

# Computação Gráfica e Interfaces

2017-2018  
Fernando Birra

# WebGL (Shaders)

2017-2018  
Fernando Birra

# Objetivos

- Aplicações típicas de:
  - vertex shaders
  - fragment shaders
- GLSL

# Aplicações típicas de Vertex Shaders

- Modelação (Transf. coordenadas) e Projeção
- “Mover” vértices
  - Morphing - Transformação/Deformação dum modelo noutro
  - movimentos ondulatórios
  - fractais
- Iluminação

# Aplicações típicas de Fragment Shaders

- Iluminação aplicada ao nível dos fragmentos



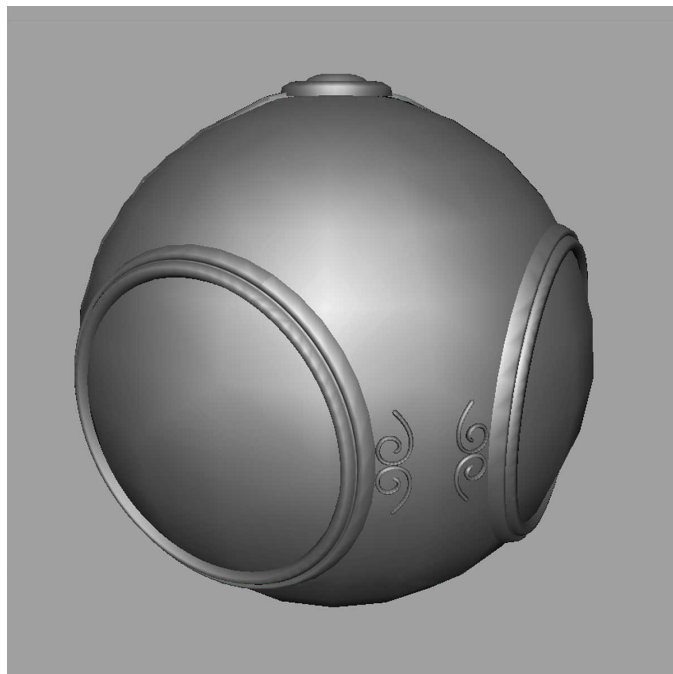
iluminação por vértice



iluminação por fragmento

# Aplicações típicas de Fragment Shaders

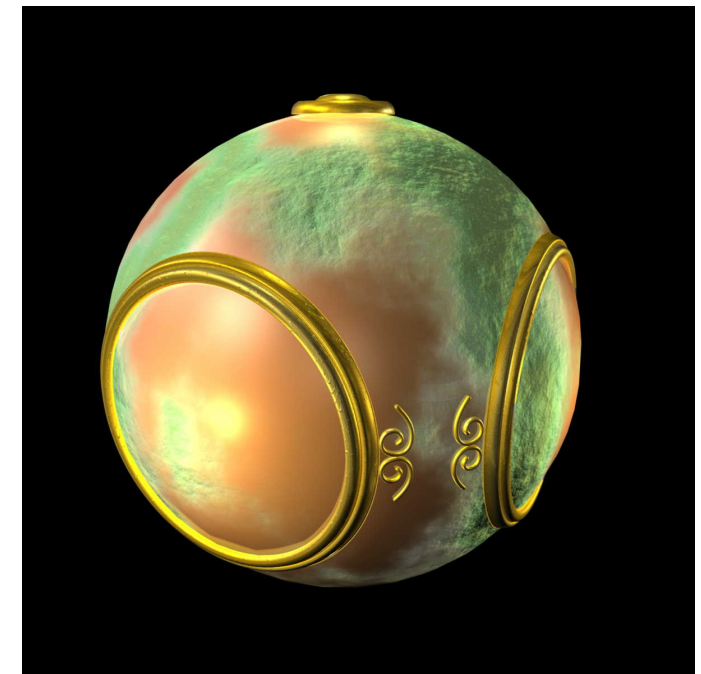
- Mapeamento de texturas



sem texturas



environment  
mapping



bump mapping

# Shaders

- Os primeiros shaders eram programados diretamente em assembler (do GPU)
- Disponibilizados no OpenGL (com o mecanismo de extensões)
- Cg (C for Graphics) - linguagem da nVIDIA para a escrita de shaders, baseada em C
- GLSL - OpenGL Shading Language

# GLSL

- OpenGL Shading Language
- OpenGL 2.0 (e posteriores)
- Linguagem de alto nível (ao estilo do C)
- Tipos de dados úteis para CG:
  - vetores, matrizes e samplers
- A partir do OpenGL 3.1, todas as apps têm que fornecer os seus shaders



