THESIS

GPU ACCELERATED CONE BASED SHOOTING BOUNCING RAY TRACING

Submitted by

Blake A. Troksa

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2019

Master's Committee:

Advisor: Branislav Notaros
Co-Advisor: Sudeep Pasricha

Hamidreza Chitsaz

ABSTRACT

GPU ACCELERATED CONE BASED SHOOTING BOUNCING RAY TRACING

Ray tracing can be used as an alternative method to solve complex Computational Electromagnetics (CEM) problems that would require significant time using traditional full-wave CEM solvers. Ray tracing is considered a high frequency asymptotic solver, sacrificing accuracy for speed via approximation. Two prominent categories for ray tracing exist today: image theory techniques and ray launching techniques. Image theory involves the calculation of image points for each continuous plane within a structure. Ray launching ray tracing is comprised of spawning rays in numerous directions and tracking the intersections these rays have with the environment. While image theory ray tracing typically provides more accurate solutions compared to ray launching techniques, due to more exact computations, image theory is much slower than ray launching techniques due to exponential time complexity of the algorithm.

This paper discusses a ray launching technique called shooting bouncing rays (SBR) ray tracing that applies NVIDIA graphics processing units (GPU) to achieve significant performance benefits for solving CEM problems. The GPUs are used as a tool to parallelize the core ray tracing algorithm and also to provide access to the NVIDIA OptiX ray tracing application programming interface (API) that efficiently traces rays within complex structures. The algorithm presented enables quick and efficient simulations to optimize the placement of communication nodes within complex structures. The processes and techniques used in the development of the solver and demonstrations of the validation and the application of the solver on various structures and its comparison to commercially available ray tracing software are presented.

ACKNOWLEDGEMENTS

I am thankful for the love and support of my family who have always been there for me throughout my academic career. I want to thank my advisor, Dr. Branislav Notaros, who convinced me to further my education and who guided and supported me during my time working in his lab. I would also like to thank the members of my committee for their contribution to my academic development. Finally, I would like to thank all fellow students who helped me become who I am today including: Cam, Stephen, Jake, Pranav, Forest, and Sanja.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 Shooting Bouncing Rays

In the field of computational physics, the necessity for high performance computing (HPC) techniques to solve large scale and computationally demanding problems is becoming ever more apparent. In Computational Electromagnetics (CEM) specifically, where numerical approximations are common, these techniques are essential. One such acceleration technique involves the parallelization of the underlying algorithms. Today, the development and abundance of graphics processing units (GPU) has created the technology to achieve massive parallelization. In contrast to traditional CEM full-wave solvers such as method of moments or the finite element method (FEM), the shooting bouncing rays (SBR) ray tracing (RT) solver can be applied to quickly analyze electrically large structures such as mine tunnels[1]. This is because full-wave solvers require the construction and solving of large systems of equations which requires significant amounts of memory and time. The full-wave technique Method of Moments (MoM) for example, has a time complexity of $O(N^3)$ and RT has a time complexity of $O(N\log^2 N)$ due to the balanced construction of the binary space partition tree where N is the number of facets in the structure. In addition, the SBR RT application requires a simple mesh composed of triangular facets to describe the scene in question whereas full-wave solvers can often demand complex higher-order meshes. The RT solver requires much less memory and computation time in order to analyze large structures at a cost of accuracy. In addition, RT solvers can benefit immensely from acceleration with GPUs.

Ray tracing uses the convenience of Maxwell's equations offered as a linear set of partial differential equations within linear, homogenous, and anisotropic propagation domains. This

allows ray tracing to solve high frequency electromagnetics problems. The SBR approach to RT involves launching a set of test rays in all directions that originate from some centralized transmitting point. These rays are then traced through the scene, and their intersections with objects in the scene are recorded as displayed in Figure 1. The SBR method is described in further detail in [2]. The electric field at a desired location in the scene is then found by summing the contribution of ideal spherical wavefronts of the electric field for each ray that intersects the reception point at the desired location. Using reflection coefficients based on surface parameters for each reflection, the final electric field at the desired observation point can be approximated from the combination of every path segment for a ray from the source to the observation point [3]. This process may be repeated for several observation points to produce a field profile at desired locations.

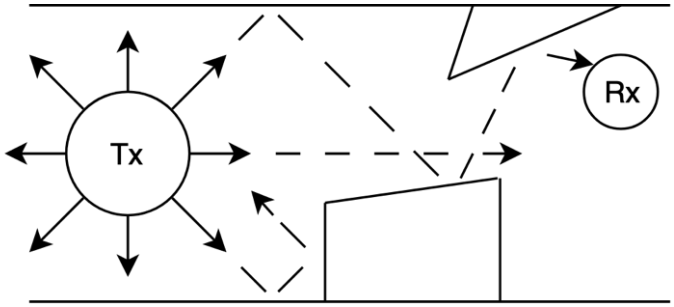


Figure 1. Shooting Bouncing Rays: Depiction of the SBR ray tracing technique with transmitting and receiving antenna and boundary obstacles.

The SBR RT application presented here is split into two execution subsections. The first subsection involves the geometric and physical path calculation of rays as they propagate through a given structure. The second subsection involves the post-processing, or field calculation, of each ray. The geometric path calculation begins by creating a mesh of the structure/environment that is under examination. Once the mesh has been constructed a collection of rays are generated that will traverse the structure. After these rays have been generated, the NVIDIA OptiX ray tracing application programming interface (API) is used to calculate the nearest intersection points

between the rays and the facets that they hit. After these intersection points are computed, the information is copied to the GPU where post processing occurs. The post processing consists of the calculation of intersections between ray cones and observation points, the removal of any double counting between adjacent ray cones, and the updating or the electric field propagated by each ray.

1.2 Goal of Thesis

The Electromagnetics Laboratory in the Electrical and Computer Engineering department at Colorado State University (CSU) is interested in expanding the types of solvers in their CEM repertoire to handle ever larger propagation problems. The main codes used at the electromagnetics laboratory include MoM and the finite element method (FEM). These computational codes provide very accurate numerical solutions to small scale electromagnetic problems. While these applications work very well for solving certain electromagnetic problems, there is not a program that solves large scale electromagnetics problems such as the path loss from an antenna within a 1km tunnel due to poor scaling to larger input sizes i.e. the $O(N^3)$ time complexity for MoM. The creation of a ray tracing solver would enable the ability to solve CEM problems within these electrically large structures. In addition, the development of a high frequency asymptotic solver provides the ability to create hybrid solvers that combine the accuracy of full-wave solvers like FEM and MoM with the speed of ray-based solvers.

The focus of this thesis is on the development of an SBR ray tracing algorithm that can solve these large-scale electromagnetics problems. While a general ray tracing algorithm will enable this computation, the ray tracing algorithm presented here focuses on accelerating and optimizing the code with high performance computing techniques. The primary source of

acceleration for this code will be achieved through general purpose graphics processing unit (GPGPU) programming. GPUs are useful for solving problems that perform sequential operations tens of thousands of times because they can perform this execution in blocks concurrently. In order to maximize the efficiency of GPUs, the structure of the algorithm as well as the management of memory within the algorithm need to be optimized. The primary focus of this thesis is to develop a shooting bouncing rays ray tracing algorithm with a high degree of parallelism, optimized code execution structure, and modularity for expanding and applying the algorithm to different problems.

1.3 Overview

Chapter 2 of this thesis discusses the approach to the geometric calculations of the ray tracing algorithm. In particular, this includes the different techniques available for the generation of the rays, consideration in the mesh necessary to preform ray tracing, as well as the approach to computing the intersection points each ray makes with the mesh.

In Chapter 3, an overview of the procedures for the post processing section of the algorithm are described in detail including the variations in computation that are common in ray tracing. Details on the approach to computing, storing, and updating the electric field for each ray are described in order to develop an understanding of the flow of data and processing on the GPU. The advantages to using ray cones as opposed to ray tubes from a computation standpoint are presented to contrast the tradeoffs made to improve the SBR RT algorithm.

Chapter 4 introduces the important optimization strategies used for writing efficient executable code on NVIDIA GPUs. This includes the importance of structured global memory accesses that limit the total number of reads and writes to memory necessary and the inevitability

of synchronization. Configuration optimization of the GPU kernels for the ray tracing program is compared with changes in performance from different configurations. Specific implementations of the ray tracing algorithm to incorporate these strategies is discussed.

Chapter 5 shows validation of the ray tracing algorithm by comparing the simulated results with those available in literature. Tunnels with uniform cross section are the main subject for validation of the ray tracing algorithm. Furthermore, simulation results compared to physically measured data is contrasted to demonstrate the usefulness of this approach as a replacement to manually generating signal strength maps. The application of the algorithm to mapping 2-dimensinal regions is presented as an additional feature.

Finally, Chapter 6 provides conclusions about the success of the algorithm, the comparison with commercially available software, and the ability to expand upon the base algorithm developed. This algorithm leaves the possibility of adding features such as edge diffraction, transmission calculations, and adaptive ray spawning to the algorithm's main functionality. Additionally, the ray tracing algorithm can be combined with other solvers to create a multifunctional and adaptive solver.

CHAPTER 2: GEOMETRIC CALCULATIONS

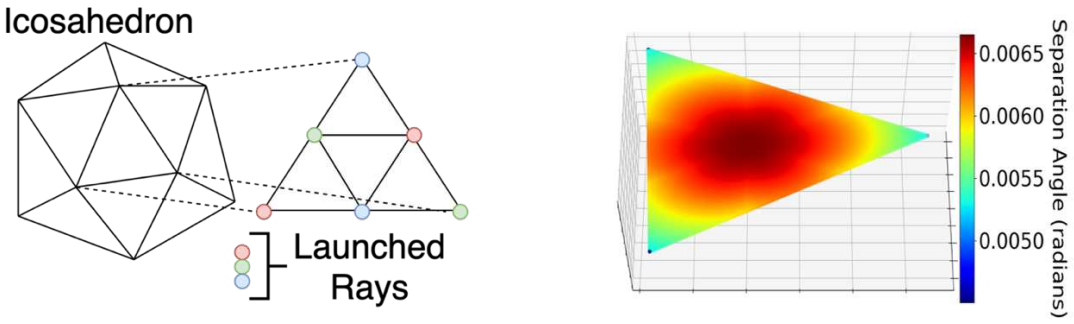
2.1 Ray Generation

The generation of rays is an important consideration to ensure a minimum number of rays needed for convergence to guarantee complete coverage of the structure under consideration and to easily determine the neighboring rays of each ray. Since ray tracing is a high frequency asymptotic solver, an infinite number of rays at infinite frequency is necessary to reach an exact solution. The SBR RT algorithm assumes that the reflection boundaries are infinite planes, but at frequencies in the GHz range this condition is well satisfied. Launching an infinite number of rays is infeasible but, in general, the more rays that can be launched the better the results will be. The number of rays necessary is determined by analyzing the accuracy of the solution as the number of rays grow and by looking at how much the solution changes by adding more rays [4]. As an example, a 5-minute run of the ray tracing application of one hundred million rays with 1 percent error could be more beneficial than a run of the application with one billion rays that leads to a solution with .5 percent error but takes an hour.

