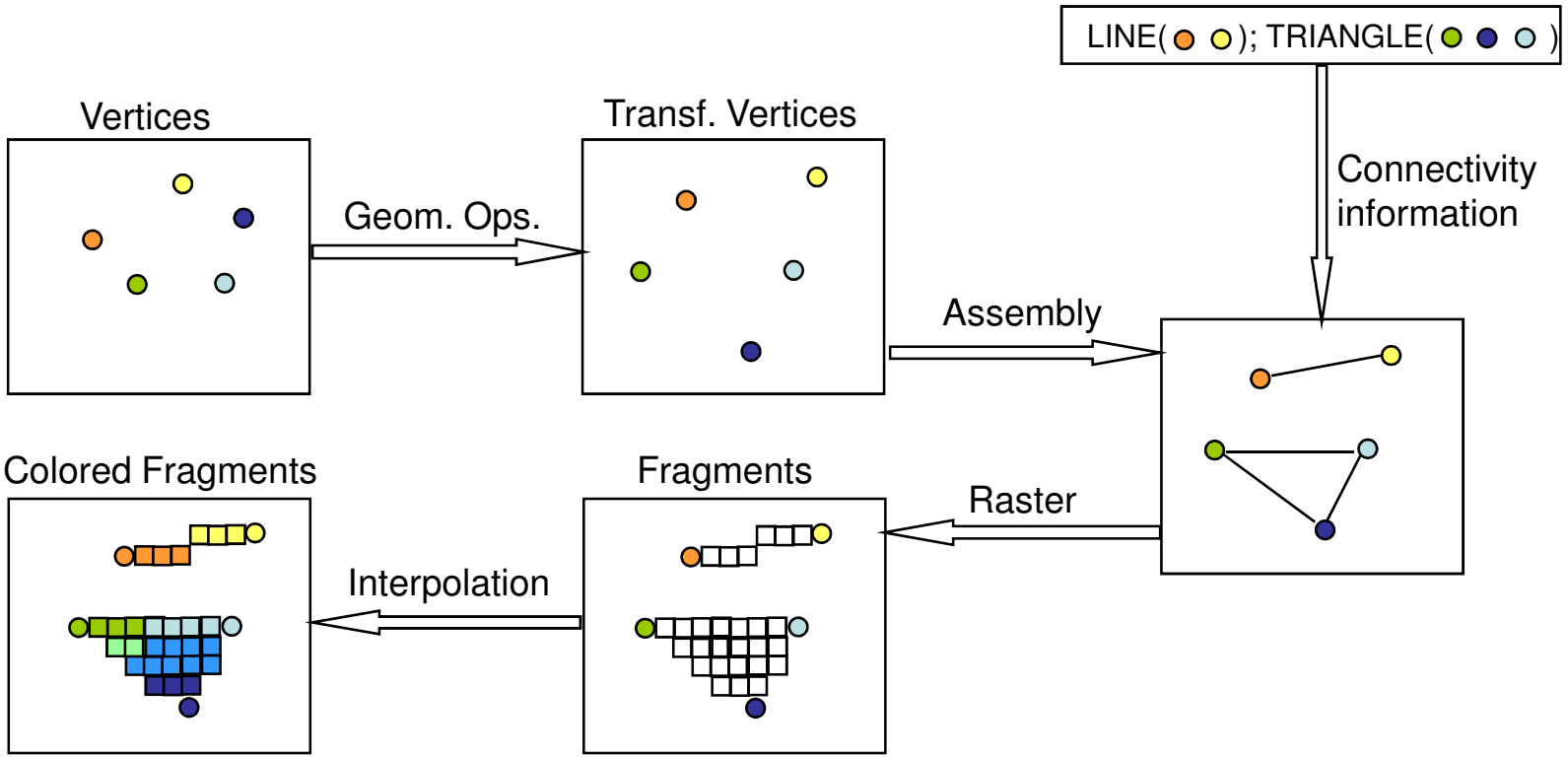
Pipeline Gráfico

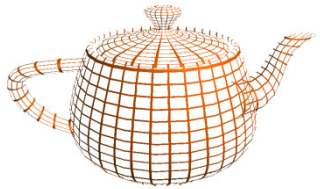
- Raster Operations
  - Dados de entrada
    - Localização do pixel
    - fragmentos processados (cor e profundidade)
  - Operações
    - Teste de scissor, alpha, stencil, profundidade (sem early Z cull)
    - Operações de blending: combina a cor do fragmento com a cor do pixel existente no color buffer



