2017

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

OPTIMIZATION OF SPATIAL PARTITIONING FOR 3D-PATHFINDING SPECIFIED FOR SPARSELY POPULATED, SIMULATED ENVIRONMENTS

Brian Abdo

A Thesis Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2017

Approved by

Advisory Committee

Tom Janicki _____

HyunBum Kim _____

Brittany Morago
_____
Chair

Accepted By

Ron Vetter
_____
Dean, Graduate School

TABLE OF CONTENTS

Page

Chapter 1: Introduction 7
1.1 Pathfinding Problem 7
1.2 Context of the Problem 8
1.3 Purpose of Paper 9

Chapter 2: Review of Literature Review and Analysis 10
2.1 Octree 10
2.2 Octree Encoding 11
2.3 Linear Octrees 13
2.4 Sparse Voxel Octree 14
2.6 Loose Octree 15
2.6 Neighbor Traversal 16
2.7 Tree Pruning 18
2.8 Research Question 19
2.9 Shortcoming of Current Methodology 20

Chapter 3: Methodology 21
3.1 Technology 21
3.1.1 Unity 21
3.1.2 Unity Profiler 21
3.1.3 Unity Gizmos 22
3.1.4 Hardware Specifications 22
3.2 Spatial Partitioning System Requirements 22
3.3 Process 23
3.3.1 Octree 23
3.3.1.1 Location Codes 24
3.3.1.2 Node Size 25
3.3.1.3 Closest Index Position 28
3.3.1.4 Recursive Traversal Initialization 29
3.3.1.5 Insert Obstacle 30
3.3.1.6 Remove Obstacle 31
3.3.1.7 Change Obstacle Position 32
3.3.2 Sparse Voxel Octree 32
3.3.2.1 Insert Obstacle 32
3.3.2.2 Recursive Removal 33
3.3.3 A* 34
3.3.3.1 End Node 35
3.3.3.2 Directional Key 35
3.3.3.3 Key by Position 36
3.3.3.4 Traverse Neighbors 38
3.3.3.5 A* Incremental 38

Chapter 4: Testing 41
4.1 RAM Overhead 42
4.2 Traversal Efficiency 42
4.3 Updating Efficiency 42

Chapter 5: Results and Analysis 44
5.1 RAM Overhead Test Results and Analysis 44
5.2 Traversal Efficiency Results and Analysis 46
5.3 Updating Efficiency Test Results and Analysis 52

Chapter 6: Conclusion, Future Work, and Afterwards 56
6.1 Future Work 56
6.2 Conclusion 57

References 60

Appendixes
A. Appendix A 62
B. Appendix B 63
C. Appendix C 64
D. Appendix D 65
E. Appendix E 67
F. Appendix F 79
G. Appendix G – Raw Data 79

ABSTRACT

Optimization of Spatial Partitioning for 3D-Pathfinding Specified for Sparsely Populated, Simulated Environments.

Abdo, Brian. Thesis Paper, University of North Carolina Wilmington.

Pathfinding in 3D-Space is one of the more novel and complex use cases in a world where the advent of new technology is abundant. Drones, Robotics, Simulations, and Game Development have dealt with the stated use case for some time. Depending on the specific requirements, a solution is presented that almost always differentiates vastly from its brethren.

More often than not, it's the pathfinding methodology that is optimized over the spatial partitioning structure itself if a partitioning structure is used at all. This paper analyzed and optimized an Octree data structure for 3D pathfinding by using and integrating current methodology in conjunction with A* pathfinding. The A* pathfinding methodology was a standard version with the stated goal of testing the usefulness of the implemented methodology for optimization in a variety of 3D-pathfinding scenarios.

Many portions of this paper are exploratory as the subject of spatial partitioning in 3D has been developed for many fields other than strictly computer science. Modeling is the primary implementer of the structure style required and much of this paper has referenced the historical and theoretical work of that field.

LIST OF TABLES

Table Page

1. 1 Octree Random Astar....47
2. 1 Octree Wall Astar....47
3. 15 Octree Random Astar....48
4. 15 Octree Wall Astar....48
5. 1 SVO Random Astar....49
6. 1 SVO Wall Astar....49
7. 15 Octree Random Pruning....52
8. 15 Octree Wall Pruning....53
9. 15 SVO Random Pruning....53
10. 15 SVO Wall Pruning....54

LIST OF FIGURES

Figure Page

1. Chessboard....7
2. Octree Structure....10
3. Morton Order....12
4. Loose Octree Design....15
5. Brewer SVO Layout....16
6. Sharing a common face (6), edge (12), or vertex (8)....17
7. Location Code Lookup - Child....25
8. Location Code Lookup - Parent....25
9. Node Size....26
10. Octree Numbering....27
11. Node Position....27
12. Closest Index Position....28
13. Recursive Traversal Initialization - Octree....29
14. Insert Obstacle....30
15. Remove Obstacle....31
16. Change Obstacle Position....32
17. Insert Obstacle - SVO....33
18. Recursive Removal....33
19. Find End Node....35
20. Directional Key....36
21. Key by Position....37
22. A* Incremental....39
23. Bit Amount....44
24. Layers....45
25. Comparison 1 SVO Tests....50
26. Comparison 15 SVO Tests....51
27. Comparison of All SVO Tests....51
28. Update Comparison....54

CHAPTER 1: INTRODUCTION

Pathfinding, or traversal from one location to another to another given a set of rules, in a 3D Space is considered one of the more complex issues in a number of fields. This is in juxtaposition to pathfinding in 2D Space where there are a number of common implementations to consider. One of the primary reasons for the contrast is the exponentially large space to analyze for the former [1].

Pathfinding has multiple components to analyze. Foremost, a pathfinding system requires a way to understand and then analyze the area around an object in question, denoted as *zeta,* integrated with a way to traverse the understood area based on the ability of the directional navigation of *zeta.*

1.1 Pathfinding Problem

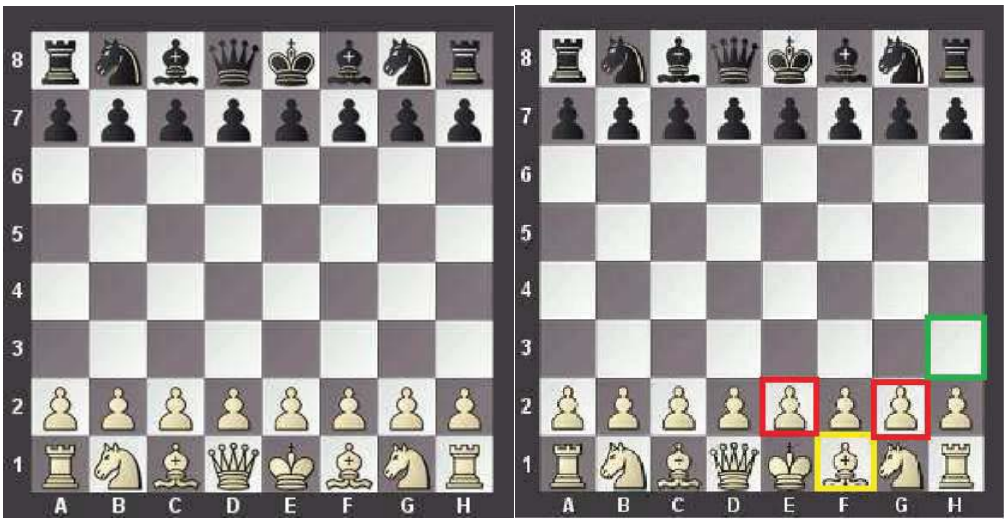


Figure 1: Chessboard [2]

Take, for example, a Chess Board. In this instance, *zeta* is the Rook at F1. To move, *zeta* needs to be able to process a lot of information. First, *zeta* is located on a 2D map split into 64 squares. It needs to understand what squares are occupied and what squares are free. Once ascertained, it analyzes its ability to move to a selected location, H3, from its position. To do

this, it traverses its neighbors looking for the most optimal path to the destination. Both G2 and D2 are occupied in this situation so it is unable to create a path. But, in the case G2 was unoccupied, *zeta* would find a path from F1 -> G2 -> H3.

1.2 Context of the Problem

Pathfinding in 3D space is more complex than the example discussed above. *Figure 1* only has 64 possible spaces to move to. If *zeta* needed to move up or down, rather than diagonal, there are entirely new problems to consider. 2D pathfinding is very efficient because there are a limited number of traversals required [3]. For every layer required to increase the scope of traversal in a vertical direction, there can be a multiplicative increase in the number of nodes required to traverse in order to find an optimal path.

One of the more novel solutions for both traversal and space partitioning is the implementation of octrees, a 3D-Modeling data structure. The data structure was developed, in part, to allow highly-complex models to be viewed in a simulated environment by partitioning space in an efficient and optimal manner [4]

Octrees, due to their nature as hierarchical structures, have the ability to provide numerous avenues for optimization of 3D pathfinding. The primary use cases that would demand such a structure would be those environments that were sparsely populated as to increase the efficiency of traversal. This will be the focus of the paper as highly populated environments, while possible with octrees, require a different focus for optimization that will not be discussed in depth here. More information may be found with reference to modeling or scanning techniques for extrapolating empty space from point cloud information and into an octree data structures as noted by Broersen, et. Al [5].

There is, however, a primary focus for all octree-like structures. That is, to partition data in such a way that it can both be massively scalable and traversable in both real-world and relative coordinate systems [5].

1.3 Purpose of Paper

The purpose of this paper is to provide an integrated approach to 3D pathfinding in a simulated environment by relying on the available tools and algorithms while optimizing and limiting the inherent weaknesses found in such a system. The focus of this paper will solely be on the optimization of the partitioning and traversal structure of an octree and its variants. This requires a focus more on simulation rather than real-world applications; therefore, while implementations have been done with UAV's, Robots, and Self-Driving Cars, their approaches only partially correlate to the specifications required by a simulation.

Furthermore, the paper will only look at sparsely populated 3D environments with orthogonally-moving vehicles. This will allow the paper to focus on specific variants of special partitioning for that type of environment. Optimization will be tested and analyzed through CPU performance and per-frame usage of the octree and octree variant methodology during pathfinding and navigation of *zeta* units. The intention is to use this paper for a follow-up study on Space-Based and Flight-Based simulations where the goal is to have an efficient and minimalistic pathfinding system with limited overhead.[1]

CHAPTER 2: REVIEW OF LITERATURE REVIEW AND ANALYSIS

[1] The development of this thesis began as proprietary work in the use of a game and was later separated from that work to exist outside of it as an extenuating system. Furthermore, much that was learned during the creation of this paper could not be included due to factors including proprietary information, not directly associated with the topic at hand, or specific to the programming language used. The last element holds the vast majority of information not included.

2.1 Octree

Octrees are primarily used, and have their foundation in, Geometric Modeling. The technique was developed by Donald Meagher in the early 1980's for the purpose of reducing computational resources when displaying and manipulating complex objects. Formerly, the more complex a model, the more complex computations required to display it. With the development of Octree Encoding, the goal was to create a more computationally efficient solution to display and manipulate 3D-Models [4].

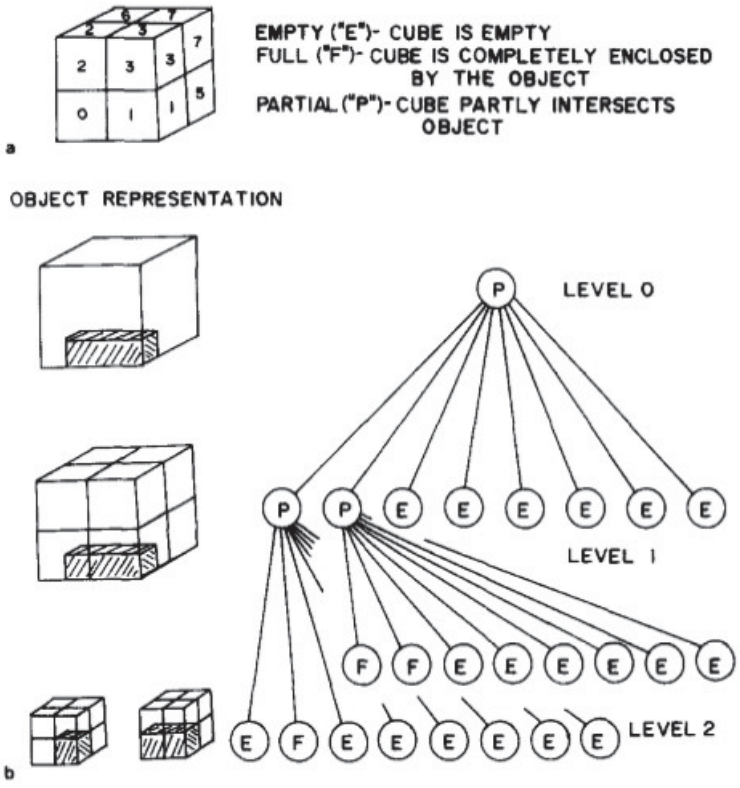


Figure 2: Octree Structure [4]

An octree is organized as a hierarchical tree with nodes denoting regional areas in a 3D space. As seen in *Figure 2*, octrees have a single root note with eight children per parent node. Nodes are visualized as primitive cubes that decrease in size exponentially per level. The highest level, level 0, is the size of the 3D space. Leaf nodes are nodes without any children. Nodes are

further defined as being EMPTY, PARTIAL, or FULL depending on how the node intersects with the positional data of objects in 3D space.

This is a brief overview of the concept of octrees. There are many variations and implementations of octrees and the above only discusses the structure of an octree. While the background will focus upon research concerning pathfinding and memory management, it is not the extent of octree research. They have been used in a variety of fields with the ability to quickly traverse nodes and find data point positional data optimally as factors in their applicability to multiple fields and implementation criteria. The ability for both factors is commonly known as Octree Encoding [4].

2.2 Octree Encoding

Octree Encoding has two parts: Indexing Order and Node Lookup. The Indexing Order of an octree is known as Morton Order when the data is stored linearly in a post-order, depth first traversal of the tree. The Node Lookup is the manner in which a node can be looked up in the tree's index.

Shown in *Figure 3* is, Morton Order, or Z-Order as it is sometimes called, traverses a grid hierarchy in a zig-zag fashion. In that manner, the nodes are then stored in a linear data structure [6].

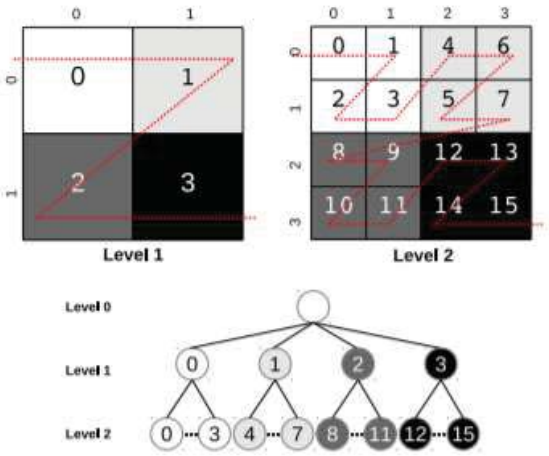


Figure 3: Morton Order [6]

To both store data and traverse neighbors, real-world data needs to be able to be converted into a representation that can easily look up the node that it is looking for. The result of the conversion process is called a location code. There are many ways in which to return a location code as it is only the result of a conversion algorithm in order to find the node in which the real-world data point lies; however, the most common way is described by *Christer Ericson in Real-Time Collision Detection.* Ericson is one of the primary sources for information referenced in this paper due to the breadth of octree variations discussed within his book.

Ericson describes Binary Interleaving as taking the vector coordinates of a node's positional data and interleaving the binary values of the three numbers into a 32-bit number. The positional data can either be fractional, depending on the node's range, or a real number. Ericson describes a fractional implementation. For example, the octree has a diameter in the x direction of 40 with values from 10 to 50. If the x coordinate for a node had the value of 25, the value would become .375 through normalization as (25-10)/(50-10) = .375. This then converts to binary which is 0.01100000. Interleaving it with the other two directional values of a data point's positional data returns an index values for the tree from the root node to a leaf node. For

example, the location code of a point could be 100111000 stating that it fell at index 4 on layer 1, index 7 at layer 2, and index 0 at layer 3 which is where it would be stored. The same process could be used for real-numbers without normalizing the coordinate values between 0 and 1[7].

World of Zero on *Youtube* had an excellent implementation to the algorithmic process by using a recursive methodology to find where a data point lay within an octree. As referenced in *Figure 12* in section *3.3.1.3,* the algorithm uses an iterative and recursive methodology to incrementally traverse the tree to look for the location code without expensive data conversion [8].

One of the more common implementations specific to pathfinding, involves point clouds. As real-world data is required for conversion, those implementations require a conversion to an octree representation for their data points. Broersen, et al., used a combination of the above two examples. A Bitwise AND function was used on each coordinate which was then interleaved to retrieve the index value where the data point would be stored [5].

2.3 Linear Octrees

Location codes can also refer to the node's lookup code within the Morton Order array discussed above [9]. An octree that contains its own location code is called a Linear octree as it is specifically an array-based methodology. Unlike the traditional pointer-based methodology, each node does not contain references to its children nodes; rather, it uses a Morton Order hash-based implementation to look up nodes by location code. For example, 000111000111 would reference index 0 – layer 1, index 7 – layer 2, index 0 – layer 3, index 7 -layer 4 [7].

