DCC192

2025/2

UFMG

Desenvolvimento de Jogos Digitais

A4: Gráficos — OpenGL e Transformações

Prof. Lucas N. Ferreira

Logística

Avisos

- A entrega do **TP0: Configuração Inicial** é nesse quarta feira!
- Nem todo mundo entrou no discord

  Favor entrar no servidor: https://discord.gg/DGgdzmnU
- Nem todo mundo está usando seu nome completo

  Favor usar Nome e Sobrenome

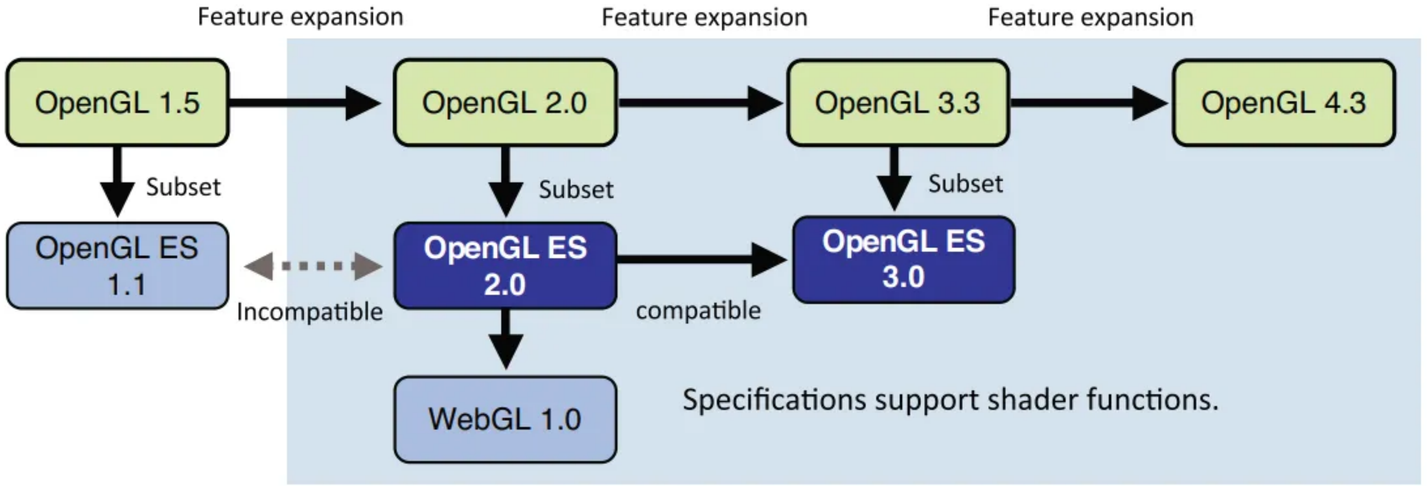
Plano de Aula

- Visão geral de um programa OpenGL/GLSL
  - Compilando Shaders
  - Contexto OpenGL e Buffers
- Transformações Geométricas
  - Coordenadas homogêneas
  - Translação, escala, rotação e reflexão
  - Combinando Tranformações

OpenGL

OpenGL (Open Graphics Library) é uma API multiplataforma para renderização de gráficos 2D e 3D. Ela é uma implementação do pipeline gráfico em GPU:

- Atua como uma ponte entre o programa e a GPU;
- Especifica comandos que são enviados à placa gráfica para desenhar objetos na tela.
- Possui versões para Web (WebGL) e para sistemas embarcados (OpenGL ES)



Desde a versão 2.0 (2004), permite programar os vertex e fragment shaders com uma linguagem chamada GLSL

GLSL

GLSL (OpenGL Shading Language) é a linguagem de programação com sintaxe similar ao C utilizada para escrever shaders, que são pequenos programas executados diretamente na GPU.

- Além dos tipos da linguagem C **`(bool, int, float, etc)`**, suporta vetores **`(vec)`** e matrizes **`(mat)`**
- Como shaders são etapas de um pipeline, eles podem especificar variáveis de entrada **`(in)`** e de saída **`(out)`**
- Shaders são compilados e ligado via comandos OpenGL **`(glCompileShader, glLinkProgram, ...)`**

```
constexpr std::string_view vertexShaderSource = R"(
#version 330 core
layout (location = 0) in vec2 aPos;
void main() {
    gl_Position = vec4(aPos, 0.0, 1.0);
}
)";
```

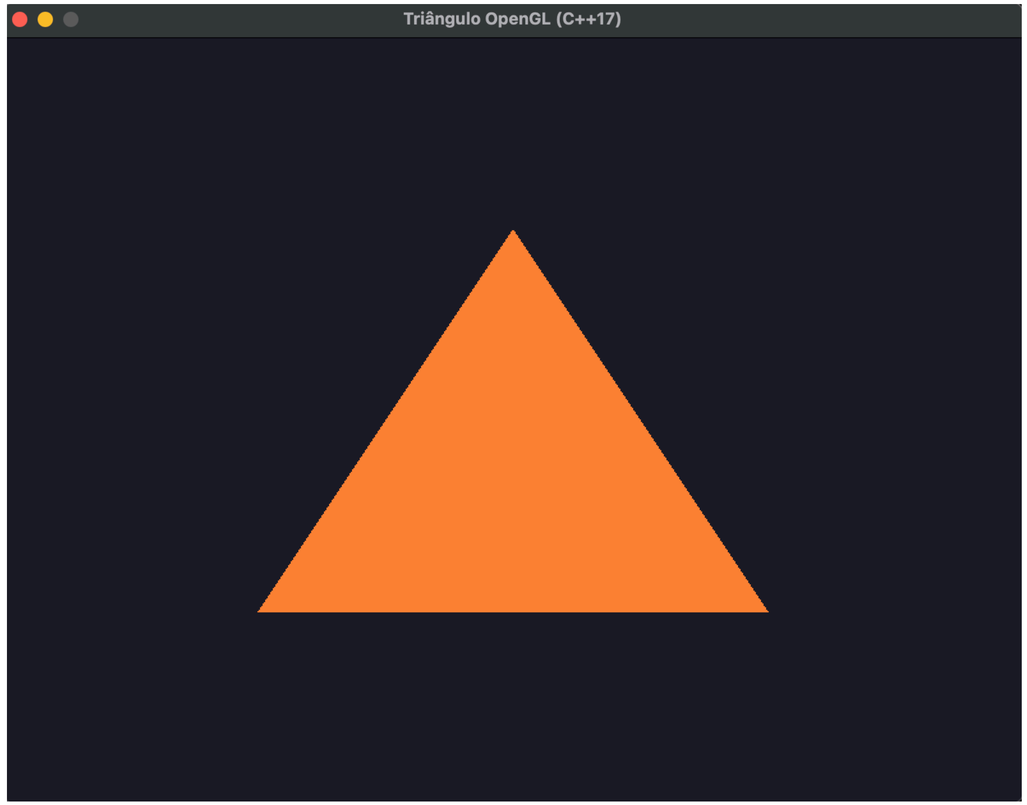
Exemplo de Vertex Shader em GLSL

```
constexpr std::string_view fragmentShaderSource = R"(
#version 330 core
out vec4 FragColor;
void main() {
    FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
)";
```

Exemplo de Fragment Shader em GLSL

Primeiro programa em OpenGL/GLSL

Um programa em OpenGL/GLSL com SDL possui as seguintes partes:

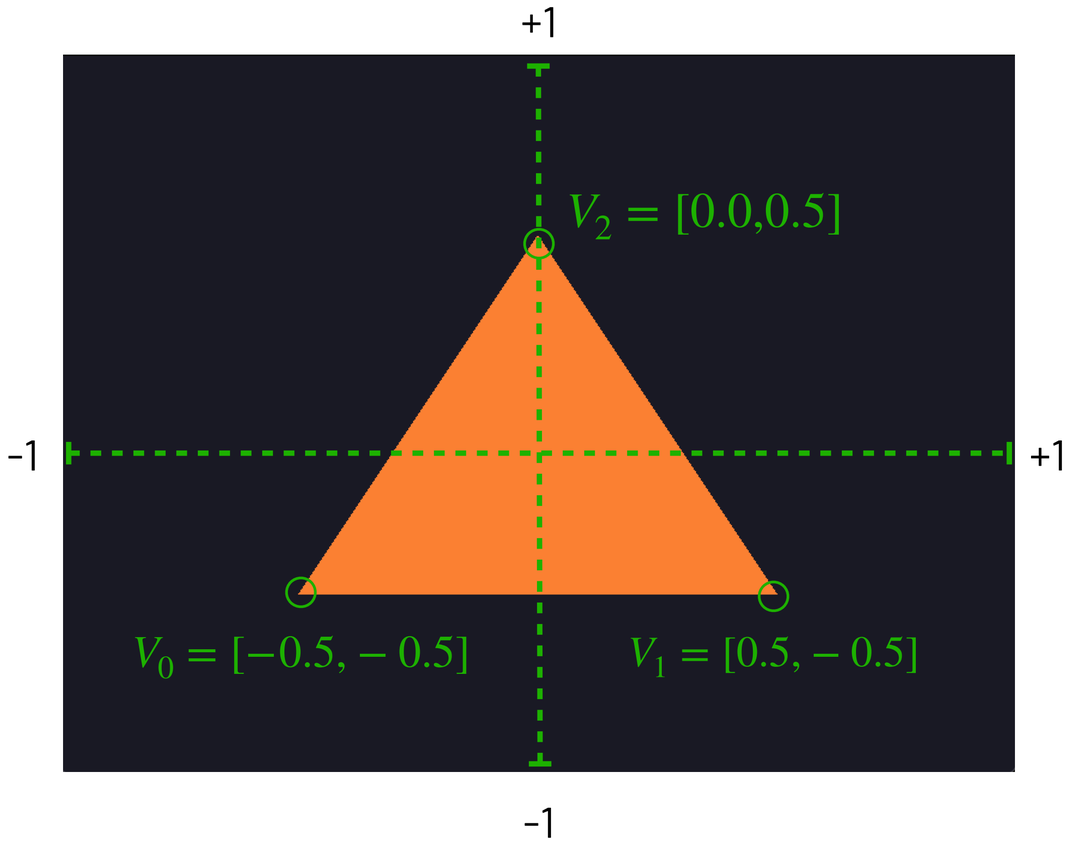


https://gist.github.com/lucasnfe/ec06ac5b7a0c44a0d3af4c87758536ad

1. Inicialização da SDL com OpenGL
2. Definir os vértices dos modelos
3. Alocar e configurar memória na GPU para os vértices:
   - Vertex Array Object (VAO)
   - Vertex Buffer Object (VBO)
   - Element Buffer Objects (EBO)
4. Escrever e compilar os vertex e fragment shaders
5. Linkar os shaders compilados em um programa na GPU
6. Limpar a tela e desenhar vértices

Definir os vértices dos modelos

Em OpenGL, especificamos modelos com uma lista de triângulos (indexada ou não), onde as coordenadas dos vértices são números reais no Sistema de Coordenadas Normalizadas:



- Uma única lista para todos os vértices do modelo;
- Cada elemento da lista é uma coordenada de um vértice;
- Valores das coordenadas x, y e z variam entre -1.0 to 1.0;
- Vértices fora desse intervalo, não serão visíveis na tela;

Por exemplo, os vértices do triângulo ao lado são específicados:

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

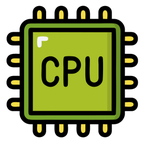
É importante manter a consistência de ordem na especificação dos vértices: sentido horário ou anti-horário

Alocar e Configurar Memória na GPU

Após especificar todos os vértices em um arranjo de floats, temos que enviá-los para a GPU, mais especificamente mapeá-los para variáveis no vertex shader, o que é feito em 3 etapas:

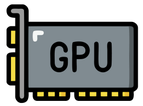
1. Alocar e ativar um Vertex Array Object (VAO) para receber as configurações de layout dos vértices

```
GLuint VAO;
glGenVertexArrays(1, &VAO); // Cria VAO
glBindVertexArray(VAO); // Liga VAO
```



-0.5	-0.5	0.5	-0.5	0.0	0.5

Vertices



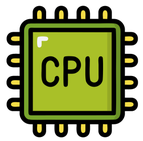
VAO

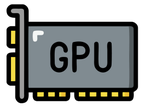
Alocar e Configurar Memória na GPU

Após especificar todos os vértices em um arranjo de floats, temos que enviá-los para a GPU, mais especificamente mapeá-los para variáveis no vertex shader, o que é feito em 3 etapas:

2. Alocar e ativar um Vertex Buffer Object (VBO) na memória da GPU para receber os vértices especificados na CPU:

```
GLuint VBO;
glGenBuffers(1, &VBO); // Cria VBO
glBindBuffer(GL_ARRAY_BUFFER, VBO); // Liga VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW); // Envia dados
```





-0.5	-0.5	0.5	-0.5	0.0	0.5	Vertices

						VAO
-0.5	-0.5	0.5	-0.5	0.0	0.5	VBO

Argumentos `glBufferData`:

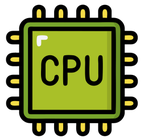
1. Índice do atributo do vértice (**`GL_ARRAY_BUFFER`**)
2. Número de dimensões do atributo (**`sizeof(vertices)`**)
3. Tipo de dado do atributo (**`vertices`**)
4. Se os dados devem normalizados (**`GL_STATIC_DRAW`**)

Alocar e Configurar Memória na GPU

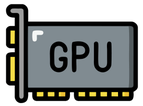
Após especificar todos os vértices em um arranjo de floats, temos que enviá-los para a GPU, mais especificamente mapeá-los para variáveis no vertex shader, o que é feito em 3 etapas:

3. Especificar no VAO como o arranjo de vértices deve ser lido pelo vertex shader:

```
glEnableVertexAttribArray(0); // Ativa atributo de vértice (location = 0)
glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
```



-0.5	-0.5	0.5	-0.5	0.0	0.5	Vertices



0	2	4	0	8	0	VAO
-0.5	-0.5	0.5	-0.5	0.0	0.5	VBO

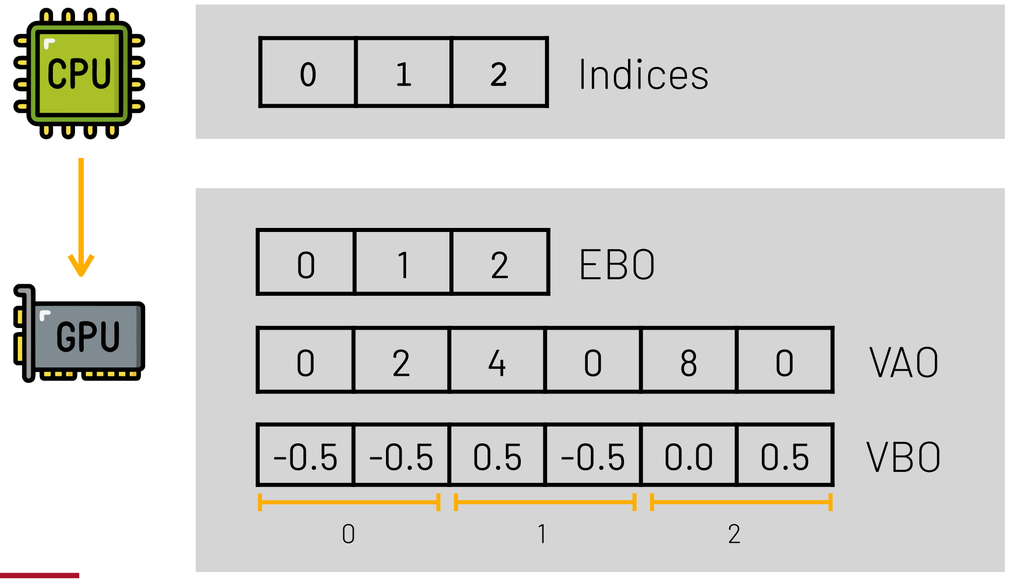
Argumentos `glVertexAttribPointer`:

1. Índice do atributo do vértice **`(location = 0)`**
2. Número de dimensões do atributo **`(1,2,3,4)`**
3. Tipo de dado do atributo **`(GL_FLOAT)`**
4. Se os dados devem normalizados **`(GL_FALSE)`**
5. Deslocamento em bytes entre vértices **`(2 * sizeof(float))`**
6. Deslocamento a partir do primeiro componente **`((void*)0)`**

Alocar e Configurar Memória na GPU

Opicionalmente, podemos alocar um Element Buffer Object (EBO) para especificar os modelos usando uma lista indexada de triângulos:

```
unsigned int indices[] = { 0, 1, 2 };
GLuint EBO;
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```



Argumentos **`glBufferData`**:

1. Índice do atributo do vértice **`(GL_ELEMENT_ARRAY_BUFFER)`**
2. Número de dimensões do atributo **`(sizeof(indices))`**
3. Tipo de dado do atributo **`(indices)`**
4. Se os dados devem normalizados **`(GL_STATIC_DRAW)`**

Escrever os Shaders: Vertex Shader

O **Vertex Shader** é um pequeno programa que roda na GPU para aplicar transformações aos vértices de entrada.

- É escrito como uma string em GLSL dentro do programa principal (ex. main.cpp)
- É compilado/linkado pela CPU e enviado pela GPU usando funções OpenGL (em tempo de execução)

```
constexpr std::string_view vShaderSrc = R"(

#version 330 core

layout (location = 0) in vec2 aPos;

void main() {

  gl_Position = vec4(aPos, 0.0, 1.0);

}
)";
```

Define a versão do GLSL usada

Declara uma variável de entrada do tipo vec2

Ponto de entrada do shader.

`gl_Position` é uma variável do tipo vec4 predefinida pela OpenGL, que armazena a posição final do vértice.

Alterar a variável `gl_Position` é equivalente a retornar o valor do vértice transformado (GLSL não usa return)

Escrever os Shaders: Fragment Shader

O **Fragment Shader** é um pequeno programa que roda na GPU para definir a cor final de cada pixel do fragmento.

- É escrito como uma string em GLSL dentro do programa principal (ex. main.cpp)
- É compilado/linkado pela CPU e enviado pela GPU usando funções OpenGL (em tempo de execução)

```
constexpr std::string_view fShaderSrc = R"(

#version 330 core

out vec4 FragColor;

void main() {

    FragColor = vec4(1.0, 0.0, 0.0, 1.0);

}
)";
```

Define a versão do GLSL usada

Declara uma variável de saída do tipo vec4

Ponto de entrada do shader

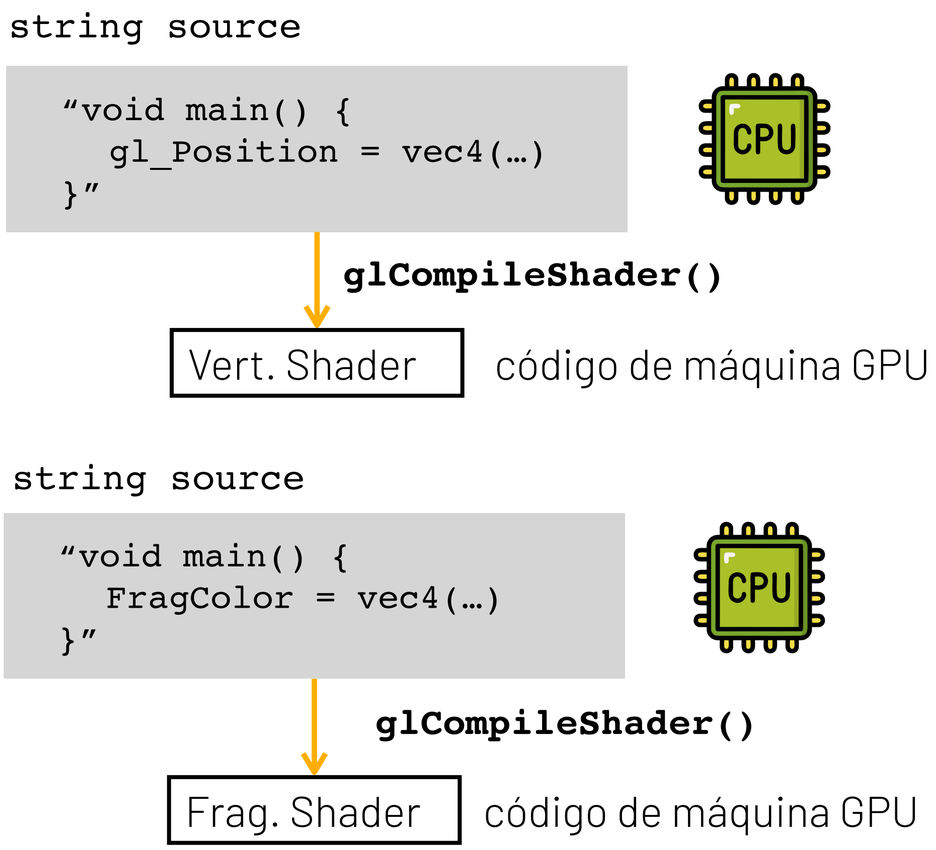
`FragColor` representa a cor final que será enviada ao framebuffer.

Alterar a variável `FragColor` é equivalente a retornar a cor do vértice (GLSL não usa return)

Compilar os Shaders

Os shaders são compilados e ligados na CPU e depois enviados a GPU. A compilação é feita usando as funções **`glCreateShader`**, **`glShaderSource`** e **`glCompileShader:`**

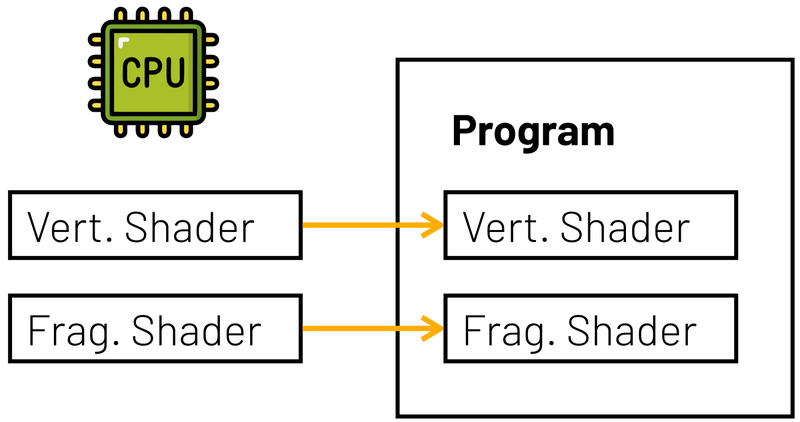
```
GLuint compileShader(GLenum type, std::string_view source)
{
    GLuint shader = glCreateShader(type);
    const char* src = source.data();
    glShaderSource(shader, 1, &src, nullptr);
    glCompileShader(shader);

