Solution for armadillo-in-a-cube

Steps

Mesh Data Loading:

I loaded the mesh data from the armadillo.bin file. This file contains the vertex positions, normals, and triangle indices needed to define the 3D model.

```
int32_t load_mesh_data(const char* filename, MeshData* out_data);
```

This function loads the mesh data from the armadillo.bin file, reading the vertex count, triangle count, vertex positions, and normals. The vertex data includes positions and normals packed together, and the triangles data defines how these vertices connect to form the model.

Framebuffer Setup and Texture Rendering:

I rendered the mesh (armadillo model) to a texture using an offscreen framebuffer. This rendered texture was later used for texture mapping the cube.

```
void init_texture(SceneData* scene, MeshData* mesh);
```

The init_texture function sets up an offscreen framebuffer and attaches a texture to it. This texture will be used to render the armadillo model. The framebuffer is used to render the armadillo model without displaying it directly on the screen, allowing the rendered result to be used as a texture.

```
void render_model(SceneData* scene, MeshData* mesh);
```

In render_model, the armadillo model is rendered to the framebuffer (and thus to the texture). The rendered texture is then ready to be applied to another object – in this case, a rotating cube.

Scene Initialization:

I initialized the vertex array objects (VAOs) and shaders for the cube and the armadillo model.

I set up two key components of the scene: the rotating armadillo model and the cube. Each of these components was given its own shader program to manage its specific rendering needs.

For the armadillo model, I implemented shaders to handle the lighting and material properties, including normal mapping and ambient, diffuse, and specular components.

```
void init_cube(SceneData* scene);
```

The init_cube function sets up the cube by generating and binding its VAO and VBO, and by defining the vertex attributes for positions, normals, and texture coordinates. It also compiles and links the shaders specific to the cube.

```
void init_model(SceneData* scene, MeshData* mesh_data);
```

Similarly, the init_model function sets up the armadillo model's VAO and VBO with the loaded mesh data and configures the vertex attributes. The shaders for the model are also compiled and linked.

Texture Mapping:

The texture created from the armadillo model was applied to the cube, which was also set to rotate slowly. This gave the cube a dynamic appearance as it continuously displayed the rendered armadillo texture.

Rendering Loop:

I continuously rendered the scene with both the rotating cube and armadillo model, applying the texture to the cube.

Both the armadillo model and the cube were animated to rotate slowly, creating a dynamic scene where the texture on the cube continuously updates as it rotates.

```
void frame(SceneData* scene, MeshData* mesh_data)
```

The frame function is the core of the rendering loop, where the cube is rendered with the texture created from the armadillo model. The cube rotates slowly, and its appearance is continuously updated by the applied texture.

```
void render_model(SceneData* scene, MeshData* mesh);
```
The render_model function, which is called before the cube is rendered, handles the offscreen rendering of the armadillo model to the texture, ensuring the texture on the cube is up-to-date.