Linear Octrees, and other memory management techniques, are important within the Scanning field as point clouds can be up to 1 billion points, and in some cases, have to be stored in 8 GB in memory. In one such example, Elsberg et al. came up with a solution that required

most data stored in a node structure to be computed rather than stored thus reducing the overall memory of each node. Typically, a node in a standard octree has an overhead of 100 bytes in RAM. This implementation decided against using a Linear Octree as node lookup has an *O(n)* overhead and they were able to come up with a solution that had *O(log n)* lookup. They did this by transferring the 64 bits required for child pointers to its parent and made the bit for each child a Boolean bit; therefore, only 8 bits were required to denote whether a child existed or not with the ability to look up children by parent [10].

2.4 Sparse Voxel Octree

Sparse Voxel Octrees are a variation on octrees with the goal of optimization of the octree so the system has the ability to process more data. Unlike octrees, Sparse Voxel Octrees, or SVOs', take into account memory management and efficiency [11]. In fact, octrees and SVOs are often used interchangeably.

This can be clearly seen with Ericson who does not mention SVOs but discusses many of the memory management techniques while Daniel Brewer in *3D Flight Navigation Using Sparse Voxel Octrees* discusses the same methodology but using SVO as the naming convention [7], [12]. For this paper, octrees will be known as data structures that implement the entire tree upon construction while SVOs prune the tree specific to static and dynamically inserted objects. The reasoning behind this distinction is to create a naming convention from a performance perspective for the testing and analysis of the paper; furthermore, it will allow the paper to show the efficiency of an optimized SVO against an edge case octree.

2.5 Loose Octree

A Loose Octree is a variant on a regular octree by accounting for dynamic objects. That is, objects that are not stationary which, for pathfinding, is a necessity. Data in an octree will

consistently update as objects move. In octrees, an object is within a node if its bounds entirely fit within the node. For dynamic objects, this can become an issue when it moves as it can "straddle a partitioning plane" [7].

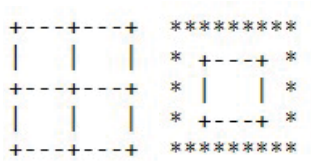


Figure 4: Loose Octree Design [13]

A Loose Octree will have inner and outer bounds for each node. The outer bounds of a node is twice the size of the original node's bound (inner bounds). The expansion of each node's bounds will cause overlap of the bounds into its neighbors' bounds.

Algorithm

1. Do{
    a. selectedNode = Find (minDist(zeta_position, current_childrenarray_positions))
    b. layer++;}
2. While (*zeta_radius* < ¼ of layer_scale_outer)

This states that the loop will continue until the object's radius is greater than 1/4th of the current layers' outer bounds scale. At that point, it will insert the object into the node closest to *zeta*'s current position. This will be done recursively. Another option would be to do a bottom up approach based on *zeta's* current position. This algorithm would make sure *zeta* was less than or equal to ½ the size of the layer's outer bounds. If it was greater than the stated size, *zeta* would be stored in the parent node; otherwise, it would be stored in the current node [13].

Loose Octrees allow for $O(1)$ insertions but can have some overhead as neighbor overlap can require more complex comparisons between objects [7].

2.6 Neighbor Traversal

There are two major points of emphasis required for pathfinding within an octree structure: Neighbor Traversal and Tree Pruning. Neighbor Traversal is a variation on the parent-child methodology for a tree's hierarchy but encompasses not just parents and children, but neighbors not associated within a specified hierarchy.

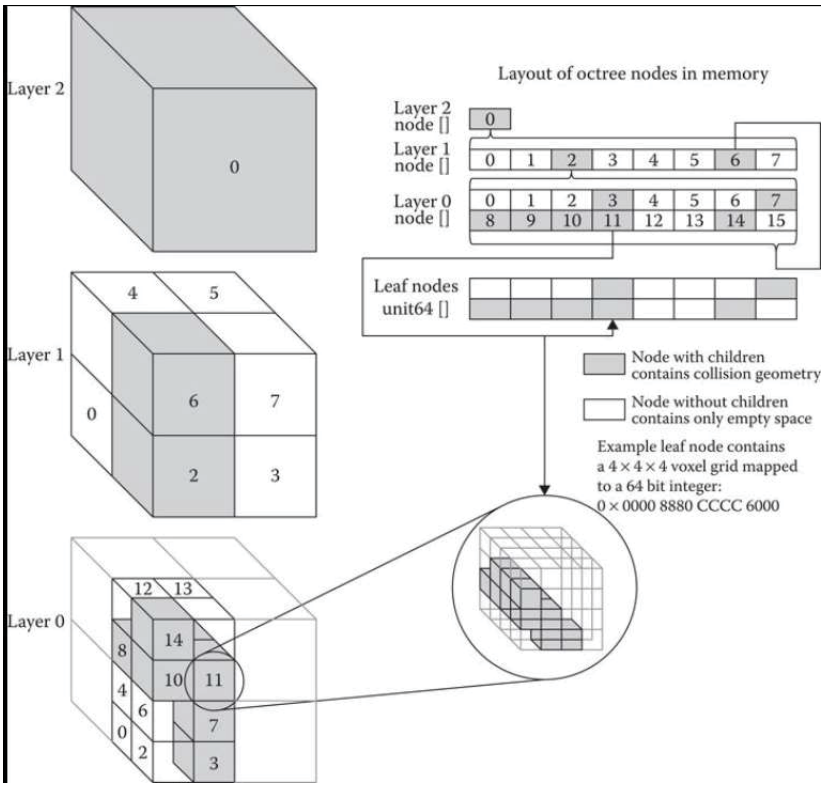


Figure 5: Brewer SVO Layout [12]

This is best explained by Daniel Brewer in *3D Flight Navigation Using Sparse Voxel Octrees.* His team used an A* implementation to traverse their octree representation with an orthogonally-moving vehicle. Neighbor links are created on a second traversal after the original octree has been instantiated (a static map was used for the environment). Links are created with neighbor nodes that share faces. For example, in *Figure 5*, 10 and 11 are neighbors as they share faces. That data is then stored in the node structure. If a node does not have a neighbor at the

same level, the node's neighbor is set to its parent's neighbor. This allows for connectivity of the entire tree [12].

A common application requiring neighbor traversal is Drone pathfinding. The process is very similar to that of a simulation; however, drones often work with point cloud representations instead of a physics-based system (visualization engine) used by a simulation [14].

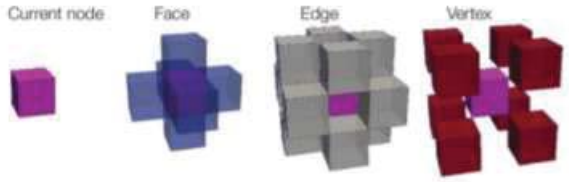


Figure 6: Sharing a common face (6), edge (12), or vertex (8) [14]

Xu et al. used real-world positional data to check for adjacency; however, their methodology relies heavily on the pathfinding methodology used (Octree State Lattice) which, while intriguing, is beyond the bounds of this paper as it pertains to a variant pathfinding methodology [15].

Kilic and Yalcin went in a different direction and used wave computing techniques which, while encompassing a directional traversal system, overwrote the cubic octree traversal methodology commonly used. Rather, as shown in the last two examples, it is the pathfinding methodology that apparently creates the traversal methodology [16].

As described in the *Introduction*, the rules of how a *zeta* unit can traverse are paramount to pathfinding; furthermore, the pathfinding methodology is the decision-maker in terms of how *zeta* units can traverse the octree and what is considered a neighbor.

As this paper will use A* for pathfinding as discussed below, Brewer's example is the most apparent reference and thus will be the foundation of the use case in the methodology.

2.7 Tree Pruning

Tree Pruning is not well-discussed in octree theory as it would only be useful in a specific set of circumstances such as dynamically-moving objects. As an object moves from one node to another based on its positional data, the octree has to be updated to notify all other *zeta* objects of the new structure. For this application, there is more material about implementation rather than optimization and, therefore, the former will be at the forefront of the discussion below

One application that has gained notoriety is *Growing and Shrinking.* In this methodology, rather than prune a branch or node of a tree when *zeta* moves, the octree is constructed around a *zeta* unit and moves with it [17].

Don Libes in *Modeling Dynamic Surfaces with Octrees,* presented a variant application of *Growing and Shrinking* which the paper states as *Expansion and Contraction.* Libes looked to solve the main problem of static octrees that they have defined bounds and the specified use case when the *zeta* unit explored an unknown location with unknown boundaries. His solution was to dynamically create a new root node above the current root node which would make the current root note one of the new root node's children. Every expansion would increase the size of the octree by eight. The shrinking operation would do the opposite; therefore, as the octree position is based on the location of the current root node, the octree has the possibility to move locations [18].

In his blog, *The Infinite Loop,* David Geier proposed a specified solution to the problem of dynamic *zeta* units not just dynamic worlds. Dynamic worlds pertain to the *Growing and Shrinking* solution above which has little to no overlap with this solution. His solution was to compile two different octrees for differentiating object type. One octree would contain static objects and the other would contain moving objects; therefore, when the tree updated only the moving object tree would update thus optimizing the overall application.

Geier's process requires a 1D-array of *zeta* units that have left the node that once contained them. For each *zeta*, traverse up the tree until the current node is able to contain the *zeta* unit. The same process occurs in the opposite direction if, at a certain point, the *zeta* unit is contained within a child node of it's current node. The node that once contained the *zeta* unit, and is now empty, can be removed along with its children. Geier uses a *Shrinking* methodology to prune the tree once able; however, he notes an issue with the methodology as all location codes have to be updated as the root node has been updated [19].

2.8 Research Question

The goals of this paper are two-fold. Primarily, this paper's aim is to analyze current octree methodology for 3D pathfinding while implementing integrated models to partition a sparsely-occupied 3D space for static and dynamic objects. The methodology that will be specifically analyzed for this aim includes tree pruning and tree traversal by attempting to optimize a spatial partitioning structure for 3D-Pathfinding purposes by structuring the data structure, in such a way, as to reduce traversal time and improving efficiency through optimal tree pruning and memory overhead. Optimization of the systems will be focused on the performance of the above methodology as a result. This paper will analyze the veracity of the current methodology by testing the implemented models in a variety of scenarios discussed in detail in the *Methodology* and *Testing* sections below.

2.9 Shortcomings of Current Methodology

The shortcomings of current methodology of octrees in relation to optimization are primarily due to the variety of fields and perspectives in current thought concerning octrees and their use in partitioning data. Foremost, there are two main bottlenecks in using octrees for 3D pathfinding: Neighbor Traversal and Tree Pruning. Neighbor Traversal is a more common

application, and therefore, there are numerous papers in numerous fields on the subject which, while beneficial, provide no specified understanding of the best approach for 3D pathfinding with multiple objects in a dynamic environment. On the other hand, dynamic tree-pruning is less common, and therefore, only a specialized segment of papers discuss the subject. Furthermore, only a few mention optimal tree pruning methodology and, therefore, options are limited especially due to the limitations of a simulation medium when most of the papers focus on real-world environments. This paper looks to implement and analyze specified solutions for both bottlenecks which is discussed, at length, in the *Methodology* and *Testing* sections below.

CHAPTER 3: METHODOLOGY

3.1 Technology

3.1.1 Unity

The Unity Game Engine is the primary tool used for the analysis in this paper. It is a popular visualization engine used to create 2D, 3D, VR, and AR games, as well as, applications for product, training, and marketing visualization in a variety of fields including, but not limited to, architecture, military, medical, and music. It has integrated tools and plugins which allow relative ease-of-use for developers and artists to produce wide-ranging applications [20].

Unity uses C# as the main language for its back-end. C# is an Object-Oriented language integrated with the .NET platform. One of the main focuses here is the use and allocation of Garbage Collection. Garbage Collection, on every instantiation of a new object, allocates memory for that particular object [21]. As more and more objects are instantiated, the overhead of garbage collection can be exacerbated.

The Unity Engine was chosen, in particular, for its visualization qualities. Unlike other IDE's, Unity has a focus on visualization and the integration of data with that visualization [22]. For testing and analyzing a sparse voxel octree this was important in ascertaining the quality and fundamental veracity of the created system.

3.1.2 Unity Profiler

The Unity Profiler is an analytics tool to aid developers with optimization. It breaks down per-frame usage and overhead in many areas including CPU usage which is pertinent to this paper's analysis [23]. As seen in *Appendix A*¸ the Unity Profiler for CPU Usage has a number of elements.

As discussed in the *Introduction,* this paper analyzes the overhead and usage of octree and octree variant methodology during traversal and navigation methodology. The Unity Profiler is a major tool in gathering the data for that analysis. It gives both a quantitative and qualitative understanding of the overall system and individual elements. Time ms is used in the quantitative analysis while GC Alloc, and Total % give a broader understanding of performance that, while not included in this paper, were great assets in the development of this paper. Total% states the total% of CPU being used by that hierarchy per-frame. GC Alloc is how much memory is being allocated for the hierarchy per-frame. Finally, Time ms shows the consumption of time for the hierarchy. This value is the same as Total % but in milliseconds, not percent [24].

3.1.3 Unity Gizmos

Unity Gizoms were be used to give a visual element for testing and analysis. It is a visual toolkit within the Unity Engine used to assist and debug development. Inherent functionality includes the ability to draw solid and wire cubes which are essential to the visualization process as seen in *Appendix B* [25].

3.1.4 Hardware Specifications

Testing was completed on an *Acer Predator ACG-710-70001 Gaming Desktop*. It has a 6th generation Intel Core i7 with 16 GB RAM and a NVIDIA GTX 970 Graphics Card. No other programs were running at the time of testing other than background windows applications.

3.2 Spatial Partitioning System Requirements

This research uses an integrated methodology from multiple areas to formulate a partitioning system that was optimal for a sparsely-populated 3D environment, as well as, for pathfinding. The core of the system includes the update and traversal methodology with optimization specifically for elevated and continuous use.

A modified Sparse Voxel Octree, or SVO, was used as the base as stated in the *Foundation* section above. As per the requirements of the paper, the SVO had a number of use cases that had to be fulfilled outside of its normal criteria.

1. The SVO needs to be able to track dynamic objects.
2. The SVO needs to be able to partially update when dynamic objects change nodes.
3. The SVO needs to be able to limit overhead through management of Garbage Collection and modifiable elements.
4. The SVO needs to be able to traverse its neighbors in an efficient manner.

These four use cases, while partially discussed in the background, are also mostly specific to the problem of the paper. Research on Scanning, specifically, is related to the problem at hand as per-frame scanning has very similar problems. Scanning requires many similar properties, the focus, at least with Elsberg's team, is insertion, deletion, and memory overhead efficiency while the focus of this paper is the required dynamic nature of the SVO. Elements taken from the paper that have been integrated into the methodology include thoughts on memory storage as discussed in the *Background* [11].

3.3 Process

The process to create a spatial-partitioning system relies on a few solid sources for a foundation. *World of Zero on YouTube* had an excellent You Tube video on recursion for Sparse Voxel Octrees, while Daniel Brewer, in his chapter titled "3D Flight Navigation Using Sparse Voxel Octrees", presented an overview on their methodology for a SVO for pathfinding in 3D-Space, and finally, *Christer Ericson in Real-Time Collision Detection* gave a great overview of Hierarchical Grid Methodology, as well as, some current research in the area of octrees [7], [8], [12].

While each source had its own methodology, none of them particularly fit, especially at the finer points, the goal of this paper. Therefore, the methodology of the paper began as an incremental exploratory project before a true algorithm could be formed. In this way, there were two different methodologies formed with one leading into the next. Both will be tested for efficiency and performance for static and dynamic objects using standard pathfinding methodology which will be discussed, at length, in the *Testing* section.

3.3.1 Octree

Even before research could begin into an SVO, an octree had to be constructed as an SVO is created from the pruning of an octree. To create the octree, a few decisions had to be made. There are generally two options to create an octree – a pointer-based version and an array-based version.

Pointer-based versions require indexing links to each node's children which has a higher memory overhead the more layers there are within a tree. Traditionally, octrees are pointer based as they do not necessarily require memory management; however, this paper will use the same memory management techniques, other than pruning, as to test the efficiency of that methodology clearly. To this, an array-based methodology was used – Linear Octrees [7], [11].

On octree construction, a Linear Octree was used as the base with a Heap Table as the data structure for storage. The octree was made of nodes with the above data which was limited as much as possible to reduce RAM required for each node.

3.3.1.1 Location Codes

Algorithm 1: Location Code Lookup - Child

Input: uInt key, uInt Child index
Output: uInt Child Key
1. Move key three bits to the left;
2. Bitwise OR function with uInt Child index;

Figure 7: Location Code Lookup - Child

Algorithm 2: Location Code Lookup - Parent

Input: uInt Child Key
Output: uInt Parent Key
1. Move Child Key three bits to the right;

Figure 8: Location Code Lookup - Parent

As referenced in *Figure 8*, the Location Code lookup is somewhat unique and geared to the purpose of dynamic octrees. Most octrees use a form of Morton Order binary interleaving to find the real-world positional data of a node [7].

It was observed that the implementation required for such a conversion added to the overhead of the system. Rather, as noted in *Figure 8*, a different, simpler approach was taken. The location codes for traversal were kept relative based on the node hierarchical data.

