The Thesis Committee for Kyle Patrick Kercher
Certifies that this is the approved version of the following Thesis:

Generic Implementation of CAD Models for Nuclear Simulation

APPROVED BY
SUPERVISING COMMITTEE:

Richard Crawford, Supervisor

Carolyn Seepersad

Generic Implementation of CAD Models for Nuclear Simulation

by

Kyle Patrick Kercher

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2019

Acknowledgements

I would like to thank my advisor Dr. Richard Crawford for his insight on various problems throughout this research. I would also like to thank Dr. Greg Thoreson and Dr. Steve Horne from Sandia National Labs for both the opportunity to perform this research and their guidance in navigating GADRAS. Also, I'd like to express my gratitude to Dr. Carolyn Seepersad for all the assistance she has provided through graduate school. Additionally, I would like to thank my family and friends for their love and support and making me who I am today. Finally, a special thanks to Sandia National Labs for sponsoring the project (SAND2019-14390 T).

Abstract

Generic Implementation of CAD Models for Nuclear Simulation

Kyle Patrick Kercher M.S.E.

The University of Texas at Austin, 2019

Supervisor: Richard Crawford

The goal of this project is to utilize the preexisting framework of GADRAS to simulate the radiation leakage from arbitrary CAD models without sacrificing speed or accuracy. The proposed solution is to use STL files to define models. Then, a three-dimension binning structure is created to contain all the elements of the file. This results in preservation of speed, without adding higher performance hardware requirements. Finally, the discretization is performed using a three-dimension framework to utilize GADRAS' refinement algorithm. The combination of these two enhancements results in an absolute error within 10% for standard conditions, and 20% for edge case conditions. The addition of arbitrary models will simplify the modeling process for complex shapes, allow for more flexible models, and allow for creation of models that are simply impossible in the current framework.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Nuclear radiation transport simulation is an invaluable asset in design when radioactive materials are involved. The governing principles of the emissions of radioactive materials are complex and depend not only on the material itself but also on other objects around it. These phenomena are generally simulated before experimentation is performed due to the high cost associated with working on and around radioactive material.

A more specific goal for radiation transport simulation is to capture a detector's response to a scene containing radioactive material. This ability allows experimental detectors to be tested in a simulation environment before an expensive prototype is created and tested. Additionally, obtaining a detector response in a simulation environment allows a user to test an existing detector under unique conditions that may be difficult to achieve in a lab setting.

1.1 State of The Art

There are two general categories of simulation that exist to solve this problem: stochastic and deterministic. The most widely used and trusted simulation package falls in the stochastic category: The Monte Carlo N-Particle (MCNP) transport code (LANL, n.d.). MCNP is a particle-based simulation technique that tracks an individual emission from a radioactive material until it leaves the scene or is somehow annihilated. The user must input material properties and source term descriptions in an input file, in addition to describing the geometry using shape primitives and Boolean operations to detail the scene. MCNP then creates particles based on the statistical descriptions of the materials in a random fashion until it has reached the number of particles dictated by the user. The user can then determine if the number of particles selected was sufficient for convergence of the problem

based on the output statistics from MCNP. For complex scenes, it is not uncommon for billions of particles to be required for good convergence. Due to the large number of iterations that must be tracked, MCNP is generally run on supercomputers to reduce the real time needed to solve a problem.

In the deterministic method of simulating radiation transport, the Boltzmann Transport equation is solved (Brunner, 2002). The transport equation is solvable for simple geometries where symmetry approximations and other simplifications apply; however, for more interesting scenes, the equation cannot be directly solved. One of the ways deterministic transport simulation programs solve the transport equation is to create a system of linear equations and approximate the solution by solving that equivalent system. Again, these systems are nontrivial to solve and the best way to decrease the time to solve is by utilizing supercomputers

An additional pseudo-deterministic way of solving the transport problem is by discretizing the scene into many one-dimensional approximations. A deterministic method is then used to solve the collection of 1D problems individually, and they are then recombined for a total solution. This is the approach taken by Gamma Detector Response and Analysis Software, or GADRAS. A user creates the desired scene using a collection of shape primitives (spheres, cones, cylinders, boxes, round-end cylinders, and split spheres) to model the desired geometry. Then, GADRAS approximates the primitives as point sources by discretizing them, if they are radioactive, into voxels, based on differences in five criteria: outside model, self-attenuation, geometric attenuation, external attenuation, and source term gradient. The outside model allows two shape primitives to overlap when building the scene, and the overlap region is not counted twice. One of the models attempts to match the surface of the other model with incremental refinements. The other four criteria refine based on the differences in their respective properties across each of the

voxels. Voxels within GADRAS are defined to have three dimensions that can be used to construct the voxel. For example, a spherical voxel has the three dimensions of radius, theta, and azimuth, and a box voxel has the dimensions of length, width, and height. The algorithm checks each dimension for differences based on the criteria and splits voxels accordingly. Once voxelized, GADRAS traces a ray from the approximate point source to the detector to determine the materials the ray passes through. Using those material properties, GADRAS determines the contribution of that source to the detector and adds it to the solution from the remaining points to determine the total detector response. GADRAS also has additional functionality to customize the detector, account for detector parameters, and create a spectrum that can be matched to data from lab experiments.

1.2 PROBLEM STATEMENT

One of the main limitations of GADRAS is that models are required to be built with the provided shape primitives. One can imagine more complex models that would be of interest in nuclear engineering, such as a torus, that would require many shape primitives to create even an approximation. In addition to complex continuous surfaces, generic complex scenes are cumbersome and time consuming to build. In addition to the time required to build the model initially, if an object is moved within the scene, every primitive needs to be adjusted individually to create this change in the object.

These approximations and inflexible models give rise to the following research task:

> **Develop a method for processing generic geometric models in GADRAS.**

A generic CAD model implementation would allow simulation of the transport problem for complex surfaces and scenes and increase the ease of working with large

scenes in GADRAS. This would provide models that are cleaner, easier to interpret, and ultimately save researchers time and money.

A solution to implementing generic CAD models in GADRAS needs to work within the framework already provided to be compatible with the existing functionality. This means a solution should be fast to ray trace and follow the same 3-dimensions rule outlined above. Additionally, it should be represented in a widely used import format to provide a CAD software independent solution.

1.3 PROPOSED SOLUTION

This thesis examines three problems that need to be solved for generic implementation of CAD models in GADRAS: a generic input file type, a fast way to ray trace the geometry represented, and a generic, model-independent voxelization method. In the next chapter, a literature review describes methods currently used for addressing each of these three problems. Chapter 3 discusses implementation of octrees for fast ray tracing of STL files, the chosen generic geometry representation. Chapter 4 discusses implementation of tetrahedral meshes in GADRAS' existing framework. In Chapter 5 generic models are compared with their respective shape primitives, and a complex scene is built and compared to an equivalent scene built using shape primitives. Chapter 6 summarizes the research and points to directions for future research.

Chapter 2: Literature Review

This chapter examines the current file format standards, methods for solving ray-triangle intersections, improvements to improve ray tracing of models consisting of large numbers of elements, and volumetric mesh generation. The methods are examined for their usefulness with respect to implementation for the problem this research seeks to solve.

2.1 Model Representation

The desire to represent an arbitrary CAD model has resulted in proposals for many different exchange formats as the industry standards. Some of the more notable file formats are STL, STEP, and IGES.

One of the earliest attempts to standardize CAD models across vendors is the IGES (for Initial Graphics Exchange Specification) file format. The IGES standard specified a vendor neutral format that is used to define a 3D model. This format was released by National Bureau of Standards (now National Institute of Standards and Technology) in 1980 (ANSI, 1982). Similarly, the STEP (for Standard for the Exchange of Product model data) standard, generally called ISO-10303-21, specifies a CAD model using a collection of operations to define the surface and was released in 1994 as an attempt to standardize CAD models ("Sustainability", 2017). Both file types have remained relevant with STEP becoming the more common type, and IGES falling off in more recent years.

In addition to the attempts at international standards for CAD files, there are more application specific file types such as SolidWorks .PRT, Catia's .CATPART, and AutoCAD's .DXF. However, there is no consensus across CAD programs to use one single type, and STEP is currently the most widely used exchange format for general CAD geometry.

This research requires a format that is both universally available and computationally simple. This leads to consideration of tessellated models, specifically the STL file, which is adopted for this research. The STL file format is a triangulation of the surface of a model, where each facet is represented by its three vertices and a surface normal. The STL format was first proposed by 3D Systems in 1987 (3D Systems, 1988) for use in their Stereolithography Apparatus, the first commercially available 3D printer. The STL format has become the de facto standard for communicating geometry information for additive manufacturing, and its ubiquity has led to its adoption for many other applications. There have been separate attempts to expand on the format by Hiller and Lipson (2009), who attempted to add material properties, and Stroud and Xirouchakis (2000), who attempted to simplify the format, respectively. Expansion of the format by Hiller et al. reduces the simplicity of the STL format by adding complexity where it is not needed. The attempt by Stroud et al. to simplify an already simple format resulted in faster construction of an STL file, not a simplification of the format.

2.2 RAY-TRIANGLE INTERSECTION

Given that a triangulated format is used for this research, a method is needed for computing the intersection between a ray and a triangle. Traditional ray-triangle intersections have been done by substituting the equation of the ray into the equation of the infinite plane of the polygon and solving for the point of intersection, and then determining if the intersection point is contained within the finite polygon. The point-in-polygon test has many different variations, such as the crossing test, Weiler angle method, and a gridding method, outlined by Haines (Haines, 1994).

These traditional methods are generally considered slow and memory intensive because of the need for both the vertices and normal of every triangle, where the normal

either needs to be calculated or stored. Algorithms exist that can solve the intersection problem without the surface normal, such as those presented by Möller and Trumbore (2005), and Badouel (1990). These algorithms take advantage of a transform of the triangle into barycentric coordinates, creating a system of equations that can be solved with only knowledge of the triangle vertices. Additionally, Shevtsov et al. (2007) eliminated the surface normal by translating the triangle using Plucker coordinates and solving the resulting problem. Both methods decrease the storage and calculation times of the ray-triangle intersection; however, the algorithms are based on the assumption that the normal is expensive to calculate. In this application the normal is read directly from the STL file and stored. Storing the normal does increase the memory footprint of the ray-triangle intersection algorithm, but in this case, the surface normal is needed throughout the processes as a sense of direction for the surface, so eliminating the normal complicates the problem for other operations.

Another opportunity for speed increases in ray-triangle intersections proposed by Reshetov et al. (2005) is to group the rays into a beam, and ray trace with the beam to eliminate entire sections of the scene. This proposed solution excels in large scenes with many sources of rays, or very large, complex scenes. Although this approach may be useful in the graphics industry, many nuclear applications have a limited number of sources, decreasing the likelihood that rays can be grouped. GADRAS' solution method is highly directional and already limits the number of rays cast for every point source to a single ray directly to the detector face.

2.3 Ray Tracing Data Structures

The naïve approach of checking every element in a tessellated polygon scene against every ray is slow and leaves many opportunities for optimization. Many different

data structures have been proposed to sort and bin the polygons to improve efficiency and speed. One of the simplest proposed methods is to bound each object into hierarchical lists. Kay and Kajiya (1986) propose to bound each convex hull of an object into hierarchical lists. The bounding volumes for all the convex hulls of the object are stored together. If the bounding volume for the entire object is intersected by the ray, then all the convex hulls are searched. This method has two major advantages over the naïve approach: early culling can be performed for an entire object if it is not in the field of view of the ray, and each subset of the object requires a simple search of a convex object. By forcing each section of the object to be convex and using the triangle connectivity information, the ray intersection for that section of the object is greatly simplified. In the application of GADRAS, this approach increases the precomputing needed to create the convex sets of every object. Finding the convex hull of a set of triangles is $O(n \log(h))$, where n is the number of points in the set and h is the number of points in the hull, in complexity using Chan's algorithm (1996). This additional time is in addition to the time required to search the hierarchal list, and then ultimately to search the convex hull for the intersections.

One attempt at building a binary tree to hold data for searching is the k-d tree proposed by Bentely (1990). The k-d tree is a collection of nodes that contain two pointers to the next two nodes, or a null pointer and the data for retrieval if it is a leaf node. A node is defined as two half spaces created by an infinite plane. The half space can be recursively divided until the object to be searched is adequately partitioned. The binary tree structure created by forming the k-d tree can be searched in $O(\log(n))$ time. The proposed data structure relies on a relatively sparse object to be sorted. If there is a high concentration of elements in one area, the tree becomes extremely unbalanced. Additionally, the structure defines the object in infinite space. If more than one object is in the scene, both trees require

a search for every ray. This can simply be addressed using bounding boxes for the objects and searching for collisions with the bounding boxes first.

Both problems with k-d-trees can be solved using the octree as proposed by Glassner (1984). This special subdivision defines hierarchal bounding boxes across an object that recursively divide into eight child nodes to define the space. This structure solves the problem of highly concentrated elements by creating a higher number of nodes in each level of the tree for locations where objects are concentrated. Although this increases the complexity of the search, the trade-off for a more balanced tree creates a more consistent search time for the octree. Other researchers have attempted to expand on the idea of an octree. Brown (1998) examines the potential for rectangular octants to better adapt to the model. Sundar et al. (2008) use a novel encoding scheme and a bottom-up population strategy to optimize the balance of the tree.

2.4 MESH GENERATION

GADRAS uses a volumetric mesh to determine the point source approximation of the sources in a scene to solve the transport problem. The preexisting meshing algorithm takes advantage of the geometry of the object, and is therefore specific to the geometric primitives supported, creating the need for a more robust meshing algorithm for arbitrary models. Tetrahedral elements are commonly used for volume meshing of arbitrary models due to their ability to easily fill any volume. One technique proposed by Rebay (1993) enhances the Bowyer-Watson algorithm that expands Delaunay triangulation into 3D space to form tetrahedra. Rebay proposes a more efficient method by taking advantage of simultaneous computation of point positions and connections within the void space created in the model through Bowyer-Watson. Both this new method and the traditional Bowyer-Watson algorithm use Delaunay tetrahedralization to create tetrahedra that are as regular

as possible. Regular elements are not useful in the nuclear application of GADRAS, as GADRAS aims to minimize the number of elements in a mesh to decrease the number of rays created to solve the system.

Löhner (1988) proposes another solution aimed at creating regular meshes, specifically for transient fluids. His method uses the triangulation of the surface of a volume to create a marching front. This front is reduced as the algorithm converges to the center of the model. This algorithm can be modified to ignore the bias for regular elements; however, the algorithm then has issues with nonconvex objects. The regularization aspect of the algorithm helps fuse different sections of the model together in the best way possible, and by eliminating this portion of the algorithm, to reduce the number of elements, the algorithm will have difficulty converging.

Below et al. (2000) show that finding the minimum number of elements to represent an arbitrary tetrahedron is NP-hard. From this information, we can assume that an attempt to find the minimum number of elements, although possible, is not worth the trade-off when attempting to maximize speed of the algorithm.

2.5 SUMMARY

In order to capture a CAD model with the highest accuracy, many operations need to be represented, giving rise to complicated file types. On the other hand, simple types, such as the STL file type, exist but add a level of approximation. There are many attempts to bridge the gap; however, the best practice is to use the one most suitable for the problem. In this case, the STL file is used for its simplicity and universality.

Speed increases in ray tracing generally come from hardware and parallelization, but a few researchers have developed data structures and unique search methods to speed

up ray tracing algorithmically. Most notably to this thesis is the Octree, which has been implemented, as described in the next chapter.

Discretization of a problem space, for instance, for finite element analysis (FEA), is a vast field with many different methods and algorithms; however, for this application a minimum element mesh is desired. This is an open field with few researchers trying to solve minimum stacking problems, but no good solutions have been found. Because of this, going forward this research merges tetrahedral splitting and initial meshing from FEA discretization for use in nuclear simulation.

Chapter 3: Surface Modeling Implementation and Results

This chapter describes the proposed method of quickly ray tracing large numbers of triangular elements. The following sections outline the implementation of an octree, a 3D spatial binning structure in GADRAS and the resulting speed and edge case analysis. The implementation assumes the input file is in the STL format; however, these results can be generalized to any convex polygonization of a 3D surface.

3. 1 Ray-Triangle Intersection

As stated in Chapter 2, the selection of the STL format for the input surface model is based on both the universality and simplicity of the file type. The rise of 3D printers and the use of STL files in their slicers has led every CAD vendor to provide the capability of producing STL files and has resulted in the creation of online repositories of STL files to share designs universally. Additionally, the nature of the STL format, a tessellation of surfaces into polygons, allows for the generalization of the code for any arbitrary convex polygonal tessellation that may arise.