    // Verifica erros de compilação
    GLint success;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &success);

    if (!success) {
        char log[512];
        glGetShaderInfoLog(shader, sizeof(log), nullptr, log);
        std::cerr << "Erro de compilação: " << log << std::endl;
    }

    return shader;
}
```



Ligar os Shaders

Após compilar os shaders, precisamos ligá-los em um programa executável na GPU, o que é feito com as funções **`glCreateProgram`**, **`glAttachShader`** e **`glLinkProgram`**:

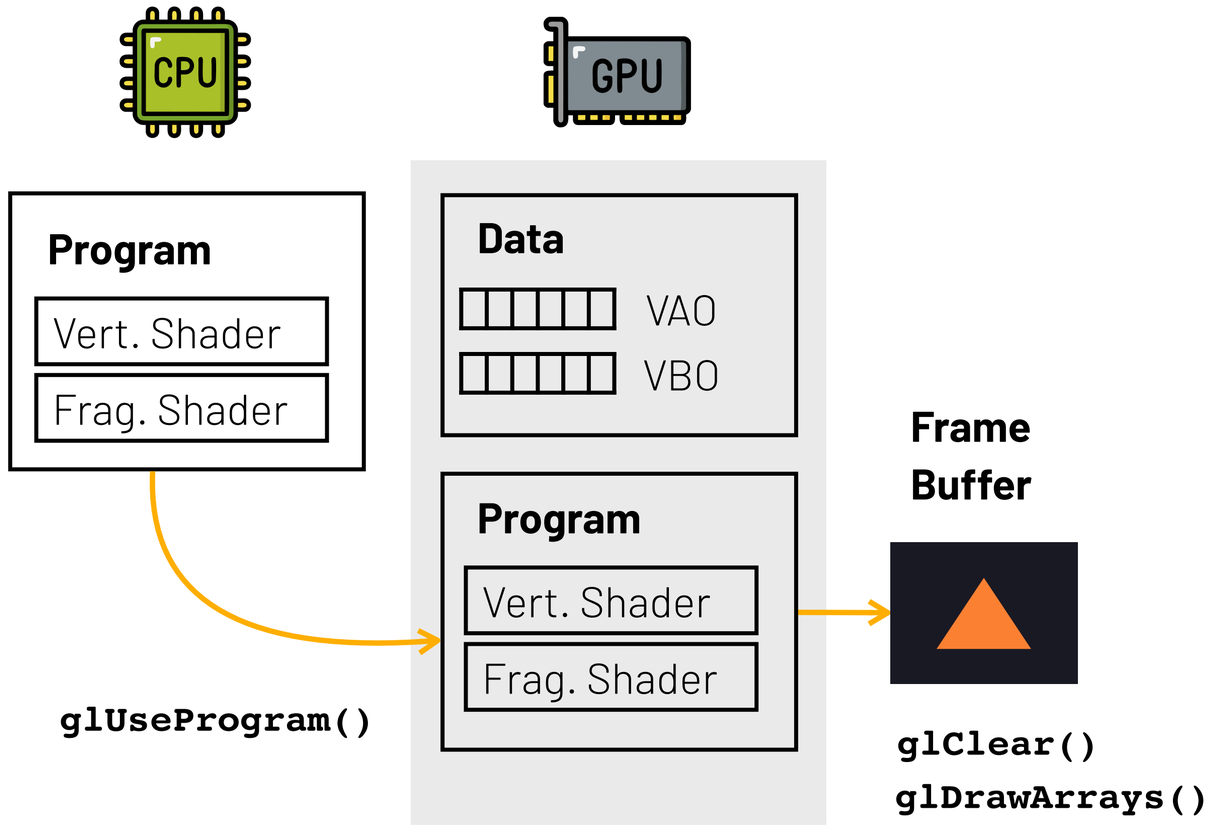
```
GLuint createShaderProgram()
{
    GLuint vShader = compileShader(GL_VERTEX_SHADER, vShaderSrc);
    GLuint fShader = compileShader(GL_FRAGMENT_SHADER, fShaderSrc);

    GLuint program = glCreateProgram();
    glAttachShader(program, vShader);
    glAttachShader(program, fShader);
    glLinkProgram(program);

    // Libera shaders após o link
    glDeleteShader(vShader);
    glDeleteShader(fShader);

    return program;
}
```



Limpar a tela e desenhar os vértices

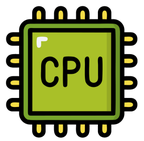
Após criar o programa executável na CPU, podemos enviá-lo para a GPU com a função **`glUseProgram`**. A GPU irá usá-lo em todas as chamadas de desenho **`glDrawArrays`** futuras:

```
GLuint shaderProgram = createShaderProgram();

// Enviar programa com os shaders para a GPU
glUseProgram(shaderProgram);

glClearColor(0.1f, 0.1f, 0.15f, 1.0f);

while (running) {
   while (SDL_PollEvent(&event)) {
        if (event.type == SDL_QUIT)
            running = false;
    }

   glClear(GL_COLOR_BUFFER_BIT);
   glDrawArrays(GL_TRIANGLES, 0, 3);

   // Double Buffer
   SDL_GL_SwapWindow(window);
}
```

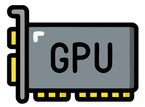


Desenhar os vértices indexados

A função **`glDrawArrays()`** é utilizada para desenhar vértices como uma lista de triângulos básica. Caso você tenha especificado o EBO, terá que usar a função **`glDrawElements()`**



-0.5	-0.5	0.5	-0.5	0.0	0.5	Vertices



0	2	4	0	8	0	VAO
-0.5	-0.5	0.5	-0.5	0.0	0.5	VBO

```
glClear(GL_COLOR_BUFFER_BIT);
glDrawArrays(GL_TRIANGLES, 0, 3);
SDL_GL_SwapWindow(window);
```

-0.5	-0.5	0.5	-0.5	0.0	0.5	Vertices

0	1	2	Indices

0	2	4	0	8	0	VAO
-0.5	-0.5	0.5	-0.5	0.0	0.5	VBO
0	1	2				EBO

```
glClear(GL_COLOR_BUFFER_BIT);
glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0);
SDL_GL_SwapWindow(window);
```

Variáveis GLSL: Ins e Outs

Como shaders fazem parte de um pipeline gráfico, eles podem criar variáveis de entrada (`in`) e saída (`out`) para transferir dados entre as etapas do pipeline.

- Para passar valores entre os shaders, basta criar uma variável de saída (out) no vertex shader com o mesmo de uma variável de entrada (in) no fragment shader:

```glsl
#version 330 core
layout (location = 0) in vec2 aPos;

out vec3 vertexColor;

void main() {
    gl_Position = vec4(aPos, 0.0, 1.0);
    vertexColor = vec3(0.5, 0.0, 0.0);
}
```

Vertex Shader

As entradas do são atributos de vértices, por isso são definidas com `layout (location = )`

```glsl
#version 330 core
out vec4 FragColor;

in vec3 vertexColor;

void main() {
    FragColor = vec4(vertexColor, 1.0);
}
```

Fragment Shader

Deve obrigatoriamente ter uma variável de saída (out) do tipo vec4 especificando uma cor.

Variáveis GLSL: Uniforms

Por padrão, as variáveis de entrada (in) do vertex shader são atributos de vértices, ou seja, possuem um valor diferente para cada vértice. Se quisermos criar uma variável com um valor único para todos os vértices, podemos definir uma variável com a palavra reservada **`uniform`**

- Variáveis **`uniform`** são globais, ou seja, são únicas por objeto de programa
- Elas podem ser acessadas tanto pelo vertex shader, quanto pelo fragment shader
- Quando definimos valores seus valores, eles são mantidos até que sejam redefinidos ou atualizados.

