NTNU – Trondheim
Norwegian University of
Science and Technology

Electromagnetic Wave Coverage via Ray Tracing on GPUs

Mads Buvik Sandvei

Master of Science in Computer Science
Submission date: July 2014
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Norwegian University of Science and Technology

Master's thesis

Electromagnetic Wave Coverage via Ray Tracing on GPUs

Author:
Mads Buvik Sandvei

Supervisor:
Anne C. Elster

A thesis submitted as my Master's project

July 2014

Problem Description

Predicting coverage of light and other signals in a landscape is an important feature in applications from decisions about where to place light poles and radio towers to understanding how the level of light and/or others are influenced by terrain, buildings, moving objects, etc.

This project focuses on using ray-tracing combined with the HPC-Lab Snow Simulator to create a simulation that can predict such coverages. Effects of winds on high towers and other artifacts that may be inferred from feature of the current HPC-Lab Snow Simulator, may also be included.

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

Electromagnetic Wave Coverage via Ray Tracing on GPUs

by Mads Buvik Sandvei

With the appearance of libraries such as NVIDIA's CUDA programming environment and and NIDIA's OptiX ray tracing framework, we can make use of the processing power of modern GPUs to perform compute-intensive simulations.

Simulation of electromagnetic waves such as radio waves based on a physically accurate model of propagation, using the method known as ray tracing, is theoretically capable of producing far more realistic coverage maps than those produced by traditional methods such as Longley-Rice which is based on statistics. However, using the ray tracing approach causes the computational cost of the simulation to become enormous, and performing a full simulation on the CPU alone could take hours, if not days.

This thesis gives an overview of the technique of ray tracing and theory surrounding the propagation of electromagnetic waves. We then use this to build a radio wave simulation application which performs all simulation on the GPU. Each wave is reflected and refracted based on dielectric properties of the medium(s), although the exact values used might not precisely correspond to the precise dielectric properties of any actual terrains, and interference caused by differences in phase is accurately accounted for. Finally, upon intersection with terrain or receiver, the signal strength is stored in a buffer mapped onto the terrain which can then be visualized.

In conclusion, the thesis argues that the results obtained show promise for great improvements on signal coverage calculation. No explicit comparison between Longley-Rice and the presented method is made, as no expectancy is made that the method has incorporated enough detail about radio wave propagation and enviornmental effects for the simulation to compare. The thesis instead hopes to show the potential in generating coverage maps by means of ray tracing...

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

Electromagnetic Wave Coverage via Ray Tracing on GPUs

by Mads Buvik Sandvei

Med ankomsten av bibliotek som NVIDIAs CUDA programmeringsmiljø og NVIDIAs Optix ray tracing rammeverk, så kan vi bruke prosesseringskraften til moderne GPUer for å utføre beregningstunge simuleringer.

Simulering av elektromagnetiske bølger som radiobølger basert på en fysiskt nøyaktig propageringsmodell, via metoden kjent som ray tracing, kan potensielt gi langt mer realistiske dekningskart enn de som gis av tradisjonelle metoder som Longley-Rice som er basert på statistikk. Men å bruke denne framgangsmåten gir en enorm beregningskostnad til simuleringen, og å utføre en full simulering på CPU vil kunne ta timer, om ikke dager.

I denne avhandlingen gis et overblikk over ray tracing teknikken og teori omkring propageringen av elektromagnetiske bølger. Vi bruker så dette til å bygge en radiobølgesimulering som utfører all simulering på GPU. Hver bølge reflekteres og refrakteres based på mediets dielektriske egenskaper, skjønt disse verdiene vil muligens ikke korrespondere til de eksakte dielektriske egenskapene til noen virkelige terreng, og interferens forårsaket av forskjelleg i fase blir gjort rede for nøyaktigt. Til slutt, ved interseksjon med terreng eller mottaker, lagres signalstyrken i et buffer som blir kartlagt på terrenget og kan deretter visualiseres.

Til konklusjon, avhandlingen argumenterer at resultatene som ble oppnådd er lovende for store forbedrelser for signalstyrkeberegning. Ingen eksplisitt sammenligning med Longley-Rice blir gjort da simuleringen ikke tar høyde for nokk detaljer om radiobølgepropagering og effekter fra omgivelsene for at simuleringen skal kunne sammenlignes. Avhandlingen håper i stedet å vise potensialet i det å generere dekningskart ved hjelp av ray tracing...

Acknowledgements

I would like to thank my advisor Dr. Anne C. Elster for her guidance and counseling on this project, Dr. Lloyd D. Clark, Atmel Norway, for helping me with Radiowave theory, and Kongsberg Gruppen through Mr. Alexander Gosling, my future employer, for inspiring the topic of this thesis.

I wopuld also like to acknowledge the contributions of NTNU and Nvidia through their CUDA Research Center and CUDA Teaching Center Programs for supporting the NTNU/IDI HPC-Lab led by Dr. Elster.

Contents

List of Figures

List of Tables

Abbreviations

EM	**E**lectro**m**agnetic
UV	**U**ltra**v**iolet
CUDA	**C**ompute **U**nified **D**evice **A**rchitecture
FLOP	**Fl**oating point **Op**eration
API	**A**pplication **P**rogramming **I**nterface
KATE	**K**DE **A**dvanced **T**ext **E**ditor
GCC	**G**NU **C**ompiler **C**ollection
NVCC	**Nv**idia **C**UDA **C**ompiler
PTX	**P**arallel **T**hread E**x**ecution

Symbols

E	Electric Field
B	Magnetic Field
ϵ	Electric Permittivity
ϵ_0	Electric Permittivity of free space
ϵ_r	Relative Electric Permittivity
σ	Electric Conductivity
μ	Magnetic Permeability
μ_0	Magnetic Permeability of free space
μ_r	Relative Magnetic Permeability
η	Characteristic Impediance
f	Frequency
ω	Angular Velocity
λ	Wavelength
δ	Depth of Penetration

Chapter 1

Introduction

Accurate simulation of radio signal coverage is an interesting and challenging problem in both civilian and military research. Knowing where radio signals will reach from any given location is important for efficient placement of transmitters, both for indoors and outdoors communication. However, the most accurate methods require a lot of computational power.

