# section 5:Multiple Drawings

shows how to draw multiple Triangles and introduces the concept of Element Buffers.

## Review

Before explaining what is a Element Buffer lets recap the Previous concepts the Vertex Array Object (VAO) and the Vertex Buffer Object (VBO).

### ▼ Vertex Buffer Object (VBO)

is a specific type `Buffer (A Block of Memory)` that allows us to store Vertex data inside of it (such as position, color, normals, and texture coordinates) in the GPU's memory. This allows for efficient access during rendering.

#### Functions:

- **Data Storage**: VBOs hold the actual vertex data needed to construct graphical objects. This data is transferred from the CPU to the GPU, allowing the GPU to access it directly without needing to fetch it from system memory every frame.
- **Efficiency**: By storing vertex data in GPU memory, VBOs reduce the overhead associated with sending data from the CPU to the GPU repeatedly. This is particularly beneficial for complex scenes with many vertices.

### ▼ Vertex Array Object (VAO)

Is an Object which contains one or more Vertex Buffer Objects and is designed to store the information for a complete rendered object.  It essentially describes how vertex data is organized and how it should be interpreted.

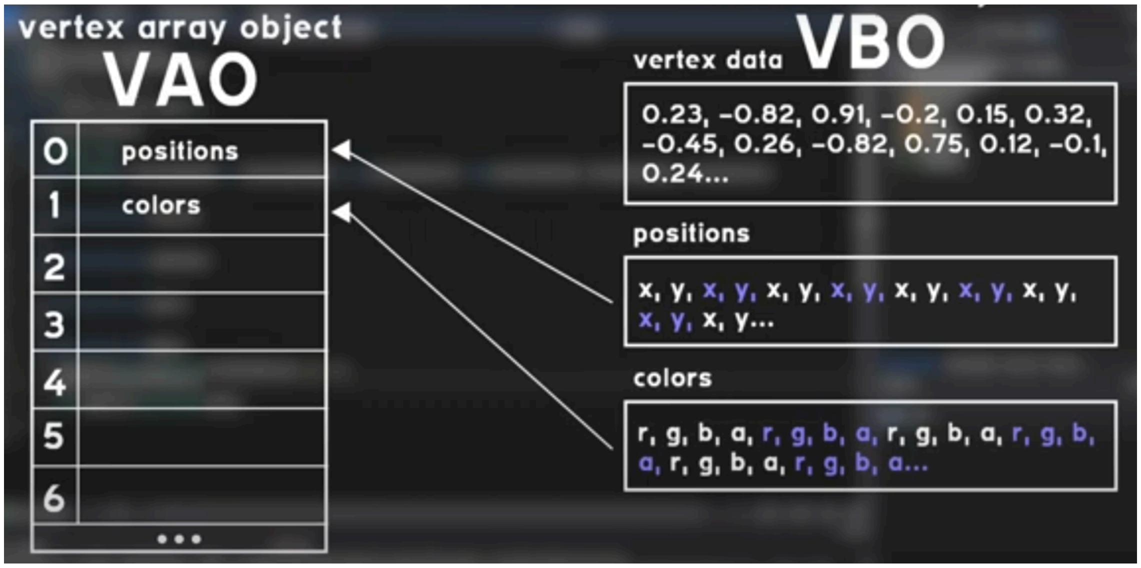
#### Functions:

- **State Management**: VAOs store all the necessary state information about vertex attributes, including which VBOs are associated with which attributes (like position, color, etc.) and how these attributes are formatted.
- **Simplified Binding**: When a VAO is bound, it automatically binds all associated VBOs and their configurations. This means that you can switch between different sets of vertex data with a single call, making rendering more efficient.

### TLDR

Essentially the VAO holds what data we are going to be sending the OpenGL pipeline. So, usually we need to send the pipeline data like vertices and colors we are drawing, but potentially more stuff as well like textures or normal maps.

Those attributes, like vertices, colors, textures, and more are stored in the Vertex Buffer Object (VBO).



---

## ▼ Element Buffer Object (EBO)

Also known as `Index Buffer objects` , are a type of `Buffer (A Block of Memory)` in OpenGL that allow us to `re-use` vertices to create multiple primitives out of them. (in our case for this notebook multiple Triangles)

They play a crucial role in optimizing memory usage and improving performance during the rendering process.

