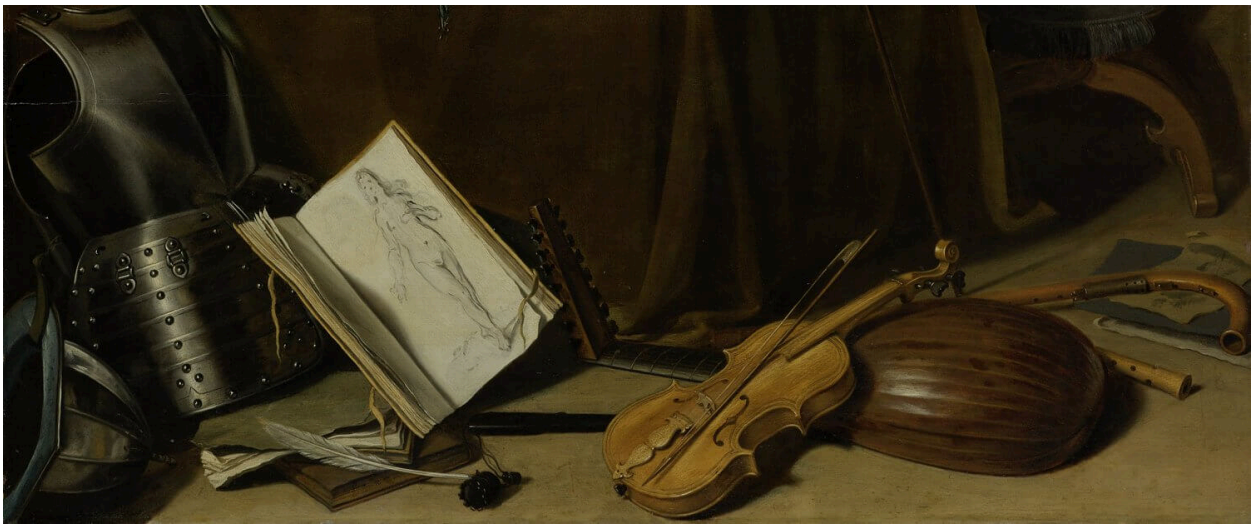

Organizing OpenGL Code with Classes

After implementing an index buffer, it's time to organize the codebase by extracting shaders into separate files and creating reusable classes for better code structure and maintainability.

Organization Steps:

1. Extract Shaders to External Files
2. Create Shader Class
3. Create VBO (Vertex Buffer Object) Class
4. Create EBO (Element Buffer Object) Class
5. Create VAO (Vertex Array Object) Class
6. Refactor Main Function

---

1. Extract Shaders to External Files

Goal: Move shader source code from C++ strings to dedicated text files.

Process:

- Open **Solution Explorer**
- Create a folder called `Shaders` in **Resource Files**
- Add new items: **Utility → Text File**

Create Vertex Shader File:

- Name: `default.vert`
- Copy the vertex shader source code

- **Remove:**
  - Variable declarations
  - String quotes (`"`)
  - Newline characters (`\n`)

Vertex Shader Code: `default.vert`

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
  gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

Fragment Shader Code: `default.frag`

```
#version 330 core
out vec4 FragColor;

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

⚙️ **Note:** This separation improves readability and makes shader editing easier without recompiling C++ code.

---

2. Create Shader Class

Purpose: Wrap OpenGL shader program into a clean, reusable class with file reading capabilities.

Header File: `shaderClass.h`

```
#ifndef SHADER_CLASS_H
#define SHADER_CLASS_H

#include<glad/glad.h>
#include<string>
#include<fstream>
#include<sstream>
#include<iostream>
#include<cerrno>
```

```
// Function to read shader text files
std::string get_file_contents(const char* filename);

class Shader
{
public:
    // Reference ID of the Shader Program
    GLuint ID;

    // Constructor that build the Shader Program from 2 different shaders
    Shader(const char* vertexFile, const char* fragmentFile);

    // Activates the Shader Program
    void Activate();

    // Deletes the Shader Program
    void Delete();
};

#endif
```

💡 **Header Guards:** `#ifndef`, `#define`, `#endif` prevent the file from being included twice, avoiding variable clashes.

Source File: `shaderClass.cpp`

```
#include"shaderClass.h"