In addition to ease of access of STL files, their simplicity allows for easier calculations. As discussed earlier, an STL file is a collection of unordered triangles represented by three corner vertices and a normal vector. A triangle is a bounded plane. Calculation of the ray-plane intersection is well documented. The parametric equation of the ray is given by:

$$\vec{P} = \vec{P}_o + t\vec{V} \tag{1}$$

where $\vec{P}_o$is the origin of the ray, $\vec{V}$is the direction of the ray with respect to $\vec{P}_o$, and $t \in \mathbb{R}, -\infty < t < \infty$, defining the infinite line containing the ray. If the direction $\vec{V}$ is a unit vector, then t is the distance (positive or negative) between the origin of the ray and the point of intersection with the plane. The implicit equation of a plane is given by:

$$\left(\vec{P} - \overrightarrow{P_1}\right) \cdot \vec{N} = 0, \tag{2}$$

where $\vec{N}$ is the normal of the plane and $\overrightarrow{P_1}$ is any point known to lie in the plane. Without loss of generally, $\overrightarrow{P_1}$ can be any of the three vertices of the triangle. To determine the point of intersection of the ray with the infinite plane, the parametric equation of the ray is substituted into the implicit equation of the plane:

$$\left(\vec{P_o} + t\vec{V} - \overrightarrow{P_1}\right) \cdot \vec{N} = 0. \tag{3}$$

This equation can be solved for the value of the parameter t at the point of intersection:

$$t = \frac{\left(\overrightarrow{P_1} - \vec{P_o}\right) \cdot \vec{N}}{\vec{V} \cdot \vec{N}} \tag{4}$$

This holds true if the denominator is not equal to 0, which is the case for a non-parallel plane and line. This provides an easy method to determine if the intersection point is reasonable distance from the ray origin by comparing the absolute value of the denominator to a tolerance.

Once the point of intersection is calculated, the next step is to determine if the point lies within the bounds of the triangle. This can be done by showing that

$$sign\left[\left(\overrightarrow{V_{pl}} X \overrightarrow{V_l}\right) \cdot \vec{N}\right] \tag{5}$$

is equal for i = 1,2,3 where the variables are defined in Figure 1. This leads to the first challenge of working with piecewise representations of surfaces. If the point of intersection lies on one of the edges or vertices of the triangle, then Equation 5 will evaluate to 0. If we consider an intersection on a line or vertex to be outside of the triangle, then we could potentially miss an object intersection. Thus, we must consider a hit on the line to be in the triangle; however, if we do this, the ray will be considered to hit multiple triangles, as each edge is shared by two triangles in a valid STL file. To resolve this, a list of hit points is

created for each object on a per ray basis. If an intersection occurs twice, the second one is ignored.

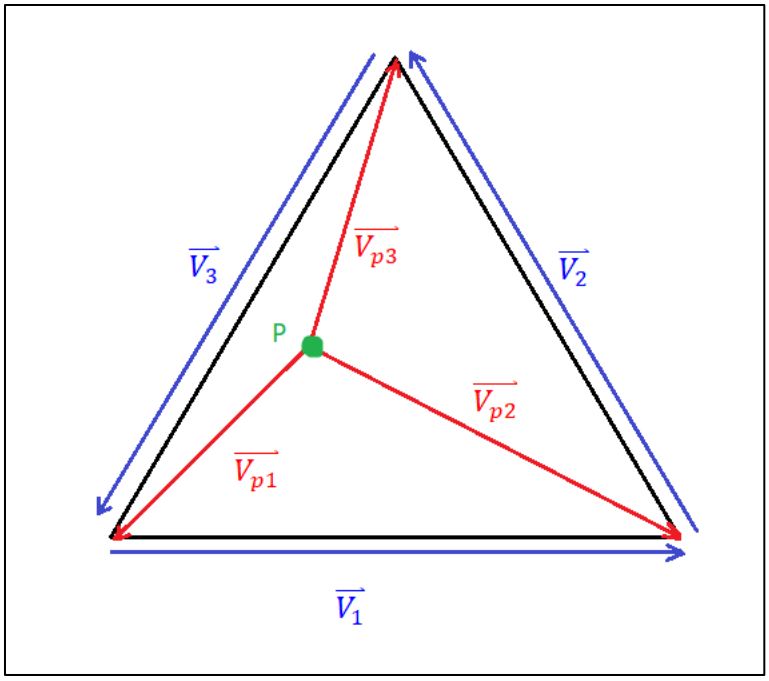

Figure 1: Point in polygon vector definition

Using an STL file allows us to ray trace an arbitrary surface with relative ease. The main issue now, as discussed previously, is the number of elements in an STL file. A file can range from a minimum of four triangles to an extremely large number depending on accuracy and model complexity.

3.2 Octree Implementation

Directly ray tracing all the elements of a model represented in the STL format, results in an algorithmic complexity of O(*nm*), where *m* is the number of rays and *n* is the number of elements (triangles) in the STL file. Nonconvex models (e.g., models with depressions or holes) create a situation where a ray can enter and exit the model any number

of times. Since an STL file is unordered, connectivity is not given explicitly. Thus, every element must be searched to ensure that every intersection is found. This means that either the number of elements searched, or the number of rays created, must be limited to increase the speed of the search.

The proposed solution to this problem uses an octree to take advantage of spatial coherence to reduce the number of elements searched, resulting in an increase in the speed of the operation. To do this, an octree is used to bin the elements of the STL file into a 3D recursive data structure. A 2D representation of the network can be seen in Figure 2. Each element, or node, of the octree comprises six planes defining the boundary, and each node has knowledge of its contents and any lower nodes. A split is defined to be the division of a node into eight equal sub-nodes, or children. An octree also has tunable parameters of maximum depth and fill limit. Maximum depth is defined as the maximum number of times a node can be split, and fill limit is defined as the maximum number of elements in a node before a split occurs.

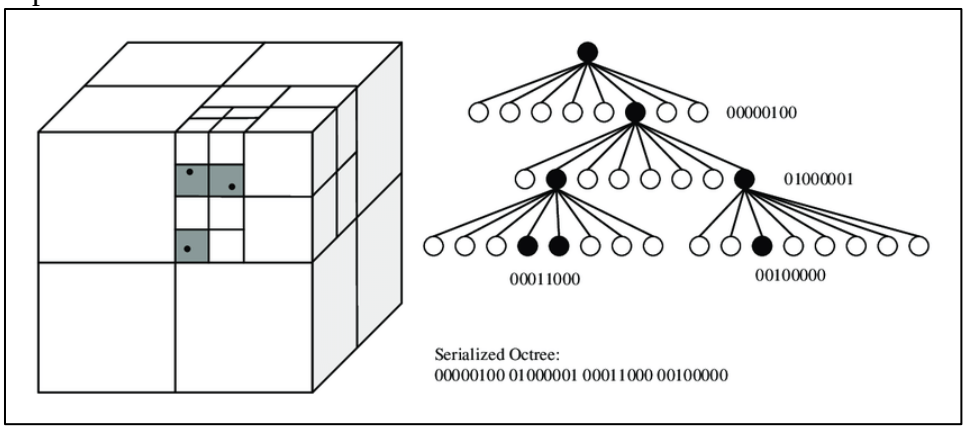

Figure 2: Simple Octree Representation (Dybedal, 2019)

The criterion for addition of an element to a node is that it is either wholly or partially contained within the node. This test is performed by determining the relative

position of each of the three vertices with respect to a plane. If all six infinite planes of the node contain at least one vertex of the element, then the element intersects the node. A vertex, $\overrightarrow{V}$, is defined to be contained by the plane, with normal, $\vec{N}$, and point, $\vec{P}$, if Equation 6 is true.

$$\vec{N} \cdot \left(\vec{P} - \vec{V}\right) > 0 \tag{6}$$

The method used to populate the octree in this implementation is to add elements one by one to a head node until the fill limit is reached. Once the fill limit is reached, the node splits and the elements contained within it are added to the child nodes based on the criterion for addition. If both the depth limit and fill limit are reached, the last node of the tree will be overloaded with more than the fill limit of the tree.

To search the tree, a ray must retrieve all the lowest level nodes it intersects and then search their contents. To retrieve the lowest level nodes, a ray is intersected with the head node to either begin the recursive process or eliminate rays that have no chance of intersection with the model. The node-ray intersection is performed by checking all six faces of the node as planes and checking the sense of the resulting intersections against the remaining planes based on Equation 6. Then, if a node is found to be intersected, the ray is checked for collisions against the node's children recursively until all nodes with zero children are found. From these nodes, a list of elements with collision potential with the ray is compiled and searched in the manner outlined in section 3.1.

3.3 Algorithmic Complexity Analysis

The results of using the octree method to reduce the number of elements searched increases the speed of the algorithm. The previous complexity of O(nm) is now reduced to O(Xm) where X is the reduced list of elements used for an octree search. This section defines a value for X.

3.3 1 Worst-Case Analysis

The worst case for intersection of a ray with an octree with equal depth in every node is intersection of the head node on opposite corners, as shown in Figure 3. This situation gives rise to the situation where the most nodes need to be checked against the ray. This worst-case scenario ray intersects the *N* unique nodes in an octree of depth, *d*, as defined by:

$$N = 6(2^{d-2} - 1) + 8(2^{d-2}) \tag{7}$$

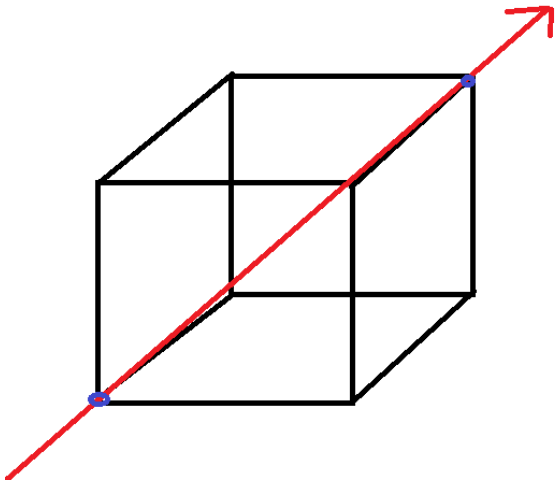

Figure 3: Worst-case ray-node intersection

For every node intersected, the algorithm will check n_d elements, all the triangles contained in a node, as well as all six of the node faces, for ray intersections. Each node face intersection can be approximated as a single element intersection for the sake of this analysis. If we assume that every node is at maximum fill depth as governed by the properties of the tree, the intersection of the model by *m* rays is O$[(n_d + 6)Nm]$. This analysis shows that the complexity of the algorithm is dependent on only the depth and maximum fill of the tree. If we attempt to characterize the depth of the tree by assuming

the depth is a function of total number of elements, *N*, and maximum fill, n_d, where the tree is perfectly balanced and at maximum depth, the depth is given by:

$$d = log_8\left(\frac{N}{n_d}\right) \tag{8}$$

This equation can produce non-integer values, where depth must be an integer, so we round up this value to avoid a depth of zero.

By combining Equations 7 and 8, the complexity of the algorithm becomes:

$$(n_d + 6)\left[6\left(2^{log_8\left(\frac{N}{n_d}\right)-2} - 1\right) + 8\left(2^{log_8\left(\frac{N}{n_d}\right)-2}\right)\right]m \tag{9}$$

By simplification of constants, the complexity can be seen to be a linear combination of three terms:

$$\left[C_1\left(2^{log_8(N)}\right) + C_2\left(2^{log_8(N)}\right) + C_3\right]m \tag{10}$$

The highest order term dominates the solutions and becomes the complexity of the problem, in this case $2^{log_8(N)}$. This term can be rewritten in the form N^X through use of base changes and logarithmic properties. This results in the complexity of the problem being $O\left[N^{\frac{1}{log_2(8)}}\right]$.

However, in real applications the elements of an STL file are not uniformly distributed, and although maximum fill limit is a tunable parameter set by the user, once maximum depth is reached, the nodes are capable of overfilling.

3.3 2 Real Analysis

If the tree is perfectly balanced and at maximum depth, the above worst-case analysis is valid. In our modeling use case of an octree, a uniform distribution of elements is almost never achieved, thus we cannot assume that the tree is perfectly balanced. The balance of the tree is related to the distribution of the triangles used to define the elements of the STL file. As uniformity increases, the complexity of the algorithm approaches the

value discussed above as a lower bound. For analysis, assume that $n_{d,real}$ is a function of the uniformity of the triangular distribution of the model. If uniformity is low, $n_{d,real}$ can be assumed to be equal to n_d. For the worst case, $n_{d,real}$ is equal to the total number of elements, divided by *N* as defined in Equation 8. This means that for a real-world analysis, the complexity of the algorithm is bound between $O\left[N^{\frac{1}{3}}m\right]$ when uniformity is high, and O[Nm], when uniformity is low. From these two bounds, it is shown that the octree is in the worst case equal to brute force search, and in the best, and more likely, case, better than brute force.

3.4 Assumptions and Limitations

The main limitations imposed on the octree come from three main assumptions: (1) the model is relatively uniformly distributed through space; (2) the surface approximation of the STL file is within the tolerance of the problem; and (3) the boundary collisions of rays are within the element.

As shown in Section 3.3, assuming the model is uniformly distributed is paramount to the speed increase gained by using an octree. This means that a model with a high degree of asymmetry will not benefit from the octree. With a higher number of elements on one side of the bounding box, the octree will hit its fill depth and over fill the leaf nodes. One example of a model that creates an imbalanced tree based in this implementation can be seen in Figure 4. The decrease in the speed of the algorithm due to an unbalanced octree can be combated by creating design guidelines to follow when using the modeling functionality of GADRAS. Although the software will still give a correct answer, processing time can be reduced by representing asymmetrical models as multiple separate models. For example, a multi-model decomposition of the asymmetric model of Figure 4

can be created by forming the circular end of the pendulum separate from the long neck, creating two symmetric models, with more well-balanced trees.

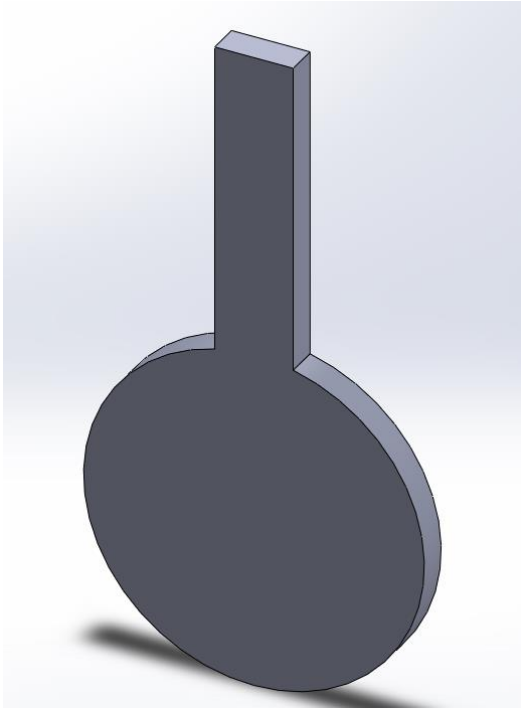

Figure 4: Asymmetric model resulting in unbalanced tree

The approximation of the STL file can be characterized by comparing the volume of known geometric shapes to the volume of their STL approximations. Table 1 shows the percent volumetric difference between six geometries and their corresponding STL files. The STL files were generated using SolidWorks, using the default coarseness value. As shown in the Table 1, the highest error is for a Cylinder with a value of 0.5% This volumetric error is comparably created by changing the radius by 0.02 units These remarkably low values show that STL triangulation should not be a major contributing factor to the overall model error.

Table 1: STL comparison of volumes

Shape	STL Volume	True Volume	% Error
Sphere	4177.36	4188.7902	0.27%
Cube	1000	1000	0.00%
Cylinder	3127.3	3141.59265	0.45%
HalfSphere	2089.47	2094.3951	0.24%
Cone	1045.22	1047.19755	0.19%
Round End Cylinder	7312.9	7330.38286	0.24%

3.5 SUMMARY

Throughout the implementation and debugging process, rays intersecting a boundary have proven to be an issue many times. This led to many features that attempt to mitigate the issue and subdue its contribution to the overall model error. As discussed previously, when intersecting a model, if the same collision point occurs multiple times, it is only recorded once. Additionally, there is a tolerance that can be adjusted to account for floating point errors when determining if two points are coincident. During the point-in-polygon test for points contained in a model, an issue arose where the collision was a vertex, or on an edge of the polygon. This was mitigated by allowing for one or both signs to be zero. Finally, if all else fails and a model is only intersected an odd number of times by an infinite ray, the information from that ray is discarded. The error created by a ray that is infinitely contained in a model before leaving the model is much more devastating than the loss of information created by discarding the ray. This is a known source of error in the model, but it should occur at a low rate due to the other factors attempting to mitigate the boundary collision errors imparted by the discretization of the surface.

Chapter 4: Voxelization

This chapter describes the process for generating a mesh of tetrahedral voxels to fit within the three-dimensional framework of GADRAS. Tetrahedral meshes provide conformity to arbitrary surfaces. Also, a tetrahedron can be split into two tetrahedra in a self-similar manner. The ability to create arbitrary self-similar tessellation is useful for defining arbitrary geometries; however, fitting tetrahedra into the framework of being defined by three fundamental dimensions (detailed in section 1.1) and ultimately the framework for GADRAS necessitates some creativity.