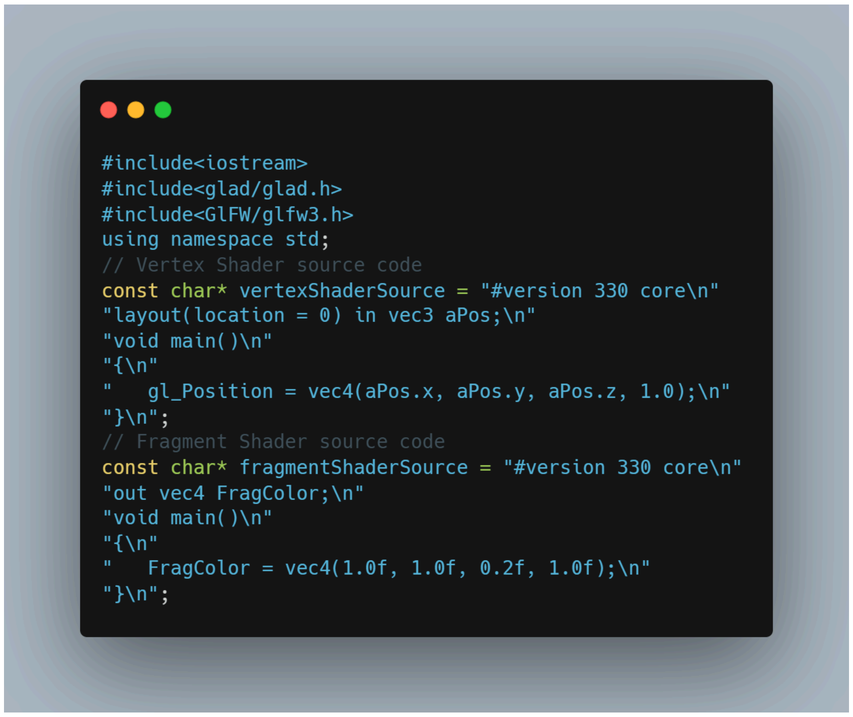
### Purpose of EBOs

**Indexed Rendering**: EBOs enable indexed rendering, which allows the use of a single set of vertex data to create multiple primitives (such as triangles) without duplicating vertex data. This is particularly beneficial for complex models where many vertices are shared among different faces.

**Memory Efficiency**: By using indices to reference vertices stored in Vertex Buffer Objects (VBOs), EBOs reduce the total amount of vertex data needed. For example, a rectangle can be defined using only four vertices instead of six by specifying the order in which these vertices are used to form triangles.

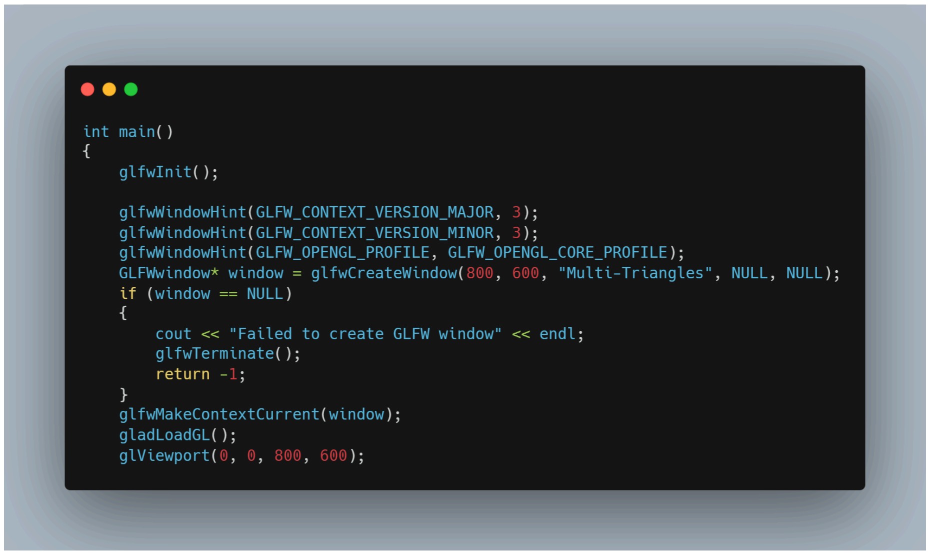
## Code Section

### ▼ Step 1 :Include libraries and shader sources



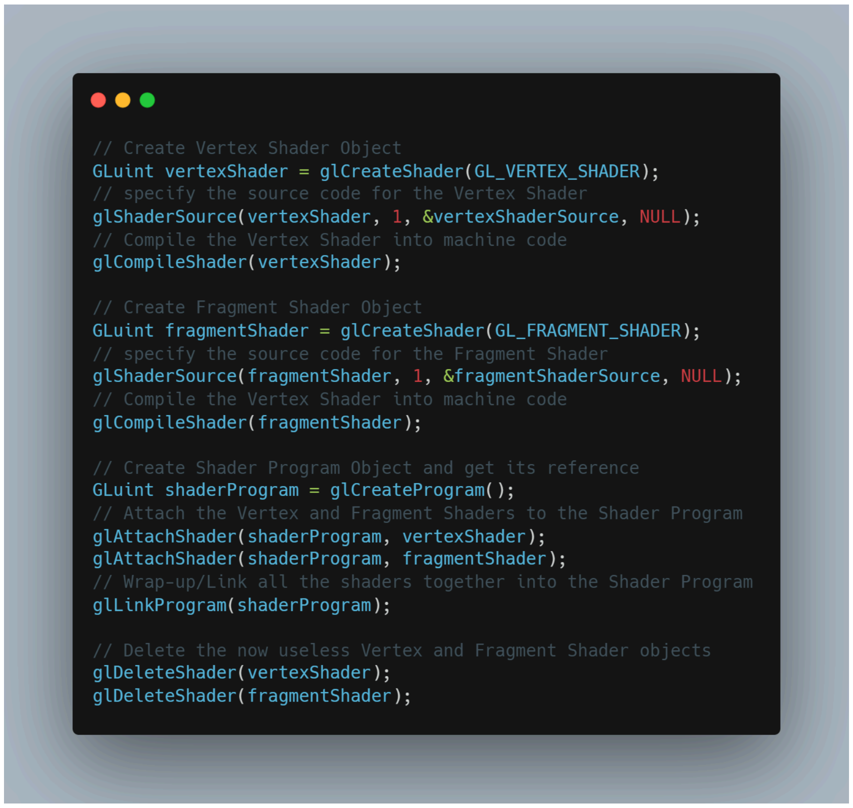
```
#include<iostream>
#include<glad/glad.h>
#include<GlFW/glfw3.h>
using namespace std;
// Vertex Shader source code
const char* vertexShaderSource = "#version 330 core\n"
"layout(location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\n";
// Fragment Shader source code
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 1.0f, 0.2f, 1.0f);\n"
"}\n";
```

