- For my second artifact, I am using an assignment from CS 330 where a 3D scene is rendered using OpenGL, GLM, and GLFW using C++ as a programming language. The application allows the user to control a camera to navigate a highly simplified outdoor scene using intuitive keyboard and mouse input [Run the program].

(46 OpenGLApp.h) The application features an OpenGLApp class to take some implementation details away from the main.cpp file. This class contains the application window, the camera, the root scene node, and some member variables and callbacks for keyboard and mouse input. It also contains references to OpenGL shader objects on the video card that affect the way the scene is rendered.

(21 OpenGLApp.h) Within the same header file, the class definition for a scene Object is outlined. This is a scene graph object that has a position and orientation in 3D space, a model and texture, and a list of child objects that allow for the creation of an object hierarchy. When a scene node is rendered, the model is displayed with its associated texture and orientation in 3D space. That scene node’s children are also rendered recursively with 3D transformations being combined in a hierarchical fashion. This means that when an object is moved or rotated, its children are also moved and rotated to follow their parent, but they also have their own local position and orientation.

(14 Model.h) The Model class contains OpenGL references to vertex arrays that store vertex information about 3D models. This class contains static functions to create common 3D primitives like spheres and cylinders. Its only non-static function is the draw function which displays the model on screen.

(5 Texture.h) The Texture class contains an OpenGL reference to image data that can be attached to 3D models to make them appear more interesting. It has a member function to load an image from a file, and a member function to tell the OpenGL library to use this texture.

(7 Camera.h) The camera class contains a position and two rotation member variables to indicate its location in 3D space. A collection of functions exist to move and rotate the camera programmatically as well as a getter function to acquire the view matrix. The view matrix is supplied to OpenGL at each frame to transform the 3D scene with respect to the camera.

(14 main.cpp) The main.cpp file contains code to initialize the GLFW windowing library and set up the scene. The window is created and the graphics shaders are defined before being passed to a new instance of the OpenGLApp class. The scene is then created programmatically by instantiating Model, Texture, and Object instances to create a scene graph and position the camera. The OpenGL app start() function is executed to begin the event loop. Finally, the GLFW library is terminated to properly exit the program.

**Structure**

- The purpose of this application was to create an abstract 3D representation of a real-world scene. My implementation depicts a minimal outdoor scene with a ground, road, water, trees, a car, and some buildings. The code correctly implements the design requiring 3D geometric primitives with textures, shading, and transformations as well as camera movement.

(1 OpenGLApp.h) The code conforms to the standard by the Google C++ Style Guide where it uses “#pragma once” guards that tell the Visual Studio compiler not to include a header file more than once to prevent circular definitions and redefinitions of classes and functions. This guard is specific to Visual Studio however, and it is recommended to use more universal “#ifndef/#endif” and “#define” macros to improve compatibility with other compilers.

(73 OpenGLApp.h) The Google C++ Style Guide also suggests making all member and non-member variable names as well as function parameters lowercase and underscores to separate words. The code in this application instead uses lowerCamelCase for variable names. This could be improved in the enhancement, but this rule is at least consistent throughout this application.

- It is generally recommended to keep lines of code under 80 characters in length to improve code readability, especially for smaller screens. The source files in this project regularly exceed 80 characters, so lines should be shortened to adhere to this guideline.

- The Google C++ Style Guide recommends using only spaces for indentation, and not tabs. The code in this project uses tabs and could be updated to indent using spaces to adhere to this guideline.

(21 OpenGLApp.h) The code is well structured and consistent in style throughout the project. Whitespaces, brackets, indentation, naming conventions, and other conventions do not change throughout the source code. One criticism I would bring up in reviewing this code now is that the Object class does not have its own header and source file, which is unlike the other classes in this application. The Object class seems out of place being defined inside the OpenGLApp header file and source file, so it should be moved to its own files for organization and consistency.

(58 Model.cpp) There is no unreachable code in the application, but there are some functions that are not used. For example, the Model class contains a function called createPyramid() that was utilized in a previous assignment but is unused in this artifact. It doesn’t make sense to remove this function because the codebase for this artifact is designed to be repurposed and reused for other graphical projects that might require a pyramid primitive.

(322 OpenGLApp.cpp) A function like the Object class’s getChildren() function is similarly unused but exists for a necessary purpose. Some utilizations of this codebase could derive value from getter functions like these even though they are unused in this specific application. Being able to iterate over child nodes in a tree is useful behavior and should not be excluded since it takes up very little space.

