OpenGL

Ricardo Dutra da Silva
Elisa de Cássia Silva Rodrigues

Universidade Tecnológica Federal do Paraná
Universidade Federal de Itajubá

2020

Programas OpenGL

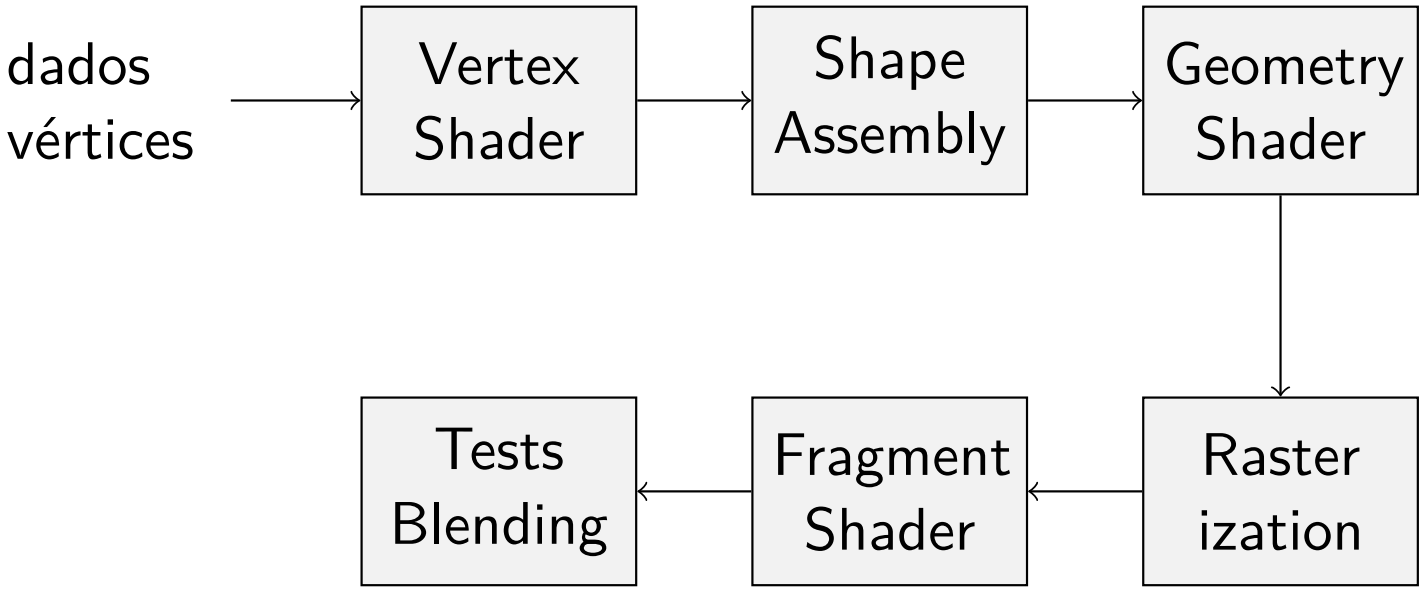
- Vamos definir pontos importantes e básicos para qualquer programa OpenGL:
  - o pipeline gráfico;
  - os shaders;
  - a linguagem GLSL (OpenGL Shading Language)
  - array objects;
  - buffer objects.

Graphics Pipeline

- Define as etapas para transformar objetos em imagens.
- A saída de cada etapa é entrada para a próxima.
- As etapas são pequenos programas: *shaders*.
- Os shaders são executados em paralelo nos *cores* de uma placa gráfica.
- Shaders são escritos na linguagem GLSL.