Specifically, each node extrapolated the location code of its children through bitwise operations. If, for example a child node was needed, a key would be formed by shifting the parent's key 3 bits to the left while shifting the child's key to the right 3 bits would be done to achieve the opposite. The former function requires the use of the bitwise OR function which only returns true if either is true [26].

For example, to find the key for the child at index 3 of the Root Node, there are a couple of steps. Foremost, there needs to be a flag bit so binary digits aren't left off; therefore, the key for the Root Node will always be 7 (0111) as this is the maximum number that can be allocated for an index. The operation to find the child node key is 7 << 3 | 3 which states move the parent's key 3 bits to the left and perform bitwise OR to append the child index. The incremental steps are as follow 7 << 3 = 56(111000) | 3 (0011) = 59 (111011). Therefore, the Location Code key

for the child node is 59. The reverse algorithm can also be applied to return the parent's key. 59 >> 3 = 7.

Both Node Size and Node Position fall under the same category as Location Code. Each can be included within a node or be calculated in real-time. To reduce memory, the latter was chosen for each option.

3.3.1.2 Node Size

The Node Size algorithm is fairly straightforward. Based on the depth of the Node, found by the key enumeration, the Node's size is scaled by multiplying the size of the octree by .5 ^ Node Depth. For example, the Root Node encompasses the entire octree; therefore, the root node is equal to the size of the octree. It's key length, and enumerated depth, would also be 1 as it is at the top of the tree. An offset is then used so inverse calculations can be more readable. The size of the octree is then multiplied by .5 to the offset. In this example, the size of the octree is 5. The node size is 5 * (.5^0) which is equal to 5. If we take one of the Root Node's children, its key length offset would be equal to 1. The child's size is 5 * (.5 ^ 1) equaling 2.5. Please refer to the Node Size algorithm in *Figure 9*.

Algorithm 3: Node Size

Input: uInt D, uInt Global Size
Output: uInt Size
1. Either Retrieve the Node depth from a stored value or by key enumeration;
2. Global Size $*$ $(.5^{D})$;

Figure 9: Node Size

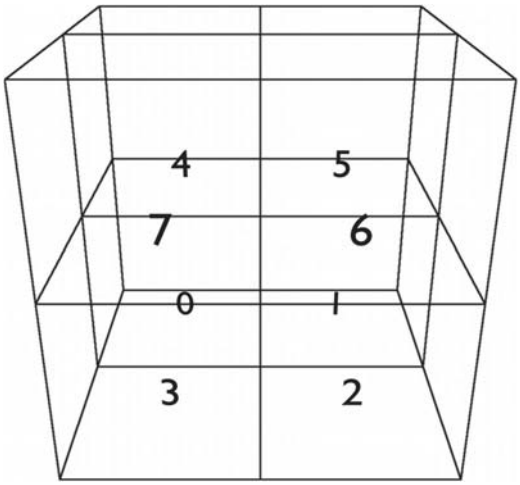


Figure 10: Octree Numbering [27]

Algorithm 4: Node Position

Input: float NodeSize, uint Index, Vecor3 Parent
Output: Vector3 Position
Step = NodeSize/2f/; Position = parent/;
if *Index Bitwise OR 1 == 1* **then**
  Position.x += step;
else
  Position.x += -step;
if *Index Bitwise OR 2 == 2* **then**
  Position.y += step;
else
  Position.y += -step;
if *Index Bitwise OR 4 == 4* **then**
  Position.z += step;
else
  Position.z += -step;

Figure 11: Node Position

The Node Position algorithm is sourced from *World of Zero on YouTube* and *Christer Ericson in Real-Time Collision Detection*. The Node Position algorithm uses the Node Size algorithm to get the step. The step is the radial offset in a given direction. Based on the value of the index (0-7), the Node's position, or center point, is offset using the *Figure 10* as a reference. The function uses Vector 3 addition with the bitwise AND function to find the new value.

Consider the entire octant, or cube, the parent node. Each parent has eight children which are offset from the parent given both half the size and radius. Refer to the Node Size example

from earlier. The Root Node's children have a size of 2.5 in comparison to 5 for their parent. The Root Node's position is at 0,0,0. The child at index 3 is a good example to review the algorithm's process.

The step is calculated as the node size / 2. In this instance, it equals 1.25. For the x value, the index is checked against 1. 0011 is the binary equivalent for 3 and 0001 is the binary equivalent for 1. With the AND function, the operation equals 0001 or 1. As that is true, the x value equals 0 (Root Node x value) plus 1.25 equaling 1.25. The y and z values are checked against 2 (0010) and 4 (0100), respectively. The results for each are 2 (0010) and 0 (0000). Therefore, the y value equals 1.25 and the z value equals -1.25.

3.3.1.3 Closest Index Position

Algorithm 5: Closest Index Position

```
Input: Vector3 Data, Node Current
Output: uInt Index
Index = 0
 if Data.x > Current.Position.x then
 |  Index +=1;
else
 |  Index += 0;
if Data.y > Current.Position.y then
 |  Index +=2;
else
 |  Index += 0;
if Data.z > Current.Position.z then
 |  Index +=4;
else
 |  Index += 0;
```

Figure 12: Closest Index Position

The reverse of the Node Size function is useful is some circumstances like traversal which will be discussed in subsection *3.3.1.4*. The Closest Index algorithm looks for the closest index position to the inserted vector 3 position among a node's children. This is done by using the bitwise OR in conjunction with the positional data inserted into the algorithm. The below example will use the output data from the example of the Node Size algorithm.

This algorithm requires both a lookup position for entered data, as well as the node position. Using the Root Node, the node position is at (0,0,0). I will use the node positional data of the child at index 3 as the lookup position. Please note, the lookup position can be any vector 3 data within the bounds of the octree. It is not required to be an exact match to a child's positional data. This example is using it only for clarity.

A vector 3 needs to be created in reference to the offset. 1, 2, and 4 are returned for x, y, and z, respectively, if the values of the lookup positional data are greater than the values of the node positional data for that specified direction. Otherwise, 0 is returned if the lookup positional data is less than the node's positional data. Based on the lookup data of (1.25, 1.25, -1.25) against (0,0,0), (1,2,0) is returned. The bitwise OR function is used on each vector 3 component. The incremental steps are as follows: 0000 | 0001 = 0001 | 0010 = 0011 | 0000 = 0011 = 3. This states that the closest child node to the lookup position would be the child node at index 3.

3.3.1.4 Recursive Traversal Initialization

Algorithm 6: Recursive Traversal Initialization - Octree

```
Input: Node Current
Key = GetKey(Current);
Position = GetPosition(Current);
Size = GetSize(Current);
if Current.Children ! Instantiated then
    Instantiate(Children (8));
else
    continue;
if Current.Depth <= Global Depth then
    Visit(Children (8));
else
    return;
```

Figure 13: Recursive Traversal Initialization - Octree

The Recursive Traversal algorithm for the octree is relatively straightforward. Foremost, the Root Node needs to be instantiated as discussed above. The size, position, key, and other information required need to be either calculated or stored. Then, it loops through each of its

children and instantiates them. Each node is stored in a global hash table and sorted by Morton Order which is the natural order of a recursive, top-down search of an octree. The methodology calls itself to instantiate all nodes in the octree until the stated depth of the tree has been reached. For example, if the stated depth is 7, 299,593 nodes would be created. The number of nodes in an octree can be calculated using d^n *-1 / d-1* [7].

There are three distinct functions required to integrate spatial partitioning into the octree. Insertion, Removal, and Update. Please note that while these functions are similar to an SVO, some of that functionality has been curtailed as it is not required by an octree.

3.3.1.5 Insert Obstacle

Algorithm 7: Insert Obstacle

```
Input: Node Current, Obstacle obstacle
if obstacle.size > Current.size * 2;
 then
    obstacle.key = Current.Parent.key;
    Current.Parent.AddObstacle(obstacle);
else
    Index = GetClosestIndexPosition(obstacle.data, Current.data);
    if Node.Depth +1 <= Global Depth;
     then
        InsertObstacle(Current.child[Index];
    else
        obstacle.key = Current.key;
        Current.AddObstacle(obstacle);
        Current.SetBitArray(Current.Index, 1);
```

Figure 14: Insert Obstacle

An obstacle is always inserted at the top of the tree which is recursively traversed until the best position for the obstacle is found. One of the most important parts of an octree is data integration. The tree needs to know where obstacles are located within the tree from any node. This is handled by using a bit array where 0 at the child index means there is no obstacle along that hierarchy (even among that child node's children) while 1 means that somewhere along that child node's hierarchy, there is an obstacle.

To implement this functionality, the parent of the node traversed in *Figure 14*, has the bit at the index associated with the node set to 1 as it is known that the current obstacle will be placed within the node's hierarchy.

The next step integrates a Loose Octree implementation into the functionality. Refer to the *Background* for more information on Loose Octrees. In this situation, the obstacle size is stored and is checked if it is greater than two times the size of the current node. This is what is considered the outer bounds of the node. If the obstacle is such a size, then the parent of the current node is destined as the obstacle's location.

If this is not the case, continue down the tree by searching for the closest index based on positional data using the *Closest Index Position* algorithm until either the Loose octree effect comes into play or the depth has been reached. If the later occurs, assign the obstacle to the current node.

3.3.1.6 Remove Obstacle

Algorithm 8: Remove Obstacle

Input: Obstacle obstacle

```
obstacle.node.Remove(obstacle);
```

Figure 15: Remove Obstacle

The removal algorithm for octrees is quick. As nodes do not have to be updated or removed for pruning, the obstacle only needs to be removed from the Hashtable associated with the node it was stored in.

3.3.1.7 Change Obstacle Position

```
Algorithm 9: Change Obstacle Position
  Input: Obstacle obstacle
  if Distance(obstalce.position, obstacle.node.position) > obstacle.node.size;
   then
     RemoveObstacle(obstacle);
     InsertObstacle(obstacle, rootNode);
  else
     continue;
```

Figure 16: Change Obstacle Position

The *Change Obstacle Position* algorithm is just as simple in its implementation. As there is no need to update or remove nodes, the steps are more straightforward. When an obstacle changes position, it notifies the octree it is associated with. If the distance between the new position and the node it is located at is greater than the radius between the node center and outer bounds of the node, the obstacle will be removed from the current node and the *Insert Obstacle* algorithm will be run.

3.3.2 Sparse Voxel Octree

The Sparse Voxel Octree methodology is very similar to the octree methodology with a few minor changes. Foremost, not all nodes are inserted on initialization. This is to preserve memory as well as allow for less nodes required to traverse during A*.

3.3.2.1 Insert Obstacle

```
Algorithm 10: Insert Obstacle - SVO
  Input: Node Current, Obstacle obstacle
  if Current.Children ! Instantiated AND Current.Depth < Global Depth ;
   then
    | Instantiate(Children (8));
  else
    | continue;
  if obstacle.size > Current.size * 2;
   then
    | obstacle.key = Current.Parent.key;
    | Current.Parent.AddObstacle(obstacle);
  else
    | Index = GetClosestIndexPosition(obstacle.data, Current.data);
    | if Node.Depth +1 <= Global Depth;
    |  then
    |  | InsertObstacle(Current.child[Index];
    | else
    |  | obstacle.key = Current.key;
    |  | Current.AddObstacle(obstacle);
    |  | Current.SetBitArray(Current.Index, 1);
```

Figure 17: Insert Obstacle - SVO

Nodes are only inserted during the *Insert Obstacle* methodology as seen in *Figure 17*. A node inserts all of its children if it is visited by the obstacle. This is because it is known that the obstacle is traversing the hierarchy, as well as, it preemptively creates a series of leaf nodes for the hierarchy.

3.3.2.2 Recursive Removal

```
Algorithm 11: Recursive Removal
  Input: Node node
  if node.obstaclesCount == 0 then
    | if node.Depth <= GlobalDepth +1 AND node.Depth > 1 then
    |  | node.parent.SetBitMask(node.index, 0);
    |  | if node.GetBitMask() == 0 AND RecursiveCheckForObstacleIs(Node.children) == 0 then
    |  |  | Tree.Remove(node.Children);
    |  |  | RecursiveRemoval(node.Parent);
    |  | else
    | else
  else
```

Figure 18: Recursive Removal

The second difference, and more complex change, is the recursive removal required to prune the tree when a hierarchy no longer contains an obstacle. When an obstacle is removed, the

tree checks the hierarchy the obstacle used to be in for pruning. This algorithm is a variation of the *Growing and Shrinking* methodology discussed in the background. Rather than altering the bounds of the SVO, only branches are pruned thus maintaining the overall bounds of the tree. The purpose of this was to enable both visualization of the methodology, as well as, test more effectively with the standardization of the tree structure.

Foremost, *Recursive Removal* algorithm begins at the prior node the obstacle just left and traverses up the tree looking for areas where it can be pruned. If the node's children contain obstacles within their hierarchies or the node, itself, contains an obstacle, no pruning occurs. If the node's children do not fall under the stated parameters, they are removed. If the node's children can be removed, the method is called for the parent with the same request. This process continues until the Root Node is reached. I am of the opinion that while this methodology works and is optimal, it can be further optimized from a bottom-up approach. This will be a topic to look at for Future Work.

*3.3.3 A**

The A* methodology used is a standard, out-of-the box methodology. This is done specifically to test the applicability of one of the more common pathfinding methodologies. The methodology used references *Sebastian Lauge on YouTube* and documentation on the *GeeksforGeeks* website. Within the A* methodology, there are optimizations on the SVO to improve traversal for pathfinding, as well as, efficiency of updates [28], [29].

The A* algorithm used for implementation is explained in a generalized format. The first step is to find the start and end node. For this implementation, the start node will always be the obstacle's current location node which is handles by the SVO.

3.3.3.1 End Node

```
Algorithm 12: Find End Node
Input: Node Current, Vector3 position, float size
Output: Node destination
if size > Current.size * 2;
 then
    return Current.Parent.key;
else
    Index = GetClosestIndexPosition(position, Current.data);
    if Node.Depth +1 <= Global Depth;
     then
        FindEndNode(Current.child[Index], position, size;
    else
        return Current.key;
```

Figure 19: Find End Node

The End Node has its own algorithm which uses the *Insert Obstacle* algorithm without any insertions or bit changes as it looks for the destination node.

Upon finding the End Node, the Start Node is instantiated with the G and H values. The G value is the node's distance from the start point (0 for start node) while the H value is the node's distance from the End Node. The Start Node is then added to the Open List.

At this point, the algorithm either stops as the End Node location key is equal to the Start Node location key meaning there is nowhere to travel, or the algorithm starts the incrementation process.

Before the implementation can be discussed, three octree processes have to be explained.

3.3.3.1 Directional Key

Algorithm 14: Directional Key

Input: AStar Path, Node Current, Vector3 Direction, float size, float globalsize
Output: uint Key
Position = Current.size * Direction + Current.position;
if *Distance(Position, RootNode.Position) > globalsize* **then**
    return 0;
else
    return KeyByPosition(star, RootNode, Position, Current.Depth, size);

Figure 20: Directional Key

The *Directional Key* algorithm checks if a node has a neighbor in an orthogonal direction – Up, Down, Left, Right, Back, or Front. This information can be found by multiplying a directional vector such as (1,0,0) to check Right. The directional vector is then multiplied by the size of the node and offset by the node position to find the location where a neighbor to the right would be if it existed. A check is made to make sure the checked position remains within the outer bounds of the octree. If the check passes, the *Key by Position* algorithm is called to request the location code of the neighbor.

3.3.3.2 Key by Position

Algorithm 13: Key By Position

```
Input: AStar Path, Node Current, Vector3 position, uint nodedepth, float size
Output: uint key
key = 0;
if size > Current.size * 2;
 then
    key = Current.Parent.key;
else
    if Current.depth <= nodedepth then
        if Current.depth <= GlobalDepth then
            Index = GetClosestIndex(position, Current.position);
            if Tree Contains KeyAtIndex(Index) then
                key = KeyByPosition(star, keycode, position, nodedepth, size);
            else
                key = Current.key;
        else
            key = Current.key;
    else
        key = Current.parent.key;
if key ! = 0 then
    if key.IsOccupied() then
        key = 0;
    else
else
```

Figure 21: Key By Position

The *Key by Position* algorithm returns a given key in a specified direction based on the *Directional Key* algorithm. The *Directional Key* algorithm denotes a direction and the *Key by Position* algorithm recursively searches for that key. A check is made for the Loose Octree to make sure the requested position is within the bounds of the outer node. If the check fails, the key is set to the current node's parent key. Otherwise, a check is made again but this time to make sure that the node is less than or equal to the node checking its neighbors' depth. If the check fails, the key is set to the parent's key.

The purpose of this is to limit the number of nodes that can be traversed. While traditional octrees allow traversal from high-to-low and low-to-high it is not necessary in this case as the obstacles inserted will likely be at the lowest depth they can possible be at upon

insertion; therefore, traversal will only occur at its level or higher levels reducing the overall nodes that need to be traversed.

If the check passes, another check occurs. This time to make sure the node is less than or equal to the depth of the octree. If the check fails, the key is equal to the node's key. Otherwise, the *Closest Index Position* algorithm is used to find the best index to traverse. If the child node does not exist, the key equals the node's key; otherwise, continue down the tree with the child node and look for the best position for the neighbor node.

Once the recursion has been completed, a final check occurs. If the node selected as a neighbor has an obstacle count greater than 0 or any of its children have occupation within its hierarchy, the node is not traversable.