(23 main.cpp) There are no leftover debugging stubs or test print statements in this application. Only necessary cout print statements are included to inform the user of errors initializing the program.

- There is no code that can be replaced by library functions from the libraries within this project. OpenGL has a bare and minimal pipeline to complete graphics operations, so the programmer must write their own logic to handle more complicated functionality.

(67 Camera.cpp) The GLM library that this project uses allows the programmer to easily perform matrix math operations and 3D transformations without needing to know the linear algebra behind them. The Camera class already makes use of these library functions to simplify the implementation. This glm::lookAt() function drastically simplifies the code required to create a view matrix representing the camera looking at a specific point in the 3D scene.

(109 main.cpp) The most obvious place where repetitive code could be condensed into a single procedure is the scene setup code. The creation of objects and their attachment to models and textures as well as their transformations could be combined into a single helper function that achieves this task in one go. While the positioning and orientation of objects is hardcoded and cannot be simplified, all of these blocks of repetitive code could be considerably reduced.

(83 & 103 main.cpp) Storage use is highly efficient in this application because models and textures are only loaded once but used in several instances of scene Objects.

(215 & 13 Model.cpp) Additionally, in the Model class, vertices are generated in RAM before being supplied to the video memory using OpenGL functions. After the model is supplied to the video card, the original vertex data is destroyed when the function returns. For example, the createCylinder() function declares a vector of floats to hold the vertex data. After a Model is generated using the constructor, the function will return, and the vector’s destructor will automatically delete the unused storage.

(50 Texture.cpp) Similarly, once images in the Texture class are loaded and supplied to OpenGL, the original image data stored in RAM is freed.

(77 OpenGLApp.cpp) There are a few constants and literals within this artifact that are unclear and hardcoded into the functionality of the program. For example, the background color in the event loop is set to an arbitrary blue color that cannot be changed programmatically when using the class externally.

(30 main.cpp) In main.cpp, graphical shader programs for the application are stored as string literals even though they are complex and hard to read. It would make sense to store information like this on the hard drive and load it as needed to simplify the code and improve readability as well as make the application more flexible in the future.

(21 OpenGLApp.cpp) There are very few modules in this codebase and the labor of graphical operations is divided appropriately between different classes such as the Camera, Model, and Texture classes. As I mentioned before however, the Object class could stand to be moved into its own header and source files.

**Documentation**

(Model.h, Texture.h, Camera.h) While much of the source code is commented in this artifact, many class definitions are uncommented and their documentation leaves a lot to be desired. I should include more comments to describe function meanings and the purpose of various classes.

(76 OpenGLApp.cpp) Some comments are not updated to reflect changes that were made to the code. This comment claims that the screen is going to be cleared to the color black, but it is instead cleared to a light blue.

(109 main.cpp) Much of the scene creation in main.cpp is not commented and does not describe the positioning and orientation of scene nodes. Additional documentation could be included here to inform the individuals reading the code of what operations and numbers correspond to which spatial transformations.

**Variables**

(272 OpenGLApp.cpp) Most variables within this artifact are properly defined. Member variable pointers are initialized to nullptr before a valid object instance is available. The constructor for the Object class demonstrates this because the “model” and “texture” variables are initially set to nullptr.

(228 main.cpp) Most variable names are meaningful and self-explanatory. For example, when setting up the scene, objects are given names like “carTire1” or “building4” to make it clear to the reader which part of the scene they comprise.

(375 Model.cpp) In the Model class, parameters are given single or two letter names that are not highly descriptive and are vague to someone unfamiliar with the program. Variables like “x” could be changed to “xCoordinate” and “r” could be changed to “red”.

(206 OpenGLApp.cpp) There are areas of the program that involve typecasting that is risky but necessary. The GLFW library invokes programmer-provided callbacks such as the key press callback and supplies the function with the current GLFW window object. The GLFW window object contains a user void\* pointer supplied by the programmer. It is difficult to be certain that the void\* pointer points to the OpenGLApp instance supplied at the beginning of the program, but this has to be assumed to be true by the programmer. One way to improve this cast would be to change it to a standard C++ “static\_cast” for clarity.

(13 Model.cpp) OpenGL functions are regularly called without checking types or properly casting in this application. For example, the vertex data is supplied to this function as a pointer to an OpenGL floating point number, but the function takes a void\* pointer. It is common not to cast values provided to OpenGL function and let the compiler infer types automatically, since types like GLfloat are almost always the same as float.