Modern GPUs with their very large number of cores, offer great performance for applications that can take advantage of parallel computing. Ray-Tracing [Whi80] is a compute-intensive visual rendering method that uses simulation of the physical propagation of light waves to improve the quality of the visualization. Fortunately, the ray tracing algorithm can take advantage of GPUs.

1.1 Motivation and Objective

Our goal is to show how to take advantage of ray tracing on GPUs to obtain an accurate propagation model for electromangentic signals with good performance.

1.1.1 Performance

Ray tracing is famous for the fantastic image quality one can achieve, but infamous for the immense computational cost involved. Fortunately, modern GPUs offer, as we mentioned, a great performance for applications that can take advantage of their parallel nature. A GPU chip has a very large number of cores, and each of these operate as vector processors to fairly large vectors. These chips are designed for computation on graphical problems, where each pixel can typically be calculated entirely independent of the other and without control-flow divergence (each pixel will have the exact same work done on it). It is appropriate to regard

the GPU as a single very wide vector processor. Through APIs such as CUDA we can implement any algorithm on the GPU, however only algorithms where a large amount of data can be processed in parallel lock-step as vectors of independent data sets can take advantage of the performance offered. Since ray tracing consists of tracing thousand or millions of entirely independent rays, it is intuitive that ray tracing might be very well suited for execution on GPUs. [TS09] explores adapting ray tracing to this model of execution with great success, and in taking advantage of this the NVIDIA OptiX API allows us to easily take advantage of GPUs to implement and accelerate ray tracing.

1.1.2 More Accurate Propagation Model

Several different methods for accurately simulating or calculating radio signal coverage have been explored. Currently the Longley-Rice model, proposed nearly 50 years ago [RLNB65], is the most popular model for approximating signal coverage. Unfortunately it is highly inaccurate, and is based on statistics of the area [Tje13] and is therefore dependent on field measurements. The model takes atmospheric changes, topographic profile, and free space into consideration, but these are also based on statistics.

This thesis will explore the usage of ray tracing to simulate radio waves, in order to calculate outdoors signal coverage. Previously, the usage of ray tracing to model radio waves has been highly limited due to the immense computational cost, but today's hardware has grown fast enough that it is worth an attempt to create an accurate simulation based on ray-tracing. Since radio waves and light are both electromagnetic waves, it is natural that we can apply the method of ray tracing to radio waves to accurately simulate the propagation of radio waves through any geometry, and in doing so make a physically accurate simulation.

A classic challenge in calculating radio coverage is including the effects of reflection and diffraction on coverage. Reflection and diffraction can allow signal to reach places it otherwise could not reach, such as behind a hill that blocks radio waves from reaching a receiver by line-of-sight or around structures. However, reflection can also interfere with, and possibly neutralize, coverage in other areas. The latter is due to the differences in phase when multiple rays reach the same point by different paths, caused by both differences in path length and changes in phase occuring on reflection. Radio waves ondulate, as their name implies, in a wave-like manner and two waves that have phase around half a period away from each other will combine destructively. For accurate calculation of coverage, it is therefore vital to be able to accurately simulate reflection and diffraction of radio waves.

Utilizing ray tracing allows us very high accuracy in determining the reflection, refraction, and direct line of sight for waves propagating out from an antenna.

None of this would have been possible without precise knowledge of ray directions, surface normals, and the exact point of impact only possible to find by tracing rays. In other radio wave applications and calculations, one would use "Fresnel Zones" to determine whether a reflection acts interferently or not. Ray tracing-based simulation allows these zones to occur naturally.

Slightly less important than reflection and diffraction, but still worth mentioning, transmission of waves through matter (refraction) can also affect the precise coverage. In and around cities, towns, or villages with scattered or systematic placement of structures, as well as out in nature with dense or sparse vegetation, a radio wave's ability to penetrate through matter can greatly affect coverage, giving coverage in areas that otherwise would have none even by reflection. In a forest, for instance, signal strength would always be close to none were it not for a radio wave's ability to penetrate through foliage and wood.

1.2 Contributions

In this thesis we will use GPUs, via NVIDIA's OptiX ray tracing framework, to accelerate the method of ray tracing and use it to implement a radio wave simulator. We use ray tracing to obtain accurate reflection, refraction, and attenuation while waves travel through matter. However, diffraction is left out in our model as this is still difficult to model accurately even with ray tracing. Our radio wave simulator is used to generate a signal coverage map over a small selection of terrains based on heightmap data, as well as to determine power incident on a single receiver placed into the map.

1.3 The HPC-Lab Snow Simulator

The Snow Simulator this project makes use of, originated from the parallelized smoke simulation by Torbjørn Vik [Vik03] which Ingar Saltvik extended into a Snow Simulation, which since has been worked on by several Master Students supervized by Dr. Anne C. Elster, including [Sal06a, Mik13, Nor12, Bab12, Ves12, Eid09, Sal06b, Gje09, San13]. The current version utilizes CUDA (implemented by Eidissen [Eid09]) to visualize the falling snow, and how it is affected by a winod field. The wind field is calculated by solving the incompressible Navier Stokes equation. The terrain of the simulator is visualized as a strip-encoded set of vertices, but is stored on disk as a heightmap with implicit coordinates.

Although this project is not directly related to neither snow nor wind simulation, we decided to make the implementation within the HPC-Lab Snow Simulator. This allows us to take advantage of the terrain model already developed in the Snow Simulator, saving some effort by reuse of components (including also basic user interface).

1.4 Thesis outline

The rest of the thesis is structured as follows:

- Chapter 2 presents background material on ray tracing, geometric intersection methods, and the OptiX framework.
- Chapter 3 presents background material on electromagnetic waves.
- Chapter 4 details how the solution was implemented.
- Chapter 5 Presents the result of the work, including images of coverages in two heightmaps and some remarks on performance.
- Chapter 6 A brief discussion on some problems with the simulation, conclusive remarks, and suggestions for future work.
- Appendix A Offers a user guide for the application.