3.3.3.3 Traverse Neighbors

The *Traverse Neighbors* algorithm is a conglomeration of the previous two algorithms. It calls the *Directional Key* algorithm six times for each orthogonal direction and returns the results to the A* requester. If a value equals 0, that means the node in that direction cannot be traversed.

3.3.3.4 A Incremental*

```
Algorithm 15: A* Incremental
if OpenList.Count > 0 then
    CurrentNode = OpenList[0];
    F = CurrentNode.H + CurrentNode.G;
    G = 0;
    H = Distance(CurrentNode.Position, EndPosition);
    while Dictionary Open do
        IncrementElement++;
        ElementF = Element.G + Element.H;
        ElementF /= Element.Size;
        if FCost <= F AND Element.H < H then
            CurrentNode = Element;
            F = FCost;
            G = Element.G;
            H = Element.H;
        else
    OpenList.Remove(CurrentNode);
    ClosedList.Add(CurrentNode);
    if CurrentNode != EndNode then
        Neighbors = TraverseNeighbors(CurrentNode);
        OpenList.Add(Neighbors);
        if Neighbors.instance.G > CurrentNode.G + Distance(CurrentNode.Position,
          Neighbor.instance.Position then
            Neighbor.instance.G = CurrentNode.G + Distance(CurrentNode.Position,
              Neighbor.instance.Position;
            Neighbor.instance.H = Current.H + Distance(Neighbor.instance.Position, EndPosition);
            Neighbor.instance.Parent = CurrentNode;
        else
    else
        Path = ReversePath(EndNode);
        InitializePath(Path);
else
```

Figure 22: A* Incremental

The A* algorithm is set up to have two major checks. Primarily, the algorithm continues while the current node is not equal to the End Node. The algorithm continues as long as the prior hasn't occurred or the open list has nodes remaining within it. If there are no longer any nodes remaining to traverse, then the algorithm cannot find a destination to the End Node.

The current node is always set to the first element in the open list for standardization purposes. It's F (G+ H), G, and H values are stored as well. Looping through all open nodes, each node has its F calculated, then normalized, by dividing the F by the size of the node. This is

to make the F value lower for larger nodes as traversing larger nodes limits the number of nodes required to traverse to complete the path. Refer to *Key by Position* for another example of this.

If the node being checked is not equal to the current node, the F value of the checked node is less than the current node's F value, and the checked node's H value is less than the current node's H value, set the checked node to the current node.

Once the loop has completed, remove the current node from the open list and add it to the closed list. Check that the current node is not equal to the End Node. If the check passes, find all neighbors of the node using *Traverse Neighbors.* For all keys that are not equal to 0 and are not within the closed list, check their cost values. If the neighbor node hasn't been added to the open list, add it; otherwise, if the neighbor node's G value is > the current node's G value + the distance between the neighbor node and the current node, make the current node the parent of the neighbor node and set the neighbor's G value to the *Figure 22* and the H value to the current node's H value + the distance between the neighbor node and End Node.

If the current node is equal to the End Node, initialize the path by reversing it. Starting from the End Node, find the parent of each node recursively until the Start Node is found and reverse the created list. In the manner of the movement system used, run the script to start the system.

CHAPTER 4: TESTING

Testing was completed using three different, overall tests on both the octree and SVO methodology for analysis. The goal of the testing phase is to ascertain the scalar increase in efficiency with the optimized SVO, the overall efficiency of the SVO, the efficiency of the SVO during pathfinding, and the efficiency of the SVO during movement.

Tests were run inside of the Unity Engine and use both visualization elements, as well as, the Unity Profiler for analysis. Both the octree and SVO will have a standardized size of 50 and a depth of 4. This means that the overall size is 500 metric (meters) units and there are five potential layers within each hierarchy.

Two major scenario types were used with each test: Wall and Random. The Wall test will use arrays of 64 cubit, static obstacles in a series of 10 layers to form a wall that bisects the tree. The purpose is to visually ascertain the quality of the pathfinding as well as provide a differentiation of nodes. Due to the standardization of the octree and SVO information, this scenario will create the same tree structure each time. The Random test is a more realistic representation of a simulation and, perhaps, suits an SVO's purpose better. 64 cubit, static obstacles are randomly placed within the tree. While this creates a different tree structure each time, the purpose of this test is to look more at the traversal methodology in a complex environment in comparison to a grouped structure where the open spaces are relatively clear.

There are a number of visualization aides to help with the testing process. *Appendix C* is a good example of a simple tree pruning test based on *zeta* location while *Appendix D* contains examples of the Random and Wall tests, as well as, traversal and navigation iteration plugins.

4.1 RAM Overhead

The RAM Overhead test is the most straightforward of the tests. Using both scenarios, the test will look at the required overhead for running each test. This test was taken from Elsberg et. al and their attempts to reduce the overall storage size of octrees in RAM. They referenced the length of the byte array of an octree as its size in memory [10], [11]. The test will use serialization to retrieve the byte array of each tree. The length of the byte array shows the current memory required to store the tree in RAM.

4.2 Traversal Efficiency

The Traversal Efficiency test will look at the algorithm designed for pathfinding methodology – specifically A* pathfinding. This test was extrapolated from Daniel Brewer's research, specifically his presentation "Getting Off the Navmesh Navigating" at *GDC 2015.* He used a variety of visualization testing techniques which were taken to make an analytical test on the efficiency of traversal [12], [30].

Refer to *Figure 22* for the A* algorithm. Using both tree methodologies and both tests, the following tests will be run – 1, 3, 5, 10, and 15 *zeta* pathfinding tests from random starting points to random destinations. If a destination is occupied or there is no logical path, the *zeta* instance will not complete the traversal and the resulting failure will be added to the results. In the likelihood the CPU cannot run a test for the octree, this will be noted in the results. Dynamic obstacle avoidance is not included within the testing apparatus and, therefore, *zeta* instances may have intersecting traversals.

The test series used the Unity Engine, .Net Stopwatch, and the Unity Profiler to gather results as it looks at milliseconds per step, overall milliseconds, number of steps, garbage collection per

step, and overall garbage collection. Each test will be run 100 times. If a test cannot be run by the octree, the test will not be run each time to test that specific case.

4.3 Updating Efficiency

The Updating Efficiency test looked at the Tree Pruning algorithm. Please refer to *Recursive Removal* algorithm for more information. The same information that applies to the *Traversal Efficiency* section applies to this test series. In fact, the initialization of each test will require the successful completion of a traversal test. The resulting waypoint navigation from path start to path end will request an update to the tree based on current positional data. The resulting positional data change will then change *zeta's* insertion point and prune the tree where *zeta* last was. Refer to *Appendix C* for an example of this.

This test was created specifically for this paper and was not found nor extrapolated from any other source as there were limited references to theoretical papers designed with a pruning methodology that had static bounds.

This test series will also use Unity Engine, .Net Stopwatch, and the Unity Profiler to gather results but will only look milliseconds per frame, garbage collection per frame, and percent of system usage per frame. Each test will be run 100 times. If a test cannot be run by the octree, the test will not be run each time to test that specific case.

CHAPTER 5: RESULTS AND ANALYSIS

5.1 Ram Overhead Test Results and Analysis

The Ram Overhead test was run with the idea to specifically compare the overhead caused by initialization of the octree and SVO methodology. The data was separated into layers to analyze an iteration's overhead from an in-memory and usage standpoint.

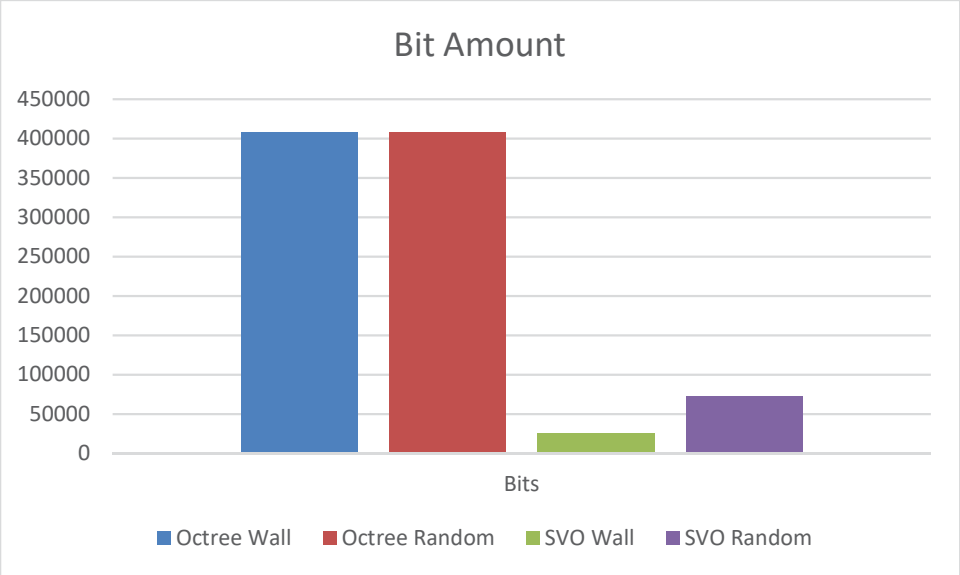


Figure 23: Bit Amount

Figure 23 denotes the Bit Amount for each iteration. This graph shows the in-memory overhead from an iteration. That is, RAM overhead that will directly affect CPU performance. As this test was run without the implementation of *zeta* objects, the in-memory RAM denotes only Map overhead which gives a standard to work from.

Naturally, the octree iterations are quite large and equal within memory as an octree initializes all nodes even if there is no data contained within the nodes. This was, however, not the interesting facet of this test. Remarkably, even though there were nearly ten times as many

obstacles (500 – 64) in the Wall test, the Random test for the SVO had a higher in-memory overhead of 72729 bits to 26097 bits. The supposition based on the results is that the spread-out nature of the Random test, while having less obstacles, required more nodes due to the higher-level hierarchies that had to be created in more than one area. In respect to the comparison between SVO and octree, in-memory, there really is no comparison. The SVO is vastly superior and as the focus of the SVO is to work within sparsely populated environments, it will never reach the in-memory overhead of an octree if used correctly.

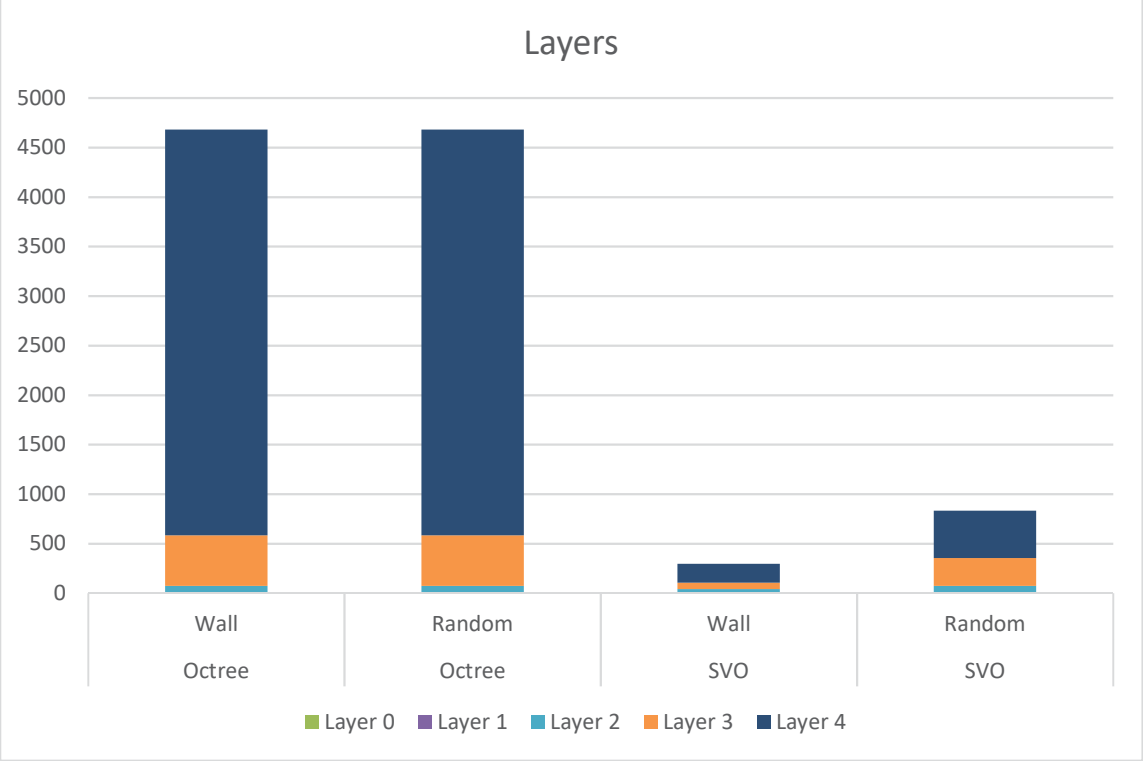


Figure 24: Layers

The second part of the test analyzed the number of nodes, grouped by layer, instantiated per iteration. Please note the layer differentials in layer 2 for the SVO Wall and SVO Random or refer to the raw data in the *Appendix K.* The Random test had double the number of active nodes within this layer which exponentially increased the number of nodes further down the tree. The Random test ended with more than double the number of nodes than the Wall test. Another

observation that can be examined from this data is that there are, on average, 87 bits required per Node. This is not too bad but, I believe, with further optimization can be reduced to reside within a 32 – 64-bit range. One way to implement this change would be to alter the uInt variables from uInt (32 bit) to uInt16 reducing the overhead for those factors by half; however, this would also restrict the size and depth of the system as a 32-bit number, within this system, can contain 32 layers while a 16-bit number can only contain 16 layers within a system.

5.2 Traversal Efficiency Results and Analysis

Over forty tables and graphs were compiled from the raw data results of the tests for this paper. Naturally, not all of these results can be discussed in a focused manner within this paper; therefore, only the most interesting or focal points to the analysis will be discussed and shown. Furthermore, individual tests *will not* be compiled into a general analysis of a test type. For example, the Random Octree test with 1 *zeta* will not have its data compiled with the 3, 5, 10, and 15 test results to give an overall analysis because each test has differentiating factors due to the alteration of units and map type. These factors will be discussed in the following two sections. Refer to *Appendix K* for the full compilation of the results.

The Map type analysis refers to Random and Wall tests for both the octree and SVO. Foremost, the overall extenuating data will be analyzed as there needs to be a generalized understanding of both the efficiency and flaws of each design.

1 Octree Random Astar	
Min of Ellapased Milliseconds	4
Max of Ellapased Milliseconds	591
Average of Ellapased Milliseconds	240.11
Standard Deviation of Ellapsed Milliseconds	155.3207106
Min of Traversals	6
Max of Traversals	3725
Average of Traversals	1573.11
Standard Deviation of Traversals	1030.481097
Min of Total Frames	0
Max of Total Frames	1
Average of Total Frames	0.94
Standard Deviation of Total Frames	0.238683257

Table 1: 1 Octree Random Astar

1 Octree Wall Astar	
Min of Ellapased Milliseconds	20
Max of Ellapased Milliseconds	554
Average of Ellapased Milliseconds	247.03
Standard Deviation of Ellapsed Milliseconds	150.1772996
Min of Traversals	35
Max of Traversals	3625
Average of Traversals	1552.42
Standard Deviation of Traversals	992.2934987
Min of Total Frames	0
Max of Total Frames	1
Average of Total Frames	0.96
Standard Deviation of Total Frames	0.196946386

Table 2: 1 Octree Wall Astar

Both octree tests seem to be relatively similar in all aspects. This should be related to the first test run concerning RAM Overhead. As all nodes have been instantiated, the map type likely will not have an effect on the overall results which, depending on the environment, can be beneficial or an exacerbated problem.

As for the actual results, the data is more variant than expected or desired. Both the average number of traversals and average elapsed milliseconds are higher than desired as well, even though, the number of frames required for such high averages does not seem to have any frame lag. In some cases, the Traversal methodology was quick enough to be completed before a

frame was rendered by Unity resulting in the minimum number of frames for this test to be 0. This was a rare occurrence as the majority of data samples required a frame as seen by the small difference between the average and the maximum number of frames. If there was any lag on the real-time environment, the process would have had a larger standard deviation of total frames and a higher average. Due to the concern that this may be a factor with the increase in *zeta* units, an analysis for the 15 *zeta* units for both tests was included in *Tables' 3 and 4.*

15 Octree Random Astar	
Min of Ellapased Milliseconds	4
Max of Ellapased Milliseconds	4748
Average of Ellapased Milliseconds	1981.055333
Standard Deviation of Ellapsed Milliseconds	1221.767926
Min of Traversals	2
Max of Traversals	3788
Average of Traversals	1556.238667
Standard Deviation of Traversals	1019.081339
Min of Total Frames	0
Max of Total Frames	1
Average of Total Frames	0.962
Standard Deviation of Total Frames	0.191259998

Table 3: 15 Octree Random Astar

15 Octree Wall Astar	
Min of Ellapased Milliseconds	0
Max of Ellapased Milliseconds	4629
Average of Ellapased Milliseconds	1945.08
Standard Deviation of Ellapsed Milliseconds	1208.23054
Min of Traversals	2
Max of Traversals	3789
Average of Traversals	1535.972
Standard Deviation of Traversals	1010.854987
Min of Total Frames	0
Max of Total Frames	1
Average of Total Frames	0.954
Standard Deviation of Total Frames	0.209554946

Table 4: 15 Octree Wall Astar

Both of these results tables don't show that much differentiation in comparison to the 1 *zeta* tests. The standard deviation of total frames held steady, for the most part, with just a slightly higher average than the 1 *zeta* tests. While this is good news for the overall system, there seems to be an underlying issue as the elapsed milliseconds standard deviation grew larger. More testing will need to be done to look at this issue as, from the current results, it is not apparent what could be causing it other than the inherent increase in processes at the same time.