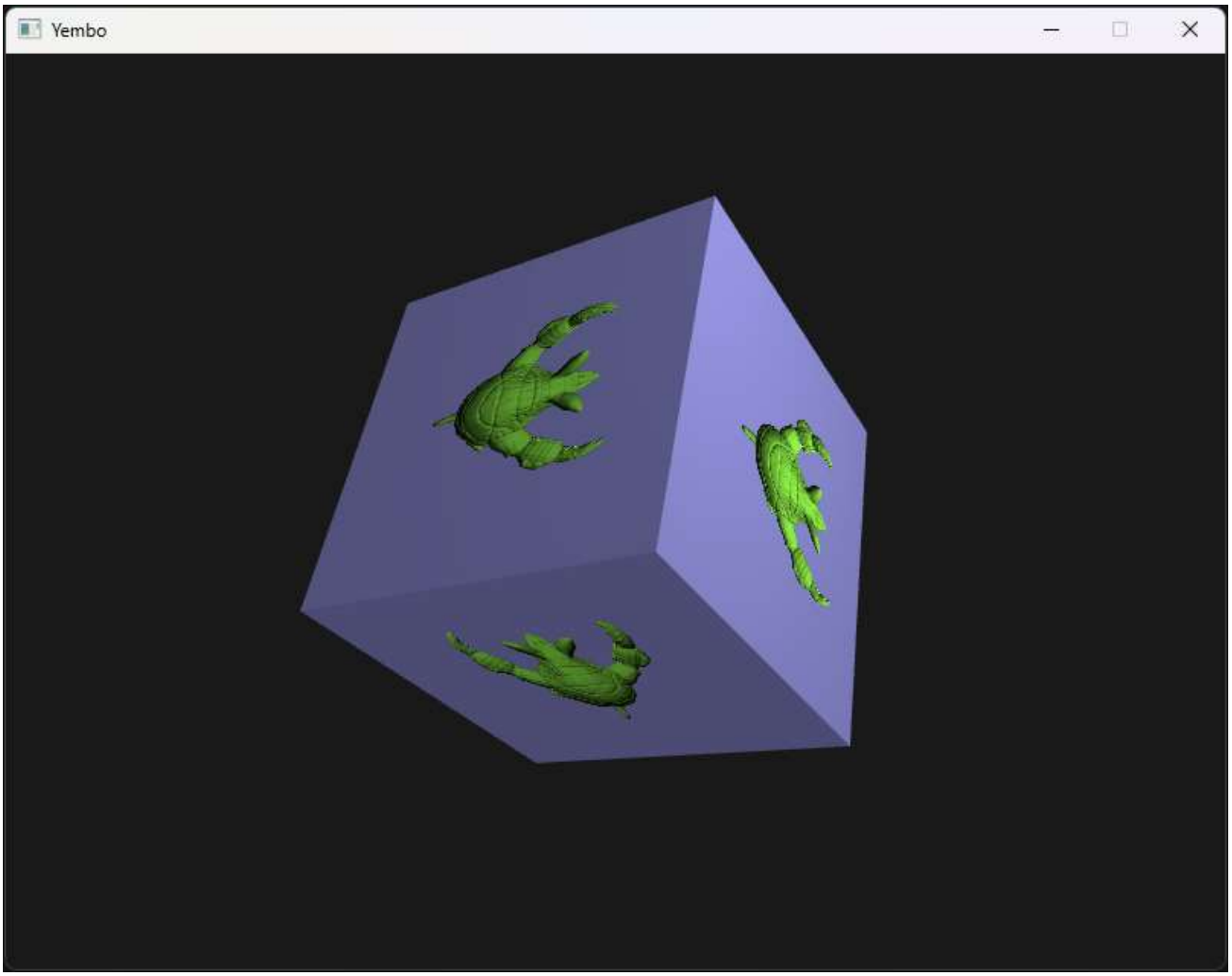
Optimization and Cleanup:

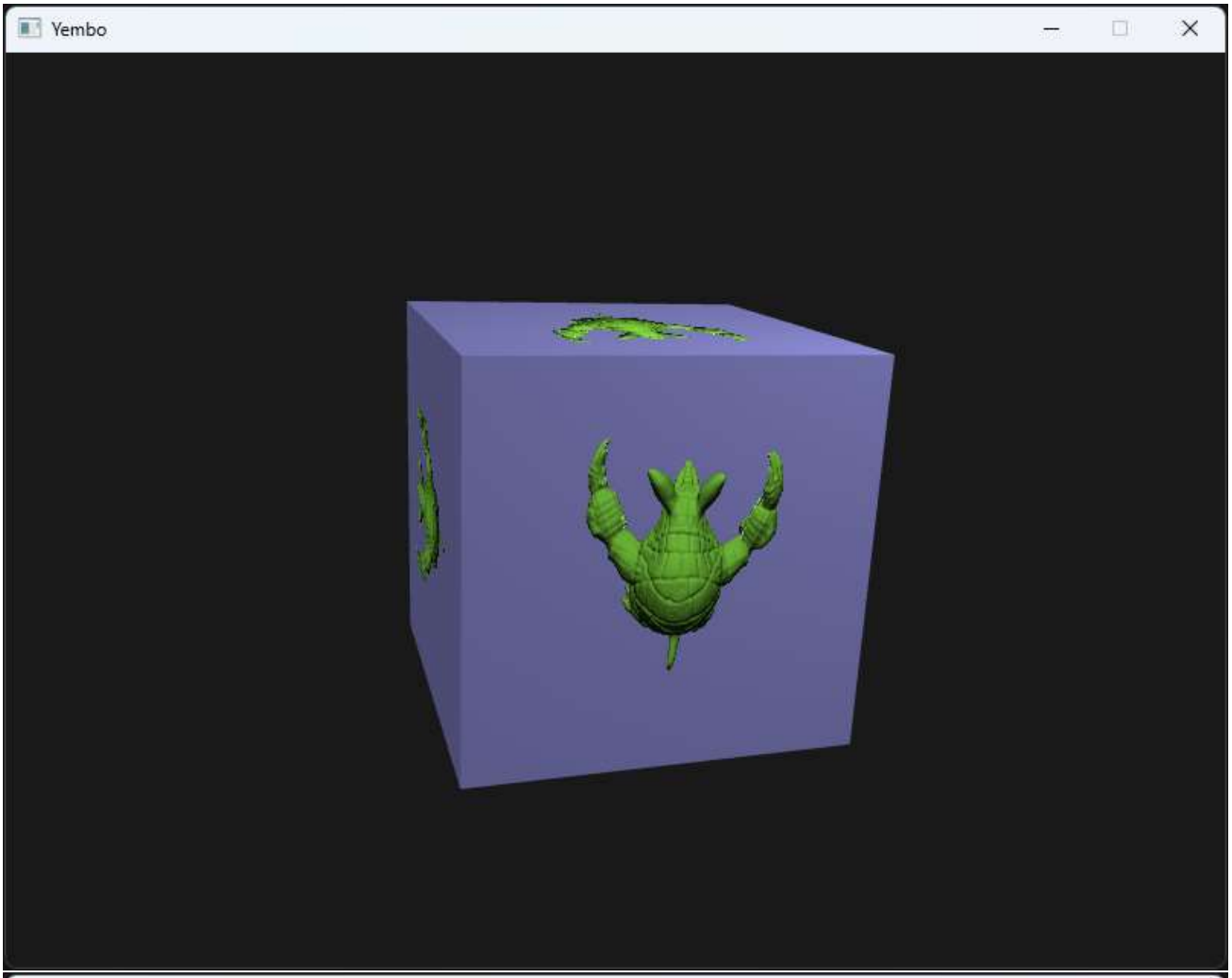
I cleaned up all allocated resources to prevent memory leaks.