Graphics Pipeline

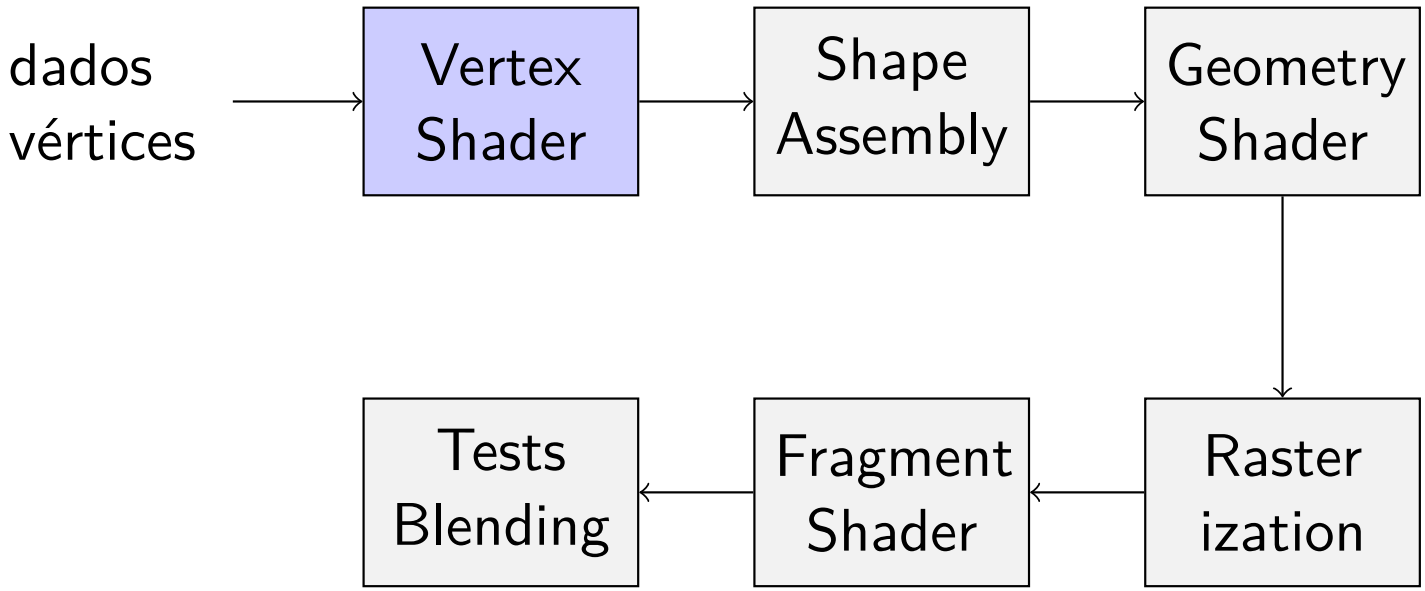
- Etapas.



- A entrada são atributos de vértices que definem objetos 3D.
- Atributos como coordenadas espacias, cores, etc.