Pipeline Gráfico

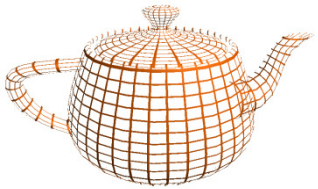
- Resumo Visual



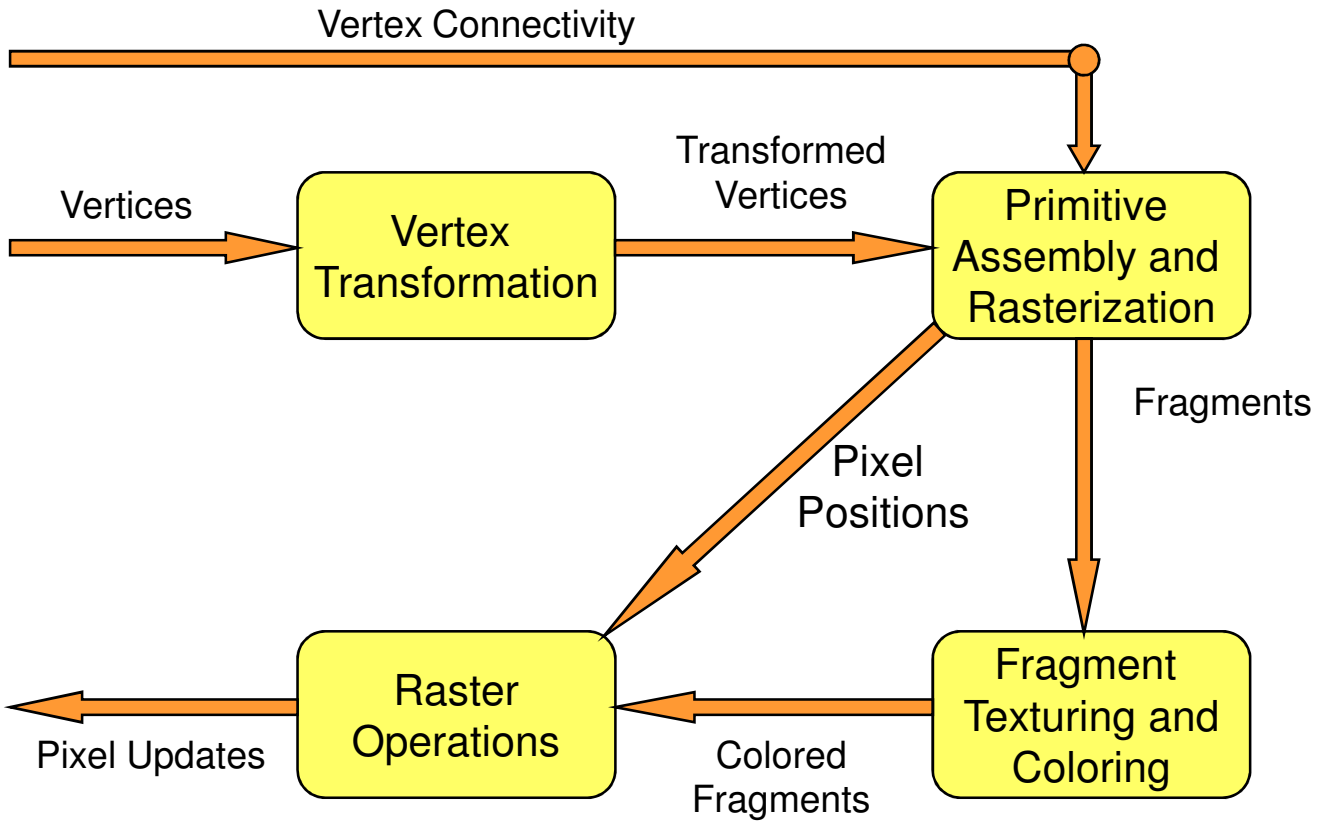


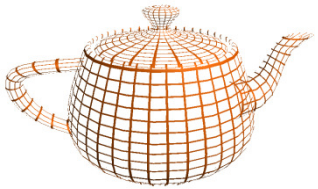
GLSL Sumário

- Evolução do Hardware Gráfico PCs
- Pipeline Gráfico Fixo
- Pipeline Gráfico Programável
  - Processador de Vértices
  - Processador de Fragmentos
- GLSL
  - Aplicação OpenGL
  - Shader I/O
  - Sintaxe
  - Exemplos
- GLSL vs. Cg vs. HLSL vs. Renderman

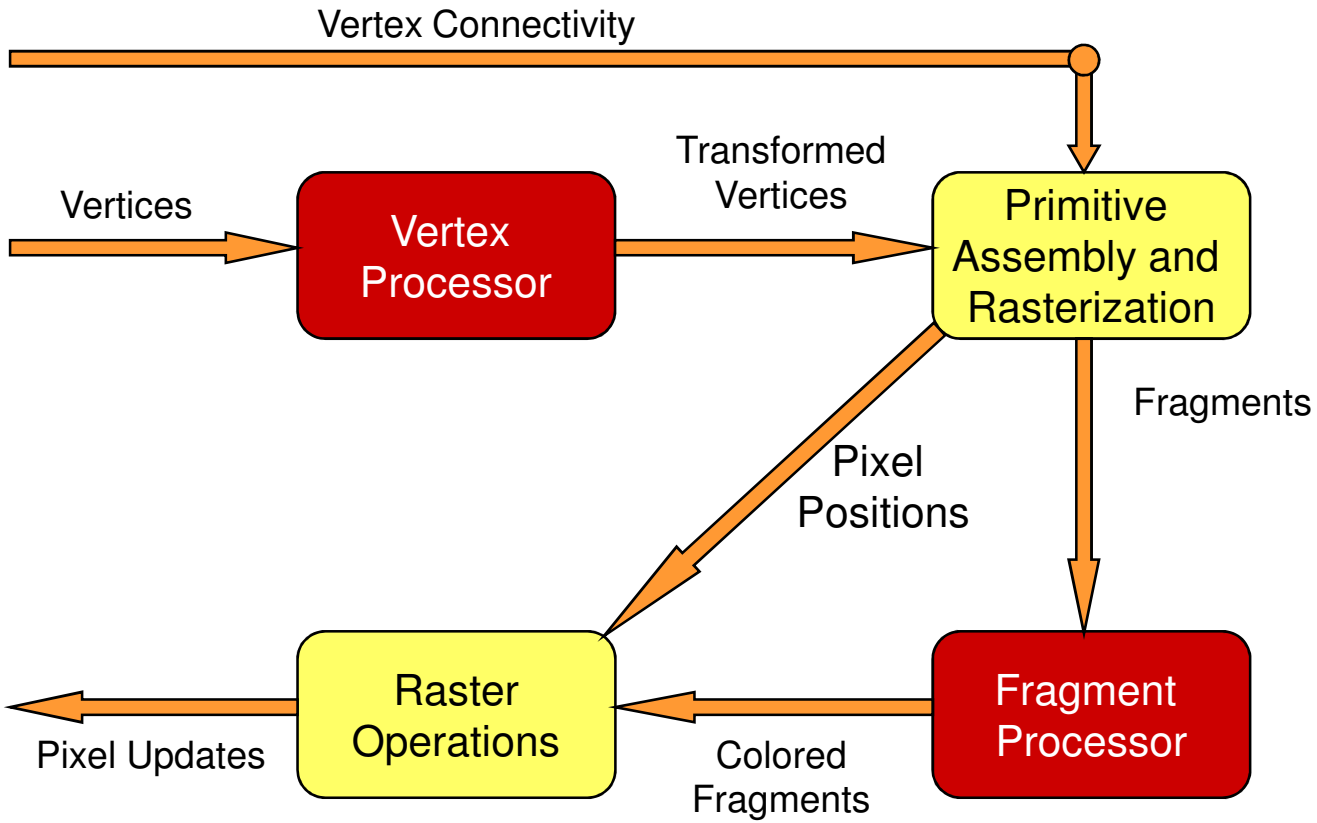


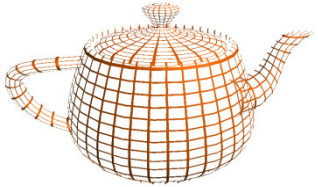
Pipeline Programável



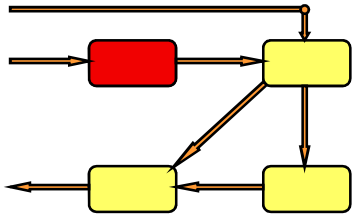


Pipeline Programável





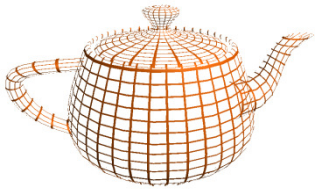
Processador de Vértices



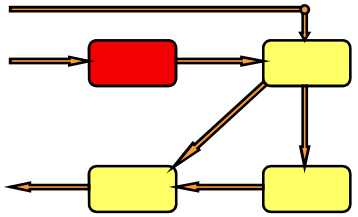
- Unidade programável que opera em vértices e nos dados associados (normais, coordenadas de textura, ...)

- Esta unidade é responsável por operações como por exemplo:

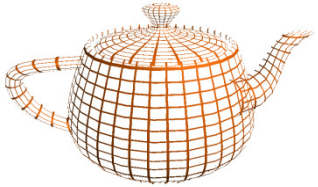
  - Transformação de vértices (matrizes modelview e projection)
  - Transformação de normais e sua normalização
  - Geração de coordenadas de textura
  - Transformação de coordenadas de textura
  - Iluminação (por vértice)
  - ...



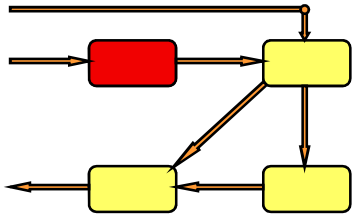
Processador de Vértices



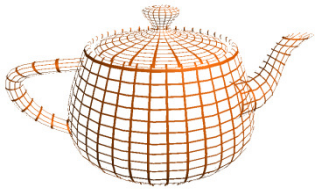
- Programas que correm nesta unidade são *vertex shaders*.
- Um *vertex shader* substitui toda a funcionalidade do *pipeline* fixo...
- ... logo tem de implementar todas as operações desejadas...
- ...i.e. não é possível ter um *vertex shader* para transformar as coordenadas do vértice e ter a iluminação calculada pela componente fixa.