Chapter 2

Ray Tracing & NVIDIA OptiX

In this chapter an overview of the technique of ray tracing will be given, followed by a brief description of some simple geometric identities used, and an overview of NVIDIA's CUDA and OptiX which we will use to implement ray tracing.

2.1 Ray Tracing

Ray Tracing is a technique that is primarily used to generate computer graphics, by reverse-tracing the path of light from each pixel of a camera in some scene. While the computational cost involved in performing Ray Tracing is high, the resulting imagry can be very realistic, as shown in Figre 2.1, which took the creator 560 hours to render [Tra09].

When tracing a ray, we record the starting point of the ray, its direction, and then examine the scene for surfaces that will intersect the ray. Using some logic, the exact point of intersection is determined. At this point the ray tracer can either update some buffer and exit, generate new rays (such as refraction, reflection, and shadow rays), or any combination of these. Refraction refers to rays that continue into matter (such as light traveling through glass), reflection to rays that bounce off objects (such as most visible light, particularly off of polished surfaces such as glass or metal), and finally shadow rays are used to determine luminosity based on whether paths to light sources are blocked. Shadow rays are not relevant to forward ray-tracing, as sources being blocked is implicit in rays not reaching the surface.

FIGURE 2.1: Ray Tracing can give *very* realistic images. [Tra09]

2.1.1 Ray Casting

Ray Tracing, first introduced by [Whi80], builds on the technique of ray casting, first introduced by [App68] (only later given the name ray casting). The primary difference between ray casting and ray tracing is that the former stops once a surface has been intersected, while the latter recursively defines rays (for example to enable accurate reflections and shadows).

2.2 Backwards vs. Forwards Ray-Tracing

"Ray tracing" is typically used to refer only to backwards ray tracing, which implies that each pixel of the resulting picture is given one direction (as if a camera), and then the ray of "light" that would hit that pixel is traced. The alternative is forward ray tracing, which will be relevant for this thesis. Forward tracing rays implies that ray sources are placed around in a scene and rays are traced out from these and whenever they hit scenery some logic is run (such as updating a luminosity buffer). The number of rays needed to get a good picture from forward tracing is prohibitively large, and is therefore not usually considered.

2.3 Defining a camera

When defining a camera for ray tracing, it is important to define a camera that gives a proper field of view. Defining a simple rectangle where each pixel looks straight forward would lead to a very strange and narrow field of view. Instead a field of view more reminiscient of a real camera should be applied. In this thesis two cameras will be considered: *Pinhole Camera* for backwards tracing and *Spheric Camera* for forward tracing.

2.3.1 Pinhole Camera

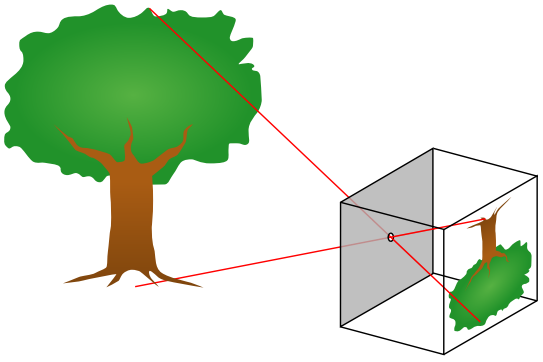


FIGURE 2.2: A pinhole camera [Pbr08]

Figure 2.2 shows a rough representation of a pinhole camera, where the back end of the camera works as a large square retina. A pinhole camera as a ray tracing source can be defined by defining the pinhole as the "eye", and the image as the back of the camera (like a retina). This can be represented by a pinhole coordinate, a vector to the center of the image, a vector to one of the sides of the image, and finally a vector to the top or bottom of the image. Rays are then spawned from the individual pixels at the back and sent in the direction of the pinhole. In practice, though, it's a good idea to start the ray at the pinhole (or add the distance to the pinhole to the minimum distance to trace, which most ray-tracing frameworks will define), as the output will become very strange if an object happens to appear between the pinhole and the image.

2.3.2 Spheric Camera

When trying to simulate a radio transmitter, ideally we want to spawn rays in all directions from the "eye". This presents a fundamental challenge: How do we define this "camera"? The simplest solution is to use spherical coordinates to define directions, as in Figure 2.3. However, when tracing radio waves this results in a problem as we will see later in Section 3.4.

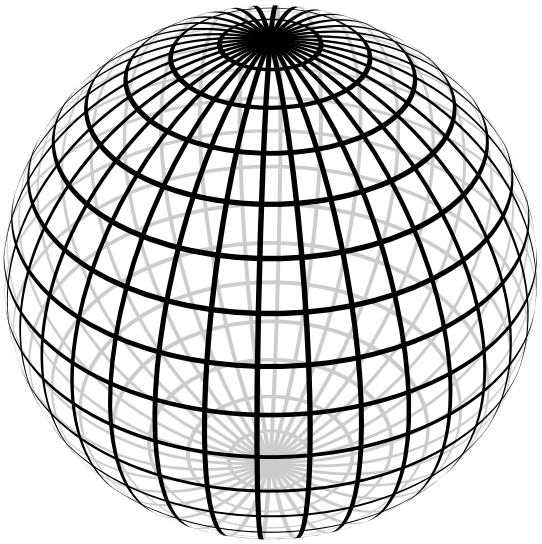


FIGURE 2.3: A sphere with lines drawn as if from spherical coordinates. [Gee09]

2.4 Intersecting the scene

Easily the most computationally demanding part of ray-tracing, the logic behind intersecting the geometry can be either simple or complex depending on what geometric figures are to be intersected. Examples of very simple geometry to intersect are axis aligned boxes and spheres, which can be intersected using simple equations that take very few floating point operations to compute. However, in more realistic scenes most figures, such as a human hand or a field of grass, can't be represented by simple equations, at which point determining intersections becomes much more challenging. However, such highly complex scenes will not be considered in this Thesis.

