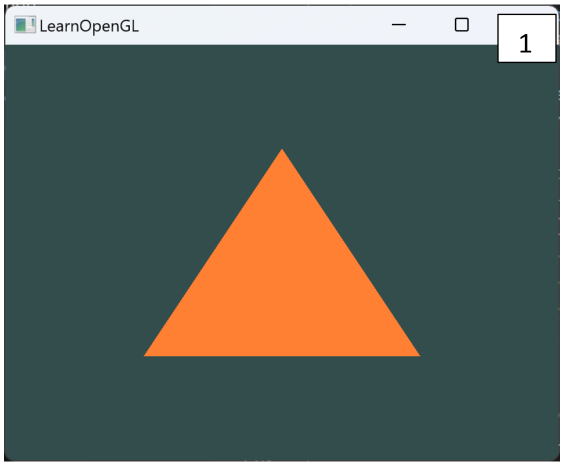
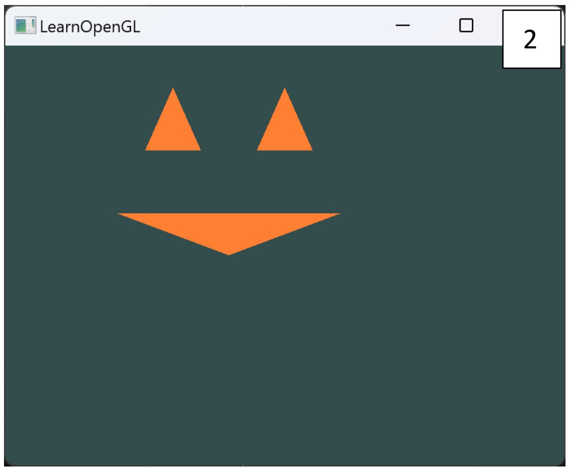
ICE 3.1: Preliminaries: First Shapes (VBOs, VAOs and C++ Classes)

This ICE familiarizes you with using vertex buffer objects (VBOs) and vertex array objects (VAOs) to create basic 2D shapes.
This ICE Assumes you have completed the Hello Triangle tutorial at www.learnopengl.com.





1. Start with the Hello Triangle solution from the reading (available in the reading before Element Buffer Objects here). Create a new folder with the .vscode and glad.c files from last lesson as needed. Ensure that the code runs. Remember that only one main function call may exist in our project folder.
2. **Vertex Array Objects:** Vertex Array Objects are used to store attribute pointers (or vertex data attribute descriptions). Here we will set up a structure to help manage these for us.
    a. Create a new header file, vertex_attribute.hpp.
    b. **Inside vertex_attribute.hpp:** Add an important preprocessor directive to the file (this prevents the items in this code from being defined multiple times if included multiple times).
    #ifndef VERTEX_ATTRIBUTE_HPP
    #define VERTEX_ATTRIBUTE_HPP
        add all the include statements from hello_triangle.cpp (delete from the cpp)
        …more will go here shortly…
    #endif //VERTEX_ATTRIBUTE_HPP
    c. Define an Named Structure, called AttributePointer (see C++ Structures (struct) (w3schools.com) under the title "Named Structure") with the following data members:
        i. An integer representing the number of values per vertex (e.g., 3 if just x, y, and z values).
        ii. An integer representing the type of data (e.g., GL_FLOAT—which is the most common).
        iii. A Boolean value representing whether or not the data should be normalized (typically false).
        iv. An integer representing the number of BYTES in the stride for this attribute.
        v. An integer representing the number of BYTES that must be offset before you get to the first value in this attribute.

d. Define a VAO structure with the following data members:
    i. An unsigned integer representing the VAO identity
    ii. An std::vector<AttributePointer> used for storing any number of attributes pointers.
    iii. You will need to include <vector> for part 2.d.ii.
e. Create a forward declaration for a function called BuildAttribute that returns an AttributePointer structure and accepts as input values for each of the data members in part 2.c above.
f. Create a forward declaration for a function called BindVAO that accepts a VAO structure, a VBO identifier (unsigned int), and an optional buffer type (default is GL_ARRAY_BUFFER) and returns nothing.
    i. At this point you will need to #include <glad/glad.h> if you have not yet done so.
g. Create **vertex_attribute.cpp**, include vertex_attribute.hpp (use #include "vertex_attribute.hpp" – the quotes indicate a local file in your project), and define the body of the functions you defined in part 3e and 3f.
    i. BuildAttribute just needs to create and return an AttributePointer Struct.
    ii. BindVAO needs to do the following:
        1. Bind the VAO passed as input (using its identity).
        2. Bind the buffer object (VBO) passed as input (using its identity)
        3. Iterate through the attribute pointers in the VAO's vector of attributes and do the following:
            a. Call glVertexAttribPointer with the following parameter arguments:
                i. Index is the iteration of your for loop (e.g., i)
                ii. Size is the number of values per vertex.
                iii. Data type is the data type in the given VAO Struct.
                iv. Stride is the stride bytes from the VAO Struct.
                v. For the last parameter, offset pointer, use (void*)(intptr_t)offset_bytes.
                vi. Necessary to ensure the pointer is the expected size (avoid a warning).
            b. Enable the attribute pointer (using the index of the attribute)
h. Test it out! Refactor your code in hellow_triangle.cpp so that it uses the new functions to draw the triangle. This is an intermediate step before creating a basic shape class.
    i. Add your vertex_attribute.cpp file to your tasks.json arguments.
    ii. Include your header file at the top of hello_triangle.cpp (#include "vertex_attribute.hpp").
    iii. Create the VAOStruct, named vao, and generate the vertex array object as before, but with the VAOStruct's identity field as input (e.g., glGenVertexArrays(1, &(vao.id)).
    iv. Create an attribute pointer for the position attribute (see glVertexAttribPointer noting that the first value is just the index of the attribute pointer in our vector of attributes).

v. Use the pushback vector member function to push the attribute pointer into the VAOStruct's vector.

vi. In the render loop (the while loop in the program), use the BindVAO function to bind the VAO and the triangle VBO.