(32 OpenGLApp.h) An unused variable called “usePerspective” shows up in the OpenGLApp class that is always true. This variable was used initially to allow the programmer to switch between orthographic and perspective camera modes, but is not given any getters or setters to access it. It would be a good idea to simply remove this variable and always allow the application to use perspective mode.

**Arithmetic Operations**

- There are no instances of equality checks between floating point numbers.

(21 OpenGLApp.h) There is no high precision floating point arithmetic required for this application to run correctly. 3D translation and rotation through matrix multiplication incurs rounding errors that are completely negligible, especially since these errors are not allowed to propagate over time. One consideration for the Object class is that scene graph transformations will become subject to rounding error that snowballs as the depth of the object hierarchy increases because transformations are combined using matrix multiplication at each level of the tree. This is not a concern for simple scene graphs.

(167 Model.cpp) There are a few instances of division where zero divisors are not prevented. Here the constant 2π is being divided by the variable “longitudeDivisions”, which could easily be zero if the programmer makes a mistake. Adding checks for this could prevent division by zero errors from occurring.

**Loops and Branches**

(209 OpenGLApp.cpp) The keyboard callback function contains conditional branches that do not cover all possible cases. The “action” variable is checked for equality to “GLFW\_PRESS” and “GLFW\_RELEASE” constants but there is a third constant “GLFW\_REPEAT” that is not accounted for.

(224 Model.cpp) The variable “radius” here does not change and can be initialized before the loop begins, as well as the variable “uInc”. There are more examples of this in other primitive generating functions in the Model class.

- The code does not attempt to manipulate index variables during loop execution in any of the source files.

(62 OpenGLApp.cpp) The only loop that is not guaranteed to terminate is the main event loop, which is acceptable since it depends on the user clicking on the ‘X’ button to close the window.

**Defensive Programming**

(124 OpenGLApp.cpp & 257 main.cpp) There are only two examples of array indexing in this application. In OpenGLApp.cpp, the infoLog char array is initialized to a size of 512, and the function responsible for filling this array, “glGetShaderInfoLog()”, is similarly passed 512 as a value to indicate the max size of the char array. This magic constant of 512 could be replaced with a compile-time constant variable. In main.cpp, the “backgroundBushes” array is set to a constant size of 20 and the loop runs through exactly 20 iterations. This constant of 20 could similarly be replaced with a variable.

(14 main.cpp) The command line arguments in the main function are not tested for validity because they are not utilized in the program.

- There are no output variables that need to be assigned in this program aside from graphical objects which are clearly being output to the display upon launching the application.

(84 main.cpp) All of the correct data is being operated on, although there is no realistic way to validate images that are loaded into the program. It is assumed that the textures that are loaded from the hard drive represent the appropriate graphics.

(274 main.cpp) No explicit object allocations are made within this program since all objects are stored on the stack. However, a very important GLFW library function is not invoked before the program terminates. The function glfwDestroyWindow() should be called at the end of the event loop to clean up GLFW resources before glfwTerminate() is called.

- There doesn’t seem to be a way to check if input devices exist using GLFW functions, so keyboard and mouse callbacks simply won’t be invoked if input devices are not available.

(25 Texture.cpp) Images are the only files being accessed in this application, and in case an image file does not exist when it is loaded here, the program will not enter into this conditional branch to copy the texture data. This is a safeguard to prevent the program from crashing if the file does not exist.

- In enhancing this artifact, I will address the previously mentioned weaknesses, limitations, and vulnerabilities. I will also add new features to this application to demonstrate my understanding of data structures and algorithms. Firstly, I will implement a radix sorting algorithm to allow the programmer to sort objects in the scene graph tree by a new index member variable. This index variable will indicate the drawing or rendering order of 3D objects in the scene graph. Implementing this feature will target the outcome CS-499-03 because I will be programming a solution to solve a logic problem of sorting and implementing it algorithmically in software. Secondly, I will implement an algorithm for the scene graph that will collapse entire nodes and their children recursively into a single list of nodes instead of a tree. Implementing this feature will target the CS-499-03 outcome by showing how I can clearly articulate approaches to solving complex logic problems algorithmically by converting one data structure to another. Lastly, I will implement an algorithm for deep copying whole nodes of a tree using a stack data structure. This feature will also target the CS-499-03 outcome. Enhancing this artifact will partially fulfill the CS-499-04 outcome since I will be iteratively testing code as I develop and creating more robust and efficient code by mitigating existing design flaws.