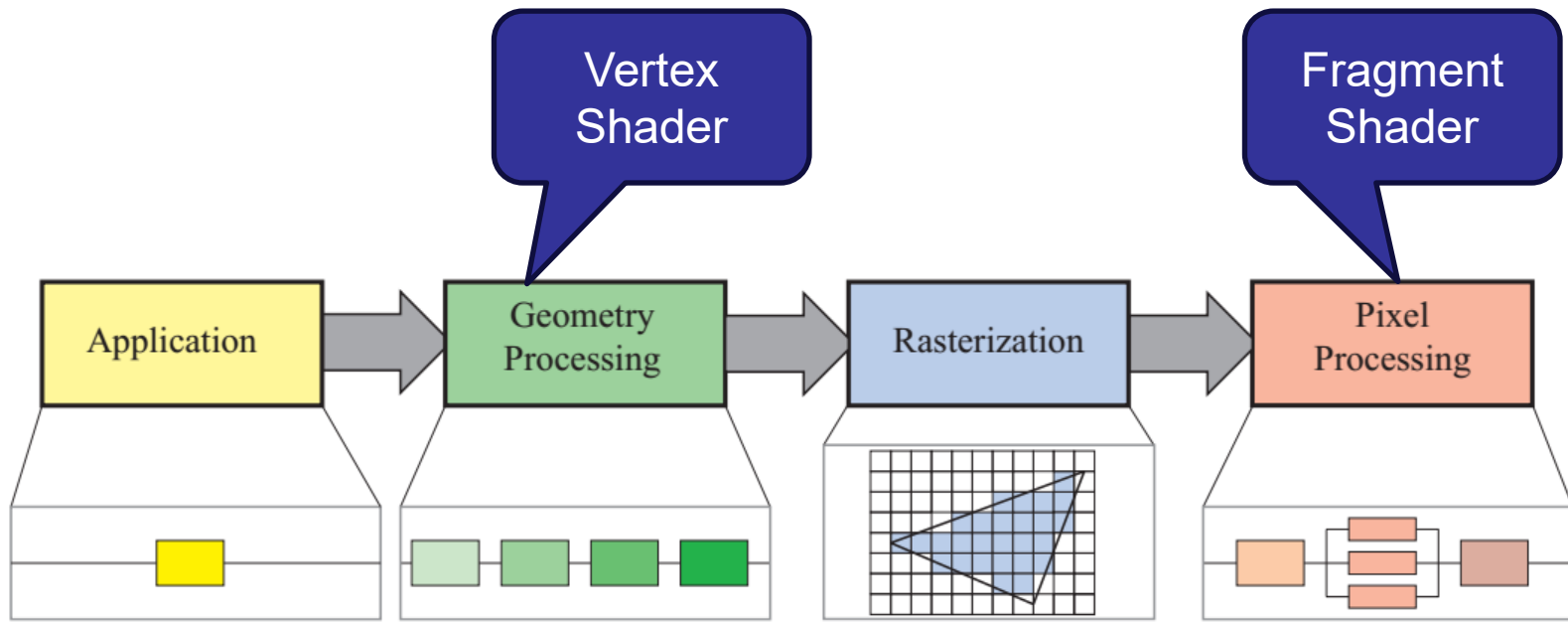


Introdução aos *shaders* (OpenGL moderna)

por Rossana B Queiroz

Pipeline Gráfico



[Akenine-Möller *et al.* 2018]

Vertex Shader

- Primeiro estágio do pipeline programável. Ele **processa cada vértice individualmente**.
- Processamento dos Vértices
 - Processamento e/ou redirecionamento de seus **atributos** (posição, normal, cor, coordenadas de texturas)
 - Alguns destes atributos são enviados para outras etapas do pipeline
 - Mapeamento de coordenadas: aplica **transformações na geometria** a partir das matrizes de view, projection e model.
 - `gl_Position` recebe o resultado dessas transformações

Fragment Shader

- É executado para cada **fragmento (potencial pixel)** que será desenhado na tela.
- Determina a **cor final de cada fragmento**, aplicando operações como textura, iluminação e cálculos de sombreamento.
- Realiza operações de **blending** (mesclagem de cores) e cálculos de transparência.
- Controla a precisão e os efeitos de cada pixel, permitindo a criação de efeitos visuais complexos (pós-processamento).