The spawning techniques for the ray tracing application involve sampling points on surfaces that surround some ray origin point. Once these sampled points are determined, each ray for the ray tracing algorithm is launched from the origin to one of the sampled points. Different spawning techniques such as uniform grid sampling and Fibonacci spiral sampling have been tested and provide different advantages in regard to sampling spacing and ease of formation [5]. The best balance for the SBR RT application it to discretize the twenty faces of an icosahedron into a grid like structure. The total number of points on one of the faces is therefore represented by the triangle number calculated using $\frac{n*(n+1)}{2}$. This spawning enables near-uniform spacing for

complete coverage but also, due to the nature of the discretization, it provides an efficient way to determine which rays are neighbors to other rays. This is important for tracking which rays can overlap with other rays since the overlap is exclusive to direct neighbors. Another advantage to the icosahedron structure of ray spawning is the convenience for batching the ray tracing algorithm. The more rays that are launched for a given SBR RT simulation, the more the simulation needs to be divided into batches to ensure memory limits are not exceeded and since the RT algorithm is only considering Maxwell's equations in linear media, this decomposition of the solution is possible. The icosahedron provides a simple solution to split up a simulation into twenty batches due to the twenty different faces. Additionally, each face of the icosahedron can also be divided into smaller regions however, it was discovered that twenty batches from the icosahedron were enough for over 250 million rays which was ample density for the structures tested.

Another important feature of the icosahedron ray spawning is the ability to easily compute the separation angle, alpha, between a ray and its neighbors in order to preform double count removal. In some cases, alpha has been chosen as a constant value but the distribution of alpha across a face of the icosahedron varies significantly as depicted in Figure 2.



(a) (b)

Figure 2. Icosahedron Ray Spawning: (a) Icosahedron used for ray spacing in the initial radiation pattern of the antenna. More subdivisions can be added for more rays to be spawned. (b) Separation angle distribution for one of the faces of the icosahedron.

The reason the icosahedron is beneficial in this scenario can be explained by considering one of the other spawning techniques. In the case of the Fibonacci spiral spawning technique, while the

sampling is distributed ideally over a sphere, the ability to determine which points are closest to other points is not trivial. All the points would need to be checked with all other points to determine the closest neighbor or the points would need to be sorted in some way to help with the complexity of finding the closest points for all points in the spiral. The icosahedron however, solves this problem since the neighbor rays are already recorded and tracked for use in double count removal. This adjacency map that tracks a ray's neighbors makes determining the angle between a ray and its neighbors as simple as calculating the max separation angle between a ray and its, at most, 6 other neighbors. The difference in accuracy from choosing a static alpha is shown in Figure 3(a) compared to the ray-based alpha shown in Figure 3(b). In order to ensure complete coverage of a given structure, the separation angle is chosen as the max separation angle between a ray and its neighbors Figure 3(b). If the angle is chosen as the minimum distance between a ray and its neighbors, then the entire scene is not covered and there can be observation points that do not receive any rays which is evident in the noise towards the end of the signal in Figure 3(c).

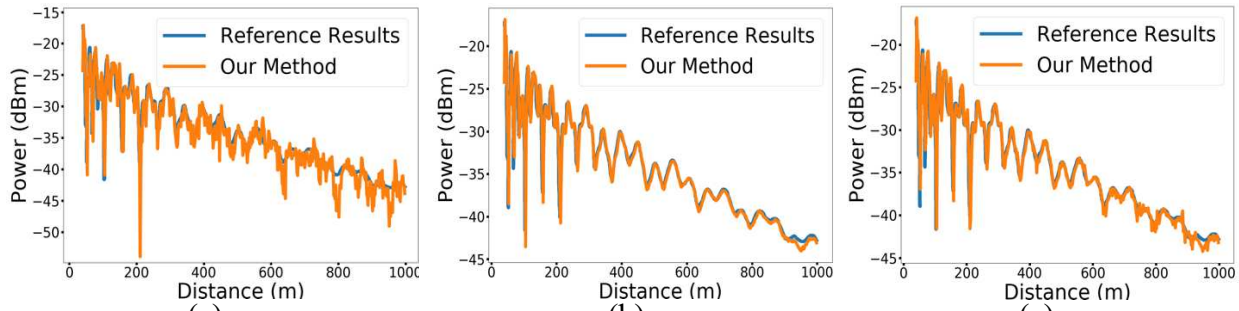


Figure 3. Effect of Separation Angle: Different alpha approximations on the path loss calculations of a dielectric waveguide for alpha as (a) constant (b) max angle between neighbors (c) min angle between neighbors. These results were obtained on a lossy waveguide of dimensions 4m x 4m of ε_r =5 with the receivers and a transmitter located 2.1m from the floor and 1.1m from the wall [8]. The frequency is 1GHz.

Lastly, since the faces of the icosahedron are all the same and the spacing between all the rays is the same, the adjacency map for the entire ray spawning procedure only needs to be computed for

the first face. This is beneficial since it reduces the execution time for the ray generation portion of the code. Once the rays are created the rays can then be traced through given structures that are converted into meshes.

2.2 Mesh Creation

The input to the OptiX Prime API is a wavefront (OBJ) file that contains the information regarding the location and orientation of triangles constituting a certain structure. Figure 4 displays an example of the meshing for portions of the Colorado School of Mines Edgar research mine in Idaho Springs, Colorado.

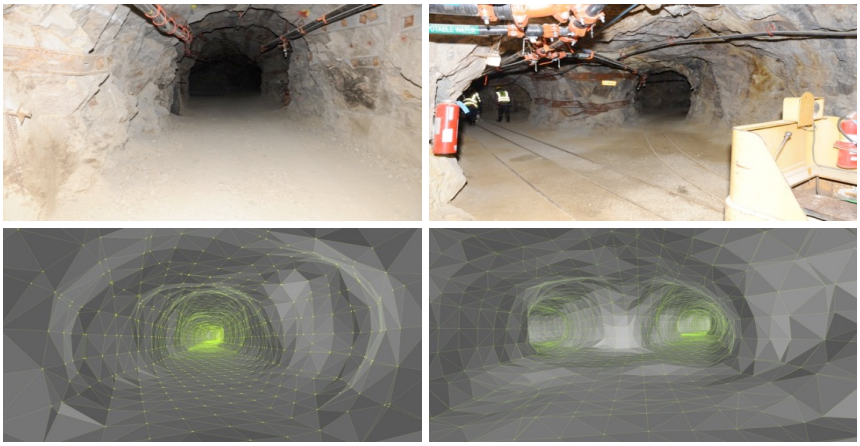


Figure 4. Creation of Meshes: Illustration of the meshing algorithm applied to LiDAR data collected in the Colorado School of Mines Edgar research mine.

The quality and approximations in the mesh given to the ray tracing algorithm is a factor in the accuracy of the solution. In environments that have surfaces with high curvature, more triangles are needed to accurately represent these boundaries. In SBR RT, the facets need to be electrically large, around 10 wavelengths in order to ensure the plane wave approximation is as accurate as possible. Typically, the creation of meshes involves the collecting of point cloud data, then the creation of a triangle mesh from this point cloud data. Considerations in the density of the point cloud data and subsequently the size of the triangles is very important especially in rough structures

such as tunnels or mines because, at high frequencies, the surface roughness can be electrically large and if the point cloud data is not dense then certain features in the structure would be ignored. Additionally, a mesh can be created without point cloud data for example in smooth environments such as building hallways where the walls are uniform and have an easily measurable shape.

2.3 Ray Facet Intersection Computations

Once the rays have been generated, the adjacency map has been created, and the separation angle (alpha) has been computed for each ray the rays are used to populate a buffer of ray type objects. This buffer is one of the main programming components available in the NVIDIA OptiX Prime ray tracing API. This buffer is passed off to the OptiX ray intersection program that runs on the GPU and determines the path of a ray from an origin point to a destination point along the direction specified in the spawning. The OptiX API is used since NVIDIA has spent many years developing OptiX to be used as a general-purpose ray tracing engine [6]. An essential feature and a primary reason for using OptiX is the efficient construction of the binary space partition (BSP) tree. The binary space partition tree is a tree type data structure that stores subdivisions of an environment and allows for efficient ray triangle intersection tests by pruning out sections of the space that are not physically reachable from other sections of the space. An efficient binary space partition tree reduces the total number of facets that a ray has to check in order to find which one it hits first [7]. Since this software is available free under non-commercial use, the OptiX API was chosen to eliminate the need to develop a new binary space partitioning scheme that would potentially be inferior to the OptiX BSP-tree. In order to construct the BSP-tree and perform these ray-facet intersection tests, the OptiX ray tracing API needs a geometry description file to build a mesh. The OptiX API then loads the mesh and executes a closest hit query for every ray in the ray

buffer and calculates the nearest intersection with a triangle for that ray. After these intersections are computed, the rays are reflected along the facet that they have intersected using the law of reflection. This translation involves converting the barycentric coordinate system of the ray's intersection on the triangle to the Cartesian coordinate system.

Certain anomalous scenarios can occur while a ray is in the geometric portion of the SBR RT algorithm. The first scenario is if the ray escapes the mesh that it is propagating in. This happens rarely and is a product of numerical precision error that allows rays that hit directly on the boundary of two facets to escape the mesh structure. This is mitigated in the ray tracing application by removing these rays from the buffer meaning the information the ray contained is no longer considered in the simulation. The frequency of this scenario is very low and therefore this removal technique allows for a simple solution to escaped rays. The second scenario a ray can encounter is if it leaves the region of interest, or the section where the observation points are located, without leaving the actual mesh. In order to limit this region of interest, a bounding box consisting of multiple kill planes is created to terminate any ray that intersects the box. After the rays are reflected, the information regarding the previous path segment the ray has travelled is used in post processing to determine any intersections the ray may have had with an observation point.

CHAPTER 3: POST PROCESSING

3.1 Electric Field Calculations

Post processing begins by copying all data pertinent to each ray necessary to perform ray sphere intersection tests, compute the electric field at the observation points, and update the electric field associated with each ray. Each ray object in the SBR RT application tracks information relating to its interaction with the environment it is propagating in. Every ray computes the distance it has travelled up to the current reflection point in order to calculate the phase change imparted by the medium the ray propagates in given by

$$\frac{e^{-j\beta r}}{r} \tag{1}$$

$$r = ray\ distance\ traveled;\ \beta = wavenumber\ (function\ of\ frequency)$$

In most scenarios the medium of propagation is assumed to be air. Once the ray has reflected, the triangles or facets that have been intersected are recorded in order to determine the attenuation of the electric field based on the material composition of the facet. This attenuation is described using the Fresnel coefficients for normal polarization (2) and parallel polarization (3) which are dependent on the properties of the material that the environment is comprised of.

$$r_{\perp} = \frac{n_i \cos(\theta_i) - n_t \cos(\theta_t)}{n_i \cos(\theta_i) + n_t \cos(\theta_t)} \tag{2}$$

$$r_{\parallel} = \frac{n_i \cos(\theta_t) - n_t \cos(\theta_i)}{n_i \cos(\theta_t) + n_t \cos(\theta_i)} \tag{3}$$

As stated above, ray tracing is designed based on the linearity of Maxwell's equations. Given this notion, we propagate the plane waves directly along the path the ray takes. This

approximation reduces the complexity of the computations needed to find the resulting electric field. The first computational step in post processing is the calculation of the initial radiation pattern of each ray based on a source antenna. The radiation pattern of the transmitting antenna is discretized and sampled at points corresponding to the points located on the face of the icosahedron. A Hertzian dipole antenna is used for most simulations in the basic version of the ray tracing algorithm due to simplicity of this radiation pattern. The radiation pattern of the Hertzian dipole is symmetric in the azimuthal direction and depends only on the polar angle making the calculation of the initial field a simple trigonometry calculation. The incorporation of a new antenna is as simple as defining a function to map each point on the icosahedron to a point on the radiation pattern created by the antenna. If a function to map the points is not feasible then a dataset of electric field points created by sampling a radiation pattern equal to the number of rays can be created and mapped accordingly.