### ▼ Step 2: Create Window , load OpenGL ,and define the viewport



```
int main()
{
    glfwInit();

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Multi-Triangles", NULL, NULL);
    if (window == NULL)
    {
        cout << "Failed to create GLFW window" << endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    gladLoadGL();
    glViewport(0, 0, 800, 600);
```

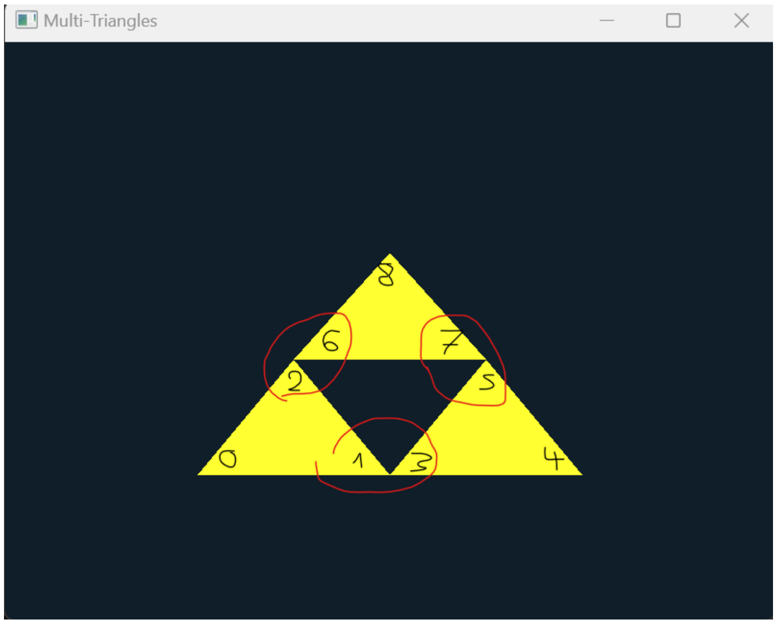
## ▼ Step 3 : create our Shaders and Shader Program



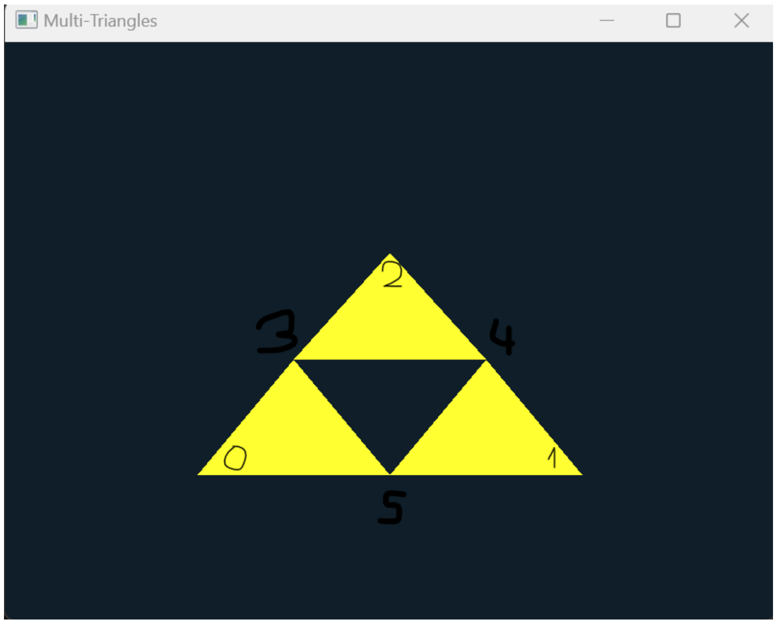
```
// Create Vertex Shader Object
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
// specify the source code for the Vertex Shader
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
// Compile the Vertex Shader into machine code
glCompileShader(vertexShader);

// Create Fragment Shader Object
GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
// specify the source code for the Fragment Shader
glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
// Compile the Vertex Shader into machine code
glCompileShader(fragmentShader);

// Create Shader Program Object and get its reference
GLuint shaderProgram = glCreateProgram();
// Attach the Vertex and Fragment Shaders to the Shader Program
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
// Wrap-up/Link all the shaders together into the Shader Program
glLinkProgram(shaderProgram);

// Delete the now useless Vertex and Fragment Shader objects
glDeleteShader(vertexShader);
glDeleteShader(fragmentShader);
```

