Assignment on OpenGL:

Shader Programming

Submitted in partial fulfillment of the requirements for

IS F311 - COMPUTER GRAPHICS

PREPARED FOR

Dr. Sundaresan Raman
Department of CSIS

PREPARED BY :

Sanju S: 2021A7PS2537P

Ohiduz Zaman: 2021A7PS2005P

Project Report: Shader Programming:

Objective:

The primary objective of this assignment is to gain a comprehensive understanding of GLSL (OpenGL Shading Language) shader programming. The focus is on developing fundamental and advanced shader programming skills to enhance the graphical rendering capabilities and performance of OpenGL applications.

Introduction:

The project aimed to delve into shader programming as a fundamental aspect of computer graphics, essential for crafting immersive 3D rendering engines. As demand rises for realistic graphics in various applications like games and simulations, understanding shader programming becomes increasingly crucial. This report focuses on the project's exploration of OpenGL, GLSL, vertex buffer management, and advanced lighting techniques like Phong shading.

Methodology:

Tools and Technologies

- **OpenGL**: Used for rendering graphics. This API provides a set of functions to render 2D and 3D graphics.
- **Glad**: manages function pointers for OpenGL
- **GLSL (OpenGL Shading Language)**: Used for writing custom shaders.
- **C++**: The programming language used to implement the rendering engine.
- **GLFW**: A library used for OpenGL context creation, window management, and handling inputs.
- **GLM (OpenGL Mathematics)**: A header-only library used for matrix and vector operations.
- **Stb_image.h** : To load textures

Implementation Steps

1. **Setup OpenGL Context and Window using GLFW**: Configured and initialized the window and OpenGL context.
2. **Shader Programming**: Wrote vertex and fragment shaders to handle vertex processing and pixel coloring, respectively.

3. **Vertex Buffer Management**: Created and managed vertex buffer objects (VBOs) to store vertex data.
4. **Implementing Lighting Models**: Implemented the Phong lighting model to achieve realistic lighting effects, including diffuse, ambient, and specular lighting.

Results:

Vertex Shaders:

- Rendering of basic 3D models and applying geometric transformations (rotation, translation, scaling, etc) on them using vertex shader.
- Manipulating colors of vertices at the vertex shader and passing them to the fragment shader.
- Applying different projections such as orthographic and perspective to the object.

Fragment Shaders:

- Manipulating colors at pixel level using fragment shader.
- Applying textures to the object using correct texture coordinates.
- Implementation of phong lighting model to get a realistic lighting effect with various combinations of ambient, diffuse and specular lighting.

Challenges and Solutions:

Challenges:

1. **Shader Understanding:** Understanding the intricacies of GLSL and integrating it seamlessly with the OpenGL pipeline posed a significant challenge. Developing shaders capable of handling vertex processing and pixel coloring efficiently, especially in the context of implementing advanced rendering techniques like texture mapping and perspective projection, required a deep understanding of shader programming principles.

2. **Integration with OpenGL Pipeline:** Ensuring smooth integration of shaders with the OpenGL rendering pipeline was another challenge. Coordinating vertex and fragment shaders to effectively process vertex data and apply appropriate color values demanded meticulous attention to detail and thorough testing.

3. **Debugging Shader Errors:** Debugging shader errors, such as compilation errors or incorrect rendering outcomes, was a challenging aspect of shader programming. Identifying and rectifying syntax errors, logical inconsistencies, or algorithmic flaws in shaders required patience, careful scrutiny, and often trial-and-error experimentation.

Solutions:
1. **Incremental Development:** Adopting an incremental development approach helped in tackling shader-related challenges effectively. Breaking down shader programming tasks into manageable components allowed for systematic implementation and refinement of shader functionalities. By focusing on one aspect at a time, such as vertex transformations or texture mapping, we could address specific challenges more efficiently.

2. **Thorough Testing and Validation:** Rigorous testing and validation of shaders were essential to ensure their correctness and reliability. Employing various testing techniques, such as unit testing, integration testing, and visual inspection, helped in identifying and resolving shader errors and inconsistencies early in the development process.

3. **Documentation and Collaboration:** Maintaining comprehensive documentation and fostering collaboration among team members facilitated knowledge sharing and problem-solving. Clear documentation of shader code, including comments and explanations, aided in understanding and troubleshooting complex shader logic.