```
GLint colorLocation = glGetUniformLocation(shaderProgram, "vertexColor");
glUniform3f(colorLocation, 0.7f, 0.0f, 0.2f); // Red
```

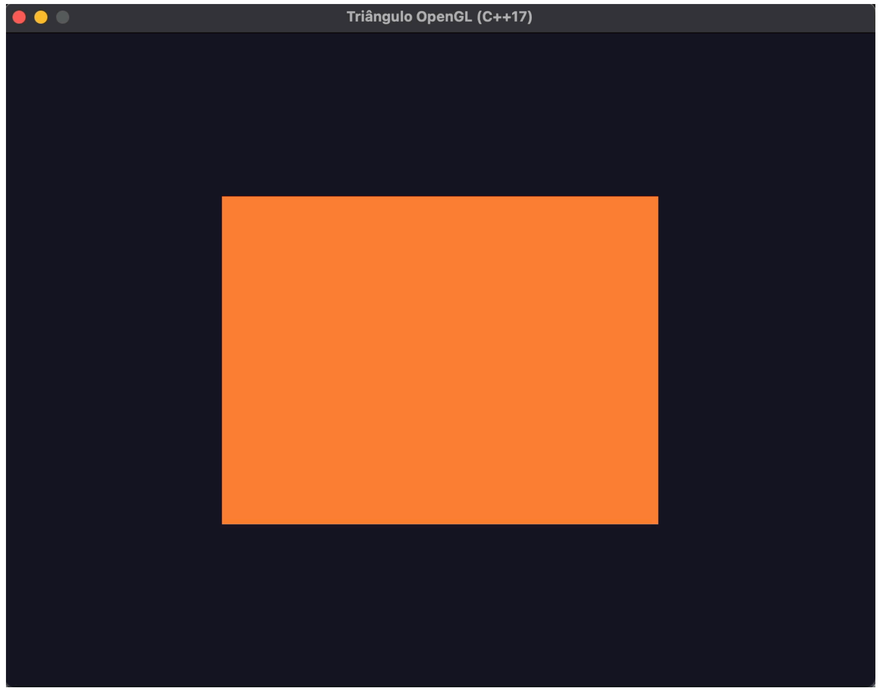
Programa Principal (CPU)

```
#version 330 core
out vec4 FragColor;

Uniform vec3 vertexColor;

void main() {
    FragColor = vec4(vertexColor, 1.0);
}
```

Fragment Shader (GPU)

Exemplo de uso de Uniforms

Programa exemplo de como uniforms em conjunto com variáveis de entrada em GLSL:



A cor do triângulo é alterarda para quando o usário pressiona as teclas R, G e B.

```
GLint colorLocation = glGetUniformLocation(shaderProgram, "inColor");
glUniform3f(colorLocation, 0.7f, 0.0f, 0.2f); // Red
```

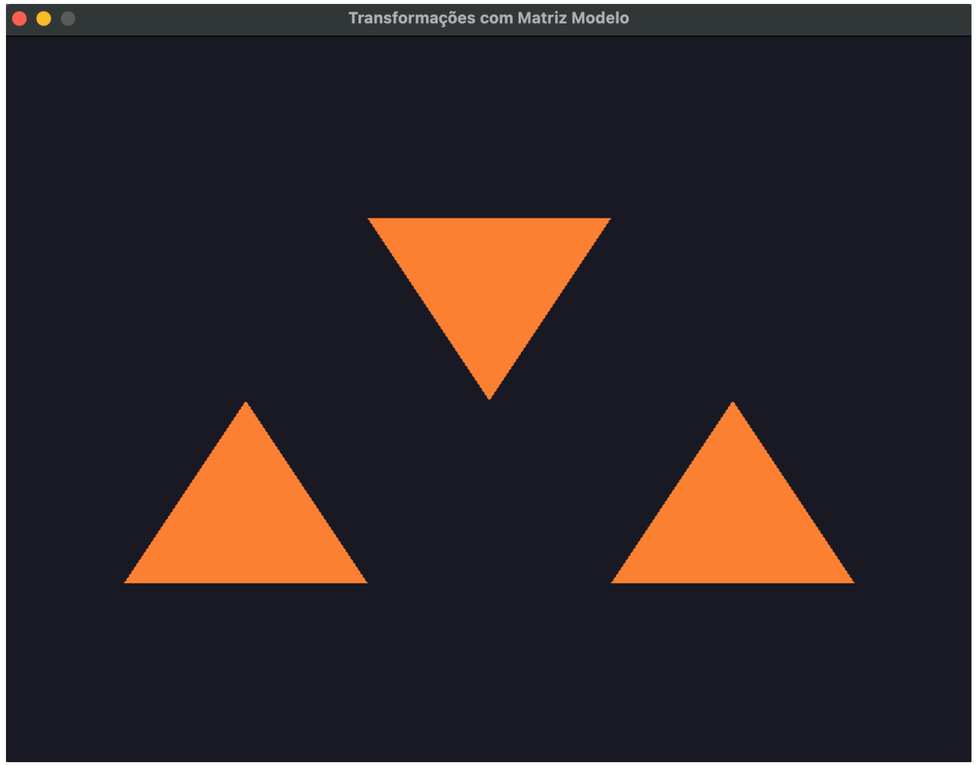
```
layout (location = 0) in vec2 aPos;
uniform vec3 inColor;
out vec3 outColor;

void main() {
    gl_Position = vec4(aPos, 0.0, 1.0);
    outColor = inColor;
}
```

```
in vec3 outColor;
out vec4 FragColor;