# Vertex Shader

um valor por vértice

tipo

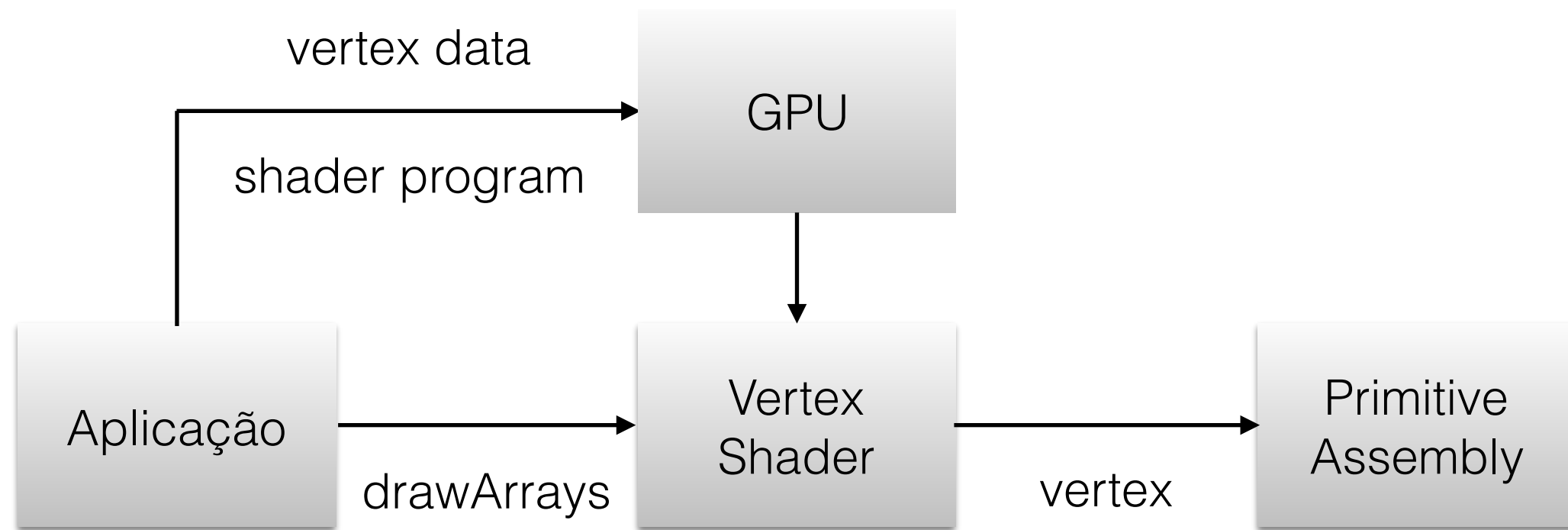
valor fornecido  
pela aplicação e colocado  
num buffer

```
attribute vec4 vPosition;  
  
void main(){  
    gl_Position = vPosition;  
}
```