## ▼ Step 4 : Define our Vertices and the Order of our Vertices (Indices)

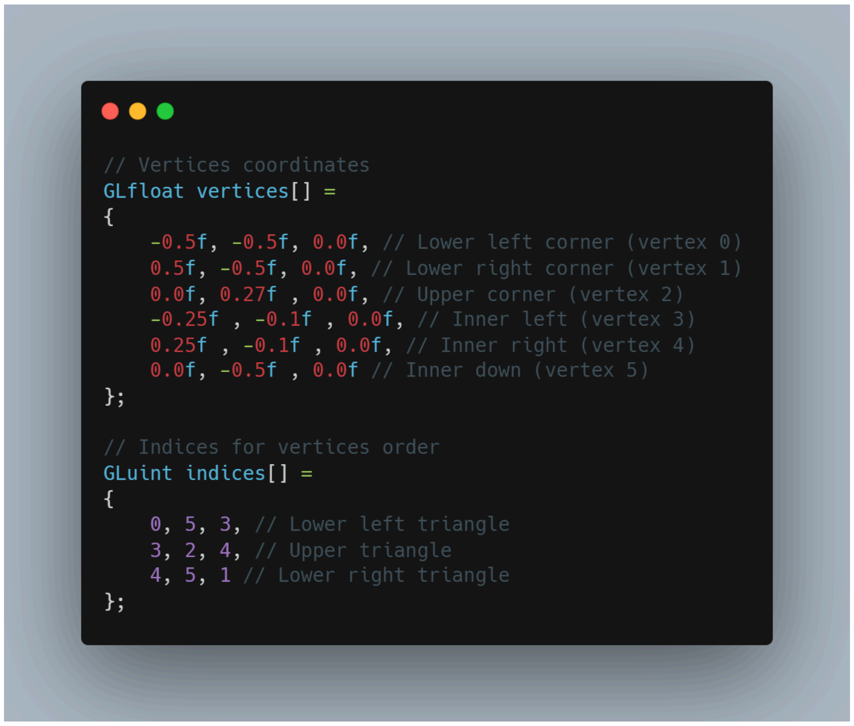
our end goal is to draw 3 triangles, so does this mean that we need 9 vertices ?



as you can see in the figure above if we used 9 vertices to draw our triangles some vertices were duplicated this is a waste of memory and there is an easier solution for it ,which is to use an element buffer to `re-use` our vertices in the order that we want .



this is much easier to code and consumes less memory space lets see how do we implement this in code.

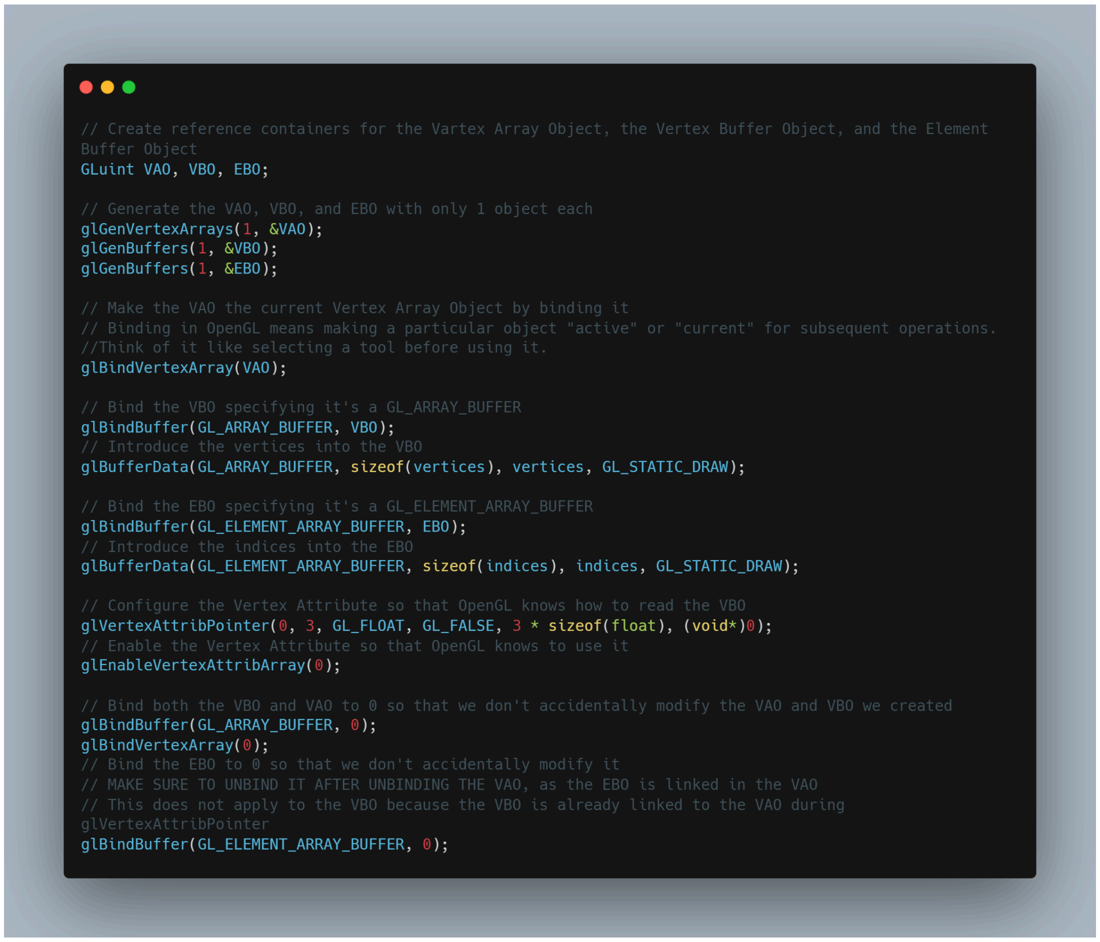


```
// Vertices coordinates
GLfloat vertices[] =
{
    -0.5f, -0.5f, 0.0f, // Lower left corner (vertex 0)
    0.5f, -0.5f, 0.0f, // Lower right corner (vertex 1)
    0.0f, 0.27f , 0.0f, // Upper corner (vertex 2)
    -0.25f , -0.1f , 0.0f, // Inner left (vertex 3)
    0.25f , -0.1f , 0.0f, // Inner right (vertex 4)
    0.0f, -0.5f , 0.0f // Inner down (vertex 5)
};

// Indices for vertices order
GLuint indices[] =
{
    0, 5, 3, // Lower left triangle
    3, 2, 4, // Upper triangle
    4, 5, 1 // Lower right triangle
};
```

we defined a list called `vertices` of `GLfloat` datatype to store our vertices coordinates like last section but this time we also defined a second list called `indices` of `GLuint` datatype, this list will hold the order of the vertices and determine how they should be connected to form triangles, allowing us to `re-use` vertices.

