Nathan Johnson

SNHU

CS-499-13459-M01

3-2 Milestone Two: Enhancement One: Software Design and Engineering

03/23/2025

Enhancement One: Software Design and Engineering

Artifact Description

I chose a 3D graphics project from CS 330: Computational Graphics and Visualization. This program, built in C++ with OpenGL, renders a fully navigable three-dimensional world. It features textured objects, dynamic lighting, and realistic shading. Users can move freely through the scene, looking at objects from any angle while the lighting shifts in real time. The program uses ambient and directional lighting models, along with texture mapping, to create an immersive experience that reacts to user input.

Justify the inclusion of the artifact

The program worked, but the code was a mess. The main file was bloated with vertex and index generators for different shapes, making it hard to follow. Global variables were scattered everywhere, making it nearly impossible to track data changes. The structure was chaotic, and making any changes was slow and frustrating. Yet, despite the clutter, the program did what it was supposed to. That made it the perfect candidate for improvement. I saw a chance to take something functional but tangled and rebuild it into something clean, logical, and well-structured.

Improvements and Enhancements

I tackled the problem by refactoring the code and creating a proper class hierarchy. I built a base Shape class and derived classes for specific shapes like Cube, along with specialized types like Light. This restructuring pulled the vertex and index generators into their respective classes, making the code far more organized. As I worked, the solution became clear, each shape needed its own defined space, a class that held not just its form but its behavior. It was now like each of the items was properly stored on shelves rather than scattered across a floor, these shapes would be easier to find, maintain, and modify when they each had a designated place in the architecture.

Through implementing constructors that accept size parameters, I enhanced the flexibility of shape creation in the program. The Cube class constructor now takes a cubeSize parameter that determines the dimensions of generated vertices. This approach allows me to create cubes of different sizes by simply passing different values to the constructor, as demonstrated in the BuildScene() function where I create one cube with size 1.0f and another with size 1.5f. While the code still supports scaling through OpenGL's matrix transformations (applied in the draw method), the size parameter in the constructor provides a more direct way to define an object's base dimensions before any additional scaling is applied. This dual approach gives more control over how objects are sized and positioned in the scene which offers flexibility for different use cases.

I also developed a Scene class to manage objects and lights, simplifying the main program logic and improving resource management through proper cleanup in destructors and the use of smart pointers for memory management.

Course Outcomes

I refined and improved this artifact, going beyond what I had originally planned. My goal was to create a stronger coding environment by restructuring and modularizing the code. I built a clear class hierarchy, followed best practices, and ensured the code was easier to maintain and scale. Each of these objectives was fully met.

Beyond those goals, I also strengthened how the project communicates its structure. The code is now well-organized, with clear documentation and logical class divisions. Its modular design makes it easier to understand, extend, and work with in the future. The refactored code with well-named classes, methods, and variables serves as a form of technical communication, making it easier for other developers to understand the code's purpose and function.

Classes lock data away while global variables leave everything exposed. Security comes down to controlling who can access what. Private data stays protected, but public data can be changed by anyone. Getters and setters create safe pathways to access information. This professional approach isn't just theoretical, it's practical. Our system can now handle new shapes of different sizes without major rewrites. The code also adapts to new needs.

Reflection on Process

Isolation matters in software design. When shapes live in their own classes, problems stay contained within clear boundaries instead of spreading throughout the codebase. Bugs have fewer places to hide, and changes to one component hurt less because they don't ripple through the entire system. Looking back, I should have planned more thoroughly at the start, though the hardest decision remained choosing between scaling methods. The solution allowed us to use both approaches as needed. Throughout the refactoring, I systematically eliminated magic numbers and replaced global variables with properly named constants. The code just works better now, it's much easier to change stuff, makes more sense when you read it, and you can add new things later without messing everything up.

OpenGL and object design don't naturally work together. OpenGL needs state while objects need independence. Finding balance between them was the main challenge. Testing each piece after changes kept the program working correctly. Functions needed to do the same job, just with cleaner code. Over time, messy code became organized design. The program runs better now and is easier to update. This project taught me how to fix code while keeping what works - a key skill for any software engineer.