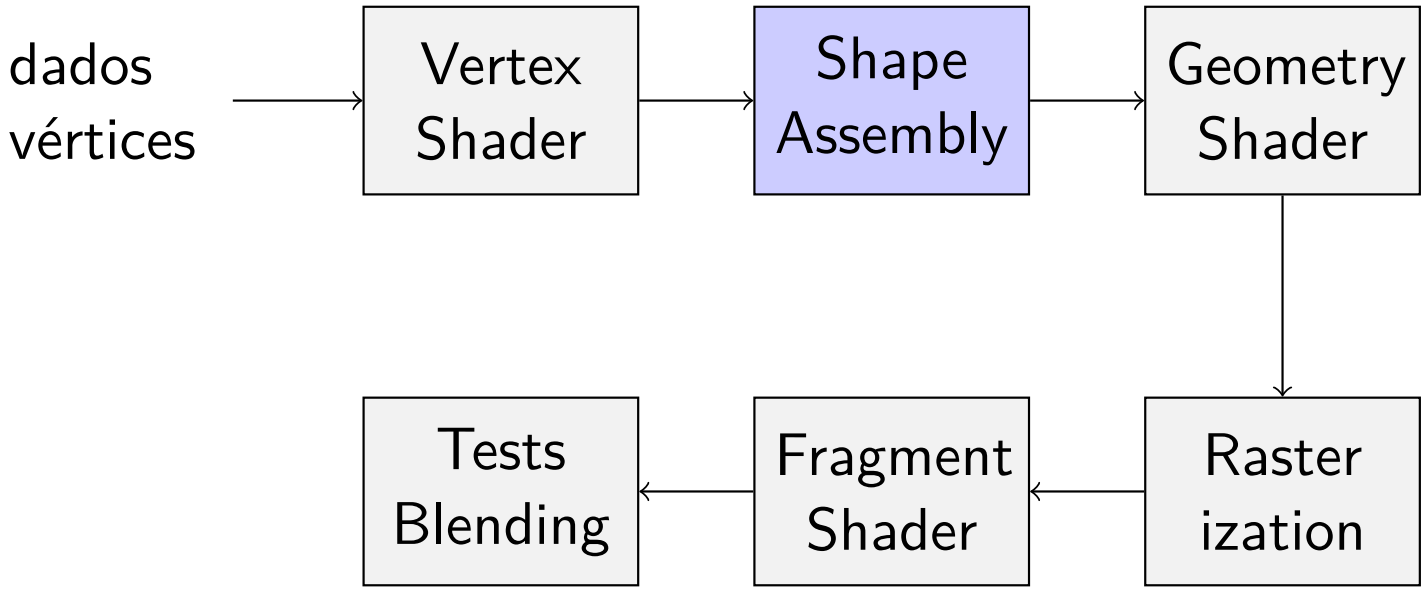
Graphics Pipeline

- Vertex shader: processa a posição dos vértices.
- Aplica transformações geométricas, projeções, etc.



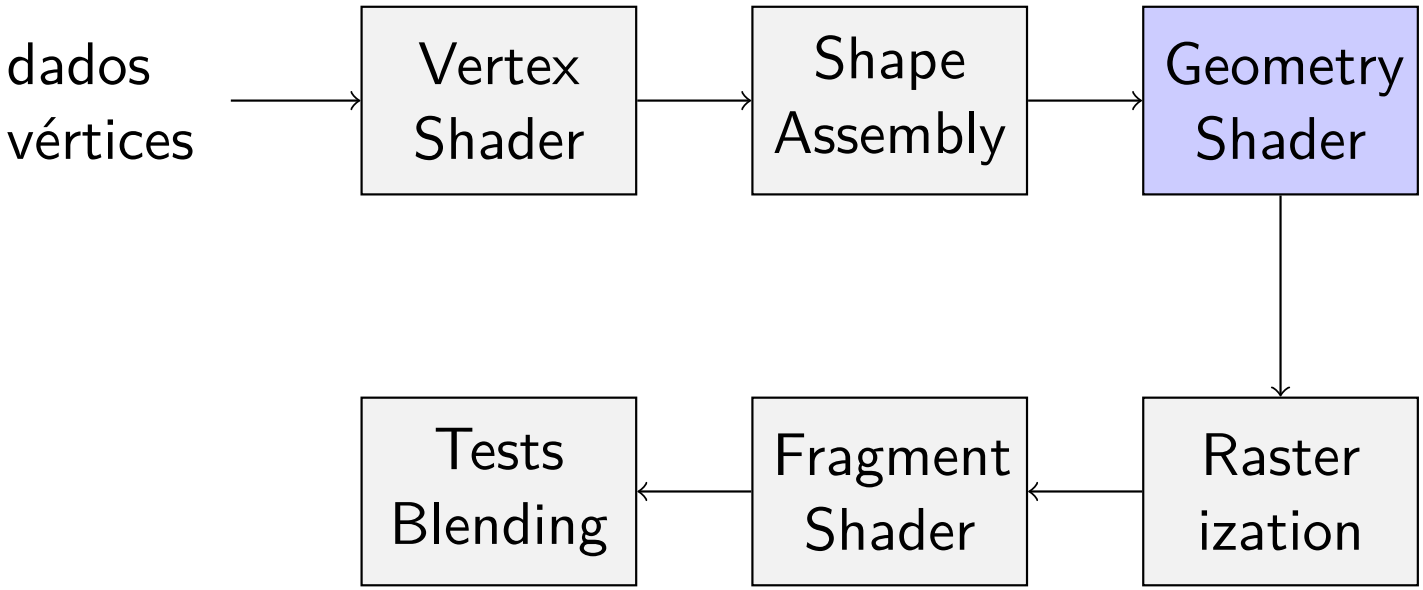
Graphics Pipeline

- Shape assembly: combina os vértices para criar formas como linhas e triângulos.