After the initial electric field is computed, the electric field for each subsequent ray-facet intersection can be computed. In order to handle the change in magnitude and phase of the electric field for each reflection, the normal and parallel polarization must be calculated for the incident and reflected rays. The decomposition of an intersection into these two components can be seen in Figure 5 below where (a) depicts the parallel polarization and (b) depicts the perpendicular polarization of a uniform plane time-harmonic electromagnetic wave incident on a surface. After the initial field is calculated but before the electric field from a reflection is updated for a ray, an intersection test needs to be performed in order to determine if a ray has contributed to one of the observation points along its current ray segment from the previous reflection to the current reflection.

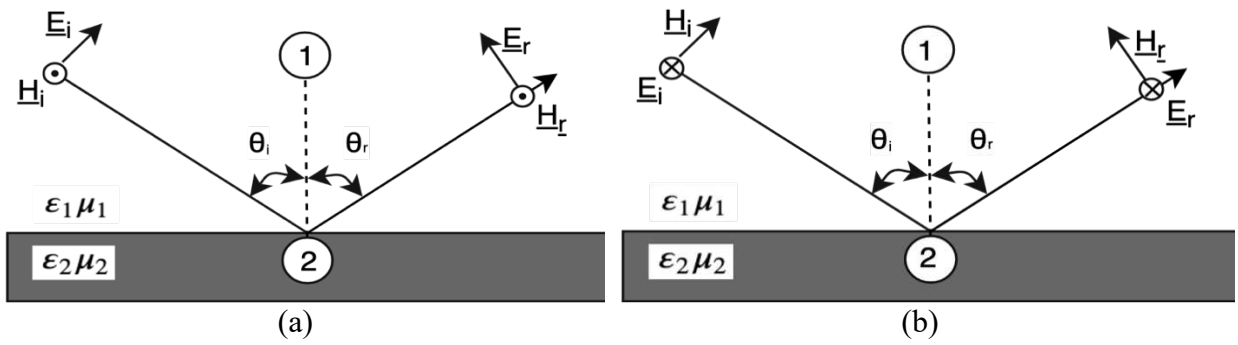


Figure 5. Reflecting Plane Wave: The reflection of a plane wave along the surface boundary shows the electric and magnetic field vectors for (a) parallel and (b) perpendicular polarizations of a uniform time-harmonic plane wave.

3.2 Sphere Intersection

Two differing techniques have emerged to handle the gap that exists between rays due to physical limits on the number of rays that can be launched. Subsequently, these two techniques dictate the type of intersection tests that need to occur between a ray and an observation point. One technique uses ray tubes or convex geometric shapes that fit in a grid like structure with neighboring tubes. The ray tube technique works by tracing a polygonal shape consisting of bounding rays representing the edges of the tube, and the central ray, representing the field of the ray tube. The second, alternative, technique uses cones to surround each ray in which neighboring cones overlap in order to cover the entire space. Figure 6 displays the difference between these two techniques. Ray tubes are advantageous because there is no overlap between adjacent ray tubes. This reduces the amount of processing needed in the post processing calculations because it eliminates the need for double count removal. Also, when using ray tubes, the test for determining if a reception or observation point is contained within a tube is a very efficient test. A drawback to using ray tubes occurs if the structure under consideration has curved boundaries because two possibilities can arise when ray tubes reflect off the curved surface. The first possibility is to reflect the entire ray tube on the plane that is tangent to the central ray of the ray

tube. When using this approach, the ray tubes will either disperse, in the case of convex structures, meaning there will be gaps between the tubes or the ray tubes will overlap, in the case of concave structures, meaning the distinction between individual ray tubes is lost. The second possibility is to reflect each vertex of the ray tube on its own tangent plane, but this means the tubes lose the rigid polygonal shape that they originated with [8]. In order to solve the issue with curved boundaries, we use a cone-based ray discretization technique. In the cone-based ray tracing technique, the reception points for the rays are modelled as spheres and the overlap between different spheres can be eliminated by tracking which rays have intersected the spheres thus eliminating any overlap or double counting that may occur.

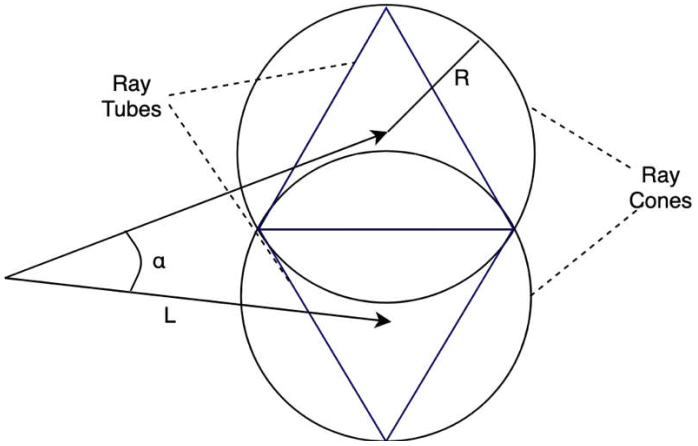


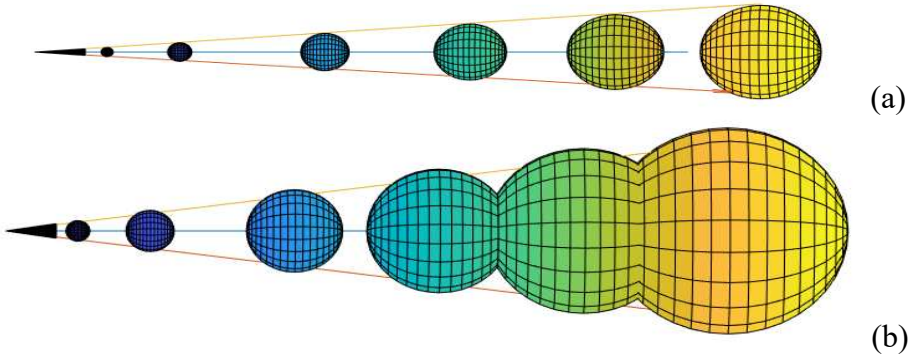
Figure 6. Ray Cones vs. Ray Tubes: Visualization of the different space fillers between rays constructed as triangles and ray cones.

The intersection tests performed determine how many rays have hit a sphere or spheres representing the observation points. This number is then used to allocate the corresponding space in memory to store information about these hits. This chunk of memory stores a specific trio of intersection information including the ray that hit the sphere, which sphere was hit, and at what distance along the segment did the ray hit the sphere. Once the information trio is collected, the field contribution from each ray is factored into the current total field of the sphere. The reason for computing the number of intersections before adding the field is to be able to perform double count

removal. As the rays move further from the transmitting origin point, the radius of the ray cones expand as shown in Figure 7(a). Subsequently, the size the reception spheres grow is determined by the equation below where distance is the total distance a ray has travelled and α is the separation angle between a ray and its neighbors.

$$Radius_{Sphere} = Distance * \alpha/\sqrt{3} \tag{4}$$

It is important that the spacing between observation points is large enough compared to the number of rays launched and therefore the separation angle as well as the distance the rays can travel, otherwise the reception spheres can begin to overlap with each other as shown in Figure 7(b) causing numerical error in the Electric Field.



(a)

(b)

Figure 7. Reception Sphere Growth: (a)The expansion of the reception spheres is a function of the ray propagation distance. (b) The reception spheres have potential to grow too large and overlap with adjacent reception spheres inducing error.

Since the ray cone is growing continuously as it approaches the observation point, the distance to the reception sphere is determined by projecting the ray's current direction vector, between the current and previous reflection, onto the vector between the observation point and the previous reflection. An illustration of this scene is displayed in Figure 8. Once the radius of the sphere has been computed, the test to determine if a ray will intersect the sphere is performed by projecting the sphere onto the plane normal to the ray's direction thereby creating a disc of the same radius as the sphere. Next, the algorithm tests to see if any point along the ray's path intersects with a

point on the disc. These points are recorded in the information trio and then refined using double count removal to remove any trio that correspond to overlapping rays.

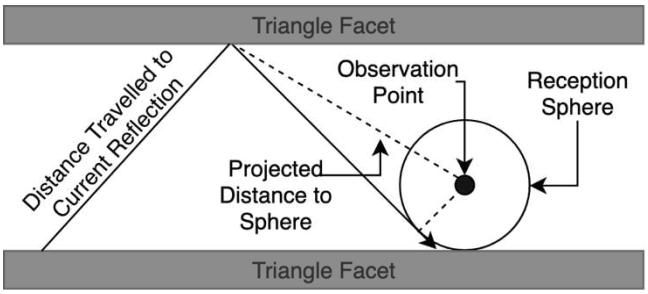
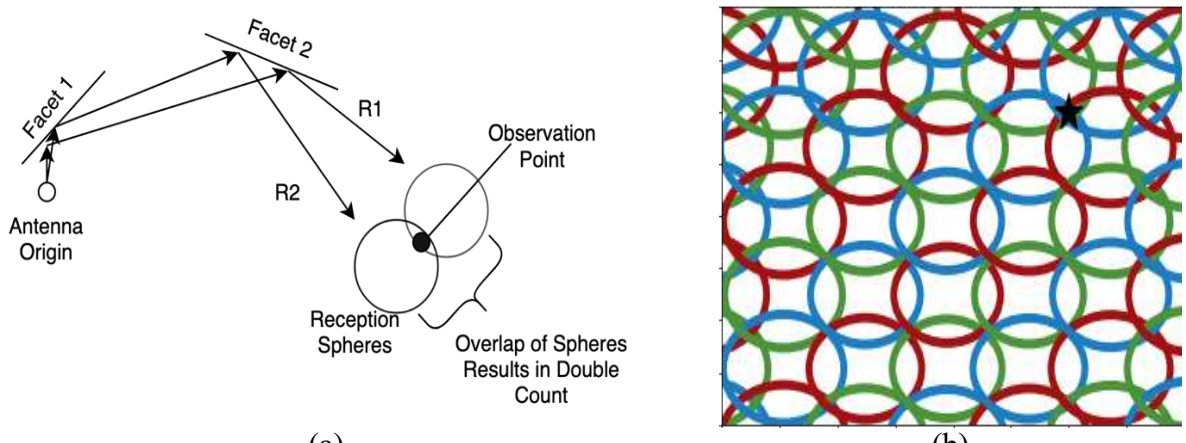


Figure 8. Ray Sphere Intersection: Graphical representation for the scene of a ray approaching a sphere and the corresponding components used in the calculation of the reception sphere radius.