// Reads a text file and outputs a string with everything in the text file
std::string get_file_contents(const char* filename)
{
    std::ifstream in(filename, std::ios::binary);
    if (in)
    {
        std::string contents;
        in.seekg(0, std::ios::end);
        contents.resize(in.tellg());
        in.seekg(0, std::ios::beg);
        in.read(&contents[0], contents.size());
        in.close();
        return(contents);
    }
    throw(errno);
}
```

```
// Constructor that build the Shader Program from 2 different shaders
Shader::Shader(const char* vertexFile, const char* fragmentFile)
{
    // Read vertexFile and fragmentFile and store the strings
    std::string vertexCode = get_file_contents(vertexFile);
    std::string fragmentCode = get_file_contents(fragmentFile);

    // Convert the shader source strings into character arrays
    const char* vertexSource = vertexCode.c_str();
    const char* fragmentSource = fragmentCode.c_str();

    // Create Vertex Shader Object and get its reference
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    // Attach Vertex Shader source to the Vertex Shader Object
    glShaderSource(vertexShader, 1, &vertexSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(vertexShader);

    // Create Fragment Shader Object and get its reference
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    // Attach Fragment Shader source to the Fragment Shader Object
    glShaderSource(fragmentShader, 1, &fragmentSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(fragmentShader);

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

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

// Activates the Shader Program
void Shader::Activate()
{
    glUseProgram(ID);
}
```

```
// Deletes the Shader Program
void Shader::Delete()
{
    glDeleteProgram(ID);
}
```

Key Implementation Details:

- `get_file_contents()` reads shader files and returns content as string
- Constructor reads files, compiles shaders, links program
- Shaders are deleted after linking (no longer needed)

---

3. Create VBO (Vertex Buffer Object) Class

Purpose: Encapsulate vertex buffer creation and management.

Header File: `VBO.h`

```
#ifndef VBO_CLASS_H
#define VBO_CLASS_H

#include<glad/glad.h>

class VBO
{
public:
    // Reference ID of the Vertex Buffer Object
    GLuint ID;

    // Constructor that generates a Vertex Buffer Object and links it to vertices
    VBO(GLfloat* vertices, GLsizeiptr size);

    // Binds the VBO
    void Bind();
    // Unbinds the VBO
    void Unbind();
    // Deletes the VBO
    void Delete();
};

#endif
```

⚙️ **Data Type:** `GLsizeiptr` is OpenGL's type for sizes in bytes.

Source File: `VBO.cpp`

```
#include"VBO.h"

// Constructor that generates a Vertex Buffer Object and links it to vertices
VBO::VBO(GLfloat* vertices, GLsizeiptr size)
{
    glGenBuffers(1, &ID);
    glBindBuffer(GL_ARRAY_BUFFER, ID);
    glBufferData(GL_ARRAY_BUFFER, size, vertices, GL_STATIC_DRAW);
}

// Binds the VBO
void VBO::Bind()
{
    glBindBuffer(GL_ARRAY_BUFFER, ID);
}

// Unbinds the VBO
void VBO::Unbind()
{
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

// Deletes the VBO
void VBO::Delete()
{
    glDeleteBuffers(1, &ID);
}
```

VBO Functions:

- **Constructor:** Generates buffer, binds it, and uploads vertex data
- **Bind:** Makes this VBO the active array buffer
- **Unbind:** Unbinds any array buffer (ID = 0)
- **Delete:** Frees GPU memory

---

4. Create EBO (Element Buffer Object) Class

Purpose: Manage index buffer for indexed drawing (reduces vertex duplication).

Header File: `EBO.h`

```
#ifndef EBO_CLASS_H
#define EBO_CLASS_H
```

```
#include<glad/glad.h>

class EBO
{
public:
    // ID reference of Elements Buffer Object
    GLuint ID;

    // Constructor that generates a Elements Buffer Object and links it to indices
    EBO(GLuint* indices, GLsizeiptr size);

    // Binds the EBO
    void Bind();
    // Unbinds the EBO
    void Unbind();
    // Deletes the EBO
    void Delete();
};

#endif
```

Source File: EBO.cpp

```
#include"EBO.h"

// Constructor that generates a Elements Buffer Object and links it to indices
EBO::EBO(GLuint* indices, GLsizeiptr size)
{
    glGenBuffers(1, &ID);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ID);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, size, indices, GL_STATIC_DRAW);
}

// Binds the EBO
void EBO::Bind()
{
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ID);
}

// Unbinds the EBO
void EBO::Unbind()
{
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
```

