Lista 1 – Processamento Gráfico

1. O que é a GLSL? Quais os dois tipos de shaders são obrigatórios no pipeline programável da versão atual que trabalhamos em aula e o que eles processam?

O GLSL (OpenGL Shanding Language) é uma linguagem de programação usada para escrever programas de shaders.
Os dois tipos de shaders qu são obrigatórios no pipeline programável são o Vertex Shader e o Fragment Shader. O Vertex Shader é responsável por processar as coordenadas dos vértices dos objetos 3D, geralmente aplicando transformações geométricas, como rotação e escala. Eles também calculam a posição final dos vértices após a projetão na tela. O Fragment Shader controla a aparência dos fragmentos de pixels na tela. Eles determinam a cor de cada pixel com em diversos fatores, como iluminação, sombras, texturas e materiais.

2. O que são primitivas gráficas? Como fazemos o armazenamento dos vértices na OpenGL?

As primitivas gráficas são elementos básicos usados na computação gráfica para representar objetos ou formas geométricas no espaço tridimensional. Estas primitivas são os blocos fundamentais para desenhar objetos mais complexos. Como exemplos, temos: vértices, triângulos, linhas, etc…

Para armezanas os vértices no OpenGl seguimos alguns passos:

A- Definição dos vértices: criamos um conjunto de coordenadas dos vértices para descrever a primitiva que queremos renderizar. Neste caso, estmoas definindo vértices 3D(x,y,z).

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

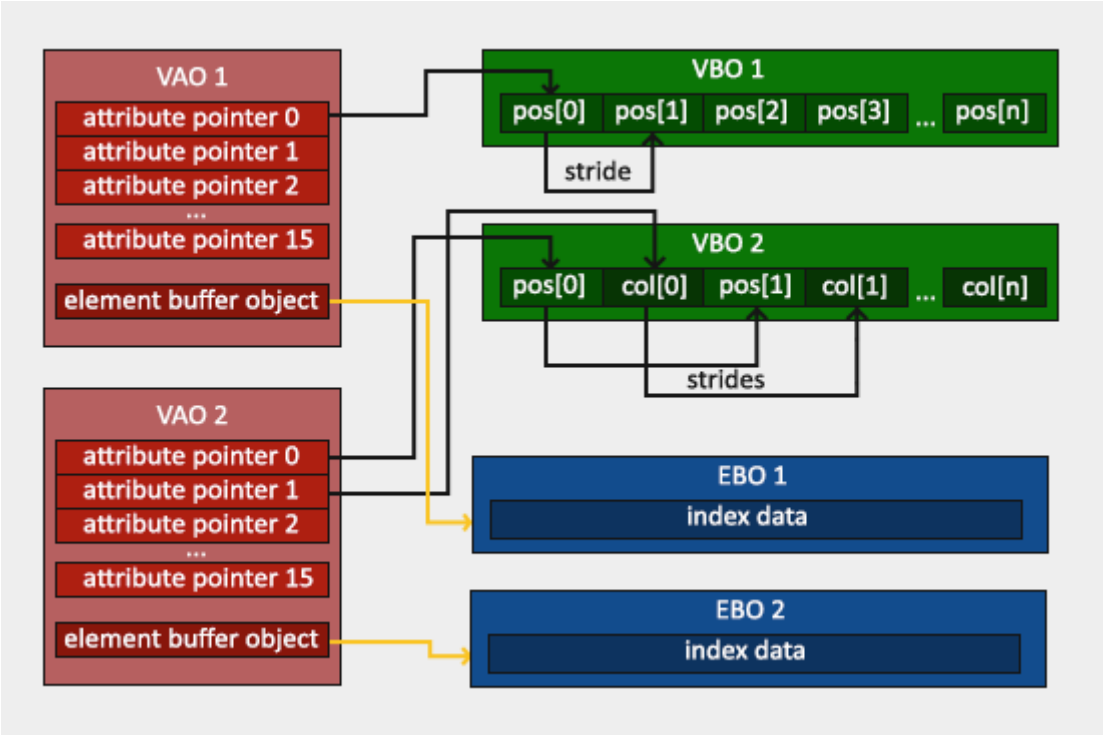
B- Criação de um buffer de vértices e load dos dados dos vértices no buffer: A OpenGL fornece uma api para a criação de um buffer de vértices que será armazenado na memória da GPU e também para load dos dados dos vértices no buffer.

```
// Vertex Buffers
unsigned int VBO, VAO, EBO;
//Generate Vertex Array VAO
glGenVertexArrays(1, &VAO);
// Generate n VBO and store the ID
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);
//bind VAO
glBindVertexArray(VAO);
// Bind Buffer type
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// Sends user data to buffer
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
// Sends user data to buffer
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);
```