1 SVO Random Astar	
Min of Ellapased Milliseconds	1
Max of Ellapased Milliseconds	67
Average of Ellapased Milliseconds	27.13
Standard Deviation of Ellapsed Milliseconds	13.15383341
Min of Traversals	8
Max of Traversals	280
Average of Traversals	121.34
Standard Deviation of Traversals	73.46918821
Min of Total Frames	0
Max of Total Frames	1
Average of Total Frames	0.83
Standard Deviation of Total Frames	0.377525168

Table 5: 1 SVO Random Astar

1 SVO Wall Astar	
Min of Ellapased Milliseconds	0
Max of Ellapased Milliseconds	22
Average of Ellapased Milliseconds	15.86
Standard Deviation of Ellapsed Milliseconds	5.388015133
Min of Traversals	2
Max of Traversals	209
Average of Traversals	33.98
Standard Deviation of Traversals	29.27369281
Min of Total Frames	0
Max of Total Frames	1
Average of Total Frames	0.87
Standard Deviation of Total Frames	0.337997669

Table 6: 1 SVO Wall Astar

The SVO tests, on the other hand, provided more fruitful data. The efficiency here was more palpable than the octree tests and, clearly, provided better results. *Tables' 5 and 6* will be for reference throughout this section; as certain elements are analyzed, charts will be used in addition to compiled results tables.

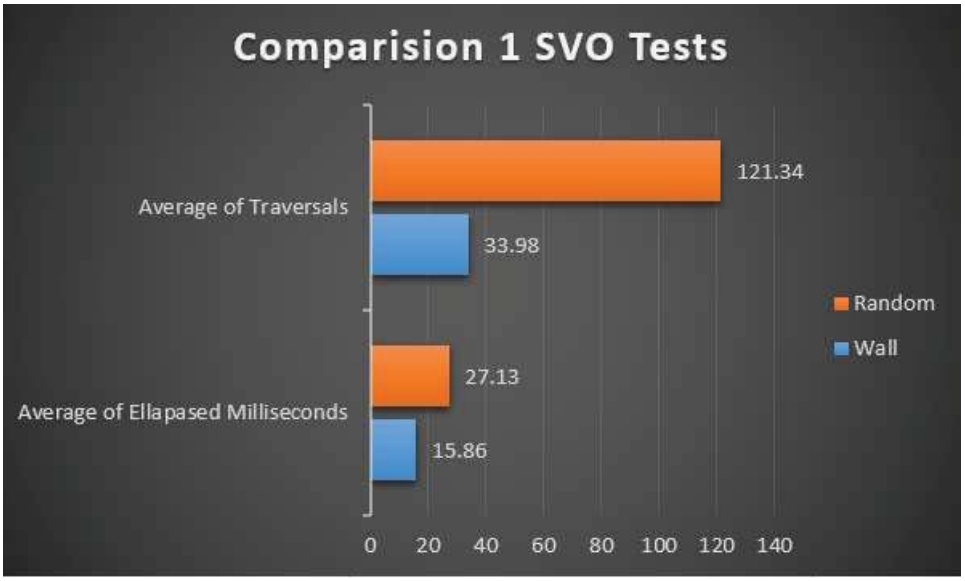


Figure 25: Comparison 1 SVO Tests

As seen in the *Figure 25*, the comparison of the Wall and Random tests for 1 *zeta* unit provide some more tangible data. The clearest marker for the efficiency of the traversal methodology is noted by the limitations of both the number of traversals and elapsed milliseconds of the tests in comparison to those of the octree. While the Random test has quite an increase in the average of traversals required, the milliseconds required shows a more limited average. This shows that the methodology can scale effectively as seen in *Figure 26*.

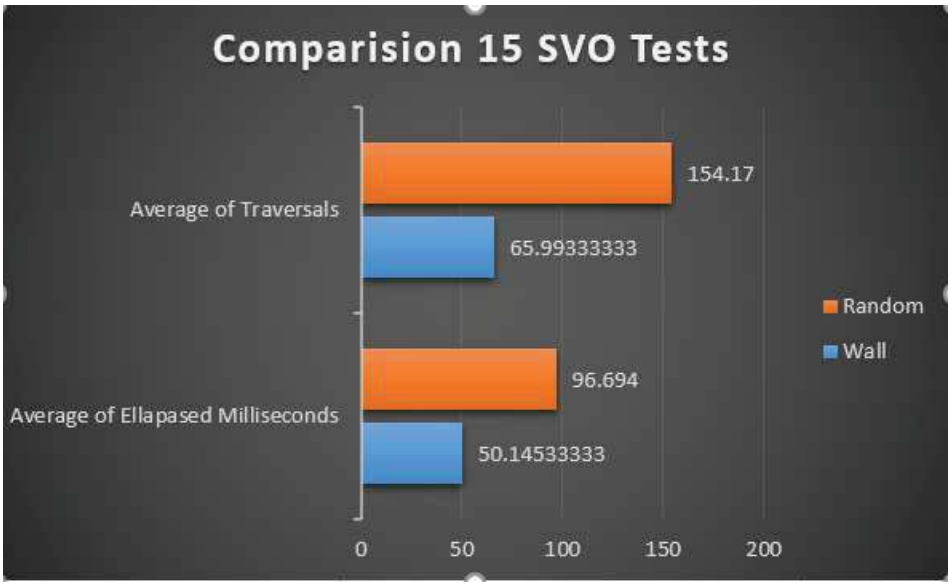


Figure 26: Comparison 15 SVO Tests

The average number of traversals required appears to have a strong correlating link between with the number of units in the system. Please note the number of traversals, on average, for the wall increased two-fold as units were in differing parts of the map which, as a result, implemented more nodes.

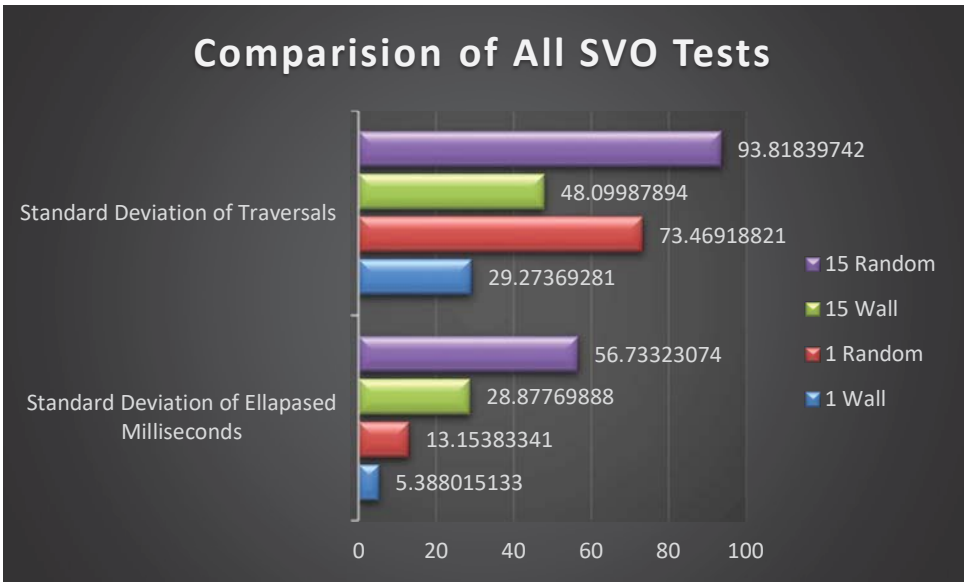


Figure 27: Comparison of All SVO Tests

Figure 27 is mostly a rehash of already stated information. One new pieces of analysis that is gleamed is the standard deviation of the 1 Random test. It is much higher than the 15 Wall test and just below the 15 Random test. This can be considered preliminary evidence that the more random a map is, the more inefficient the system is from a traversal standpoint without a consideration for the number of units; however, the efficiency of the system does not seem to consider the variance of the number of traversals as it pertains to this data. Rather, it appears that the map type and number of units plays a large role in the efficiency from a time perspective.

5.3 Updating Efficiency Results and Analysis

The Updating Efficiency tests show compiled data from the completed paths of *zeta* units as a result of the Traversal Efficiency tests in the section above. The data allocated here will be analyzed specifically to find the overhead of the pruning updates to the tree. It is noted that the octree does not use the pruning algorithm. The time required for path completion for octrees will be used in comparison to those of the SVO to denote the overhead for each SVO test. For this test, only the extents of the testing scenarios will be used as it is easier to see the overhead of the updates with larger test samples. *Tables' 7,8,9, and 10* are used for convenient references for the analytical discussion of *Figure 28*.

15 Octree Random Pruning	
Min of Ellapased Milliseconds	1789
Max of Ellapased Milliseconds	441192
Average of Ellapased Milliseconds	194575.3773
Standard Deviation of Ellapsed Millisec	111141.0234
Min of Updates	0
Max of Updates	26
Average of Updates	11.052
Standard Deviation of Updates	4.811312553
Min of Total Frames	0
Max of Total Frames	457
Average of Total Frames	219.7746667
Standard Deviation of Total Frames	83.8821636

Table 7: 15 Octree Random Pruning

15 Octree Wall Pruning	
Min of Ellapased Milliseconds	1022
Max of Ellapased Milliseconds	420855
Average of Ellapased Milliseconds	194473.0127
Standard Deviation of Ellapsed Millisec	113150.7738
Min of Updates	0
Max of Updates	24
Average of Updates	11.12333333
Standard Deviation of Updates	4.791042853
Min of Total Frames	0
Max of Total Frames	457
Average of Total Frames	219.4946667
Standard Deviation of Total Frames	82.42792941

Table 8: 15 Octree Wall Random Pruning

15 SVO Random Pruning	
Min of Ellapased Milliseconds	3077
Max of Ellapased Milliseconds	472327
Average of Ellapased Milliseconds	221134.096
Standard Deviation of Ellapsed Millisec	126769.1704
Min of Updates	0
Max of Updates	33
Average of Updates	11.47266667
Standard Deviation of Updates	5.582893398
Min of Total Frames	0
Max of Total Frames	668
Average of Total Frames	248.8453333
Standard Deviation of Total Frames	102.7387151

Table 9: 15 SVO Random Pruning

15 SVO Wall Pruning	
Min of Ellapased Milliseconds	1887
Max of Ellapased Milliseconds	511287
Average of Ellapased Milliseconds	232697.4427
Standard Deviation of Ellapsed Millisec	132535.8647
Min of Updates	0
Max of Updates	38
Average of Updates	12.152
Standard Deviation of Updates	6.335481161
Min of Total Frames	0
Max of Total Frames	639
Average of Total Frames	257.3646667
Standard Deviation of Total Frames	114.8222791

Table 10: 15 SVO Wall Pruning

Figure 28: Update Comparison

Figure 28 shows the average milliseconds required per Update. This time includes movement information, changing *zeta's* position, as well as, pruning information once *zeta's* position changes. The octree has relatively the same time required. This is expected as there is no pruning information needed.

In comparison, the SVO time required is quite large. Approximately 2000 milliseconds

per update is not an efficient overhead for the system. This is likely caused due to the top-down approach which, while optimal for instantiation and traversal, may be suboptimal for updates as updates would likely be at the lower levels of the tree requiring more traversals than necessary.

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 Future Work

The future work for this paper dwarf's the scope of the system here. Foremost, this paper's topic was defined very narrowly as to delve deep into how to optimize in a particular instance. Starting with the specific and moving to the more general, areas of future work will be discussed below.

Parallelization

One of the ways noted in quite a few papers to improve octree efficiency was to implement parallelization on the process. This was not included in this paper due to limitations of hardware, time, and understanding of the field. This is one area that should be looked into to improve the efficiency of the structure even further.

Cloud-Based SVO

A Cloud-Based SVO, that is, a Sparse Voxel Octree in a Cloud Based Network would reduce the RAM overhead of the SVO, as well as provide, a parallelization structure. This would definitely be able to improve the optimization of the system; however, the purpose of this paper was single-machine focus to show how optimal the design could be. This is more of a hardware focus and could be a viable path forward for further optimization.

Unreal and Lower-Level Languages

The Unreal Engine, unlike the Unity Engine, has a backend in C++. A large majority of the development for this paper dealt with the optimization of memory and memory management. C++ does not manage the memory for developers which allows for better code efficiency especially in cases where the overhead is in a number of bytes like this paper's focus is.

6.2 Conclusion

This paper's stated goal was to optimize a spatial partitioning structure for 3D-Pathfinding purposes by structuring the data structure, in such a way, as to reduce traversal time and improving efficiency through optimal tree pruning and memory overhead. To test the algorithms and methodology implemented for this purpose, three different test series were used to analyze RAM Overhead, Traversal Efficiency, and Tree Pruning Efficiency.

The octree was used as a benchmark for success, especially in the RAM overhead. The SVO, as noted, is quite good at reducing the overhead of the system; furthermore, the integration of Loose and Linear Octrees provided a significant reduction in the overhead as well. As for the results, it was clear the SVO succeeded in its goal in being optimal. There are a few remaining issues that should be addressed such as moderately-high overhead for pruning, as well as, the usage or indication of the belief A* is a viable option.

As per the *Background,* A* is unviable as an algorithm in multi-unit systems due to the multiplicative increase in overhead caused by each additional unit; therefore, while A* is used for this paper, it is not a recommended solution; A* was primarily used to demonstrate the structure could be efficient even with inefficient pathfinding methodology.

It should be noted, the reductions undertaken with the SVO to improve efficiency like removing the ability to traverse to neighbor facing children and the usage relative location codes may not be useful or efficient for all methodologies. This implementation was created specifically to improve efficiency while reducing the desire for an optimal path in lue of the former.

Overall, the resulting implementation and analysis exceeded expectations and therefore, the methodology used for this paper can be shown to be effective for sparsely populated,

simulated environments where pathfinding required an array of units with the desire to travel orthogonally in space.

REFERENCES

[1] X. Cui and H. Shi, "A*-based Pathfinding in Modern Computer Games", *IJCSNS International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 125-130, 2011.

[2] *Fun-chess-board-set-innovative-decoration-chess-board-setup*. 2017.

[3] P. Kumar, L. Bottaci, Q. Mehdi, N. Gough and S. Natkin, "Efficient Pathfinding for 2D Games."

[4] D. Meagher, "Geometric modeling using octree encoding", *Computer Graphics and Image Processing*, vol. 19, no. 1, p. 85, 1982.

[5] T. Broersen, F. Fichtner, E. Heeres, I. de Liefde, O. Rodenberg, E. Verbree and R. Voute, "Using a linear octree to identify empty space in indoor point clouds for 3D pathfinding", in *Agile 2016*, Helsinki, 2016.

[6] J. Baert, A. Lagae and P. Dutré, "Out-of-Core Construction of Sparse Voxel Octrees", *Computer Graphics Forum*, vol. 33, no. 6, pp. 220-227, 2014.

[7] C. Ericson, *Real-time collision detection*. Amsterdam: Elsevier, 2005, pp. 285-349.

[8] *Lets Make An Octree in Unity*. 2016.

[9] S. Frisken and R. Perry, "Simple and Efficient Traversal Methods for Quadtrees and Octrees", *Journal of Graphics Tools*, vol. 7, no. 3, pp. 1-11, 2002.

[10] J. Elseberg, D. Borrmann and A. Nuchter, "Efficient Processing of Large 3D Point Clouds", 2011.

[11] J. Elseberg, D. Borrmann and A. Nüchter, "One billion points in the cloud – an octree for efficient processing of 3D laser scans", *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 76, pp. 76-88, 2013.

[12] D. Brewer, "3D Flight Navigation Using Sparse Voxel Octrees", in *Game AI Pro 3*, S. Rabin, Ed. New York: CRC Press, 2017.

[13] T. Ulrich, "Notes on spatial partitioning", *Tulrich.com*, 2017. [Online]. Available: http://www.tulrich.com/geekstuff/partitioning.html. [Accessed: 26- Nov- 2017].

[14] O. Rodenberg, E. Verbree and S. Zlatanova, "INDOOR A* PATHFINDING THROUGH AN OCTREE REPRESENTATION OF A POINT CLOUD", *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences*, vol. -21, pp. 249-255, 2016.

[15] S. Xu, L. Heng, D. Honegger and M. Pollefeys, "Real-time 3D navigation for autonomous vision-guided MAVs", 2015.

[16] V. Kilic and M. Yalcin, "An active wave computing based path finding approach for 3-D environment", 2011.

[17] "Space Partitioning Tutorial: Piko3D's Dynamic Octree « Piko3D", *Piko3d.net*, 2017. [Online]. Available: http://www.piko3d.net/tutorials/space-partitioning-tutorial-piko3ds-dynamic-octree/. [Accessed: 26- Nov- 2017].

[18] D. Libes, "Modeling dynamic surfaces with octrees", *Computers & Graphics*, vol. 15, no. 3, pp. 383-387, 1991.