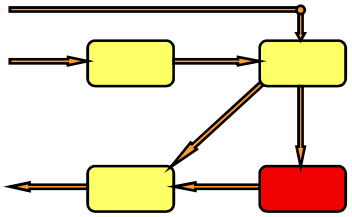
Processador de Vértices



- Este processador opera num vértice de cada vez, isoladamente, e sem conhecimento dos restantes vértices.
- Um *vertex shader* deve obrigatoriamente calcular uma posição e, opcionalmente, cor e outros atributos.
- O processador tem acesso ao estado do OpenGL actual, incluindo algumas variáveis pré-calculadas pelo próprio OpenGL.
- O processador de vértices pode aceder a texturas (no hardware mais recente).



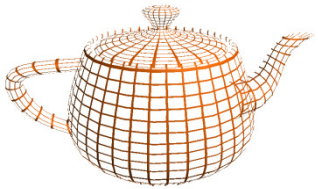
Processador de Fragmentos



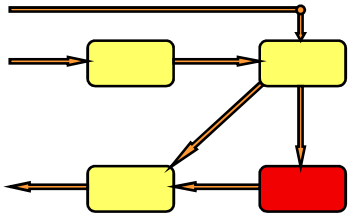
- Unidade programável que opera sobre fragmentos.

- Esta unidade é responsável por operações como por exemplo:

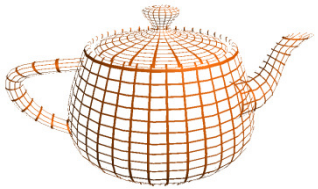
  - Operações sobre valores interpolados (cores, normais, coordenadas de textura, e outros atributos)
  - Acesso a texturas
  - Aplicação de texturas
  - Cálculo de Nevoeiro
  - Cálculo da cor



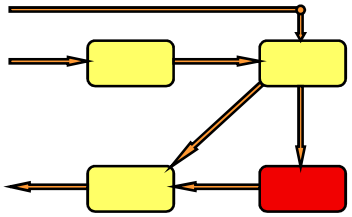
Processador de Fragmentos



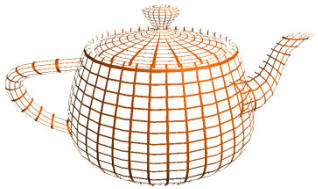
- Programas que correm nesta unidade são *fragment shaders*.
- Um *fragment shader* substitui toda a funcionalidade do *pipeline* fixo...
- ... logo tem de implementar todas as operações desejadas...
- ...i.e. não é possível ter um *fragment shader* para aceder a texturas e ter o nevoeiro calculado pela componente fixa.



Processador de Fragmentos

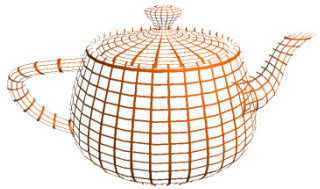


- Este processador opera num fragmento de cada vez, isoladamente, e sem conhecimento dos fragmentos vizinhos.
- Um fragment shader pode calcular cor e/ou profundidade ou alternativamente "descartar" o fragmento.
- O processador tem acesso ao estado do OpenGL actual, assim como algumas variáveis pré-calculadas pelo próprio OpenGL.
- O *fragment shader* não pode alterar a coordenada (x,y) do pixel a escrever.
- O *fragment shader* não tem acesso ao framebuffer.



GLSL Sumário

- Evolução do Hardware Gráfico PCs
- Pipeline Gráfico Fixo
- Pipeline Gráfico Programável
  - Processador de Vértices
  - Processador de Fragmentos
- GLSL
  - Aplicação OpenGL
  - Shader I/O
  - Sintaxe
  - Exemplos
- GLSL vs. Cg vs. HLSL vs. Renderman

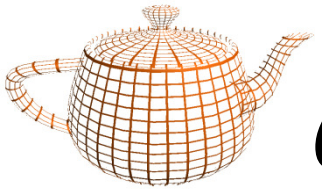


GLSL - Aplicação OpenGL

- Uma aplicação pode substituir a funcionalidade fixa relativamente aos vértices e/ou aos fragmentos, não sendo obrigada a implementar os dois.

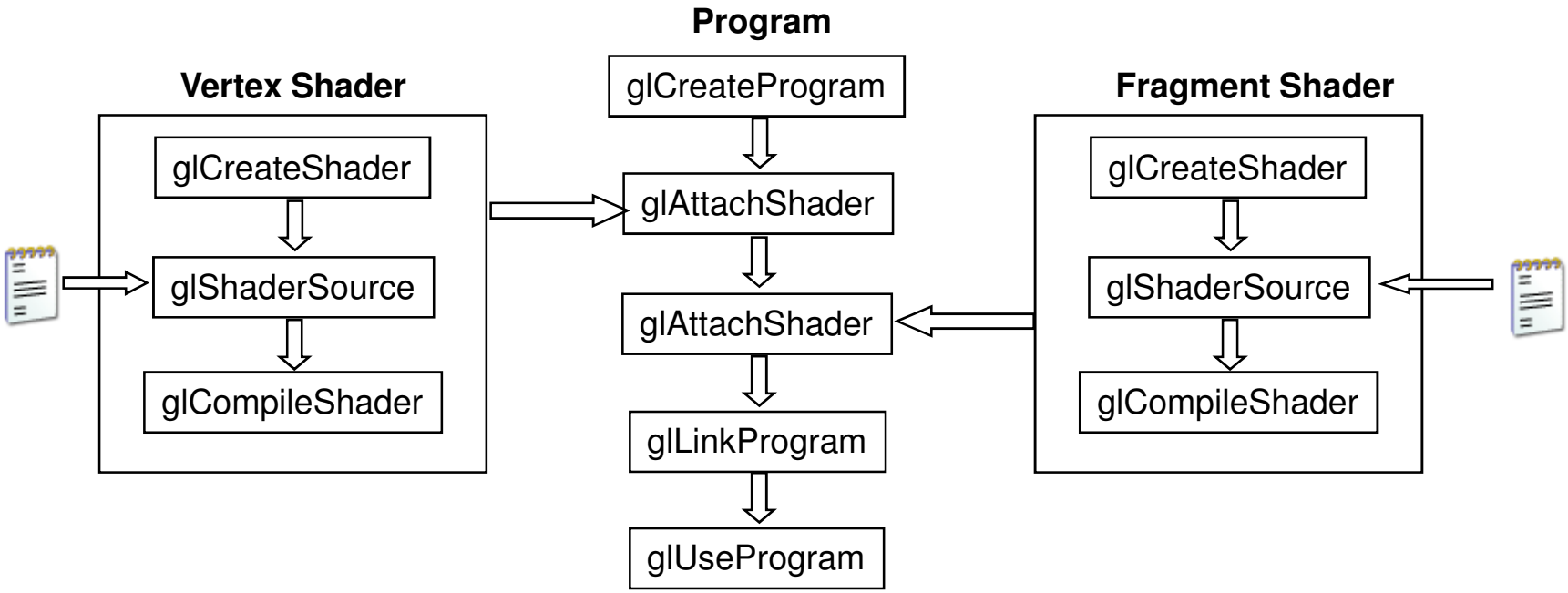
- Para cada *shader* é necessário proceder a uma compilação.