3.3 Double Count Removal

Since the entire environment of interest needs to be covered with cones, it is necessary to ensure no gaps in the computation domain by having the cones overlap. Due to this overlap, not all intersections between a cone and an observation point are valid intersections. In order to know which are valid and which are not the intersections need to be tracked and added in a certain order and this process is called double count removal [9-10]. Figure 9(a-b) illustrates what this overlap can look like by taking a cross section of the wavefront of cones at a distance from the ray origin point.



(a) (b)

Figure 9. Cone Double Count Wavefront: (a) Illustration of the cone representing the overlap between cones and their neighboring cones. (b) The black star represents an observation point and shows how multiple adjacent cones intersect the same observation point due to the overlap.

The observation point lies in a region in the wavefront within multiple cones meaning all those cones will contribute to the electric field of that observation point and lead to inaccurate electric field calculations. In order to remove the inaccuracies with double count removal, the adjacency map created during ray generation is used to determine if a cone and its neighbors have intersected the same sphere. The procedure for this begins by looping through the stored sphere intersection trio data created during the ray sphere intersection tests and finding which adjacent rays have been received by the same sphere by looking up their ray IDs in the adjacency map. If the rays are adjacent rays, then only one of the rays is counted and the other is ignored for that specific sphere intersection test. Removing double counted cones from the observation points provides for a more accurate solution as displayed in figure 10.

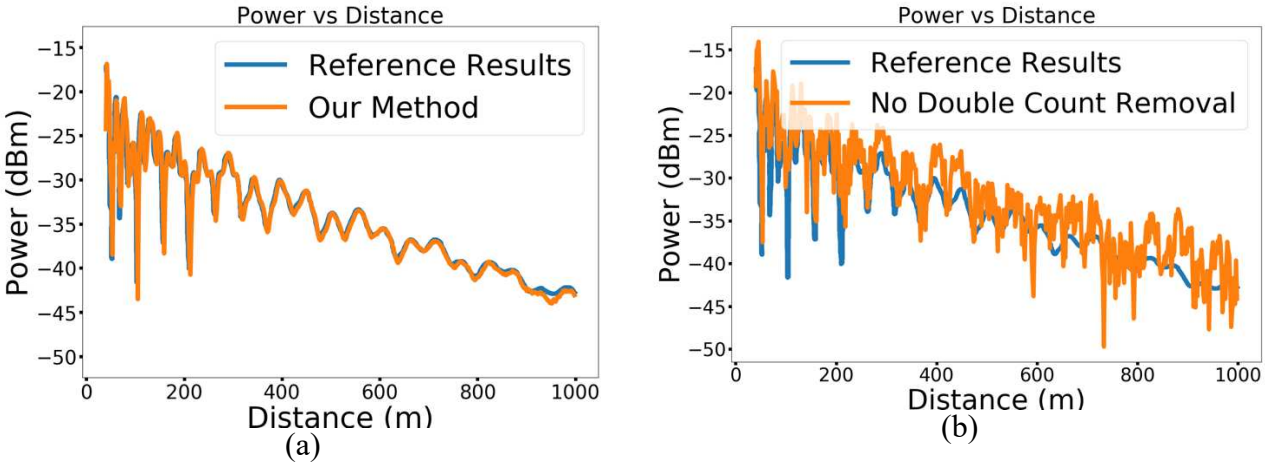


Figure 10. Effects of Double Count Removal: (a) Simulation of the dielectric waveguide with the removal of double counted ray cones. (b) The same dielectric waveguide is simulated without the removal of overlapping rays. These results were obtained on a lossy waveguide of dimensions 4m x 4m of ε_r =5 with the receivers and a transmitter located 2.1m from the floor and 1.1m from the wall [8]. The frequency is 1GHz.

After the double counted cones are removed from the observation point calculations, the electric field from each ray is added to the total electric field for each observation point. The observation points start with 0+0j initial electric field and the electric field is altered via the x, y, and z complex field components carried by the ray. Once the electric field at the observation

points have been modified, the electric field for each ray is updated and the ray tracing algorithm returns to the geometric subsection in order to calculate the next reflection point for each ray. The post processing subsection of the RT application is the most time intensive part and therefore requires the highest level of GPU acceleration to reduce the computation time.

CHAPTER 4: GPU CALCULATIONS

4.1 Overview of GPU Parallel Programming

The shooting bouncing ray tracing program involves the spawning, propagation, electric field computation, and electric field summation of millions of rays. The ability to launch millions of rays is highly dependent on the structure and efficiency of the code as well as the level of parallelization achieved. SBR ray tracing is a highly parallel procedure because each ray propagates independently and any computation involving a ray can be performed without interference from other rays. In order to achieve very quick and efficient computation time GPUs are used. The reason GPUs are effective with this type of highly parallel programming is because each basic work unit, or thread, on the GPU can compute, track, and update an individual ray's information. In contrast to Computer Processing Units (CPU), GPUs are optimized for data parallel throughput computations and CPUs are optimized for low latency access to cached data sets. CPUs are designed to control logic for out of order and abstract execution whereas GPUs perform best on organized instruction sets with limited logical branching. GPUs are also architecturally more tolerant of memory latency and have physically more transistors that can be dedicated to arithmetic computation [11]. GPUs are used in the RT application to enable large scale and efficient parallelization.

The compute unified device architecture parallel computing platform, or CUDA for short, is the primary language used to program on NVIDIA GPUs. The CUDA programming platform permits the creation of parallel computational instructions to get maximum efficiency and minimize computation time. The device code for NVIDIA GPUs is written in CUDA C/C++ and the executable code on the GPU runs kernels which are synchronous computation batches

containing sequential instruction sets. The sequential code located inside the kernels is executed by the threads that are spawned at the start of the kernel execution.

Understanding the structure of NVIDIA GPUs is important for efficient parallelization of the SBR RT application. GPUs are inherently parallel devices that contain many arithmetic compute units and many different memory storage units to handle copious amounts of numerical operations. Execution procedure on GPUs is broken down into the following sub-elements. The finest execution unit on the GPU is the thread. GPU Threads are similar to threads on a CPU and execute instructions sequentially. Multiple threads however, can execute in parallel on the different hardware housed on the GPU. The threads on the GPU are grouped into thread blocks and all thread blocks used in the execution of the kernel compose the grid. When a kernel is executed on the GPU, the thread blocks are distributed to the available streaming multiprocessors (SM) located on the GPU. Thread blocks do not begin execution until a SM has enough resources for all of the threads inside the thread block to execute. These multiprocessors have their own set of shared memory for all the threads inside that block. The threads located in the block running on the SM cannot all run at one time and the streaming multiprocessor typically, in the case of the NVIDIA GTX 1060 GPU used for the SBR RT application, can execute a group of 32 threads, known as a warp, at one time.

Race Conditions are extremely prevalent in parallel programs and especially in the RT application. Namely, the main race condition occurs when the electric field at an observation point is modified by adding the electric field from all the rays that have intersected the same observation point. Since many threads on the GPU are all executing concurrently, many of the threads would be modifying the memory containing the electric field for the observation point simultaneously. This means that the electric field for each observation point may not be updated properly by the

threads doing the post processing calculations. In order to eliminate this race condition, the threads need to have some synchronization to ensure mutual exclusion when reading or writing to the memory where the electric field for the observation points is stored. The downfall with this synchronization is a loss in computation time due to threads having to wait for other threads to update the electric field.

Effective GPU programing ensures work is given to all the pipelines available on the GPU and increases computation throughput by hiding latency of the memory pipelines. In order to maximize efficiency on the GPU, ideally all the SMs should be running code continuously with other computations running asynchronously and memory constantly being transferred to the GPU. The amount of memory available on the GPU and on the host computer are the limiting factors of the SBR RT application. The memory requirements of the GPU only allow a set number of rays to be computed at a time. Because of this constraint, for some larger structures, the rays need to be split into batches before being sent to the GPU as mentioned in Chapter 2. The first type of batching mentioned involves the subdivision of the faces of the icosahedron that are used to launch the rays initially. The other type of batching used in the SBR RT application is the batching of the reflection calculations for each of the rays. If the ray's reflection calculations were not batched, then storing all of the path information for a ray over hundreds of reflections would overflow the memory. This is because each ray reflection consists of the ray's previous intersection point represented as three floating point numbers, current intersection point also represented as three floating point numbers, and the ID of the triangle that was hit represented by an integer type. All of this stored information equates to 28 bytes of memory for each reflection. For 100 million rays, this would be 2.8 gigabytes of storage per reflection. As the number of reflections increases, the available memory would decrease so only information about a ray's current reflection segment is tracked to limit this

memory strain. After all paths in a batch of rays have been computed the next batch is processed and this procedure is repeated until no remaining rays need to be traced for intersections with the observation spheres or the limit on the number of reflections has been reached.

4.2 GPU Optimization of RT

Maximum efficiency with any kind of parallelization especially on GPUs, requires the optimization of the program to get the best performance possible. The first type of optimization prevalent with all NVIDIA CUDA GPU programs is the launch configurations of the kernel. The launch configurations refer to the number of blocks and number of threads per block that are created to run the code within the kernel. The downside of having too many spawned threads is that some threads will be idle while the other threads are doing all the work. Conversely, if there are too few threads then there could be resources on the GPU such as CUDA cores and SMs that are not being utilized for computation. In order to get best efficiency, all compute unites on the GPU should remain occupied with computation. The ray tracing application is memory bound and therefore requires higher occupancy in order to achieve better efficiency. Various kernel launch configurations were tested beginning with a thread block size of 128 threads and adjusted to the number of threads per block in increments of 32. The results are depicted below in the Figure 11.

Effect of thread work load on execution time	
Threads per Block	Time/simulation (s)
32	315.748
64	314.295
128	317.535
256	316.218
512	322.737

Figure 11. Launch Configuration: The change in execution time given different numbers of rays per thread. The similar execution times indicate each thread is preforming many tasks and that the data locality for each thread may not be optimal.

The number of blocks is also an important consideration and typically 1,000 or more thread blocks are sufficient because this number of thread blocks enables the code to be evenly distributed among the GPU hardware, but also highly modular since additional GPUs can execute the thread blocks that would be waiting for an SM.

The next optimization to consider when GPU programming is the global memory throughput. Global memory is the memory allocated on the GPU device from the host. Transferring information from host memory to device memory is very slow and the number of copies needs to be minimized. In the ray tracing application, the data for each ray is transferred to the GPU only once before the post-processing after each reflection. The largest memory storage on the GPU is global memory which is up to 6 GB in the case of NVIDIA GTX 1060 GPU. Global memory is accessible by all threads running across all thread blocks in the grid, but it has very high latency [11]. All memory operations performed in the execution of the kernel are issued per warp meaning the warp accesses memory in 32 word chunks. If the 32 threads access continuous addresses in memory, then the amount of accesses for that group of threads is reduced. These types of accesses to global memory are called coalesced accesses and they ensure all threads access a continuous chuck in memory and therefore reduce the total number of bytes that are accessed. In order to achieve the coalesced memory accesses, the SBR RT application avoids scattered address patterns and global memory storage with large strides between accesses.

Efficient parallelization of the double count removal procedure is a very important aspect of the GPU optimization for the SBR code. In order to parallelize this portion of the code, adjacent rays need to be split into different classes. The convenience of the icosahedron spawning pattern means with three distinct classes, all adjacent rays can be a member of a different class. Figure 12

shows an example of the different classes displayed as three colors plotted over the faces of the icosahedron.

