Simulating RF Field Propagation with Stochastic Ray Tracing

by

Timothy J. Kleinow

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Engineering
(Computer Engineering)
in the University of Michigan-Dearborn
2020

Master's Thesis Committee:

Associate Professor Sridhar Lakshmanan, Chair
Associate Professor Paul Watta
Lecturer Michael Putty

Table of Contents

List of Figures

Abstract

This work details the development of an application for fast simulations of the steady state far-field electromagnetic (EM) field strength and power in arbitrary environments. These environments consist of radiating antennas and solid 3-Dimensional (3D) occluding bodies. The simulation is accomplished using a variation of stochastic ray tracing that uses Monte Carlo integration to solve the light transport equation. The primary variations to the standard algorithm that are proposed here are twofold. First, a grid acceleration structure is used to reduce the number of computationally expensive ray-triangle intersection tests that need to be performed. The grid is chosen over other acceleration structures, as the requirement to compute field strength within a volume necessitates stepping the rays through the space regardless of whether the grid is used or not. The second variation is the implementation of diffraction. Existing ray tracers neglect diffraction as they typically deal with light of optical frequencies above 400 THz, where the amount of diffracted light around any large-scale object is negligible. As this application must handle much lower frequencies to simulate radio interactions, diffraction is implemented using a novel technique that involves extending the edges of triangles by constant width “diffraction margins” and allowing rays that hit the margins to bend inward probabilistically according to the Heisenberg momenta uncertainty associated with the new information about the position of the ray’s associated “photon bundle” due to its closeness to the surface.

Chapter 1: Introduction

In this chapter we will be considering the motivation for the development of this application and the technical background and problems facing it, as well as giving a review of existing techniques, and a broad overview of the techniques used here.

1.1 Motivation

EM simulation packages are used for previewing potential scenarios before they are tested empirically, to get an idea of what configurations to test and what results to expect, as well as to justify field testing. Depending on the specifics of the testing, the real field tests may even be partially or completely eliminated by judicious use of simulations, which is important when safety, legal, or financial issues make certain tests more difficult to complete.

Of course, other applications already exist to perform these simulations. Packages like AWR Analyst [1], COMSOL Multiphysics [2], and Altair's FEKO and WinProp [3] all offer options to simulate the interactions between a propagating field and occluding bodies using techniques such as Finite Element Analysis (FEA) [4] [5], Method of Moments (MoM) [6] [7], Finite Difference Time Domain (FDTD) [8] [9], Empirical Methods (EM) [10] [11], Standard Ray Tracing (SRT) [11] [12], and the Dominant Path Model (DPM) [13]. The primary problem with these implementations is performance. Using WinProp's SRT to simulate a field from a

single antenna interacting with several metal boxes in an area with a resolution of 300x150 cells can take several hours for a single simulation if ray scattering is enabled, which severely limits the number of reflections, refractions, diffractions, and rays that can be considered, resulting in a worse-than-linear time vs. accuracy tradeoff. If a simulation using a maximum of 1 reflection takes 1 minute, the same simulation allowing for 2 reflections could take an hour, and allowing for 3 reflections can increase the projected time to upwards of two weeks.

The other major problem with existing applications is not a technical issue per se, but a problem with their proprietary implementation, which results in legacy features and difficulty interacting with established standards. Once again with WinProp as an example, this means that viewing a simulation in "3D mode" is a special operation that disables most other functionality to show a limited version of the setup with difficult, non-standard camera controls. Instead of being able to import industry standard 3D model files for use as occluders in propagation simulations, the user must use the files in the proprietary database format or else a program (WallMan) separate from the simulator (PropMan) to build up their occluders piece by piece, frequently requiring the user to manually type in the 3D coordinates of model vertices one at a time for models with non-trivial geometry.

The primary goal of this application is to address these issues and present a novel simulation algorithm with better performance than existing methods, packaged in an application with a modern interface, a full 3D simulation editor, and the ability to load standard format files allowing occluder 3D models to be constructed in standard 3D modeling or CAD programs and then imported easily.

1.2 Background 1: Diffraction Issues

We propose the use of Monte Carlo Ray Tracing (MCRT) [14] as a fast alternative to existing wave-equation solving (FEA, MoM, FDTD) or analytical ray evaluating (EM, SRT, DPM) algorithms. See section 1.1 or 1.5 for details. This allows for very rapid visualization, a favorable time vs. accuracy tradeoff, and easy adjustment if more accuracy is needed after a simulation has completed as additional ray contributions can simply be appended to the previously simulated field strength. However, this approach has some unique problems that have kept it from being used for sub-optical frequencies.

The foremost of these problems is the issue of diffraction. Diffraction is the term used to describe the bending behavior of a wave as it encounters an obstacle, slit, or corner. The wave will be partially incident on the obstacle and either reflected or transmitted, and partially unimpeded as it flows past. The areas of the wavefront that move past the obstacle will spread out after passing the obstacle, resulting in a soft conical "shadow" trailing off from the obstacle. If the obstacle is small enough relative to the wavelength it can appear as if the wave almost ignored the obstacle entirely. Inversely, if the wavelength is very small relative to the obstacle the amount of diffraction will be smaller and we will instead see a sharper shadow. Figure 1 shows these effects in a simulated ripple tank [15].

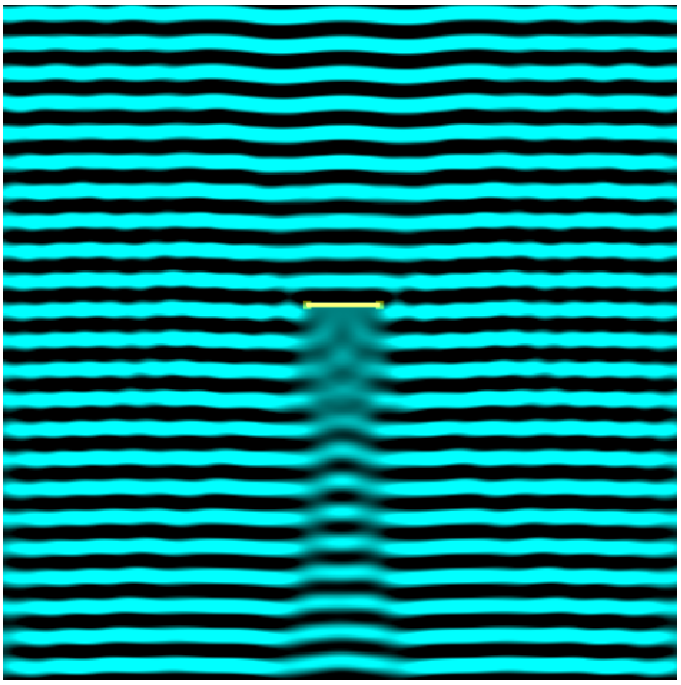

Figure 1: Simulated ripple tank diffraction showing a plane wave incident on an obstacle, including the effects of diffraction. Source: generated using P. Falstad's Ripple Tank applet.

The diffraction of light is a result of the fact that light waves satisfy the wave equation, but it can be elegantly described without resorting to the difficult wave equation solutions by the Huygens-Fresnel Principle [16], which states that a wavefront may be reinterpreted as the sum of infinitely many spherical waves emanating from the points on the surface of the wavefront. In this way, diffraction can be viewed as the sum of those spherical wavelets that were on the surface of the portions of the wavefront that went past the obstacle without interacting with it. Because these wavelets are spherical, they will continue in all directions, including behind the obstacle and past it, resulting in the appearance of the wave bending around the obstacle. Because these wavelets are evenly distributed over the surface of the preceding wavefront, they will tend to cancel out in all but the forwards direction, as the distance between them implies a phase difference that will cause points separated by a half wavelength to destructively interfere.

Therefore, diffraction can be seen as coming primarily from the parts of the wavefront that just narrowly miss the obstacle, relative to the light's wavelength. At optical frequencies, the wavelength is so short (< 800 nm) that the effects of diffraction are completely negligible for simulations on a scale of meters. This is not so at radio frequencies, and diffraction effects must be taken into account. Simulating diffraction using traditional stochastic ray tracing approaches is impossible, as rays are only tested for when they hit an object and cannot tell how close they came if they missed.

1.3 Background 2: Monte Carlo (Stochastic) Ray Tracing

Monte Carlo Ray Tracing (MCRT), also known as stochastic ray tracing, is an algorithm for approximating a solution to the light transport integral equation [17]. The basic idea is to

approximate a continuous radiative field using a finite number of rays from the field's source, which are allowed to recursively reflect off of and refract through physically based materials using optical laws and probability distributions [18]. In a typical implementation, the goal is to determine the color and intensity of light incident on a virtual camera lens or eye, which is displayed to the user as a single image on a screen, showing what the camera or eye would see. In order to do this, the individual contributions of many rays from the eye out into the scene are averaged per pixel in a process known as Monte Carlo integration, which essentially uses the law of large numbers and the definition of integration as an infinite sum of infinitesimal quantities to approximate the solution to the light transport integral. Figure 2 shows the use of MCRT for rendering realistic 3D scenes with complicated materials.

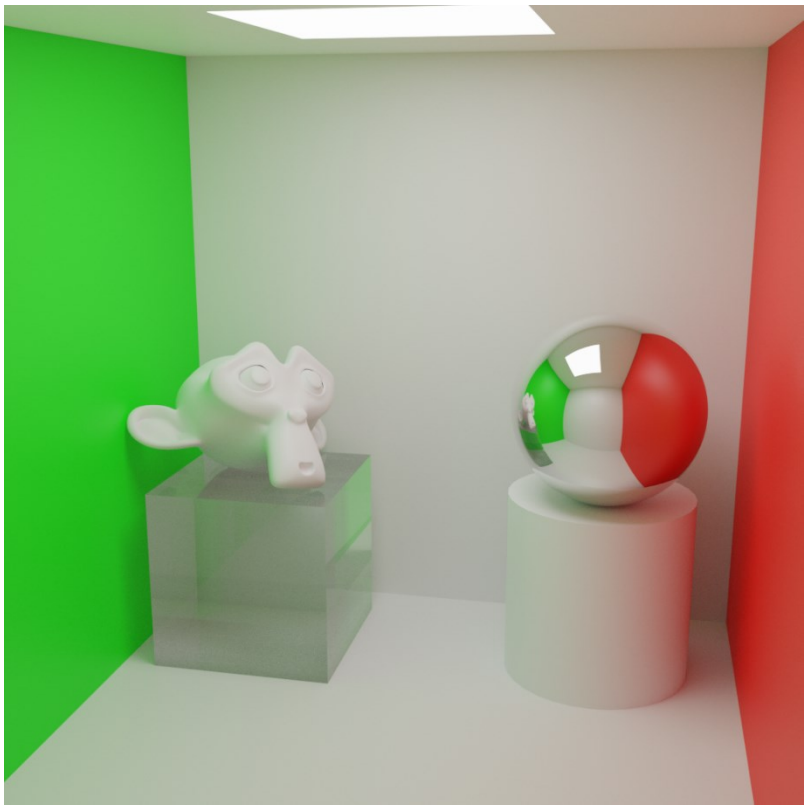

Figure 2: Light transport solved using stochastic ray tracing in Blender.

In our case, we are not trying to simulate a camera's view of the EM field, but the field itself. This means we need to divide a volume into a regular grid and use the grid cells, or voxels, as the bins for sampling instead of pixels, but this does not otherwise change much about the approach. Notably, we rely on the same the ray-triangle intersection test method as standard MCRT of determining whether a ray will hit an object. Our method will include a 3D viewport to allow the user to visualize the scene, which will be rendered for display on traditionally on the GPU. Because all 3D models must be composed of triangles to be rendered by the GPU, our MCRT implementation need only work with general arrangements of triangles and other geometrical forms may be ignored. The ray-triangle intersection test is applied to each ray-triangle pair to determine the closest object, if any, hit by each ray. The algorithm can then use the angle of incidence and the geometric and material properties of the object to trace additional reflected and refracted rays recursively [14].

One problem with MCRT is the computational cost of performing the ray-triangle intersection test. To reduce this cost, we must reduce the number of tests we perform. The most common approach to achieve this is the use of an acceleration structure – an organizational technique that allows large groups of triangles to be discounted immediately as being impossible to hit, leaving only a few triangles that still need testing. Different approaches exist, such as KD-trees [19] (of which octrees are a special case), Bounded Volume Hierarchies [20] (BVHs), and uniform grids [21]. An analysis of these techniques side by side can be found in [22]. We will be using the uniform grid approach because we are already simulating the field strength over cells of a volume, which means the costly grid traversal must be performed regardless, and so the normally most expensive part of using the grid structure does not contribute anything additional to the computation time. However, the grid must still be built, which amounts to determining the

cells that each triangle intersects so that only rays that enter that cell need to be tested for intersection with the triangle. This operation is typically expensive and runs in cubic time with respect to overall resolution for a 3D grid, ([23] alternatively states this as constructing the grid in linear time with respect to number of cells.) but once the structure is built it does not need to be rebuilt unless the contents change.

1.4 Problem Statement and Novel Contributions

There are three primary problems facing this project, which will be covered in more detail including their ultimate solutions in the following three chapters. The first of these is the acceleration of the ray tracing algorithm. As previously noted, we have chosen to use the grid acceleration structure to reduce the number of ray-triangle intersection tests, which have been shown to represent the major bottleneck in ray tracing applications. However, building the grid involves finding the list of triangles that intersect a voxel, for every voxel in the grid. A naïve approach limits the search space to a bounding volume around the triangle and performs a triangle-cube intersection test on every voxel in the bounding box to determine those that intersect the triangle. This amounts to a potentially expensive operation being performed with $O(N^3)$ time complexity in triangle size N (length of the longest edge, measured in number of cells and therefore relative to the grid resolution). We will need to find a way to optimize this in order to simulate fields with high resolution. We propose a novel technique for optimized grid construction which moves the majority of operations into constant or linear time by a method of planer steps. This is detailed in Chapter 2.

The second technical problem faced here is the simulation of diffraction with ray tracing. Because ray tracing relies on the ray-triangle intersection algorithm, such as the Möller-Trumbore algorithm [24] to determine whether rays hit objects, there is no way for a ray to know to bend inward toward an object that it missed to simulate part of a diffracted wave. In order to diffract, the rays must somehow be alerted when they pass close by an obstacle, even when they do not ever intersect it. Furthermore, the rays may not simply bend according to any arbitrary distribution – the bending must have a physical basis and agree with analytical results, such as the solution to the far-field single slit diffraction pattern, to be considered a solution. Our solution relies heavily on [25] [26], but we also demonstrate a novel method of diffraction margins, which enables our approach to work with general 3D scenarios as opposed to being limited to only mathematically defined simple apertures.