## ▼ Step 5 : VAO, VBO, and EBO

Now that we defined our vertices and indices, we need to create the VAO (Vertex Array Object), VBO (Vertex Buffer Object), and EBO (Element Buffer Object) to store and manage our vertex and index data on the GPU.



```
// Create reference containers for the Vartex Array Object, the Vertex Buffer Object, and the Element
Buffer Object
GLuint VAO, VBO, EBO;

// Generate the VAO, VBO, and EBO with only 1 object each
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glGenBuffers(1, &EBO);

// Make the VAO the current Vertex Array Object by binding it
// Binding in OpenGL means making a particular object "active" or "current" for subsequent operations.
//Think of it like selecting a tool before using it.
glBindVertexArray(VAO);

// Bind the VBO specifying it's a GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, VBO);
// Introduce the vertices into the VBO
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// Bind the EBO specifying it's a GL_ELEMENT_ARRAY_BUFFER
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
// Introduce the indices into the EBO
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

// Configure the Vertex Attribute so that OpenGL knows how to read the VBO
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
// Enable the Vertex Attribute so that OpenGL knows to use it
glEnableVertexAttribArray(0);

// Bind both the VBO and VAO to 0 so that we don't accidentally modify the VAO and VBO we created
glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
// Bind the EBO to 0 so that we don't accidentally modify it
// MAKE SURE TO UNBIND IT AFTER UNBINDING THE VAO, as the EBO is linked in the VAO
// This does not apply to the VBO because the VBO is already linked to the VAO during
glVertexAttribPointer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

When you call `glBindVertexArray(VAO)`, you're essentially telling OpenGL "this is the VAO I want to work with right now." Any vertex attribute configurations or buffer associations you make after this point will be stored in this bound VAO.

---

1. `glBindBuffer(GL_ARRAY_BUFFER, VBO)` :

- First parameter `GL_ARRAY_BUFFER` : Tells OpenGL this buffer will contain vertex attributes (like positions, colors, etc.)
- Second parameter `VBO` : The ID of your buffer object that you created earlier
- This command is like saying "Hey OpenGL, I want to work with this specific VBO now"

1. `glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW)`:

- First parameter `GL_ARRAY_BUFFER`: The type of buffer we're putting data into (must match the bound buffer type)
- Second parameter `sizeof(vertices)`: The size in bytes of your vertex data
- Third parameter `vertices`: Pointer to your actual vertex data array
- Fourth parameter `GL_STATIC_DRAW`: A usage hint telling OpenGL how we plan to use this data:
    - `GL_STATIC_DRAW`: Data will be set once and used many times (best for objects that don't move)
    - (There are also other options like `GL_DYNAMIC_DRAW` for data that changes often)

It's like:

1. First line: "I'm selecting this specific container (VBO) to put vertex data in"
2. Second line: "Now fill this container with this much data, here's the data, and I plan to use it this way"

---

The code for setting up an EBO follows the exact same pattern as a VBO, just with different parameters:

- `GL_ARRAY_BUFFER` becomes `GL_ELEMENT_ARRAY_BUFFER`
- `vertices` becomes `indices`
- The size is `sizeof(indices)` instead of `sizeof(vertices)`

---

`glVertexAttribPointer` and `glEnableVertexAttribArray` these two commands tell OpenGL how to interpret the vertex data:

1. `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0)`:

- First parameter `0`: The location of the vertex attribute (position = 0 in this case)
- Second parameter `3`: Number of components per vertex (x, y, z = 3 values)
- Third parameter `GL_FLOAT`: The data type of each component
- Fourth parameter `GL_FALSE`: Whether to normalize the data
- Fifth parameter `3 * sizeof(float)`: The stride (distance between consecutive vertices)
- Sixth parameter `(void*)0`: The offset where this attribute begins in the buffer (VBO)
    - `(void*)0` tells OpenGL to start reading from 0 (the beginning) of the currently bound VBO

Think of it like: "Hey OpenGL, for attribute 0, read 3 floats at a time, don't normalize them, each vertex takes up space of 3 floats, and start reading from the beginning"

1. `glEnableVertexAttribArray(0)`:
    - Simply activates the vertex attribute at location 0
    - Like turning on a switch to say "yes, use this attribute"

---

the final step is to Unbind VBO,VAO, and EBO in this order so that we don't accidently modify it later to do this we will use the following :

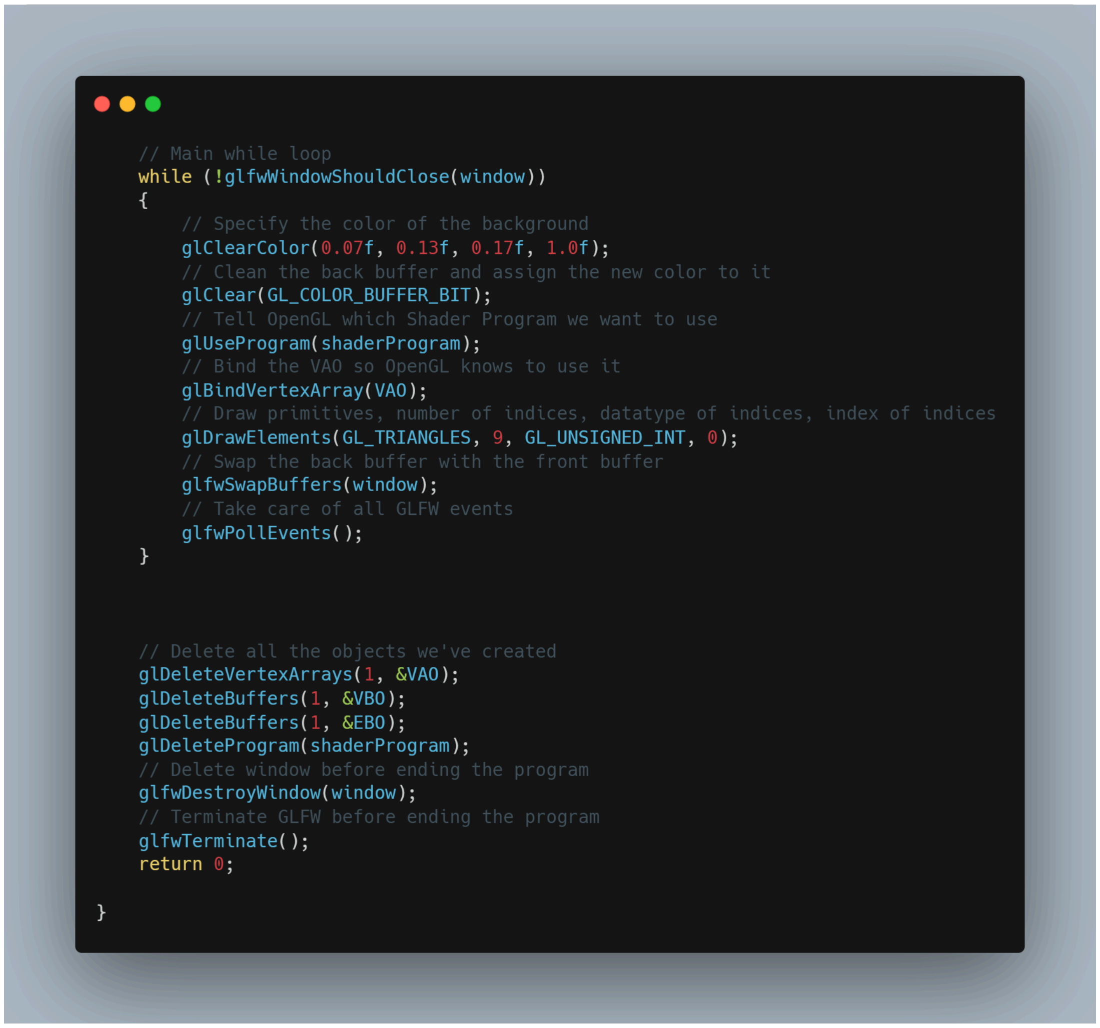
1. `glBindBuffer(GL_ARRAY_BUFFER, 0)` :
   - Unbinds the VBO (binding to 0 means "no buffer")
   - Like saying "I'm done working with this VBO"
   - Order doesn't matter for VBO because it's already linked to VAO during `glVertexAttribPointer`
2. `glBindVertexArray(0)` :
   - Unbinds the VAO
   - Like saying "I'm done configuring this VAO"
3. `glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0)` :
   - Unbinds the EBO
   - IMPORTANT: Must be done AFTER unbinding VAO
   - Why? Because EBO is part of VAO state. If you unbind EBO first, it would also unbind it from the VAO

Think of it like:

- VAO is like a settings package that remembers both VBO and EBO configurations
- VBO settings are "photographed" by VAO during `glVertexAttribPointer` , so unbinding VBO doesn't affect VAO
- EBO is actively linked to VAO, so you must keep EBO bound until after VAO is unbound

## ▼ Step 6: Creating a loop

create a loop to continuously render the scene