4.1 Tetrahedral Voxel Definition

A tetrahedron can be split by a new vertex in three ways: a new edge vertex, a new face vertex, and a new interior vertex. These three methods result in two, three and four children tetrahedra respectively, as shown in Figure 5. By choosing to split into two elements, tetrahedral voxels parallel the other voxels implemented in GADRAS. When one of the GADRAS voxel types is split, it is bisected along the dimension of interest. This same idea can be applied to tetrahedra if they are defined using three dimensions. A tetrahedron, ABCD, divided by point O, as described in Figure 6, is used as an example for subsequent definitions and explanations.

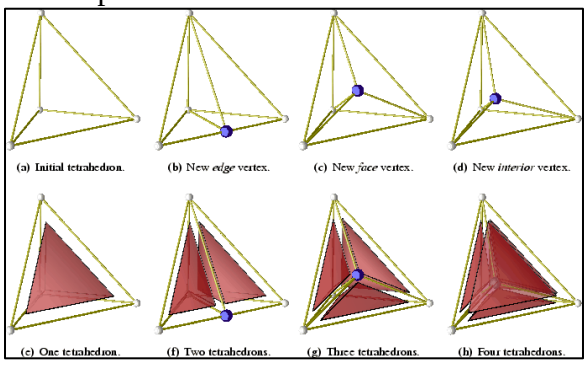

Figure 5: Tetrahedral Divisions (Wessner, 2006)

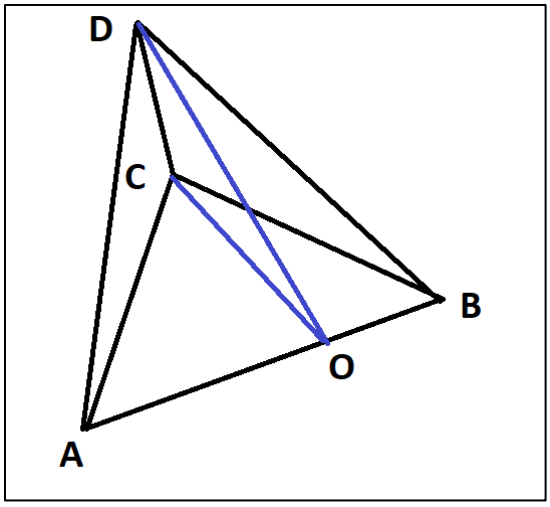

Figure 6: Prototype Split with Labels

Based on the idea that a two-element split results from splitting a tetrahedron across a dimension, pairs of features can be identified as potential candidates used to define the meaning of a dimension for this geometry. The elements on one side of the plane used to divide the tetrahedron in Figure 6 are point A, edges AC and AD, and face ACD, while the elements on the other side are point B, edges BC and BD, and face BCD. From this we can attempt to pair elements on opposite sides of the split and determine the number of resulting dimensions. The results of this analysis are given in Table 2.

Table 2: Potential Dimensions

Elements	Relation	Quantity in Tetrahedron
A; B	Point pair	6
AC; BC	Edges with shared vertex	12
AD; BC	Opposite edges	3
ACD; BCD	Faces with shared edge	6

The only pair of elements resulting in three dimensions is opposite edges. Although other voxel types use their three dimensions to fully define the geometry, a tetrahedron using this opposite edge approach does not. Instead, these dimensions can be used only as a means for refinement. The framework provided by GADRAS dictates that a dimension must be defined by a direction and two points. For the case of opposite edges, the direction is a normalized cross product of the two edges, and the two points are defined as the midpoints of the two edges. One issue with this framework is that, as the tetrahedron is divided, the dimension is dependent on the two edges that are used. As the tetrahedron divides, the order of the points provided changes. One method to solve this is to be consistent in providing the points to the constructor by creating a scheme for labeling the points. Because the tetrahedron can change its orientation in space as it is divided, and is not guaranteed to be regular, this becomes very difficult. Instead, the points are sorted within the constructor to keep some form of homogeneity among the dimensions. The scheme chosen is to sort the four points from high to low in the absolute coordinate system by X value, then Y value, and Z value. By doing this, the dimensions are in the same general direction and provide a level of order to the construction of otherwise arbitrary tetrahedra.

The next major issue is to determine the vertex to add to the tetrahedron to split the desired dimension. Each dimension has four edges to which a vertex can potentially be added when splitting the voxel. Additionally, the GADRAS framework returns the optimal position to split a voxel in terms of the dimension. Figure 7 outlines the four possible solutions for a split on the dimension defined by the cross product of AC and BD for our prototype tetrahedron. Each option is equally valid and reduces the attenuation seen in the AC:BD direction. Because all options are equally valid, the longest side is chosen in an attempt to keep the resulting tetrahedra as regular as possible. The division point is then calculated by creating a plane using the point desired to split through, and the two points

not on the edge the vertex will land on. Then, the edge to be split is intersected with this plane to determine the split point on the dimension. This is mathematically represented by:

$$P_1 + \left[\frac{abs((P_s - P_3 \times P_s - P_4) \cdot (P_1 - P_s))}{\|P_s - P_3 \times P_s - P_4\|}\right] \|P_1 - P_2\|, \quad (11)$$

where P_s is the point the split is desired to go through, P_1 and P_2 are the points defining the edge being split, and P_3 and P_4 are the remaining two points of the tetrahedron.

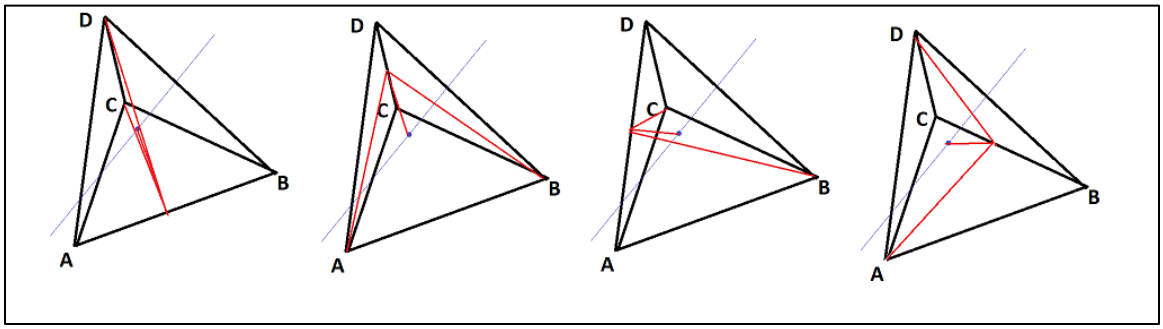

Figure 7: Potential splitting configurations

4.2 Initial Mesh Generation

The initial mesh is generated by determining the maximum length of the model defined by the STL file. This length is then divided into equally spaced segments. A three-dimensional grid of cubes is then formed around the center of the model. The cubes are then divided into five tetrahedra. Once the field of tetrahedra is created, every voxel is checked to determine if the center of the element lies within the model. If it does not, it is removed as shown in Figure 8.

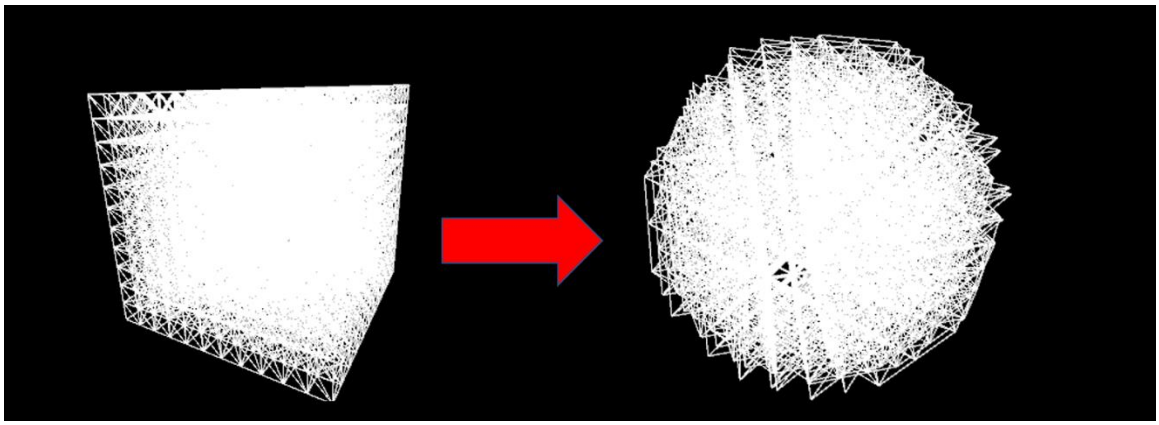

Figure 8: Initial mesh generation of a sphere

The space filling initial mesh generation technique provides a solution that is independent of external software and provides an opportunity to test the outside model refinement detail in section 4.3. Future work will focus on improving the initial mesh generation for a faster, more accurate model.

4.3 Outside Model Refinement

The adaptive mesher in GADRAS allows for overlapping models. This gives rise to a scene with two overlapping models, model A and model B, with model B having higher priority. A voxel of model A can be considered to be outside of model A if it exists in the overlap zone between model A and model B, because model B has higher priority. This happens when composite shapes are formed from the available shape primitives. The generic model implementation must do this in order to be useful with the adaptive mesher and the preexisting shape primitives. The coarse initial mesh presents the opportunity to refine the ability of the tetrahedral voxel to conform to this GADRAS standard. Using the splitting rules defined in section 4.1, the model will never fully be contained in the surface defining the model. When the desired split point is projected onto the edge to be split, the

resulting plane is not guaranteed to completely align with the surface of the model. This creates a need for a special case, where a voxel must conform to a surface intersecting it.

The method implemented to accomplish this is based on the idea that a ray drawn along every edge should not intersect the surface. If it does, the edge needs to be split at the point of intersection to drive the model to conform with the surface. This approach results in expanding the preexisting framework to allow for six dimensions for this special case. For overlapping models of tetrahedral voxel type, every edge is checked for intersection. If it intersects the surface, the point of intersection is directly selected to be the added vertex discussed in section 4.1. This method is slow and expensive as it must execute six ray searches on every tetrahedron on the boundary. A better approach is to avoid overlapping models of tetrahedral type. This is easily accomplished due to the arbitrary nature of a CAD model. A designer can explicitly create the scene to avoid overlaps. This recommendation only works if the initial mesh generation technique creates a mesh that was already completely contained in the surface model.

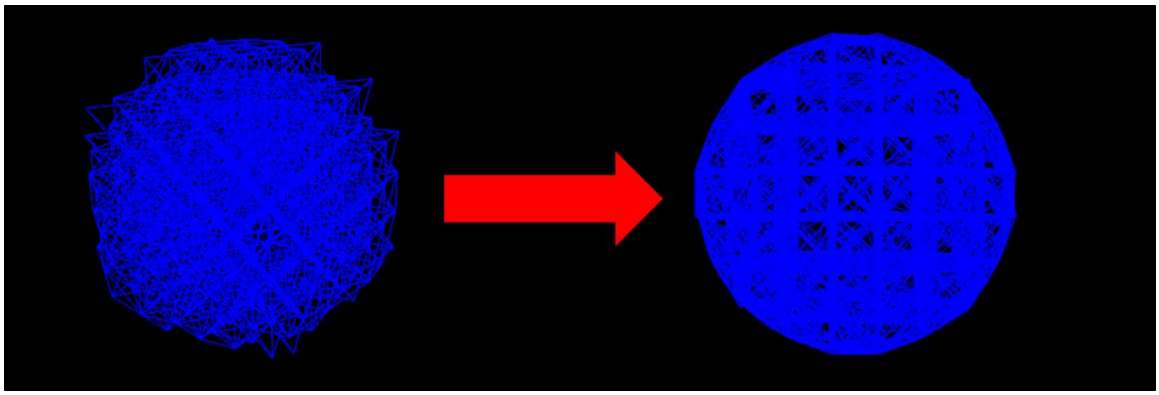

Figure 9: Outside model refinement for a sphere

As discussed previously, the coarse initial mesh generation is the perfect opportunity to test the feasibility of this outside model splitter. Figure 9 shows a sphere

refined to be completely contained within a surface model. The resulting mesh was analyzed using the volume of a sphere of known radius, and the total volume of all the mesh elements. For a radius of 10 cm, a volume of 4188 cm^2 should be seen. The resulting overlapping model has a volume of 4155.61 cm^2. This corresponds to an error of 0.79%. The volume of the tetrahedral mesh is always less than the theoretical volume, because when the edge is split the points are guaranteed to be within the surface mesh. Consequently, all the faces and edges of the tetrahedra must also be on or within the surface. This causes the surface of the volumetric mesh to always be an underapproximation of the true surface mesh.

4.4 REFINEMENT VERIFICATION

This section examines the tetrahedral refinement process for three of the four remaining refinement criteria. The fourth criterion, source term gradient, requires legacy code changes for full functionality. Since this is a proof of concept, this criterion is ignored.

4.4.1 Self-Attenuation

Self-attenuation is the attenuation due to the material the model itself is made of, as the name suggests. To test the tetrahedral refinement process for this criterion, a model is created with an extremely high attenuation. For this model, a cube of attenuation of 100 cm^{-1} was chosen. The model should create an extremely dense mesh on the surface facing the detector, as these voxels have relatively high importance compared to interior models. They become very dense as the mesher attempts to refine the voxels to have the same attenuation at both points defining the dimension. To control all other variables, the other methods for refinement have been disabled. Figure 10 shows the results of the mesh, and

as predicted, a large number of voxels have been created on the surface of the model due to importance and self-attenuation.

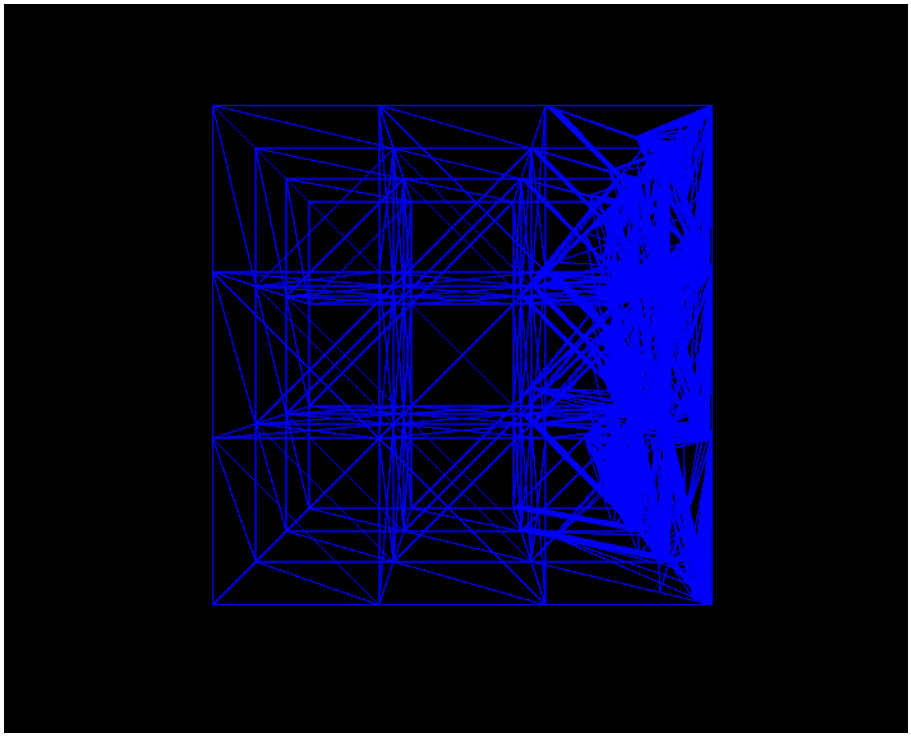

Figure 10: Self-attenuation validation

4.4.2 External Attenuation

The second criterion validated is the external attenuation splitter. This refinement method checks for different attenuation from outside models on each voxel. To see this phenomenon and validate it for the tetrahedral mesh, a slab is created with a sphere between it and the detector with an attenuation of 100 cm^{-1}. The mesher should attempt to split voxels on the edge of the shadow of the sphere, if the detector were a light source. One would expect to see a faint outline of a sphere in the slab's mesh. Figure 11 shows the results of the above situation and validates that this method works for tetrahedral meshes.