Finally, there is the problem of application development. Part of this project is the development of a user-friendly Graphical User Interface (GUI) application to perform these simulations. It is not enough to simply have a command-line tool that runs simulations blindly. The development of an entire application is non-trivial and can take a lot of time even for experienced teams who are dedicated to the front-end and need not concern themselves with technical work. In particular, we will need a 3D viewport to be able to see our simulation setup, as well as standard User-Interface (UI) tools for handling user control and feedback. There is no novel contribution to the field here, but it is nevertheless a requirement of the project.

1.5 Related Work

Several alternative techniques exist to simulate EM phenomena, including the previously mentioned FEA, MoM, and FDTD. Finite element analysis [4] [5] is used to approximately solve the partial differential equations (PDEs) associated with the EM field (Maxwell's equations, or the wave equation directly) over a region by cutting the space into a mesh consisting of a finite number of relatively small sub-regions or "elements". Each of these is a simple, flat (i.e. space is not locally curved), and convex region over which the PDEs can be solved analytically as boundary-value problems. Then, the results from all the individual elements can be interpolated between them to produce an approximate solution across the entire original region. FEA handles discrete material boundaries, and the variable element size allows scenarios with both large simple structures and concentrated complexity to be solved gracefully. Unfortunately, the method of splitting a region into small subregions works only when the original region was reasonably electrically small to begin with. Using FEA to model large-scale free-space propagation is not generally feasible.

Similarly, finite difference time domain methods [8] [9] solve Maxwell's PDEs by dividing a simulation volume into a pair of uniform grids of cells, an E-grid for the electric field and an H-grid for the magnetic field, staggered by half the size of a cell so that the corners of the cells in each grid lie on the centers of the cells in the other grid. Then, material properties are assigned to each cell indicating whether it is free-space, conducting metal, or some form of dielectric. Sources and initial conditions can then be added to the E and H field grids, and the field is allowed to evolve to steady state by using the PDEs directly to update the values of cells in one grid based on the mathematical curl of the other according to Faraday's and Ampère's laws. FDTD is the most likely candidate for extending the capabilities of the new application.

Because we are already producing a uniform grid of EM field values, it would be possible to insert a subregion into our grid over which to perform FDTD using the external values as the source in order to achieve greater fidelity in problematic edge cases. Otherwise, FDTD has similar problems to FEA. Because there are upper bounds on the time step and grid size to maintain stability and accuracy with the FDTD algorithm, simulating large scale propagation requires a huge amount of computer memory to hold the grids and computing time to get through enough timesteps to allow the wavefront to propagate through the entire domain and reach steady-state.

The method of moments [6] [7], also called the Boundary Element Method (BEM), works a little differently. Instead of using the PDE form of Maxwell's equations, they reformulate the problem in terms of integral equations that can be solved using the general mathematical technique known as the method of moments. At a high level, BEM once again subdivides a region into a mesh of small elements, but here we instead use Green's functions [27] to determine solutions between each source-element pair. This has a few problems of course — for one, the time complexity is quite bad, requiring numerical integration over both source and field patches for every possible pair of them. Additionally, for the Green's function approach to work properly it usually requires linear, homogenous materials, which restricts the types of simulations that it can handle. Nevertheless, MoM is considered invaluable for simulations where the surface area to volume ratio is low, or when a simulation consists of a lot of free-space, as it does not need to mesh the internal volume of a region but only its surface.

The preceding methods are general electromagnetic analysis techniques, and as such are not as well suited to the unique task of simulating field propagation as some other options that

were developed specifically to do so. Among these are the empirical models approach, standard ray tracing, and the dominant path model.

Empirical Models (EM) consider only the direct path from a transmitter to a receiver, using data collected empirically or mathematical models with strong assumptions to predict the path loss [10] [11], for example, applying a 3dB drop in signal strength for each wall through which the direct line-of-sight, or simply ignoring all obstacles and using the free-space path loss directly. Empirical models are the fastest and simplest option, but they typically work well only within a narrow band of frequencies and there is no way to improve their accuracy with additional compute time.

Standard Ray Tracing (SRT) [12] [11] has a some similarities with MCRT (the method employed in this project). It starts by generating rays, each a triangular pyramid oriented with its tip on the transmitter location and its bottom face being one of the faces of a regular icosahedron centered on the transmitter. The rays can then be traced out, using intersection tests to determine where they hit occluders and be recursively reflected, refracted, and diffracted based on the material properties of the hit occluders, the laws of geometrical and physical optics, and also the Uniform Theory of Diffraction (UTD) [28] for wedges when hitting an occluder near its edge. After all rays have been traced to an appropriate depth, those whose pyramidal base hit the receiver can be adjusted for accuracy based on Fermat's principle of least time and the imaging method. Unfortunately, standard ray tracing can be very slow and has poor time complexity when used to simulate an entire field, as every point needs to be simulated independently and computations cannot generally be reused between adjacent positions in the simulation volume. This means that most of the computation time used to calculate each point is wasted tracing rays down dead ends. To simulate a 1000 by 1000 grid of receiver points with a single transmitter

requires 20 million rays to be traced, only a few of which will ever make it to the receiver for each point.

Finally, we have the Dominant Path Model (DPM) [13]. This works very similarly to standard ray tracing but preemptively limits the rays to be traced to just the one which is likely to contribute the most to the final received field power. It does so by evaluating a heuristic function for each path that eventually reaches the receiver. This heuristic considers distance, frequency, interaction losses, and waveguiding. Paths are found by considering all permutations of local points of interaction (convex corners) and checking if there is line-of-sight to the receiver at each step and recording the heuristic result if there is. Then the dominant path can be found to be the path with the lowest heuristic path-loss. This reduces the number of traced rays, but it also adds the additional calculation of the heuristic function, which must be calculated for each path to the receiver. The number of such paths will be proportional to the factorial of the number of convex corners, which is problematic. The main advantage in using DPM is actually increased accuracy. Surprisingly, typical vector building databases used to test simulation software on urban scenarios contain many inaccuracies, and DPM works to smooth these out by ignoring paths that should have too much path loss – even if, due to errors in the database, these paths actually contribute far too much to the received power [13].

1.6 Overview

The goal of this project is to use a modified form of Monte Carlo ray tracing to simulate the received RF power at every cell in a volume containing arbitrary setups of transmitting antennas and occluding obstacles. The two novel aspects to this are the use of an efficient

uniform grid construction strategy for accelerating ray tracing (Chapter 2) and the inclusion of diffraction margins around the edges of occluders to allow for realistic diffraction effects within a stochastic ray tracer (Chapter 3). We will then move on to an explanation of some key features of the application development and user interface (Chapter 4) which enables the use of this technology in practice, before finishing with a consideration of the desired features that have yet to be implemented and potential further refinements to the techniques as future work and some concluding statements (Chapter 5).

Chapter 2: Optimized Grid Construction and Traversal

In order to improve the simulator's performance to an acceptable level, we need to use an acceleration structure to minimize the use of the ray-triangle intersection formula by preemptively rejecting triangles that cannot possibly be hit. We will use the grid acceleration structure [21], which allows rays to test for intersection only with triangles that are in the same voxel as them. The cost of using an acceleration structure is the overhead involved in constructing it, which mut be done whenever the geometry (triangles to be simulated) changes, and the traversal, which is simply the operations that must be performed to step the rays through the structure. In this chapter we will consider the grid acceleration structure and an optimized method of constructing the grid, as well as an implementation detail that will speed up our ray tracing later and a discussion of the grid traversal algorithm.

2.1 Overview of the Technique

The grid acceleration structure is probably the easiest to understand of all the ray tracing acceleration structures and can be summed up as, "Test ray-triangle collisions only against triangles that intersect the grid cell that the ray is currently in". This raises a couple of questions: "How can we determine which cells the triangles intersect?" and "How can we determine which cell a ray is in and how does the ray move from cell to cell?". Possible answers to both of these

will be detailed in coming sections. Figure 3 shows an example illustrating a triangle intersecting cells.

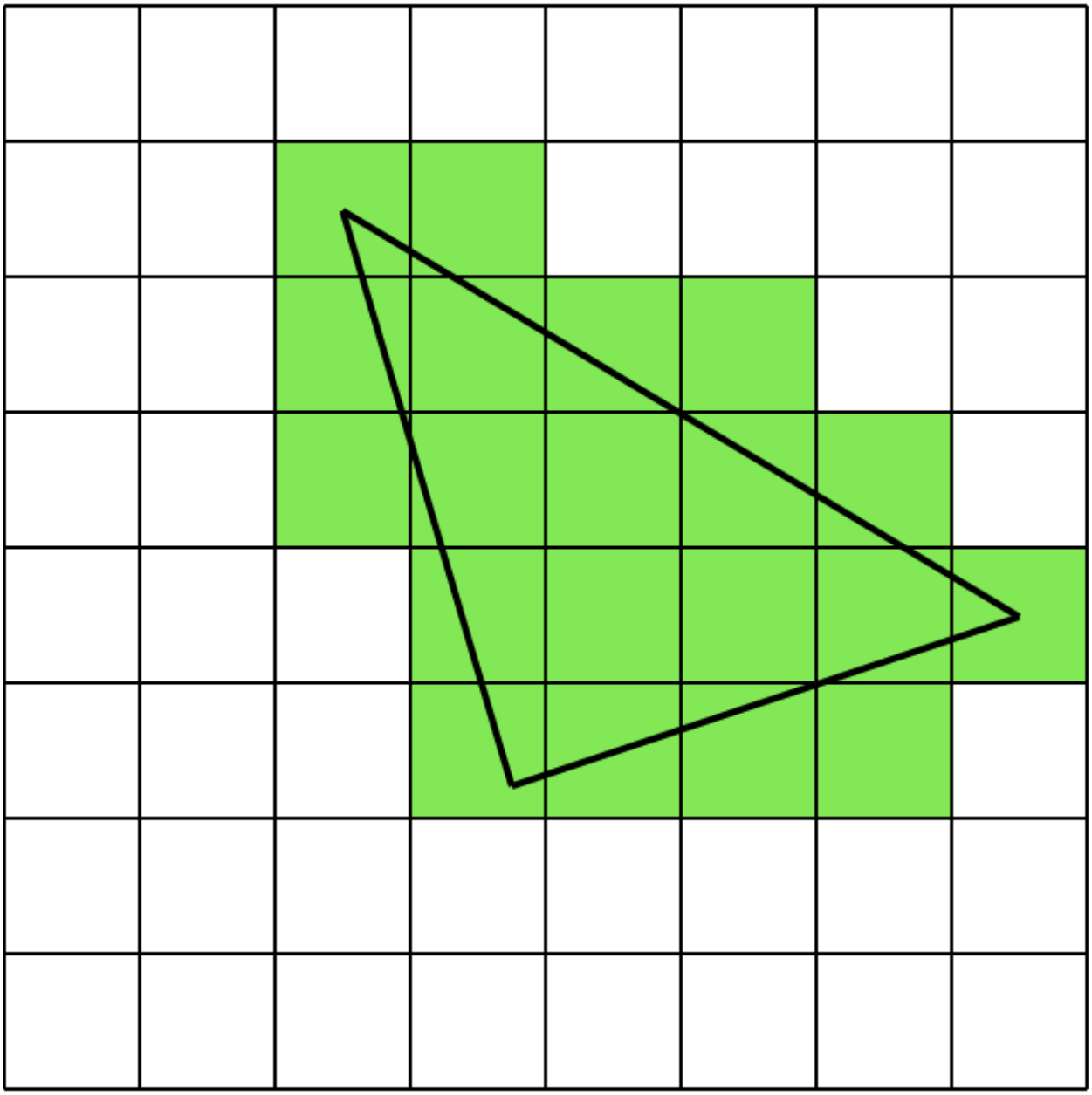

Figure 3: An illustration of the voxelization of a triangle, in this case into a 2D grid for viewing purposes.

For now, it is sufficient to understand that we will have a 3D grid of cells, each indicating the list of triangles that intersect it, and a collection of rays that step through this grid one cell at a time testing for intersection only with those triangles in that cell's list at each step. This

drastically reduces the average number of triangle intersection tests that must be performed per ray, as each ray only intersects a linear number of cells. Additionally, the smaller the cells are made to be, the more likely it is that a ray through that cell will hit a triangle that intersects the cell, asymptotically reaching a maximum of just a single intersection test as the grid resolution goes to infinity. Of course, this would also result in an infinite grid traversal time — a tradeoff must be considered to optimize the resolution. In [29] the optimal number of cells M is considered to be

$$M = \rho N, \tag{1}$$

where ρ is an empirically found density, typically between 4 and 8, and N is the number of triangles under consideration. Unfortunately, optimal grid resolution is not really a possibility for this application of the technology because it must be up to the user to determine the precise resolution that they need along all three axes to produce the desired simulation results.

2.2 Preemptive Inverse Transform

The ray tracing algorithm naturally depends on the precise orientation of the simulation volume it is to take place in. If this volume is translated, dilated (including non-uniformly), or rotated it will result in different relative positions for all of the antennas and occluders in the volume, and therefore also all of the rays emitted from these antennas and the triangles making up these occluders.

In order to accurately reflect these transformations to the simulation volume, the transforms need to be applied in the ray tracer in several different places, and this becomes quite expensive and results in brittle code that is more difficult to maintain. To ameliorate these issues,

we can find the inverse of the transform matrix used to transform the simulation bounds and apply this matrix to each triangle and ray as their data is being generated. Then the simulation can be assumed to happen in a unit cube from (0, 0, 0) to (1, 1, 1), and all of the transformation processing that happened inside of the ray tracer may be removed, as the translation is now (0, 0, 0) and adding vector zero has no effect, and the scale and rotation are the identity matrix, and multiplication by the identity matrix also has no effect and can be safely removed.

2.3 Optimizing Grid Construction

The construction of the grid acceleration structure requires that a 3D array of lists of triangles be populated such that each cell has a list of all triangles that intersect it. Because the ray-triangle intersection test is still performed, false positive errors in the population of these lists are tolerable and will not affect the simulation other than slightly slowing it down, however they are still to be avoided if possible.