```
}

// Deletes the EBO
void EBO::Delete()
{
    glDeleteBuffers(1, &ID);
}
```

⛔ **Critical Difference:** EBO uses `GL_ELEMENT_ARRAY_BUFFER` instead of `GL_ARRAY_BUFFER`

5. Create VAO (Vertex Array Object) Class

Purpose: Manage vertex array configuration and link with VBO attributes.

Header File: `VAO.h`

```
#ifndef VAO_CLASS_H
#define VAO_CLASS_H

#include<glad/glad.h>
#include"VBO.h"

class VAO
{
public:
    // ID reference for the Vertex Array Object
    GLuint ID;

    // Constructor that generates a VAO ID
    VAO();

    // Links a VBO to the VAO using a certain layout
    void LinkVBO(VBO& VBO, GLuint layout);

    // Binds the VAO
    void Bind();
    // Unbinds the VAO
    void Unbind();
    // Deletes the VAO
    void Delete();
};

#endif
```

💡 **Note:** LinkVBO's `layout` parameter corresponds to the location in the vertex shader.

Source File: `VAO.cpp`

```
#include"VAO.h"

// Constructor that generates a VAO ID
VAO::VAO()
{
   glGenVertexArrays(1, &ID);
}

// Links a VBO to the VAO using a certain layout
void VAO::LinkVBO(VBO& VBO, GLuint layout)
{
   VBO.Bind();
   glVertexAttribPointer(layout, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
   glEnableVertexAttribArray(layout);
   VBO.Unbind();
}

// Binds the VAO
void VAO::Bind()
{
   glBindVertexArray(ID);
}

// Unbinds the VAO
void VAO::Unbind()
{
   glBindVertexArray(0);
}

// Deletes the VAO
void VAO::Delete()
{
   glDeleteVertexArrays(1, &ID);
}
```

LinkVBO Parameters:

- `layout`: Corresponds to shader's `layout (location = 0)`
- `3`: Number of components per vertex (x, y, z)
- `GL_FLOAT`: Data type of each component
- `GL_FALSE`: Don't normalize data
- `0`: Stride (tightly packed)

- `(void*)0`: Offset in the buffer

6. Refactor Main Function

Goal: Clean up main.cpp using the new classes for a triangle mesh.

Updated main.cpp

```
#include<iostream>
#include<glad/glad.h>
#include<GLFW/glfw3.h>
#include"shaderClass.h"
#include"VAO.h"
#include"VBO.h"
#include"EBO.h"

// Vertices coordinates
GLfloat vertices[] =
{
    -0.5f, -0.5f * float(sqrt(3)) / 3, 0.0f,    // Lower left corner
    0.5f, -0.5f * float(sqrt(3)) / 3, 0.0f,     // Lower right corner
    0.0f, 0.5f * float(sqrt(3)) * 2 / 3, 0.0f,  // Upper corner
    -0.5f / 2, 0.5f * float(sqrt(3)) / 6, 0.0f, // Inner left
    0.5f / 2, 0.5f * float(sqrt(3)) / 6, 0.0f,  // Inner right
    0.0f, -0.5f * float(sqrt(3)) / 3, 0.0f      // Inner down
};

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

int main()
{
    // Initialize GLFW
    glfwInit();

    // Tell GLFW what version of OpenGL we are using
    // In this case we are using OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Tell GLFW we are using the CORE profile
```

```
// So that means we only have the modern functions
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

// Create a GLFWwindow object of 800 by 800 pixels, naming it "YoutubeOpenGL"
GLFWwindow* window = glfwCreateWindow(800, 800, "YoutubeOpenGL", NULL, NULL);
// Error check if the window fails to create
if (window == NULL)
{
    std::cout << "Failed to create GLFW window" << std::endl;
    glfwTerminate();
    return -1;
}
// Introduce the window into the current context
glfwMakeContextCurrent(window);

//Load GLAD so it configures OpenGL
gladLoadGL();
// Specify the viewport of OpenGL in the Window
// In this case the viewport goes from x = 0, y = 0, to x = 800, y = 800
glViewport(0, 0, 800, 800);

// Generates Shader object using shaders defualt.vert and default.frag
Shader shaderProgram("default.vert", "default.frag");

// Generates Vertex Array Object and binds it
VAO VAO1;
VAO1.Bind();