Como enviar dados para os shaders?

A partir de variáveis do tipo *uniform*

A partir de buffers com as informações dos vértices e seus atributos

- Vertex Buffer Object
- Vertex Array Object



Uniforms

- Envia dados constantes, que não variam entre os diferentes vértices ou fragmentos, como matrizes de transformação, luzes, ou cores.
- Uniforms são variáveis globais nos shaders que podem ser definidas a partir do código da CPU. Elas permanecem constantes durante a execução de um draw call.
 - Desde que declaradas no código do shader, podem ser acessadas pelo vertex e pelo fragmente shader

Enviando dados para a GPU

- Exemplo (enviando uma cor)
 - Selecionar o programa de shader ao qual quer enviar

```
glUseProgram(shaderID);
```

- Geração do identificador e envio

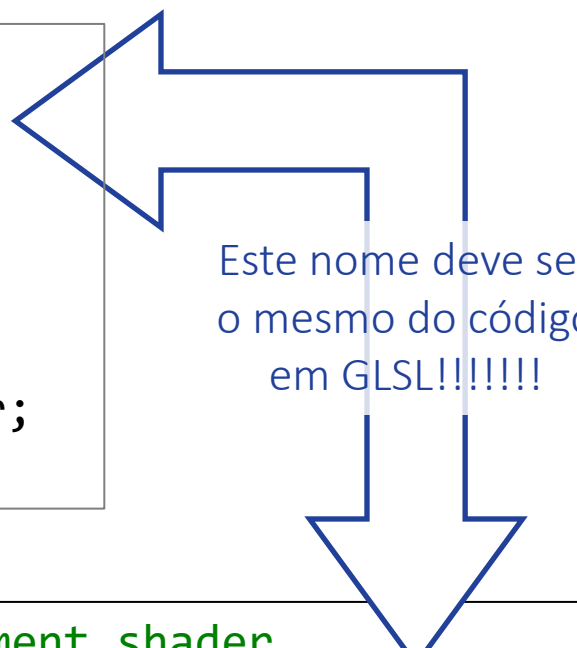
```
//Enviando a cor desejada (vec4) para o fragment shader  
GLint colorLoc = glGetUniformLocation(shaderID, "inputColor");  
assert(colorLoc > -1);  
glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f);
```

Enviando dados para a GPU

Código do shader (GLSL)

```
#version 410
uniform vec4 inputColor;
out vec4 color;

void main()
{
    color = inputColor;
}
```



Este nome deve ser
o mesmo do código
em GLSL!!!!!!

Código em C++

```
//Enviando a cor desejada (vec4) para o fragment shader
GLint colorLoc = glGetUniformLocation(shaderProgram, "inputColor");
assert(colorLoc > -1);
glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f);
```


Enviando dados para a GPU

- O comando para envio varia de acordo com o tipo de dado

```
glUniform4f(colorLoc, 1.0f, 0.0f, 0.0f, 1.0f);
```