There are a few fairly simple optimizations that can be added to save on computer memory for the grid. First, storing triangle data in the lists for every cell is a pointless duplication of data — we need only store triangle pointers into a separate geometry buffer that can hold the actual data. Second, the grid is likely to be sparse, with many empty cells representing free-space — allocating a triangle pointer list for them is unnecessary. We can instead allocate a grid of index values that indicate which triangle pointers in a triangle buffer are associated with each cell. They do this by specifying simultaneously the index into the buffer of the first triangle in the associated cell, as well as the index of the one-after-last triangle of the preceding cell. This means that empty cells are indicated by the index value for those cells being

the same as the index value of the next cell, and the difference between subsequent cell indices gives the number of triangles in the preceding cell, and looping from index[cell] to index[cell+1] always gives all of the triangle pointers associated with that cell. In total, we need to allocate a geometry buffer of triangles, a triangle buffer of pointers to positions in the geometry buffer, and an index buffer of integers to associate the triangle pointers with cells.

This may seem overcomplicated, but it enables us to have a jagged array — otherwise, every cell would need an array of triangle pointers with length equal to the largest number of triangles present among all cells, or else a hard limit on the number of triangles per cell would need to be enforced resulting in decreased simulation accuracy. Without a heuristic to determine which triangles to leave out, there is no way to guarantee that putting a limit on the maximum number of triangles per cell would not throw out large important triangles, resulting in holes in the geometry during simulation.

Concerning the actual population of these structures, there are three main approaches. In the simplest and most naïve approach, for each triangle a box-triangle intersection test such as [30] is applied to every cell within a bounding-volume around a triangle, for example, for each of the x, y, and z components, looping over every cell position from the floor of the least value among the triangle vertices to the ceiling of the greatest value among triangle vertices, and applying the triangle-in-box test. This test can be performed either against the actual cell at the correct location or using a more static unit-box test and pre-transforming the triangle coordinates. While simple and direct, this method has an obvious flaw in that it uses a cubic time-complexity algorithm to solve a problem that seems to be inherently 2D. Intuitively a quadratic solution should exist as the triangle consists of a 2D area, which only intersects a number of cells proportional to its size squared.

This intuition would be correct, as there is indeed an algorithm capable of voxelizing a triangle in $O(S^2)$ time. The Digital Differential Analyzer (DDA) [31] essentially determines the axis-oriented slopes of the triangle and rasterizes it using a 3D equivalent of the scanline algorithm in the plane perpendicular to the axis of least slope. That is, by starting at one vertex of the triangle the DDA can use the precalculated slopes to step from one cell to the next along the line segment from the starting position to one of the remaining vertices. Then, it slides the line segment towards the remaining vertex and repeats the scan. The DDA method has both good time complexity and a good practical implementation, but it is very difficult to get the line segment sliding step right and the results are not generally watertight. Variants of the approach exist [32] that use a GPU running a standard pipeline rendering pass and take advantage of the heavily optimized GPU rasterization process to perform the voxelization all at once in the fragment shader, using the pixel position and the depth pass information to populate the grid. This runs into the problem of triangles that only just touch a fragment and don't cover the center not being counted, as the rasterizer ignores pixels whose center lies outside of the triangle.

The technique used here is neither of these. Instead, we have developed a novel solution referred to as the "planar steps method" involving the use of a box-in-triangle test that is more efficient by far than the previous triangle-in-box test and allows a high degree of information reuse. The method starts at the triangle creation level. For the purposes of ray tracing, it is beneficial to store unit vectors normal to the plane of each triangle with the triangle vertex data, as these normals are needed for computing reflections and refractions and their computation is much cheaper if it is frontloaded rather than deferred to when the values are needed during a ray collision. Then, the plane equation for the triangle can be determined. The plane equation is given as

$$N_x(x - A_x) + N_y(y - A_y) + N_z(z - A_z) = 0,$$

$$\Rightarrow N_x x + N_y y + N_z z = N \cdot A,$$

$$\Rightarrow N \cdot (x, y, z) = N \cdot A, \tag{2}$$

for normal vector *N* and a point on the plane *A* simply states that any point *(x, y, z)* whose dot product with the plane normal is equal to a constant offset is on the plane. To calculate this offset, we can find the dot product of *A* with the normal. In practice we may use any of the triangle vertices for *A*, as these vertices are points on the plane by definition.

To determine the voxels in which a triangle lies using the box-in-triangle test, we must first compute the distance from the box's center to the triangle plane. Fortunately, this is exactly what the plane equation gives. To find the distance from a box's center to the plane, we can simply take the dot product of the center point with the plane normal and add the offset, remembering that this will give us a negative distance value for points behind the plane. This is

$$D = N \cdot C + N \cdot A, \tag{3}$$

where *D* is the distance and *C* is the box center. Then, we can determine the greatest extent of the box toward the plane, D_{max}, by taking the dot product of the normal with the vector from the box's center to the corner most aligned with the normal, which in practice means taking the sum of the absolute values of the products of like terms, rather than just the sum of the products of like terms as in a normal dot product. This is

$$D_{max} = |N_x B_x| + |N_y B_y| + |N_z B_z| + \epsilon_t, \tag{4}$$

where B is the vector from the center of a box to one of its corners. Here we also add an additional "triangle thickness" epsilon term to ensure that floating point errors do not result in a triangle on an edge being rejected by voxels on both sides.

Now that we have the maximum extent of any box towards the triangle, we could loop over all cells in the bounding box around the triangle and use the plane equation with the cell's center and compare the absolute value of the distance to the maximum to determine if the cell is too close to the triangle to avoid hitting it. There is a more efficient way, however. Because the cell-plane distance is just the dot product between the normal and the cell center, and the cells are uniformly arranged, we can actually precompute the independent x, y, and z components of the dot products for every cell separately — in linear time. For instance, we can see from the expansion of the dot product in (3) that

$$D = N_x C_x + N_y C_y + N_z C_z + N \cdot A, \tag{5}$$

where $N \cdot A$ is a constant (the plane offset) that can be computed once and stored with the rest of the triangle data, and the x, y, and z components of the distance are independent. We can optimize this further by noting that because the grid is regular, the quantity

$$S = C_{a+1} - C_a, \tag{6}$$

that is, the distance between subsequent box centers along axis a for $a = x$, $a = y$, *and* $a = z$, is a constant. This quantity is simply the width of a box along the respective axis, so we can do some rearranging to get

$$D = D_x + D_y + D_z + N \cdot A,$$

$$D_a = N_a C_a,$$

$$D_{a+1} = N_a C_{a+1} = N_a(C_a + S) = N_a C_a + N_a S, \quad (7)$$

where the +1 indicates the next cell along an axis, showing that we can use mathematical induction to find the contribution of the distance to the plane along each axis by adding a constant to the previous distance. Then, we find the distance to the plane of the center of just the first box in the bounds that enclose the triangle and use three loops over the x, y, and z axes to generate the distance contributions up to the last box in the bounds and store all the computed values, allowing us to compute the components of the distance to the plane of every cell in a 3D grid in linear time and space.

This novel technique is very powerful, as we can now compute the distances of every cell in a 1000x1000x1000 grid using 3000 operations, instead of 1 billion as would be required by a simple iterative test and we do not have the accuracy issues that rasterization and DDA approaches suffer from. We can then proceed to perform the cubic time iteration over all cells in the triangle bounds, computing the final distance with just two additions, one of which can be precomputed to move it into quadratic time. Thus, while the algorithm is still, technically speaking, $O(N^3)$, the cubic portion consists only of a single addition and one or two comparisons to check if the absolute value of the distance is close enough for the cell to hit the plane. Additionally, instead of performing all three loops over the x, y, and z axes we can instead remove the innermost loop by calculating the cell indices along this axis which will intersect the plane based on the distances of the other two axes, making the algorithm truly quadratic. On top of this, the overwhelming majority of the algorithm can be moved into constant or linear time, potentially outperforming even the other quadratic methods without sacrificing accuracy as they do.

Finally, while the ray tracer would work fine with every cell lying in the plane of the triangle being considered to contain the triangle, the use of diffraction margins requires that, along with the normals, we must also pre-calculate edge-normal plane-tangent vectors for each edge of the triangle. These are unit length vectors lying in the plane of the triangle and perpendicular to each of the edges. Figure 4 illustrates these vectors on a triangle in 3D.

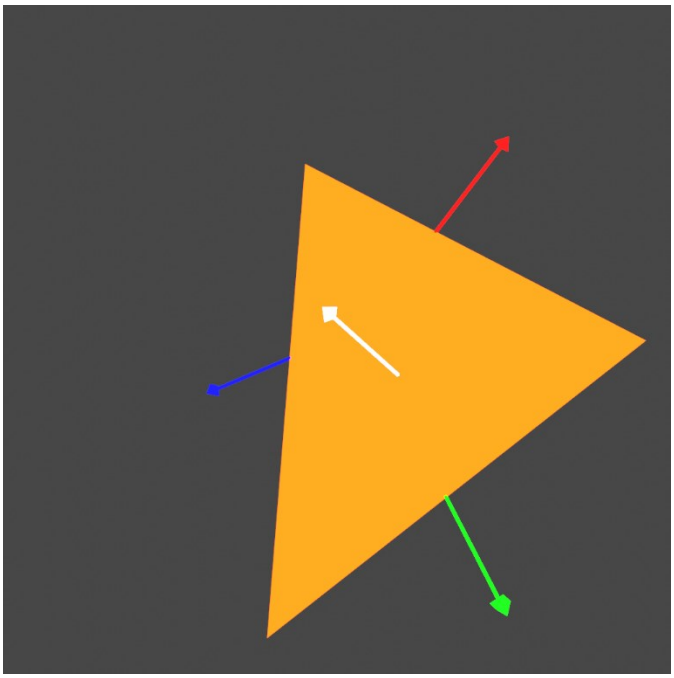

Figure 4: A visual representation of the edge-normal plane-tangent vectors (red, green, blue), as well as the normal vector (white), of a triangle.

By adding calculations for the components of distances from the cell centers to planes defined by these vectors, in addition to the triangle plane, we can further refine the test to exclude those voxels that do lie in the plane of the triangle but are entirely on the wrong side of one of the half-spaces that define the triangle's edges. This results in an algorithm that is perfectly accurate and does not produce any false positives for only a little extra up-front computation, which will save a lot of time for the ray tracer.

2.4 Grid Traversal

When the simulation is started, all of the grid information including the geometry buffer, triangle buffer, index buffer, and the empty field buffer to be populated, as well as a buffer containing the rays to trace and some uniform parameters, are initialized and transferred into the GPU VRAM. Once there, each invocation of the ray tracing kernel uses its global ID to load the ray data for that invocation from the ray buffer so that it can begin tracing the ray.

The algorithm to step through the cells of a uniform grid is known as the digital differential analyzer, which was already discussed briefly in the context of its use in a triangle voxelization algorithm. Here, we begin by determining the cell in which the ray starts. This will be nothing more than the component-wise floor function of the component-wise product of the ray origin o and the resolution along each axis as a vector R.

$$initial\ cell\ = \lfloor o \odot R \rfloor = \left(\lfloor o_x R_x \rfloor, \lfloor o_y R_y \rfloor, \lfloor o_z R_z \rfloor\right) \tag{8}$$

Where the circled-dot operator indicates component-wise multiplication. This is a very simple calculation because, thanks to the preemptive inverse transform, we are working in a simulation bounded by the unit cube from the origin, regardless of what the actual simulation bounds are doing. Once we have the starting cell, we can find the directions the ray will be stepping in, either positive or negative for each of the three dimensions based on the sign of the ray components. The step period is found as the absolute value of the inverse of the resolution multiplied by the ray direction d, component wise;

$$T_{step} = \frac{1}{|d \odot R|} = \left(\frac{1}{|d_x R_x|}, \frac{1}{|d_y R_y|}, \frac{1}{|d_z R_z|}\right). \tag{9}$$

This is the amount of time it takes for the ray to go from one cell threshold to the next in a particular direction. The vector quantity time-to-next-threshold is initialized to be the step period multiplied by the current distance from the ray origin to the edges of the current cell, in the direction of ray travel. For a ray traveling in the positive x, y, and z directions this is

$$T_{next} = T_{step} \odot (o \odot R - initial\ cell). \tag{10}$$

Otherwise, for any component of the ray direction that is negative one can simply take one minus the gap instead. The time is initialized to zero, and the actual traversal can finally begin.

The grid traversal algorithm occurs in an infinite loop, which will be broken when the ray exits the simulation bounds (or another event causes the ray to exit early). Because the ray steps through only the cells intersecting a line, it will always exit the loop in linear time with respect to the grid resolution. After each step, the ray is tested for intersection with all triangles that intersect that cell, which will frequently be none in a simulation with plenty of free-space. Then we need to determine which threshold the ray will cross next. This will simply be the smallest of the components of the time-to-next-threshold variable, so we choose the index of x, y, or z depending on which threshold has the least time before it will be crossed. If we had intersected any triangle in the previous step, the time of intersection would have been returned. If that time is less than the time of crossing the closest threshold, then we need to perform the hit instead. This will be discussed in more detail in the next chapter, but we essentially determine whether the hit will be a reflection, refraction, or diffraction, update the ray origin and direction, and restart the loop with the new ray properties. This is important as recursion, the standard method of ray tracing implementation, can be problematic for performance critical applications, especially on the GPU.

If there was no hit, then we need to update the current time to the time at which we determined we would hit the next threshold, increment the time-to-next-threshold of the threshold we just passed by the step period so that it now reflects the time it will take to hit the next cell in the same direction, adjust the index of the current cell, and test to make sure that that cell index is not outside of the simulation bounds and return if it is. We also need to add the contribution of the ray power P_{added} to the cell we just left. This is done by atomically adding the ray power I multiplied by the time the ray spent in that cell (found as the time of entering the next threshold minus the current time, before the current time is updated) to the field buffer at the current cell location.

$$P_{added} = I(T_{next}[closest] - T_{current}) \tag{11}$$

All in all, this algorithm requires just two comparisons and an addition per cell to find which threshold is closest and step. Stopping when the ray exits a finite volume adds an extra addition and comparison in order to keep track of where in the grid the ray is and return if it exits the bounds. The additional requirement to write the ray power contributions to the field buffer adds another addition, an assignment, and an atomic add, because we now need to keep track of the current time and the cell index (which is not the same as the ray position – cell index is a single integer pointing to where in the field buffer to add the field strength) as well as writing the incremented field strength value to the field buffer safely without race conditions. The triangle intersection test is quite expensive, but fortunately we rarely need to execute it because most cells do not intersect any triangles, and most rays will stop after the first intersection test.

2.5 Results

We began testing the algorithm using the Stanford bunny [33] a standard 3D model used for practical testing of 3D rendering software and demonstration purposes. The model as rendered by our application inside of a simulation volume is shown in Figure 5.