To intersect the scene, the first step is to determine what objects to test for intersection, and the second to test all of these, picking the one that intersects closest to the origin (if any). To determine what objects to test, a simple but naive method would be to just test every single object. But this would be a huge waste of cycles, as one can quickly determine that some objects aren't worth testing, such as those that lie behind the origin or are just nowhere near the ray's path. To determine this, the typical approach is to construct a hierarchy of *bounding volumes*.

2.4.1 Bounding Volumes

Bounding Volumes are an essential key to the intersection of rays and objects. In short, a bounding volume is just an extremely simple geometric object that fully contains an object or a set of objects. The simplest example of a bounding volume

is a box, which is simply constructed such that it contains the object(s) and as little extra space as possible (minimal bounding box). A ray that does not even intersect this box does thus not need to care about whatever complex objects lie inside and can go on to test other objects. The bounding volume does not need to be a box, othertimes a sphere can contain the object(s) while wasting much less space (while still being simple to intersect).

2.4.2 Hierarchy of Bounding Volumes

To further intersection, we can build a hierarchy of bounding volumes. One large bounding volume can contain an arbitrary number of other smaller bounding volumes, and thus starting at a higher level in the hierarchy we can start ray tracing by pruning away large numbers of objects contined within, significantly reducing rendering times. The downside to creating a detailed hierarchy is that constructing the hierarchy itself becomes a challenge, as well as computationally expensive.

Furthermore, a bounding volume does not need to contain a full object. For more complex object, there is typically one large bounding volume containing the whole volume, then several volumes containing subsets of the object. For example, when bounding a large heightfield, one large box contains the entire field, and then one can build a hierarchy of bounding volumes containing a tree of bounding volumes containing gradually decreasing sections of the field. The bounding volume must, however, contain the whole of all geometric primitives it is intended to contain.

2.4.3 Different Types of Hierarchies

To construct the hierarchy of bounding volumes, there are several different techniques that are popularly used. Choice in technique is a tradeoff between time spend to construct the hierarchy vs. time to trace. Technique should be chosen such that construction + rendering time is minimal. For scenes that need to be re-rendered but do not change, or do not change much/often, constructing a very expensive hierarchy is typically preferable, while for scenes that rapidly change the

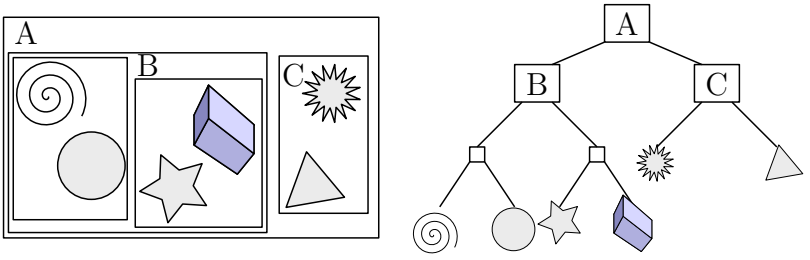


FIGURE 2.4: An example of a bounding volume hierarchy. [Sch11]

hierarchy should be simple. Some frameworks, such as NVIDIA's *OptiX*, take care of constructing such a hierarchy, so that the programmer only needs to specify type with a string such as "Bvh" or "Sbvh" without any knowledge of how the hierarchies actually work. As we will make us of OptiX, and not implement such hierarchies ourselves, further details on these will not be discussed.

2.5 Intersecting Objects

2.5.1 The ray equation

Intersecting a bounding volume is very simple, but doesn't tell much about intersecting the actual objects that it contains. Once a bounding volume is intersected, we must use whatever we know about the object to determine an intersection as well as the exact point of intersection. The basic idea when intersecting a ray is to use the ray equation to determine when $\mathbf{p} = (x, y, z)$ intersects some equation describing a surface. The ray equation in 3 dimensions is:

$$\mathbf{p} = \mathbf{p_0} + t * \mathbf{d} \tag{2.1}$$

Where $\mathbf{p}$ gives positions on the ray, $\mathbf{p_0}$ refers to the starting point of the ray, t is the variable parameter, and $\mathbf{d}$ is the direction. This gives a parametrization of x, y, and z on t:

$$\begin{aligned} x &= x_0 + d_x t \\ y &= y_0 + d_y t \\ z &= z_0 + d_z t \end{aligned} \tag{2.2}$$

For simple objects like spheres we can use the equation for the surface of a sphere, and for a box we can use equations for rectangles for each side, to determine when $\mathbf{p}$ is a point on the surface (if ever). In the remainder of this section we will introduce intersection of Spheres, Triangles, Heightmaps, and a quadratic interpolation of heightmap cells.

2.5.2 Intersecting a sphere

Arguably the simplest intersection, intersecting a sphere takes fairly few floating point operations and is solved by a simple quadratic equation. For most readers the equation for the surface of a sphere should be familiar:

$$x^2 + y^2 + z^2 = r^2 \tag{2.3}$$

For a sphere centered at the origin, we only need to insert the ray parametrization and expand to get a quadratic equation:

$$(d_x^2 + d_y^2 + d_z^2)t^2 + 2(d_x x_0 + d_y y_0 + z_0 d_z)t + x_0^2 + y_0^2 + z_0^2 - r^2 = 0 \tag{2.4}$$

with $a = d_x^2 + d_y^2 + d_z^2$, $b = 2(d_x x_0 + d_y y_0 + z_0 d_z)$, and $c = x_0^2 + y_0^2 + z_0^2 - r^2$. Expressed in terms of vectors the equation can be rewritten to:

$$(\mathbf{d} \cdot \mathbf{d})t^2 + 2(\mathbf{d} \cdot \mathbf{o})t + \mathbf{o} \cdot \mathbf{o} - r^2 = 0 \tag{2.5}$$

Where $\mathbf{d}$ is the ray direction, and $\mathbf{o}$ is the ray origin. Clearly, $a = \mathbf{d} \cdot \mathbf{d} \equiv 1$, which is great for numerical accuracy as it means the solution to the quadratic equation will *never* divide by zero or near-zero values.