void main() {
    FragColor = vec4(outColor, 1.0); // Define a cor do pixel
}
```

https://gist.github.com/lucasnfe/166ee873de3eb23b40afc78d88772f93

Transformações Geométricas

Para criar múltiplas instâncias de um mesmo objeto, mantemos uma única lista de vértices representando esse objeto e aplicamos **transformações geométricas** aos vértices:



- Transformações geométricas mais comuns em jogos:
  Translação, Rotação, Escala e Reflexão
- Vértices 3D são representado em 4D, o que chamamos de **coordenadas homogêneas**
- Transformações são representadas por matrizes $M_{4\times4}$, que são combinadas em uma única matriz final, também $M_{4\times4}$, chamada de **model matrix**
- Apenas a **model matrix** é enviada ao vertex shader, que realiza a multiplicaçãm em todos os vértices em paralelo.

https://gist.github.com/lucasnfe/bd7c8f8ffc28db956451cb9386dc83a3

Coordenadas Homonêneas

Em aplicações gráficos, pontos e vetores em 3D são tipicamente representados com uma dimensão a mais, o que chamamos de **coordenadas homogêneas**:

Para representar pontos $P_i \in \mathbb{R}^3$ em coordenadas homogênas, adicionamos uma dimensão $w = 1$:

$P_i \in \mathbb{R}^3$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Por hora, vamos utilizar apenas essa representação com $w = 1$ para vértices, que são pontos.

Para representar vetores (direções) $V_i \in \mathbb{R}^3$ em coordenadas homogênas, adicionamos dimensão $w = 0$:

$V_i \in \mathbb{R}^3$

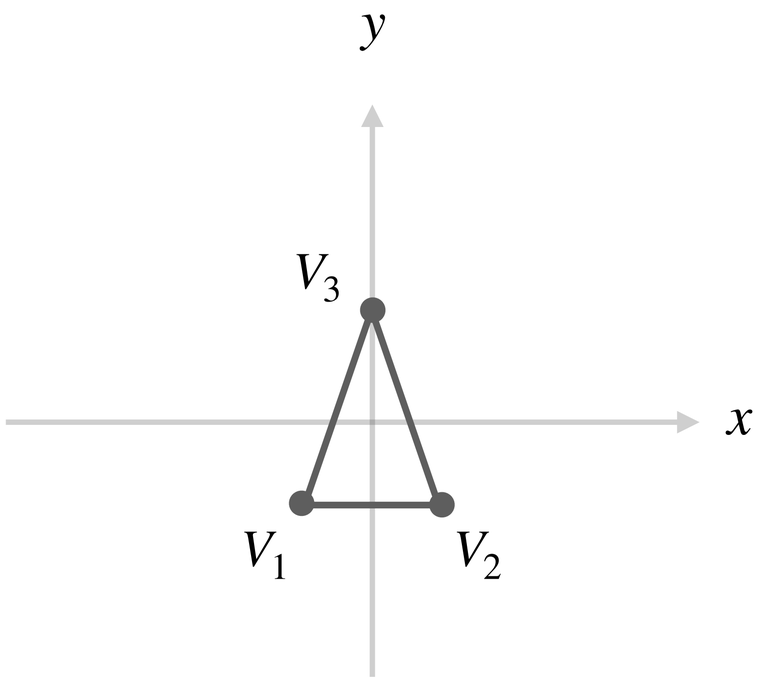
$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

Mais tarde no curso, iremos usar essa representação com $w = 0$ para outras operações

Coordenadas Homogêneas nos permitem combinar as transformações em uma única matriz, o que é mais eficiente, pois evita tráfego CPU x GPU

Translação

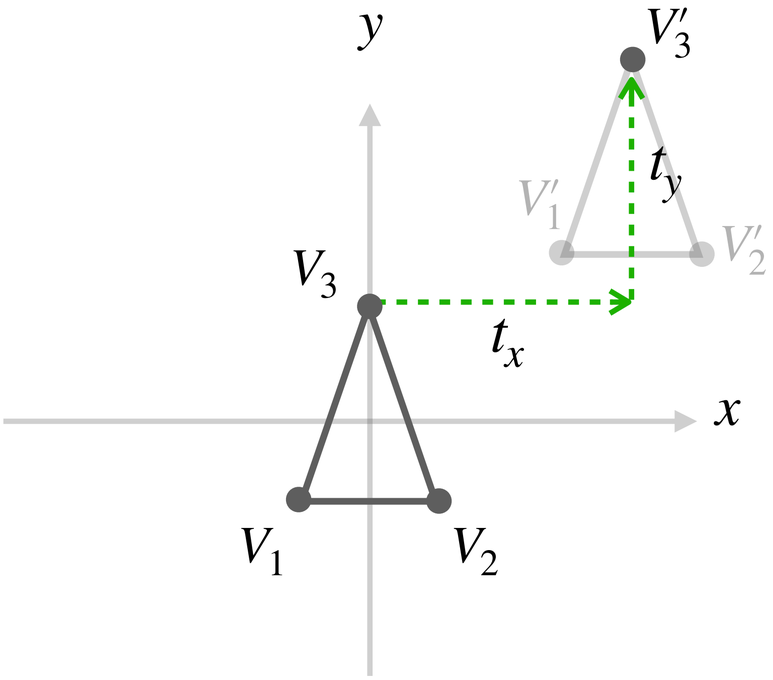
Uma transformação de **translação** em coordenadas homogêneas $T(x, y, z)$ é presentada por uma matriz identidade, onde a quarta coluna contém o deslocamento em cada eixo:



$$V_i' = T(x, y, z) \cdot V_i$$

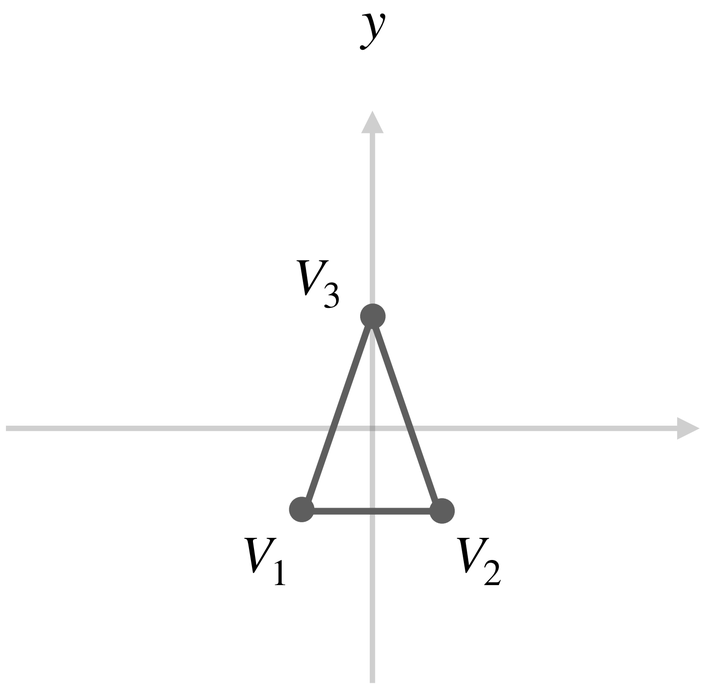
$$V_i' = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

- t_x: deslocamento em x
- t_y: deslocamento em y
- t_z: deslocamento em z



Escala

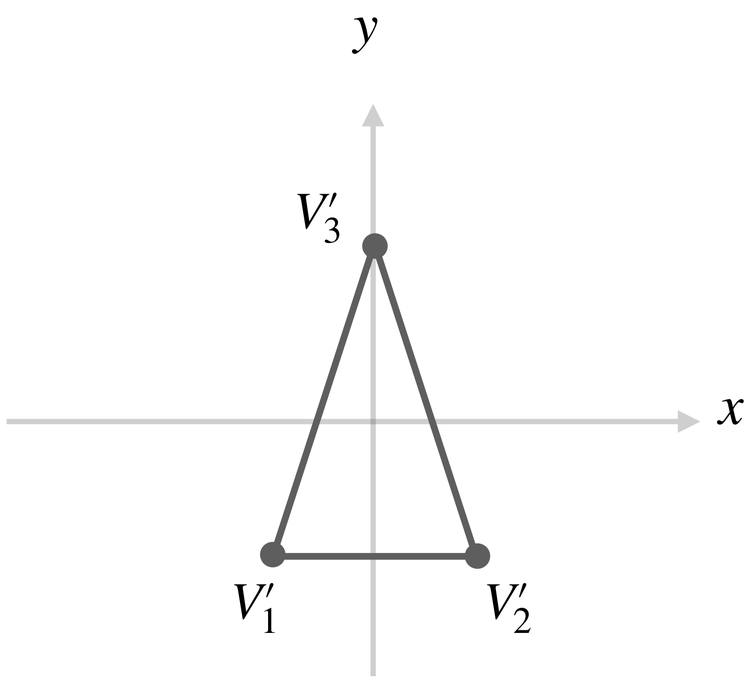
Uma transformação de **escala** em coordenadas homogêneas $S(x, y, z)$ é presentada por uma matriz identidade, onde a diagonal principal contém os fatores de escala em cada eixo:



$$V_i' = S(x, y, z) \cdot V_i$$