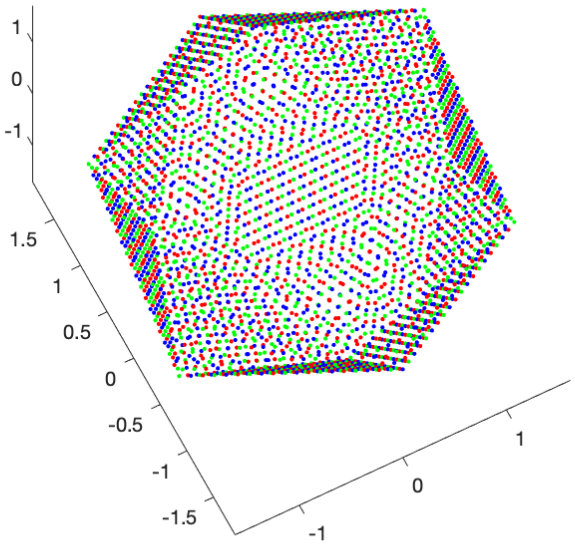


Figure 12. Adjacent Classes of Icosahedron: The three distinct classes displayed as red, green, and blue dots plotted over the icosahedron display how adjacent rays are members of different classed.

The reason the adjacent rays need to be split into separate classes is to ensure that no two adjacent rays that double count on the same observation point are processed at the same time. If the rays were processed at the same time, for example, then there is a race condition between the threads to see which thread will update the field first. The threads can also execute the code at the same instance in time meaning the rays that should have been flagged as duplicates are not.

The speedup from these acceleration modifications to the code can be seen in Figure 13 below. The timing information for the sequential code was obtained during an early version of the SBR RT algorithm that did not include double count removal. The double count removal should slow the algorithm down even further as this is one of the slowest portions of the RT algorithm. This is because the algorithm was migrated to a GPU version early on to help with the development process by lessening the execution time needed for a single run. The sequential code was run

exclusively on the CPU using only a single thread. The table shows how the GPU can dramatically reduce the execution time of the SBR RT algorithm making it an even more effective tool for solving larger structures. While the number of rays used in an execution of the SBR algorithm is important for convergence, it is usually not necessary to launch hundreds of millions of rays or hundreds of reflections. The results displayed in Chapter 5 demonstrate the accurate solutions obtained using only millions of rays.

Number of rays	Number of reflections	Sequential code	Parallelized code	Speedup
10,000	20	4.251 s	0.344 s	12.36 ×
1,000,000	80	24 m 40.096 s	11.125 s	133.04 ×
100,000,000	40	21 h 14 m 17 s	9 m 46 s	130.47 ×

Figure 13. GPU Speedup: Speedup from the modifications to the SBR RT algorithm when compared to the original serial code.

CHAPTER 5: RESULTS

5.1 Ray Tracing Application Validation

Validation of the SBR RT application is presented here using several tunnels of various uniform cross sections. Tunnels are a challenging environment for RT solvers due to the large distances covered, the curvature of the backs of the tunnels, and the roughness present along the walls. Tunnel environments do provide the benefit of only considering the reflected rays and not the transmitted rays since any transmitted ray would immediately leave the region of interest where the observation points are. The reason this environment was chosen for validation is because it is a corner case for the RT algorithm and enables a rigorous test for the algorithms ability to work well in these difficult environments. The simulations in these tests were performed by creating a line of reception points directed down the long axis of the tunnels at evenly separated intervals. The solutions obtained using the SBR RT algorithm are compared with computed results of other CEM solvers, physically measured data, as well as state of the art commercial ray launching software.

The first tunnels tested are compared to another SBR ray tracing algorithm found here [12]. The reason these tunnels were used initially was to test how our RT algorithm compared to existing algorithms. The two tunnels are meant to simulate the application of the RT algorithm in real world traffic tunnel environments with real world parameters. The frequency used in these tests is 1 GHz since this is around the carrying frequencies of cell phones [12]. The first tunnel simulated is a rectangular tunnel that covers a distance of 150 meters with a cross section of 8 meters by 5 meters. The second tunnel has a rectangular base but a curved ceiling and also stretches a distance of 150 meters. Figure 14(a) and Figure 14(b) both show the application of the SBR RT algorithm to these

two tunnels. The figure above demonstrates that the SBR RT algorithm agrees closely with published results of a different SBR method. Figure 14(a) shows very close agreement from 40m to 140m down the length of the tunnel. The discrepancy between our SBR solution and the results in Figure 14(b) is a product of the different number of discretized segments on the ceiling of the curved tunnel. These two simulations demonstrate the accuracy and validity of our algorithm compared to another SBR algorithm.

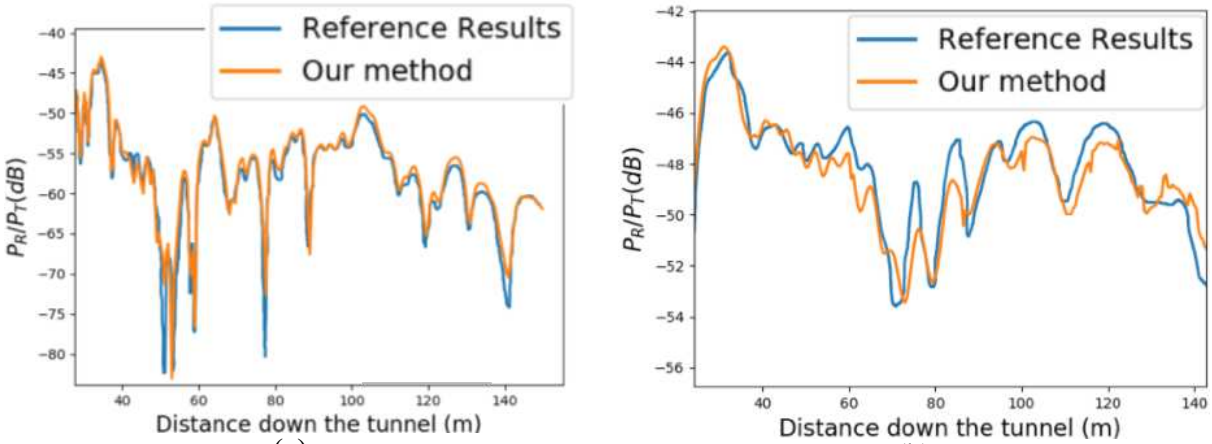


Figure 14. SBR and SBR: Received/transmitted powers in two tunnels: comparisons of solutions using our SBR RT method with previously verified results obtained by a different version of SBR modeling [12]. (a) 5m x 8m rectangular tunnel (b) arched tunnel. The frequency used in these simulations is 1GHz.

Next, the SBR RT algorithm was tested for a much longer tunnel against an image-theory based ray tracing algorithm presented here [8]. The results from this simulation are displayed in Figure 15 below. Image-theory based ray tracing is a computationally slower but more accurate solver when compared to SBR RT solvers. The image-theory model uses an exact ray path computation, i.e., the image positions of every reflection for every receiver position across every facet within a scene are computed. This means image theory has an exponential asymptotic complexity of $O(N^k)$ scaling by the number of observation points (N) and the number of reflections (k). However, due to the exact path calculations from the image point reflections, the phase error

caused by slight variation in ray paths is eliminated. There is exceptional agreement between the two results as far as 900 m down the tunnel, which demonstrates the ability of the SBR algorithm to correctly compute transmission path lengths at very large distances from the transmitter, a feature traditionally reserved for the dramatically less efficient image-theory ray tracing, and a necessary capability for this proposed research.

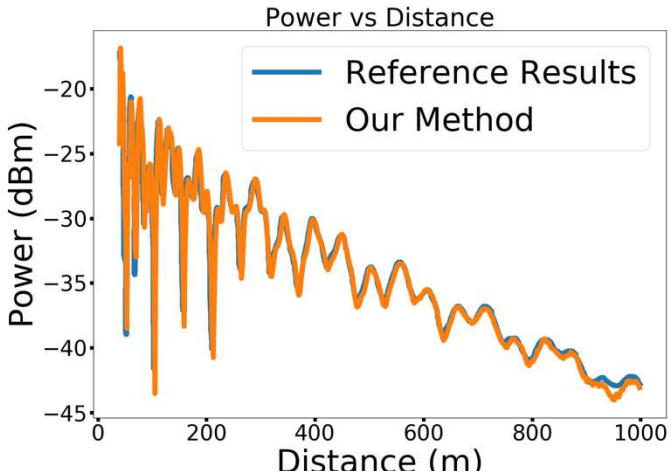


Figure 15. SBR and Image Theory: Received/transmitted power using our method compared published image theory RT results [8]. These results were obtained on a lossy waveguide of dimensions 4m x 4m of ε_r =5 with the receivers and a transmitter located 2.1m from the floor and 1.1m from the wall. The frequency is 1GHz.

The next tunnel tested is an empty arched tunnel presented here [13] with the SBR solution compared to results obtained from a simulation using Fast Multipole Method (FMM) combined with the fast Fourier transform (FFT) [13]. The results for this simulation are displayed in Figure 16. The SBR RT solver shows close agreement to the results from the FMM-FFT solver, an accurate full-wave CEM technique. The reason for the disagreement further down the tunnel is from the lack of including edge diffraction in the SBR algorithm.

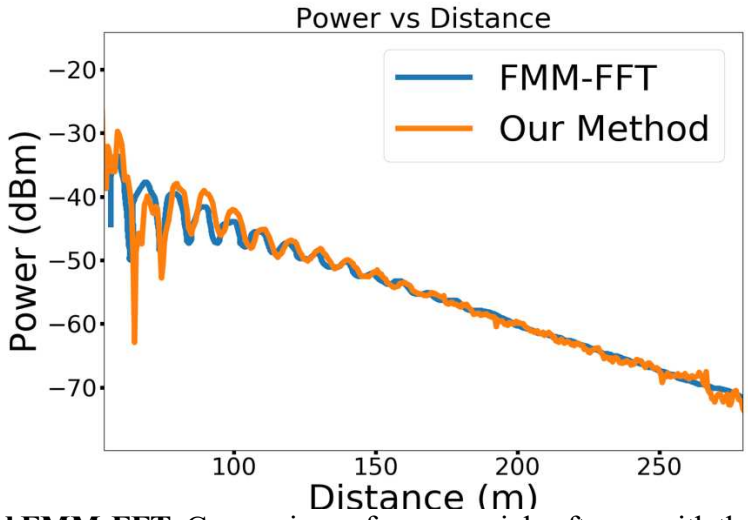


Figure 16. SBR and FMM-FFT. Comparison of commercial software with the SBR RT algorithm presented here and results from an FMM-FFT solver. These results were obtained on an arched traffic tunnel found here [13]. The frequency used in the simulation is 915 MHz.

The SBR RT algorithm is also compared with a commercial SBR RT program called REMCOM's Wireless InSite on the same tunnels used in Figure 14 and the results from this comparison are shown below in Figure 17.