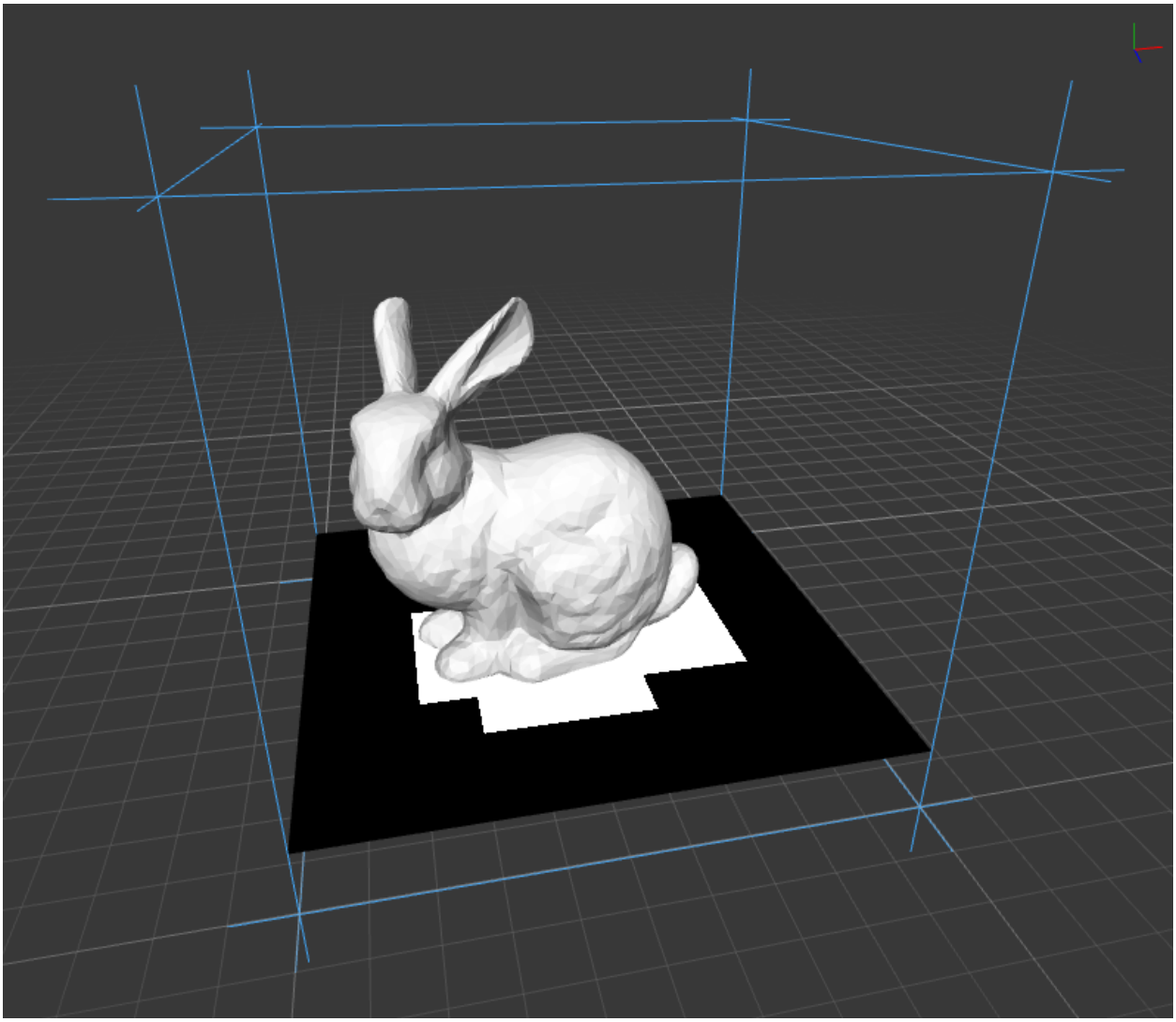

Figure 5: The Stanford bunny in a voxelization volume.

We voxelized the bunny into a 10x10x10 grid and exported the results. Figure 6 shows the vertical slices individually, marked from lowest to highest. The blank slices below and above have been removed. Here we have simply marked any cell as white if it has at least one triangle

assigned to it. An inspection of the grid data and further testing with integration into the ray tracer indicate that the approach is sound in practice as well as mathematically correct in theory.

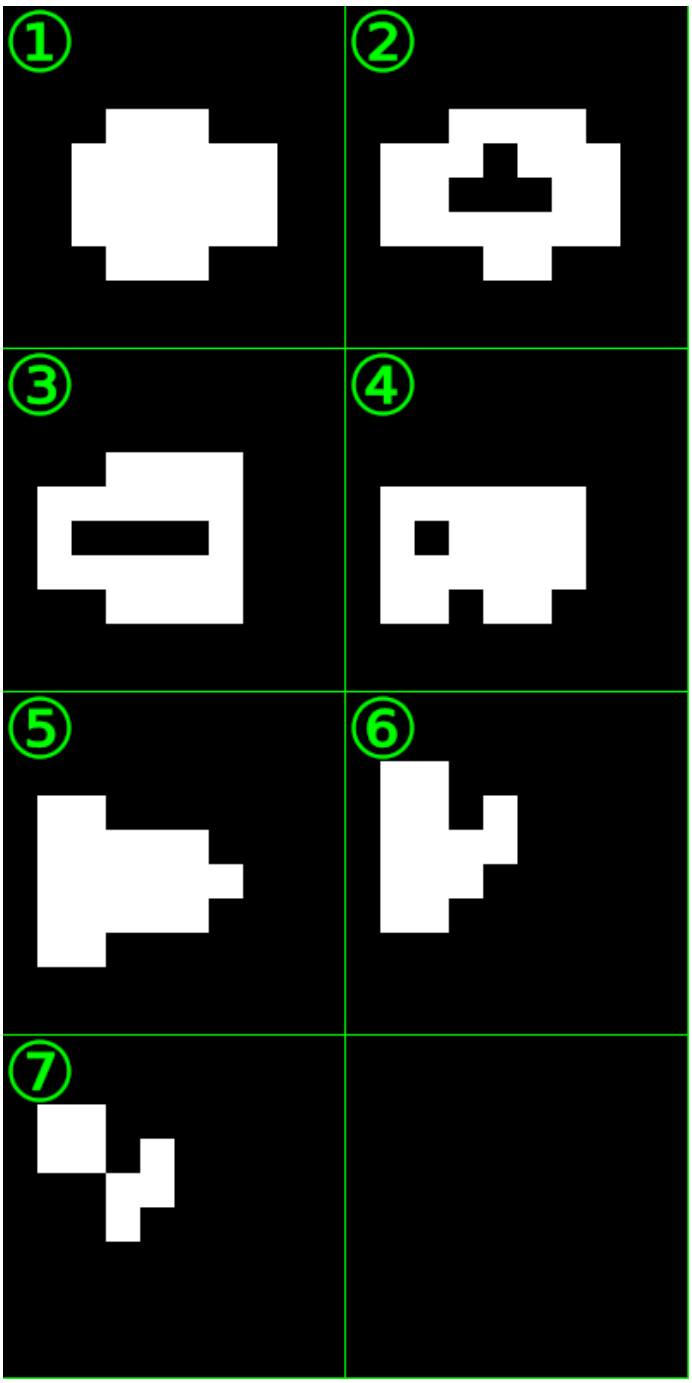

Figure 6: The voxelization of the Stanford Bunny illustrated as slices.

Next, we can begin to consider the performance of this new algorithm as it compares to the iterative triangle-in-box testing method. To evaluate it, we implemented an alternative voxelizer using the triangle-in-box method on transformed triangle coordinates to compare against. First, we evaluated the implementation and confirmed that it gave identical results to the new planar steps method. Then, we set up a cubic voxelization environment, set the resolution to 10x10x10, and voxelized the Stanford bunny. We repeated the test ten times for both the new and old voxelization methods and recorded the results before moving on to a 20x20x20 resolution, then 30x30x30, and so on, stopping at 300x300x300. The results of this testing are shown below in Figure 7.

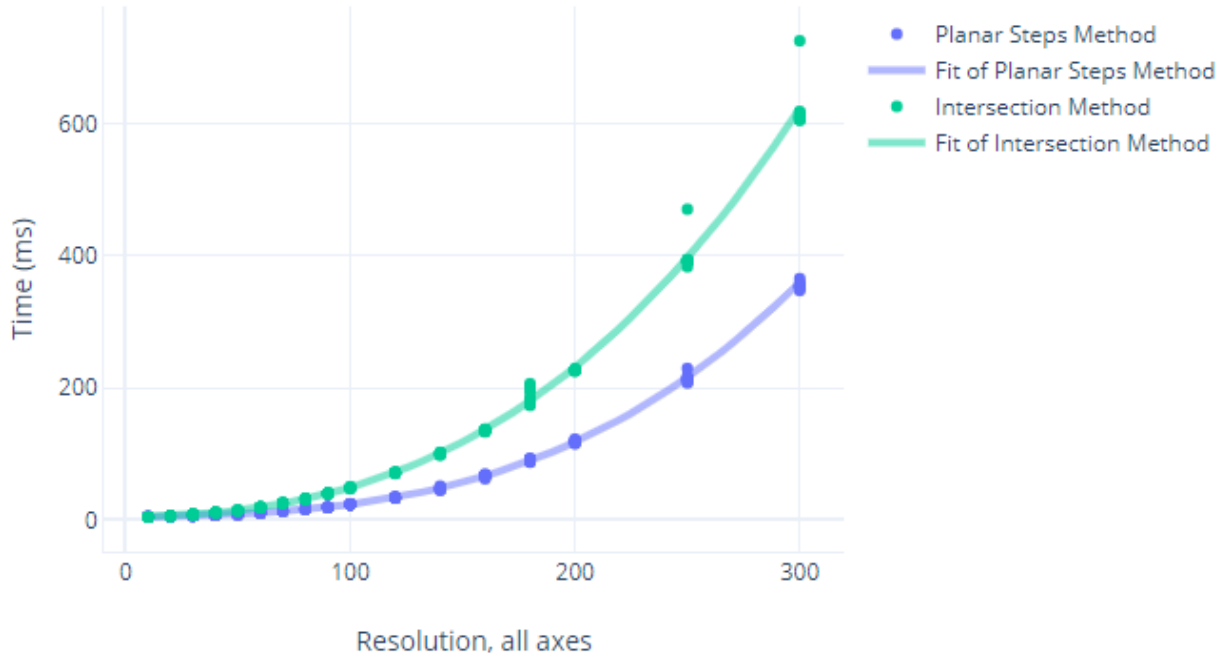

Figure 7: A comparison indicating the relative performance of the old and new voxelization strategies.

From this we can see that the planar steps method has significantly better performance than the iterative triangle-box intersection method, which should not be overly surprising. We have also included in the scatter plot a 3rd order line of best fit for each of the trials. The ratio of the coefficient on the X^2 term to the coefficient on the X^3 term for the new method is also three orders of magnitude greater than the same ratio for the old method, indicating the relatively small effect of the cubic time operations.

Chapter 3: Ray Traced Diffraction

This section will cover the process of developing a novel system for enabling diffraction within an otherwise standard Monte Carlo ray tracer. We will first consider the use of a watertight algorithm for performing the ray-triangle intersection tests to eliminate erroneously transmitted rays. Then, we will discuss the use of Heisenberg's uncertainty principle in determining the deflection angle of rays that pass by an object probabilistically, and the development of diffraction margins for determining when and by how far these close passes happen. Finally, we can test and evaluate the method for performance and quality.

3.1 Watertight Ray-Triangle Intersection

Before we can begin to develop the method of diffraction margins, we need to finish our ray tracing algorithm with a ray-triangle intersection test. There are common implementations of this test that work very well, such as the Möller-Trumbore algorithm [24]. There are also more optimized versions [34] that can offer increased performance. However, these algorithms are designed for use in an optical renderer, where allowing a ray to leak through a gap at a triangle edge or corner is inconsequential. These leaks occur due to floating point inaccuracies – a ray incident on an edge may fail the intersection test for the triangles on both sides of the edge. In our case, we cannot afford to allow rays to leak through the edges, as this can degrade the quality of the simulations.

We can resolve these leaks using a watertight ray-triangle intersection algorithm [35]. At a high level, this algorithm calculates the barycentric coordinates of the intersection point between the ray and the plane that contains the triangle. Then, if all of the barycentric coordinates have the same sign, the ray will have hit the triangle. Barycentric coordinates are a special form of basis vectors that are attached to the vertices of a polygon such that any point in the plane of the polygon may be described as the sum of the polygon vertices multiplied by their respective barycentric coordinates as coefficients. In this way, a point on top of one of the vertices has the coordinate associated with that vertex equal to one and all other coordinates zero. For a triangle, the average of the three vertices occurs when all three coordinates are equal to one-third, $\alpha = \beta = \gamma = 0.\overline{33}$. See Figure 8. [36]. A very useful property of barycentric coordinates is the fact that a point outside of the polygon will have at least one

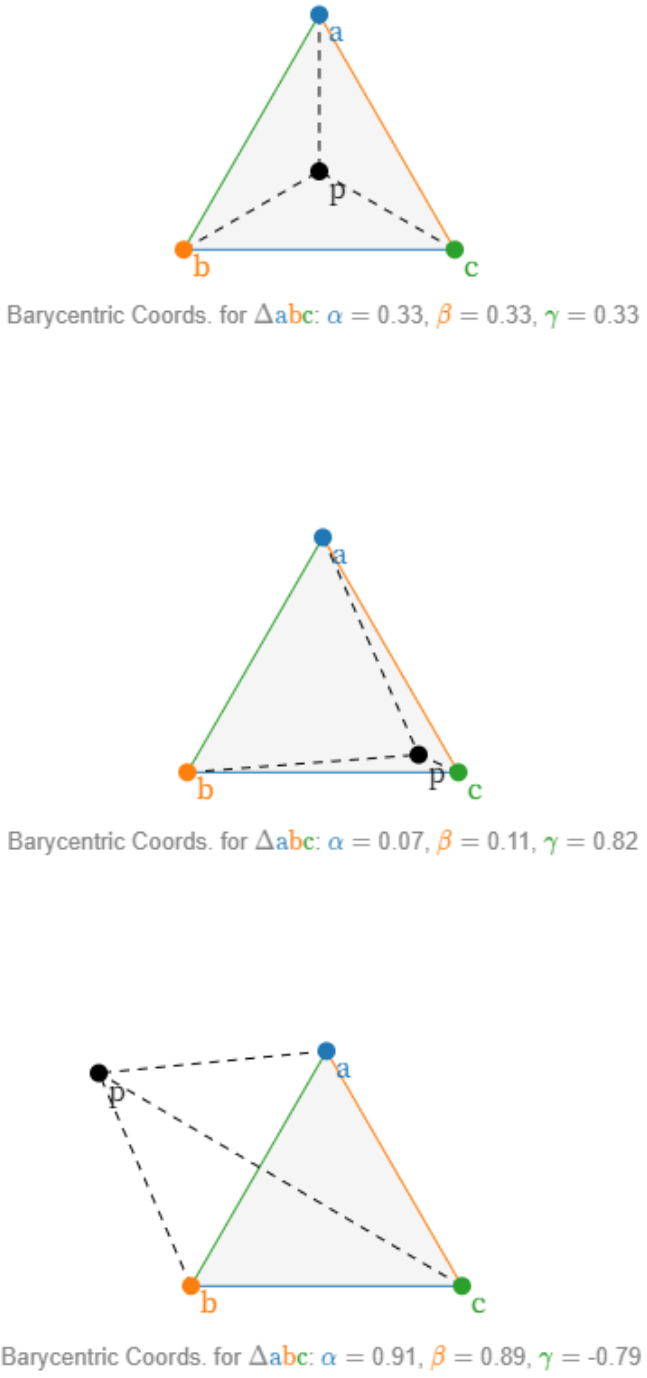

Figure 8: Three examples of barycentric coordinates of points with reference to an equilateral triangle. Source: generated using code by T.J. Jankun-Kelly.

negative-valued coordinate. This allows us to test a point to see if it is inside of a polygon simply by testing the sign of its barycentric coordinates.