$$V_i' = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot s_x \\ y \cdot s_y \\ z \cdot s_z \\ 1 \end{bmatrix}$$

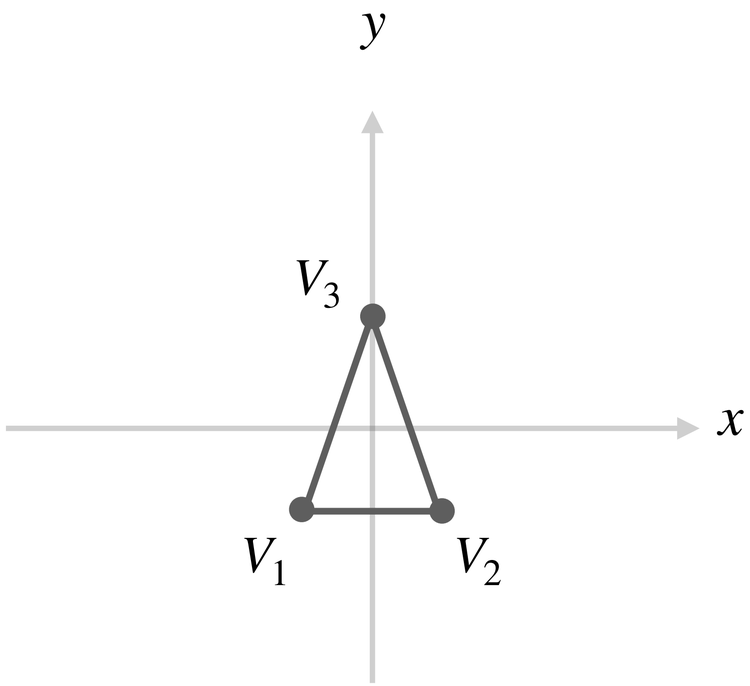
- s_x: escala em x
- s_y: escala em y
- s_z: escala em z



É importante observar que se o modelo não estivesse centralizado no sistema de coordenadas, a escala mudaria o centro do objeto.

Rotação em 2D

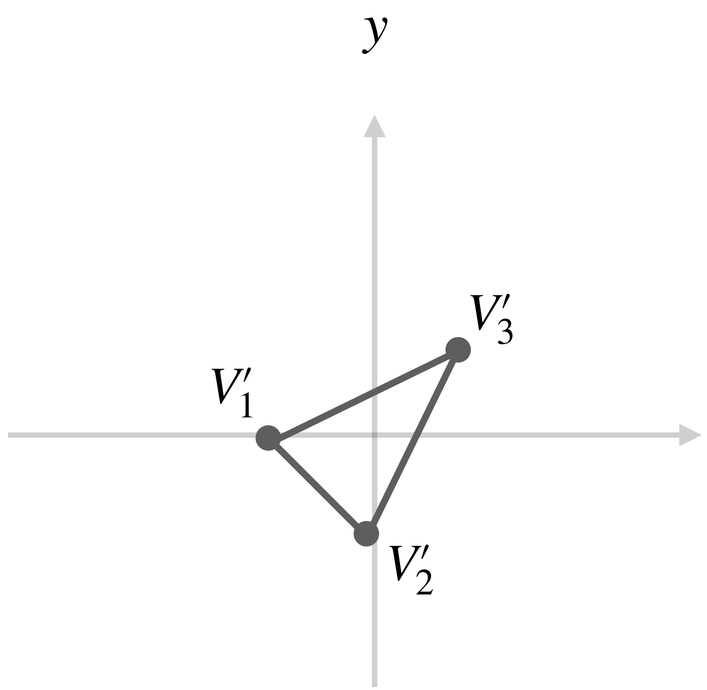
Antes de falarmos de rotação em 3D utilizando coordenadas homogêneas, vamos ver como realizar uma transformação de **rotação** em 2D, com uma matriz de rotação R:



$$V'_i = R \cdot V_i$$

$$V'_i = \begin{bmatrix} cos\ \theta & -sin\ \theta \\ sin\ \theta & cos\ \theta \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

- θ: ângulo de rotação



Rotação em 2D

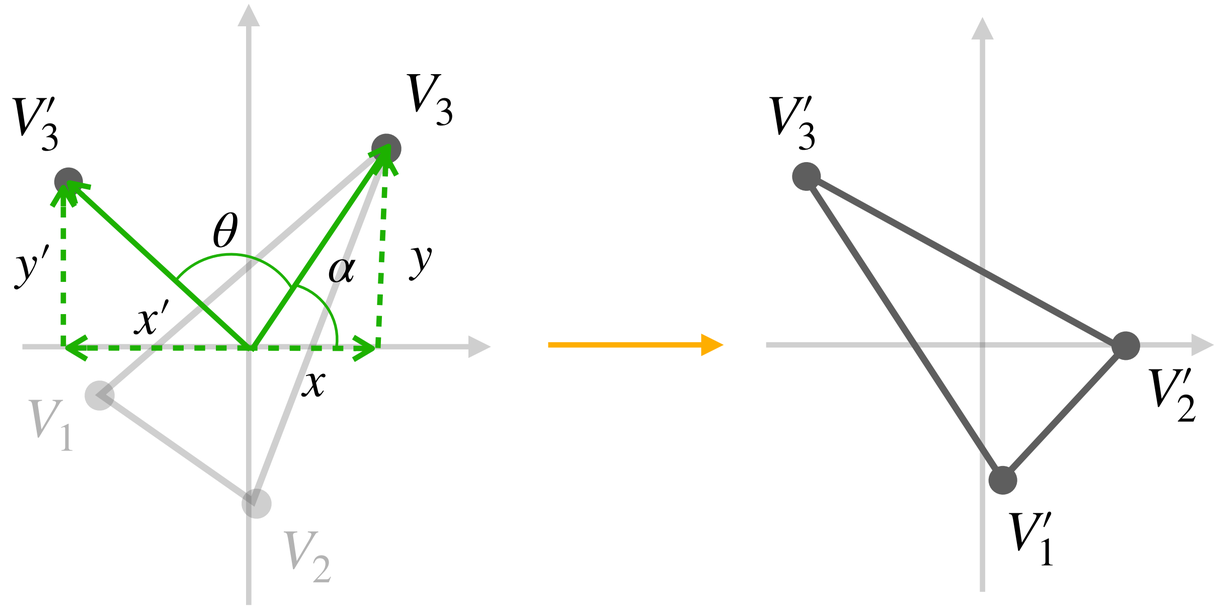
Vejamos como a matriz de rotação $\boldsymbol{R}$ é construída em 2D:

$$x = r \cdot cos\ \alpha \rightarrow x' = r \cdot cos(\alpha + \theta)$$
$$y = r \cdot sin\ \alpha \rightarrow y' = r \cdot sin(\alpha + \theta)$$

$$x' = r \cdot cos\ \alpha \cdot cos\ \theta - r \cdot sin\ \alpha \cdot sin\ \theta$$
$$y' = r \cdot cos\ \alpha \cdot sin\ \theta + r \cdot sin\ \alpha \cdot cos\ \theta$$

$$x' = x \cdot cos\ \theta - y \cdot sin\theta$$
$$y' = x \cdot sin\ \theta + y \cdot cos\theta$$

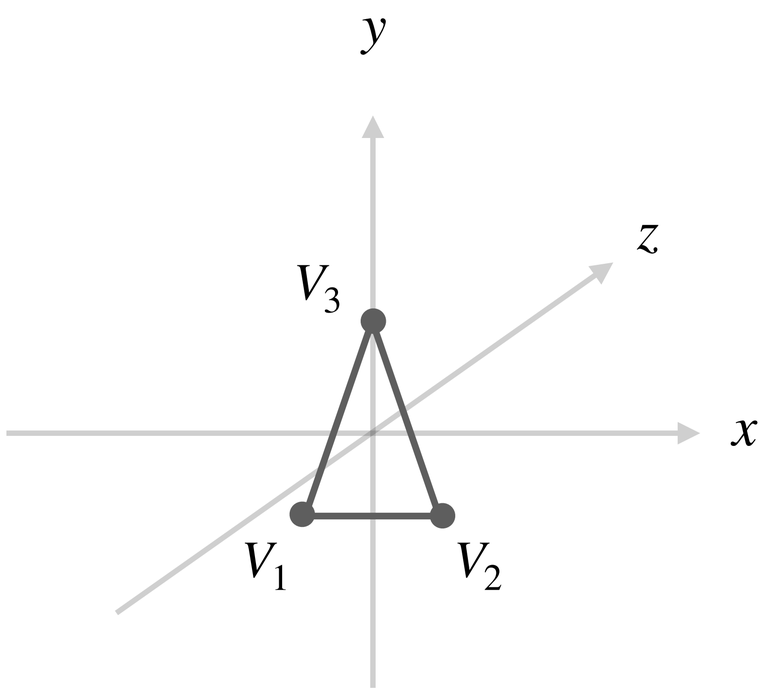
$$V' = \begin{bmatrix} cos\ \theta & -sin\ \theta \\ sin\ \theta & cos\ \theta \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} \rightarrow V' = R \cdot V_i$$



É importante observar que se o modelo não estivesse centralizado no sistema de coordenadas, a rotação também mudaria o centro do objeto.

Rotação em 3D

Em 3D, temos 3 diferentes eixos para fazer a rotação: x, y e z. As matrízes de rotação são diferentes para cada eixo. Abaixo elas estão definidas já em coordenadas homogêneas:



$$\xrightarrow{R_x(\theta)} \quad V_i' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\ \theta & -sin\ \theta & 0 \\ 0 & sin\ \theta & cos\ \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

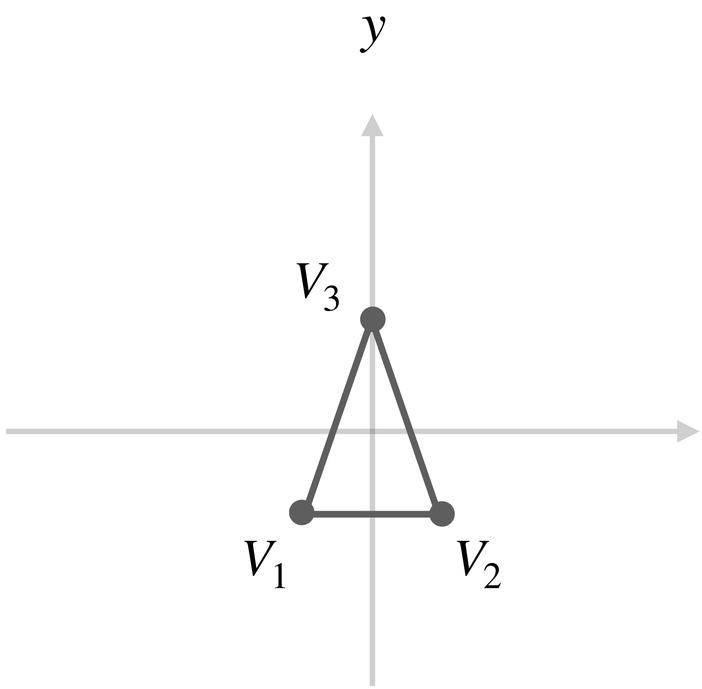
$$\xrightarrow{R_y(\theta)} \quad V_i' = \begin{bmatrix} cos\ \theta & 0 & sin\ \theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\ \theta & 0 & cos\ \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\xrightarrow{R_z(\theta)} \quad V_i' = \begin{bmatrix} cos\ \theta & -sin\ \theta & 0 & 0 \\ sin\ \theta & cos\ \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