4. **Utilizing Online Resources and Communities:** Leveraging online resources, tutorials, and shader programming communities proved invaluable in overcoming shader-related challenges. Seeking guidance from experienced developers, exploring online forums, and referring to established best practices helped in gaining deeper insights into shader programming techniques and resolving specific issues encountered during development.

Key Learnings:

- **Deep Dive into OpenGL**: Gained substantial insights into modern 3D graphics processing and rendering techniques.
- **Shader Programming Skills**: Enhanced understanding of shader programming and its impact on visual outcomes.
- **Mathematical Skills**: Strengthened knowledge of mathematical concepts used in computer graphics, such as transformations and lighting calculations.

Conclusion:

The project showcased the capabilities of a custom 3D rendering engine while providing a solid foundation in computer graphics principles. Through hands-on experience with OpenGL and GLSL, significant proficiency was gained in graphics programming, laying a robust groundwork for future projects in graphics-intensive applications.

Run Instructions:

- Cd into each folder
- make
- ./gl

References

1. **OpenGL Documentation**: https://www.opengl.org/documentation/
2. **LearnOpenGL.com**: https://learnopengl.com/ - A comprehensive guide to learning OpenGL.
3. **OpenGL Tutorials by Victor Gordon** : https://www.youtube.com/playlist?list=PLPaoO-vpZnumdcb4tZc4x5Q-v7CkrQ6M-
4. **OpenGL Crash Course** : https://www.youtube.com/watch?v=45MIykWJ-C4

Appendix

Code Snippets

Shader Initialization

```
string tmpvss = readShaderFile(vertexShaderSourceFilePath);
    vertexShaderSource = tmpvss.c_str();
    string tmpfss = readShaderFile(fragmentShaderSourceFilePath);
    fragmentShaderSource = tmpfss.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, &vertexShaderSource, 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, &fragmentShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(fragmentShader);
```

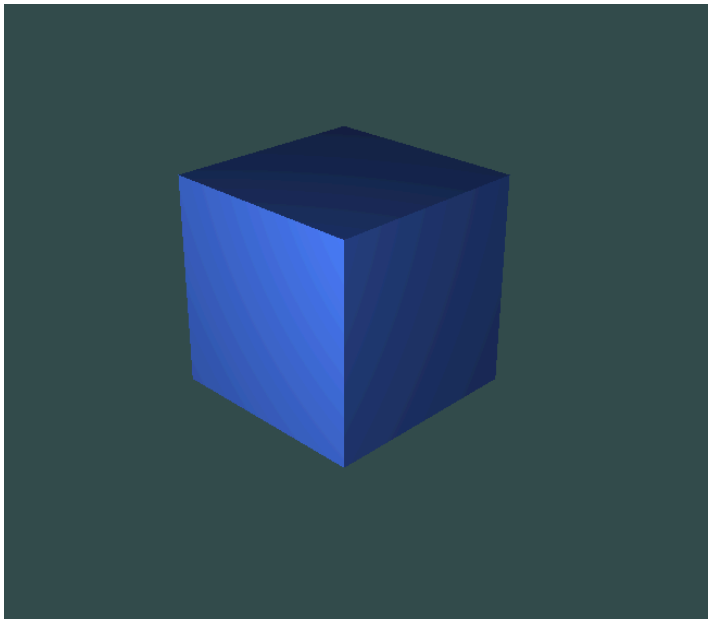
```
    // Create Shader Program Object and get its reference
    GLuint shaderProgram = glCreateProgram();
    // Attach the Vertex and Fragment Shaders to the Shader Program
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    // Wrap-up/Link all the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

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

Rendering Loop

```
while (!glfwWindowShouldClose(window))
    {
        // Input handling
        // processInput(window);

        // Rendering commands
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // Enable depth testing
        glEnable(GL_DEPTH_TEST);

        // Use shader program
        glUseProgram(shaderProgram);

        // Bind vertex array
        glBindVertexArray(VAO);

        // Draw the cube
        glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);

        // Swap buffers and poll IO events
```

```
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
```

Sample Output



Full Code:

https://github.com/sanju-suresh/CoGra-Assign2