**Milestone Three: Data Structures and Algorithms**

Harrison Bergeron

CS 499: Computer Science Capstone

Prof. Brooke Goggin

July 24, 2022

The artifact I chose for the data structures and algorithms category is a 3D OpenGL scene viewer where a [3D environment](#Figure1) is programmatically constructed to resemble an illustration called Southern Freeway by Hiroshi Nagai (2017). This artifact came from CS 330 and I initially created it to demonstrate my understanding of OpenGL concepts like vertex arrays, transformations, textures, lighting, and shader programs. I selected this item for my ePortfolio because it contains a special Object class representing a tree data structure. This class makes it possible to construct hierarchical 3D scene graphs using trees, where child nodes inherit the translation, rotation, and scaling of parent nodes. There were multiple opportunities to implement new algorithms and utilize new data structures to enhance this artifact by modifying the Object class.

I enhanced the code for this application in three different ways. Firstly, I created a function for deep copying tree nodes by traversing all the descendants in a tree. I made the unique choice to implement this tree traversal using a stack data structure instead of recursion to showcase my ability to emulate recursive behavior without self-invoking functions. Instead of invoking a function recursively, a special object is pushed onto a stack data structure to keep track of the current tree depth and iterations. My implementation is analogous to the concept of function stack frames in regular recursion. This enhancement targets the CS-499-03 outcome because I needed to find a creative workaround for a common tree traversal algorithm without using a feature that is language dependent. I showed that I was able to solve a logic problem involving algorithms and data structures by substituting recursion with a stack data structure. I made appropriate tradeoffs when selecting a data structure and algorithm to achieve a similar time and space complexity as recursion. In C++, the call stack has memory limitations that could limit the depth of tree traversal; my implementation uses a stack data structure located on the heap which broadens the amount of space available for deep traversals. Deep copying complex objects is a common goal in software development, and I demonstrated my ability to achieve this goal in two ways. In the [code](#Enhancements) as well as in [figure 2](#Figure2), I showcase my enhancement by deep copying the root scene node and displaying it next to the original.

Secondly, I enhanced the artifact by implementing a radix sort algorithm. In OpenGL applications, it is a common goal to manually set the drawing order of objects in order to guarantee that blending works correctly. For example, transparent objects are often rendered after every other object to prevent graphical bugs involving the depth buffer. My algorithm implementation guarantees that the children of every Object node will be sorted by their “draw index” before being drawn. This enhancement targets the CS-499-03 outcome because I used innovative techniques to accomplish an OpenGL industry standard of controlling rendering order of 3D objects. I also evaluated sorting algorithms to determine that comparison-based sorting algorithms usually have a lower limit of O(n log n) time complexity, and the non-comparison-based radix sort achieves O(n) time complexity when the number of digits are constant. Since Objects in this artifact have an integer index, they were the perfect candidate for a radix sort which is highly efficient when many objects will need to be sorted. I made design tradeoffs regarding space efficiency because this algorithm requires allocating an extra array for temporary storage in order to maximize time efficiency. In the [code](#Enhancements) as well as in [figure 3](#Figure3), I showcase my enhancement by creating a list (std::vector) of Objects with varying draw indices, and sorting them using my radix sort implementation. The radix sort function is also automatically called in the Object::Draw() function.

My last enhancement adds functionality for converting an Object tree data structure into a one-dimensional list data structure. The Object::Collapse() function can be invoked on a scene graph node which causes all the descendants in the tree to be traversed and added to a list. Ultimately, all hierarchical information is lost when the nodes are gathered into a list, but 3D transformations are combined recursively to preserve the position and orientation of the nodes in 3D space, even though nodes no longer inherit parent transformations. To demonstrate this function, I programmatically create a literal 3D binary tree with branches and leaves using recursive lambda functions. The position and rotation of each branch is inherited from its parent, which allows for easy generation of a tree model using only simple transformations. I then use the Object::Collapse() function to convert this complex hierarchy into a single parent node with a long list of all descendant nodes, showing that the tree has indeed been converted to a one-dimensional list (but remains graphically identical to the original model). This enhancement targets the CS-499-03 enhancement because I implemented an algorithm to convert one common data structure into another, which is a common industry goal. When data needs to be stored or transmitted, it is often useful to be able to convert multidimensional data structures into one-dimensional data structures that can be sent over a stream. There are multiple ways to convert a tree data structure to another data structure, and I clearly articulate my approach using recursion and lists (std::vector) to solve the complex logic problem of traversing and converting this data structure to another form, which targets the CS-499-03 outcome because it achieves the goal of making necessary tradeoffs in algorithm and data structure design to optimize a program. It is clear from the working product in the [code](#Enhancements) and from [figure 4](#Figure4) that I can clearly articulate my ideas regarding algorithm design and data structure utilization.