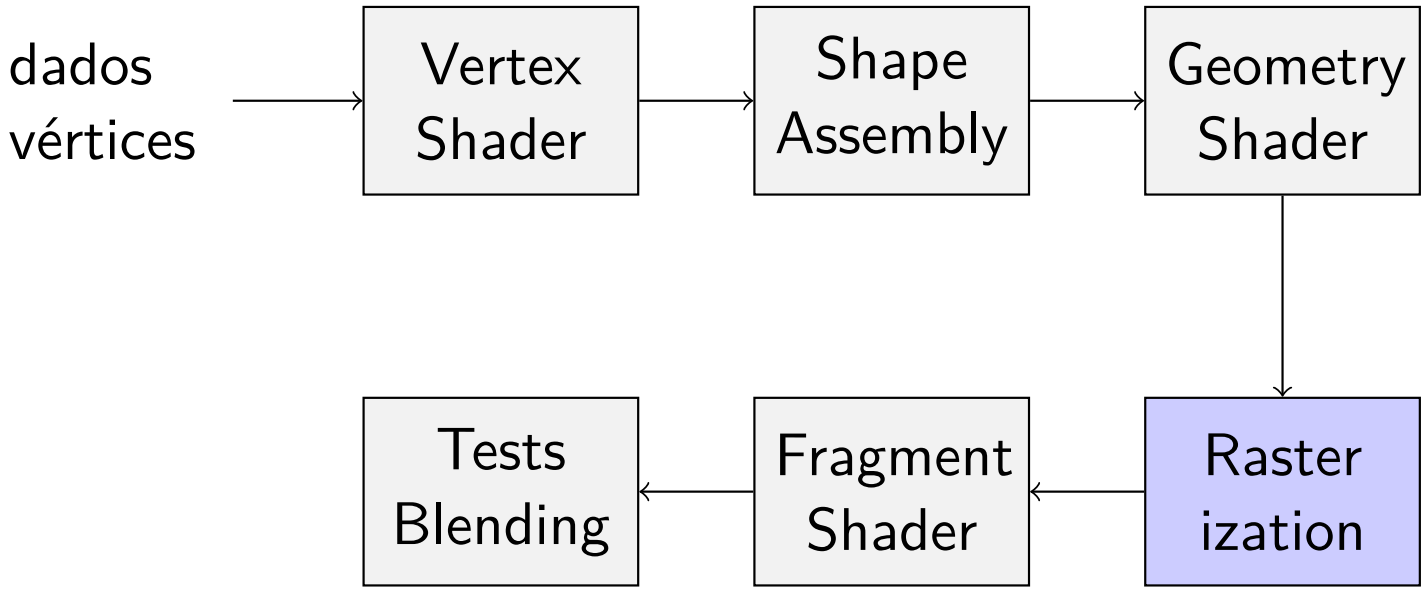
Graphics Pipeline

- Geometry shader: cria outras geometrias necessárias (como subdivisões de polígonos).