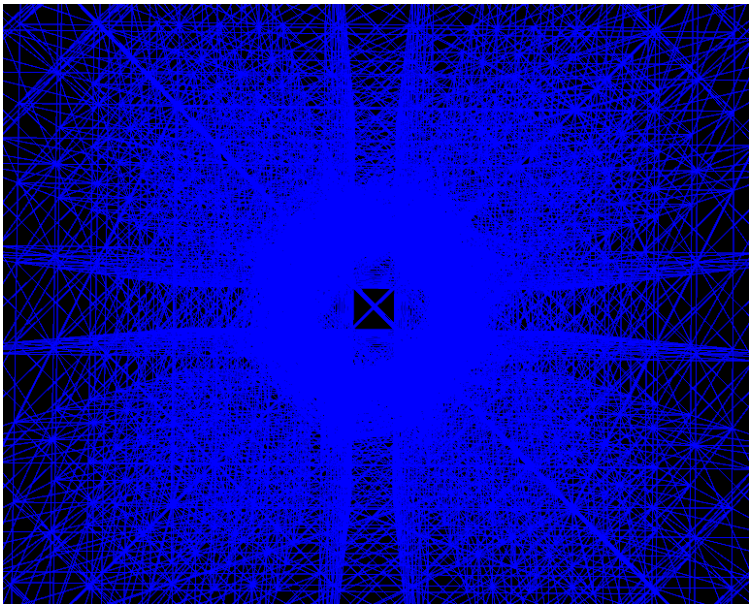

Figure 11: External attenuation validation

4.4.3 Geometric Attenuation

The last, and most difficult to capture refinement criterion is geometric attenuation. This attenuation is based on the distance of the detector from the source. A real-world display of this attenuation occurs with stars. A more distant star is dimmer than one of equal intensity that is closer. To check this refinement method, a detector is placed very close to the model. If the splitter is working, there should be a gradient of splits on the surface, with more divisions as the model gets further from the detector. Being the most difficult to capture refinement method, Figure 12 demonstrates its functionality through the scene described above by placing a detector 0.1 cm away from the surface of a 10 cm cube with a detector at the center of the face. A large number of divisions can be seen on the two elements closest to the detector, and at the point furthest from the detector. This is because the method checks for difference in geometric attenuation across a voxel, and

geometric attenuation only takes places at large distances. Additionally, the nature of this attenuation makes it very difficult to capture.

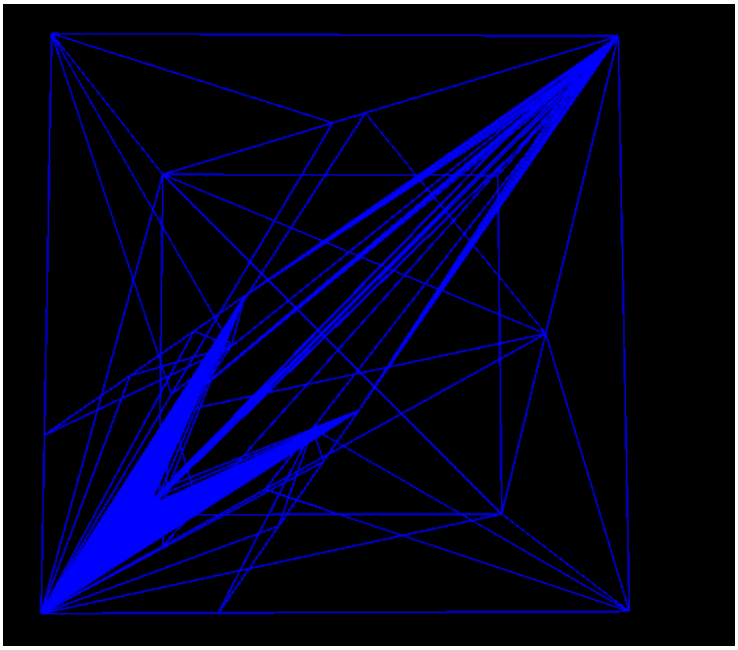

Figure 12: Geometric attenuation validation

Chapter 5: Results and Verification

This chapter outlines the verification process used to determine the effectiveness of the proposed implementation of arbitrary CAD models in GADRAS. The generic implementation is first compared to the GADRAS shape primitives to test error on well documented and experimentally validated geometries. Then, the tetrahedral mesh is compared to the currently existing framework for adaptive meshing within GADRAS to compare the performance of more complex shapes. Next, an entire scene is created using the adaptive mesher and shape primitives to examine a scenario of interest and the potential reductions provided by the tetrahedral mesh. Finally, a model with very complex geometries is examined on its own to demonstrate the benefits of creating models that are impossible to accurately make using shape primitives.

5.1 GADRAS Primitive Comparison

Although the generic CAD model implementation is designed for highly complex scenes and impossible to model surfaces, a comparison to the six fundamental shape primitives in GADRAS allows for verification. The shape primitives in GADRAS have been computationally and experimentally validated. This provides a method of inexpensive comparison between the two implementations.

5.1.1 Experimental Setup

All six primitives (Spheres, Boxes, Cones, Cylinders, Round Cylinders, and Caps) were created using SolidWorks to generate STL files. Each geometry was modeled to have a value of 10 cm for every fundamental dimension defining it. Spheres were defined by the radius, boxes by the length, width and height, cones by the major radius and the length, cylinders and round cylinders by the radius and the length, and caps were forced to be

hemispheric and defined by the radius. The equivalent geometries were also created using GADRAS and the same parameters. Then meshes were generated for both implementations using their respective voxelization. From the mesh, the volume and the center of every voxel was recorded and used to determine detector contribution. Detector contribution is found by:

$$D = V * e^{-\mu_1 t_1 - \mu_2 t_2 \ldots - \mu_n t_n} \quad (12)$$

where V is the volume of the element, $\mu_1$is the attenuation coefficient for the material the ray passes through, and t_1 is the thickness of that material. Then the error was quantified as the percent difference between the GADRAS detector contribution and the tetrahedral detector contribution. The error was then found for every geometry with various mesh sizes and various attenuations at a detector position 30 cm from the center of the model located on the X axis. The mesh settings, and granularity of the initial mesh were held constant for every test, in addition to the source being uniformly distributed throughout the volume.

5.1.2 Results and Discussion

Using the overlapping model refinement initial mesh, the detector contribution was determined for the tetrahedral mesh and compared to GADRAS for attenuations from 0 to 100 cm^{-1}, and mesh densities of 25,000, 50,000, 100,000, 125,000. Some common values for attenuation are 10 cm^{-1} , 55 cm^{-1} and 90 cm^{-1} for a 60 KeV gamma ray for iron, lead, and gold, respectively. Figure 13 shows the attenuation versus error plotted for a cube model. As more voxels are allowed, the error converges towards a value of approximately 8% for all attenuations. This experiment shows that increasing voxel count causes the error to trend towards a single value.

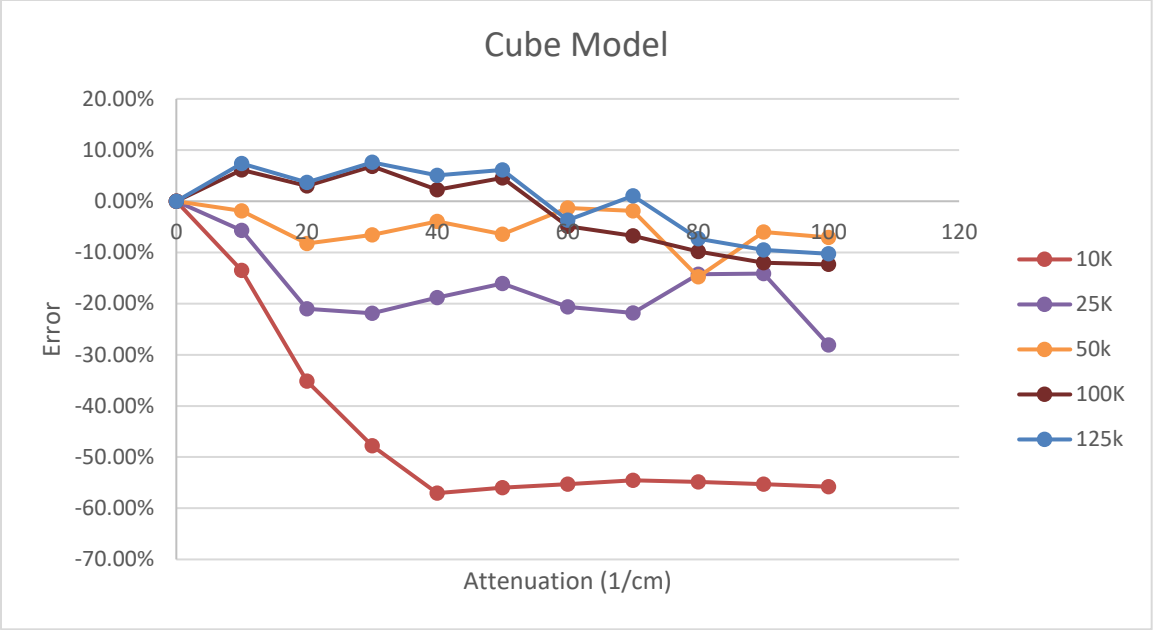

Figure 13: Cube model error

Cubes looked promising, but Figure 14 shows the same progression for a spherical model with drastically different results. The results of this model do not converge to any meaningful value and the error is still significant at attenuations over 10 cm^{-1}. The main difference between the two models causing this error was hypothesized to be the initial mesh. The meshing strategy outlined in section 4.1 uses an initial cubic mesh, which clearly defines the cube model with extreme accuracy. On the other hand, the spherical model must use the outside model convergence routine to refine this cubic mesh into a sphere. This takes many voxels, and still results in a very high error for the volume. Appendix A presents the results of the same test for the other four models which show the trend observed with spherical models holds for all models attempting to refine the mesh.

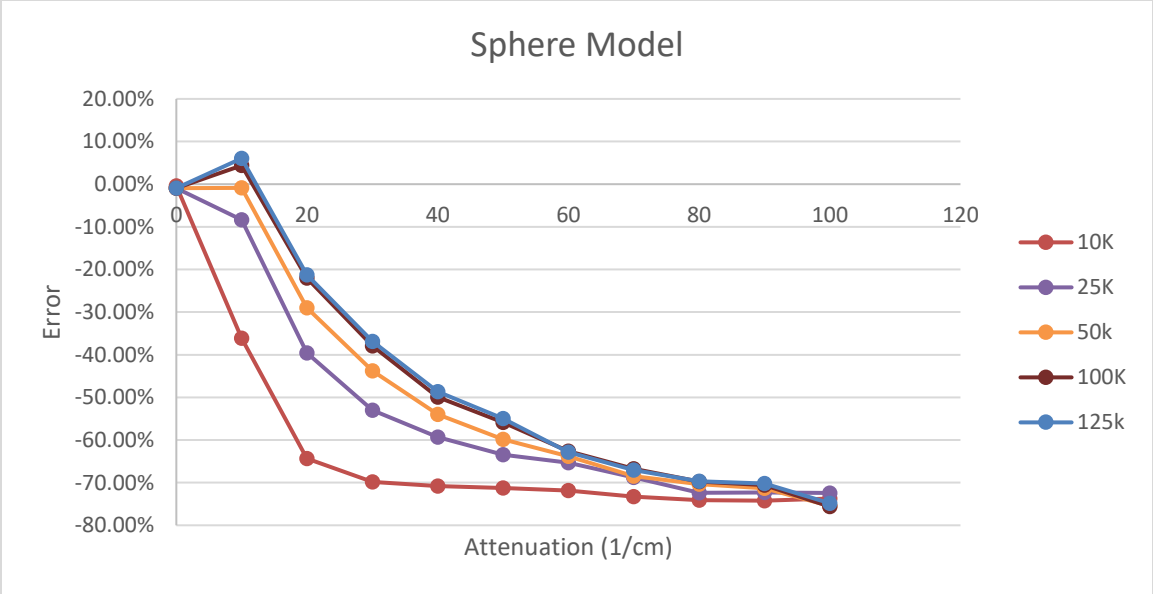

Figure 14: Sphere model error with initial mesh refinement

The idea of having a poor initial mesh translating into large errors can be quantitatively seen by graphing the error at a 100 cm^{-1} against the initial volume error in the mesh as shown in Figure 15. A solution to this source of error is to create a better initial mesh to reduce the effect of this error on the models.

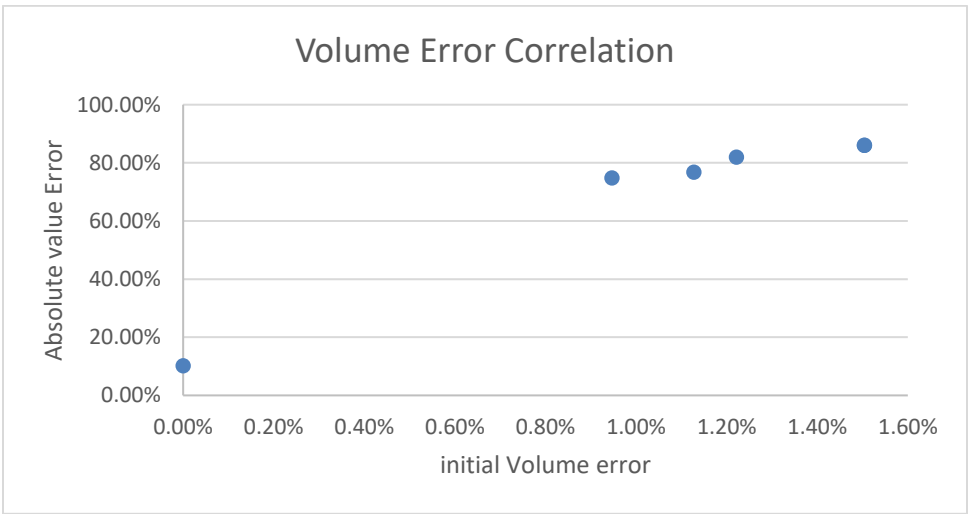

Figure 15: Volume error total error correlation

A simple solution to keep the mesher contained entirely within GADRAS and test this theory is to use convex geometries and create a voxel using the triangulation of the surface for three points and the center of the volume as the fourth point. The results of using this initial mesh for cubes is shown in Figure 16 and for spheres in Figure 17. The cubic model now demonstrates the same trend, with a significantly higher percent error. This is likely due to the nature of the mesh. With the simple initial mesh, the voxels were close to regular and evenly distributed across the geometry. With the alternative method, the elements are thin and irregular, increasing the number of divisions needed to create a point source with the tetrahedron. Spheres on the other hand, now have error that is relatively flat for all attenuations, but as the mesh density increases, the error converges to a high, positive error. This means that the tetrahedral mesh is now over valuing the detector contribution of the model. Appendix B contains the graphs for the other four geometries exhibiting the same effect.

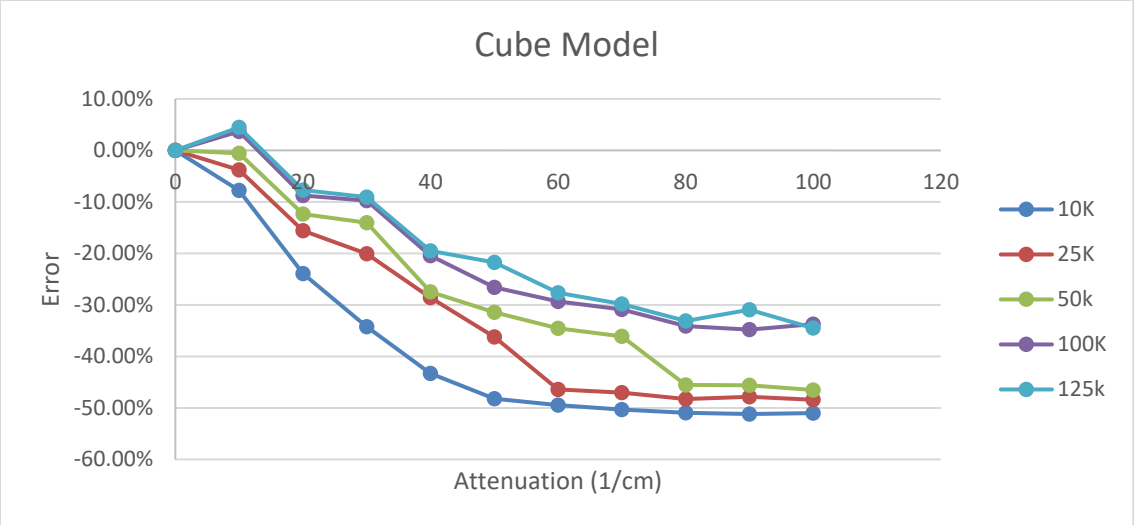

Figure 16: Cube model central-STL mesh error

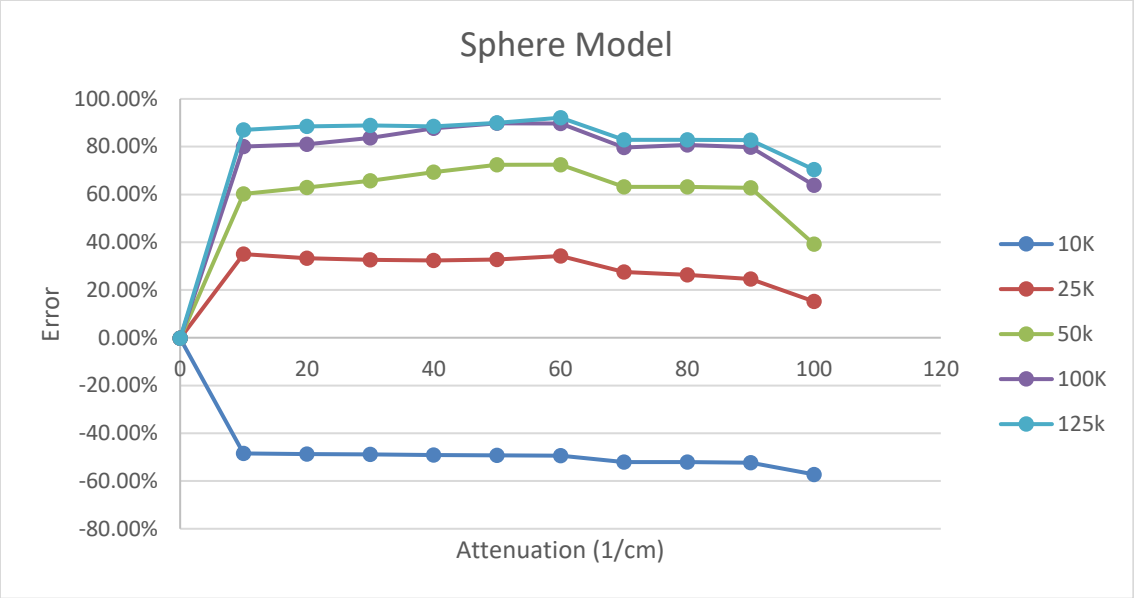

Figure 17: Sphere model error with contained initial mesh

The error shown from the new initial mesh means that an element is now being under attenuated. This is likely due to the approximate nature of the STL file. As the elements are reduced to smaller and smaller voxels near the surface of the model, the likelihood of hitting the surface at an unfavorable angle increases, amplifying the error contributed by the approximation. Figure 18 shows a 2D example of this effect. Ray A and Ray B are both traveling to the same location, but Ray A hits the red approximate surface significantly before hitting the gray exact surface. This error can occur for many reasons, but the most likely is the voxels close to the surface, which have a higher chance of collision occurring at a steep angle, making the approximation even worse and amplifying the effect.