The watertight intersection algorithm starts by determining which axis the ray is most aligned with to use as the local z, or forward, axis. A change of basis is applied such that whichever component had the greatest absolute value will be redefined to be the new local z axis, and the new local x and y axes are determined by the sign of the new local z component of the ray and the right-hand rule. Figure 9 illustrates this. In practice we are simply rearranging the indices of access, so instead of using ray_direction[0] for x we use ray_direction[dx], where dx might be 0, 1, or 2 depending on the magnitude of the ray direction components.

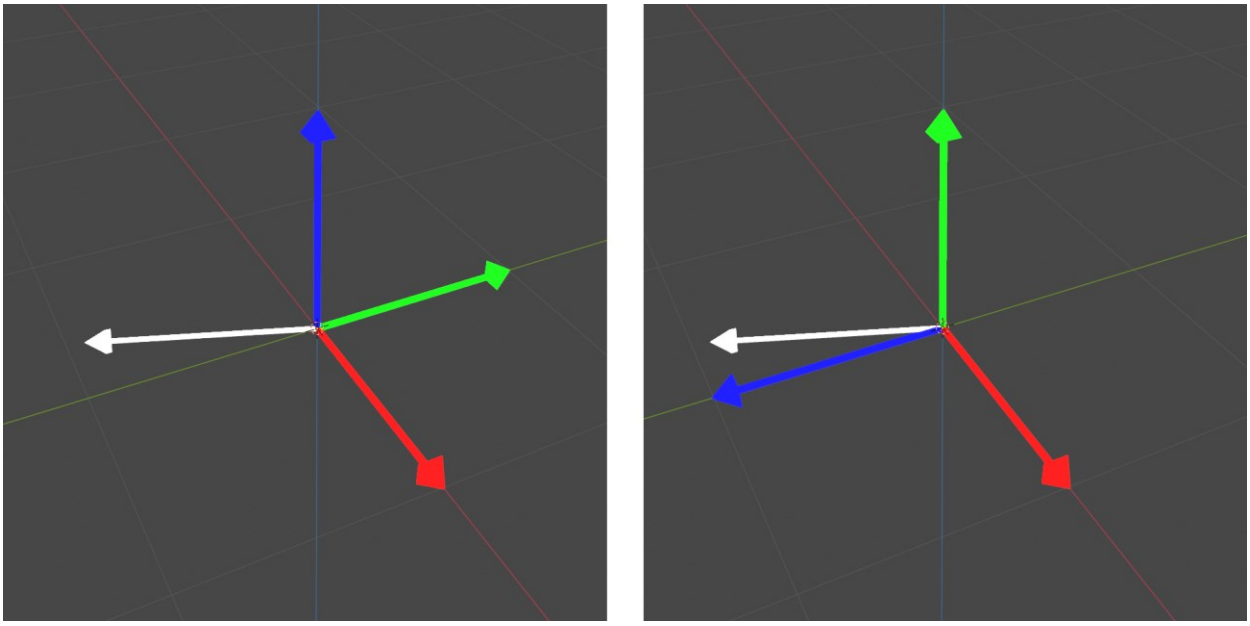

Figure 9: The basis vectors (x red, y green, z blue) used to define the ray direction (white). Left: the world basis vectors. Right: the new local basis vectors.

Now, we can compute shear constants for each of the local axes.

$$S_x = \frac{d_x^{local}}{d_z^{local}}, \qquad S_y = \frac{d_y^{local}}{d_z^{local}}, \qquad S_z = \frac{1}{d_z^{local}} \tag{12}$$

These constants represent the amount by which the ray deviates from perfect alignment from its new local z axis and are used to transform the triangle vertices by shearing them along the local z axis. This transformation of the x, y, and z coordinates

$$x = (x - o_x) - S_x(z - o_z), \qquad y = \left(y - o_y\right) - S_y(z - o_z), \qquad z = z - o_z, \tag{13}$$

applied to each of the triangles vertices is also exactly the transform that would be required to straighten out the ray to lie perfectly in line with the local z axis, which means we can now find the barycentric coordinates of the ray as it intersects the triangle plane by finding the barycentric coordinates of the origin with respect to the transformed triangle in 2D, which can be solved using Cramer's rule and results in the following:

$$U = C_x B_y - C_y B_x, \qquad V = A_x C_y - A_y C_x, \qquad W = B_x A_y - B_y A_x. \tag{14}$$

Where *A*, *B*, and *C* are the triangle vertices and *U*, *V*, and *W* are the barycentric coordinates associated with them, respectively.

In order to avoid leaking rays, we have to add an additional edge case handler here. If any of the barycentric coordinates are zero, the previous calculation must be redone using double precision arithmetic. Afterwards, we can finally test the ray for intersection by returning if the sign of all three barycentric coordinates are not the same. Typically, a point is outside of a triangle if any of the barycentric coordinates of the point are negative. Here, we must account for the case that the triangle may be backwards, resulting in all negative barycentric coordinates in the case of a hit. Finally, the hit time can be returned by taking the sum of the transformed local z components of the triangle vertices, weighted by their respective barycentric coordinates, that is,

$$T_{hit} = U\, A_z + V\, B_z + W\, C_z. \tag{15}$$

3.2 Application of Heisenberg's Uncertainty Principle

In order to ensure realistic diffraction in our simulator, we need a method for determining how to bend diffracted rays to produce the correct results. Clearly, this should be a function of distance from the object that the ray is diffracting around, as diffraction effects are most prominent around the edges of an occluder. To approach an analytical answer, we can consider Heisenberg's uncertainty principle, as was done in [26] and [25]. The principle states that the product of the uncertainty in the position Δx and the uncertainty in the momentum Δp of any particle has a lower bound

$$\Delta x \Delta p \geq \frac{\hbar}{2}, \tag{16}$$

where ħ is the reduced Planck constant, and therefore, that any apparatus that can offer more certainty in either the position or the momentum must force less certainty in the other at least to keep this lower bound on their product. A restatement of the principle gives a formula for the minimum uncertainty in momentum relative to the uncertainty in position.

$$\Delta p \geq \frac{\hbar}{2\Delta x} \tag{17}$$

We can use this to analytically determine the deflection of a diffracted ray. For the purposes of this discussion, we assume an imaginary "photon bundle" to be associated with each ray. Start from the assumption that we do not know the precise position of any ray, only its photon bundle's momentum, (the reduced Planck constant and the wavenumber *k* times the ray direction vector *d,*) and that the stored variable indicating the position of the ray origin *o* is only a guess with infinite uncertainty. For a ray that comes within a distance *x* of an occluder but does not hit it, because we know that the ray did not hit the occluder, we have a limit on how close it

could have passed and therefore new information on the ray's actual position. The ray must have passed by at a distance somewhere between 0 and x, putting a finite value to our uncertainty of $\Delta x = x$. We can use the principle to determine the uncertainty in momentum that must accompany this loss of uncertainty in position, and use basic trigonometry to find the angle of the most possibly deviated ray based on the old momentum vector and new vector uncertainty in momentum.

$$\Delta p = \frac{\hbar}{2x} \tag{18}$$

$$p_{old} = \hbar k, \qquad \left(k = \frac{2\pi f}{c} d\right)$$

$$\sigma = \operatorname{atan}\left(\frac{\Delta p}{|p_{old}|}\right) \tag{19}$$

Finally, we can interpret this angle as the standard deviation of a normal distribution and draw a sample from it to determine the actual deflection angle.

$$\theta_{deflection} \sim N(0, \sigma) \tag{20}$$

In order to sample from a normal distribution on the GPU, we need to implement some random number generator functions. Plenty of implementations of good random integer generators exist for example [37], and these can be scaled by the inverse of the max integer value to compute a uniform distribution, but acquiring a normal distribution is a little harder. We will use a technique known as inverse transform sampling [38], which uses a table of the values to which each of N uniformly spaced values from the desired distribution's CDF map. This allows us to sample a value from any arbitrary distribution by using a uniformly distributed value to choose a point on the inverse of the desired CDF.

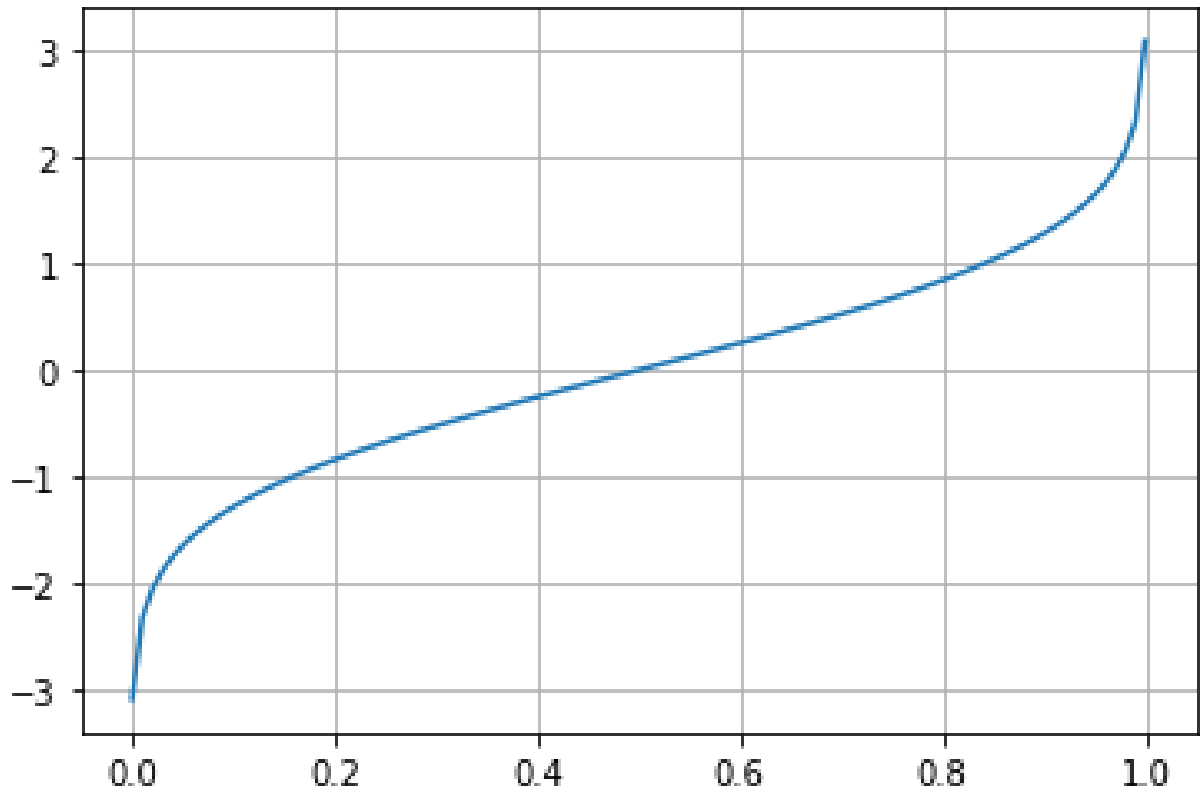

Figure 10: The inverse of the standard normal distribution's CDF.

Figure 10 shows *f(x)* = the inverse of the standard normal distribution's CDF on its domain (the region from 0 to 1 inclusive). By using a uniformly distributed x in this region we are able to draw normally distributed numbers from the function, which can be seen by noting that the probability of obtaining a value within a range is inversely proportional to the slope of the function over that range. We will need nonstandard normal distributions for our purposes, as the standard deviation depends on the miss distance. Fortunately, determining a normal distribution with a given standard deviation is quite easy, and requires only scaling the samples by the ratio of the standard deviations before and after. By choosing a standard normal distribution as the base this comes down to a simple multiplication of all samples by the new standard deviation.

3.3 Diffraction Margins

In order to actually use the preceding information to diffract our rays, we need an accurate measure of the distance from a ray to an occluder that it passed but did not hit. Because we are relying on the ray-triangle intersection test to determine all interactions between rays and occluders, it is actually impossible to detect when a ray passes close by to an occluder without hitting it. The way around this is to ensure that any ray that would be noticeably diffracted actually does hit a part of the occluder — by using diffraction margins.