variável *built-in*.  
Output obrigatório do Vertex  
Shader

**Vertex shader:** produz vértices  
em coordenadas de recorte

# Modelo de Execução



# Fragment Shader

precisão do frame buffer

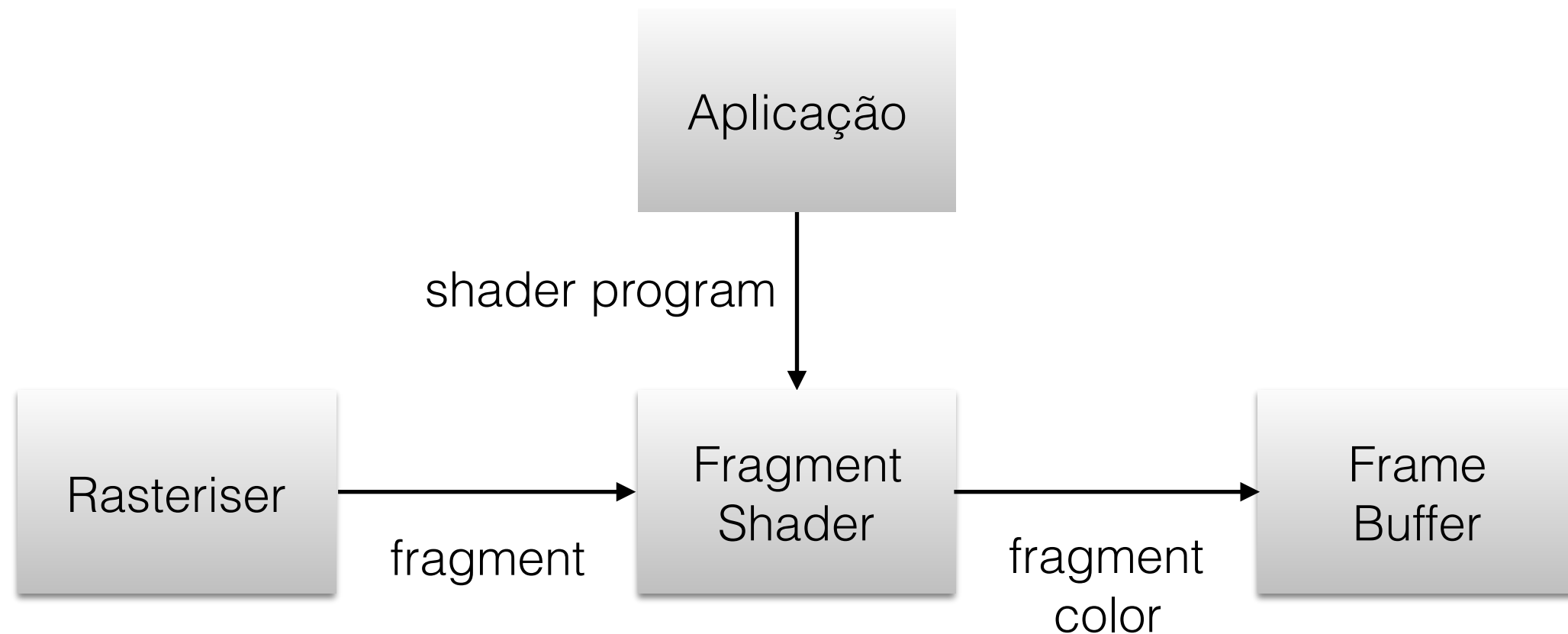
```
precision mediump float;
```

```
void main(){  
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);  
}
```

variável *built-in*.  
Output obrigatório do  
Fragment Shader

**Fragment shader:** produz cores  
para pixels

# Modelo de Execução



# Tipos de Dados

- Tipos C: int, float, bool
- Vetores:
  - float vec2, vec3, vec4
  - int (ivec) e boolean (bvec)
- Matrizes (mat2, mat3, mat4):
  - guardadas por colunas
  - acesso standard m[linha][coluna]
- Construtores ao estilo C++:
  - `vec3 a = vec3(1.0, 2.0, 3.0)`
  - `vec2 b = vec2(a)`

# Apontadores

- Não há apontadores em GLSL
- Podem usar-se registos (struct) C como resultados de funções
- Os tipos matN e vecN podem ser passados e retornados de funções:
  - `mat3 inverse(mat3 a)`
- Passagem de parâmetros por valor

# Qualificadores

- GLSL tem alguns dos qualificadores do C++, tal como `const`
- Há qualificadores novos devido ao modelo de execução ser diferente
- Variáveis podem variar:
  - Uma vez por primitiva
  - Uma vez por vértice
  - Uma vez por fragmento
  - Em qualquer altura na aplicação
- Os atributos dos vértices podem ser interpolados na fase de varrimento (rasterization) originando atributos dos fragmentos

# Qualificador `attribute`

- As variáveis com o qualificador `attribute` podem variar uma vez por cada vértice (são atributos dos vértices)
- Existem algumas variáveis *built-in*, tais como `gl_Position` (em OpenGL há um maior número)
- Exemplos:

```
attribute float temperature  
attribute vec3 velocity
```

cada vértice, ao ser processado pelo vertex shader, terá disponível o valor do atributo respetivo, previamente disponibilizado pela aplicação num buffer



# Qualificador `uniform`

- As variáveis com o qualificador `uniform` são constantes durante o desenho da primitiva
- Podem ser modificadas pela aplicação e enviadas aos shaders (ao programa)
- Não podem ser modificadas pelos shaders
- Usadas para passar informação “global”. Por exemplo:
  - tempo (numa simulação)
  - posição da luz, cor da luz, matriz de projeção, etc.

```
uniform float time
```

```
uniform vec3 lightPos
```

Os valores permanecem constantes durante todo o desenho das primitivas.

Estão acessíveis quer no vertex shader, quer no fragment shader.