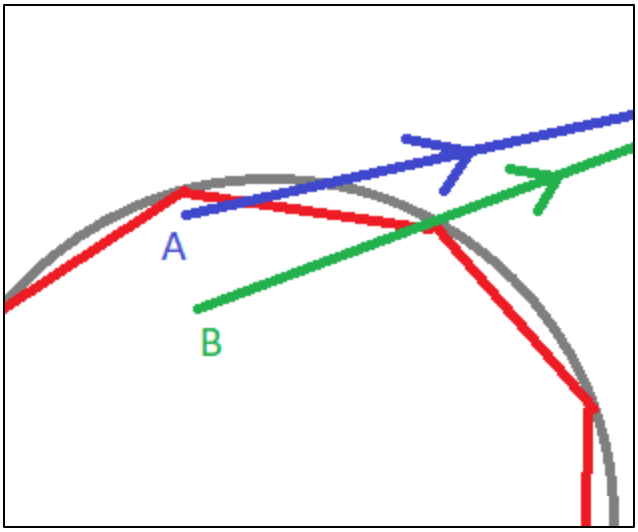

Figure 18: STL approximation error

As a check of this hypothesis, spherical models with decreasing angular tolerance for tessellation (and thus increased STL precision) were tested over a truncated range of attenuations with a constant mesh density of 50,000. The results in Figure 19 show that, as the precision of the STL file (noted as degree for the angular tolerance) increases, the error converges towards a negative value. The error shown in Figure 19 is a combination of all the sources of error. The error resulting from the STL precision is a positive value. As the precision of the STL file increases, the error component from STL precision converges to zero. This causes the total error to be dominated by the other negative error sources. This means the STL approximation error is no longer the prominent error of the model. As a final check of STL error hypothesis, the two leading models were tested over the full range of attenuations at a higher voxel count of 125,000 to decrease the error components caused by poor point source approximations. Figure 20 shows that at higher values of mesh density, the error of the more precise STL file converges towards 0% and the STL file with a positive percent error still exhibits the STL approximation error.

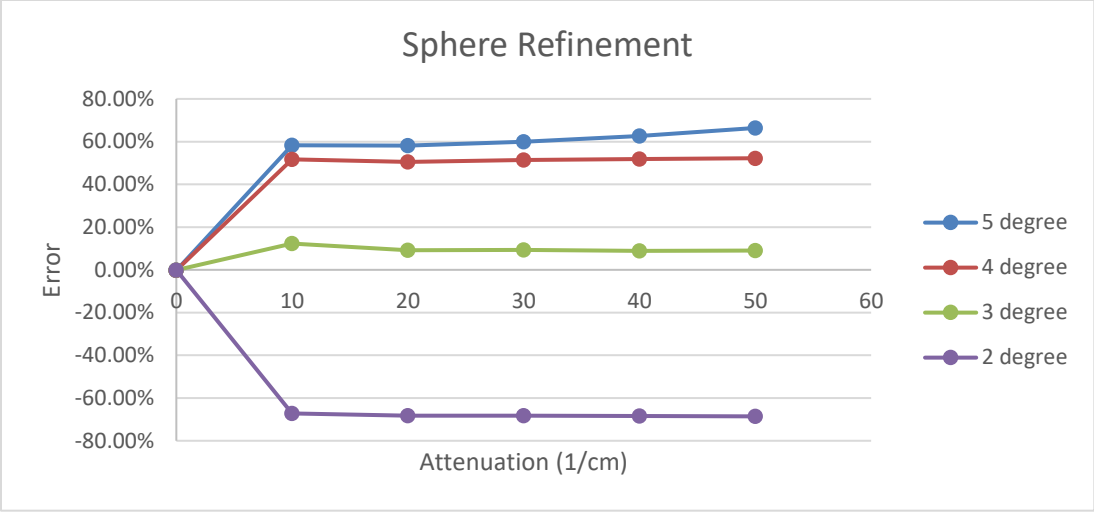

Figure 19: Spherical refinement study

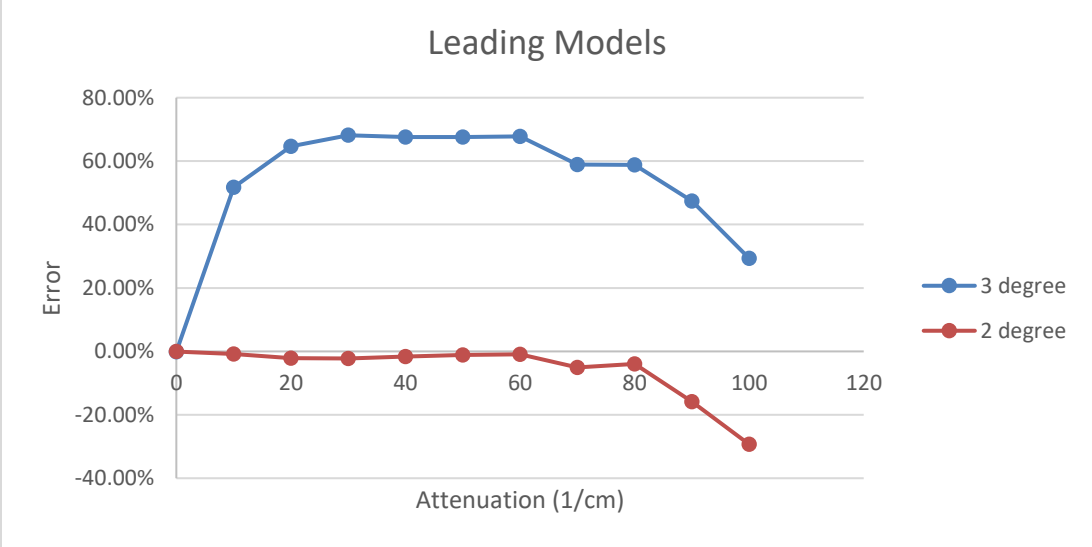

Figure 20: Leading spherical model confirmation

As a final sanity check, all 6 models were run again using more precise STL files and the same increasing attenuation and mesh densities of 50,000, 100,000, and 150,000. Spheres converge towards a value just under 20% as shown in Figure 21. This means that STL approximation error is still causing problems, however, the lower voxel count could

be used to create a precise model balancing the error between the STL approximation and the point source approximation. Figure 22 shows the error for a Cylinder under the same conditions. A cylinder shows significantly less error than the sphere model. This is likely due to the simplicity of a cylindrical STL file. Fewer approximations are made due to the flat surfaces, thus resulting in a more accurate model that is less susceptible to STL approximation error.

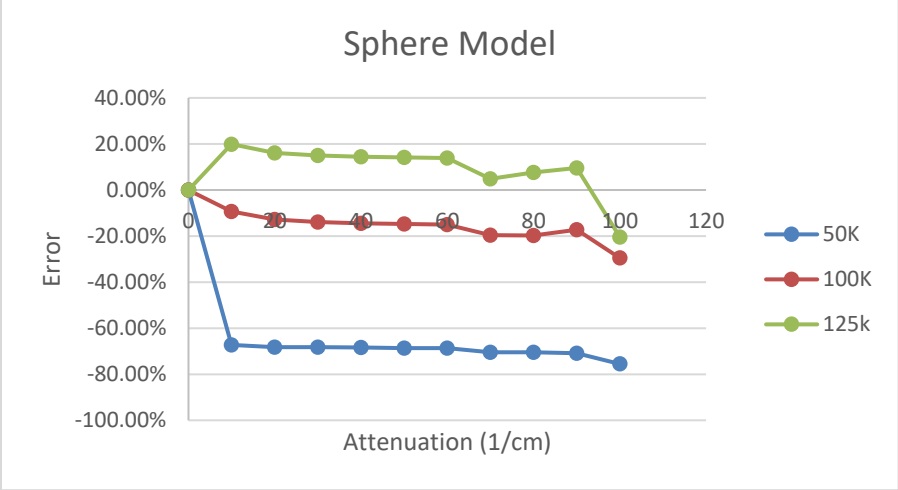

Figure 21: Refined spherical model with contained mesh

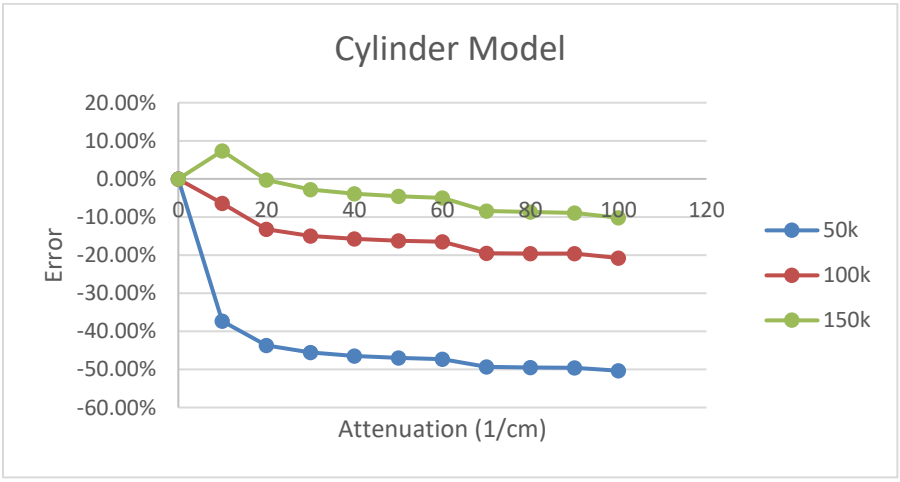

Figure 22: Refined cylindrical model with contained mesh

5.2 GADRAS Adaptive Mesher Comparison

GADRAS currently has an adaptive mesher that allows overlapping models to be adjusted. Because of this, the six fundamental voxel types can used to attempt to conform to any surface. This section compares the adaptive mesher's ability to conform to a cube for each voxel type to the tetrahedral mesh of a cube and to the GADRAS shape primitive mesh of a cube. The models are created to have edge length of 10 cm. The same testing procedures are used, with the same constants and the same variations.

5.2.1 Results and Discussion

The aim of this section is to compare the ability of the preexisting adaptive mesher to the tetrahedral mesher which could perform this sort of refinement. The adaptive mesher runs faster than the tetrahedral mesher and does not use the maximum voxel count. Figure 23 shows the spherical voxel model's attempt to conform to a cube. The error is both positive and negative as the refinement does not trend towards one side like the tetrahedral mesher. The magnitude of the error is greater than that of the tetrahedral mesher; however, the tetrahedral mesher has an exact initial mesh and only responds to the point source refinement criteria. The large initial error in the volume is more than likely the main source of error, much like when a poor initial mesh is used with the tetrahedra. The other four shape primitives' comparison plots can be found in Appendix C and show varying levels of success in matching the GADRAS cube mesh.

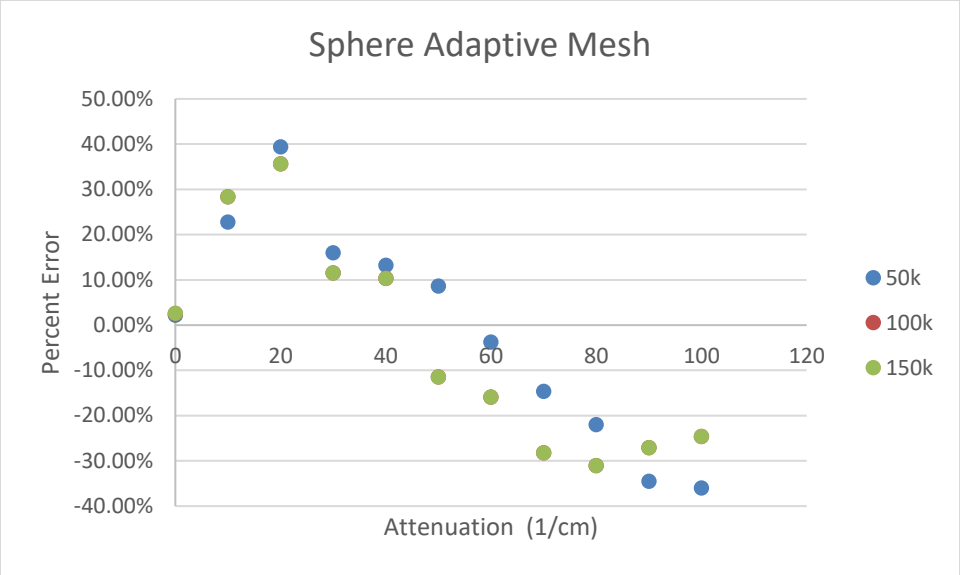

Figure 23: Spherical adaptive mesher comparison

This experiment shows that the tetrahedral mesher has a better ability to approximate point sources than the adaptive mesher under situations where the voxel shape is different than the volume it is representing This better approximation comes at the cost of speed and increased voxel count. Additionally, the flat faces of a cube are significantly easier to match than a curved surface. If a complex surface can be modeled using the shape primitives, the anticipated result will be much less accurate than the tetrahedral approximation given a good initial mesh.

5.3 GADRAS Scene Comparison

Most situations where GADRAS is useful involve a scene where some radioactive material exists in a complex environment. Due to the errors outlined in section 5.1.2, large numbers of voxels and very precise STL files are needed for complex surfaces as sources. Additionally, a good initial mesh is necessary. This leads to very large computation times and, for the simple mesher developed for testing, a convex model. For these reasons, only

the ray tracing aspect of the unstructured model was tested for a complex scene. This comparison consisted of a cylindrical source inside a rocket as depicted in Figure 24. The rocket was created using GADRAS shape primitives as well as a single STL file. Then, the error was calculated for varying levels of attenuation for the rocket for increasing level of STL precision. Both implementations use a GADRAS cylinder primitive as the source for a fair comparison.