// Generates Vertex Buffer Object and links it to vertices
VBO VBO1(vertices, sizeof(vertices));
// Generates Element Buffer Object and links it to indices
EBO EBO1(indices, sizeof(indices));

// Links VBO to VAO
VAO1.LinkVBO(VBO1, 0);
// Unbind all to prevent accidentally modifying them
VAO1.Unbind();
VBO1.Unbind();
EBO1.Unbind();

// 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
        shaderProgram.Activate();
        // Bind the VAO so OpenGL knows to use it
        VAO1.Bind();

        // 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
    VAO1.Delete();
    VBO1.Delete();
    EBO1.Delete();
    shaderProgram.Delete();

    // Delete window before ending the program
    glfwDestroyWindow(window);
    // Terminate GLFW before ending the program
    glfwTerminate();
    return 0;
}
```

Vertices Explanation:

- Creates 6 vertices forming a triangular shape with inner triangles
- Uses `sqrt(3)` for proper equilateral triangle proportions

Indices Explanation:

- `glDrawElements(GL_TRIANGLES, 9, ...)` draws 3 triangles (9 indices total)
- Each set of 3 indices forms one triangle

Summary

- **Shader organization:** Extracted shader code into `default.vert` and `default.frag` files for better maintainability
- **Shader class:** Encapsulates shader compilation, linking, and file reading functionality
- **VBO class:** Manages vertex buffer with `GL_ARRAY_BUFFER` target
- **EBO class:** Handles index buffer operations with `GL_ELEMENT_ARRAY_BUFFER` target
- **VAO class:** Manages vertex array configuration and links vertex attributes
- **Main function:** Significantly simplified using custom classes
- **Code reusability:** All classes can be reused across different OpenGL projects
- **Header guards:** Prevent multiple file inclusions and variable conflicts
- **Memory management:** Proper cleanup with Delete() methods prevents memory leaks

---

Code Implementation

Complete Project Structure:

```
OpenGLProject/
├── Header Files/
│   ├── shaderClass.h
│   ├── VBO.h
│   ├── EBO.h
│   └── VAO.h
├── Source Files/
│   ├── main.cpp
│   ├── shaderClass.cpp
│   ├── VBO.cpp
│   ├── EBO.cpp
│   └── VAO.cpp
└── Resource Files/
    └── Shaders/
        ├── default.vert
        └── default.frag
```

Build and Run:

1. Ensure all files are added to your Visual Studio project
2. Make sure `default.vert` and `default.frag` are in the project directory or specify correct paths
3. Build the project (F7)
4. Run the application (F5)

⛔ **Common Issues:**

- **File not found:** Ensure shader files are in the correct directory (usually project root)

- **Linking errors:** Verify all `.cpp` files are included in the project
- **Black screen:** Check that shaders compiled successfully and VAO is properly bound

---

Assignment

Basic Tasks:

1. **Implement all classes** following the code structure provided
2. **Add detailed comments** to each function explaining its purpose and parameters
3. **Test the program** and verify it displays the triangle shape correctly
4. **Modify background color** using `glClearColor()` to your preferred color

Intermediate Tasks:

1. **Create error checking** in the Shader constructor:
   - Check shader compilation status using `glGetShaderiv()`
   - Check program linking status using `glGetProgramiv()`
   - Print error messages if compilation/linking fails
2. **Modify the vertices** to create a different shape (square, hexagon, or star)
3. **Update the indices array** to match your new shape

Advanced Tasks:

1. **Create a second shader pair:**
   - Create `red.vert` and `red.frag` for a colored shader
   - Implement shader switching on keyboard input (press 'R' for red, 'W' for white)
2. **Add a method to VBO class** that can update vertex data dynamically:

```
void UpdateVertices(GLfloat* vertices, GLsizeiptr size);
```

1. **Implement a debug function** in the Shader class:

```
void PrintShaderInfo();  // Prints shader program ID and status
```

💡 **Bonus Challenge:**
Create a `Mesh` class that combines VAO, VBO, and EBO into a single object, making it even easier to manage geometry in your OpenGL applications.

⚙️ **Debugging Tips:**

- Use `std::cout` to print OpenGL IDs and verify objects are created
- Check OpenGL errors using `glGetError()` after each OpenGL call

- Compare your output with the source code provided in the tutorial

Expected Output:
A dark teal window (RGB: 0.07, 0.13, 0.17) displaying a white triangular mesh composed of three smaller triangles.