For a sphere not centered at the origin, we can simply subtract the sphere center from the ray origin to shift the scene such that the sphere becomes at-origin. That is, set $\mathbf{o} = \mathit{ray}.\mathbf{o} - \mathit{sphere}.\mathbf{o}$. Once the quadratic equation is solved, typically we have two possible values of t, and the smallest non-negative value of t can be chosen as the point of intersection. The surface normal is then simply the (normalized) vector from the center to the point of intersection. A complex result to the quadratic equation implies no intersection.

2.5.3 Intersecting a Triangle

To calculate the intersection between rays and triangles, we use the algorithm given by [Sun14], which bases itself on barycentric coordinate computation. Intersecting a triangle is akin to intersecting a plane, but with more work involved. Unless parallel to the plane, the ray will eventually intersect the plane containing the triangle in some point $\mathbf{p}_1$. The point $\mathbf{p}_1$ can be found by the following two equations:

$$\begin{aligned} r &= \frac{\mathbf{n} \cdot (\mathbf{v}_0 - \mathbf{p}_0)}{\mathbf{n} \cdot \mathbf{d}} \\ \mathbf{p}_1 &= \mathbf{p}_0 + r \cdot \mathbf{d} \end{aligned} \tag{2.6}$$

Where $\mathbf{v}_0$ is any point on the plane, $\mathbf{p}_0$ is the origin of the ray, $\mathbf{n}$ is the plane normal, and $\mathbf{d}$ is the direction of the ray.

Once the point $\mathbf{p}_1$ has been calculated, the next step is to determine whether this point is inside the triangle. To do this one can express the plane in terms of the sides of the triangle, and consider that any two pairs of sides are linearly independent vectors that span the plane:

$$\mathbf{p}_1 = \mathbf{t}_0 + x_1 \mathbf{u} + x_2 \mathbf{v} \tag{2.7}$$

Where $\mathbf{u}$ and $\mathbf{v}$ are any two sides of the triangle, x_1 and x_2 are the parametrization variables, and $\mathbf{t}_0$ is the point shared by $\mathbf{u}$ and $\mathbf{v}$. Trivially, for the point $\mathbf{p}_1$ to lie in the triangle, both x_1 and x_2 must be nonnegative and less than or equal to 1. Less obvious is that the sum $x_1 + x_2$ must be less than or equal to 1. This yields the following equation:

$$\mathbf{p}_1 - \mathbf{t}_0 = \mathbf{w} = x_1\mathbf{u} + x_2\mathbf{v} \tag{2.8}$$

Which can be solved to:

$$\begin{aligned} x_1 &= \frac{(\mathbf{u}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{v}) - (\mathbf{v}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{u})}{(\mathbf{u}\cdot\mathbf{v})^2 - (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})} \\ x_2 &= \frac{(\mathbf{u}\cdot\mathbf{v})(\mathbf{w}\cdot\mathbf{u}) - (\mathbf{u}\cdot\mathbf{u})(\mathbf{w}\cdot\mathbf{v})}{(\mathbf{u}\cdot\mathbf{v})^2 - (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})} \end{aligned} \tag{2.9}$$

2.5.4 Intersecting a Heightmap

Very relevant to this thesis, which concerns tracing rays in terrain, being able to intersect a heightmap is essential if we want to be able to ray trace a scene that has a natural ground and not just a flat floor. The form of heightmap that will be considered here is a 2 dimensional buffer containing height data for coordinates that are implicit in the indexes and thus equally spaced, forming square cells.

Intersecting a heightmap is very straightforward. Once the lowest level of the bounding volume hierarchy is reached, we calculate the entry point and then simply traverse the cells contained within the bounding box, following the direction of the ray, checking each for intersection. When traversing the cells of the heightmap, it is useful to convert positional data from universal coordinates to coordinates into the heightmap:

$$\mathbf{p_m} = \frac{\mathbf{p} - \mathbf{b_0}}{|cell|} \tag{2.10}$$

where $\mathbf{b0}$ is the global coordinate of the $(0, 0)$ index of the heightmap, $|cell|$ gives the width of a cell, and $\mathbf{p_m}$ becomes the coordinate into the heightmap. Once $\mathbf{p_m}$ has been calculated, indexing into the heightmap is as easy as taking the floor of the coordinates, and finding the position at which the ray exits the current cell (and thus the next cell to test) is as simple as finding the next point at which either of the x or z components of $\mathbf{p_m}$ change integer value. Note that the height coordinate is not altered in $\mathbf{p_m}$ from $\mathbf{p}$, as the heightmap gives height in global values and is not relative.

Now that we can easily traverse the cells, it's time to test them for intersections. To do this, we must first decide upon what type of geometry we want to use to join all 4 corners of the cell. It is very rare that a simple plane quadrilateral is able to describe all 4 corners, so we must make up something more complex. One way to join all 4 corners is to use a quadratic equation to create a curving surface,

or for more simplicity we can just divide the cell into two triangles. Thereafter, it is simply to use relevant geometric equations to determine intersection and point of intersection.

One shortcut that can be noted when intersecting a cell, is that it can be skipped if a simple conditions is met: The ray starts and finishes above, or below, the cell. This approach of making an implicit bounding box can often be useful and save computation time.

2.5.5 Quadratic interpolation of a heightmap

An alternative to triangles is, as mentioned, a quadratic interpolation. The key behind a quadratic interpolation is to note that a line can be drawn joining two opposite sides of a quadrilateral that is also parallel to the adjoining sides. By making a surface that consists of an infinitely dense number of these lines the surface will be continuous and smoothly curving.

The next question is of course how to intersect this surface. To do so we express the height Y of the heightmap in terms of the XZ coordinates, and then insert the ray equation. Consider an axis aligned cell with bottom left corner in $(0, 0)$, sides of length 1, and heights c_{00}, c_{10}, c_{01}, and c_{11}. We let the lines go between the two sides aligned to the x axis, the heights h_1 and h_2 of the two ends of any one line can thus be expressed as:

$$\begin{aligned} h_1(x) &= c_{00} + (c_{10} - c_{00}) * x \\ h_2(x) &= c_{01} + (c_{11} - c_{01}) * x \end{aligned} \tag{2.11}$$

At any point (x, z) in the cell the height can then be expressed as:

$$y(x, z) = h_1(x) + (h_2(x) - h_1(x)) * z \tag{2.12}$$

An intersection can then be found by calculating at which point, if any, does Equation 2.12 hold true if we insert the ray-parametrization for x and z on t into Equation 2.12, and set the left-hand side to the ray-paramtrization of y on t. Note that we redefine x_0, y_0, and z_0 as the entry points in into the cell (not the ray origin), and scale d_x, d_y, and d_z by the size of the cells.