[19] D. Geier, "Advanced Octrees 3: non-static Octrees", *The Infinite Loop*, 2017. [Online]. Available: https://geidav.wordpress.com/2014/11/18/advanced-octrees-3-non-static-octrees/. [Accessed: 26- Nov- 2017].

[20] "Unity - Fast Facts", *Unity*, 2017. [Online]. Available: https://unity3d.com/public-relations. [Accessed: 26- Nov- 2017].

[21] "Garbage Collection", *Docs.microsoft.com*, 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/index. [Accessed: 26- Nov- 2017].

[22] U. Technologies, "Unity - AEC", *Unity*, 2017. [Online]. Available: https://store.unity.com/industries/aec. [Accessed: 26- Nov- 2017].

[23] U. Technologies, "Unity - Manual: Profiler overview", *Docs.unity3d.com*, 2017. [Online]. Available: https://docs.unity3d.com/Manual/Profiler.html. [Accessed: 26- Nov- 2017].

[24] U. Technologies, "Unity - Manual: CPU Usage Profiler", *Docs.unity3d.com*, 2017. [Online]. Available: https://docs.unity3d.com/Manual/ProfilerCPU.html. [Accessed: 26- Nov- 2017].

[25] U. Technologies, "Unity - Scripting API: Gizmos", *Docs.unity3d.com*, 2017. [Online]. Available: https://docs.unity3d.com/ScriptReference/Gizmos.html. [Accessed: 26- Nov- 2017].

[26] "| Operator (C# Reference)", *Docs.microsoft.com*, 2017. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/or-operator. [Accessed: 26- Nov- 2017].

[27] *Octree Numbering.* [Online]. Available: https://www.volume-gfx.com/wp-content/uploads/2013/02/octreeNumbering.png. [Accessed: 26- Nov- 2017].

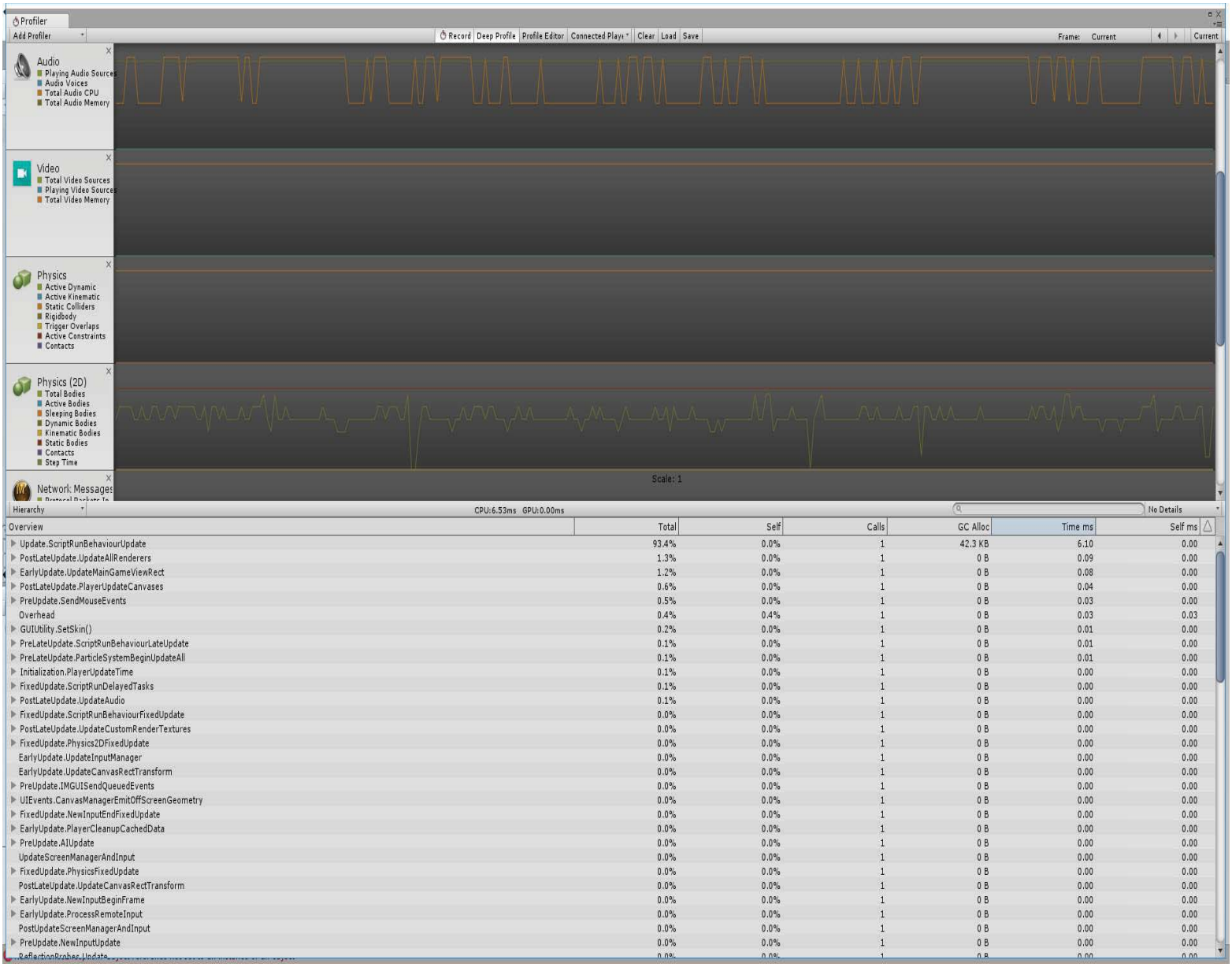
[28] S. Lauge, *A* Pathfinding (E01: algorithm explanation)*. 2014.

[29] "A* Search Algorithm - GeeksforGeeks", *GeeksforGeeks*, 2017. [Online]. Available: http://www.geeksforgeeks.org/a-search-algorithm/. [Accessed: 26- Nov- 2017].

[30] D. Brewer, "Getting Off the Navmesh Navigating", in *GDC*, 2015.

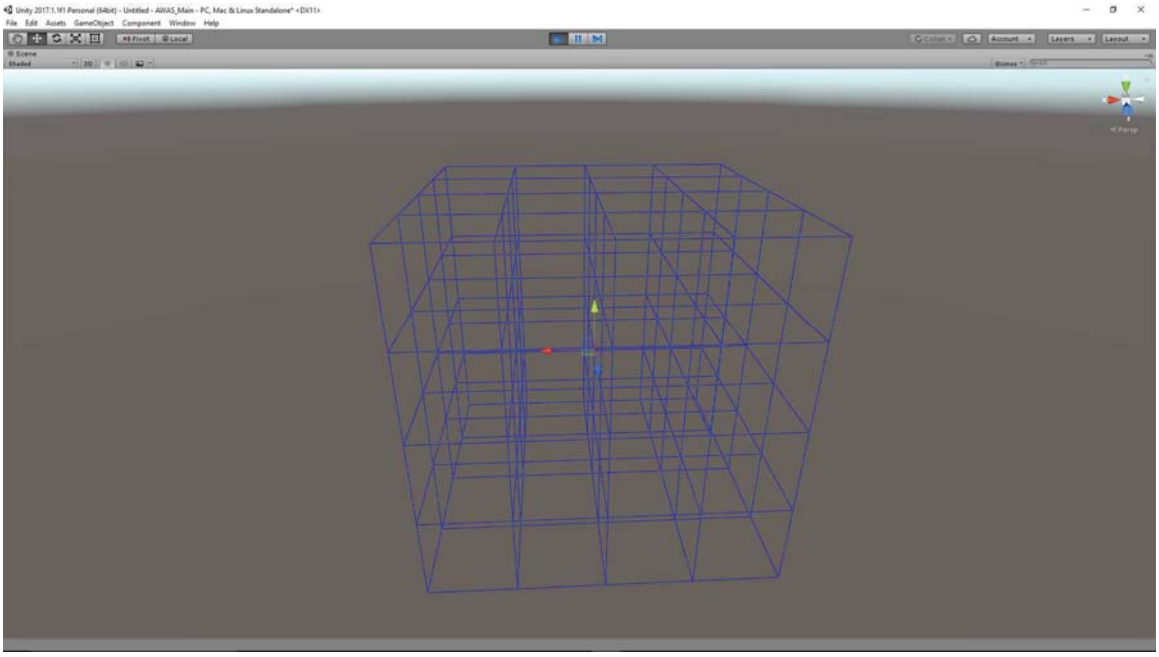
APPENDIX A

This is the Unity Profiler. It is used for visualizing performance as well as testing on the back-

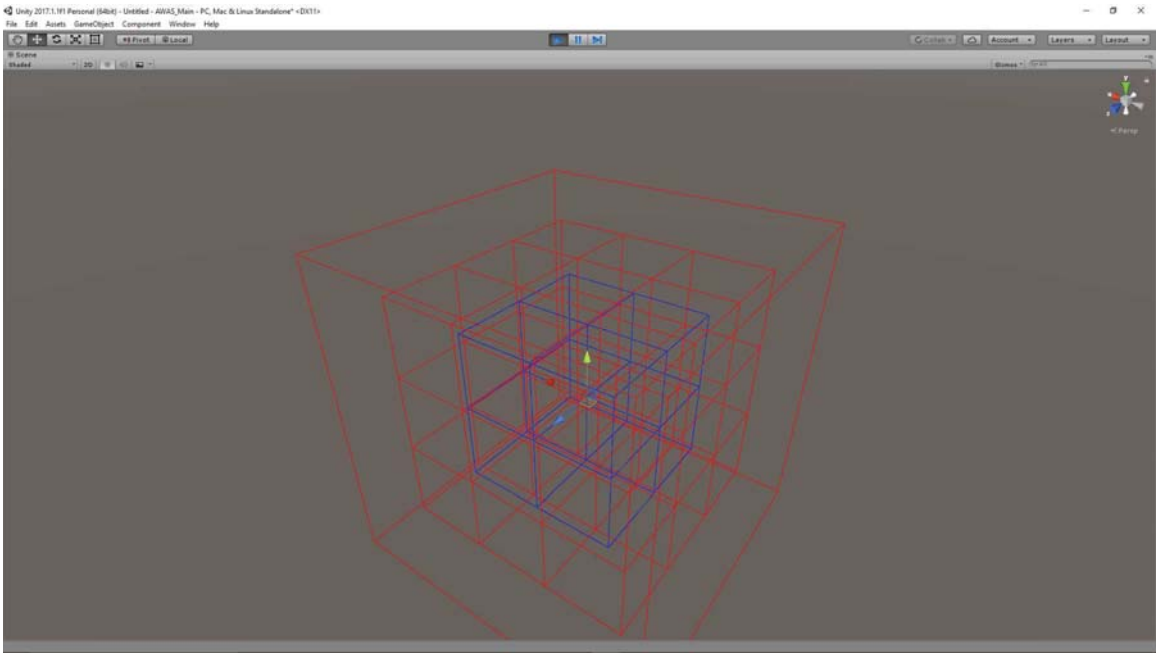


APPENDIX B

This is an example of the Unity Gizmos toolkit that will be used for visualization purposes throughout the paper.

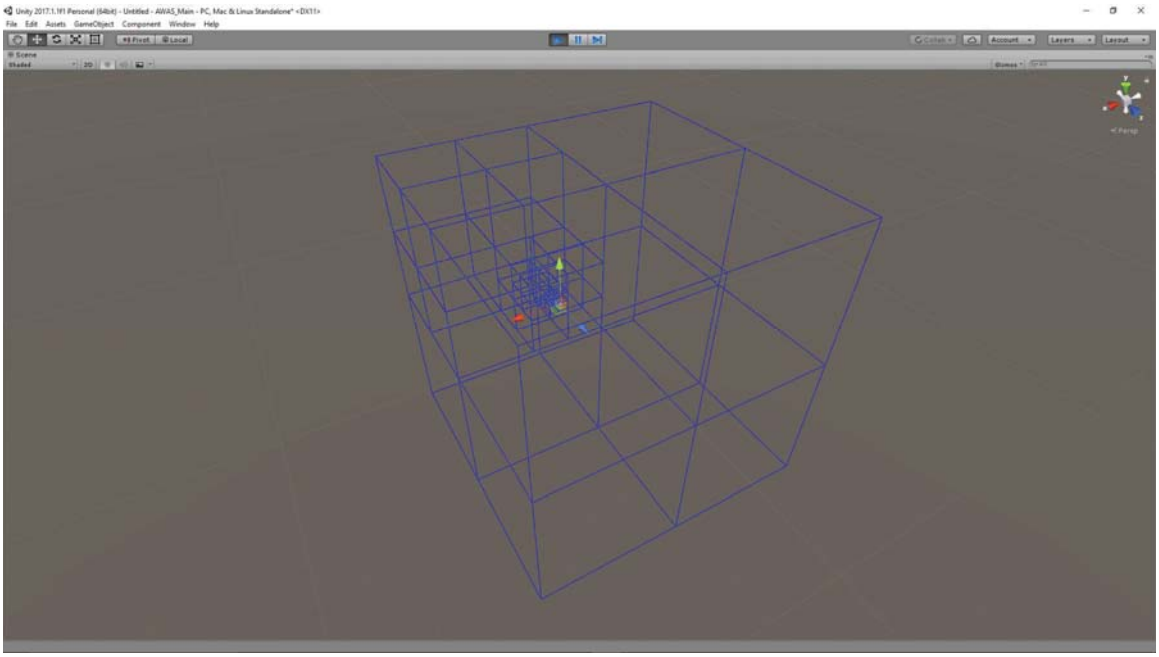


This is an example of a Loose Octree representation. The red signifies the outer bounds of each node while the blue signifies the inner bounds.

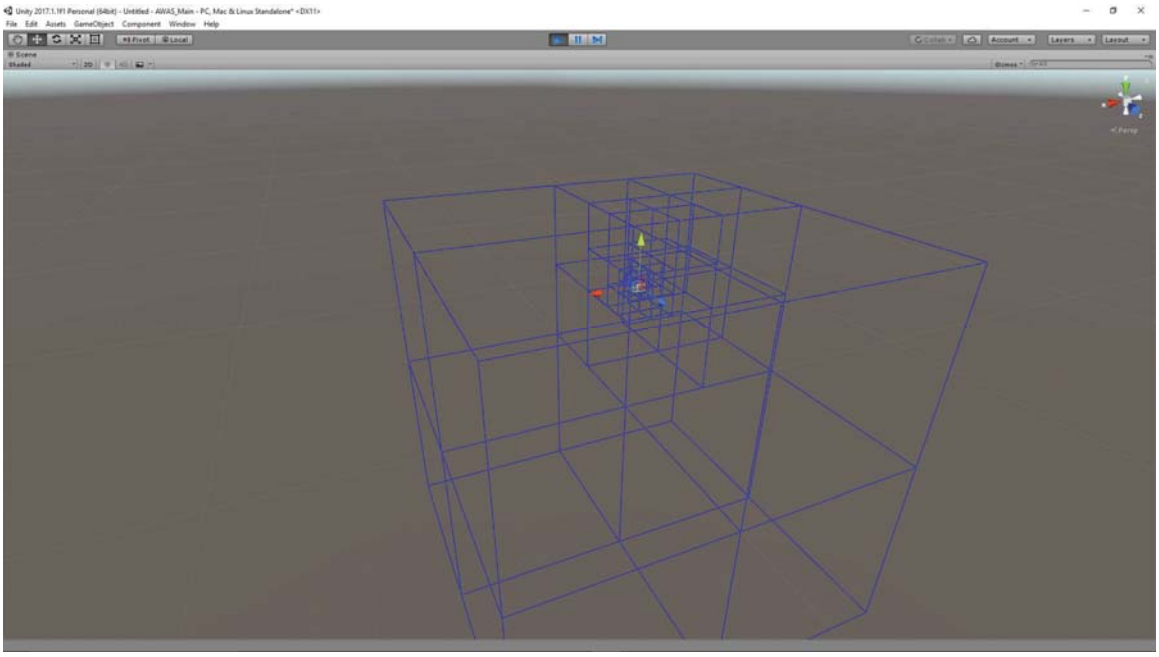


APPENDIX C

This is representative of the old *zeta* position in preparation for movement.

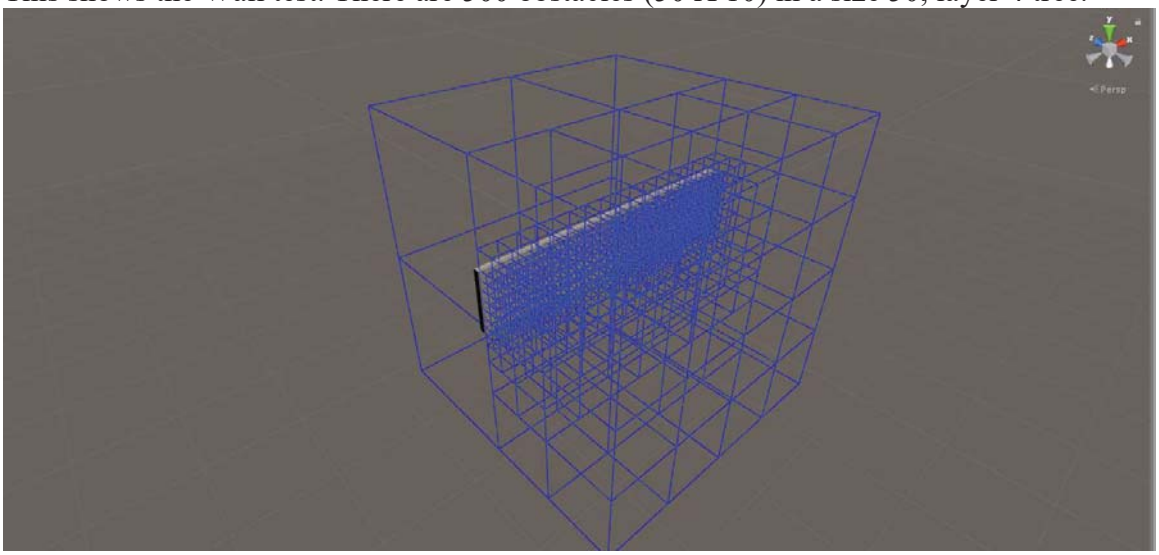


This is the result after movement from the prior image. The tree was pruned an octant and inserted nodes into *zeta's* current octant.