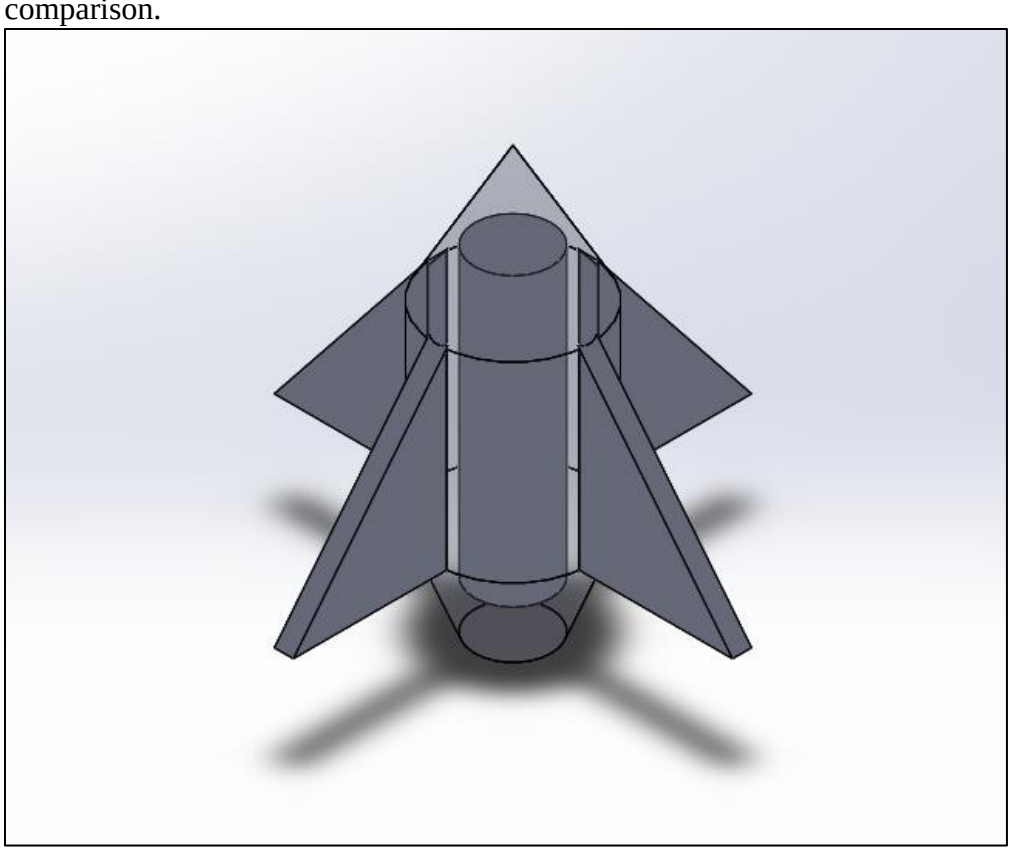

Figure 24: Mock rocket for ray tracer test

5.3.1 Results and Discussion

Although this section does not deal with the tetrahedral volume mesh, the ray tracer is a critical element in deciding if and where a voxel needs to be split. By examining the ray tracer error, the contribution to the total error can be seen and further understood for

future work. Figure 25 shows the contribution of the source cylinder to a detector on the x-axis (radial to the rocket). Since the graph has such large outliers, the only information gained, is that there are extreme outliers. These may be caused by poor handling of the ray when it intersects a triangle on the edge, or somehow gets through the STL file without being attenuated. Figure 26 shows the same data with the outliers removed. Without outliers (error over 100%), the trend holds that as the precision of the STL file increases, the error decreases. Using only the ray tracer through unstructured meshes, the error from the STL file is much less pronounced, as the data is more than likely polluted by the same error that caused the extreme outliers to a lesser extent. More instances of this experiment can be seen with different detector positions in Appendix D.

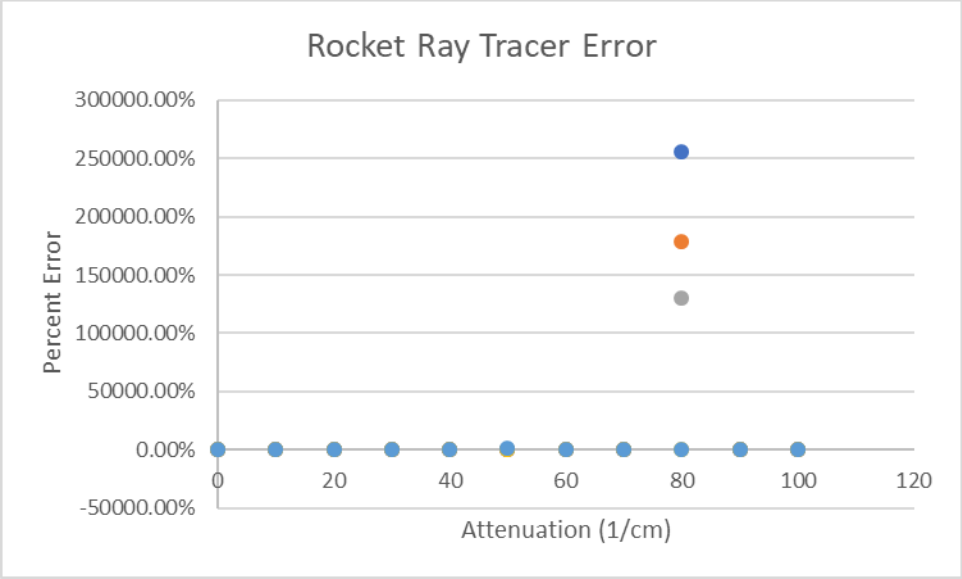

Figure 25: Extreme outlier ray tracer example

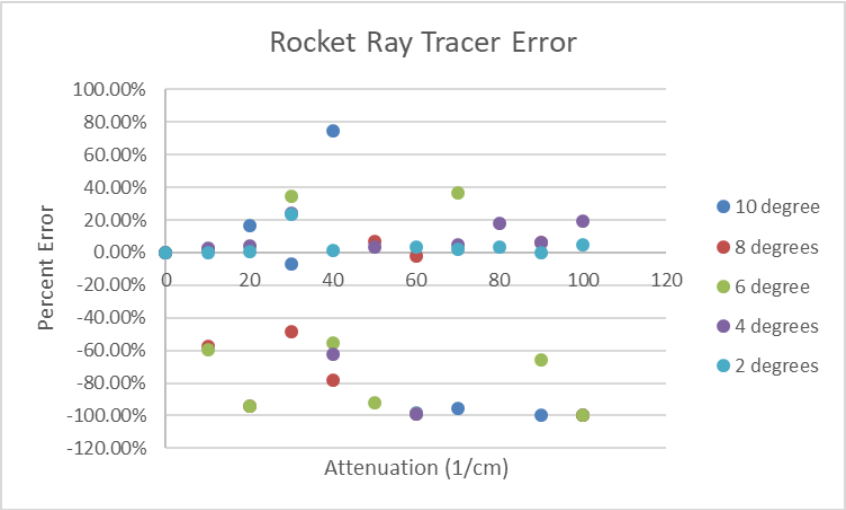

Figure 26: Polluted ray tracer data

Using a different detector position of (30,30,0), the true ray tracer error correlation with STL precision can be seen without major influence from outliers contributing to the totals as shown in Figure 27. Increasing the precision of the STL file clearly reduces the error of the ray tracer. This plot exemplifies the error that is compounded on the tetrahedral mesher caused by the STL file error.

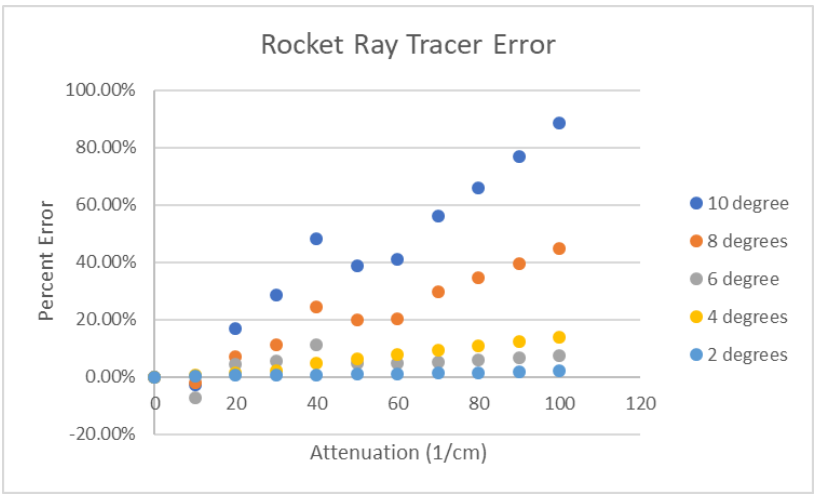

Figure 27: STL precision trend on ray tracer

5.4 Summary

Through three different simulation experiments, the error of the unstructured model is better understood. The comparison to GADRAS models showed that the initial mesh for a tetrahedral object is paramount and strongly dictates the computation of the higher attenuations. Once that was corrected, the STL file precision contributed substantially to the error. It was shown that increasing the precision of the model decreases the influence of the STL precision error on the total error. However, it was shown that there is a balance between mesh density and STL precision that needs to be carefully examined for each model. By using the existing adaptive mesher, it was determined that the tetrahedral mesher results in less error than the adaptive mesher for some primitives, and on par with the adaptive mesher for others. Finally, by ray tracing a complex scene, the floating-point errors and possible unattenuated rays demonstrated influence on the error. Once eliminated, it was shown that the effect due to STL approximation can be resolved by increasing precision.

Chapter 6: Conclusion

This research developed the capability to use unstructured models in GADRAS, a nuclear radiation transport simulation package. A prototype was developed with the restrictions that the input file type be universal and fit within the existing framework for compatibility with existing algorithms. The prototype utilizes the STL file format for the input file due to its simplicity and ubiquitous nature. To increase the computational efficiency of intersecting large numbers of triangles, an octree data structure was created to take advantage of spatial coherence of the STL objects.

Given that the STL format only defines the surface of an object, a method to generate the volumetric meshes needed for nuclear simulation was developed. Tetrahedral meshing was chosen, as tetrahedra, compared to other mesh element shapes, can most easily be generated to approximate a general object. Thus, tetrahedral voxels were used in the GADRAS framework. The framework calls for each voxel to be defined by three dimensions. By using opposite edge pairs as “dimensions”, tetrahedra integrate well with GADRAS algorithms. This was visually tested to ensure that split criteria were triggered correctly.

Once both the surface mesh, and volumetric mesh were fully integrated, the full unstructured model implementation was compared with existing GADRAS models as a form of validation. These tests revealed three critical components to the total error associated with an unstructured model: point source refinement, STL precision, and unattenuated rays. Point source refinement error reduction requires finer meshes and more voxels to decrease error. STL precision captures all error resulting from the approximate nature of the STL file. This error increases as voxel count increases. Finally, by testing the ray tracer, unattenuated rays can cause large amounts of error in the final value.

Through testing and debugging, the unstructured modeling implementation developed in this research has been shown to be a viable method that is competitive with an existing adaptive mesher in GADRAS, with the added benefit of conforming to complex surfaces. This modeling flexibility comes at the expense of execution speed.

6.1 Future Work

As a proof of concept, this implementation shows a viable path forward. In order to achieve the best result, the three main components of error must be addressed. A proposed solution to the large number of voxels generated is a better initial mesh. Many of the early refinements of the voxels are necessary to guarantee that the initial mesh is contained within the surface model. Once done, they are generally irregular and not optimal for point source refinement. This can be solved by generating a better initial mesh that can deal with nonconvex objects and create regular tetrahedra with the minimum number of elements possible.

STL precision error can be mitigated by increasing the STL file precision in the CAD software of choice. This will cause the time to ray trace the model to increase, so refinement of the octree subdivision depth should be considered. Additionally, time can be saved by streamlining error handling in the ray tracer for nonconvex objects. This error handling also needs to be improved to decrease the likelihood of unattenuated rays, thus eliminating this problem.

By decreasing the impact of the three main causes of error, unstructured modeling can be fully implemented in GADRAS. This would decrease the time needed to create models in GADRAS and allow complex geometries to be modeled that could not previously be modeled.

Appendices

Appendix A

The following charts show the error of tetrahedral implementation compared to GADRAS shape primitives for varying levels of attenuation and maximum voxel count. The initial mesh is formed using a space filling tetrahedral pattern and then refined to be contained in the surface model.