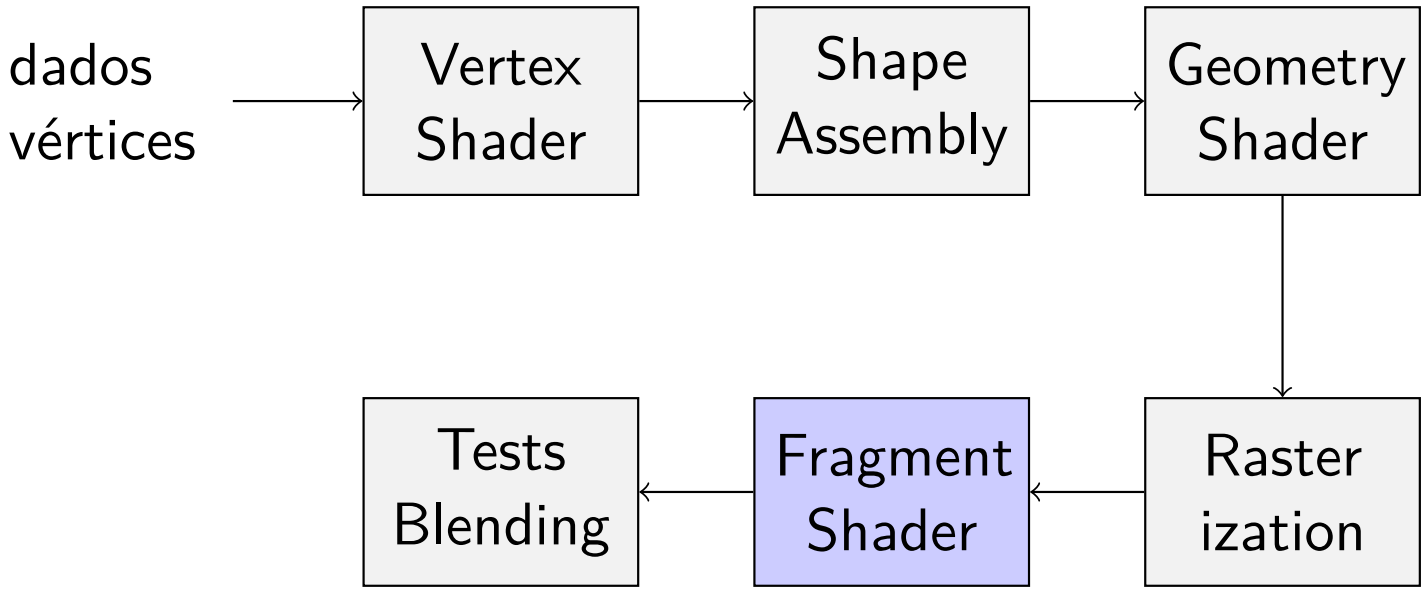
Graphics Pipeline

- Rasterization: mapeia as geometrias para pixels.



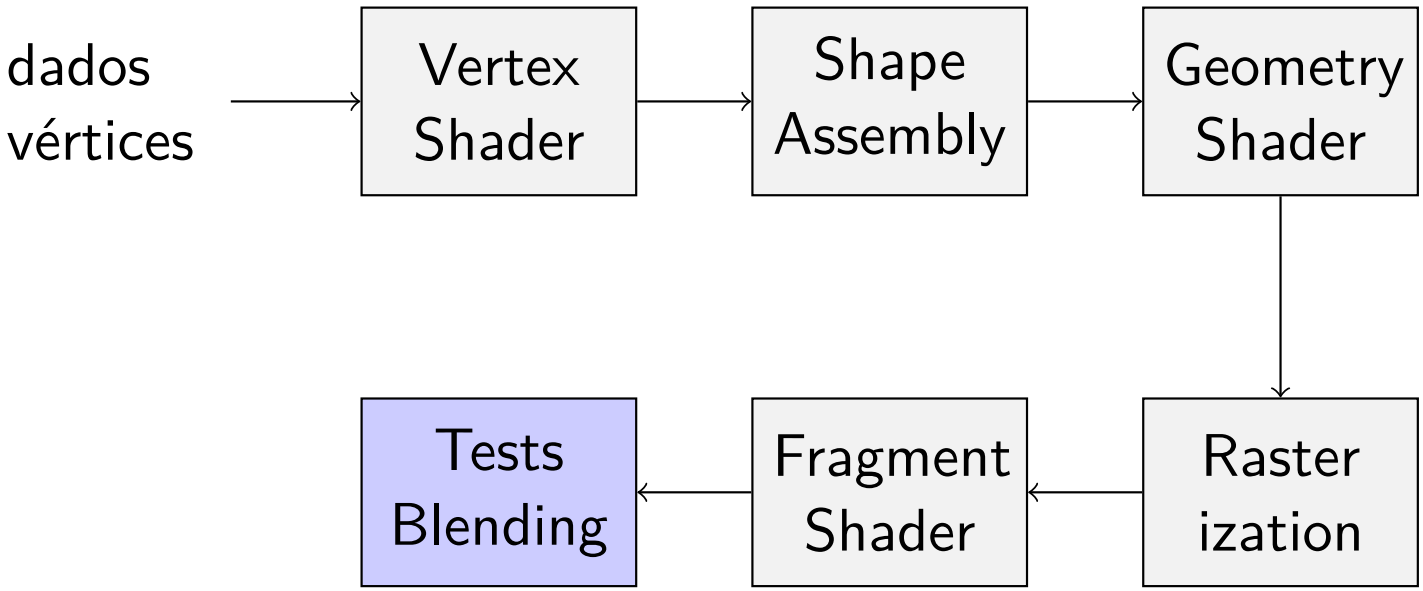
Graphics Pipeline

- Fragment shader: mapeia para os pixels as cores, iluminação, sombras, etc.