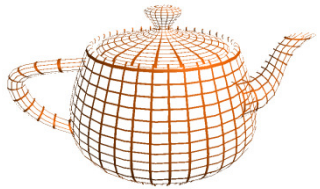
- Depois de compilados os *shaders* são incorporados num programa que depois será "linkado".



GLSL - Aplicação OpenGL

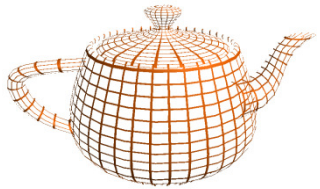
- OpenGL Setup





GLSL - Aplicação OpenGL

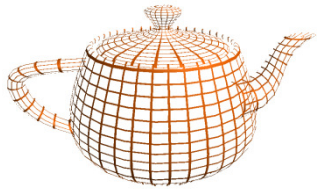
- Notas:
  - Um programa pode incluir somente um tipo de shader, por exemplo só incluir vertex shader. Nestes casos para o shader omisso será utilizada a funcionalidade fixa.
  - Depois de um programa estar "linkado", pode-se alterar o código dos shaders, e até compilá-los novamente, sem que isso altere o shader.
  - Se um programa estiver a ser utilizado, ao realizar a operação "link" de novo, substitui-se o programa actual (caso a operação de link tenha sucesso).



GLSL - Aplicação OpenGL

- Código para criação de shaders

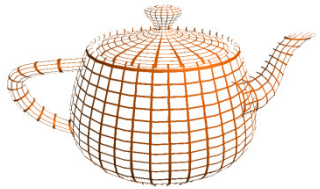
```
GLuint v,f,p;

v = glCreateShader (GL_VERTEX_SHADER);
f = glCreateShader (GL_FRAGMENT_SHADER);

vs = textFileRead("stripes.vert");
fs = textFileRead("stripes.frag");

glShaderSource (v, 1, &vs, NULL);
glShaderSource (f, 1, &fs, NULL);

glCompileShader (v);
glCompileShader (f);
```



GLSL - Aplicação OpenGL

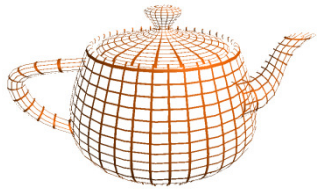
- Construção do programa

```
...
p = glCreateProgram ();

glAttachShader(p,v);
glAttachShader(p,f);

glLinkProgram (p);

glUseProgram (p);
```



GLSL - Aplicação OpenGL

- InfoLog
  - A cada shader ou programa está associado um log, com mensagens sobre o processo de compilação ou link, respectivamente.
  - Através deste log, para alem de ser possível detectar se as operações ocorreram sem erros, ainda pode ser dada informação relativamente à capacidade do hardware para executar o shader.

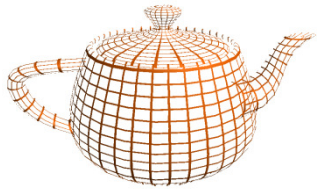


Comunicação Aplicação -> Shader

- Os shaders podem ser parametrizados com valores a receber da aplicação. Estes parâmetros são de leitura para os shaders, i.e. os shaders não podem alterar o seu valor.

- Existem duas categorias para estes valores:

  - Uniform
  - Attribute



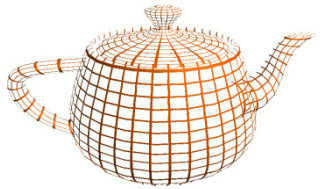
Comunicação Aplicação -> Shader

- Parâmetros **Uniform**

  - São parâmetros que não variam ao longo da primitiva, ou seja, não devem ser modificados entre glBegin e glEnd.

  - Ex: Posição da luz, índice de refracção, etc...

  - Acessíveis no vertex e fragment shader



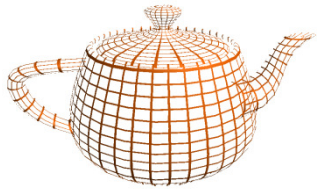
Comunicação Aplicação -> Shader

- Parâmetros **Attribute**

  - São tipicamente valores associados a vértices, tal como acontece com normais, ou coordenadas de textura.

  - Ex: A cada vértice pode estar associada uma tangente.

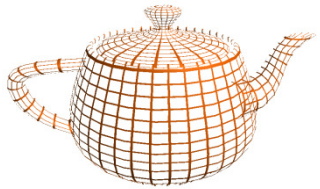
  - Só são acessíveis no vertex shader!



Comunicação Aplicação -> Shader

- Parâmetros **Attribute**: Exemplo

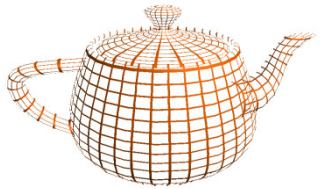
```
glBegin(GL_TRIANGLE_STRIP);
    glVertexAttrib1f(atrito,1.0);
    glVertex3f(1,2,3);
    glVertexAttrib1f(atrito,2.0);
    glVertex3f(4,5,6);
    ...
glEnd();
```



Comunicação Aplicação -> Shader

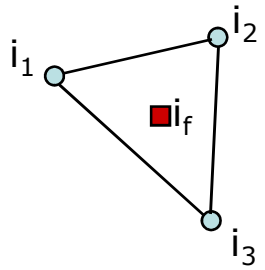
- Parâmetros **Attribute**: Vertex Arrays

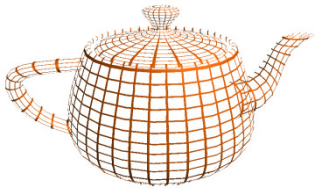
```
void setup()
{
     glVertexPointer(3, GL_FLOAT, 0, verts);
     glColorPointer (3, GL_FLOAT, 0, colors);
     glVertexAttribPointer (VELOCIDADE, 3, GL_FLOAT,
                                    GL_FALSE, 0, velocities);
     glEnableClientState (GL_VERTEX_ARRAY);
     glEnableClientState (GL_COLOR_ARRAY);
     glEnableVertexAttribArray (VELOCIDADES);

}
```