3- Explique o que é VBO, VAO e EBO, e como se relacionam (se achar mais fácil, pode fazer um gráfico representando a relação entre eles).

- Um VBO (Vertex Buffer Object) é um buffer de memória na GPU usado para armazenar os dados dos vértices de objetos gráficos.
- Um VAO (Vertex Array Object) é um objeto que encapsula o estado da configuração dos VBOs. Ele registra os dados nos VBOs devem ser interpretados durante o processo de renderização. VAOs simplificam a troca entre objetos gráficos durante a renderização, pois armazenam as configurações necessárias.
- Um EBO (Element Buffer Object) é um buffer adicional que pode ser usado para otimzar o armazenamento de índices dos vértices. Ele é frequentemente utilizado em conjunto com VBOs quando se renderizam objetos complexos com vértices repetidos.

Na imagem abaixo, podemos ver a relação dos três tipos de objetos.

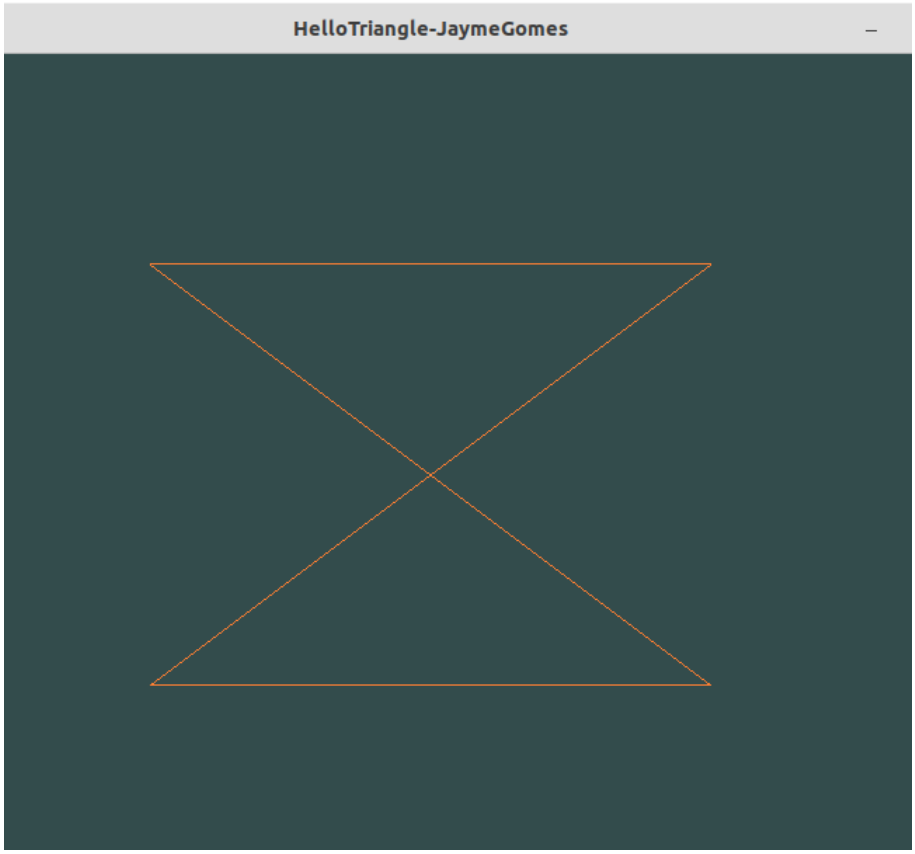


- Um VBO armazena os dados dos vértices, como suas coordenadas, cores e outros atributos.
- Um EBO (quando usado) armazena os índices que apontam para os vértices nos VBOs.
- Um VAO armazena as configurações que dizem à OpenGL como interpretar os dados nos VBOs (por exemplo, como os atributos estão organizados e onde encontrar os dados nos VBOs).