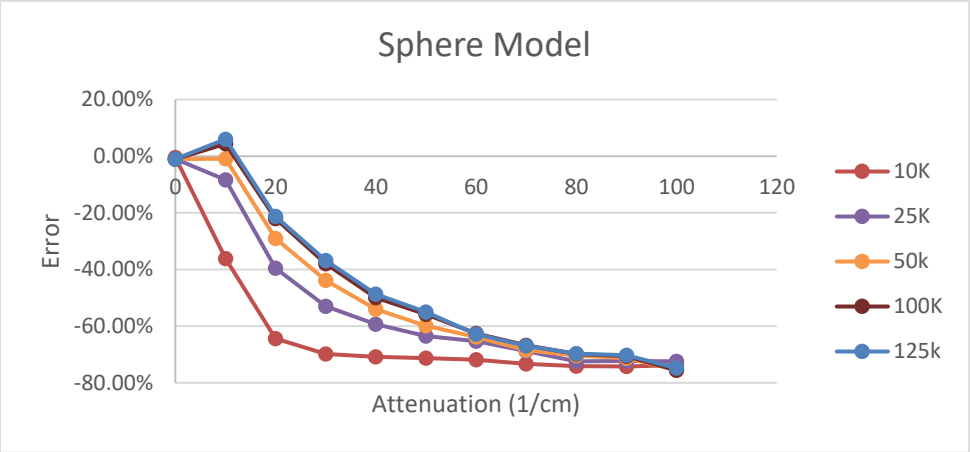

Figure 28: Sphere model bad initial mesh

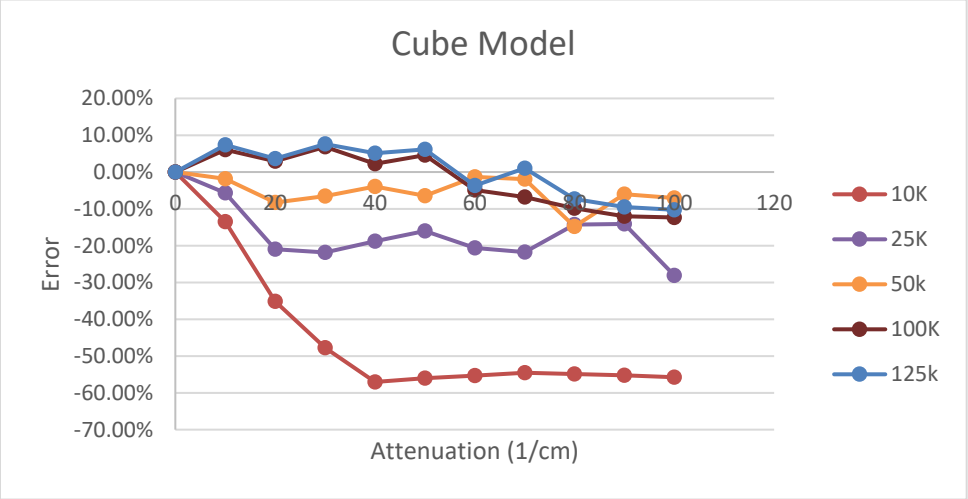

Figure 29: Cube model bad initial mesh

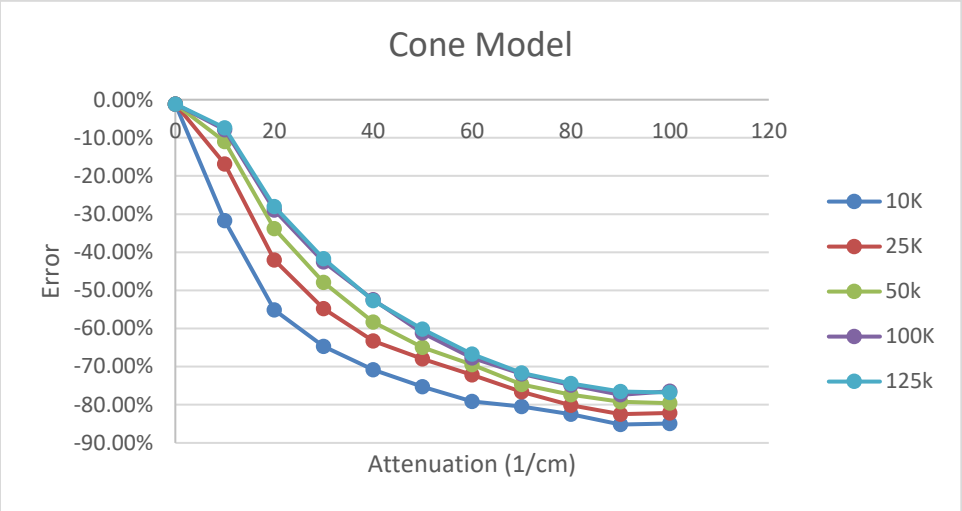

Figure 30: Cone model bad initial mesh

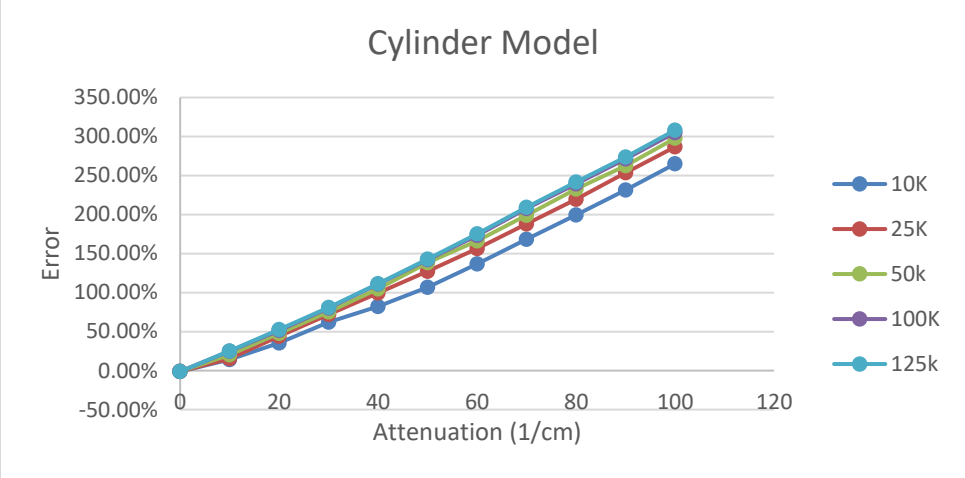

Figure 31: Cylinder model bad initial mesh

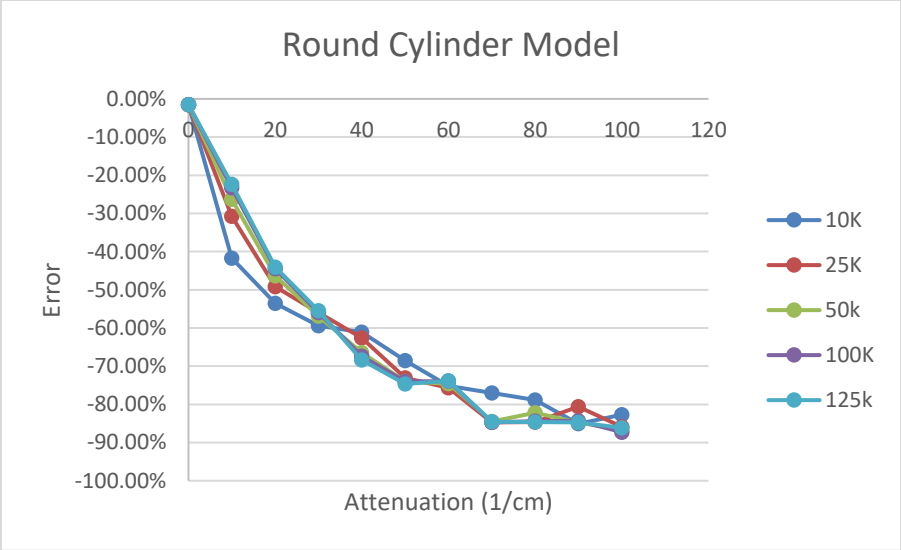

Figure 32: Round Cylinder model bad initial mesh

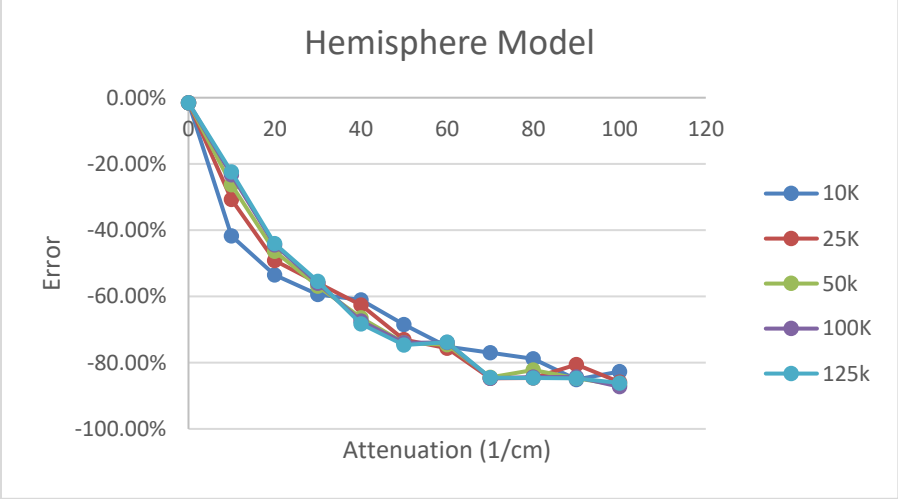

Figure 33: Hemisphere model bad initial mesh

APPENDIX B

The following charts show the error of tetrahedral implementation compared to GADRAS shape primitives for varying levels of attenuation and maximum voxel count. The initial mesh is formed by using the triangles of the surface mesh and the center of the model. This method is only viable for convex geometries.

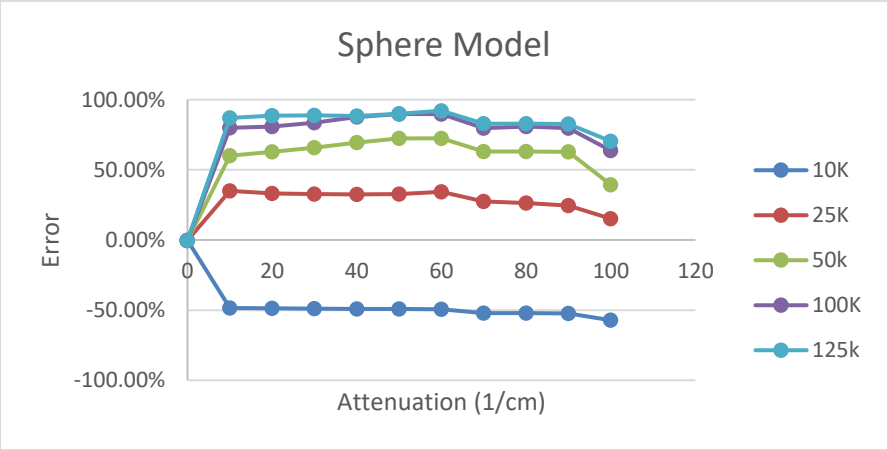

Figure 34: Sphere model contained initial mesh

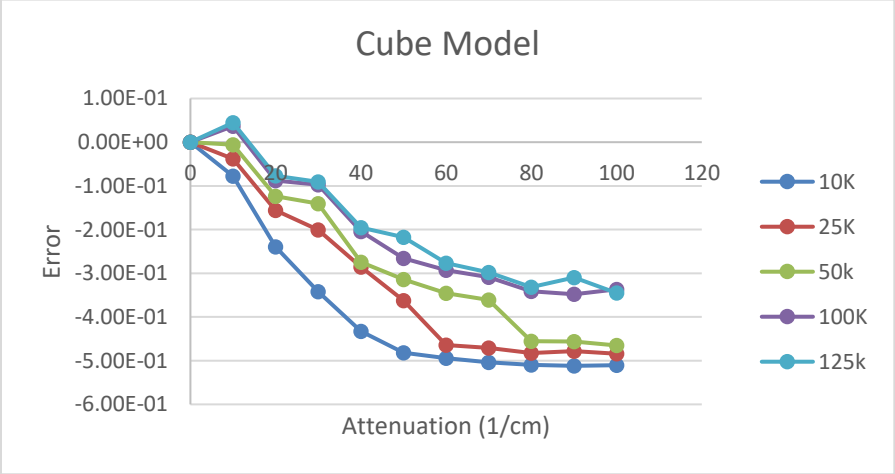

Figure 35: Cube model contained initial mesh

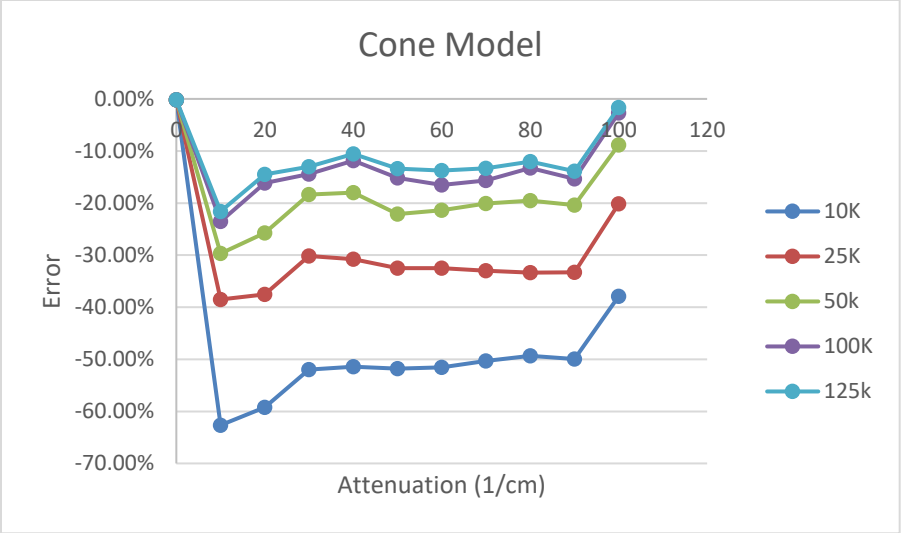

Figure 36: Cone model contained initial mesh

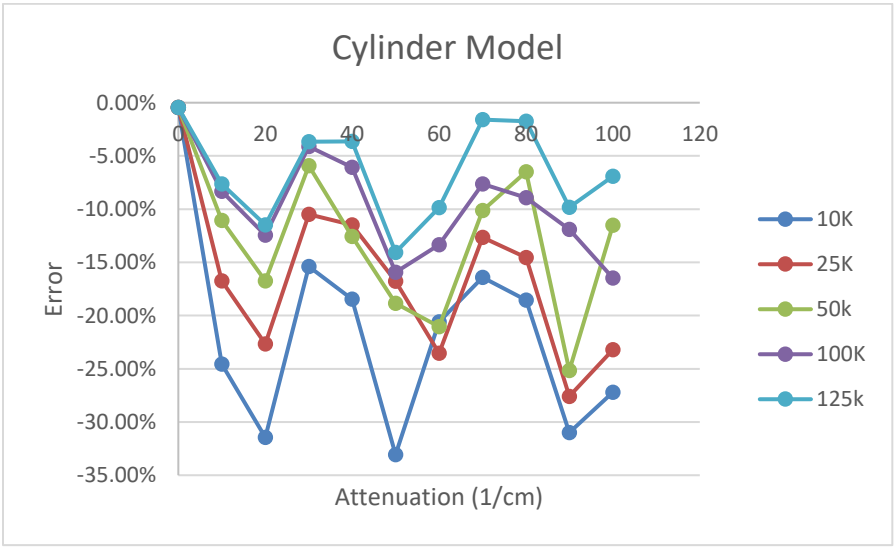

Figure 37: Cylinder model contained initial mesh

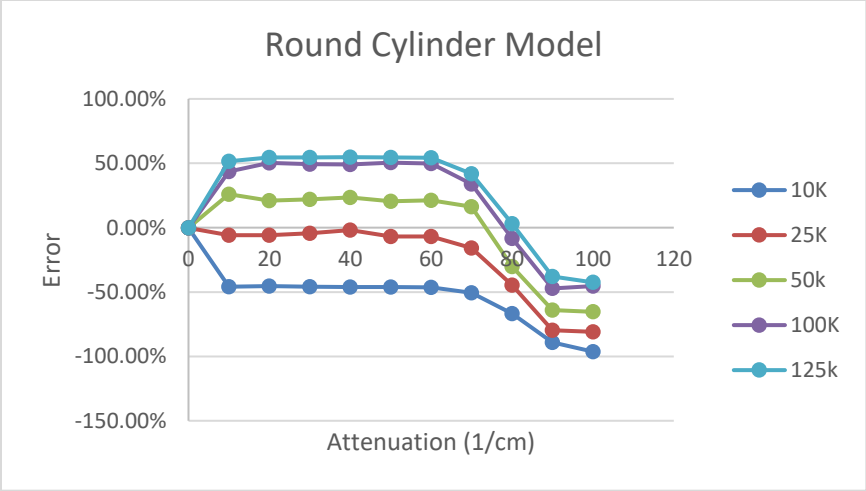

Figure 38: Round Cylinder model contained initial mesh

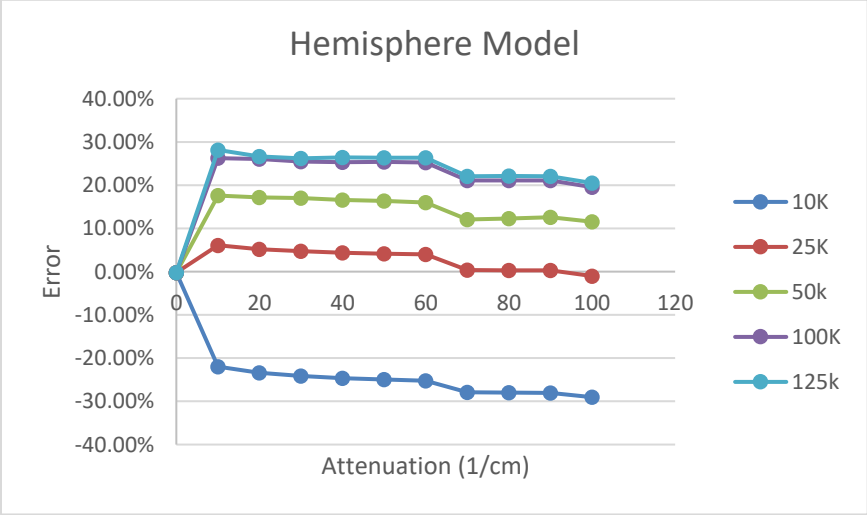

Figure 39: Hemisphere model contained initial mesh

APPENDIX C

The following charts show the error of tetrahedral implementation compared to GADRAS shape primitives for varying levels of attenuation and maximum voxel count. The initial mesh is formed by using the triangles of the surface mesh and the center of the model. This method is only viable for convex geometries. The models used are very precise STL files, where the allowed angle is 2 degrees.