Graphics Pipeline

- Tests e Blending: verifica a ordem dos objetos (aqueles que aparecem na frente de outros ou são obstruídos) e combina cores entre esses objetos.



Graphics pipeline

- Vamos focar nos shaders para duas etapas: vertex shader e fragment shader.
- Sempre precisamos definir pelo menos esse dois.
- Os outros são para funções mais avançadas que não serão cobertas aqui.

Etapas para Desenhar

- Na sequência serão vistas as etapas de um programa para:
  - criar os dados de entrada (vértices);
  - criar buffer objects para enviar os dados para a GPU;
  - criar array objects;
  - definir e compilar os shaders;
  - desenhar uma primitiva (ponto, linha ou triângulo).
- Cada uma dessas etapas será analisada com base no programa `triangle.cpp`.
- Os parâmetros das funções podem variar ligeiramente em python (ver `triangle.py`).

Definição de um Objeto

- Objetos são definidos por vértices.
- Vértices são comumente coordenadas (x, y, z).
- Internamente o OpenGL trabalha com coordenadas homogêneas (x, y, z, w).
- Geralmente usamos $(x, y, z, 1)$, como será visto adiante.

Normal Device Coordinate (NDC)

As coordenadas x, y, z, após o Vertex Shader, devem estar entre -1 e 1 para serem desenhadas. Podemos definir um objeto fora destes intervalo e, posteriormente, uma projeção ajusta os valores visíveis. Isto será visto mais à frente no curso.

Definição de um Objeto

- Analisando o código temos, a função `initData` define um triângulo por três vértices com coordenadas 3D, (x,y,z).
- É usado um array `float` que armazena as coordenadas já em NDC: $(-0.5, -0.5, 0.0)$, $(0.5, -0.5, 0.0)$, $(0.0, 0.5, 0.0)$.

Exemplo

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

Array Buffer Object

- Os atributos dos objetos precisam ser copiados para um buffer object do OpenGL.
- Um buffer em memória da GPU.
- Atributos são comumente coordenadas, cores, etc.
- Precisamos de três funções para criar e popular o buffer:
  - `glGenBuffers`
  - `glBindBuffer`
  - `glBufferData`
- Precisamos de duas função para dizer como os atributos devem ser interpretados e para habilitá-los:
  - `glVertexAttribPointer`.
  - `glEnableVertexAttribArray`.

Array Buffer Object

`glGenBuffers`

Cria identificadores únicos para buffers. Parâmetros:

- *n* número de buffers requeridos.
- *buffers* array para guardar os identificadores.

`glBindBuffer`

Torna o Buffer ativo, funções na sequência vão modificar este buffer específico. Parâmetros:

- tipo `GL_ARRAY_BUFFER` para um buffer de vértices;
- identificador do buffer.

Array Buffer Object

glBufferData

Copia dados para o buffer. Parâmetros:

- tipo `GL_ARRAY_BUFFER` para um buffer de vértices;
- número de bytes que serão copiados;
- array com os dados;
- como a placa trata os dados: `GL_STATIC_DRAW` para dados estáticos.

- Pesquise e explique outros modos: `GL_DYNAMIC_DRAW`, `GL_STREAM_DRAW`.

Array Buffer Object

`glVertexAttribPointer`

Define como os dados devem ser interpretados pelo Vertex Shader.
Parâmetros:

- localização do atributo no shader, o valor passado é definido como location no shader (como veremos logo);
- número de valores no atributo;
- tipo do atributo;
- define se o dado deve ser normalizado, em geral usamos falso;
- stride: número de posições entre atributos, útil quando array de dados possui tipos de dados intercalados, por exemplo, coordenadas e cores;
- deslocamento para início dos dados. 0 se começa na primeira posição do array.

Array Buffer Object

`glEnableVertexAttribArray`

Habilita o atributo. Parâmetro:

- localização do atributo como definido no primeiro parâmetro da função anterior.

Array Buffer Object

- No exemplo criamos um buffer.
- Informamos que ele está sendo usado, é o buffer que deve ser modificado.
- Copiamos os vértices do triângulo.