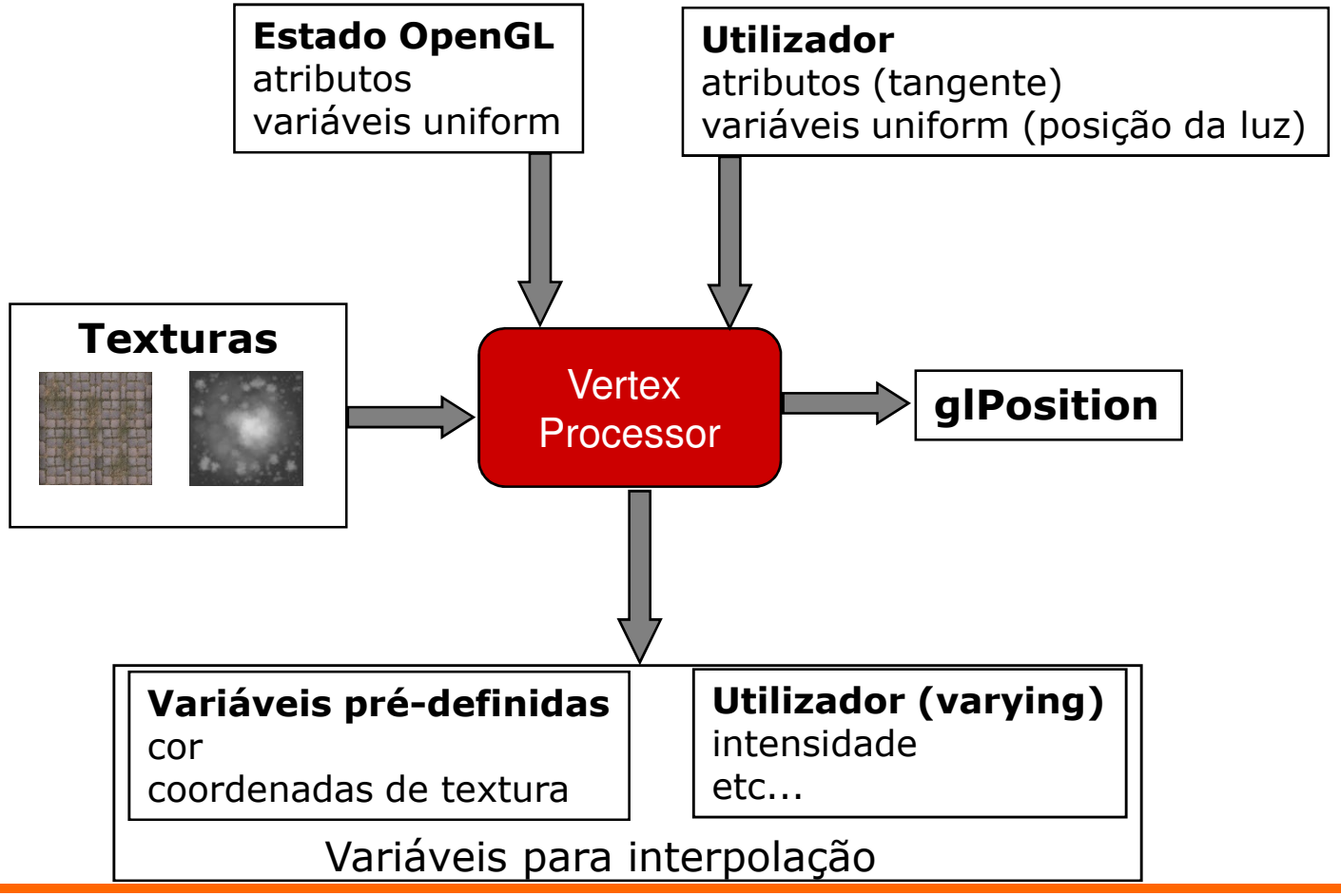
Comunicação Vertex -> Fragment

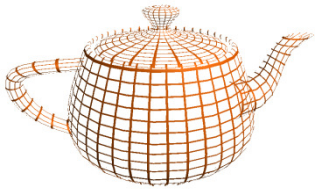
- Variáveis **varying**
  - Vertex Shader
    - declaração, ex: **varying** float intensidade;
    - atribui um valor à variável (i_1,i_2,i_3)
  - Fragment shader
    - declaração idêntica ao vertex shader
    - recebe valor interpolado (i_f)
    - variável só de leitura



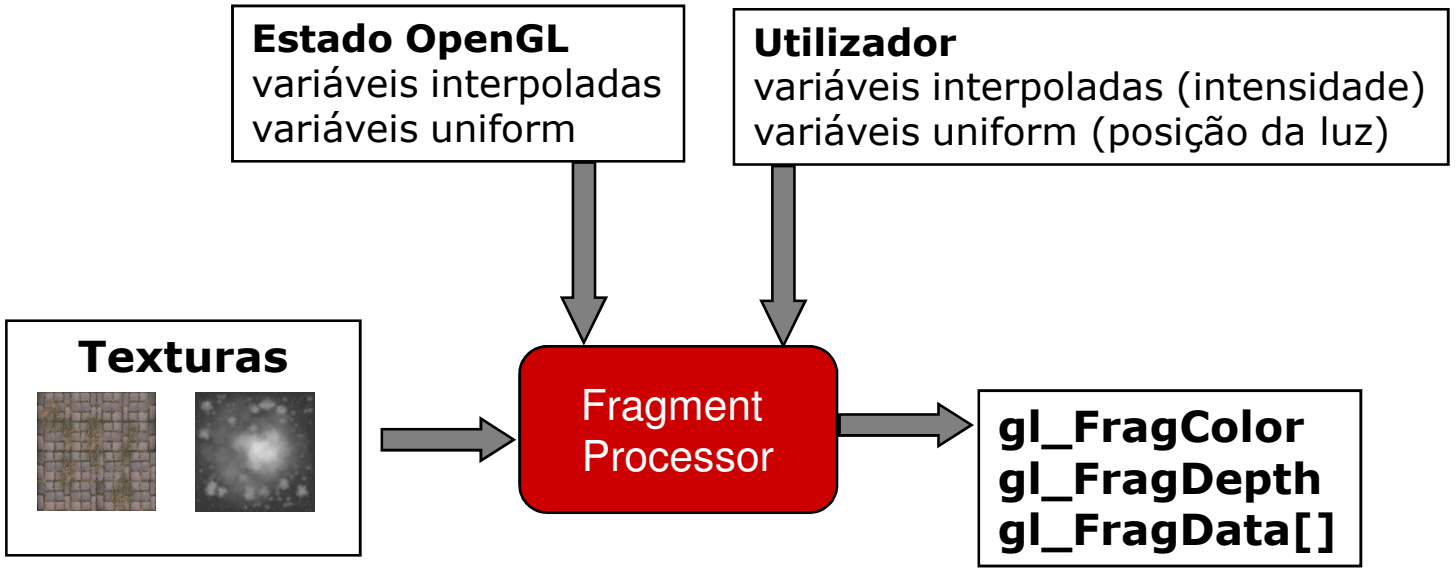


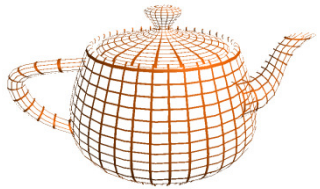
Vertex Shader I/O





Fragment Shader I/O





GLSL - Tipos de Dados

- Escalares
  - `float`
  - `int`
  - `bool`

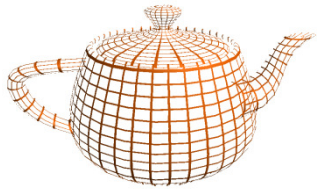
```
float a,b=2.0;
int c,d=1;
bool e,f= true;
```

- Vectores
  - `vec{2,3,4}`
  - `ivec{2,3,4}`
  - `bvec{2,3,4}`

```
vec3 v = vec3(1,2,3);
vec3 a = v, b = vec3(4);
```

- Matrizes
  - `mat{2,3,4}`
  -

```
mat3 m = mat3(1.0); // matriz identidade
mat3 n = mat3(v,a,b);
```



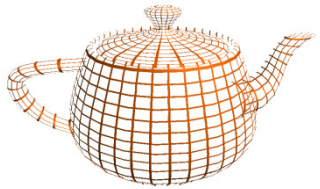
GLSL - Tipos de Dados

- Selectores para Vectores
  - x,y,z,w
  - r,g,b,a
  - s,t,p,q

```
- vec3 v = vec3(1.0,2.0,3.0);
- float f = v.x;                    // f = 1.0
- float y = v[1];                   // y = 2.0
- vec3 u = v.zyx                    // u = (3.0,2.0,1.0)
```

- Selectores para Matrizes

```
- mat2 m = mat2(1.0);          // m = matriz identidade
- vec2 v = m[0];               // v = (1.0,0.0) primeira coluna
- float f = m[0][0]            // f = 1.0
```