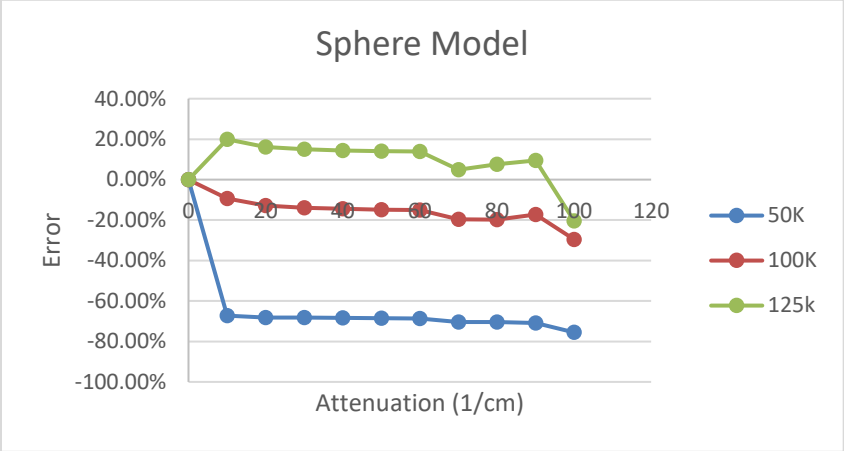

Figure 40: Sphere model contained initial mesh refined

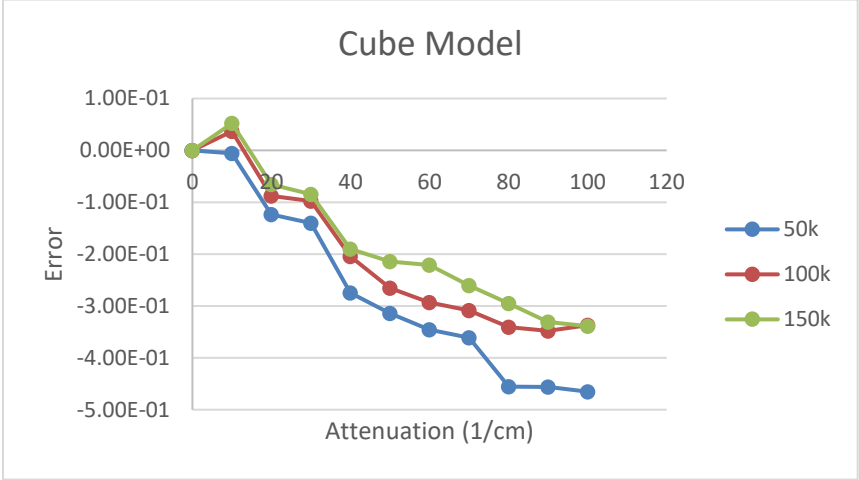

Figure 41: Cube model contained initial mesh refined

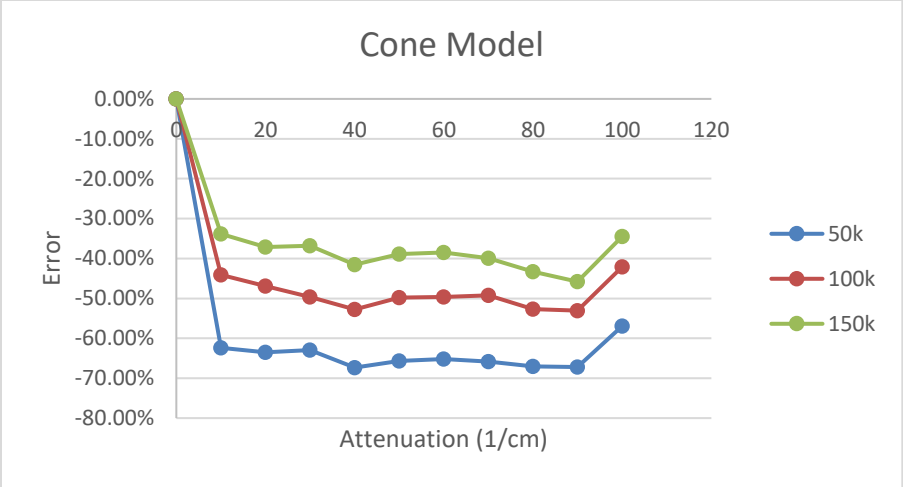

Figure 42: Cone model contained initial mesh refined

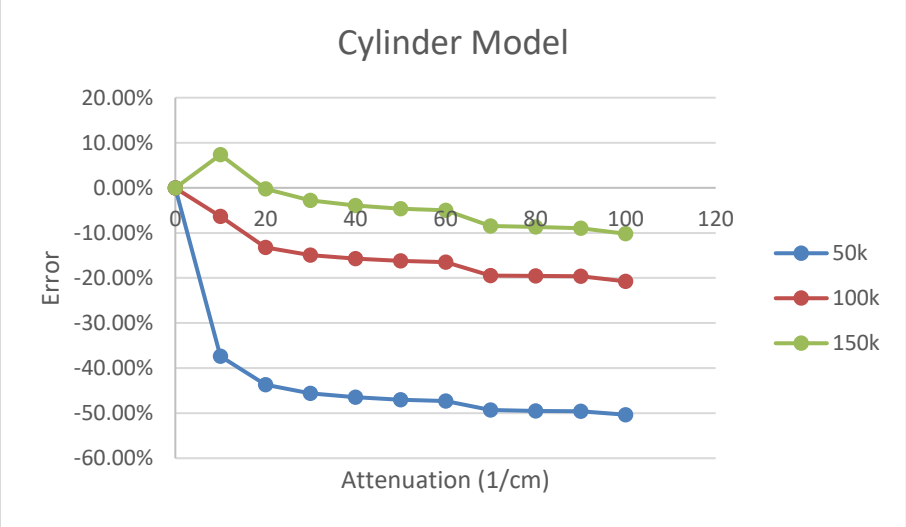

Figure 43: Cylinder model contained initial mesh refined

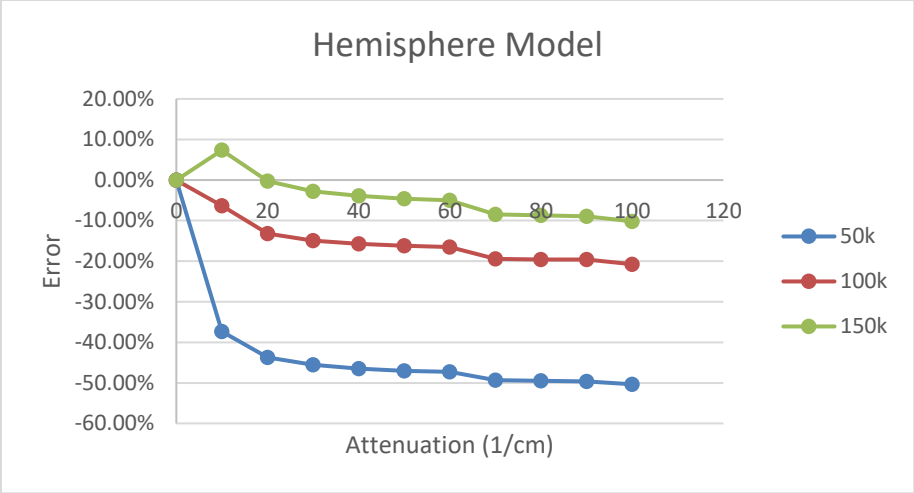

Figure 44: Hemisphere model contained initial mesh refined

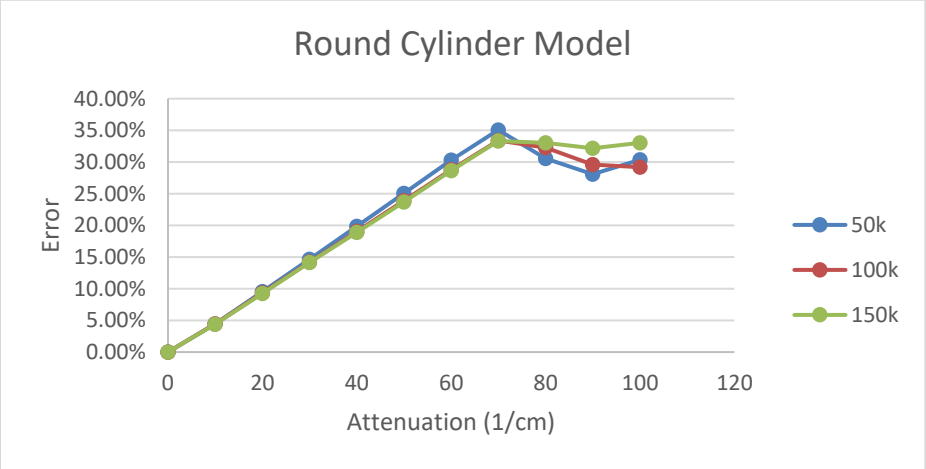

Figure 45: Round Cylinder model contained initial mesh refined

APPENDIX D

The following charts show the error of tetrahedral implementation compared to GADRAS implementation of a mock rocket for use with the ray tracer. These graphs show various detector positions and have the outliers removed.

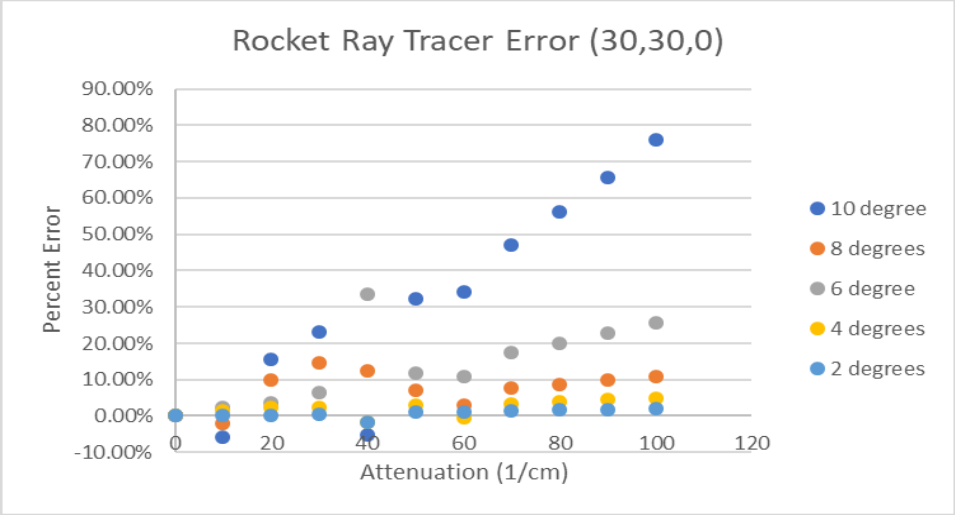

Figure 46: Ray tracer validation det (30, 30, 0)

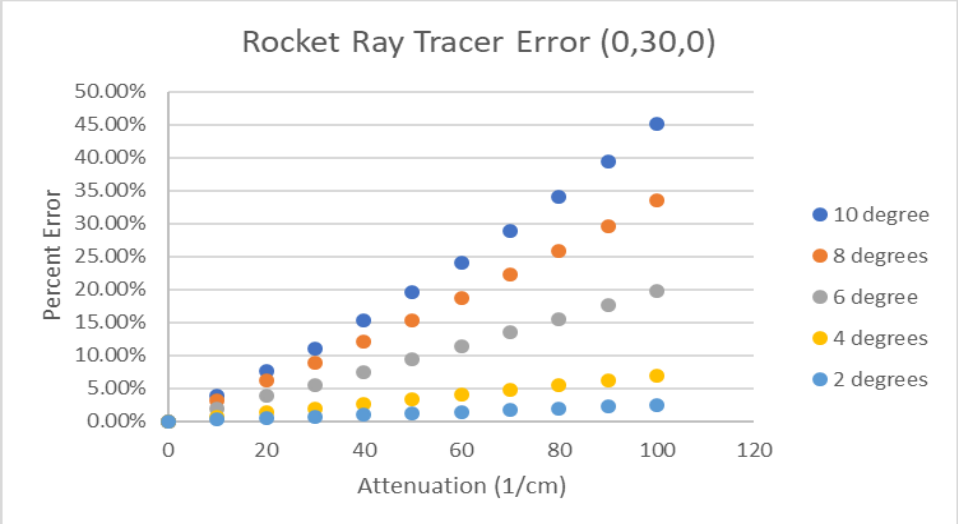

Figure 47: Ray tracer validation det (0, 30, 0)

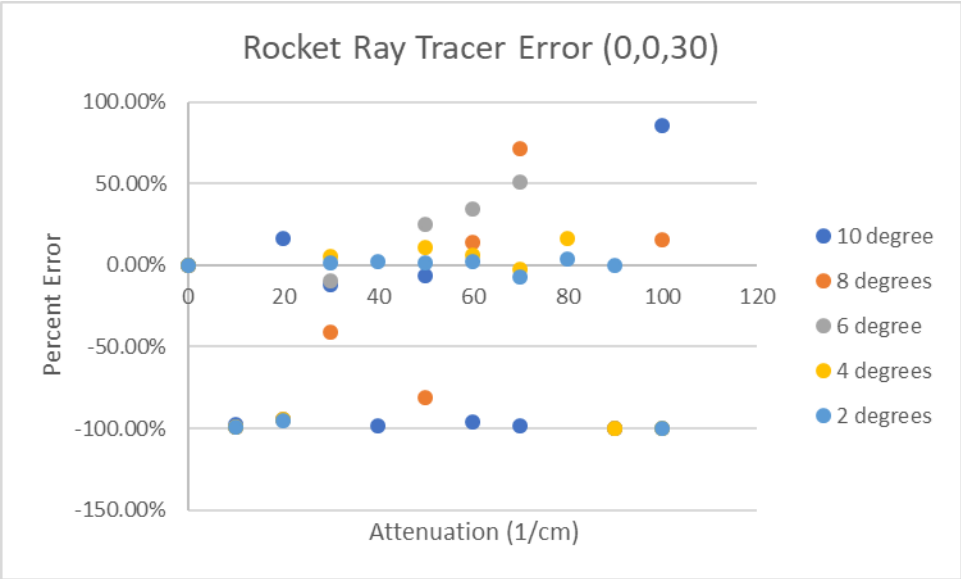

Figure 48: Ray tracer validation det (30, 30, 0)

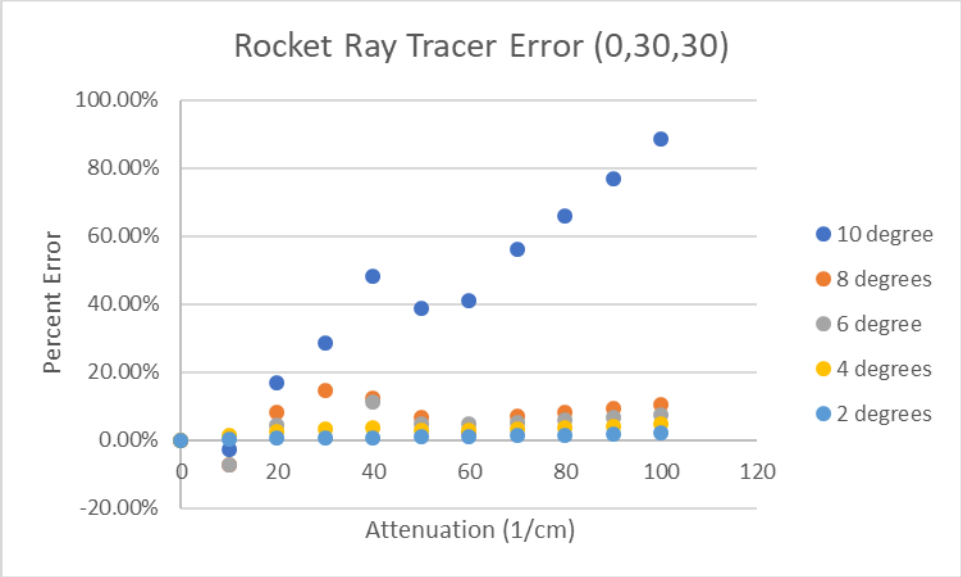

Figure 49: Ray tracer validation det (0, 30, 30)

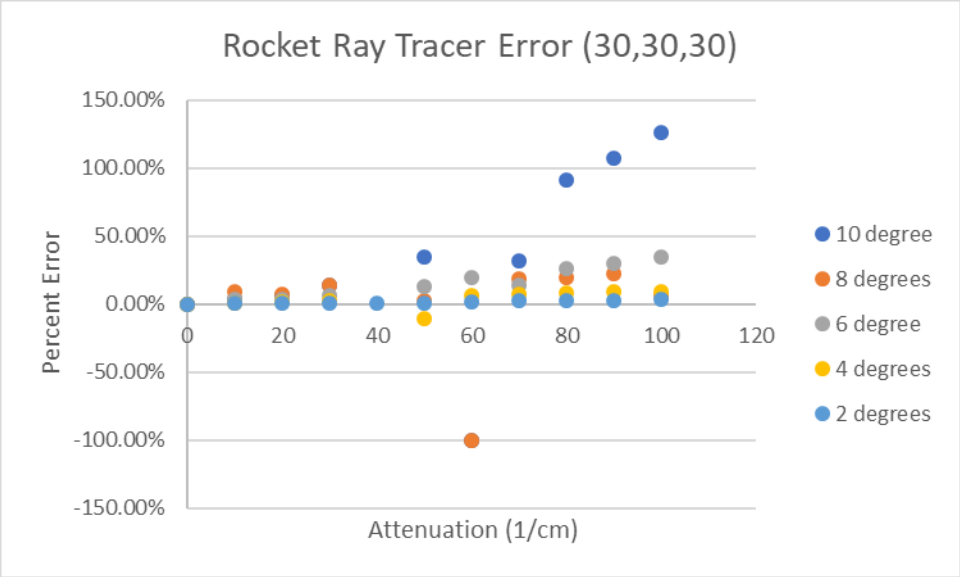

Figure 50: Ray tracer validation det (30, 30, 30)

Works Cited

ANSI (1982). Digital Representation for Communication of Product Definition Data. *American national standard engineering drawing and related documentation practices*

Badouel, D. (1990). An efficient ray-polygon intersection. In *Graphics gems* (pp. 390-393). Academic Press Professional, Inc.

Bentley, J. L. (1990). K-d trees for semidynamic point sets. In *Proceedings of the sixth annual symposium on Computational geometry* (pp. 187-197). ACM.

Below, A., De Loera, J. A., & Richter-Gebert, J. (2000, February). Finding minimal triangulations of convex 3-polytopes is NP-hard. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms* (pp. 65-66). Society for Industrial and Applied Mathematics.

Brown, S. (1998). Evaluation of subdivision methods used in octree ray tracing algorithms. Thesis. Rochester Institute of Technology.

Brunner, T. A. (2002). Forms of approximate radiation transport. *Sandia report*.

Chan, T. M. (1996). Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, *16*(4), 361-368.

Dybedal, Joacim & Aalerud, Atle & Hovland, Geir. (2019). Embedded Processing and Compression of 3D Sensor Data for Large Scale Industrial Environments. Sensors. 19. 636. 10.3390/s19030636.

Glassner, A. S. (1984). Space subdivision for fast ray tracing. *IEEE Computer Graphics and applications*, *4*(10), 15-24.

Haines, E. (1994). Point-in-polygon strategies. *Graphics gems IV*, *994*, 24-26.

Hiller, J. D., & Lipson, H. (2009, August). STL 2.0: a proposal for a universal multi-material additive manufacturing file format. In *Proceedings of the Solid Freeform Fabrication Symposium* (Vol. 3, pp. 266-278).

Kay, T. L., & Kajiya, J. T. (1986, August). Ray tracing complex scenes. In *ACM SIGGRAPH computer graphics* (Vol. 20, No. 4, pp. 269-278). ACM.

LANL. (n.d.). A General Monte Carlo N-Particle (MCNP) Transport Code. Retrieved from https://mcnp.lanl.gov/.

Löhner, R. (1988). An adaptive finite element solver for transient problems with moving bodies. In *Computational Structural Mechanics & Fluid Dynamics* (pp. 303-317). Pergamon.

Möller, T., & Trumbore, B. (2005, July). Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses* (p. 7). ACM.

Rebay, S. (1993). Efficient unstructured mesh generation by means of Delaunay triangulation and Bowyer-Watson algorithm. *Journal of computational physics*, *106*(1), 125-138.

Reshetov, A., Soupikov, A., & Hurley, J. (2005). Multi-level ray tracing algorithm. *ACM Transactions on Graphics (TOG)*, *24*(3), 1176-1185.

Shevtsov, M., Soupikov, A., Kapustin, A., & Novorod, N. (2007, September). Ray-triangle intersection algorithm for modern CPU architectures. In *Proceedings of GraphiCon* (Vol. 2007, pp. 33-39).

Stroud, I., & Xirouchakis, P. C. (2000). STL and extensions. *Advances in Engineering Software*, *31*(2), 83-95.

Sundar, H., Sampath, R. S., & Biros, G. (2008). Bottom-up construction and 2: 1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, *30*(5), 2675-2708.

Sustainability of Digital Formats: Planning for Library of Congress Collections. (2017). Retrieved from https://www.loc.gov/preservation/digital/formats/fdd/fdd000448.shtml.

Wessner (2006). Mesh Refinement Techniques for TCAD Tools. Retrieved from http://www.iue.tuwien.ac.at/phd/wessner/diss.html

3D Systems, Inc.(1988, July), StereoLithography In Interface Specification