The bulk of the resulting equation manipulation will not be shown here as the equation grows quite sizable, but in the end we get the following quadratic equation:

$$d_x d_z y_d t^2 + (d_x y_b + d_z y_c + (x_0 d_z + z_0 d_x) y_d - d_y) t + y_a + x_0 y_b + z_0 y_c + x_0 z_0 y_d - y_0 \tag{2.13}$$

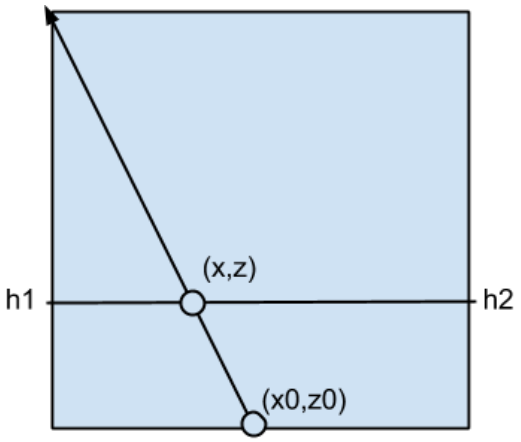


FIGURE 2.5: Any point in the cell can be described using h_1 and h_2.

Where $y_a = c_{00}$, $y_b = c_{10} - c_{00}$, $y_c = c_{01} - c_{00}$, and $y_d = c_{11} - c_{01} - c_{10} + c_{00}$. This is then solved as any regular quadratic equation, with $a = d_x d_z y_d$, $b = d_x y_b + d_z y_c + (x_0 d_z + z_0 d_x) y_d - dy$, and $c = y_a + x_0 y_b + z_0 y_c + x_0 z_0 y_d - y_0$.

Note that if the height values of a cell perfectly represent a plane, then the value of y_d and a will become 0. If this is the case then, rather than solving the quadratic equation (which would lead to a division by 0), the quadratic term may be dropped and the equation solved as a linear equation.

2.6 Geometry

In this section, a brief review of some geometric concepts and formulas will be presented.

2.6.1 Dot Product

The dot product of two vectors is defined as:

$$\mathbf{u} \cdot \mathbf{v} = \sum_{i=0}^{d} u_i v_i \tag{2.14}$$

However, more important to this thesis, is the geometric definition:

$$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \|\mathbf{v}\| \cos\theta \tag{2.15}$$

Where $\|\mathbf{u}\|$ is the *magnitude* or *length* of $\mathbf{u}$, and θ is the angle between the two vectors. For unit vectors, this reduces to:

$$\hat{\mathbf{u}} \cdot \hat{\mathbf{v}} = \cos\theta \tag{2.16}$$

2.6.2 Cross product

The cross product is defined as:

$$\mathbf{u} \times \mathbf{v} = ((u_2v_3 - u_3v_2), (u_3v_1 - u_1v_3), (u_1v_2 - u_2v_1)) \tag{2.17}$$

However, like the dot-product, the geometric definition is more interesting.

$$\mathbf{u} \times \mathbf{v} = \|\mathbf{u}\|\|\mathbf{v}\| \sin\theta\hat{\mathbf{n}} \tag{2.18}$$

Where $\hat{\mathbf{n}}$ is the normal for the plane spanned by the two vectors. In this thesis the most interesting use for the cross product, is expressing the plane spanned by two vectors by acquiring the normal by normalizing the right-hand side of Equation 2.17

2.6.3 Rodrigues' rotation formula

Rodrigues' rotation formula[wik14b] is an efficient formula for rotating any vector $\mathbf{v}$ in 3 dimensions given an axis $\mathbf{k}$ to rotate around:

$$\mathbf{v}_{rot} = \mathbf{v}\cos\theta + (\mathbf{k} \times \mathbf{v})\sin\theta + \mathbf{k}(\mathbf{k} \cdot \mathbf{v})(1 - \cos\theta) \tag{2.19}$$

2.7 GPUs

In this section programming GPUs and the CUDA platform will be given a brief introduction. Ever since classical serial processors began to meet the power wall, massively parallel processors such as the GPU have gained enourmous grounds on the CPU in terms of potential number of operations per second (typically measured in floating point operations per second, FLOPs). It is therefore often desirable to perform computationally heavy tasks on the GPU, rather than the CPU. This difference in performance comes from the difference in architecture. Since CPUs are usually designed for high serial performance, they become very complex in order to find ways around the power wall, and so few cores fit on each chip, and they have a poor FLOP to watt ratio. Whereas the GPU, designed to run massively parallel tasks, is not concerned about serial performance, can

therefore be much simpler, and run at lower clock rates. This simplicity, and low power consumption, allows for many cores, each of which is typically a vector processing unit.

2.7.1 Programming GPUs

As a result of the architectural difference, programming a GPU is not as simple as programming a CPU. GPUs work by the SPMD taxonomy, distributing a single program across multiple cores. Each of these cores are in turn SIMD vector processors, and it is appropriate to treat the GPU as if it were a plain vector processor. Vector processors operate with data parallelism, performing the same operation on thousands to millions of datasets at a time. It is a natural consequence of this that for best performance, a GPU program should operate without divergence. In addition, the massive parallelism would require a massive memory bandwidth should only a few operations be performed for every dataset loaded from memory, and the programmer should therefore take care to tailor his program to the underlying memory system and the primary candidate for optimization becomes that of memory usage.