GLSL - Tipos de Dados

- Estructuras
  - Definição

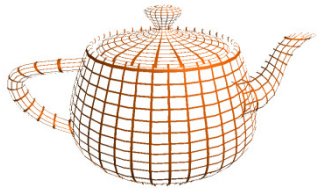
```
struct luz {
     vec3 pos;
     vec3 cor;
};
```

  - Declaração e inicialização

```
luz luz1;
luz luz2 = luz(vec3(1.0,2.0,3.0),vec3(1.0,0.0,0.0));
```

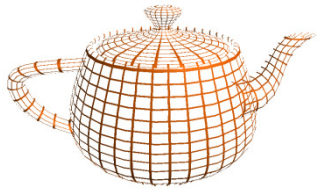
  - Utilização

```
luz1.pos = vec3(1.0,2.0,3.0);
```



GLSL - Tipos de Dados

- Arrays
  - `vec3 pontos[5];`
  - `vec4 vertices[];`
  - No caso de um array não definir a sua dimensão, o compilador procura o índice mais elevado no shader.
  - Neste caso os índices têm de ser constantes.



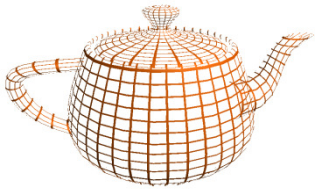
GLSL - Tipos de Dados

- Casting
  - Realizado através de construtores

```
float a = 3.0;
int b = int(a);
```

- Constantes

```
const int texturas = 10;
```



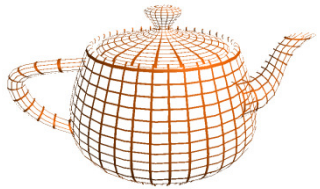
GLSL - Funções

- Semelhante ao C

```
vec3 f() {
     ...
}
```

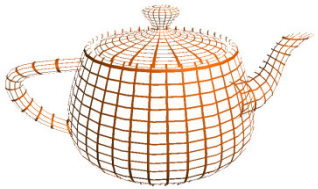
- Tipos dos parâmetros
  - in
  - out
  - inout

```
vec3 f(in vec3 a, out vec3 b) {
     ...
}
```



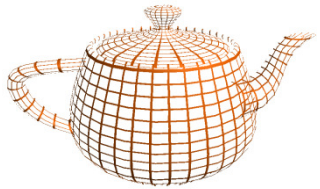
GLSL - Controle de fluxo

- Instruções iguais ao C
  - if else
  - for
  - while
  - do while



GLSL - Exemplos

- "Olá Mundo"
- Cor
- Toon Shader



GLSL – Olá Mundo

- A funcionalidade fixa, *ff*, determina que um vértice deve ser transformado através da seguinte operação:
  - Vres = Matriz PROJECTION * Matriz MODELVIEW * vertex
- O vertex shader recebe os vértices através da variável:

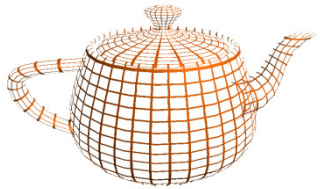
```
attribute vec4 gl_Vertex
```

- O vertex shader deve transformar o vértice recebido e escrever o resultado em:

```
vec4 gl_Position;
```

- Variáveis de estado do OpenGL acessiveis:

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
```

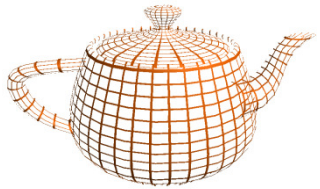


GLSL – Olá Mundo

- Para obter uma funcionalidade semelhante à ff, o vertex shader deverá então calcular gl_Position utilizando uma das seguintes opções:

```
gl_Position = gl_ProjectionMatrix *
                      (gl_ModelViewMatrix * gl_Vertex);

gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

gl_Position = ftransform();
```

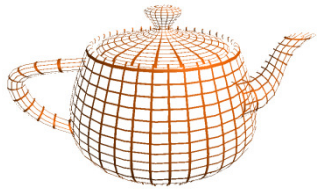
- Esta última alternativa garante a funcionalidade idêntica à ff.



GLSL - Olá Mundo

- Vertex Shader

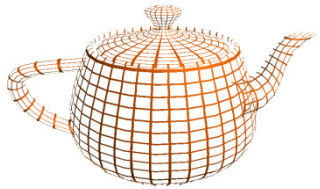
```
void main (void) {

    gl_Position = ftransform();
}
```



GLSL - Olá Mundo

- O Fragment shader é responsável por escrever a variável gl_FragColor.

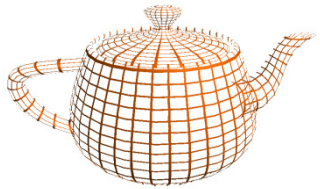
```
void main(void) {
    gl_FragColor = vec4(1,0,0,1);
}
```



GLSL - Cor

- Código OpenGL

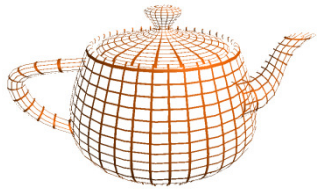
```
glBegin(GL_QUADS);
      glColor3f(1,0,0);
      glVertex3f(-1,-1,0);

      glColor3f(0,1,0);
      glVertex3f(1,-1,0);

      glColor3f(0,0,1);
      glVertex3f(1,1,0);

      glColor3f(1,1,0);
      glVertex3f(-1,1,0);
glEnd();
```



GLSL - Cor

- Variáveis de estado acessiveis nos shaders:

```
attribute vec4 gl_Color; // só vertex shader

struct gl_MaterialParameters {
    vec4 emission;   // Ecm
    vec4 ambient;    // Acm
    vec4 diffuse;    // Dcm
    vec4 specular;   // Scm
    float shininess; // Srm
};
uniform gl_MaterialParameters gl_FrontMaterial;
uniform gl_MaterialParameters gl_BackMaterial;
```



GLSL - Cor

- Vertex Shader lê cor especificada na aplicação (glColor) em:

```
attribute vec4 gl_Color;
```

- Vertex Shader escreve em:

```
varying vec4 gl_FrontColor;
varying vec4 gl_BackColor;
```

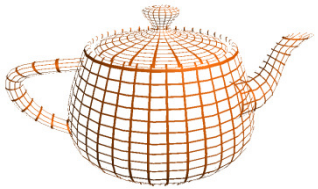
- Fragment Shader lê cor interpolada em:

```
varying vec4 gl_Color;
```

  - Nota: não é a mesma variável que está acessível no vertex shader. Esta variável é interpolada pelo OpenGL a partir das variáveis `gl_FrontColor` ou `gl_BackColor`.

- Fragment Shader escreve em:

```
vec4 gl_FragColor
```



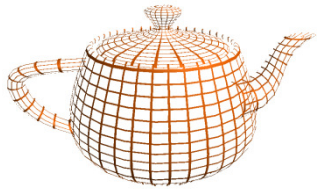
GLSL - Cor

- Exemplo