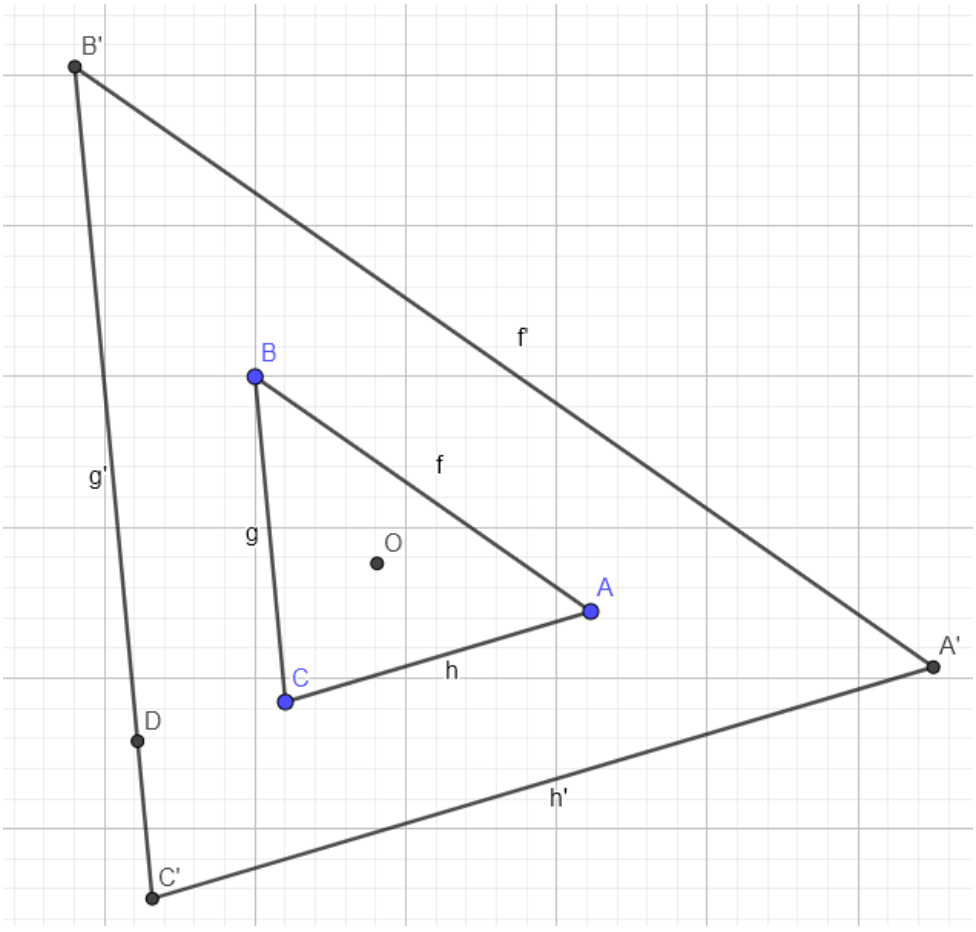

Figure 11: Constant width diffraction margins.

Figure 11 illustrates the desired geometrical result — a translation of each of the triangle vertices such that the new edges formed between them are parallel to the old edges and a

constant perpendicular distance from them. The diffraction margins are nothing more than a preemptive enlargement of all triangles by a constant width. This occurs at the same time as the preemptive inverse transform step and amounts to adding to each of the triangle vertices a weighted sum of the two edge vectors that meet to make up that vertex. The weights on these vectors are inversely proportional to the projection of the respective edge vector onto the other edge's normal.

$$A' = A + \left[\frac{\overline{CA}}{\overline{CA} \cdot T_{AB}} + \frac{\overline{BA}}{\overline{BA} \cdot T_{CA}}\right] m$$

$$B' = B + \left[\frac{\overline{CB}}{\overline{CB} \cdot T_{AB}} + \frac{\overline{AB}}{\overline{AB} \cdot T_{BC}}\right] m$$

$$C' = C + \left[\frac{\overline{AC}}{\overline{AC} \cdot T_{BC}} + \frac{\overline{BC}}{\overline{BC} \cdot T_{CA}}\right] m \tag{21}$$

This is then multiplied by the diffraction margin width, m, which is up to the user to determine based on their desired level of accuracy. A more intuitive formula for it can be derived in terms of the frequency of the radiation and the desired minimum diffraction-angle-standard-deviation ε to consider.

$$\varepsilon = \operatorname{atan}\left(\frac{\Delta p}{|p_{old}|}\right) = \operatorname{atan}\left(\frac{\frac{\hbar}{2x}}{\hbar k}\right) = \operatorname{atan}\left(\frac{c}{4\pi f x}\right). \tag{22}$$

Setting x = m to find the value at the very edge of the margin yields

$$\varepsilon = \operatorname{atan}\left(\frac{c}{4\pi f m}\right). \tag{23}$$

And solving this for the margin width given a desired epsilon yields

$$m = \frac{c}{4\pi f \tan\left(\varepsilon \frac{\pi}{2}\right)}. \tag{24}$$

Setting a value of epsilon of 0.001 in our testing produces margins that transition from diffraction to the lack thereof smoothly without any jump. Higher values may be appropriate for speeding up simulation where the user has determined that these small errors are inconsequential.

Moving back into the ray tracing kernel, we can revisit the ray triangle intersection algorithm to make it give us a distance value to use in our calculation of the deflection angle standard deviation. In general, the barycentric coordinates give a measure of the normalized distance from each edge of the triangle. If a coordinate is zero, then the point lies on an edge. If a coordinate is less than zero, it lies past an edge and should not be considered to be inside the triangle. We can map these coordinates to give us instead the distance from the edge of the original triangle without additional margins, and check if this distance is less than zero for all three edges to see if the ray hit the actual triangle.

$$D_{edgeBC} = m - U\overline{A'B'} \cdot T_{BC},$$

$$D_{edgeAC} = m - V\overline{B'A'} \cdot T_{AC},$$

$$D_{edgeAB} = m - W\overline{A'C'} \cdot T_{AB}. \tag{25}$$

If any of these distances are positive, then the ray hit only the margin and not the actual triangle, meaning we will have a diffraction. The distance can then be used directly in the calculation of the deflection angle standard deviation, as it corresponds to a physical distance absolutely, rather than to a distance local and relative to the triangle.

3.4 Results

In order to test the ray tracer's ability to perform diffraction, we set up a scenario shown in Figure 12, including two occluding triangles with a gap between their edges. We added an isotropic point source antenna and a simulation volume including the antenna and occluder.

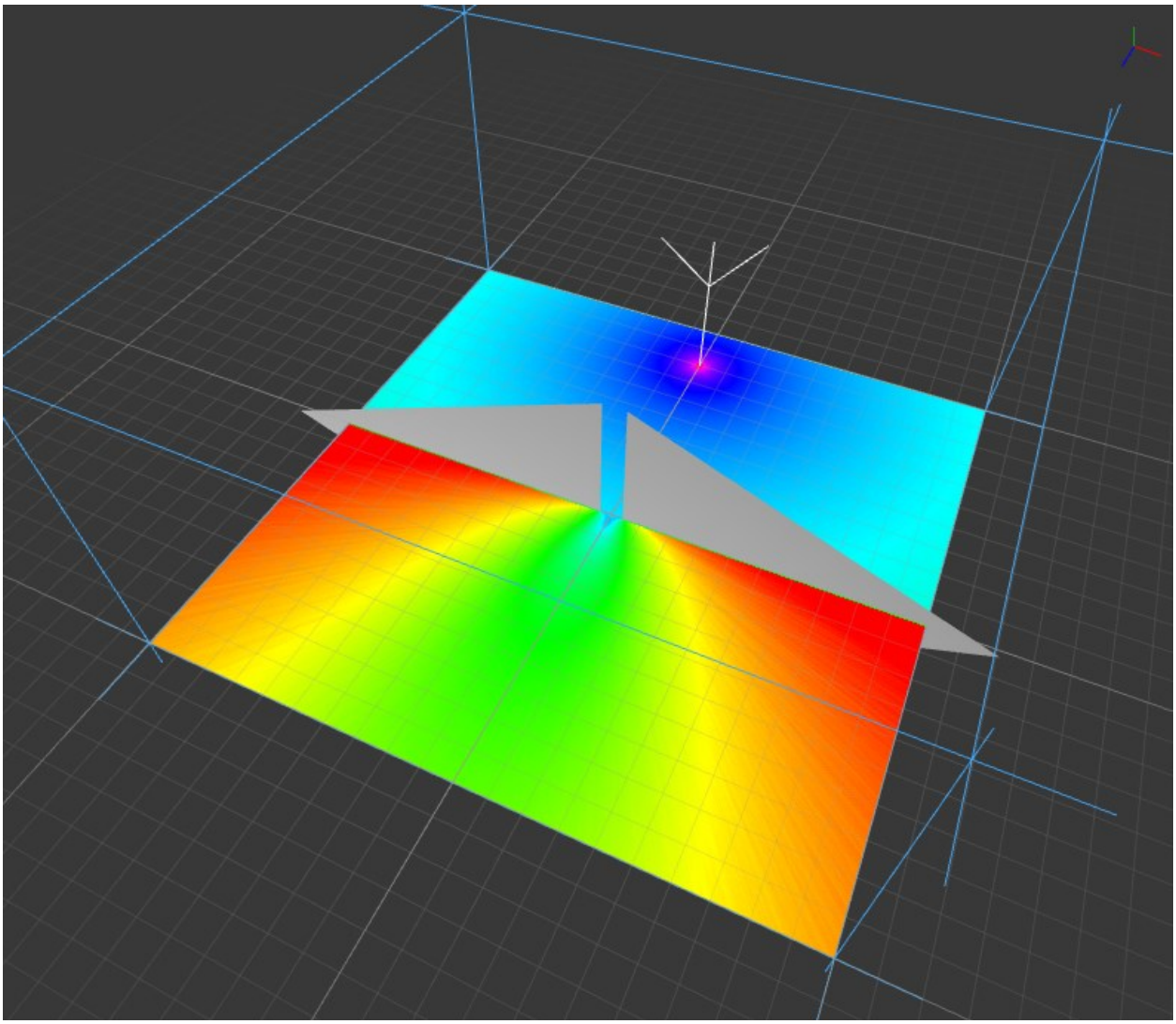

Figure 12: The setup for the diffraction test.

The simulation frequency was set to 5.9 GHz, the antenna power was 1 W, the volume was 2x1x2 meters, and the gap was 0.075m or 75mm. The coloration is based on the intensity by mapping a user-defined range to the range of integers 0 to 360 linearly and using this as the hue

angle for an HSV color with full saturation and value. In this case, the scale lower bound was set to -90dB and the upper bound was 10dB. Figure 13 shows more clearly the results as exported from the application into a .png file.

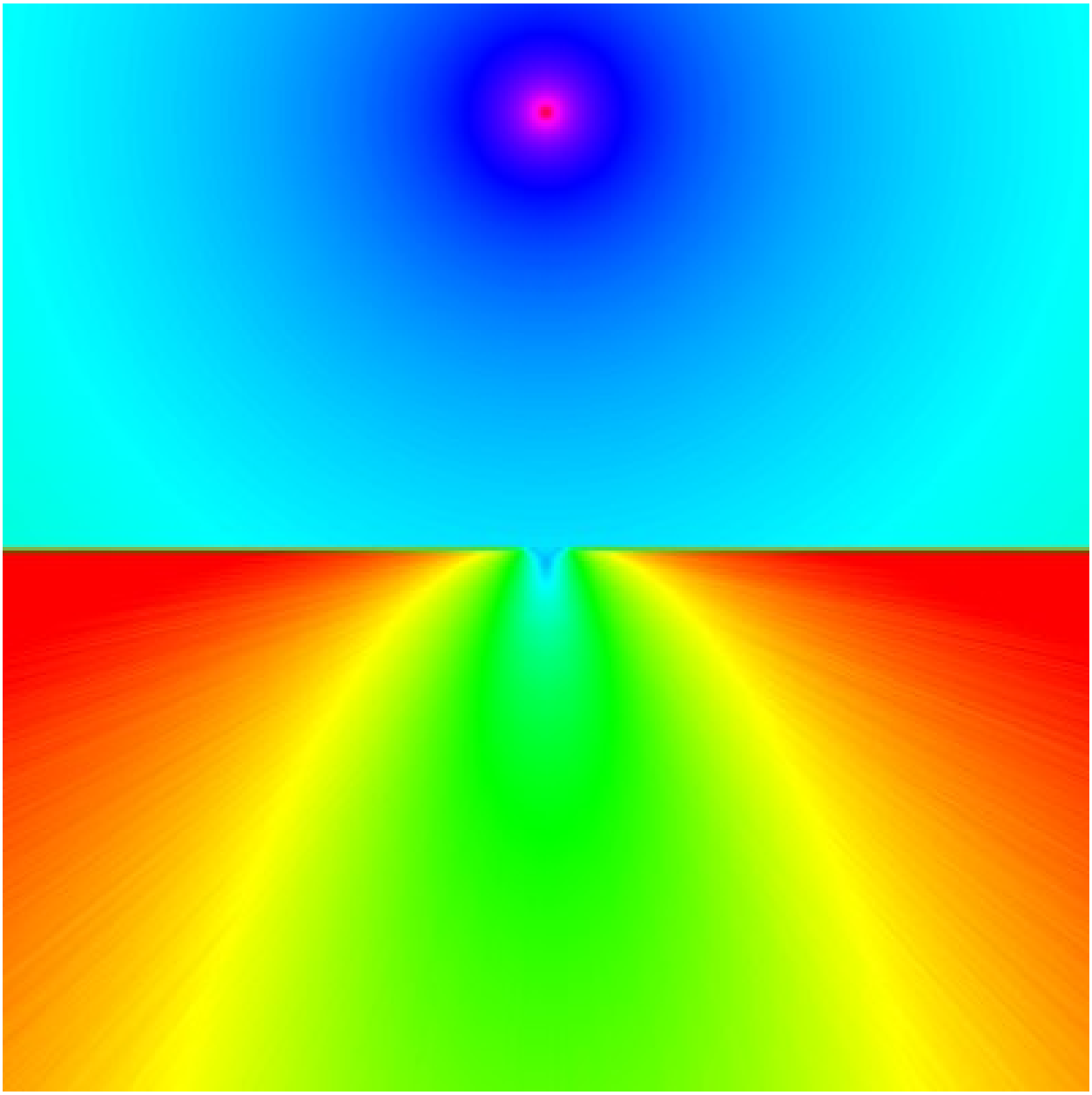

Figure 13: The exported results of the diffraction past a single slit.

While these results are certainly promising, we need to confirm that this corresponds to a real-world diffraction scenario. Using the Huygen's-Fresnel Principle, we can derive an analytical solution to the diffraction past a single slit to be [38]

$$I(\theta) = I_0\, I(0)\, sinc^2(\beta), \tag{26}$$

where

$$\beta = k\frac{D}{2}\sin(\theta)\,, \tag{27}$$

and we have intensity (power) I as a function of the angle theta from the slit, the initial power I_0, direct undeflected power $I(0)$, wavenumber k, and slit width D. We took the intensity values along the bottom row of the simulation results and plotted them, along with a moving-average filtered version to eliminate some of the noise. We also plotted the analytical equation along the same path to compare. The results of this are shown in Figure 14.

