Universidade Lusófona de Humanidades e Tecnologias

ULHT260-2267
Licenciatura em Engenharia Informática
Computação Gráfica
Nuno Cruz Garcia – p6040@ulusofona.pt

Aula Prática 03

Temas
OpenGL – Shader Class
OpenGL Shading Language (GLSL)

Objectivo
No final desta ficha devemos ter uma classe *shader* pronta a usar. Isto será útil para o projecto e para os próximos labs.

Como vimos na última aula prática (aulaPratica02.pdf), é necessário definir um *Vertex Shader* e um *Geometry Shader* para conseguirmos renderizar uma imagem usando o OpenGL. A figura seguinte relembra o pipeline a que nos referimos. É importante ter em consideração a parte do pipeline em que cada shader se encontra para perceber qual é o tipo de input e output que nos interessa codificar.

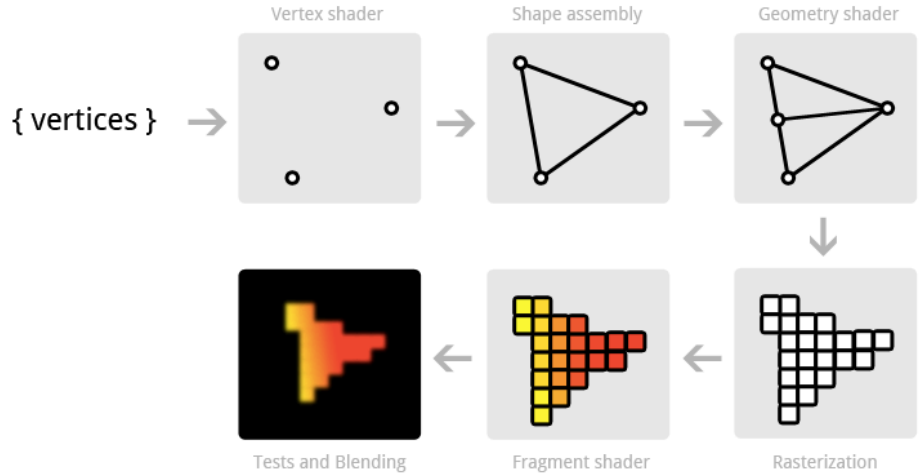


Figura 1 - from: https://learnopengl.com/Getting-started/Hello-Triangle

O objectivo desta ficha prática é explorar a linguagem em que os *shaders* são escritos, para podermos renderizar objectos mais interessantes.

O OpenGL Shading Language é semelhante à linguagem C.
A estrutura de um *shader* é a seguinte:

```
#version version_number          // versão
in type in_variable_name;        // inputs
in type in_variable_name;
```

```
out type out_variable_name;      // outputs

uniform type uniform_name;       // uniforms

void main()           // ponto de entrada do programa
{
  // process inputs
  ...
  // output processed stuff
  out_variable_name = weird_stuff_we_processed;
}
```

Por exemplo, os inputs do vertex shader são os atributos dos vértices. No exemplo da última aula helloTriangle.cpp, o único atributo que havia para processar era a posição dos vértices:

O GLSL implementa vários tipos de dados, para além dos habituais *int, float,* etc, também oferece o tipo de dados *vecn.*
Por exemplo, *vec3* é um vector de 3 floats. Na última aula declarámos o input usando este tipo de dados: `layout (location = 0) in vec3 aPos;`

Alguns exemplos de como usar o tipo de dados *vecn:*

```
vec2 someVec = vec2(0.5, 0.7);
vec4 differentVec = someVec.xyxx;  // cria o vector (0.5, 0.7,
                                   0.5, 0.5)
vec4 otherVec = someVec.xxxx + differentVec.yxxy;
```

O tamanho máximo é *vec4*.

Exercício 1:
Usando o programa da última aula, programe um shader que inverta o triângulo, isto é que o rode 180º.

Inputs e Outputs

O que deve necessariamente ser feito:

- O vertex shader deve receber como input vectores de vértices. É por isso necessário definir onde e como ler as propriedades dos vértices.
  - Na última aula: `layout (location = 0) in vec3`
- O fragment shader deve necessariamente ter como output um *vec4* que representa um vector de cores.
  - Na última aula: `FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);`

O OpenGL liga automaticamente uma variável de output a uma variável de input do *shader* seguinte se ambas tiverem o mesmo nome e tipo de dados. Isto é feito no bloco de código comentado com *//* LINKING SHADERS *** do triangle.cpp.

Exercício 2:
O programa da última aula determinava a côr do triângulo no *fragment shader*.
Neste exercício, declare uma variável de output no *vertex shader* e use-a para determinar a côr do polígono, em vez de isto ser feito no *fragment shader*.