```
// Aplicação OpenGL

glBegin(GL_QUADS);
          glColor3f(1,0,0);
          glVertex3f(-1,-1,0);
          glColor3f(0,1,0);
          glVertex3f(1,-1,0);
          glColor3f(0,0,1);
          glVertex3f(1,1,0);
          glColor3f(1,1,0);
          glVertex3f(-1,1,0);
glEnd();
```

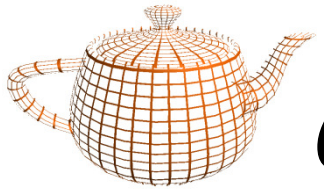
```
// Vertex Shader

void main(void) {

          gl_FrontColor = gl_Color;
          gl_Position = ftransform();
}
```

```
// Fragment Shader

void main()
{
          gl_FragColor = gl_Color;
}
```



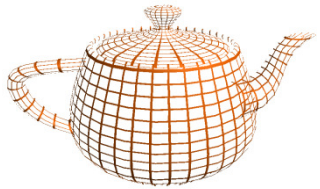
GLSL - Toon Shading

- O efeito "Cartoon" pode ser aproximado de uma forma simples criando um shader com um número limitado de cores.
- Pretende-se preservar a contribuição especular.
- Cor escura para zonas fronteira
- Número reduzido de cores para os restantes casos



GLSL - Toon Shading

- Cálculo da cor baseado na intensidade reflectida
- Número reduzido de cores => discretização
- Maior intensidade => cor mais clara



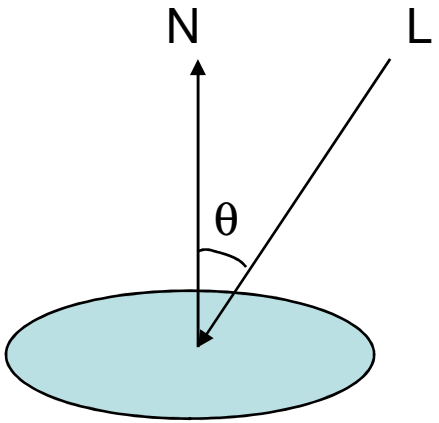
GLSL – Toon Shading

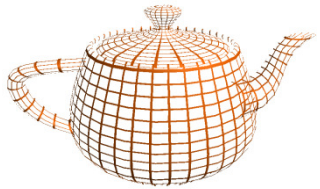
- Reflexão difusa (Lambert)
  - A intensidade reflectida de um objecto é proporcional ao ângulo entre a direcção da luz e a normal do objecto.

Intensidade da luz

$$I = I_p * K_d * \cos(\theta)$$

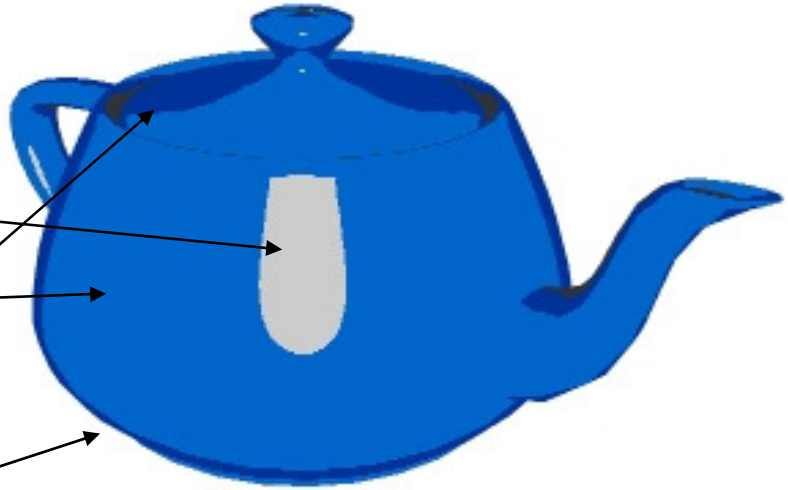
coeficiente de reflexão difusa

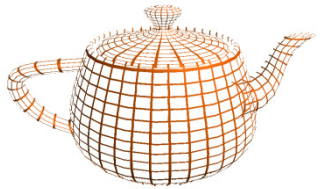




GLSL – Toon Shading

- Algoritmo
  - if (cos(θ) > 0.98)
    - cor = (0.75, 0.75, 1.0)
  - else if (cos(θ) > 0.65
    - cor = (0.0, 0.25, 1.0)
  - else if (cos(θ) > 0.25)
    - cor = (0.0, 0.2, 0.6)
  - else
    - cor = (0.2, 0.2, 0.2)

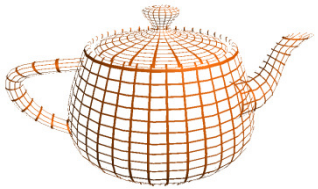




GLSL - Toon Shading

- Vertex Shader
  - Transforma vértices e normais (espaço câmara)

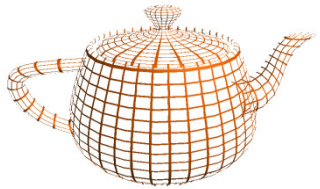
```
// Vertex Shader

varying vec3 normal;

void main(void) {

   normal = gl_NormalMatrix * gl_Normal;
   normal = normalize(normal);
   gl_Position = ftransform();
}
```



GLSL - Toon Shading

- Fragment Shader
  - normaliza normal
  - calcula cosseno
  - aplica algoritmo

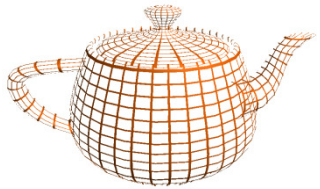
```
varying vec3 normal;

vec4 c0 = (0.2, 0.2, 0.2, 1.0);
...
vec4 c3 = (0.75, 0.75, 0.75, 1.0);

void main() {

        vec3 n = normalize(normal);
        co = dot(vec3(0,0,1), n);
        if (co > 0.98)
                color = c3
        else if (co > 0.65)
                color = c2;
        else if (co > 0.25)
                color = c1;
        else
                color = c0;

        gl_FragColor = color;
}
```



GLSL - Toon Shading

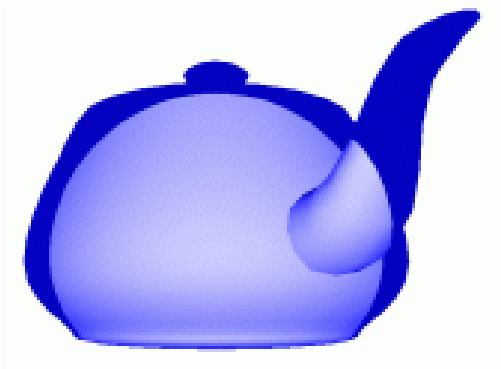
- Utilização de variáveis **Uniform**
  - cores e limites