I met all the course objectives I planned to meet in my initial ePortfolio plan for the data structures and algorithms category. When creating the first enhancement featuring a deep copy function, I encountered the challenge of how to organize data within the stack data structure to mimic recursion. It took me a considerable amount of time to decide on the minimal amount of data required for each stack element in order to implement a deep copy. In the end, I discovered that the original node, the copy node, and an iterator marking the current location in the loop through node children were all that was necessary. I learned that using a stack in place of recursion is almost visually identical to recursion despite being more difficult to articulate in code. A technical challenge I faced when implementing the second and third enhancement was using C++ lambdas recursively. I dealt with compiler errors that prevented lambdas from capturing themselves for self-invocation. I learned that declaring the lambda in advance before initializing it allows the function to reference itself for the purpose of recursion. Lastly, in the third enhancement I had difficulty remembering how matrix transformations compose. In order to implement the C++ code to combine parent and child transformations, I had to read up on GLM documentation to learn about how to multiply matrices to combine their linear transformations.

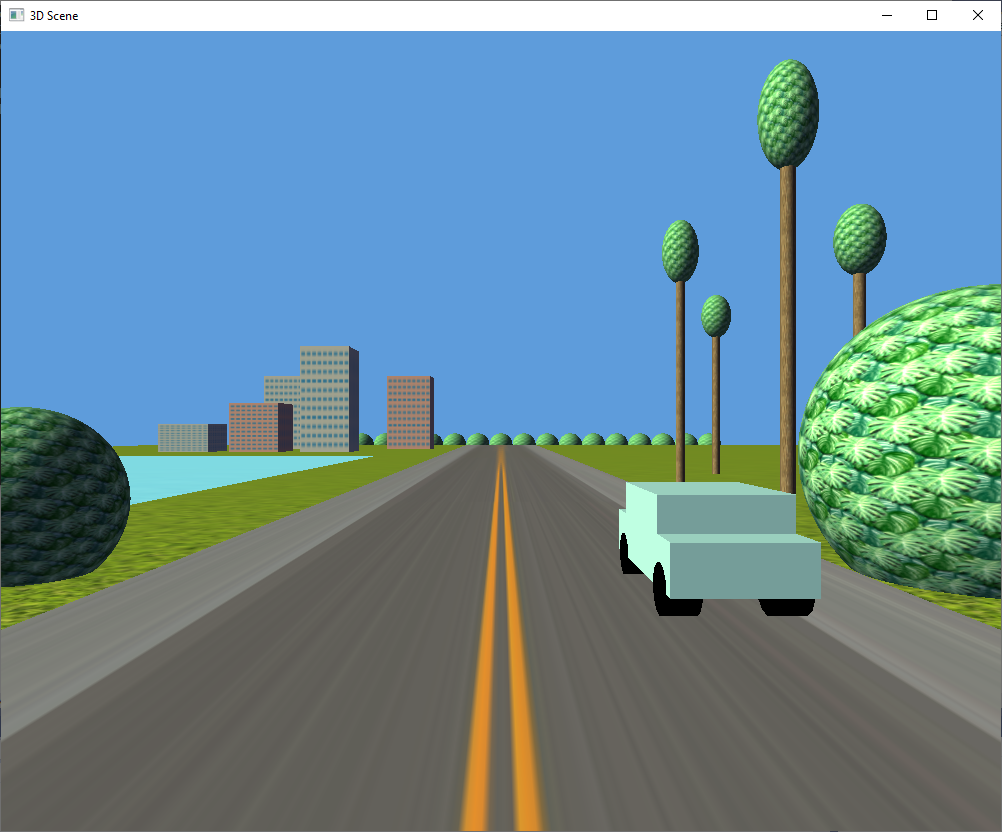


Figure 1 - Original Artifact

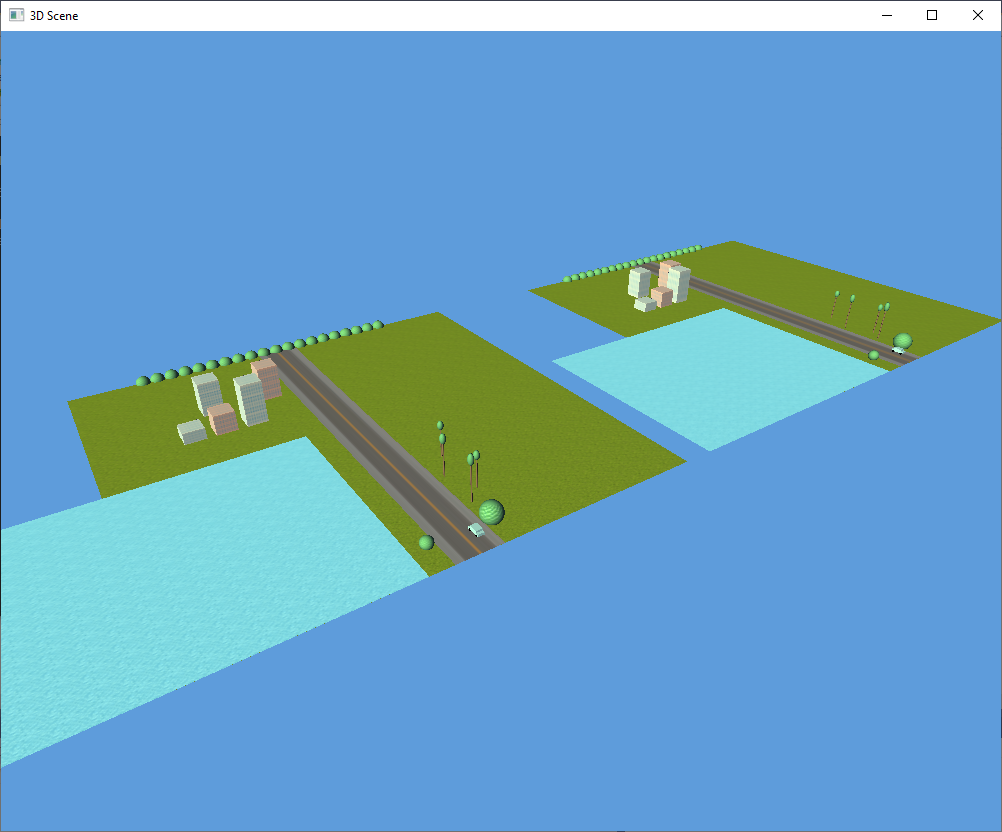


Figure 2 - Deep copying a scene graph tree data structure using a stack

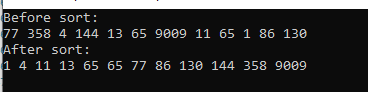


Figure 3 - Radix sort on the children of a tree node based on their draw\_index\_ variable



Figure 4 - Generating a literal tree using a tree data structure (in front) and converting it to a list data structure (behind)

**Code Enhancement Sections**

Enhancement 1: Object.h 54-60 / Object.cpp 128-171 / main.cpp 275-298

Enhancement 2: Object.h 61-78 / Object.cpp 173-233 / main.cpp 304-361

Enhancement 3: Object.h 79-87 / Object.cpp 235-280 / main.cpp 367-500

**References**

Nagai, H. (2017). *Southern Freeway* [Painting]. https://thevinylfactory.com/wp-content/uploads/2019/04/hiroshi-nagai\_cover-art2.jpg