A única instrução do main() do *fragment shader* deverá ser: FragColor = vertexColor;

Uniforms

Vamos agora passar uma cor directamente da aplicação para o *fragment shader*. Os *uniforms* são variáveis globais, que podem ser acedidas por todos os *shaders*.
Por exemplo, _dentro_ do *fragment shader*, podemos declarar um

```
uniform vec4 ourColor;
```

Esta variável pode ser actualizada desde o código c++ da aplicação, isto é desde o loop de renderização, através do seu endereço de memória:

```
int vertexColorLocation = glGetUniformLocation(shaderProgram,
"ourColor")
glUseProgram(shaderProgram); //é necessário activar o programa.
glUniform4f(vertexColorLocation, red, green, blue, alpha);
```

Exercício 3:
Varie a cor de forma periódica com o tempo. Isto é, o valor da cor vermelha (por exemplo) deve variar no intervalo [0,1] de forma uniforme e periódica em função do tempo.
O tempo em segundos pode ser obtido através de:

```
float timeValue = glfwGetTime();
```

E um valor entre [0,1] pode ser obtido utilizando o seno.

```
float aValue = (sin(timeValue) / 2.0f) + 0.5f;
```

Conclusão: Os *uniforms* são úteis para passar informação desde a aplicação para os shaders.

Tantas cores quanto vértices

No exercício anterior o polígono tem apenas uma cor. Se quisermos atribuir uma cor a cada vértice, o mais fácil será adicionar uma propriedade por vértice, em vez de declarar *n uniforms* para *n* vértices.

O VBO que criámos na aula passada continha apenas coordenadas passará agora a guardar uma cor rgb:

```
float 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
};
```

O *vertex shader* deve ser actualizado considerando esta nova propriedade.

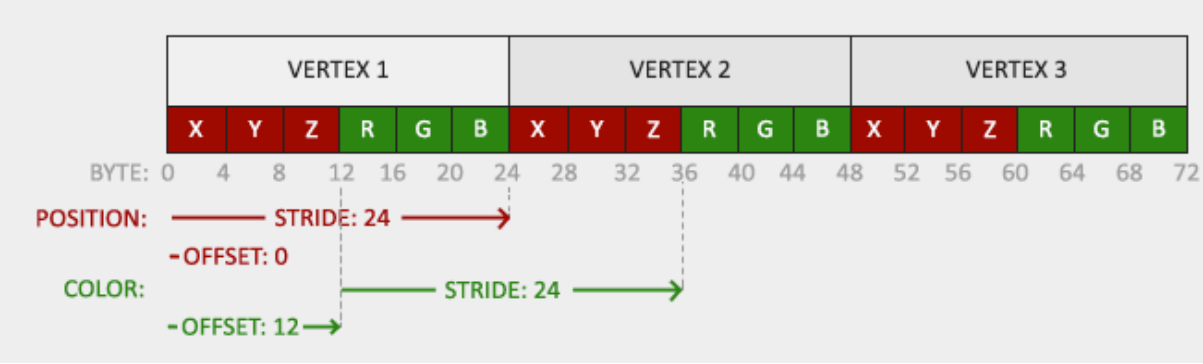
```
layout (location = 0) in vec3 aPos;   // the position variable has
                                      attribute position 0
layout (location = 1) in vec3 aColor; // the color variable has
                                      attribute position 1
```

E a função main() deve também reflectir que *aColor* é a cor que nos interessa.

```
ourColor = aColor; //ourColor está definida como variável de
                   output.
```

Exercício 4:

Use como código base o código desenvolvido para o exercício 2.
O vector de dados guardado no VBO tem o seguinte aspecto:



Adicione duas linhas de código (glVertexAttribPointer e glEnableVertexAttribArray) de forma a reflectir estas mudanças no VBO. Actualize também a stride da chamada glVertexAttribPointer para o atributo posição.
Verifique a palete de cores resultado da interpolação das três cores especificadas.

Exercício 5:
Usar *shaders* envolve muitas operações desde a sua criação, compilação, linkagem, gestão de *uniforms* etc. A criação de uma classe é uma boa forma de encapsular estes passos, tornar o código mais legível e facilitar o seu uso.
Use o ficheiro Shader.hpp e implemente o exercício 4 usando a classe.
Crie dois ficheiros, shader.vs e shader.fs, que devem conter as strings dos programas shaders vertex e fragment. A sua definição desaparece do ficheiro da aplicação. Esta apenas passa a conter:

```
Shader myShader("path/shader.vs", path/shader.fs");
While …
      myShader.use();
      myShader.setFloat("someUniform", 1.0f);
      Draw()
```

References:
https://learnopengl.com/
https://open.gl/