```
varying vec3 normal;

uniform vec4 c0,c1,c2,c3;
uniform float t0,t1,t2;

void main() {

        vec3 n = normalize(normal);
        co = dot(vec3(0,0,1), n);
        if (co > t2)
                color = c3
        else if (co > t1)
                color = c2;
        else if (co > t0)
                color = c1;
        else
                color = c0;

        gl_FragColor = color;
}
```



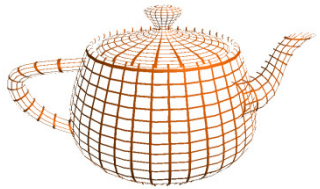
GLSL – Mais Exemplos

- Luzes

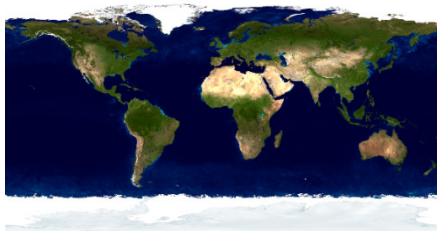


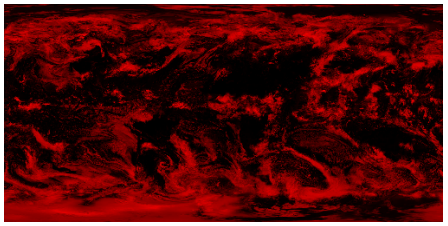
com shaders

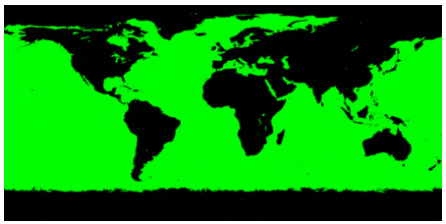
ff

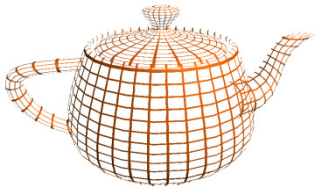


GLSL – Mais Exemplos

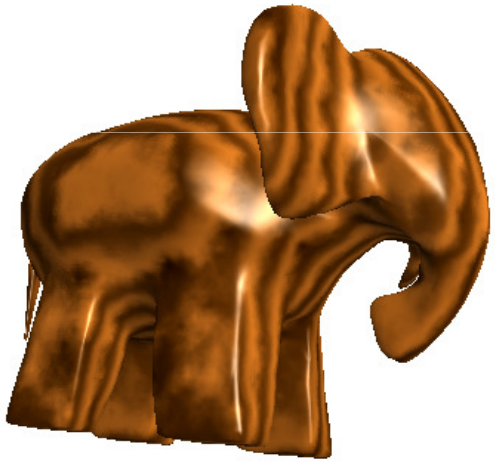


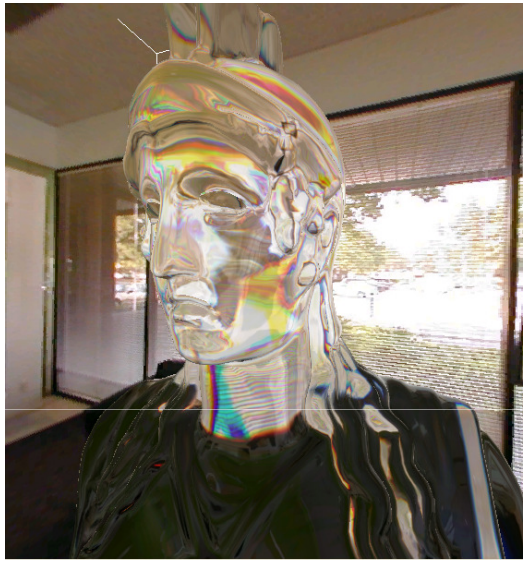


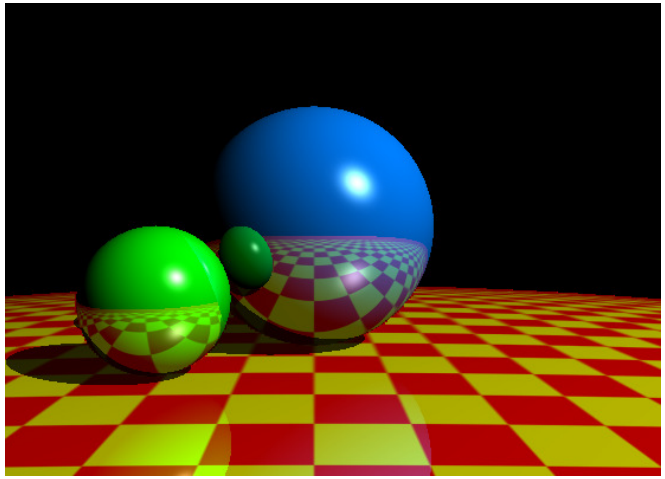


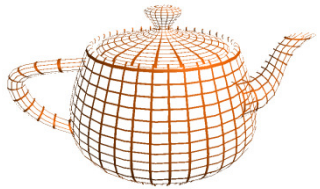


GLSL - Mais Exemplos



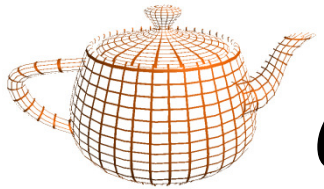






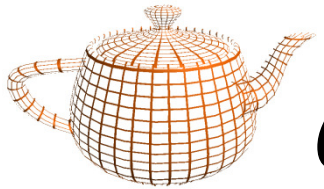
GLSL Sumário

- Evolução do Hardware Gráfico PCs
- Pipeline Gráfico Fixo
- Pipeline Gráfico Programável
  - Processador de Vértices
  - Processador de Fragmentos
- GLSL
  - Aplicação OpenGL
  - Shader I/O
  - Sintaxe
  - Exemplos
- GLSL vs. Cg vs. HLSL vs. Renderman



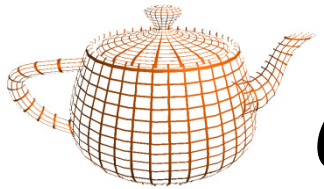
GLSL vs. HLSL vs. Cg vs. Renderman

- GLSL vs. HLSL vs. Cg
  - Sintaxe muito semelhante (baseado no C)
  - Idêntica divisão de tarefas entre vértices e pixels
  - Compilação é diferente
    - GLSL -> runtime, Driver
    - HLSL e Cg -> pré-compilação, biblioteca exterior ao driver



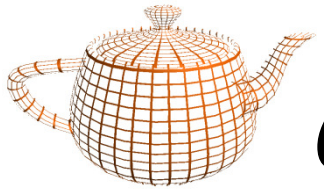
GLSL vs. HLSL vs. Cg vs. Renderman

- GLSL vs. HLSL vs. Cg
  - Plataformas
    - OpenGL => GLSL e Cg
    - D3D => HLSL e Cg
  - Variáveis varying
    - GLSL => nome a definir pelo programador
    - HLSL e Cg => slots pré-definidos (TEXCOORD[0], ...)



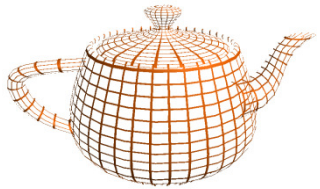
GLSL vs. HLSL vs. Cg vs. Renderman

- Renderman
  - Lançamento (muito) anterior (1988)
    - GLSL – 2001
    - HLSL – 2002
    - *Cg* – 2002
  - Objectivo: Qualidade de Imagem (vs. Real-time)
  - Conceitos com varying e uniform foram herdados pelas novas linguagens



GLSL vs. HLSL vs. Cg vs. Renderman

- Renderman
  - Definição conceptual de shaders
    - Displacement shaders
    - Surface shaders
    - Light shaders
    - Volume shaders
    - Image shaders
- GLSL, HLSL, Cg
  - Definição com objectivo de mapear no hardware actual
    - Vertex shader
    - Fragment shader



GLSL - Bibliografia, Ferramentas

- "OpenGL Shading Language", Randi Rost, Addison-Wesley.
- "OpenGL Shading Language Spec", *www.opengl.org/documentation/glsl*
- "OpenGL 2.0 Spec", *www.opengl.org/documentation/current_version/*

- RenderMonkey *www.ati.com/developer*
- Shader Designer *www.typhoonlabs.com*
- *FXComposer developer.nvidia.com*
- ShaderGen *developer.3dlabs.com*