```
    // Main while loop
    while (!glfwWindowShouldClose(window))
    {
        // Specify the color of the background
        glClearColor(0.07f, 0.13f, 0.17f, 1.0f);
        // Clean the back buffer and assign the new color to it
        glClear(GL_COLOR_BUFFER_BIT);
        // Tell OpenGL which Shader Program we want to use
        glUseProgram(shaderProgram);
        // Bind the VAO so OpenGL knows to use it
        glBindVertexArray(VAO);
        // Draw primitives, number of indices, datatype of indices, index of indices
        glDrawElements(GL_TRIANGLES, 9, GL_UNSIGNED_INT, 0);
        // Swap the back buffer with the front buffer
        glfwSwapBuffers(window);
        // Take care of all GLFW events
        glfwPollEvents();
    }



    // Delete all the objects we've created
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
    glDeleteProgram(shaderProgram);
    // Delete window before ending the program
    glfwDestroyWindow(window);
    // Terminate GLFW before ending the program
    glfwTerminate();
    return 0;

}
```

however this time we use `glDrawElements()` instead `glDrawArrays()` :

because `glDrawElements()` allows us to :

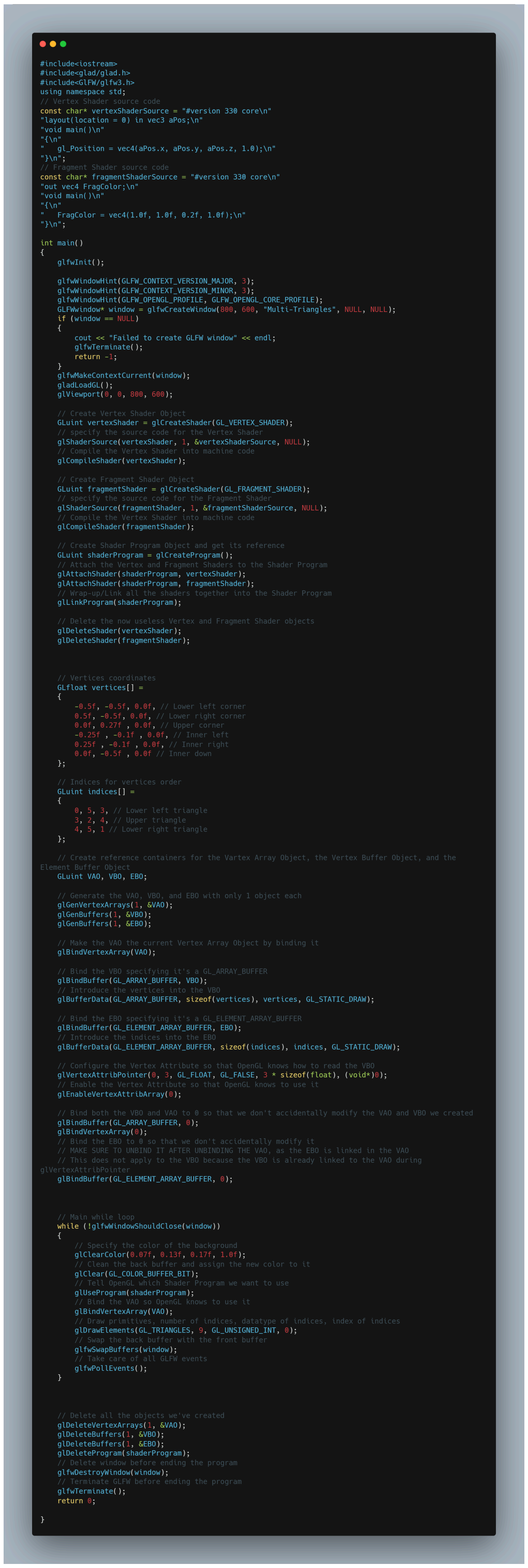
specify which vertices to use for drawing by providing an index buffer. This means you can reuse vertices without duplicating their data in the vertex buffer.

## Parameters :

1. **mode**:
    - This parameter specifies the type of primitives to render. It can take various symbolic constants, such as:
        - `GL_POINTS`
        - `GL_LINES`
        - `GL_TRIANGLES`
        - `GL_TRIANGLE_STRIP`
        - `GL_TRIANGLE_FAN`

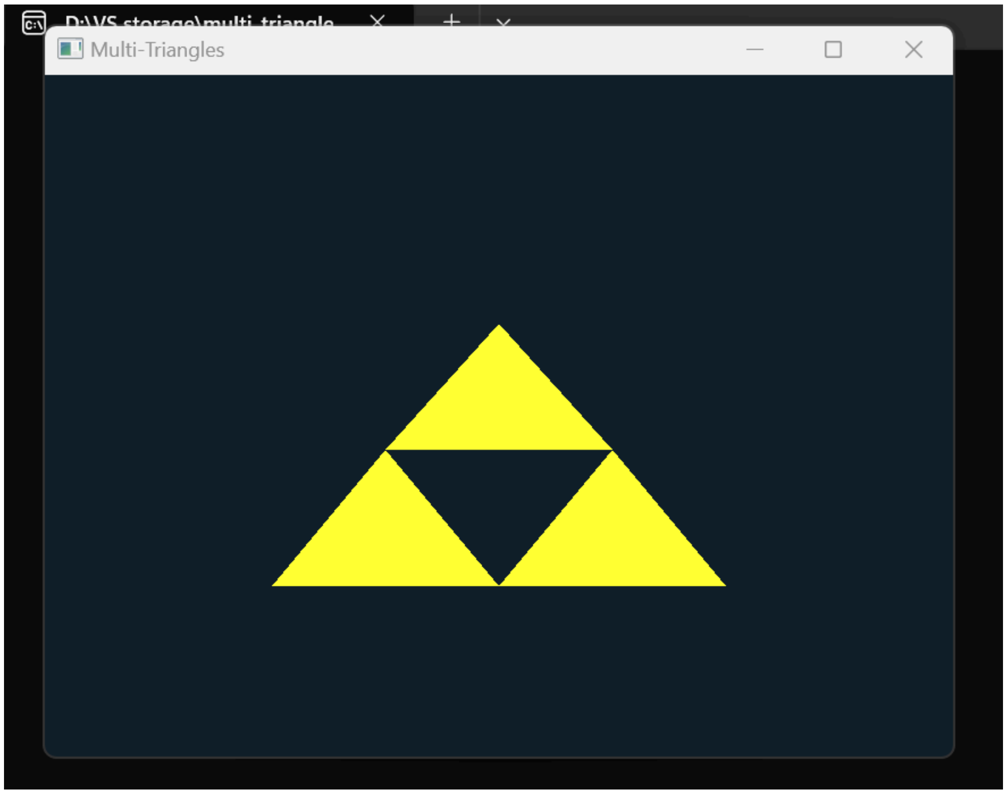
    - Additional modes like `GL_LINE_STRIP_ADJACENCY` and `GL_TRIANGLES_ADJACENCY` are available in OpenGL 3.2 and later versions.
  - The choice of mode determines how the vertices are connected to form the desired geometric shape.
2. **count**:
  - This parameter indicates the number of elements (indices) to be rendered. It specifies how many vertices will be processed based on the indices provided in the element array.
3. **type**:
  - This parameter defines the data type of the indices in the element array. It must be one of the following:
    - `GL_UNSIGNED_BYTE`
    - `GL_UNSIGNED_SHORT`
    - `GL_UNSIGNED_INT`
  - This specifies how to interpret the values in the indices array when accessing vertex data.
4. **indices**:
  - This parameter specifies an offset or pointer to the first index in the array that contains the indices for the vertices to be drawn. If using an Element Buffer Object (EBO), this value is typically set to `0`, indicating that the indices are stored in the currently bound buffer for `GL_ELEMENT_ARRAY_BUFFER`. The indices will be used to reference vertices in your vertex array

## ▼ Full Code & Output