3- Explique o que é VBO, VAO e EBO, e como se relacionam (se achar mais fácil, pode fazer um gráfico representando a relação entre eles).

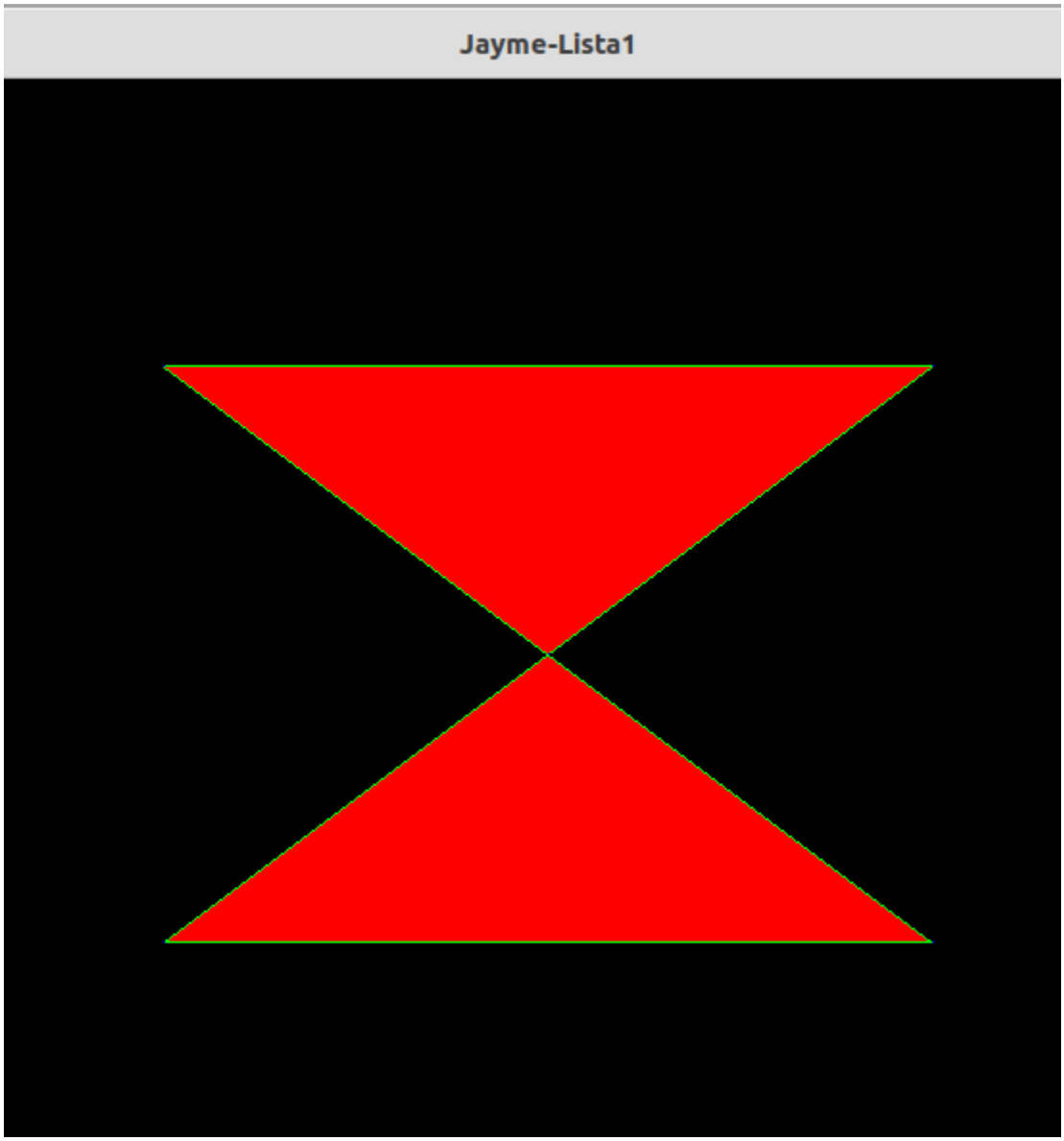
5. Faça o desenho de 2 triângulos na tela. Desenhe eles:
    a. Apenas com o polígono preenchido