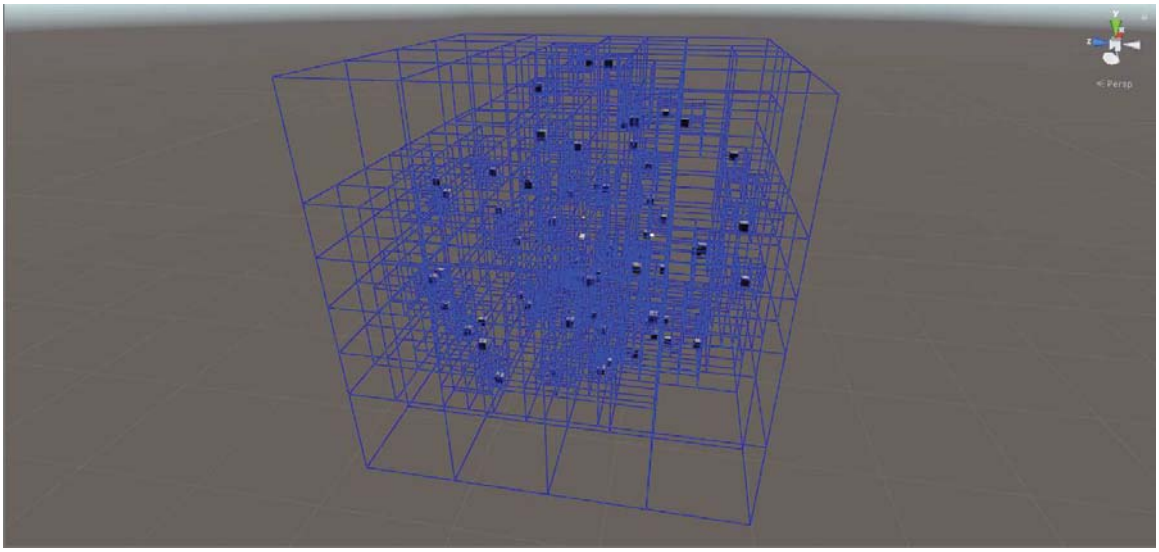


APPENDIX D

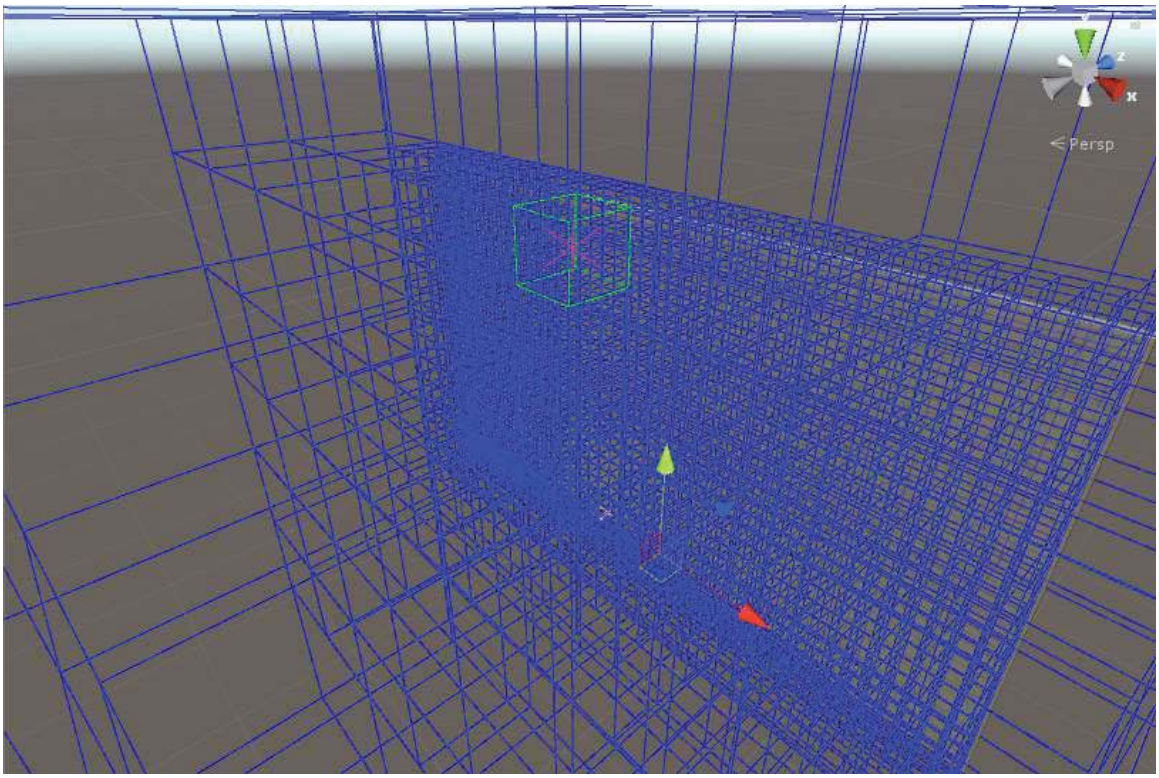
This shows the Wall test. There are 500 obstacles (50 X 10) in a size 50, layer 4 tree.



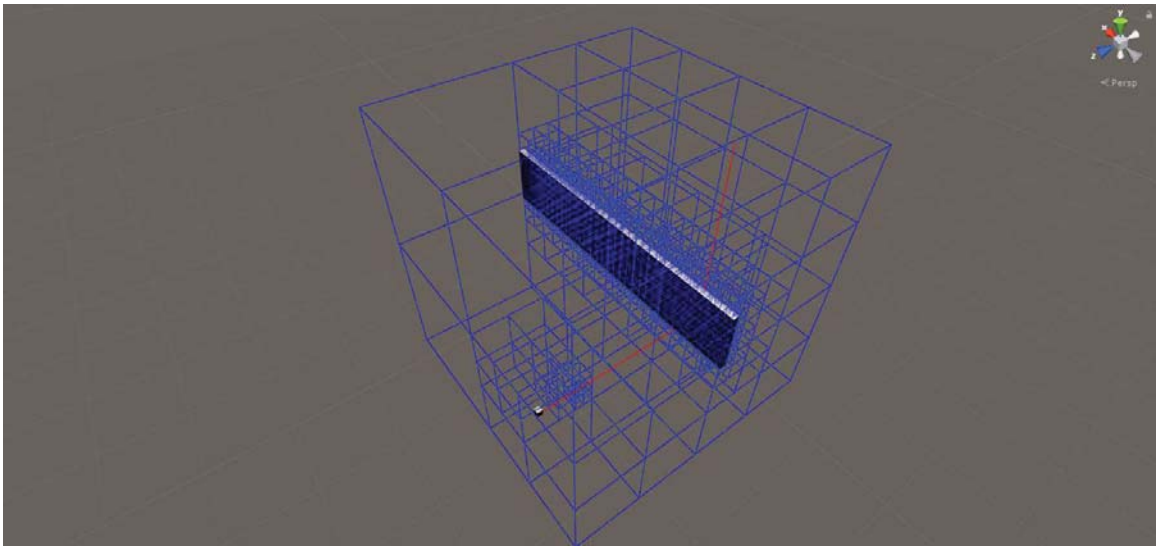
This shows the Random Map test. There are 64 random obstacles in a size 50, layer 4 tree.



The green box is an example of a step within the traversal methodology. It represents the current node in A* and helps to visualize the traversal process.



The red line is an example of the visualization of the navigation methodology. It shows from *zeta's* current location what path will be taken to get to the end node.



APPENDIX E

Octree/Sparse Voxel Octree

This piece of code denotes the Octree/SVO implementation for one tree with all the methods and variables.

```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using UnityEngine;
[System.Serializable]
public class StandardSVO :
MonoBehaviour {
    public float size = 5;
    public int depth = 2;
    public bool Octree = false;
    public static StandardSVO Instance { get; set; }
    int[] numNodesLayer = new int[6];
    public List<Vector3> obsStartPoints = new List<Vector3>();
    public List<Vector3> obsEndPoints = new List<Vector3>();
    public Hashtable CurrentTree = new Hashtable();
    Node _rootNode;
    private void Awake()
    {
        Instance =
this;
        init();
    }

    public void init()
    {
        _rootNode = new Node();
        _rootNode.assignedObstacles = new Hashtable();
        _rootNode._childBitMask = new int[8] { 0, 0, 0, 0, 0, 0, 0, 0 };
        //_rootNode.key = CreateKey("", 0);
        _rootNode.key = 7;
        InsertNode(7, _rootNode.key);
        if (Octree)
    VisitLinearOctree(_rootNode);
    }

    #region Tree Structure
    [System.Serializable
] public struct Node
    {
        [System.NonSerialized]
        public Vector3 center;
```

```
44          public float x;
45          public float y;
46          public float z;
47          //public string key;
48          public uint index;
49          public uint key;
50          public int[] _childBitMask;
51          [System.NonSerialized]
52          public Hashtable assignedObstacles;
53
54      }
55
56      uint CreateKey(uint parent, uint
index) 57   {
58          return ((parent << 3) |
index); 59  }
60      uint RetreiveParent(uint
key) 61 {
62          return (key >>
3); 63   }
64      //https://stackoverflow.com/questions/1489830/efficient-way-to-
          determine- number-of-digits-in-an-integer
65      uint GetNumberOfDigits(uint
i) 66   {
67          return i > 0 ? (uint)System.Math.Log10((double)i) + 1 :
1; 68   }
69
70      Node InsertNode(uint index, uint
key) 71 {
72
73          if (CurrentTree.ContainsKey(key))
74          {
75              return
(Node)CurrentTree[key]; 76      }
77          else
78          {
79              Node _child = new global::StandardSVO.Node();
80              _child.assignedObstacles = new Hashtable();
81              _child._childBitMask = new int[8] { 0, 0, 0, 0, 0, 0, 0, 0 };
82              _child.key = key;
83              _child.index = index;
84              _child.center = NodePosition(index, key);
85              _child.x = _child.center.x;
86              _child.y = _child.center.y;
87              _child.z = _child.center.z;
88
89              //GameObject g = GameObject.CreatePrimitive(PrimitiveType.Cube);
90              //g.SetActive(false);
91              //g.transform.position = _child.center;
92              //g.name = key.ToString();
93              //g.transform.localScale = NodeSize(key) * Vector3.one;
94              //Debug.Log(key);
```

```
 95                 CurrentTree.Add(key,
_child); 96
 97                 return _child;
 98
 99             }
10
 0
10
 1
10
 2
103         }
```

```
104
105     bool ChildrenFilled(Node node)
106     {
107         bool holder = false;
108         for (int i = 0; i < node._childBitMask.Length; i++)
109         {
110             if (node._childBitMask[i] == 1)
111                 holder = true;
112         }
113         return holder;
114     }
115     void removeNode(Node node)
116     {
117
118         if (node.assignedObstacles.Count == 0)
119         {
120             if (GetNumberOfDigits(node.key) <= depth + 1 &&
                  GetNumberOfDigits (node.key) > 1)
121             {
122                 Node parent = (Node)CurrentTree[RetreiveParent(node.key)];
123                 //SetBitMask(parent, (int)node.index, 0);
124                 parent._childBitMask[node.index] =
0; 125
126                 if (/*!ChildrenContainObstacle(node) &&*/
                    !ChildrenFilled (node)) //This is Flag bit 9 uint 0 >
                    7 = 0
127                 {
128                     bool obs = false;
129
130                     for (int i = 0; i < 8; i++)
131                     {
132                      if (CurrentTree.ContainsKey(CreateKey(node.key, (uint)
                       i)))
133                      obs |= CheckPathForObstacles((Node)CurrentTree
                       [CreateKey(node.key, (uint)i)]);
134                     }
135                     if (!obs)
136                     {
137                         for (int i = 0; i < 8; i++)
138                         {
139                             CurrentTree.Remove(CreateKey(node.key, (uint)i));
140                         }
141                         removeNode(parent);
142
143                     }
144                 }
145
146             }
147
148
149         }
150
```

