Cody VanGosen

Southern New Hampshire University

CS-499 Computer Science Capstone

Professor Kraya

March 16, 2025

**CS-499 Milestone One Code Review Script**

**Introduction** Hello, my name is Cody VanGosen, and this is my Milestone One code review for CS-499. In this review, I will be examining the existing functionality, structure, and potential enhancements of my previously developed OpenGL rendering project. The project is designed to generate a 3D-rendered scene of a charcuterie board, incorporating multiple objects such as cheese, sausage, and a cup, each with texture mapping and transformations. This review will focus on three key areas: software engineering and design, algorithms and data structures, and databases. Throughout this review, I will identify areas for improvement and discuss planned enhancements that align with course outcomes.

**Software Engineering and Design** To begin, let us examine the software engineering and design aspects of this project. The program is structured using a modular approach, with SceneManager handling rendering logic and ViewManager managing the camera and viewport settings. The use of separate header and implementation files helps in organizing functionality and improves maintainability. The application follows OpenGL best practices by utilizing shaders for object rendering and GLM for mathematical operations.

Upon analysis, several inefficiencies in the code structure were noted. Looking at SceneManager.cpp, we observe that the method for rendering sausage slices contains **repetitive transformation assignments**. The following block, found in multiple locations within the file, applies transformation settings and draws the mesh:

scaleXYZ = glm::vec3(0.5f, 0.5f, 0.5f);

XrotationDegrees = 0.0f;

YrotationDegrees = 0.0f;

ZrotationDegrees = 0.0f;

positionXYZ = glm::vec3(6.0f, 0.4f, -2.0f);

SetTransformations(scaleXYZ, XrotationDegrees, YrotationDegrees, ZrotationDegrees, positionXYZ);

SetShaderMaterial("sausage");

SetShaderTexture("sausageTexture");

SetTextureUVScale(1.0, 1.0);

m\_basicMeshes->DrawCylinderMesh();

Since these transformations only slightly vary for each slice, it would be beneficial to **refactor this into a loop or a reusable function** that accepts positional parameters. This would reduce redundant code and enhance maintainability. For instance, a function such as the following could streamline the process:

void SceneManager::RenderSausageSlice(glm::vec3 position) {

scaleXYZ = glm::vec3(0.5f, 0.5f, 0.5f);

XrotationDegrees = 0.0f;

YrotationDegrees = 0.0f;

ZrotationDegrees = 0.0f;

SetTransformations(scaleXYZ, XrotationDegrees, YrotationDegrees, ZrotationDegrees, position);

SetShaderMaterial("sausage");

SetShaderTexture("sausageTexture");

SetTextureUVScale(1.0, 1.0);

m\_basicMeshes->DrawCylinderMesh();

}

Additionally, the **destructor of SceneManager** is missing explicit deletion of dynamically allocated objects such as shaders and textures. A recommended enhancement is to ensure that all allocated memory is properly freed in the destructor:

SceneManager::~SceneManager() {

for (int i = 0; i < m\_loadedTextures; i++) {

glDeleteTextures(1, &m\_textureIDs[i].ID);

}

m\_objectMaterials.clear();

}

For enhancements, I plan to introduce a TextureManager class to centralize texture loading and prevent redundant OpenGL calls. This will improve efficiency by reducing the overhead associated with reloading and reapplying textures. Furthermore, adding error handling for OpenGL function calls will ensure that failures in shader compilation or texture binding are caught and logged properly.

**Algorithms and Data Structures:** Next, let us evaluate the implementation of algorithms and data structures within the project. The application relies on GLM to perform vector and matrix transformations for object positioning and rendering. The SceneManager maintains a vector of object materials, leveraging std::vector<OBJECT\_MATERIAL> for flexible storage and retrieval.

A significant inefficiency is that **each object transformation is recalculated every frame, even for static objects**. Instead of executing transformation calculations every frame in RenderScene(), it would be beneficial to precompute matrices for static objects and store them in a **lookup table**. Implementing an **object manager** to track static and dynamic objects separately would further enhance efficiency. A recommended approach is to define a mapping for static transformations:

std::unordered\_map<std::string, glm::mat4> staticTransforms;

void SceneManager::PrecomputeStaticTransforms() {

staticTransforms["table"] = glm::translate(glm::mat4(1.0f), glm::vec3(0.0f, 0.0f, 0.0f)) \*

glm::scale(glm::mat4(1.0f), glm::vec3(20.0f, 1.0f, 10.0f));

}

Then modify RenderScene() to use these precomputed values:

if (staticTransforms.find("table") != staticTransforms.end()) {

m\_pShaderManager->setMat4Value(g\_ModelName, staticTransforms["table"]);

}

This adjustment will optimize rendering performance by eliminating redundant calculations.

**Databases and Data Management** Currently, the project does not use an external database for asset management. Instead, textures and materials are stored in in-memory structures, and scene configurations are hardcoded. This limits flexibility, as any modifications to the scene require changes to the source code. While shader uniform variables are effectively used for material properties, a more scalable approach would be beneficial.

The ideal enhancement would be to introduce **a JSON-based scene description file** that loads objects, their transformations, and materials dynamically at runtime. A simple JSON structure might look like this:

{

"objects": [

{ "type": "cheese", "position": [1.0, 0.4, -2.0], "scale": [0.5, 0.5, 0.5], "texture": "cheeseTexture" },

{ "type": "sausage", "position": [-1.5, 0.4, -0.5], "scale": [0.5, 0.5, 0.5], "texture": "sausageTexture" }

]

}

By parsing this file at runtime, the application gains **dynamic configurability**, reducing the need for hardcoded transformations in C++. Additionally, cover the breakout game as a backup option.

**Conclusion** In summary, this code review identified key areas for improvement in software engineering, algorithms, and data management. By refactoring repetitive code, implementing transformation caching, and integrating a JSON-based scene configuration system, I will enhance the efficiency, maintainability, and scalability of the project. These improvements will align with the course objectives and demonstrate my ability to refine existing systems to meet industry standards. Thank you for reviewing my code analysis, and I look forward to implementing these enhancements in the next stages of CS-499.