```
#include<iostream>
#include<glad/glad.h>
#include<GLFW/glfw3.h>
using namespace std;
// Vertex Shader source code
const char* vertexShaderSource = "#version 330 core\n"
"layout(location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\n";
// Fragment Shader source code
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 1.0f, 0.2f, 1.0f);\n"
"}\n";

int main()
{
    glfwInit();

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    GLFWwindow* window = glfwCreateWindow(800, 600, "Multi-Triangles", NULL, NULL);
    if (window == NULL)
    {
        cout << "Failed to create GLFW window" << endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    gladLoadGL();
    glViewport(0, 0, 800, 600);

    // Create Vertex Shader Object
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    // specify the source code for the Vertex Shader
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(vertexShader);

    // Create Fragment Shader Object
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    // specify the source code for the Fragment Shader
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(fragmentShader);

    // Create Shader Program Object and get its reference
    GLuint shaderProgram = glCreateProgram();
    // Attach the Vertex and Fragment Shaders to the Shader Program
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    // Wrap-up/Link all the shaders together into the Shader Program
    glLinkProgram(shaderProgram);

    // Delete the now useless Vertex and Fragment Shader objects
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);



    // Vertices coordinates
    GLfloat vertices[] =
    {
        -0.5f, -0.5f, 0.0f, // Lower left corner
        0.5f, -0.5f, 0.0f, // Lower right corner
        0.0f, 0.27f , 0.0f, // Upper corner
        -0.25f , -0.1f , 0.0f, // Inner left
        0.25f , -0.1f , 0.0f, // Inner right
        0.0f, -0.5f , 0.0f // Inner down
    };

    // Indices for vertices order
    GLuint indices[] =
    {
        0, 5, 3, // Lower left triangle
        3, 2, 4, // Upper triangle
        4, 5, 1 // Lower right triangle
    };

    // Create reference containers for the Vartex Array Object, the Vertex Buffer Object, and the
Element Buffer Object
    GLuint VAO, VBO, EBO;

    // Generate the VAO, VBO, and EBO with only 1 object each
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
    glGenBuffers(1, &EBO);

    // Make the VAO the current Vertex Array Object by binding it
    glBindVertexArray(VAO);

    // Bind the VBO specifying it's a GL_ARRAY_BUFFER
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // Introduce the vertices into the VBO
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Bind the EBO specifying it's a GL_ELEMENT_ARRAY_BUFFER
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    // Introduce the indices into the EBO
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW);

    // Configure the Vertex Attribute so that OpenGL knows how to read the VBO
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
    // Enable the Vertex Attribute so that OpenGL knows to use it
    glEnableVertexAttribArray(0);

    // Bind both the VBO and VAO to 0 so that we don't accidentally modify the VAO and VBO we created
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
    // Bind the EBO to 0 so that we don't accidentally modify it
    // MAKE SURE TO UNBIND IT AFTER UNBINDING THE VAO, as the EBO is linked in the VAO
    // This does not apply to the VBO because the VBO is already linked to the VAO during
glVertexAttribPointer
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);



    // Main while loop
    while (!glfwWindowShouldClose(window))
    {
        // Specify the color of the background
        glClearColor(0.07f, 0.13f, 0.17f, 1.0f);
        // Clean the back buffer and assign the new color to it
        glClear(GL_COLOR_BUFFER_BIT);
        // Tell OpenGL which Shader Program we want to use
        glUseProgram(shaderProgram);
        // Bind the VAO so OpenGL knows to use it
        glBindVertexArray(VAO);
        // Draw primitives, number of indices, datatype of indices, index of indices
        glDrawElements(GL_TRIANGLES, 9, GL_UNSIGNED_INT, 0);
        // Swap the back buffer with the front buffer
        glfwSwapBuffers(window);
        // Take care of all GLFW events
        glfwPollEvents();
    }



    // Delete all the objects we've created
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &EBO);
    glDeleteProgram(shaderProgram);
    // Delete window before ending the program
    glfwDestroyWindow(window);
    // Terminate GLFW before ending the program
    glfwTerminate();
    return 0;

}
```

**Output:**


D:\VS storage\multi_triangle
Multi-Triangles