Real Time Graphics – Exercise 1: Drawing a 2D Scene using VBO/VAO and Indices

Jan Eberwein, Johannes Eder

February 11, 2025

Abstract

This report describes the process of implementing a 2D scene of a low-poly panda using modern OpenGL techniques. The goal was to leverage Vertex Buffer Objects (VBOs), Vertex Array Objects (VAOs), and Element Buffer Objects (EBOs) to efficiently manage vertex data and indices. This exercise provided valuable experience with shader-based rendering, efficient memory usage, and structured graphics programming. The resulting scene showcases the versatility and performance of modern OpenGL for real-time graphics applications.

1.1 Introduction

The objective of this exercise was to design and render a simple 2D scene using modern OpenGL tools. The task required creating a low-poly panda composed of multiple triangles, where the vertices, indices, and attributes were carefully managed to achieve efficient rendering.

This exercise serves as an introduction to the core principles of OpenGL's programmable pipeline. Specifically, it emphasizes the use of:

- Vertex Buffer Objects (VBOs) for storing vertex data.
- Vertex Array Objects (VAOs) for managing vertex attribute configurations.
- Element Buffer Objects (EBOs) to define how vertices are connected via indices.
- Shader programs to control rendering operations on the GPU.

These tools collectively enable developers to build structured, high-performance graphics applications. Furthermore, the exercise explores how modern OpenGL methodologies allow for scalability and flexibility when rendering more complex scenes.

1.2 Implementation

The implementation involved several key steps, from defining vertex data to rendering the final scene. Each step is detailed below.

1.2.1 Vertex and Index Data

The panda was modeled using 43 vertices, each defined with a position (`x, y`) and color (`r, g, b`). The vertices were arranged to form distinct features of the panda, such as the head, ears, and limbs. Indices were used to connect these vertices into triangles, reducing redundancy and improving memory efficiency.

For example, the head was constructed as a fan of triangles centered at the top vertex, with indices defining connections to surrounding vertices. This approach minimized the number of vertices required while maintaining a detailed structure. Furthermore, the use of indices facilitated adjustments to the model without needing to modify the underlying vertex data.

1.2.2 Shaders

Shaders are small programs executed on the GPU to process vertex and fragment data. In this exercise, two shaders were implemented:

Vertex Shader: The vertex shader transformed 2D vertex positions into clip space and passed color attributes to the fragment shader.

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
out vec3 ourColor;
void main()
{
    gl_Position = vec4(aPos, 0.0, 1.0);
    ourColor = aColor;
}
```

Fragment Shader: The fragment shader used the interpolated color data to compute the final pixel color.

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
void main()
{
    FragColor = vec4(ourColor, 1.0f);
}
```

These shaders demonstrated the modularity of OpenGL's programmable pipeline, where the same shaders can be reused or adapted for different scenes and purposes.

1.2.3 Buffer Setup

The VAO, VBO, and EBO were initialized and configured to manage vertex data and indices. The VBO stored the vertex data, including positions and colors, while the

EBO defined how vertices were connected into triangles. The VAO encapsulated the configuration of vertex attributes and buffer bindings.

This separation of concerns ensured that modifications to one part of the pipeline, such as vertex attributes, did not interfere with other aspects of the rendering process.

```
void initBuffers() {
    glGenVertexArrays(1, &VAO);
    glBindVertexArray(VAO);

    glGenBuffers(1, &VBO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(Vertex),
                 vertices.data(), GL_STATIC_DRAW);

    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                 (void*)0);
    glEnableVertexAttribArray(0);

    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
                          (void*)(2 * sizeof(float)));
    glEnableVertexAttribArray(1);

    glGenBuffers(1, &EBO);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(
                 unsigned int), indices.data(), GL_STATIC_DRAW);
}
```

1.2.4 Rendering

The rendering process involved binding the shader program, activating the VAO, and issuing a draw call using `glDrawElements`. This approach ensured efficient rendering by reusing vertex data and avoiding unnecessary state changes.

Additionally, the use of an EBO allowed for significant memory savings and simplified updates to the model's geometry.

```
void renderScene() {
    glUseProgram(shaderProgram);
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
}
```

1.3 Testing and Results

The program was tested on a machine with OpenGL 3.3 compatibility. The following tests were performed:

- Correct rendering of the panda shape and colors.

- Dynamic resizing of the viewport to ensure the scene scales correctly.
- Performance evaluation to confirm smooth rendering without significant frame drops.
- Validation of the shaders to ensure proper color interpolation and geometry alignment.

The rendered scene (Figure 1.1) met all expectations. The panda's shape was accurately represented, and its features were clearly distinguishable. The use of indexed drawing reduced memory usage and improved rendering performance. Furthermore, testing demonstrated the scalability of the pipeline, indicating its suitability for more complex models.

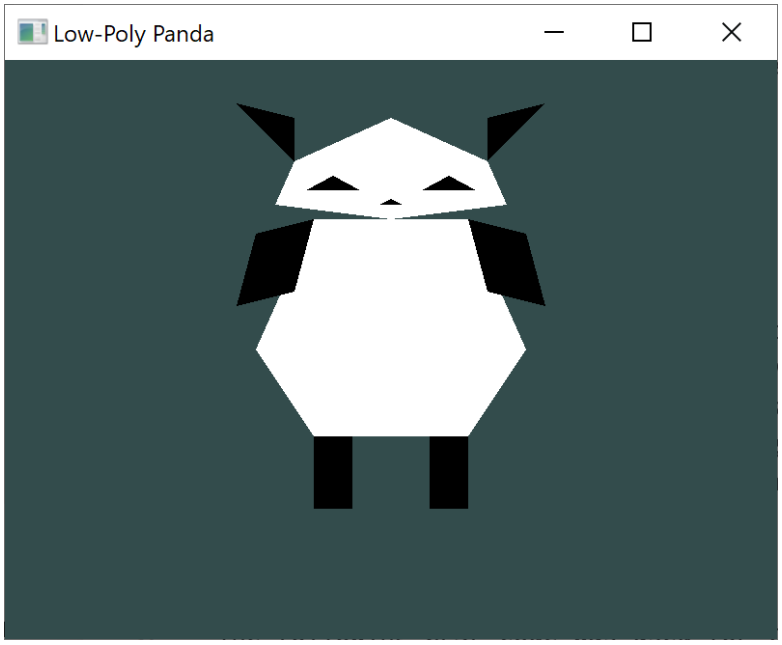

Figure 1.1: Rendered 2D low-poly panda scene. The panda's features, including the head, ears, and body, are composed of triangles with distinct colors.

1.4 Conclusion

This exercise demonstrated the effectiveness of modern OpenGL techniques for rendering 2D graphics. The use of VBOs, VAOs, and EBOs allowed for efficient memory management and reduced redundancy. Shader-based rendering provided flexibility and control over the visual appearance of the scene. Furthermore, the modular design of the pipeline ensures adaptability for future enhancements, such as adding animations or integrating textures. Overall, this assignment served as a valuable introduction to OpenGL's programmable pipeline and the foundational principles of real-time graphics programming.