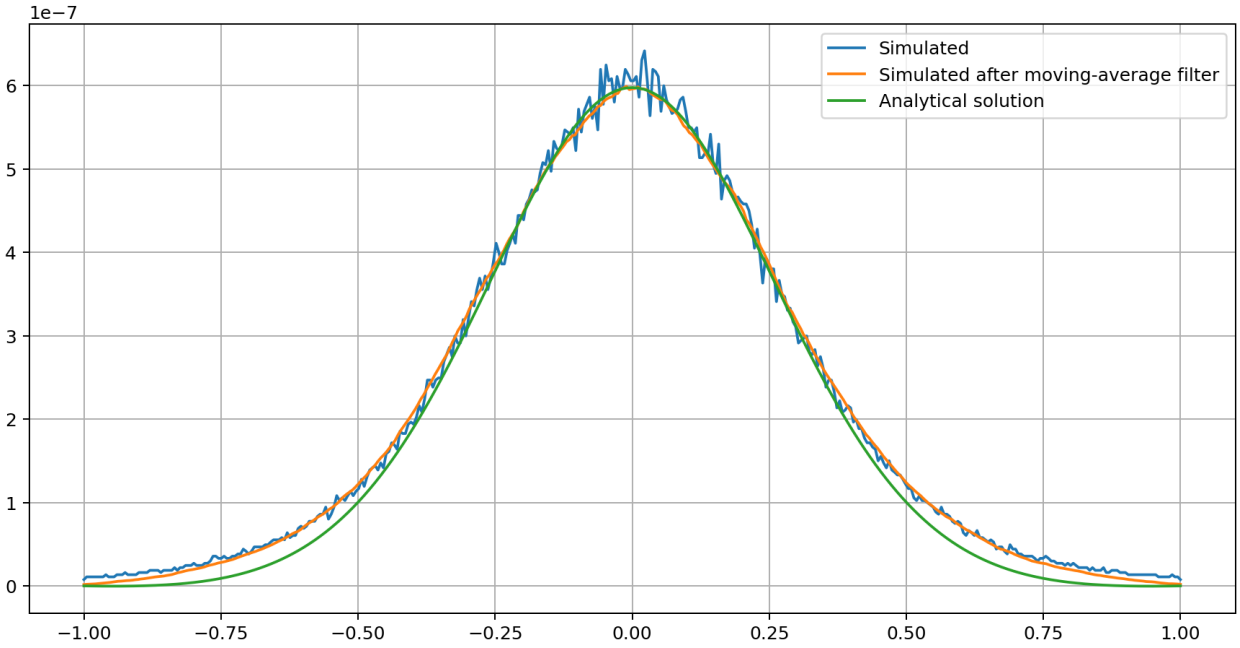

Figure 14: Intensity along the path 1 m away from the slit, simulated with and without filtering and solved analytically.

From this we can see that the simulation matches the expected results quite well. Notably, the simulation overestimates the intensity in a pair of regions between the main lobe and the endpoints. This is because the use of the normal distribution for the stochastic deflection angle does not take into account interference effects — in short, it can predict only the envelope of the diffraction, not the precise structure. For our purposes this is entirely fine and was expected, but it does mean that a very finely resolved small-scale simulation will lack the precision of a true wave solving method.

Finally, we can consider the performance of the ray tracer. We examine a scenario with an antenna surrounded by 16 uniformly separated triangles of different sizes in an inner and outer ring. We stepped up the number of rays traced by powers of two and timed how long the simulation took, running ten trials of each case to reduce the likelihood of undetected statistical anomalies. The results of this are shown in Figure 15.

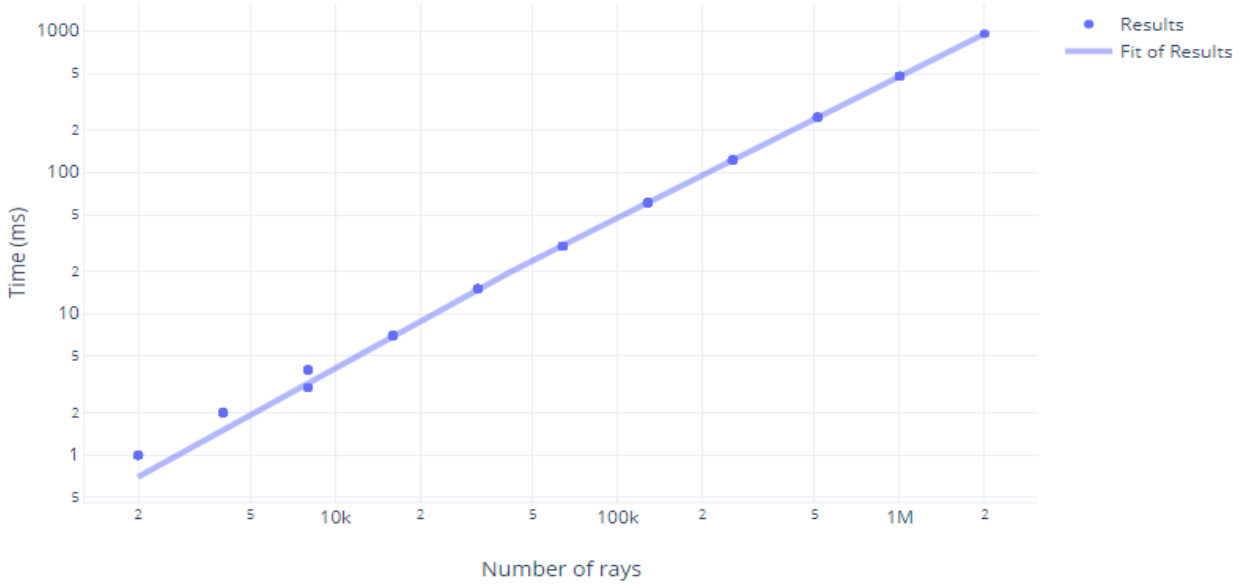

Figure 15: Simulation time vs. number of rays simulated.

This indicates that the time is linear in number of rays, which is to be expected, and that the ray tracing algorithm is capable of running a little over 2 million rays per second on our AMD Vega 56 GPU. This translates to only 500 nanoseconds per ray. Alternatively, we can hold the number of rays constant and increase the grid resolution. We used an empty unit cube as our simulation volume for these tests and evaluated only the time to trace all 10000 rays, not the time of the entire simulation as before. The results are shown in Figure 16.

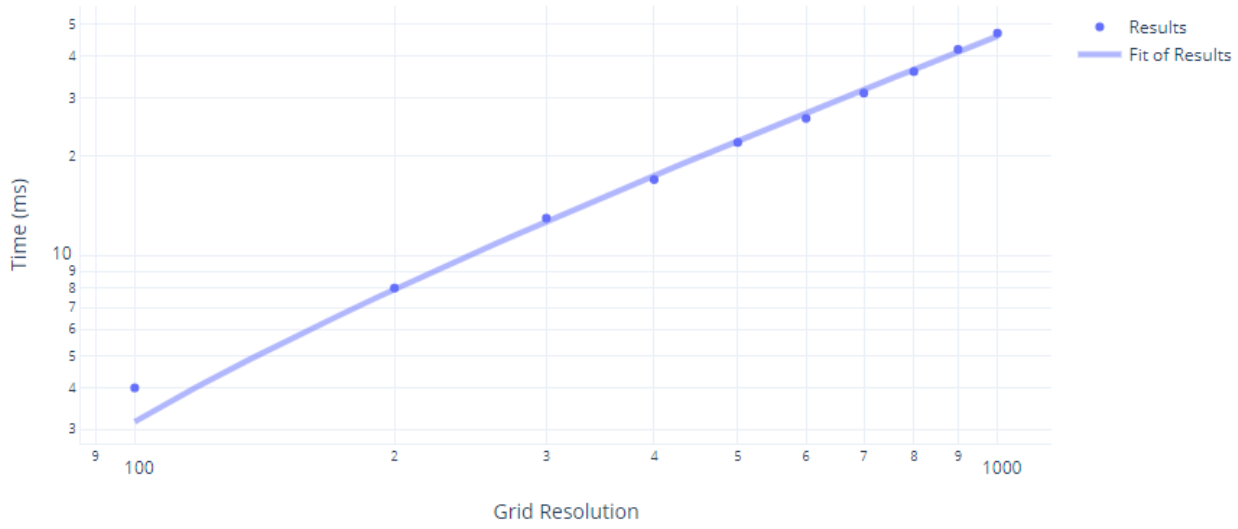

Figure 16: Ray tracing time vs. grid resolution.

We evaluated the ray-trace times for resolutions in steps of 100 from 100x100x100 to 1000x1000x1000. We find a linear time complexity for ray tracing relative to grid resolution, once again as expected.

Chapter 4: Application Development

The technologically simplest but practically most time-consuming aspect of this project is the development of the actual user interface and application. This section covers the overall architecture of the application and how it was constructed in Qt, some details regarding the 3D viewport, and consideration of the final resulting application for the purposes of this thesis. Much more work is planned here, but time constraints require a final submission.

4.1 Application Overview

This application is developed using the Qt libraries [39] for a clean, portable GUI. Qt functionality is used to open a platform-independent window containing a tool panel, a 3D viewport, and a scene hierarchy view, the three main graphical components of this application which will be discussed below.

The scene hierarchy view displays a tree data structure displaying the list of independent layers, each of which is simply a container for lists of antennas, occluders, and simulation contexts that the user has organized together. The ability to group together related parts of the project into layers allows for more easily performing related but substantially different simulations in a single project, for example, simulating the field intensity from a number of

antennas positioned on the roofs of cars around a loop, and another simulation in which all the cars are replaced with trucks with the antennas moved up analogously.

The tool panel is a context-dependent widget that allows the user to view and edit the parameters of the scene elements, including simulation contexts, antennas, or occluders. This includes moving, scaling, and rotating depending on the type of object selected (e.g. antennas cannot be scaled), as well as more specific operations such as setting the simulation parameters for a simulation context.

On the backend, we used OpenCL [40] to support efficient computation on heterogenous hardware consisting of both CPUs and GPUs (or even FPGAs where implementations exist). OpenCL allows us to write a single, unified version of the simulation ray tracer code which can then be executed on any hardware we have the OpenCL driver for, making our application truly cross platform, except for the manufacturers who have stopped supporting OpenCL on certain lines of products.

4.2 3D Viewport

In order to work easily in a 3D environment, the user must be able to navigate a real-time 3D viewport. This necessitates the use of a 3D graphics library. We selected OpenGL [41] for this purpose for its portability, and fortunately Qt comes with libraries for getting OpenGL to run in a panel alongside our other widgets.

By inheriting from the QOpenGLWidget and one of the QOpenGLFunctions_* classes, we can build the viewport as a discrete widget, ready to plug into the overall user interface. The class gives us access to two important methods that will be called automatically by Qt, namely

initializeGL() and paintGL(). The initialization function is called once, after an OpenGL context has been established, meaning OpenGL functions can be called here and afterward but will not work if called before. We use the initialization function to set up the state in which we want to do our rendering, including setting the background color, blending function, multisampling, and disabling backface culling. In the paint function, we draw the floor grid and an indicator of the current orientation of the axes. We also call the hierarchy's render function, which in turn calls the render function of each layer, which in turn calls the render function of each antenna, occluder, and simulation context. We also perform a technique known as color picking, where we render all objects in the scene with a unique RGB color ID, query the color of the pixel under the mouse, and clear the screen before rendering the actual graphics. This allows us to determine the object over which the user is hovering and act on a mouse click by, for example, selecting the object under the mouse cursor.

Even using OpenGL as a base, the development of a 3D application is non-trivial. We needed to develop an .obj file loader to import 3D models from external programs, a VBO-IBO-VAO kit for handling vertex data and transferring it to the GPU according to OpenGL's rules, and a shader library capable of parsing OpenGL shader source code (which we wrote) to build shader objects that were connected to uniform parameter values and vertex-specific attributes.

To simplify our toolchain for purposes of discussion, the OBJ loader opens an .obj file and parses out a list of vertices, a list of texture coordinates, and a list of normal vectors. These are then used to build triangles based on a final list of face indices. The list of triangles is kept in the model class, as well as being transferred to the GPU. The transfer occurs by loading the vertices and their indices into a corresponding Vertex Buffer Object (VBO) and Index Buffer Object (IBO) with an additional Vertex Array Object (VAO) used to hold binding information.

OpenGL calls can then be used to request that the GPU render using these buffers as the source for vertex data. When this occurs, the GPU invokes one instance of a tiny program called the vertex shader for each vertex in the VBO. This performs processing per-vertex, such as multiplication by a camera matrix to transform the scene to respond to the user dragging the mouse. After this initial processing, the vertices are used to rasterize triangles into pixel-size fragments. Each fragment has a location on the screen and some inherited data from the vertices of the triangle used to produce it. Once available, the fragments are then passed to the fragment shader, another tiny program that the GPU invokes one of for each fragment. The fragment shader applies, in our case, some simple lighting logic to the solid surfaces and solid coloring on the lines, plus a texture lookup for coloring the simulation results. The output of the fragment shader step is the finished rendered frame to be displayed, allowing the user to see all the components of the simulation. The understanding of this dataflow is not an academic endeavor. All of the components to perform these tasks must be built up by the developer of the application.

4.3 Resulting Application

A screenshot of the final version of the application, (at least for the purposes of this document – more work is planned as will be discussed shortly,) is shown in Figure 17.