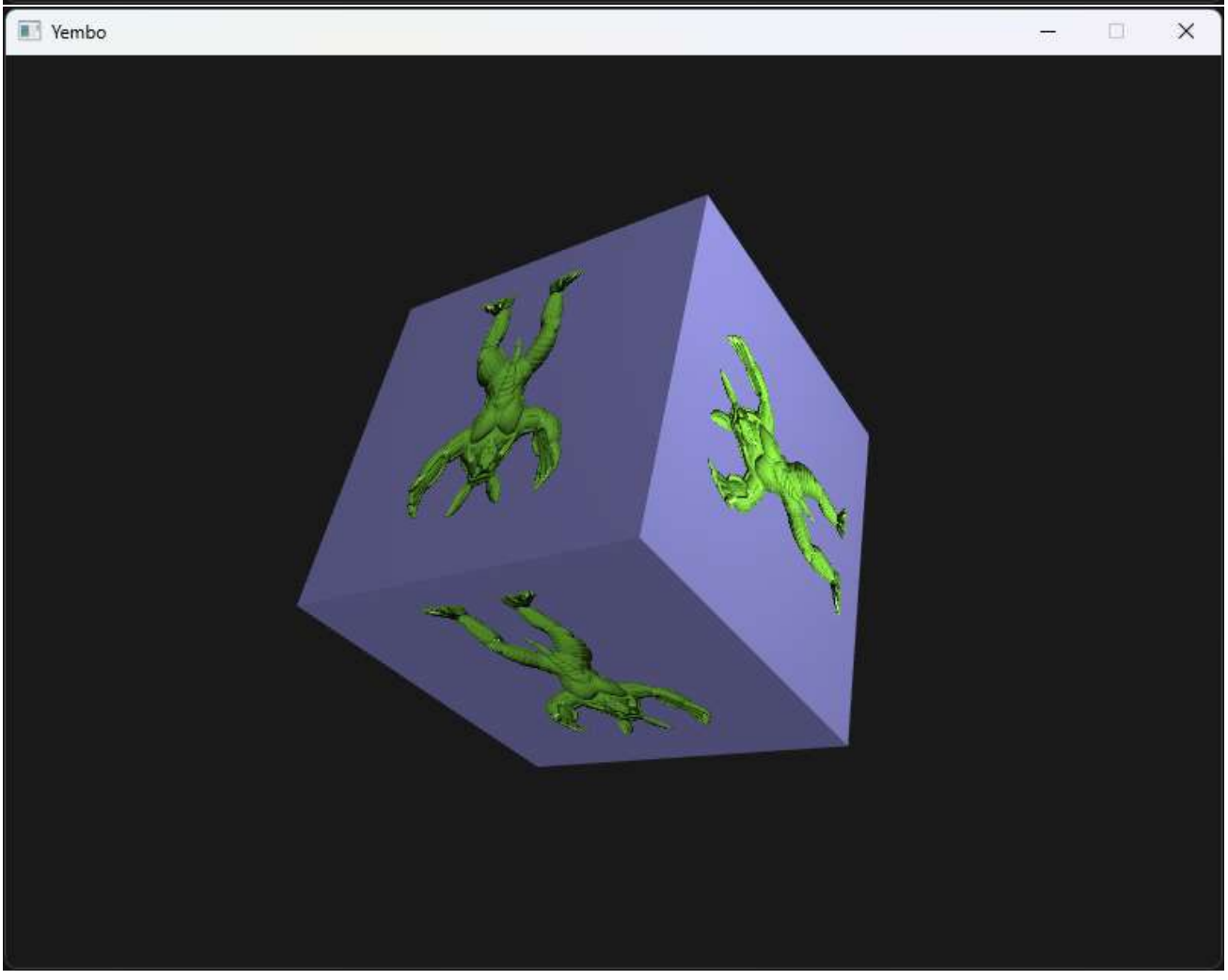
I ensured that the OpenGL resources such as vertex arrays, buffers, and shaders were properly managed and cleaned up at the end of the program to prevent memory leaks.

```
// Clean up resources before exiting
glDeleteVertexArrays(1, &scene.cube_vao);  // Delete the cube's VAO
glDeleteVertexArrays(1, &scene.model_vao); // Delete the model's VAO
glDeleteProgram(scene.basic_program);      // Delete the basic shader program
glDeleteProgram(scene.model_program);      // Delete the model shader program
free(mesh.vertex_data);   // Free the vertex data memory
free(mesh.triangles);     // Free the triangle index memory
glfwDestroyWindow(window); // Destroy the GLFW window
glfwTerminate();           // Terminate GLFW
return 0; // Return success code
```
Proper resource management is crucial in graphics programming. Here, VAOs, shaders, and dynamically allocated memory are freed and GLFW resources are terminated properly when the application exits.

Screenshots




Yembo


Yembo