b. Apenas com contorno

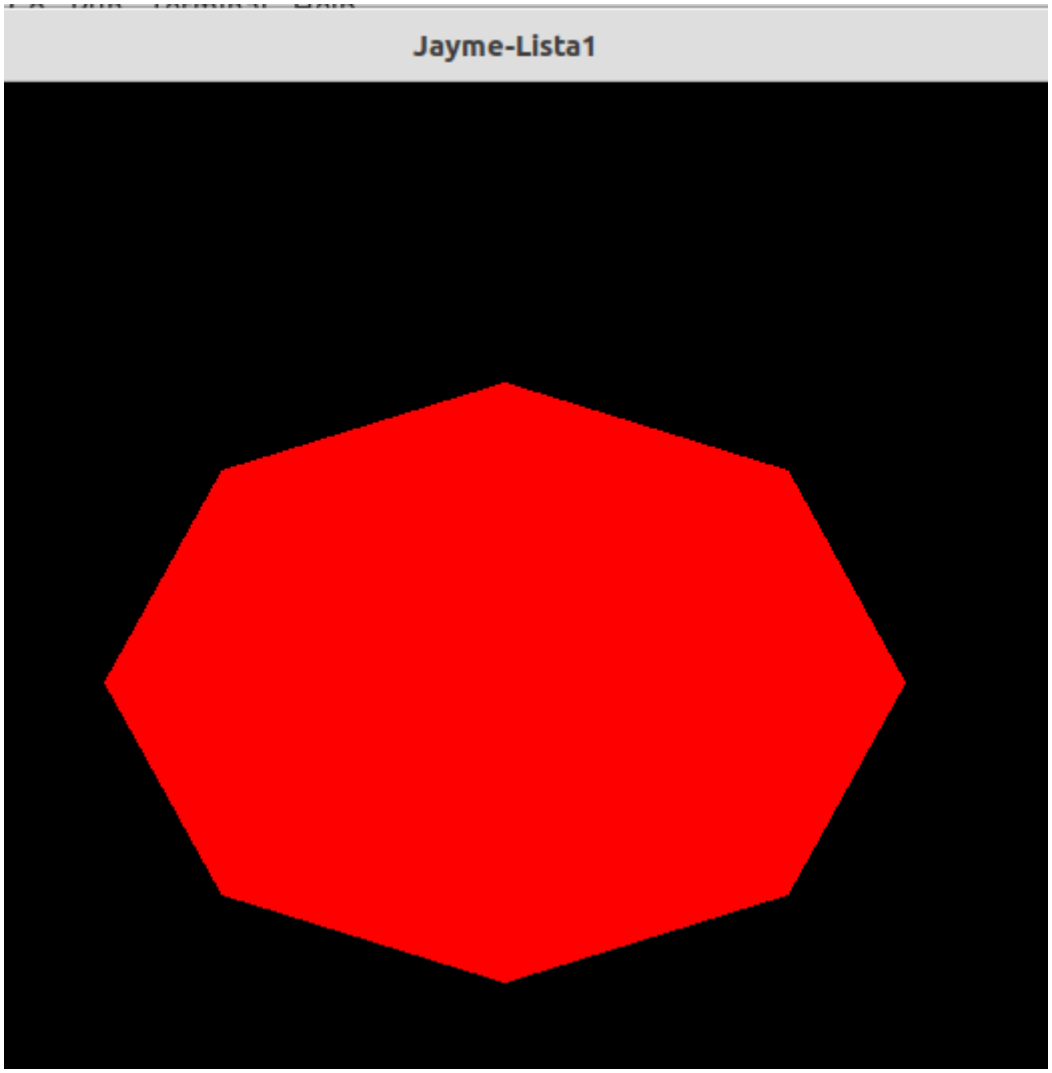


c. Apenas como pontos

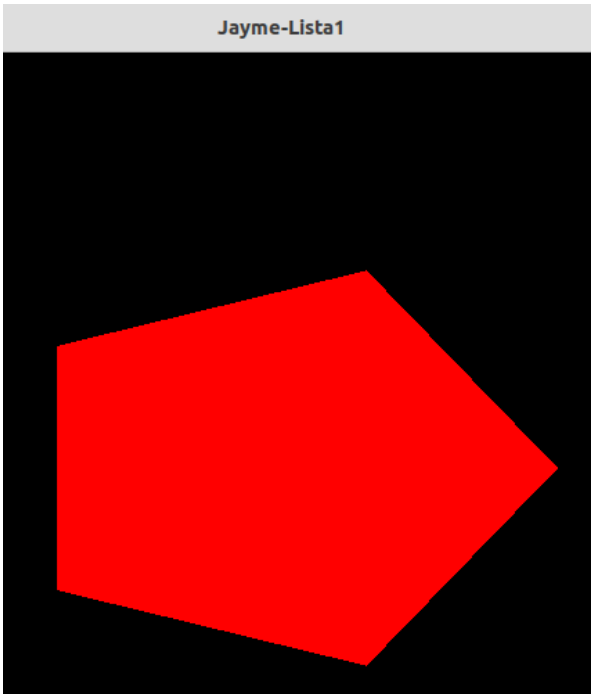
d. Com as 3 formas de desenho juntas



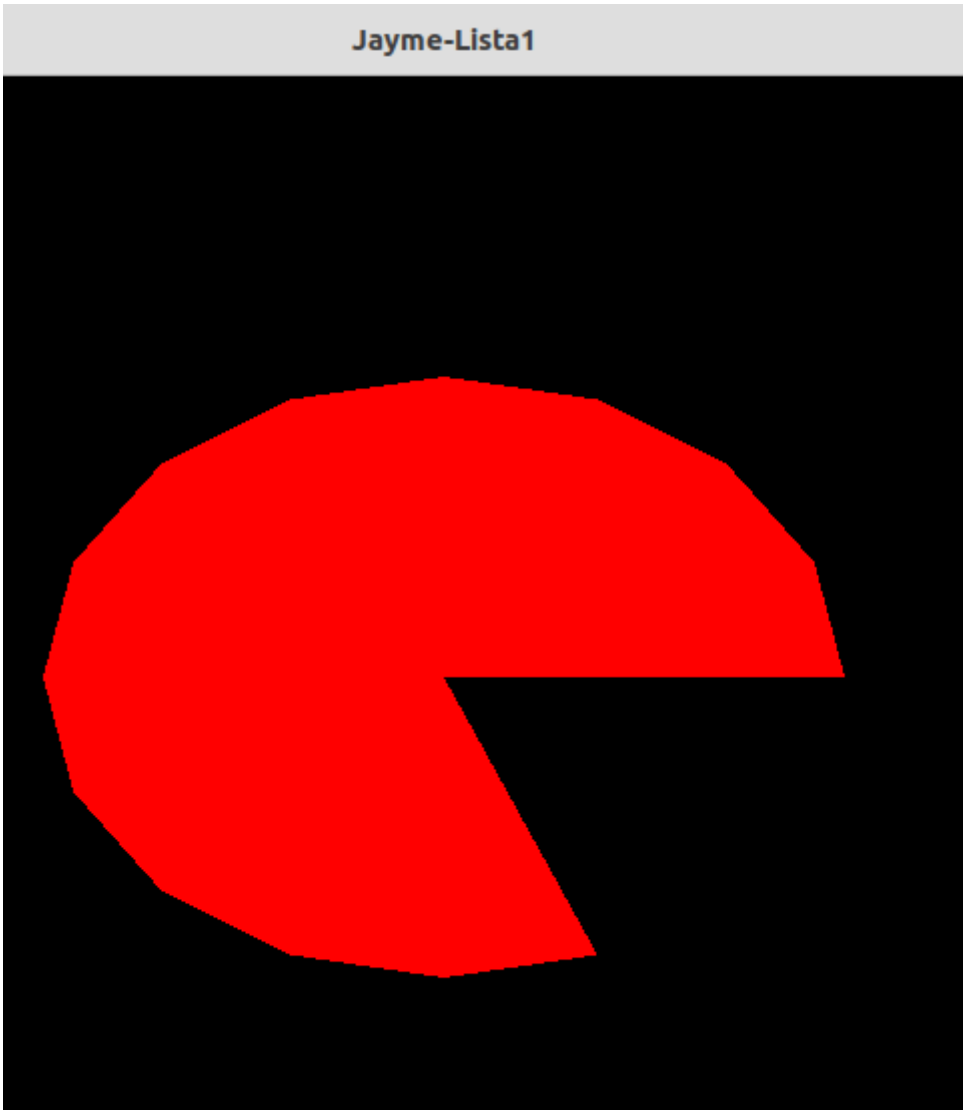
6-
A-



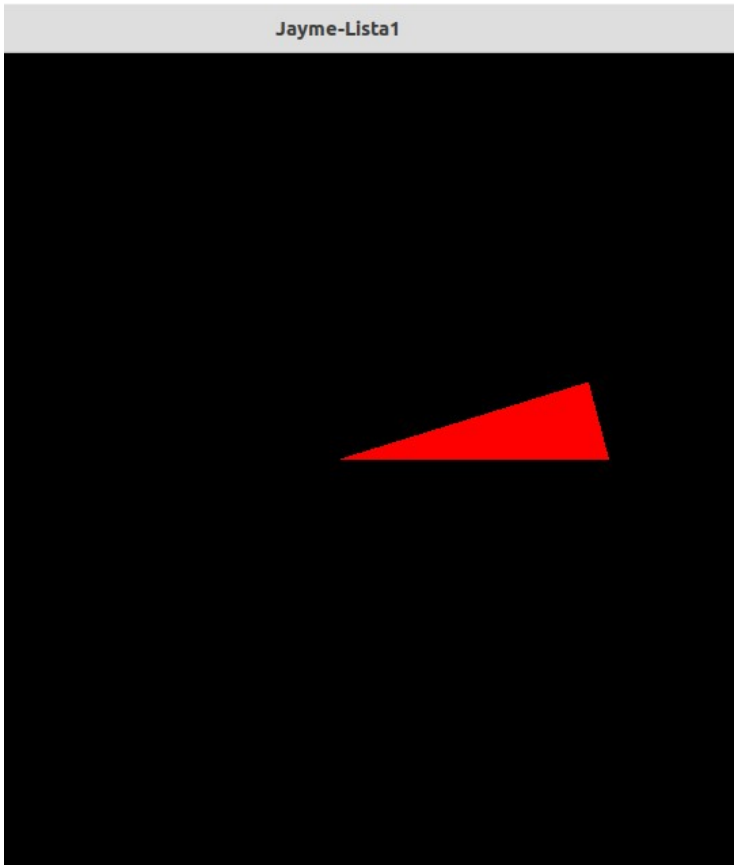