151

```
152     }
153     bool CheckPathForObstacles(Node node)
154     {
155         bool found = false;
156         if (node.assignedObstacles.Count > 0)
157             found =
true; 158
159         if (GetNumberOfDigits(node.key) <= depth & !found) //otherwise
            continue down
160         {
161             for (int i = 0; i < 8; i++)
162             {
163                 if (CurrentTree.ContainsKey(CreateKey(node.key, (uint)i)))
164                     found |=
                        CheckPathForObstacles((Node)CurrentTree[CreateKey
                        (node.key, (uint)i)]);
165             }
166
167
168         }
169         return found;
170
171     }
172
173     void TraverseToLowestChildren(Node _current)
174     {
175         if (CurrentTree.ContainsKey(CreateKey(_current.key, 0)))
176         {
177             for (int i = 0; i < 8; i++)
178             {
179
180
                        TraverseToLowestChildren((Node)CurrentTree[CreateK
                        ey (_current.key, (uint)i)]);
181
182
183             }
184         }
185         else
186         {
187             removeNode(_current);
188         }
189     }
190
191
192     void VisitNode(Obstacle Obstacle, Node node)
193     {
194         if (GetNumberOfDigits(node.key) > 1)
195         {
196             Node parent = (Node)CurrentTree[RetreiveParent(node.key)];
197             //SetBitMask(parent, (int)node.index, 1);
198             parent._childBitMask[node.index] = 1;
```

```
199             //Debug.Log(parent.key + " " + ChildrenFilled(parent));
200         }
```

```
201         if (GetNumberOfDigits(node.key) <= depth && !Octree) //Add Children
              of Node
202         {
203             for (int i = 0; i < 8; i++)
204             {
205
206                 InsertNode((uint)i, CreateKey(node.key, (uint)i));
207
208             }
209
210         }
211         if (Obstacle.Size > NodeSize(node.key) * 2) //Loose Octree
212         {
213             Node parent = (Node)CurrentTree[RetreiveParent(node.key)];
214             parent.assignedObstacles.Add(Obstacle.GUID, Obstacle);
215             Obstacle.KeyPosition = parent.key;
216         }
217         else
218         {
219             if (GetNumberOfDigits(node.key) <= depth) //otherwise continue down
220             {
221                 uint index = (uint)GetIndexOfPosition(Obstacle.CurrentPosition,
                      node.center);
222                 VisitNode(Obstacle, InsertNode(index, CreateKey(node.key, (uint)
                      GetIndexOfPosition(Obstacle.CurrentPosition, node.center))));
223
224             }
225             else // Obstacle is stored in current node (at lowest depth possible)
226             {
227                 node.assignedObstacles.Add(Obstacle.GUID, Obstacle);
228                 Obstacle.KeyPosition = node.key;
229
230             }
231         }
232
233     }
234
235
236
237     Vector3 NodePosition(uint index, uint key)
238     {
239         float step = NodeSize(key) / 2f;
240         Vector3 newPos =
gameObject.transform.position; 241
242
243         if (GetNumberOfDigits(key) > 1)
244         {
245             newPos = ((Node)CurrentTree[RetreiveParent(key)]).center;
246             uint i =
index; 247
248             newPos.x += ((i & 1) == 1 ? step : -step);
249             newPos.y += ((i & 2) == 2 ? step : -step);
```

```
250             newPos.z += ((i & 4) == 4 ? step : -step);
251
252
253         }
254         return newPos;
255     }
256
257     public float NodeSize(uint key)
258     {
259         double _currentSize = GetNumberOfDigits(key);
260         _currentSize -= 1;
261         _currentSize = size * (Mathf.Pow(.5f, (float)_currentSize));
262         return (float)_currentSize;
263     }
264     public int GetIndexOfPosition(Vector3 lookupPosition, Vector3 nodePosition)
265     {
266         int index = 0;
267
268         index |= lookupPosition.y > nodePosition.y ? 2 : 0;
269         index |= lookupPosition.x > nodePosition.x ? 1 : 0;
270         index |= lookupPosition.z > nodePosition.z ? 4 :
0; 271
272         return index;
273     }
274     #endregion
275     #region GUI
276     private void OnDrawGizmos()
277     {
278         numNodesLayer = new int[6];
279         DrawNodes(_rootNode)
; 280
281     }
282     bool ChildrenContainObstacle(Node node)
283     {
284         bool obst = false;
285         if (CurrentTree.Contains(CreateKey(node.key, (uint)0)))
286         {
287             for (int i = 0; i < 8; i++)
288             {
289                 if (((Node)CurrentTree[CreateKey(node.key,                    ↵
                      (uint) i)]).assignedObstacles.Count > 0)
290                     obst = true;
291             }
292         }
293         return obst;
294     }
295     void DrawNodes(Node node, int nodeDepth = 0)
296     {
297
298         Gizmos.color = Color.blue;
299         if (node.assignedObstacles.Count > 0)
300         {
```

```
301                 Gizmos.DrawWireCube(NodePosition(node.index, node.key), Vector3.one
                        * NodeSize(node.key));
302
303             }
304
305
306
307             //Gizmos.color = Color.red;
308             //Gizmos.DrawWireCube(NodePosition(node.key), Vector3.one *
                  NodeSize (node.key) * 2);
309              numNodesLayer[GetNumberOfDigits(node.key)-1]++;
310             if (GetNumberOfDigits(node.key) <= depth)
311             {
312                 for (int i = 0; i < 8; i++)
313                 {
314                     if (CurrentTree.Contains(CreateKey(node.key, (uint)i)))
315                     {
316                         DrawNodes((Node)CurrentTree[CreateKey(node.key,
(uint)i)]); 317
318                     }
319                 }
320             }
321         }
322
323         #endregion
324         #region Obstacle
Necessities 325
326         public void InsertObstacle(Obstacle obstacle)
327         {
328             VisitNode(obstacle, _rootNode);
329         }
330         public void RemoveObstacle(Obstacle Obstacle)
331         {
332             Node _node = (Node)CurrentTree[Obstacle.KeyPosition];
333             _node.assignedObstacles.Remove(Obstacle.GUID);
334             if (!Octree)
335                 //TraverseToLowestChildren(_node);
336                 removeNode(_node);
337         }
338
339         public bool ChangeObstaclePosition(Vector3 pos, Obstacle Obstacle)
340         {
341
342             Node _node = ((Node)CurrentTree[Obstacle.KeyPosition]);
343             if (Vector3.Distance(pos, _node.center) > NodeSize(_node.key)/* /
                  2f*/)// LOOSE OCTREE
344             {
345                 RemoveObstacle(Obstacle);
346                 InsertObstacle(Obstacle);
347                 return true;
348             }
349             else
```

```
350             return false;
351
352     }
353
354     #endregion
355     #region Octree
356     void VisitLinearOctree(Node node)
357     {
358         if (GetNumberOfDigits(node.key) <= depth)
359         {
360
361
362             for (int i = 0; i < 8; i++)
363             {
364
365                 VisitLinearOctree(InsertNode((uint)i, CreateKey(node.key, (uint)
                      i)));
366             }
367         }
368     }
369     #endregion
370     #region
A* 371
372     public uint FindEndNode(Obstacle Obstacle, Vector3 Destination)
373     {
374         return EndNode(Obstacle, _rootNode, Destination);
375     }
376     uint EndNode(Obstacle Obstacle, Node node, Vector3 Destination)
377     {
378
379         uint key = 0;
380         if (Obstacle.Size > NodeSize(node.key) * 2) //Loose Octree
381         {
382
383             key = RetreiveParent(node.key);
384         }
385         else
386         {
387             if (GetNumberOfDigits(node.key) <= depth) //otherwise continue down
388             {
389                 uint index = (uint)GetIndexOfPosition(Destination, node.center);
390                 uint keycode = CreateKey(node.key,
                      (uint)GetIndexOfPosition (Destination, node.center));
391                 if (CurrentTree.ContainsKey(keycode))
392                     key = EndNode(Obstacle, (Node)CurrentTree[keycode],
                          Destination);
393                 Else
394                     key = node.key;
395                 // key = EndNode(Obstacle, InsertNode(index, CreateKey(node.key,
                      (uint)GetIndexOfPosition(Destination, node.center))),
                      Destination);
396
```

```
397                 }
398                 else
399                 {
400                     key = node.key;
401 
402                 }
403             }
404             //Debug.Log(key);
405             return
key; 406
407         }
408 
409         uint DirectionalKey(AStar star, Node node, Vector3 Direction, float
              size, float globalsize)
410         {
411             Vector3 pos = NodeSize(node.key) * Direction + node.center;
412             //Debug.Log(node.center + " " + pos);
413             //if directional position is within the SVO continue on, else return 0
414             if (Vector3.Distance(pos, _rootNode.center) > globalsize )
415             {
416                 //Debug.Log(node.center + " 0" + Vector3.Distance(pos,
417                 return 0;
418 
419             }
420             else
421                 return KeyByPosition(star, _rootNode, pos,
                      GetNumberOfDigits (node.key), size);
422         }
423 
424         uint KeyByPosition(AStar star, Node node, Vector3 position, uint
              nodedepth, float size)
425         {
426             uint key = 0;
427             if (size > NodeSize(node.key) * 2) //Loose Octree
428             {
429                 Node parent = (Node)CurrentTree[RetreiveParent(node.key)];
430                 //parent.assignedObstacles.Add(Obstacle.GUID, Obstacle);
431                 key = parent.key;
432             }
433             else
434             {
435                 if (GetNumberOfDigits(node.key) <= nodedepth)
436                 {
437                     if (GetNumberOfDigits(node.key) <= depth) //otherwise continue
                          down
438                     {
439                         uint index = (uint)GetIndexOfPosition(position, node.center);
```

```
440                 uint keycode = CreateKey(node.key,
                    (uint)GetIndexOfPosition(position, node.center));
441                 if (CurrentTree.ContainsKey(keycode))
442                     key = KeyByPosition(star, (Node)CurrentTree[keycode],
```

```
                    position, nodedepth, size);
443                 else
444                     key = node.key;
445
446
447             }
448             else // Obstacle is stored in current node (at lowest depth possible)
449
450
451
452
453
454
```

{

}
}else
{

```
key = node.key;
455                 //If the depth > than node depth, set the key to the parent
                      of this depth and then find the face nodes one level lower
                      if possible
456                 //If there are nodes 1 level lower, it will add them in
                      retroactivly. Further, if any of those children are
                      filled, this key won't be added but they might still be
457                 Node parent = (Node)CurrentTree[RetreiveParent(node.key)];
458                 //parent.assignedObstacles.Add(Obstacle.GUID, Obstacle);
459                 key =
parent.key; 460
461                 FindFaceNodes(star,
node); 462
463             }
464         }
465         if (key != 0)
466         {
467             // Debug.Log(key + " " + ((Node)CurrentTree
                  [key]).assignedObstacles.Count + " " +
                  ChildrenFilled((Node) CurrentTree[key]));
468             if (ChildrenFilled((Node)CurrentTree[key]) || ((Node)CurrentTree
                  [key]).assignedObstacles.Count > 0) //Don't allow to go to
                  traverse if obstacle there
469                 key = 0;
470         }
471
472         return key;
473     }
474
475     void FindFaceNodes(AStar star, Node node)
476     {
477         //Find the nodes on the face by calculating nodes closest to
              the directional position of the currentnode on astar
478
479     }
480
481     public uint[] TraverseNeighbors(AStar star, Node node)
482     {
483         //Find Back, Left, Right, Forward, Up, Down
```

```
484
485             uint[] neighbors =  new uint[6];
486             neighbors[0] = DirectionalKey(star, node, Vector3.left,
                  NodeSize (node.key), size);
487             neighbors[1] = DirectionalKey(star, node, Vector3.right,
                  NodeSize (node.key), size);
488             neighbors[2] = DirectionalKey(star, node, Vector3.up,
                  NodeSize(node.key), size);
489             neighbors[3] = DirectionalKey(star, node, Vector3.down,
                  NodeSize (node.key),  size);
490             neighbors[4] = DirectionalKey(star, node, Vector3.back,
                  NodeSize (node.key),  size);
491             neighbors[5] = DirectionalKey(star, node, Vector3.forward,
                  NodeSize (node.key), size);
492
493
494             return neighbors;
495         }
496
497         void RetrieveMemoryUsage()
498         {
499             long size = 0;
500             using (Stream s = new MemoryStream())
501             {
502                 BinaryFormatter formatter = new BinaryFormatter();
503                 formatter.Serialize(s, CurrentTree);
504                 size = s.Length;
505             }
506
507             Debug.Log(CurrentTree.Count + " " + size + " " + numNodesLayer[0] +   " "
                  + numNodesLayer[1] +  " " +numNodesLayer [2]  + " " + numNodesLayer[3]
                   + " " + numNodesLayer[4] /*/*+ " " + numNodesLayer[5]*/);
508         }
509
510
511         private void Update()
512         {
513             RetrieveMemoryUsage();
514         }
515         #endregion
516     }
```

APPENDINX F

A Incremental*

This shows all methodology and variables for the A* implementation. Refer to the Octree/SVO code for some of the connections.

```
using MovementEffects;
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using UnityEditorInternal;
using UnityEngine;
using UnityEngine.Profiling;

public class AStar {


    public SimplePath path { get; set; }
    public StandardSVO SVO { get; set; }
    uint SartNode { get; set; }
    uint EndNode { get; set; }
    public Vector3 EndPosition { get; set; }
    NodeValue CurrentNode { get; set; }
    Dictionary<uint, NodeValue> OpenList = new Dictionary<uint, NodeValue>();
    public Dictionary<uint, NodeValue> ClosedList = new
      Dictionary<uint, NodeValue>();
    public bool init = false;
    public List<uint> _path = new List<uint>();
    public Stopwatch AStarWatch = new Stopwatch();
    public int increments = 0;
    public int startFrame = 0;
    public int endFrame = 0;
    public Vector3 StartPos;
    public AStar (SimplePath
p) {
        path = p;
        SVO = StandardSVO.Instance;
        AStarWatch.Start();
        ChartPath();
    startFrame = Time.frameCount;
    }

    void ChartPath()
    {
        //Debug.Log("CHART");
        //Debug.Log(path.Destination);
        if (EndNode == 0)
```

```
44              EndNode =
                  SVO.FindEndNode(path.GetComponent<Obstacle>
                  (),path.Destination );
45          EndPosition= ((StandardSVO.Node)SVO.CurrentTree[EndNode]).center; ;
46          //Add start node to open list
47          //you will start with this node to go through a*
48          NodeValue startNode = new NodeValue();
49          startNode.key = path.GetComponent<Obstacle>().KeyPosition;
50          SartNode = startNode.key;
```

```
51
52
53
54

55
56
57
58
59
60
61
62
63
64
65

66
67
68
69
70
71
72
73
74
75
76
77
78
79
80          }
81
```

```
//Debug.Log(startNode.key + " " + EndNode);
startNode.Center = ((StandardSVO.Node)SVO.CurrentTree[SartNode]).center;
startNode.gValue = 0;
startNode.hValue = Vector3.Distance(startNode.Center, ((StandardSVO.Node)
  SVO.CurrentTree[EndNode]).center);
StartPos = startNode.Center;

OpenList.Add(startNode.key, startNode);

//If EndNode equals StarNode do not run AStar if
(SartNode == EndNode)
{
    OpenList.Clear();
    //Debug.Log("CURRENT LOCATION");
    EndNode = 0;
    path.Destination = UnityEngine.Random.insideUnitSphere *
      StandardSVO.Instance.size / 2f;
    ChartPath();
}
else
{
  //  Timing.RunCoroutine(pause());
    //Debug.Log("RUN");
    Timing.RunCoroutine(IncrementAStar());


}
82      IEnumerator<float>
IncrementAStar() 83
84      {
85
86          increments++;
87          //Debug.Log(OpenList.Count);
88          //While AStar Open List > 0 and current node != end node
89          if (OpenList.Count > 0)
90          {
91             // Debug.Log("OPEN");
92              CurrentNode = OpenList.ElementAt(0).Value;
93              //Find Current Node F, G, H
Cost 94
95              float f = CurrentNode.hValue + CurrentNode.gValue;
96              float g = 0;
97              float h = Vector3.Distance(CurrentNode.Center,
EndPosition); 98
99              //Compare with other nodes to find lowest f cost
100             using (var enumerator = OpenList.GetEnumerator())
```

```
101             {
102                 while (enumerator.MoveNext())
103                 {
104                     float fCost = enumerator.Current.Value.gValue
                        +enumerator.Current.Value.hValue;
105                     fCost /= StandardSVO.Instance.NodeSize
                        (enumerator.Current.Value.key);//Want to find larger nodes -
                        >traverse less nodes this  way
106                      //set lowest f cost node to current
node 107
108                     if (CurrentNode.key != enumerator.Current.Value.key && fCost
                        <= f && enumerator.Current.Value.hValue < CurrentNode.hValue)
109                     {
110                         CurrentNode = enumerator.Current.Value;
111                         f = fCost;
112                         g = CurrentNode.gValue;
113                         h = CurrentNode.hValue;
114
115                     }
116                 }
117
118             }
119             //add current node to close list
120
121             OpenList.Remove(CurrentNode.key);
122             ClosedList.Add(CurrentNode.key, CurrentNode);
123
124                     //Find Neighbors and add to Open list
125             if (CurrentNode.key != EndNode)
126             {
127                 //Debug.Log(CurrentNode.key);
128                 //Debug.Log(((StandardSVO.Node)SVO.CurrentTree
                      [CurrentNode.key]));
129                 if
                      (((StandardSVO.Node)SVO.CurrentTree[CurrentNode.key]).Equals
                      (null))
130                 {
131                    // Debug.Log("NULLED OUT");
132                     EndNode = 0;
133                     path.Destination = UnityEngine.Random.insideUnitSphere *
                        StandardSVO.Instance.size / 2f;
134                     ChartPath();
135                 }
136                 else
137                 {
138                     uint[] neigbhors = StandardSVO.Instance.TraverseNeighbors
                        (this, ((StandardSVO.Node)SVO.CurrentTree[CurrentNode.key]));
139                     for (int i = 0; i < neigbhors.Length; i++)
140                     {
141                         AddToOpenList(neigbhors[i]);
142                     }
143                     Timing.RunCoroutine(IncrementAStar());
```

```
144                 }
```

```
145             }else
146             {
147                 //Initialize Path once End Node is found
148 
149                 InitializePath();
150             }
151 
152 
153 
154         }else
155         {
156             path.GetComponent<MovingObstacle>().FindNewDestination(0);
157         }
158         yield return 0;
159     }
160     void AddToOpenList(uint key)
161     {
162         // if key = 0/ can't add
163 
164         if (key != 0)
165         {
166             NodeValue value ;
167             ClosedList.TryGetValue(key, out value);
168 
169             if (value.key == 0) //Check to make sure that the item isn't in the ➲
                  closed list
170             {
171                 OpenList.TryGetValue(key, out value);
172                 float _NeighborGCost = CurrentNode.gValue +                    ➲
                    Vector3.Distance (CurrentNode.Center, value.Center);
173                 float _NeighborHCost = CurrentNode.hValue +                    ➲
                    Vector3.Distance (value.Center, EndPosition);
174 
175                 if (value.key == 0)
176                 {
177                     value = new NodeValue();
178                     value.gValue = _NeighborGCost;
179                     value.hValue = _NeighborHCost;
180                     value.pathParent = CurrentNode.key;
181                     value.key = key;
182                     value.Center =  ((StandardSVO.Node)                        ➲
                       SVO.CurrentTree [key]).center;
183                     OpenList.Add(key,
value); 184
185                 }
186                 else        //Update node if calcualted g cost for this node is ➲
                      <than prior calculated cost
187                     //The point of this is to find most optimal path -> the h    ➲
                      score will naturally update when the g score does b/c it    ➲
                      has partial composition to the g score
188 
189                 {
```

```
190                         if (_NeighborGCost < value.gValue)
191                         {
192                             value.gValue = _NeighborGCost;
193                             value.hValue = _NeighborHCost;
194                             value.pathParent = CurrentNode.key;
195                         }
196                     }
197 
198                 }
199 
200 
201             }
202 
203 
204         }
205         void InitializePath()
206         {
207             //Debug.Log("INITALIZE PATH");
208            // Debug.Log(CurrentNode.key);
209             _path.Add(CurrentNode.key);//End Node
210             while (CurrentNode.key != SartNode)
211             {
212                 CurrentNode = ClosedList[CurrentNode.pathParent];
213                 _path.Add(CurrentNode.key);
214 
215             }
216             //_path.Add(CurrentNode.key);
217 
218             _path.Reverse();
219             endFrame = Time.frameCount;
220             ;
221             AStarWatch.Stop();
222             //UnityEngine.Debug.Log("ASTAR " + path.GetComponent<Obstacle>().GUID  +"
                    " + AStarWatch.ElapsedMilliseconds + " " + increments + " "
                  +ClosedList[_path[0]].Center
223             //    + " " + EndPosition + " " + startFrame + " " +
endFrame); 224
225             //for (int i = 0; i < _path.Count;
i++) 226         //{
227             //    Debug.Log(_path[i]);
228             //}
229             init = true;
230             path.move = true;
231         }
232 
233         public struct NodeValue
234         {
235             public uint key;
236             public uint pathParent;
237             public float gValue;
238            public  float hValue;
239            public Vector3 Center;
```

```
240
241         }
242
243
244
245 }
246
```

APPENDIX G – Raw Data

Raw Data of Results in Excel Spreadsheet Format

https://drive.google.com/drive/u/0/folders/172nmW3EXi6b__GYbNLDqCXfTZdJ87m8O