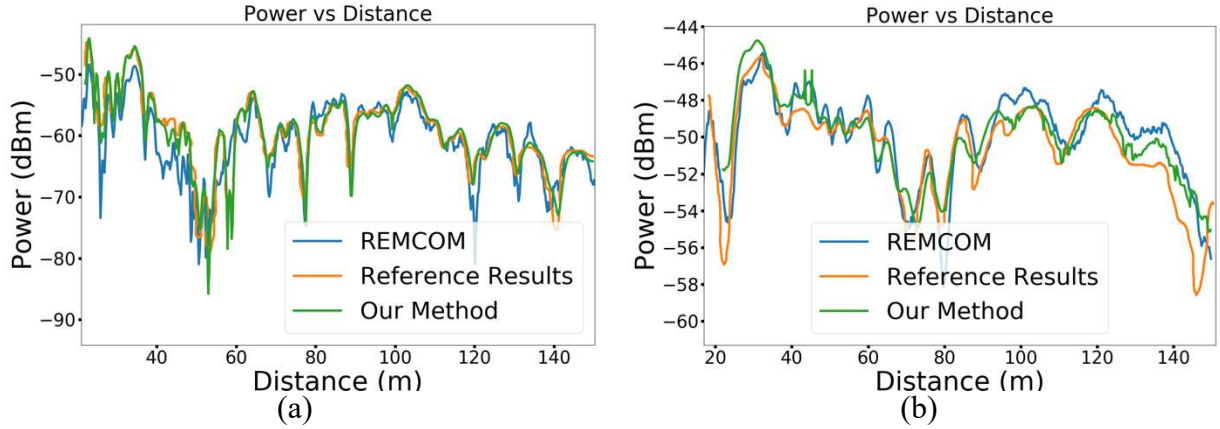


Figure 17. SBR and REMCOM: The comparison of REMCOM Wireless InSite with the SBR RT algorithm. (a) The rectangular 5m x 8m tunnel and (b) the arched tunnel. The frequency used in these two tests is 1GHz.

Additionally, ray tracing can be used to construct received signal strength (RSS) maps via simulation of the SBR RT algorithm with a 2D grid of observation points. Traditionally, RSS maps are constructed by placing a transmitting dipole antenna inside some structure and measuring a

grid of observation points manually [14]. This manual collection of data to create these maps is expensive, time-consuming, and causes any inaccuracies if the environment changes. The SBR RT algorithm can create maps as accurately as physical measurements since the error in the measurements is exchanged with numerical error in the solver. Furthermore, the SBR RT algorithm reduces cost and time for creating the maps since the data for the map can be simulated without hiring people to physically walk through a structure collecting data. The SBR RT algorithm is applied to compute the RSS map within one of the tunnels in the Colorado School of Mines Edgar Research Mine. The results from this simulation are displayed in Figure 18.

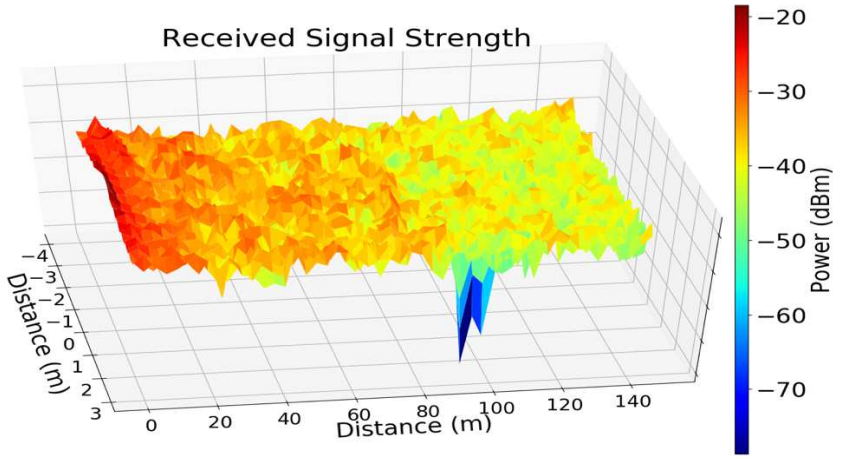
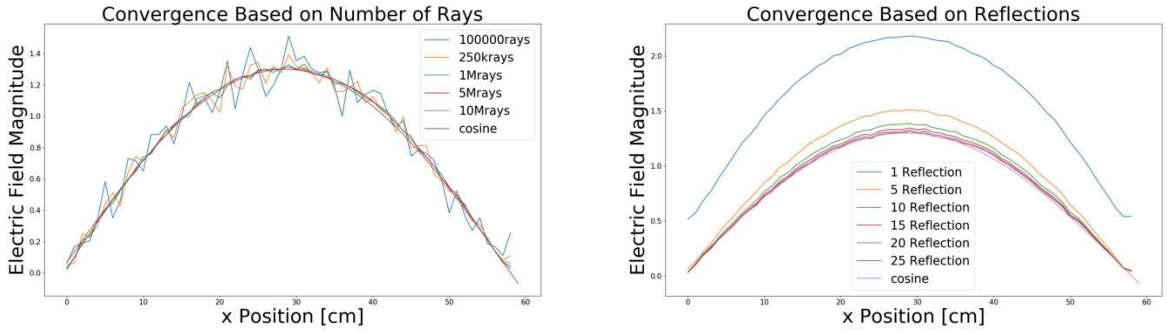


Figure 18. Received Signal Strength: Mapping of the received signal strength for a section of a tunnel in the Colorado School of Mines Edgar Mine demonstrates the ability of the SBR RT algorithm to map the received signal strength with only LiDAR data. The frequency used in the simulation is 2.45 GHz.

The path loss from the transmitting antenna is computed using our SBR RT method and demonstrates how the structure of the tunnel affects signal strength in different areas. For instance, the sharp spike along the surface at around 90 m is a result of the tunnel jutting out and blocking the line of sight view to the antenna thereby decreasing the RSS.

Convergence of the SBR RT algorithm is computed using a PEC waveguide by calculating the electric field at a cross-sectional plane at a distance of 10 wavelengths from the transmitting antenna. The size of the waveguide is constructed such that the TE_{10} mode is the only excited mode. The analytical solution for the TE_{10} mode is a cosine and the comparison between the analytical cosine and the SBR RT algorithm is displayed in Figure 19. The results from this convergence test show that the number of rays and reflections for the rectangular waveguide does not need to be very high in order to reach convergence.



(a) (b)

Figure 19. Convergence: Convergence for the SBR RT algorithm for a PEC electric waveguide by changing the number of (a) rays and (b) reflections. The frequency is 375 MHz.

5.2 Results from Measurements

In addition to comparing the RT solver with published results, the solver is compared to data collected from RF measurements. The equipment used in the measurements consisted of a transmitting antenna, a receiving antenna, a software defined radio (SDR), and a spectrum analyzer. The SDR was used to drive the signal of the transmitting antenna and the spectrum analyzer was used to read out a time-averaged signal at the frequency of the transmitting signal. The environment used for this test was an open field with no buildings and limited obstacles that would affect the signal. Once the measured results were obtained on an open field the analytical

solution was computed using Friis formulas for a single ground plane reflection. Results from this simulation are displayed in Figure 20.

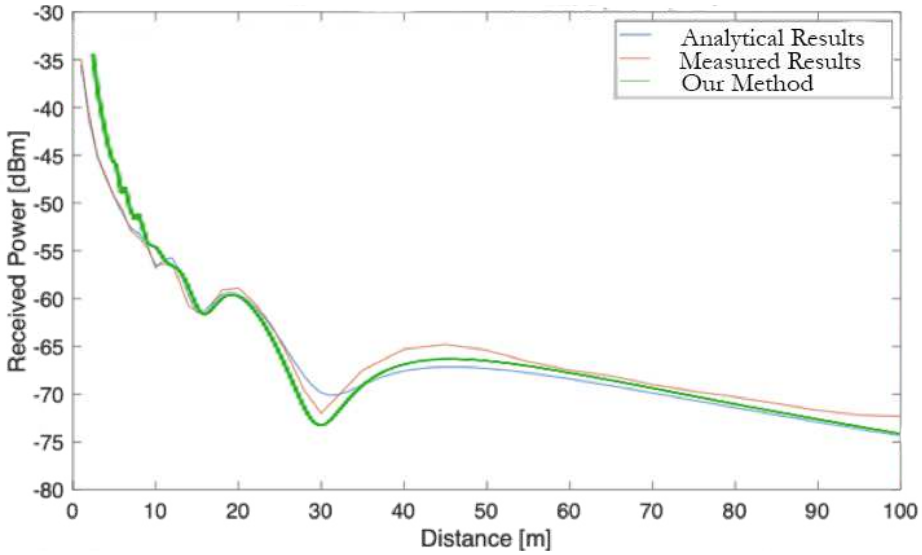


Figure 20. Measured and SBR: Comparison of analytical and measured received power for open domain, ground reflection. The frequency used is 1GHz.

In particular, the validation with measured data demonstrates that our simulated solution closely agrees with real-world signal behavior. As shown in Figure 20, the agreement between the collected data from the RF measurement system, the analytically calculated data, and the SBR RT data is excellent. This agreement demonstrates empirically that the RF measurement system is valid and the SBR RT algorithm approximates the solution very well.

CHAPTER 6: CONCLUSION

6.1 Conclusions and Future Work

The developed cone-based shooting bouncing rays ray tracing solver is an effective and accurate computational electromagnetics solver for large environments. The SBR RT algorithm is organized in a way to enable ease of modification for the addition of any new technique that could improve the accuracy or speed of the code. The algorithm incorporates techniques such as ray batching and double count removal to improve the accuracy of the solution. The use of graphics processing units and the optimization of the code has created a solver that is efficient in launching millions of rays and converging to solutions quickly.

The numerical results support the conclusion that SBR RT can be applied to electrically large structures quickly at only a modest accuracy cost. The results demonstrate that this loss in accuracy is little and an acceptable tradeoff for the time saved when compared to full-wave solvers. The speed gained from the acceleration, achieved from the high level of parallelism, has created an algorithm that competes with and exceeds, from both a speed and accuracy standpoint, current state of the art commercial software. The capability of the SBR RT algorithm to contend with commercial software means that the algorithm developed has achieved the original goals of computational efficiency and solution correctness.

This work describes only partially the types of problems to which the SBR RT algorithm can be applied. In addition to the application of tunnel environments, the solver can be used to further analyze indoor environments with techniques such as RSS mapping. Furthermore, the base algorithm can be expanded to include transmission through surfaces, the edge diffraction of rays,

and exact ray path correction by hybridizing the method with image-theory solvers while maintaining the superior time complexity of SBR.

This algorithm is the first step in enabling the Computational Electromagnetics Laboratory at CSU to solve problems that would not be possible using FEM and MoM. The algorithm is also a key component to creating more advance hybridized CEM solvers by combining the computational speed of high frequency asymptotic solvers with the accuracy of full-wave solvers. Further development of the algorithm will hopefully enable more accurate solutions in very challenging environments and open the door for complex analysis in environments that traditionally have not been solvable.

REFERENCES

[1] B. M. Notaroš, C. Key, S. B. Manić, B. Troksa, M. M. Ilić and S. Savić, "Efficient electromagnetic modeling of wireless signal propagation in underground mine tunnels," *2018 International Conference on Electromagnetics in Advanced Applications (ICEAA)*, Cartagena des Indias, 2018, pp. 625-625.