2.7.2 CUDA - Compute Unified Device Architecture

Utilizing GPUs for computation requires some platform to program them. Different GPU vendors have historically only used their own proprietary and unportable programming platforms (such as AMD's Compute Abstraction Layer (CAL) [AMD10] and NVIDIA's Compute Unified Device Architecture (CUDA) [NVI14a]), while most now support the OpenCL platform. NVIDIA continues to primarily support their CUDA platform, whereas AMD has entirely abandoned their CAL platform in favor of OpenCL. This thesis builds on a snow simulator, implemented in CUDA, and makes use of NVIDIA's OptiX framework, which exists only on CUDA, and will therefore make use of the CUDA platform.

CUDA, first released in 2007, is a programming environment consisting of the NVCC compiler, libraries, debugging and profiling tools, and plugins for some IDEs. CUDA programming is C++ with extensions, where the host code can be written freely as C++, while the device code is written primarily as C with a few C++ features (such as Templates). The NVCC compiler then compiles the host-side code using the available C++ compilers (such as GCC), effectively just acting as a wrapper for host-side code, whereas device-side code is compiled down to device binary code. NVCC can also, akin to most C/C++ compilers, stop the compilation of device-side code at the assembly stage and write this to a file.

This is important for OptiX programming (see Section 2.8 below), as the OptiX framework builds its program objects from .ptx files.

2.8 NVIDIA OptiX

In this section NVIDIA's ray tracing framework, OptiX[NVI12], will be introduced. NVIDIA's OptiX is a framework for programming ray tracing applications on CUDA devices [NVI14b]. It builds on an observation that most ray tracing implementations build on the same overall process, of which OptiX implements all invariant steps and takes care of most lower level details. OptiX does not itself deal with any rendering, and is therefore suitable for general purpose ray tracing, both online and offline rendering, as well as any ray tracing that is not actually used for rendering.

2.8.1 The OptiX Pipeline

As previously mentioned, OptiX implements certain steps and takes care of lower level details, but leaves many steps of the ray tracing pipeline for the programmer to define. Which results in several subprograms that is to be programmed by the user, while OptiX connects this into a fully fledged ray tracer. The subprograms that is to be written by the user are the following eight, some of which there may be more than one in each ray tracer:

- **Ray Generator**: This program is responsible for defining the camera (or source of rays), creating each ray, initializing each ray's appropriate attributes (such as direction, and user-defined attributes like signal strength).
- **Bounding Box**: This program is responsible for defining the coordinates of two opposite corners of a box that (preferrably minimally) bounds an object in the scene.
- **Intersection**: This program is responsible for determining whether a given ray intersects a given object.
- **Closest Hit**: This program is called whenever a ray intersects an object, and that ray is looking for its *closest* hit, and the closest hit has been determined. This program then modifies any output buffers based on its ray's attributes and what it hit, possibly also launching any additional rays from the point of intersection.

- **Any Hit**: This program is called whenever a ray intersects an object, and that ray is just looking for any hit. This is usually only used by shadow rays which trace towards a light source trying to determine whether a point is in that light's shadow or not.
- **Selector Visit**: This program is called whenever a selector node is to be tested for intersection. It is responsible for determining which of the selector node's children is to be visited.
- **Miss**: This program is called when a ray misses all geometry, and is responsible for setting the output buffer to some default value (or using some environment map to determine the value).
- **Exception**: Called when an error occurs, such as max depth of recursion is reached and the OptiX stack is overflowing. This usually just sets an output buffer to some value and then retires.

2.8.2 OptiX Runtime

OptiX consists of one host-side API and one device-side API. The device-side API consists primarily of macros and a few functions (such as the function to report an intersection), and is otherwise writted as regular CUDA code. This CUDA code should then be compiled not to binary format, but to NVIDIA's *Parallel Thread Execution (PTX)* language. PTX is NVIDIA's assembly-like intermediate language that is later assembled into the lower level device-specific assembly language. As mentioned in Section 2.8.1 the user can define 8 different forms of programs to be used by optix. These are compiled into separate .ptx programs, but before execution they are compiled into a single monolith-kernel by OptiX, to eliminate recursion and unify variable references.

The host-side API is a C-based API used to initialize an OptiX context, load the aforementioned PTX files, define geometry, associate the various geometry with shaders, create device-side buffers, and load textures. A C++ wrapper to this C API also exists. The typical program flow of an OptiX application, is to first initialize its context, select what GPU device(s) to utilize, load all .ptx programs, declare and set global device-side variables, define all geometry and materials (associating bounding box and intersection programs with geometry, and closest and any-hit programs (shaders) with the materials), compile the geometry group, and associate a bounding volume hierarchy with said group. The scene is then ready to be ray-traced.

2.9 Related Work on Ray Tracing and NVIDIA OptiX

Plenty of articles have been written on both Ray Tracing and its predecessors. [FFBE13] uses Ray Casting to extract information from X-ray datasets. [Ped12] explores implementing Ray Tracing on the CPU as well as the GPU, and [TS09] gives an in-depth exploration of challenges faced and techniques used to implement an efficient ray tracer on the GPU. Ray Tracing on the GPU using NVIDIA's OptiX has already been explored by several students at NTNU: [Lud10, Nor13, Bab11, Lud09, Ped11].

Chapter 3

Radio Waves

This chapter presents physics related to Electromagnetic Waves relevant for this thesis, focusing primarily on the radio spectrum. An electromagnetic (**EM**) wave or electromagnetic radiation is a wave of radiant energy propagating through space in the form of photons. An EM wave can be characterized by the equation

$$v = f\lambda$$

Where v is the speed, f the frequency, and λ the wavelength. The frequency f is constant, meaning it *never changes* once the ray has been spawned, while the speed v may change depending on the medium of propagation, and the wavelength λ adjusts to maintain frequency.