Exemplo

```
glGenBuffers(1, &VBO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);

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

Array Buffer Object

- Definimos como o atributo coordenadas deve ser interpretado:
  - sua localização é a 0 para uso no shader;
  - atributo tem três parâmetros (x, y e z);
  - do tipo float;
  - sem normalização;
  - coordenadas estão separadas por 3 valores float;
  - dados começam já no início do buffer.
- Habilitamos o uso do atributo.

Exemplo

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3*sizeof(float),
(void*)0);

glEnableVertexAttribArray(0);
```

Vertex Array Object

- O Vertex Array Object (VAO) guarda as definições de atributos e buffers a partir do momento em que é habilitado.
- Isto facilita na hora de desenhar, basta habilitar e desabilitar o VAO que queremos.

`glGenVertexArrays`

Cria identificadores únicos para VAOs. Parâmetros:

- *n* número de objetos requeridos.
- *arrays* array para guardar os identificadores.

`glBindVertexArray`

Torna o VAO ativo e guarda buffers e atributos definidos na sequência. Parâmetro:

- identificador do array object.

Vertex Array Object

- Chamamos no exemplo ao configurar o buffer e ao desenhar (veremos na sequência).

Exemplo

```
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

// Vertex buffer
// Set attributes.

// Desabilita VAO.
glEnableVertexAttribArray(0);
```

Shaders

- Vertex shader: define transformações sobre os vértices.
- Fragment shader: define propriedades dos pixels: cores, texturas, etc.
- Escritos em GLSL (OpenGL Shading Language), linguagem parecida com C.
- Começa com a versão do OpenGL.
- Em seguida definimos variáveis de entrada e saída.
- No main definimos o que deve ser feito.

Vertex Shader

- A linha 1 define OpenGL 3.3.
- A linha 2 define vetor de entrada com 3 valores que irá receber as coordenadas dos vértices.
- "layout (location = 0)" é a localização para o atributo, como definida na função `glVertexAttribPointer`.

Exemplo

```
1 #version 330 core
2 layout (location = 0) in vec3 position;
3
4 void main()
5 {
6     gl_Position = vec4(position.x, position.y, position.z, 1.0);
7 }
```

Vertex Shader

- Na linha 6 transformamos as coordenadas (x,y,z) em coordenadas homogêneas (x,y,z,w).
- Especificamente (x,y,z) vira (x,y,z,1).
- As coordenadas são atribuídas à variável `gl_Position` que é usada no pipeline do OpenGL.

Exemplo

```
1 #version 330 core
2 layout (location = 0) in vec3 position;
3
4 void main()
5 {
6     gl_Position = vec4(position.x, position.y, position.z, 1.0);
7 }
```

Fragment Shader

- A linha 2 define uma variável de saída que irá armazenar a cor dos pixels.
- A linha 6 define a cor com um vetor de 4 posições (R,G,B,A).

Exemplo

```
1 #version 330 core
2 out vec4 FragColor;
3
4 void main()
5 {
6    FragColor = vec4(1.0f, 0.0f, 0.0f, 1.0f);
7 }
```

Shaders

- Estes são shaders simples.
- O vertex shader está copiando as coordenadas do vértices para a saída.
- O fragment shader está definindo a cor como vermelha.

Compilando os Shaders

- O próximo passo é compilar os shaders para criar o programa que será executado no pipeline.
- Nos próximos exemplos usaremos `createShaderProgram`, que combina todas as funções.
- Neste exemplo, a função `initShaders` contém os passos necessários.
- Para maiores informações ver o Apêndice.
- A função `glUseProgram` deve ser chamada para usarmos os shaders criados quando formos desenhar.

`glUseProgram`

Usamos esta função para ativar o programa compilado.

Display

- Finalmente, podemos desenhar o triângulo.

glDrawArrays

Desenha o array de dados. Parâmetros:

- tipo da primitiva;
- posição de início no array;
- quantos vértices devem ser desenhados.