[2] M.F.Cátedra and J.Perez, Cell Planning for Wireless Communications. Norwood, MA, USA: Artech House, 1999.

[3] B. M. Notaros, *Electromagnetics,* New Jersey : PEARSON Prentice Hall; 2010.

[4] B. Troksa, C. Key, F. Kunkel, S. V. Savić, M. M. Ilić and B. M. Notaroš, "Ray tracing using shooting-bouncing technique to model mine tunnels: Theory and verification for a PEC waveguide," *2018 International Applied Computational Electromagnetics Society Symposium (ACES)*, Denver, CO, 2018, pp. 1-2.

[5] C. Key, B. Troksa, F. Kunkel, S. V. Savić, M. M. Ilic and B. M. Notaroš, "Comparison of Three Sampling Methods for Shooting-Bouncing Ray Tracing Using a simple Waveguide Model," *2018 IEEE International Symposium on Antennas and Propagation & USNC/URSI National Radio Science Meeting*, Boston, MA, 2018, pp. 2273-2274.

[6] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: a general purpose ray tracing engine. ACM Trans. Graph. 29, 4, Article 66 (July 2010), 13 pages.

[7] A. Santos, J. M. Teixeira, T. Farias, V. Teichrieb, and J. Kelner, "Understanding the Efficiency of kD-tree Ray-Traversal Techniques over a GPGPU Architecture," *International Journal of Parallel Programming*, vol. 40, no. 3, pp. 331–352, 2011.

[8] D. Didascalou, *Ray-optical wave propagation modeling in arbitrarily shaped tunnels*, 2000.

[9] V. Mohtashami and A. A. Shishegar, "A new double—counting cancellation technique for ray tracing using separation angle distribution," *2008 IEEE International RF and Microwave Conference*, Kuala Lumpur, 2008, pp. 306-310.

[10] N. Noori, A. A. Shishegar and E. Jedari, "A New Double Counting Cancellation Technique for Three-Dimensional Ray Launching Method," *2006 IEEE Antennas and Propagation Society International Symposium*, Albuquerque, NM, 2006, pp. 2185-2188.

[11] C Woolley, "GPU optimization fundamentals", 2013.

[12] Shin-Hon Chen and Shyh-Kang Jeng, "SBR image approach for radio wave propagation in tunnels with and without traffic," in *IEEE Transactions on Vehicular Technology*, vol. 45, no. 3, pp. 570-578, Aug. 1996.

[13] A. C. Yucel, W. Sheng, C. Zhou, Y. Liu, H. Bagci and E. Michielssen, "An FMM-FFT Accelerated SIE Simulator for Analyzing EM Wave Propagation in Mine Environments Loaded With Conductors," in *IEEE Journal on Multiscale and Multiphysics Computational Techniques*, vol. 3, pp. 3-15, 2018.

[14] J. Zhang, G. Han, N. Sun and L. Shu, "Path-Loss-Based Fingerprint Localization Approach for Location-Based Services in Indoor Environments," in *IEEE Access*, vol. 5, pp. 13756-13769, 2017.

APPENDIX A: PUBLICATIONS OF THE AUTHOR

Conference Papers

[1] B. Troksa, C. Key, F. Kunkel, S. V. Savić, M. M. Ilić and B. M. Notaroš, "Ray tracing using shooting-bouncing technique to model mine tunnels: Theory and verification for a PEC waveguide," *2018 International Applied Computational Electromagnetics Society Symposium (ACES)*, Denver, CO, 2018, pp. 1-2.

[2] C. Key, B. Troksa, F. Kunkel, S. V. Savić, M. M. Ilic and B. M. Notaroš, "Comparison of Three Sampling Methods for Shooting-Bouncing Ray Tracing Using a simple Waveguide Model," *2018 IEEE International Symposium on Antennas and Propagation & USNC/URSI National Radio Science Meeting*, Boston, MA, 2018, pp. 2273-2274.

[3] B. M. Notaroš, C. Key, S. B. Manić, B. Troksa, M. M. Ilić and S. Savić, "Efficient electromagnetic modeling of wireless signal propagation in underground mine tunnels," *2018 International Conference on Electromagnetics in Advanced Applications (ICEAA)*, Cartagena des Indias, 2018, pp. 625-625.