B-



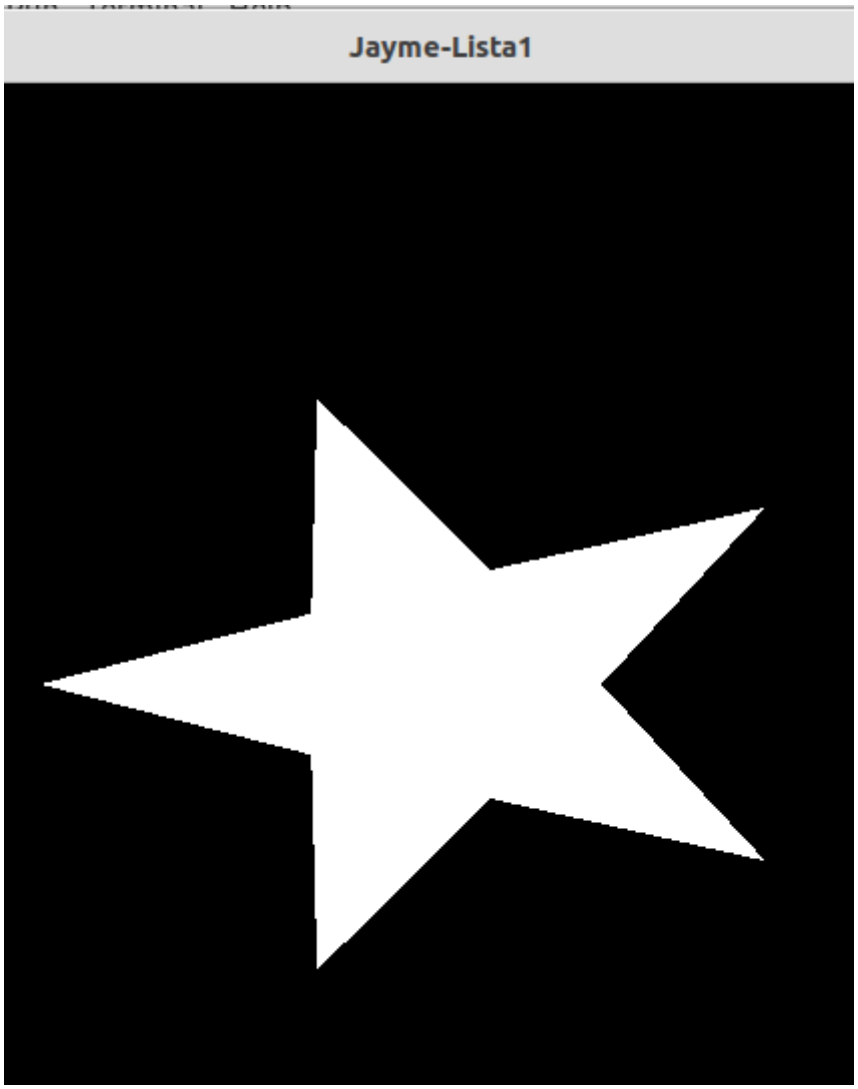
C-



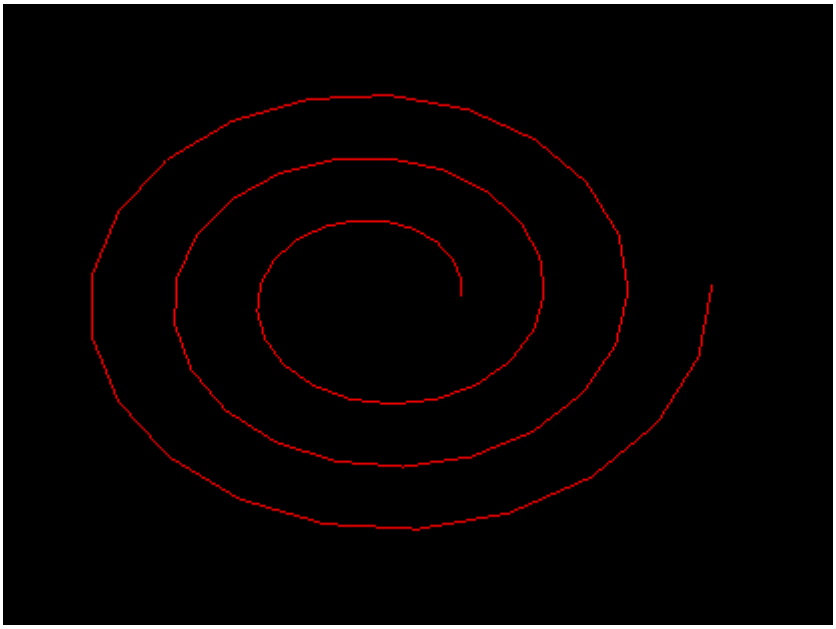
D-



E-



7-



8-
A-
VBO (Vertex Buffer Object): O VBO armazenaria os vértices do triângulo. Cada vértice é representado por um conjunto de coordenadas (x, y, z) e, opcionalmente, informações de cor para cada vértice (r, g, b). Portanto, o VBO seria organizado da seguinte forma:

x1, y1, z1, r1, g1, b1
x2, y2, z2, r2, g2, b2
x3, y3, z3, r3, g3, b3

VAO (Vertex Array Object): O VAO agrupa os buffers e define a configuração de atributos para o Vertex Shader. No VAO, é definido como os dados do VBO são interpretados. Para representar o triângulo, definimos os atributos de posição e cor. A configuração do VAO incluiria os seguintes passos:

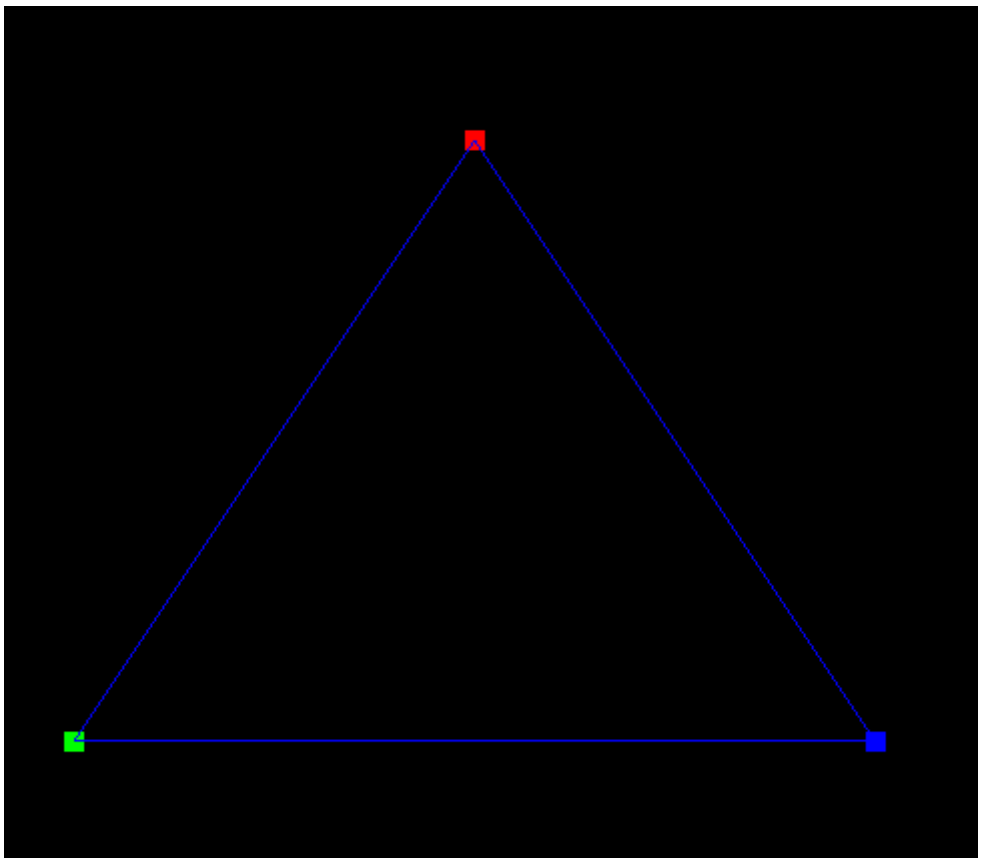
- Especificar como os dados de posição e cor são interpretados.
- Associar o VBO ao VAO.
- Definir a ordem dos vértices usando o EBO (Element Buffer Object) ou diretamente no VBO.

EBO (Element Buffer Object): Para economizar memória e evitar a duplicação de vértices, podemos usar o EBO para especificar a ordem dos vértices. Isso é útil quando temos vários triângulos compartilhando vértices em uma cena mais complexa. No entanto, para um único triângulo, o EBO é meio que opcional.

B-
No Vertex Shader, precisamos definir as variáveis de atributo para os dados de posição e cor e, em seguida, usar essas variáveis para processar os vértices. Por exemplo:

```
#version 330 core

layout(location = 0) in vec3 inPosition;  // Atributo de posição
layout(location = 1) in vec3 inColor;     // Atributo de cor

// Saída para o fragment shader
out vec3 fragColor;

void main() {
    // Processamento do vértice aqui
    gl_Position = vec4(inPosition, 1.0);

    // Defina a cor de saída
    fragColor = inColor;
}
```

Neste exemplo, usamos layout(location) para especificar as localizações dos atributos de posição e cor. No Vertex Shader, inPosition e inColor representam os atributos de posição e cor, respectivamente. gl_Position é usado para definir a posição final do vértice após o processamento no shader. fragColor é uma saída que passa a cor do vértice para o Fragment Shader.



9-