3. **BasicShape Class:** Create a new class, called BasicShape, which will store information related to a vertex buffer object and vertex array object used to draw a shape. Done well, objects created from this class will be able to draw themselves, manage the deletion of VBOs and VAOs when they go out of scope, and, in the future, rotate, scale, and move on their own. See a class tutorial at C++ OOP (Object-Oriented Programming) (w3schools.com).
    a. Create a file, called basic_shape.hpp, and create similar preprocessor directives as in step 2.b, but using BASIC_SHAPE_HPP instead.
    b. Create a new class BasicShape, using the format found in C++ Classes and Objects).
    c. Declare the following (forward declarations for the functions only):
        i. Use the keyword protected (instead of public).
        ii. Protected data members for the following (see C++ Access Specifiers (w3schools.com) – for a discussion on private, public, and protected access specifiers):
            1. A VAO structure for the VAO used to draw this shape (include vertex_attribute.hpp)
            2. Unsigned integer for a VBO identifier, set to 0 initially.
            3. An integer representing the number of vertices in the VBO, set to 0 initially.
            4. A GLuint representing the primitive to use (default is GL_TRIANGLES)
        iii. Under these data members, put public: in your class to indicate that the functions below are available to use in other classes and functions).
        iv. Add a public constructor that accepts no arguments (see Constructors in the class tutorial).
        v. Add a public member function called Initialize that accepts the following arguments and returns nothing:
            1. A VAO structure
            2. A pointer to an array of float values containing vertex data.
            3. An integer representing the number of BYTES in the array.
            4. An integer representing the number of vertices in the array.
            5. An OPTIONAL GLuint argument representing the primitive to draw with a default value of GL_TRIANGLEs.
        vi. A public member function, called Draw, that accepts a single argument for an unsigned integer representing the identifier of a shader program to use.
        vii. A public member function called DeallocateShape that accepts no arguments and returns no value.
    d. Create a file called basic_shape.cpp that includes basic_shape.hpp and is used to define the member functions declared in part 3.c.
        i. The constructor sets the values to the defaults in part 3.c

ii. The initialize member function:
    1. ASSUMES that the VAO was already generated (you will still do this in the main program).
    2. Sets, the VAO, number of VBO vertices, and primitive to the corresponding input values. (use this→vao = vao) if the name of your VAO data member is vao and the name of your vao input is vao).
    3. Generates, binds, and loads data into the VBO (setting the VBO unsigned int value to the number associated with the generated object).
        a. Use glGenBuffers(GL_ARRAY_BUFFER,&(this→vbo)) if your VBO data member is named vbo)

iii. Define the draw member function that:
    1. Uses the shader program provided as an argument
    2. Binds the VAO
    3. Draws the shape
    4. Unbinds the VAO (glBindVertexArray(0))

iv. Define the DeallocateShape to delete the VBO and VAO;

4. Use the new class to draw the triangle! In hello_triangle.cpp:
    a. Include the new class header file.
    b. Add the .cpp file to your tasks.json args.
    c. Define a BasicShape variable in main.
    d. Initialize the Shape object with the VAO Structure created in step 2.h.
    e. Draw the object inside the render loop
5. **MORE SHAPES!** Now that it is easy to create, initialize, and draw new shapes. Create a second set of vertices and place a triangle in another location in the scene (remember coordinates must be in the range [-1,1] for x and y values. Leave z as 0.0 for all points).
6. **OPTIONAL (ENCOURAGED) REFACTOR PRACTICE:** Practice searching for ways to make your code easier to read and manipulate. Start by refactoring hello_triangle.cpp to place the initialization code in a separate header file called environment_setup.hpp.
    a. At the top of the new file, add preprocessor directives below (good practice):
        #ifndef ENVIRONMENT_SETUP_HPP
        #define ENVIRONMENT_SETUP_HPP
        add all the include statements from hello_triangle.cpp (delete from the cpp)
        ….content of file …
        #endif \\ENVIRONMENT_SETUP_HPP
    b. Inside environment_setup.hpp, create a function called `InitializeEnvironment` that returns a pointer to a `GLFWwindow`.
    c. You will need to have parameters for the screen width, screen height, and window title.
    d. Implement `InitializeEnvironment.` Copy and paste all the code necessary for initializing and configuring GLFW, creating a window, setting the current context and setting the window resize callback. The function should also initialize GLAD. Return the created window (type `GLFWwindow*`). Note the following:
        i. glfwCreateWindow only accepts C style strings for the title…use the string member function c_str() to get the C style string.

ii. Since the original checks for window creation and GLAD initialization should return NULL (instead of -1).
iii. InitializeEnvironment needs to return the created window at the end.