A rotação no eixo z é a mesma rotação que fazemos em 2D.

Reflexão

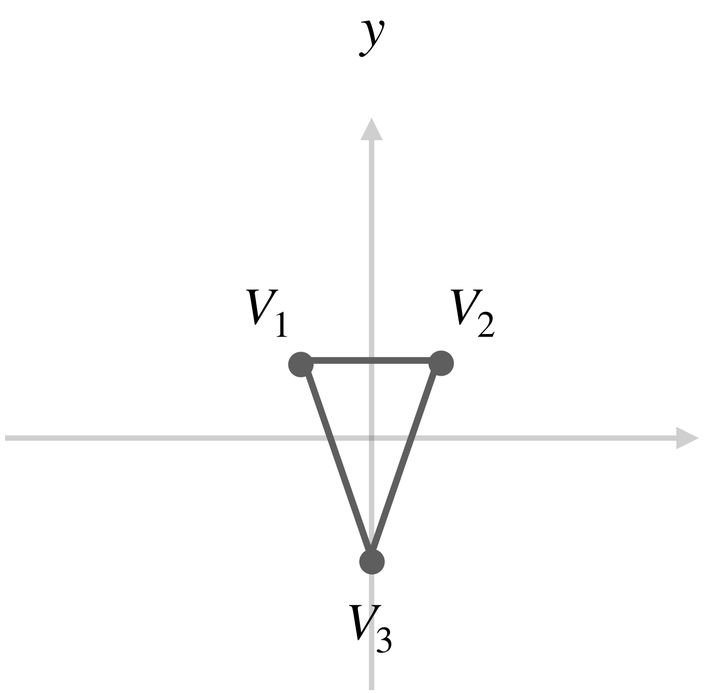
Uma transformação de **reflexão** pode ser realizada com uma matriz de escala $S(x, y, z)$, porém especificando valores negativos para um ou três dos eixos:



$$V_i' = S(x, y, z) \cdot V_i$$

$$V_i' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot 1 \\ y \cdot -1 \\ z \cdot 1 \\ 1 \end{bmatrix}$$

- s_x: escala em x
- s_y: escala em y
- s_z: escala em z



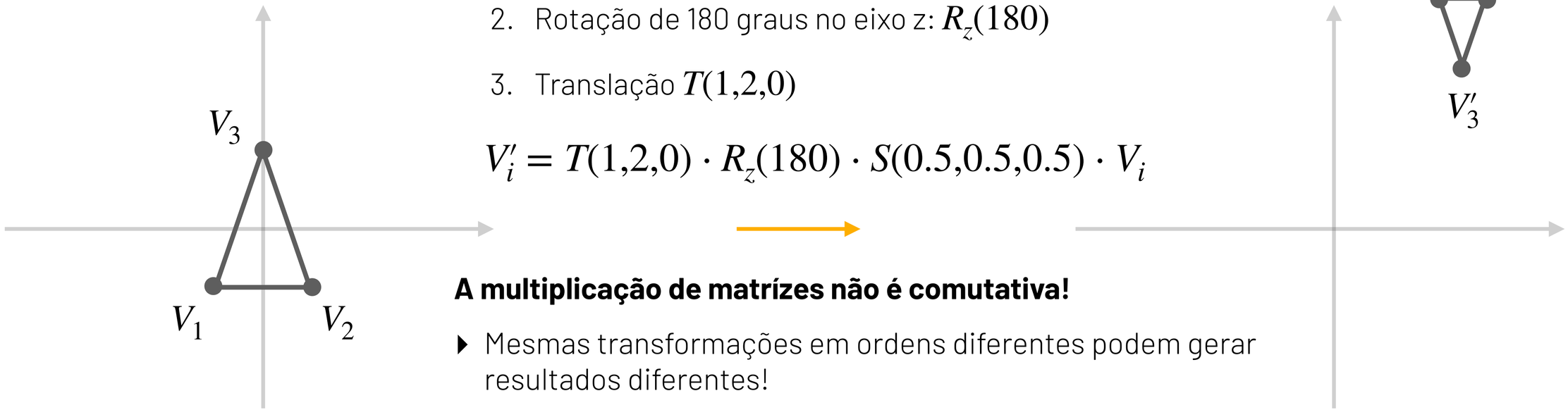
É importante observar que se apenas dois dos fatores de escala foram negativos, o objeto irá rotacionar π radianos (180 graus)

Combinando Transformações

Utilizando coordenadas homogêneas, podemos relizar composições de transformações multiplicandos as matrizes de transformação:

1. Escala uniforme de 0.5: $S(0.5,0.5,0.5)$
2. Rotação de 180 graus no eixo z: $R_z(180)$
3. Translação $T(1,2,0)$

$$V_i' = T(1,2,0) \cdot R_z(180) \cdot S(0.5,0.5,0.5) \cdot V_i$$



A multiplicação de matrízes não é comutativa!

- Mesmas transformações em ordens diferentes podem gerar resultados diferentes!
- Para evitar problemas, realizamos as transformações na ordem do exemplo acima (leia as multiplicações de trás para frente): 1. Escala, 2. Rotação e 3. Translação

Criando Matrizes de Transormação em OpenGL

Como os modelos podem possuir muitos vértices, é mais eficiente aplicar as transformações na GPU. Para isso, criamos e combinamos as matrízes na CPU e as enviamos ao Vertex Shader:

Programa Principal (CPU)

1. Buscar o endereço da matriz (uniform) na GPU com a função `glGetUniformLocation()`
2. Criar uma *model matriz* na CPU multiplicando diferentes matrizes de transformações
3. Enviar *model matriz* para a GPU com a função `glUniformMatrix4fv()`

Vertex Shader (GPU)

1. Criar uma matriz 4×4 (uniform) no vertex shader
2. Multiplicar essa matriz pelos vértices de entrada

```
GLint modelLoc = glGetUniformLocation(shaderProgram, “model");

Matrix4 model = Matrix4::CreateTranslation(Vector3(0.0f, 0.25f, 0.0f));
model = Matrix4::CreateScale(Vector3(0.5f, 0.5f, 1.0f)) * model;
model = Matrix4::CreateRotationZ(Math::ToRadians(180.0f)) * model;

glUniformMatrix4fv(modelLoc, 1, GL_FALSE, model3.GetAsFloatPtr());
```

```
#version 330 core
layout (location = 0) in vec2 aPos;
uniform mat4 model;

void main() {
    gl_Position = model * vec4(aPos, 0.0, 1.0);
}
```

Próxima aula

A5: Game Loop

- Técnicas de controle de tempo em jogos
- FPS fixo vs. FPS dinâmico