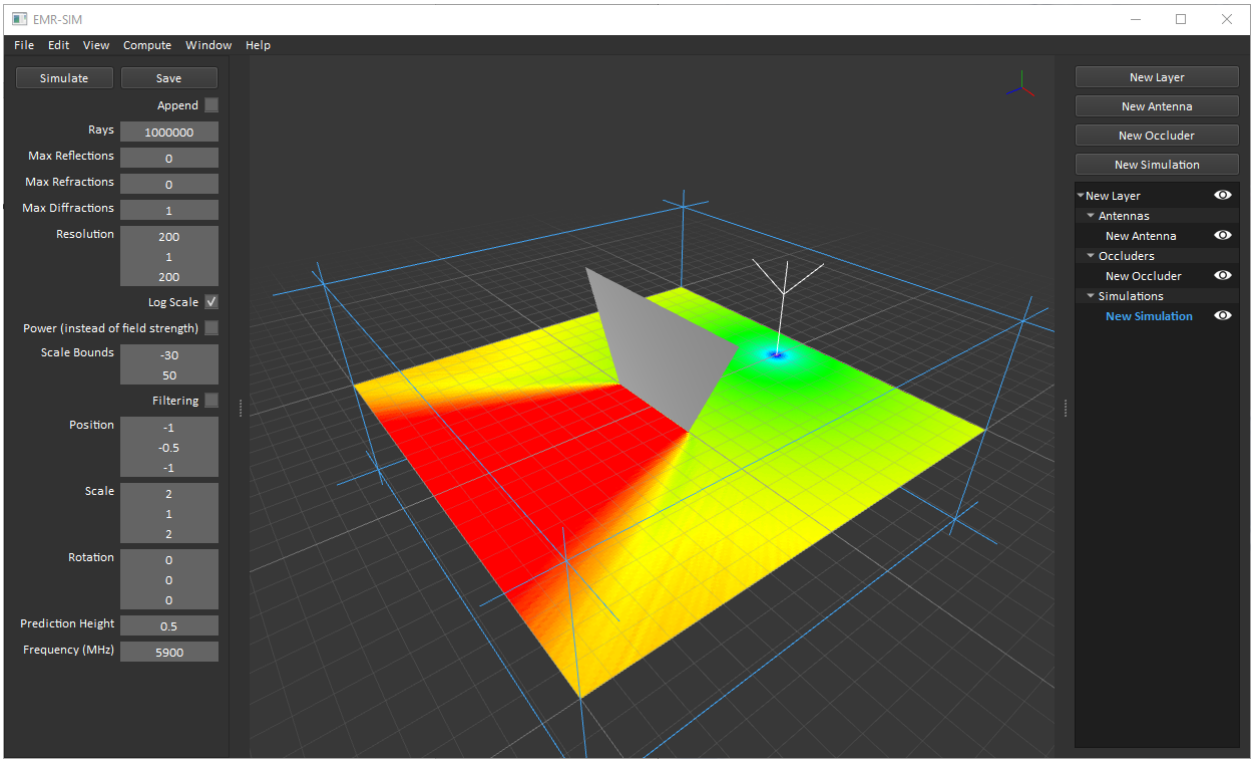

Figure 17: The final application.

Here we have the tool panel on the left, showing the options for the simulation context which is shown in blue because it has been selected. On the right we have the hierarchy view, with the antenna, occluder, and simulation organized into the tree and buttons for creating additional objects. The eye icons can be toggled to show or hide individual objects or entire layers, allowing the user to see under or around large obstacles and reduce clutter.

The camera can be controlled using the mouse by dragging with the left mouse button to pan the camera and with the middle mouse button to rotate it, or using the keyboard shortcuts to view the selected object, flip the camera to the other side of whatever is being looked at, or lock to one of the basis axes. Projects can be saved and loaded to/from small files, and a list of available compute devices allows the user to choose which devices to use for computation.

In total, this application has many desirable features, but is still lacking a lot of necessary functionality like an undo/redo stack, CPU compute via multithreading for CPUs that do not support OpenCL, proper user-facing documentation, and dozens of other features that were not strictly necessary and had to be cut for time reasons.

One very notable improvement our application shows compared to the alternative field propagation simulation techniques as enumerated in section 1.5 is our ability to simulate an entire volume at once. Technically speaking, there is nothing stopping any of these methods from performing full simulations over a 3D volume instead of simulating only a 2D slice at a given height, but aside from the empirical models, they are simply too slow to get away with this. If simulating a 200 by 200 grid takes SRT an hour, then simulating a 200 by 200 by 200 volume will take it over a week, because information is not reused between cells. Our approach uses the same batch of rays to simulate the entire volume, meaning we can simulate much larger spaces, including full volumes, in a much shorter amount of time.

It should also be noted that this tool simulates far-field interactions, and that it is not recommended for the user to place an antenna less than 2 wavelengths away from any occluders. For a microwave frequency antenna this is not an undue burden — at 1 GHz the distance is only about 60 cm. However, as the frequency is dropped the size of the simulation volume needs to be scaled up in order to accommodate the increased size of the near-field region.

Chapter 5: Continued Work and Concluding Remarks

This final section will give a consideration of the remaining work that needs to be done on the application and desired further features, as well as a brief conclusion to the thesis.

5.1 Continued Work

The list of work that still needs to be done to take this application from an academic implementation of a novel technique to a finished, publicly viable product is still quite long. It would be tedious to go through every feature and fix that is currently listed as needed, as well as incomplete. However, the development of this application does not end with this thesis, and work is expected to continue to develop the project fully.

The most critical issue to fix involves the ray tracer leaking occasional rays when diffraction is enabled. These rays are able to penetrate solid geometry by first hitting a diffraction margin and then ignoring another triangle that they should have hit due to floating point issues. This issue will be resolved with some additional work, but GPU bug fixing tends to be a significant time investment, as debugging code running on the GPU presents unique difficulties. Furthermore, the Qt libraries for the tree used in the scene hierarchy view have some issues when adding items that are sorted below other items, resulting in occasional overlap. This must be

fixed by reimplementing some base functionality, which has not been done yet as again, it would be time consuming for a project with a strict deadline.

As for desired features, an implementation of a grab function, where the selected object can be dragged, scaled, or rotated using the mouse directly instead of through the interface would speed up work considerably. At the moment, only a single compute device can be selected at a time, and multithreaded CPU compute is not available unless your CPU supports OpenCL directly, so adding multi-device compute and CPU compute would be beneficial. An implementation of the undo-stack is expected for this kind of application, which will require some refactoring of user actions. Finally, the application needs a lot of documentation, both internally and in the client-facing interface. A function to open a PDF manual from inside the app was trivial to add, but actually writing that manual will take more time.

It would be helpful to develop some method of determining convergence empirically, such as having the user specify a minimum amount of change per iteration of the simulation algorithm and stopping when that change is not observed. Implementing a criteria for convergence is not especially difficult, and is considered to be planned work.

5.2 Conclusion

This project set the goal of developing an application to enable a new kind of electromagnetic field propagation simulation using Monte Carlo ray tracing. To enable this, we had to develop novel optimizations to existing algorithms as well as entirely new approaches based on extensive research. We were able to test our approaches for both correctness and performance and we showed that our methods offer a significant advantage in many situations by

allowing easier development of simulation environments using third party tools and faster simulation times than existing alternatives including the ability to compute entire volumes simultaneously, rather than computing slices one at a time.

Overall, this project was entirely successful, and finishing the remaining work that still needs doing is only a matter of time. All the interesting technical challenges have been resolved, and the application works as it should in spite of a few rough edges. Work will continue on this project to develop the ideas shown here into a fully realized and marketable field propagation solution for standalone use, as well as the potential integration into existing software as a complementary simulation technique alongside empirical models, standard ray tracing, and the dominant path model.

References

[1] "3D EM Analysis," Cadence Design Systems, Inc., 2020. [Online]. Available: https://www.awr.com/awr-software/products/analyst. [Accessed 30 July 2020].

[2] "Comsol," Comsol, 2020. [Online]. Available: https://www.comsol.com/. [Accessed 30 July 2020].

[3] "Altair University," Altair Engineering, Inc., 2020. [Online]. Available: https://altairuniversity.com/feko-student-edition/. [Accessed 30th July 2020].

[4] A. Harish, "Numerical Approach: Finite Element Analysis," Simscale, 10 March 2020. [Online]. Available: https://www.simscale.com/blog/2016/10/what-is-finite-element-method/. [Accessed 30 July 2020].

[5] M. Koshiba and K. Inoue, "Simple and efficient finite-element analysis of microwave and optical waveguides," IEEE Transactions on Microwave Theory and Techniques, vol. 40, no. 2, pp. 371-377, 1992.

[6] K. Siegrist, "The Method of Moments," 2020. [Online]. Available: https://www.randomservices.org/random/point/Moments.html. [Accessed 30 July 2020].

[7] W. C. Gibson, The Method of Moments in Electromagnetics, 2nd Edition, CRC Press, 2015.

[8] J. B. Schneider, "Understanding the Finite-Difference Time-Domain Method," 2020. [Online]. Available: https://eecs.wsu.edu/~schneidj/ufdtd/ufdtd.pdf. [Accessed 3 August 2020].

[9] A. Taflove and S. C. Hagness, "The Yee Algorithm," in Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd Edition, Artech House, pp. 58-78.

[10] F. M. Landstorfer, "Wave Propagation Models for the Planning of Mobile Communication Networks," European Microwave Conference, vol. 1, no. 29, pp. 1-6, 1999.

[11] Z. Yun and M. F. Iskander, "Ray Tracing for Radio Propagation Modeling: Principles and Applications," IEEE Access, vol. 3, pp. 1089-1100, 2015.

[12] G. Yang, K. Pahlavan and J. F. Lee, "A 3D propagation model with polarization characteristics in indoor radio channels," Proceedings of GLOBECOM '93. IEEE Global Telecommunications Conference, vol. 2, pp. 1252-1256, 1993.

[13] G. Woefle, R. Wahl, P. Wertz, P. Wildbolz and F. Landstorfer, "Dominant Path Prediction Model for Urban Scenarios," IST Mobile and Wireless Communications, no. 14, 2005.

[14] J. Buck, "The Recursive Ray Tracing Algorithm," 16 December 1999. [Online]. Available: http://www.geocities.ws/jamisbuck/raytracing.html. [Accessed 11 July 2020].

[15] P. Falstad, "falstad.com," [Online]. Available: https://www.falstad.com/ripple/. [Accessed 30 July 2020].

[16] B. Kapralos, M. Jenkin and E. Milios, "Acoustical Diffraction Modeling Utilizing the Huygen's-Fresnel Principle," IEEE International Workshop on Haptic Audio Visual Environments and their Applications, pp. 6-12, 2005.

[17] E. Veach and L. J. Guibas, "Metropolis Light Transport," Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, p. 65–76, 1997.

[18] E. Veach and L. Guibas, "Bidirectional Estimators for Light Transport," Proceedings of Eurographics Rendering Workshop, pp. 147-162, 1994.

[19] S. Woop, G. Marmitt and P. Slusallek, "B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes," Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pp. 67-77, 2006.

[20] M. Stich, H. Friedrich and A. Dietrich, "Spatial Splits in Bounding Volume Hierarchies," Proceedings of the Conference on High Performance Graphics 2009 (HPG ’09), pp. 7-13, 2009.

[21] J. G. Cleary and G. Wyvill, "Analysis of an Algorithm for Fast Ray Tracing using Uniform Space Subdivision," The Visual Computer, vol. 4, pp. 65-83, 2005.

[22] N. Thrane and L. O. Simonsen, "A Comparison of Acceleration Structures for GPU Assisted Ray Tracing," 1 August 2005. [Online]. Available: https://www.ks.uiuc.edu/Research/vmd/projects/ece498/raytracing/GPU_BVHthesis.pdf. [Accessed 3 August 2020].

[23] M. Hapala, O. Karlik and V. Havran, "When It Makes Sense to Use Uniform Grids For Ray Tracing," Proceedings of WSCG’2011, pp. 193-200, 2011.

[24] T. Möller and B. Trumbore, "Fast, Minimum Storage Ray/Triangle Intersection," Journal of Graphics Tools, vol. 2, no. 1, pp. 21-28, 1997.

[25] E. R. Freniere, G. G. Gregory and R. A. Hassler, "Edge Diffraction in Monte Carlo Ray Tracing," Optical Design and Analysis Software, vol. 3780, pp. 151-157, 1999.

[26] R. P. Heinisch and T. S. Chou, "Numerical Experiments in Modeling Diffraction Phenomena," Appl. Opt., vol. 10, pp. 2248-2251, 1971.

[27] G. Green, "An Essay on the Application of Mathematical Analysis to the Theories of Electricity and Magnetism," Printed for the author by T. Wheelhouse, Nottingham, 1828.

[28] P. H. Pathak, G. Carluccio and M. Albani, "The Uniform Geometrical Theory of Diffraction and Some of Its Applications," IEEE Antennas and Propagation Magazine, vol. 55, no. 4, pp. 41-69, 2013.

[29] A. Lagae and P. Dutré, "Compact, Fast and Robust Grids for Ray Tracing," Proceedings of the Nineteenth Eurographics Conference on Rendering, p. 1235–1244, 2008.

[30] D. Voorhies, "Triangle-cube Intersection," in Graphics Gems III, Academic Press Professional, Inc., 1992, p. 236–239.

[31] R. Hoetzlein, "Voxelization of a triangle in 3D," 2019. [Online]. Available: https://github.com/ramakarl/voxelizer. [Accessed 12 July 2020].

[32] M. Takeshige, "The Basics of GPU Voxelization," 22 March 2015. [Online]. Available: https://developer.nvidia.com/content/basics-gpu-voxelization. [Accessed 3 August 2020].

[33] G. Turk and M. Levoy, "The Stanford Bunny," August 2000. [Online]. Available: https://www.cc.gatech.edu/~turk/bunny/bunny.html. [Accessed 3 August 2020].

[34] A. Kensler and P. Shirley, "Optimizing Ray-Triangle Intersection via Automated Search," IEEE Symposium on Interactive Ray Tracing, pp. 33-38, 2006.

[35] S. Woop, C. Benthin and W. Ingo, "Watertight Ray/Triangle Intersection," The Journal of Computer Graphics Techniques, vol. 2, no. 1, pp. 65-82, 2013.

[36] T. J. Jankun-Kelly, "Barycentric Coordinates," 6 September 2016. [Online]. Available: https://observablehq.com/@infowantstobeseen/barycentric-coordinates. [Accessed 3 August 2020].

[37] W. B. Langdon, "A Fast High Quality Pseudo Random Number Generator for NVidia CUDA," Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers, p. 2511–2514, 2009.

[38] L. Devroye, Non-Uniform Random Variate Generation, New York: Springer-Verlag, 1986.

[39] E. Hecht and A. R. Ganesan, "10.2.1 The Single Slit," in Optics, 4th Edition, Pearson, 2008, pp. 429-433.

[40] "Qt," The Qt Company, 2020. [Online]. Available: https://www.qt.io/. [Accessed 31 July 2020].

[41] "OpenCL Overview," Khronos Group, 2020. [Online]. Available: https://www.khronos.org/opencl/. [Accessed 31 July 2020].

[42] "OpenGL Overview," Khronos Group, 2020. [Online]. Available: https://www.khronos.org/opengl/. [Accessed 31 July 2020].