# Qualificador `varying`

- variáveis que são passadas do vertex shader para o fragment shader
- São interpoladas automaticamente durante o varrimento
- No WebGL, o qualificador GLSL usado é o mesmo em ambos os shaders. Exemplos:

```
varying vec4 color
```

```
varying float temperature
```

Os valores atribuídos no vertex shader a cada vértice determinam o valor acessível pelo fragment shader (por interpolação)

# Convenção para os nomes

- **atributos** passados ao vertex shader têm os nomes começados por v (vPosition, vColor, vTemperature), quer na aplicação, quer no shader

- No shader:

```
attribute vec4 vPosition;
```

- Na aplicação:

```
var vPosition = gl.getAttributeLocation(program, "vPosition");
```

# Convenção para os nomes

- variáveis passadas do vertex shader para o fragment shader começam com f (fColor, fTemperature).
- Usa-se o mesmo nome nos dois shaders

No vertex shader:

```
attribute float vTemperature;  
varying float fTemperature;  
  
void main() {  
    fTemperature = vTemperature;  
    gl_Position = ...  
}
```

No fragment shader:

```
varying float fTemperature;  
  
void main() {  
    gl_FragColor = colorFromTemp(fTemperature);  
}
```

função definida pelo  
programador do shader numa  
aplicação hipotética

# Convenção para os nomes

- As variáveis `uniform` têm nome sem prefixo, mas com o sufixo `Loc`
- Na aplicação:

```
var twistLoc = gl.getUniformLocation(program, "twist");
```

- Num shader:

```
uniform float twist;  
  
void main() {  
    gl_Position = posFromTwist(twist);  
}
```

função definida pelo programador do shader  
numa aplicação hipotética

# Passagem de valores para os atributos (attribute)

```
var cBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER, cBuffer);  
gl.bufferData(gl.ARRAY_BUFFER, flatten(colors), gl.STATIC_DRAW);  
  
var vColor = gl.getAttribLocation(program, "vColor");  
gl.vertexAttribPointer(vColor, 3, gl.FLOAT, false, 0, 0);  
gl.enableVertexAttribArray(vColor);
```

The number of components per attribute. Must be 1,2,3,or 4. Default is 4.

Specifies the data type of each component in the array.

Specifies the offset in bytes between the beginning of consecutive vertex attributes.

**vertexAttribPointer(index, size, type, normalize, stride, offset)**

Index of target attribute in the buffer bound to gl.ARRAY\_BUFFER

if gl.TRUE, values are normalised when accessed

Specifies an offset in bytes of the first component of the first vertex attribute in the array. Default is 0 which means that vertex attributes are tightly packed.

# Passagem de valores constantes (uniform)

Na aplicação:

```
var color = vec4(1.0, 0.0, 0.0, 1.0);  
colorLoc = gl.getUniformLocation(program, "color");  
gl.uniform4fv(colorLoc, color);
```

usando o tipo de dados vec4

Pertence a uma família de métodos: uniform\*

No fragment shader (de forma idêntica no vertex shader):

```
uniform vec4 color;  
  
void main()  
{  
    gl_FragColor = color;  
}
```

# Passagem de valores constantes (uniform)

Na aplicação:

```
var color = [1.0, 0.0, 0.0, 1.0];  
colorLoc = gl.getUniformLocation(program, "color");  
gl.uniform4fv(colorLoc, color);
```

Alternativa usando um array javascript diretamente

Pertence a uma família de métodos: uniform\*

No fragment shader (de forma idêntica no vertex shader):

```
uniform vec4 color;  
  
void main()  
{  
    gl_FragColor = color;  
}
```



# GLSL: operadores e funções

- Funções C standard:
  - trigonométricas
  - aritméticas
- vetoriais: normalize, reflect, length, dot
- overloading de operadores para vetores e matrizes

```
mat4 a;
```

```
vec4 b, c, d, e;
```

```
c = a*b; // vetor coluna
```

```
d = b*a; // vetor linha
```

```
e = c*d; // component-wise multiplication
```

# GLSL: swizzling e seleção

- Podemos nos referir aos elementos dos tipos vetor e matriz usando [] ou o operador de seleção (.) com:
  - x,y,z,w
  - r,g,b,a
  - s,t,p,q
- `a[2]`, `a.b`, `a.z` e `a.p` são exatamente a mesma componente
- Swizzling permite manipular as componentes:

```
vec4 a, b;
```

```
a.yz = vec2(1.0, 2.0);
```

```
b = a.yxzw;
```