Display

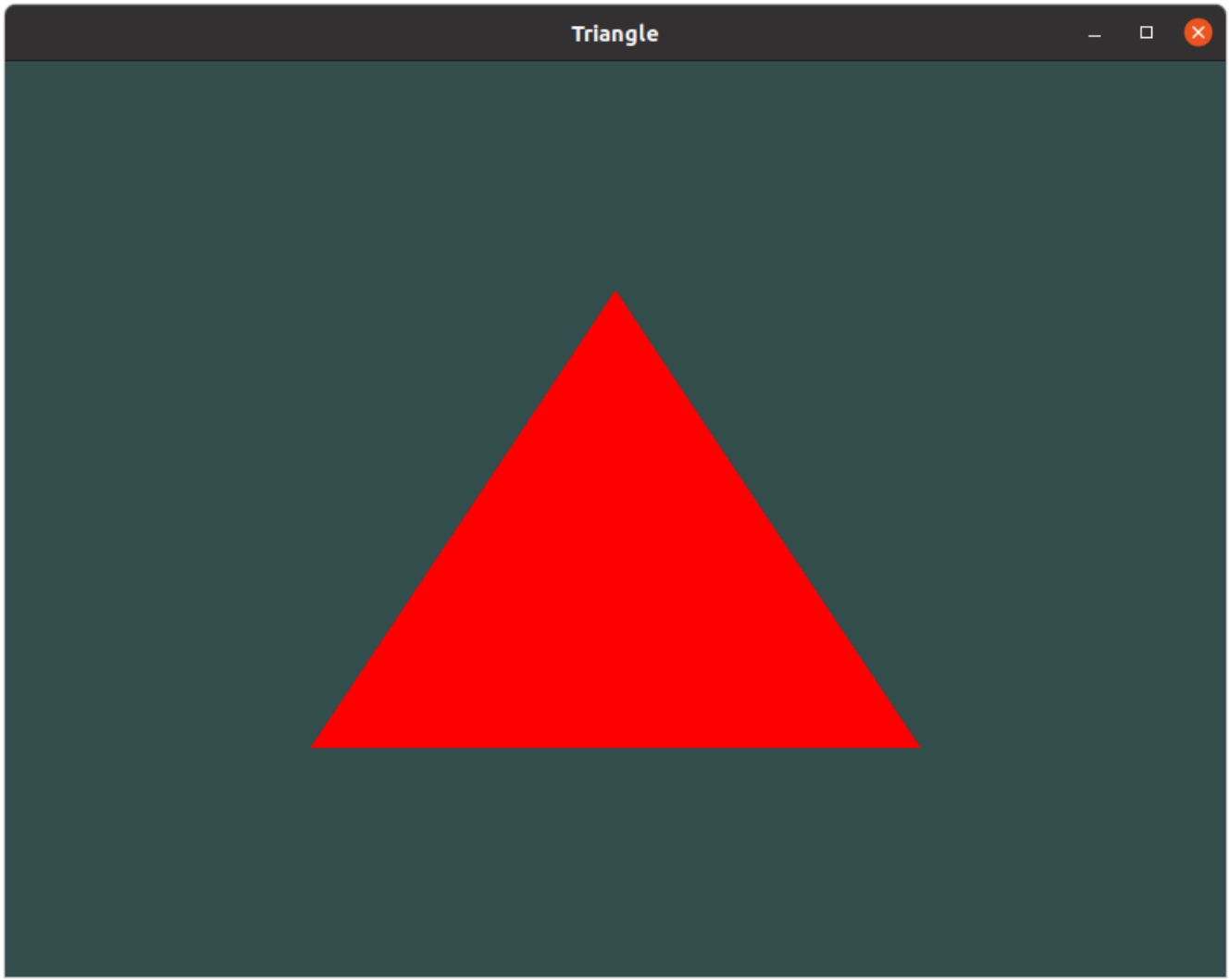
- No nosso caso desenhamos `GL_TRIANGLES`, começando no primeiro vértice (0), usando 3 vértices.

Exemplo

```
glUseProgram(program);
glBindVertexArray(VAO);
// Draws the triangle.
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Display

- O triângulo.



- Compilação dos shaders.

glCreateShader

Cria um objeto shader. Usaremos os tipos: `GL_VERTEX_SHADER` e `GL_FRAGMENT_SHADER`.

glShaderSource

Atribui o código ao objeto shader.

glCompileShader

Compila o código.

- Criação do programa com os shaders.

`glCreateProgram`

Cria um objeto programa.

`glAttachShader`

Atribui os shaders ao programa.

`glLinkProgram`

Cria o programa final (link).