APPENDIX B: CUDA C++ CODE FOR POST PROCESSING

```
#include <cuda.h>
#include <cuda_runtime.h>
#include <optixu/optixu_math_namespace.h>
#include <stdio.h>
#include <thrust/device_vector.h>
#include "commonStructs.h"
#include "primeCommon.h"

//Constants
#define pi              3.14159265358979323846
#define eps_0   8.85418782e-12;
#define mu_0    1.25663706e-6;
#define RPT     1                                               //rays per
thread
#define SPT     1                                               //spheres per
thread


//-------------------------------------------------------------------------
---
// Operations preformed with the electric field


__host__ __device__ complex3 operator*(const float3 &x, const
thrust::complex<double> &a) {
        return complex3((double)x.x*a, (double)x.y*a, (double)x.z*a);
}

__host__ __device__ complex3 operator*(const thrust::complex<double> &a,
const float3 &x) {
        return complex3((double)x.x*a, (double)x.y*a, (double)x.z*a);
}

__host__ __device__ complex3 operator*(const complex3 &a, const float3 &x) {
        return complex3((double)x.x*a.x, (double)x.y*a.y, (double)x.z*a.z);
}

__host__ __device__ thrust::complex<double> operator*(const
thrust::complex<double> &a, const float &x) {

        return thrust::complex<double>(a * (double)x);
}
```

```
__host__ __device__ thrust::complex<double> operator/(const
thrust::complex<double> &a, const float &x) {

	return thrust::complex<double>(a / (double)x);
}

__host__ __device__ complex3 operator*(const complex3 &x, const
thrust::complex<double> &a) {
	complex3 c(a*x.x, a*x.y, a*x.z);
	return c;
}

__host__ __device__ complex3 operator*(const complex3 &x, const complex3 &y)
{
	complex3 c(y.x*x.x, y.y*x.y, y.z*x.z);
	return c;
}

__host__ __device__ complex3 operator+(const complex3 &a, const complex3 &b)
{
	complex3 c(a.x + b.x, a.y + b.y, a.z + b.z);
	return c;
}

__host__ __device__ complex3 operator-(const complex3 &a, const complex3 &b)
{
	complex3 c(a.x - b.x, a.y - b.y, a.z - b.z);
	return c;
}


__device__ void printFloat3(float3 a) {
	printf("x:%f, y:%f, z:%f\n", a.x, a.y, a.z);
}

__host__ __device__ void printComplex3(complex3 a) {
	printf("x: %f, %f\ty: %f, %f\t z:%f, %f\n", a.x.real(), a.x.imag(),
a.y.real(),
		a.y.imag(), a.z.real(), a.z.imag());
}

__device__ void printComplex(thrust::complex<double> a) {
	printf("x: %f, %f\n", a.real(), a.imag());
}
```

```
__device__ thrust::complex<double> dot(const complex3 &a, const float3 &b) {
	return ((a.x * (double)b.x) + (a.y * (double)b.y)) + (a.z *
(double)b.z);
}

//------------------------------------------------------------------------
---

//Computes the initial field of a ray based on a radiation pattern
__device__ void get_initial_field(float3 ray_dir, int index, complex3*
efields_d, constants* constants_d) {
	double sphere_theta = acos((double)optix::normalize(ray_dir).z);
	double sphere_phi = pi / 2;
	sphere_phi = atan2(ray_dir.y, ray_dir.x);
	double lambda = 2.99e8 / constants_d->frequency;
	double coefficient = sqrt(120 * pi*1.64*1e-3 / 2 / pi);
	//Hertzian Dipole Radiation Pattern
	efields_d[index].x = cos(sphere_theta) * cos(sphere_phi) *
sin(sphere_theta) * lambda / 2.0;
	efields_d[index].y = cos(sphere_theta) * sin(sphere_phi) *
sin(sphere_theta) * lambda / 2.0;
	efields_d[index].z = -sin(sphere_theta) * sin(sphere_theta) * lambda /
2.0;
}


// Update Electric Field with Fresnel Coeffecients
__device__ void get_unit_vectors(int triId, float3 ray_dir, float3*
normals_d, complex3* efields_d,
	constants* constants_d, int k) {
	ray_dir = optix::normalize(ray_dir);
	float3 surf_norm = normals_d[triId];
	surf_norm = optix::normalize(surf_norm);
	if (optix::dot(surf_norm, ray_dir) > 0) {
		surf_norm *= -1;
	}
	float3 reflection_dir = ray_dir - ((2 * optix::dot(ray_dir,
surf_norm))*(surf_norm));
	reflection_dir = optix::normalize(reflection_dir);
	// Calculate cross products
	complex3 Ei = efields_d[k];
	float3 E_in = optix::cross(ray_dir, surf_norm);
	E_in = optix::normalize(E_in);
```

```
        float3 E_ip = optix::cross(E_in, ray_dir);
        E_ip = optix::normalize(E_ip);
        float3 E_rp = 2 * optix::dot(surf_norm, E_ip)*surf_norm - E_ip;
        E_rp = optix::normalize(E_rp);

        // Volume/rock constants
        double lambda = 2.9979e8 / constants_d->frequency;
        double theta_i = acos(optix::dot(reflection_dir, surf_norm));
        thrust::complex<double> epsComplex = constants_d->eps_f - constants_d-
>imaginary_j * 60 * constants_d->sigma_f * lambda;
        //Fresnel Coeffecients
        thrust::complex<double> gamma_n = (cos(theta_i) -
thrust::sqrt(epsComplex - sin(theta_i)*sin(theta_i))) /

        (cos(theta_i) + thrust::sqrt(epsComplex - sin(theta_i)*sin(theta_i)));
        thrust::complex<double> gamma_p = (epsComplex * cos(theta_i) -
thrust::sqrt(epsComplex - sin(theta_i)*sin(theta_i))) /

        (epsComplex * cos(theta_i) + thrust::sqrt(epsComplex -
sin(theta_i)*sin(theta_i)));

        complex3 Ern, Erp;

        Erp.x = E_ip.x*E_rp.x + E_ip.y*E_rp.x + E_ip.z*E_rp.x;
        Erp.y = E_ip.x*E_rp.y + E_ip.y*E_rp.y + E_ip.z*E_rp.y;
        Erp.z = E_ip.x*E_rp.z + E_ip.y*E_rp.z + E_ip.z*E_rp.z;

        Ern.x = E_in.x*E_in.x + E_in.x*E_in.y + E_in.x*E_in.z;
        Ern.y = E_in.y*E_in.x + E_in.y*E_in.y + E_in.y*E_in.z;
        Ern.z = E_in.z*E_in.x + E_in.z*E_in.y + E_in.z*E_in.z;

        complex3 Er = (Ern*gamma_n + Erp*gamma_p) * Ei;

        efields_d[k] = Er;
}

//Change Sphere Radius
__device__ double updateRadius(float3 sorigin, float3 rayOrigin, float3
rayDirection, double rayDistance, double alpha) {
        double distance = abs(optix::dot(sorigin - rayOrigin, rayDirection));
        return (distance + rayDistance) * alpha / sqrt(3.0);
}

// Calculates intersection between a ray/vector (previous - current) and a
sphere located a sorigin
```

```
__device__ float checkSphere(float3 previous, float3 current, float3 sorigin,
double distance, double alpha) {
	float3 rayDirection = optix::normalize(current - previous);
	double radius = updateRadius(sorigin, previous, rayDirection,
distance, alpha);
	//radius = fmin(.05, radius);
	//radius *= 10;
	//printf("radius: %f\n", radius);
	if (radius <= 0) return -1;
	float3 O = previous - sorigin;
	float rsquared = radius * radius;
	float b = optix::dot(O, rayDirection);
	float c = optix::dot(O, O) - rsquared;
	// The sphere is behind or surrounding the start point.
	if (b > 0){
		//return -1;
	}

	// Flatten p into the plane passing through c perpendicular to the
ray.
	// This gives the closest approach of the ray to the center.
	float disc = b*b - c;

	// Closest approach is outside the sphere.
	if (disc < 0) {
		return -2;
	}

	// Calculate distance from plane where ray enters/exits the sphere.
	float t = -b - sqrt(disc);
	if (t < 0.0f) t = 0;


	float3 i = t * rayDirection + previous;

	bool canHit = optix::length(i - previous) < optix::length(current -
previous);
	if (canHit) {
		return(optix::dot(sorigin - previous, rayDirection));
	}
	return(-4);
}


// Sychronize the update to the field at the observation points
```

```
__device__ void updateSphereField(complex3& sphereFields_d, complex3 efield){
        thrust::complex<double>* temp = &sphereFields_d.x;
        double* real = (double*)temp;
        double* imag = real + 1;
        atomicAdd(real, efield.x.real());
        atomicAdd(imag, efield.x.imag());

        temp = &sphereFields_d.y;
        real = (double*)temp;
        imag = real + 1;
        atomicAdd(real, efield.y.real());
        atomicAdd(imag, efield.y.imag());

        temp = &sphereFields_d.z;
        real = (double*)temp;
        imag = real + 1;
        atomicAdd(real, efield.z.real());
        atomicAdd(imag, efield.z.imag());
}

//Remove the double counted rays
__device__ void doubleCountFast(RayData* rayData_d, AdjacencyMap* adjMap_d,
int k, SphereData* sphereData_d,
        PostProccessData ppbd){
        int r1 = ppbd.sphereTrio_d[k].rayId;
        constants* constants_d = ppbd.constants_d;
        //check for adjacent rays
        if (!adjMap_d->beenUsed[k]) {
                double distance = ppbd.sphereTrio_d[k].pathLength;
                thrust::complex<double> impedance(73.0, 42.5);
                impedance = 1.0/(impedance * 4.0);
                thrust::complex<double> coefficient = impedance*(constants_d-
>imaginary_j * constants_d->omega * -4e-7 / 16.0);
                thrust::complex<double> pshift = thrust::exp(-constants_d-
>imaginary_j * constants_d->beta_v * (distance)) / distance;

        updateSphereField(sphereData_d[ppbd.sphereTrio_d[k].sphereId].field,
ppbd.sphereTrio_d[k].field*pshift*coefficient);
                //Update All Adjacent Rays
                for (int j = 0; j<ppbd.size; j++) {
                        int r2 = ppbd.sphereTrio_d[j].rayId;
                        if (ppbd.sphereTrio_d[k].sphereId ==
ppbd.sphereTrio_d[j].sphereId) {
                                if (adjMap_d->check(r1,r2)) {
                                        adjMap_d->beenUsed[j] = true;
```

```
                                }
                        }
                }
        }
}

//Kernel for double count removal
__global__ void updateFieldGPU(RayData* rayData_d, AdjacencyMap* adjMap_d,
SphereData* sphereData_d, int labelNum,
        PostProccessData ppbd){

        int tid = threadIdx.x;    //Thread index within a thread block
        int blockid = blockIdx.x; //Block index within the grid
        int B = blockDim.x;
        int startIndex = B*blockid*SPT + tid;
        for (int k = startIndex; k < startIndex + B*SPT; k += B) {
                if(k < ppbd.size &&
rayData_d[ppbd.sphereTrio_d[k].rayId].label == labelNum){

        doubleCountFast(rayData_d,adjMap_d,k,sphereData_d,ppbd);
                }

        }
}

//Initialize boolean array for double count removal
__global__ void prepUsedArray(int sphereHits, AdjacencyMap* adjMap_d){

        int tid = threadIdx.x;    //Thread index within a thread block
        int blockid = blockIdx.x; //Block index within the grid
        int B = blockDim.x;
        int startIndex = B*blockid*SPT + tid;
        for (int k = startIndex; k < startIndex + B*SPT; k += B) {
                if(k < sphereHits){
                        adjMap_d->beenUsed[k] = false;
                }


        }
}

//Update electric field for each ray based on Fresnel coeffecients
__global__ void updateFieldKernelGPU(RayData* rayData_d, SphereData*
sphereData_d, float3* normals_d, PostProccessData ppbd)
{
```

```
        int tid = threadIdx.x;    //Thread index within a thread block
        int blockid = blockIdx.x; //Block index within the grid
        int B = blockDim.x;
        int startIndex = B*blockid*RPT + tid;
        for (int k = startIndex; k < startIndex + B*RPT; k += B) {
                if (k >= ppbd.constants_d->nRaysPerBatch) break;
                if (!rayData_d[k].done) {
                        float3 previous = rayData_d[k].previous;
                        float3 current = rayData_d[k].current;
                        float3 ray_dir = current - previous;
                        get_unit_vectors(rayData_d[k].triId, ray_dir,
normals_d, ppbd.efields_d, ppbd.constants_d, k);
                        ppbd.efields_d[k].distance += optix::length(ray_dir);
                }
        }
        __syncthreads();
}

//Fill ray hit data array
__device__ void populateIntersections(SphereData* sphereData_d, float3
previous, float3 current,
        double alpha, int rayID, PostProccessData ppbd) {
        double hit = -9;
        for (int k = 0; k<ppbd.constants_d->nSpheres; ++k) {
                hit = checkSphere(previous, current, sphereData_d[k].position,
ppbd.efields_d[rayID].distance, alpha);
                if (hit >= 0) {
                        int index = atomicAdd(ppbd.globalIndex_d, 1);
                        Trio t(rayID, k, ppbd.efields_d[rayID].distance + hit);
//add hit here
                        t.field = ppbd.efields_d[rayID];
                        ppbd.sphereTrio_d[index] = t;
                        atomicAdd(&sphereData_d[k].count, 1);
                }
        }
}

//Fill ray hit data array
__global__ void populateSphereTrio(int reflection, RayData* rayData_d,
PostProccessData ppbd, SphereData* sphereData_d){
        int tid = threadIdx.x;    //Thread index within a thread block
        int blockid = blockIdx.x; //Block index within the grid
        int B = blockDim.x;
        int startIndex = B*blockid*RPT + tid;
```

```
	for (int k = startIndex; k < startIndex + B*RPT; k += B) {
		if (k >= ppbd.constants_d->nRaysPerBatch) break;
		if (!rayData_d[k].done) {
			float3 previous = rayData_d[k].previous;
			float3 current = rayData_d[k].current;
			float3 ray_dir = current - previous;
			//Find sphere that are hit and add them to array
			populateIntersections(sphereData_d, previous, current,
rayData_d[k].alpha, k, ppbd);
		}
	}
	__syncthreads();
}

//Compute all the intersections of a ray and an observation point(sphere)
__device__ void findAllIntersectionCounts(SphereData* sphereData_d, float3
previous, float3 current, complex3 &efield
	, double alpha, PostProccessData ppbd) {
	double hit = -1;
	for (int k = 0; k<ppbd.constants_d->nSpheres; ++k) {
		hit = checkSphere(previous, current, sphereData_d[k].position,
efield.distance, alpha);
		if (hit >= 0) {
			atomicAdd(ppbd.count_d, 1);
		}
	}
}

//Compute number of ray-sphere itersections that occur during the current
reflection
__global__ void getCounts(int reflection, RayData* rayData_d, SphereData*
sphereData_d, PostProccessData ppbd)
{
	int tid = threadIdx.x;	//Thread index within a thread block
	int blockid = blockIdx.x; //Block index within the grid
	int B = blockDim.x;
	int startIndex = B*blockid*RPT + tid;
	for (int k = startIndex; k < startIndex + B*RPT; k += B) {
		if (k >= ppbd.constants_d->nRaysPerBatch) break;
		if (!rayData_d[k].done) {
			float3 previous = rayData_d[k].previous;
			float3 current = rayData_d[k].current;
			float3 ray_dir = current - previous;
			if (reflection == 0) {
```

```
                                get_initial_field(ray_dir, k, ppbd.efields_d,
ppbd.constants_d);
                        }
                        findAllIntersectionCounts(sphereData_d, previous,
current, ppbd.efields_d[k], rayData_d[k].alpha, ppbd);
                }
        }
        __syncthreads();
}

//Main function of the post processing
extern "C" void calculateField(int reflection, RayData* rayData,
PostProccessData* ppbdPTR,
        SphereData* sphereData_d, float3* normals_d, constants* constant,
AdjacencyMap* adjMap_d) {

        bool* beenUsed_d;
        int threadsPerBlock = 128;
        int block = ceil(((double)constant->nRaysPerBatch) /
((double)threadsPerBlock*RPT));
        PostProccessData ppbd = *ppbdPTR;

        //Copy Ray information from Host
        RayData* rayData_d;
        cudaMalloc((void**)&rayData_d, constant->nRaysPerBatch *
sizeof(RayData));
        cudaMemcpy(rayData_d, rayData, constant->nRaysPerBatch *
sizeof(RayData), cudaMemcpyHostToDevice);
        ppbd.cudaInit();

        //Determine how many spheres have been hit
        getCounts << <block, threadsPerBlock >> >(reflection,
rayData_d,sphereData_d, ppbd);
        cudaDeviceSynchronize();

        //Create an array of information containing the hits and populate it
        ppbd.doubleCountInit();
        populateSphereTrio << <block, threadsPerBlock >> >(reflection,
rayData_d, ppbd, sphereData_d);
        cudaDeviceSynchronize();

        //Create a boolean array for double count removal
        cudaMalloc((void **) &beenUsed_d, sizeof(int)*ppbd.size);
        cudaMemcpy(&(adjMap_d->beenUsed), &beenUsed_d, sizeof(bool*),
cudaMemcpyHostToDevice);
```

```
        block = ceil(((double) ppbd.size) / ((double)threadsPerBlock*SPT));
        prepUsedArray<< <block, threadsPerBlock >> >(ppbd.size, adjMap_d);
        cudaDeviceSynchronize();

        //Preform post processing on the three classes of rays
        for(int labelNum = 0; labelNum < 3; labelNum++){
                updateFieldGPU<< <block, threadsPerBlock >> >(rayData_d,
adjMap_d, sphereData_d, labelNum, ppbd);
                cudaDeviceSynchronize();
        }

        //Update each rays electric field
        block = ceil(((double)constant->nRaysPerBatch) /
((double)threadsPerBlock*RPT));
        updateFieldKernelGPU << <block, threadsPerBlock >> >(rayData_d,
sphereData_d, normals_d, ppbd);

        //free any memory
        ppbd.cudaDestruct();
        cudaFree(rayData_d);
        cudaFree(beenUsed_d);
}
```