Nome do identificador

Tipo do dado a ser enviado

Número de dados a serem enviados

<https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glUniform.xhtml>

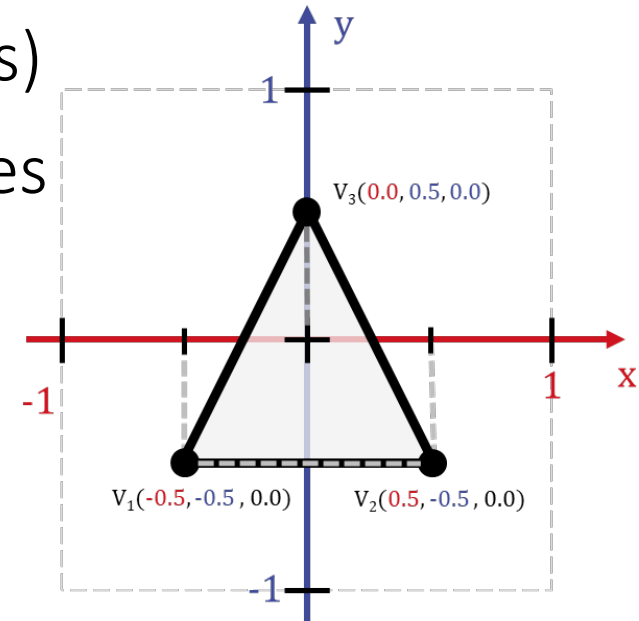
Buffers na OpenGL Moderna

- Definição de *buffer*: região de memória que serve para armazenar dados temporariamente
 - Dados provindos de algum dispositivo de entrada, antes de serem processados
 - Dados a serem enviados para algum dispositivo de saída (por exemplo, *frame buffer*) ou para serem processados pela GPU



Vertex Buffer Objects (VBOs)

- Array de dados (normalmente floats)
- Buffer para enviar dados dos vértices à GPU
 - posição, vetores normais, cores etc
- Os dados são alocados diretamente na memória da GPU
- Isso permite que os objetos sejam renderizados pela placa gráfica com maior velocidade



Esta forma poderia ter o vertex buffer:

```
{ -0.5, -0.5, 0.0,  
  0.5, -0.5, 0.0,  
  0.0,  0.5, 0.0 }
```

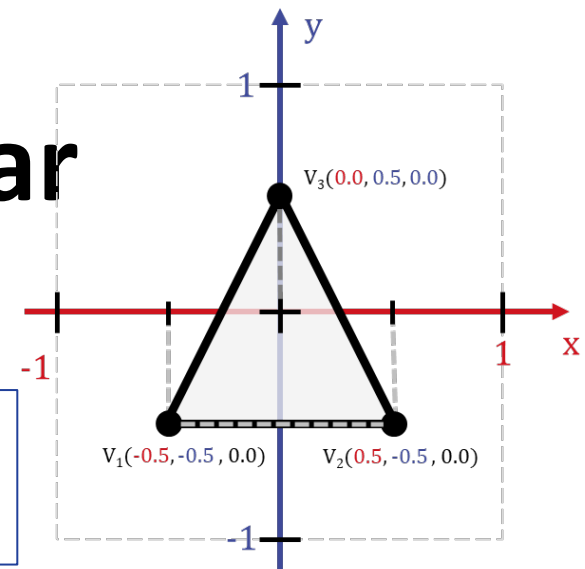


UNISINOS
Nossa sala de aula
é o mundo.

Como programar

1. Definir o array:

```
GLfloat vertices[] = { -0.5f, -0.5f, 0.0f,  
                        0.5f, -0.5f, 0.0f,  
                        0.0f, 0.5f, 0.0f };
```



2. Cada buffer na OpenGL precisa ter um identificador único, para isso precisamos gerar este ID usando a função glGenBuffers:

```
GLuint vbo;  
glGenBuffers(1, &vbo);
```

3. OpenGL possui vários tipos de buffers. Os VBOs são do tipo `GL_ARRAY_BUFFER`. Devemos agora fazer o *bind* do novo buffer criado:

```
glBindBuffer(GL_ARRAY_BUFFER, vbo);
```



Como programar (continuação)

4. Por ultimo, chamamos a função *glBufferData* que copia os dados do array definido para a memória do buffer da OpenGL:

```
glBufferData(GL_ARRAY_BUFFER, 9 * sizeof(GLfloat), vertices, GL_STATIC_DRAW);
```

ou `sizeof(vertices)`

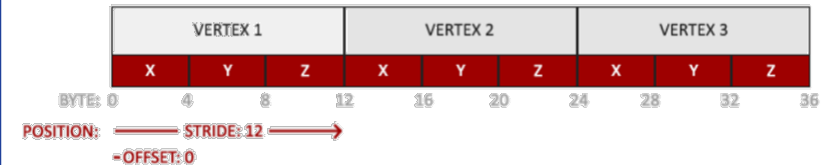
O quarto parâmetro especifica como nós queremos que a placa gráfica gerencie os dados fornecidos. Há 3 formas:

- `GL_STATIC_DRAW` – dados não vão mudar, ou muito raramente
- `GL_DYNAMIC_DRAW` – dados vão mudar com frequência
- `GL_STREAM_DRAW` – dados vão mudar cada vez que forem desenhados

Vertex Array Objects (VAOs)

- Fazem a ligação dos atributos de um vértice
 - Posição, cores, normais
- Define:
 - Qual o VBO que será usado
 - Localização dos dados desse VBO
 - Qual o formato desses dados

Em nosso exemplo anterior, os dados do buffer estão formatados da seguinte forma:



- Os valores de posição estão armazenados como floats (4 bytes)
- Cada posição é composta por 3 desses valores (x, y e z) – logo temos um total de 12 bytes por vértice
- Não possuem outros valores armazenados entre cada conjunto de 3 valores que definem o vértice – logo nosso offset é zero.
- O primeiro valor está armazenado diretamente no início do buffer (byte 0).

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);  
glEnableVertexAttribArray(0);
```



Vertex Array Objects (VAOs)

- Segue a explicação de cada parâmetro:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
```

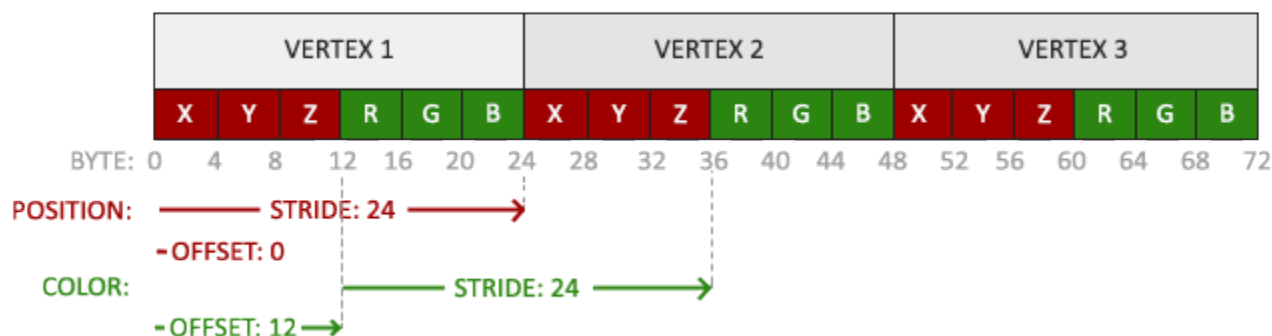
- O primeiro parâmetro, 0, refere-se a qual o atributo estamos linkando (posição, cor, normal, coord textura...). Este número será o mesmo no vertex shader, identificado com a palavra-chave **location**

```
layout (location = 0) in vec3 position;
```

- O próximo parâmetro, 3, especifica o tamanho do atributo (3 valores – xyz)
- O próximo parâmetro, GL_FLOAT, especifica o tamanho de cada dado
- O próximo parâmetro, GL_FALSE, especifica se os dados precisam ser normalizados (valores no interval de -1 a 1). Nossos dados já estão.
- Por fim, passamos o deslocamento inicial no buffer, que no nosso caso é nenhum (podemos colocar 0 ou NULL se desejarmos)

Vertex Array Objects (VAOs)

- Mais de um atributo:

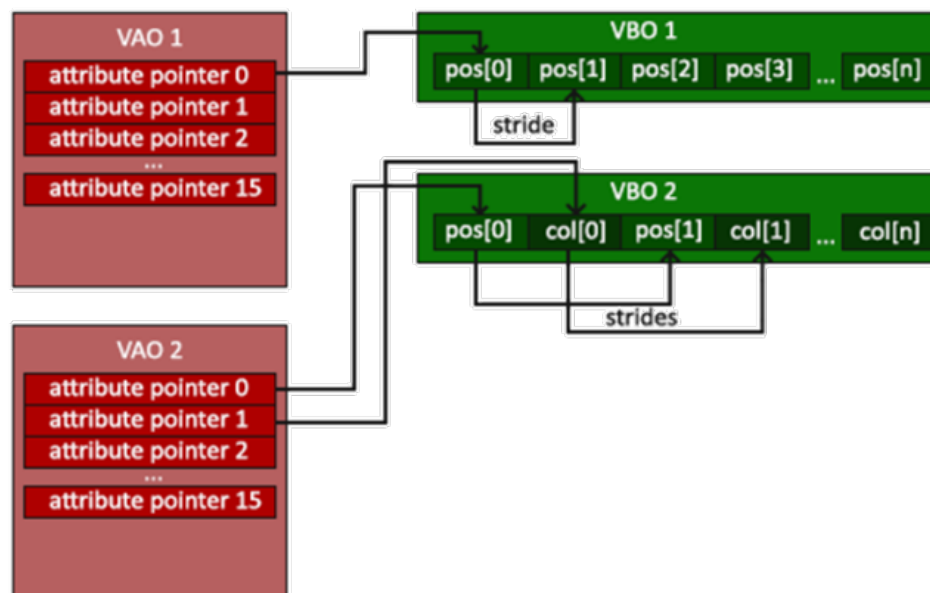


```

000řôşîţîîôŋăăţţşîćăţê
ğĭĭêşţêŷĀţţşîćRôîŋţêş , 00 , 00ĠĬĠGLĬÔĀŢ00ĠĬĠGAL'ŞÉ00 ` 000şîćêôğğĭĭôăţ0000wôîđ000 , 00
ğĭĖŋăćĭêĭêşţêŷĀţţşîćAssăŷ , 00
000çôĭôş0ăăţţşîćăţê
ğĭĭêşţêŷĀţţşîćRôîŋţêş , 00 , 00ĠĬĠGLĬÔĀŢ00ĠĬĠGAL'ŞÉ00 ` 000şîćêôğğĭĭôăţ0000wôîđ0000 , 000
şîćêôğğĭĭôăţ0000
ğĭĖŋăćĭêĭêşţêŷĀţţşîćAssăŷ , 00
    
```

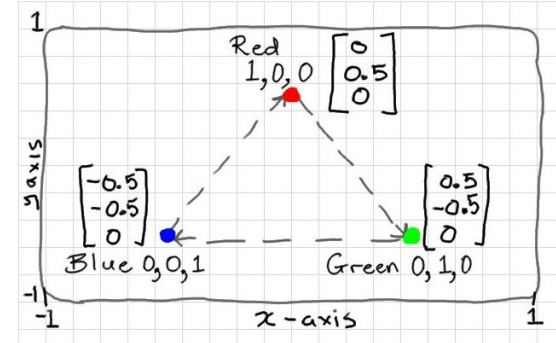

Vertex Array Objects (VAOs)

- Ideia geral:



- Podemos ter mais de um VAO, e cada VAO pode guardar mais de um atributo dos vertices (VBOs)
 - Troca de contexto entre VAOs possui um certo custo, portanto, cuidar!

Exemplo com 1 VBO – 1 array e 2 atributos



1

```
// Set up vertex data (and buffer(s)) and attribute pointers
GLfloat vertices[] = {
// Positions           // Colors
0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,  // Bottom Right
-0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,  // Bottom Left
0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f    // Top
};
```

3

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
vertices, GL_STATIC_DRAW);
```

4

```
// Position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6
* sizeof(GLfloat), (GLvoid*)0);
glEnableVertexAttribArray(0);
```

5

```
// Color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 *
sizeof(GLfloat), (GLvoid*)(3 * sizeof(GLfloat)));
glEnableVertexAttribArray(1);
```

2

```
GLuint VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);

glBindVertexArray(VAO);
```

Para mandar desenhar

- No loop do código:

```
glUseProgram(shaderProgram);  
glBindVertexArray(VAO);  
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Offset inicial

Nro de vértices