In classical physics, EM Waves are thought to be created when charged particles, primarily electrons, are accelerated. The resulting frequency / wavelength is characterized by the speed of acceleration, yielding higher frequencies / lower wavelength waves with faster acceleration. The resulting wave is made up by three vector values: Direction, Electric field, and Magnetic field. The Electric and Magnetic fields are both perpendicular to the direction, and undulate with the frequency. These two fields can be expressed as:

$$\begin{aligned} E(x,t) &= E_0 e^{-i\omega t} \\ B(x,t) &= B_0 e^{-i\omega t} \end{aligned} \tag{3.1}$$

Where $\omega = 2\pi f$ is the angular speed, E_0 is the electric field strength, B_0 is the magnetic field strength, and E and B are the instantaneous field strengths that undulate with the frequency. [Fra05, Chap 10.1]

3.1 Frequency Bands

Typically, EM Waves are divided into different classes (spectral bands) depending on their frequency. The division into classes is based on qualitative differences caused by the differences in frequencies, such as the very well known class of *Visible light*, which humans eyes are able to sense directly. Typically EM waves are divided into the following 8 classes, sorted by ascending frequency[wik14a]:

1. Radio - Very low frequency EM waves that are typically used to transmit information over long distances, and is divided into several different bands depending on government regulations.
2. Microwave - Can be considered a subclass of Radio Waves, but are high frequency enough that they can be used for thermal heating (microwave ovens), while the lower frequencies of microwaves are typically used for Wi-Fi.
3. Terahertz - A not often considered class, rarely used in practice. But can theoretically be used in warfare (disabling electronics from afar).
4. Infrared - The class just below visible light, that is typically thermal radiation. Typical uses are night vision and astronomy.
5. Visible - The peak power region of the sun's emissions. Can be directly sensed by human eyes, and its main use is thus illumination.
6. Ultraviolet (UV) - Powerful enough to directly alter the chemical structure of molecules it hits, UV waves are infamous for causing sunburns as the UV radiation from the sun that is not absorbed in the ozone layer is just strong enough to break chemical bonds.
7. X-rays - Powerful enough to ionize atoms, and is able to pass through most substances and is often used in medicine to create X-ray images of a patient's body.
8. Gamma Rays - The highest frequency waves, gamma rays have no defined upper limit to their frequency, and can be weak enough for medical uses (radiation cancer therapy) or strong enough to create a shower of particle-antiparticle pairs upon interaction with matter.

In this Thesis only Radio waves and Microwaves will be considered, as these are typically the only types of EM waves where getting coverage over any terrain or geometry is interesting.

3.2 EM Wave Interactions with Matter

It is very important for this thesis to note how EM Waves will interact with surfaces they hit, lest the simulation be only a straightforward line-of-sight simulation. A few important ways in which waves will interact with surfaces they hit will be considered: reflection, refraction, and the attenuation inherent to propagating through a lossy medium.

3.2.1 Magnetic Permeability, Electric Permittivity, and Conductivity

When calculating refraction, reflection, and depth of penetration, three properties of the propagation medium are central: *Magnetic Permeability* (μ) measured in netwons per ampere squared($\mathrm{N{\cdot}A^{-2}}$), *Electric Permittivity* (ϵ) measured in farads per meter (F/M), and *conductivity* (σ) measured in siemens per metre (S/m). Henceforth simply permeability, permittivity, and conductivity. All calculations and other values considered will ultimately depend on these three values. For free space (vacuum) these three values are $\mu_0 = 4\pi \times 10^{-7}\mathrm{N{\cdot}A^{-2}} = 1.2566370614 \times 10^{-6}\mathrm{N{\cdot}A^{-2}}$, $\epsilon_0 = \frac{1}{c_0^2 \mu_0} = 8.8541878176 \times 10^{-12}\mathrm{F/M}$, and $\sigma = 0$ S/m. The permeability and permittivity of free space are important to note as the permeability and permittivity of arbitrary mediums are typically expressed in terms of their values *relative* to that of free space. For example, the permittivity of air at 1 atmosphere is expressed as $\epsilon = \epsilon_r \epsilon_0 = 1.00058986 \times \epsilon_0 = 8.859410548 \times 10^{-12}\mathrm{F/M}$, where ϵ_r is the relative permittivity of air, also named the *dielectric constant*. Similarly, μ_r is the relative permeability.

[Fra05, Chap 6.2 and 8.2]. Note that i choose the e_r and μ_r notation for the relative permittivity and permeability, whereas [Fra05] denotes them as κ and κ_m respectively.

3.2.2 Reflection

Reflection refers to the waves that are reflected away from the new medium when an EM wave changes medium of propagation, such as going from air into water. This is what causes mirror effects on the surfaces of various materials. There are two types of reflection, *specular* and *diffuse*. Diffuse reflection is when rays are reflected in all directions from the point of impact, and will not be considered in this thesis. Specular reflection is the more familiar type where every wave only gives origin to one reflection wave, which moves outwards with the exact same angle to the normal as that of the incident wave.

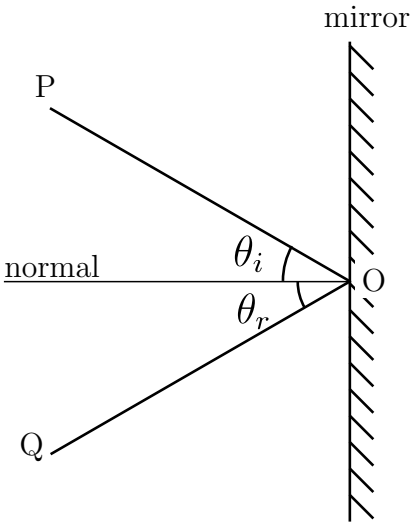


FIGURE 3.1: Reflection of a wave. [Arv05]

Specular reflection requires a certain degree of smoothness, relative to the wavelength of the incident wave, on the incident surface. One can safely consider a surface with average roughness less than one tenth of the wavelength as smooth. It is therefore often a safe simplification to treat all surfaces as smooth when considering radiowaves, while greater care must be shown for microwaves and higher frequencies.

Unless we have total reflection, the reflected wave will not be as powerful as the incident wave as a portion is lost to refraction. The reflection coefficient, that is the ratio of reflected strength to the incident strength, depends on the polarization of the wave and is expressed as:

$$R_v = \frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2 \cos\theta_t} \tag{3.2}$$

$$R_p = \frac{n_2 \cos\theta_i - n_1 \cos\theta_t}{n_2 \cos\theta_i + n_1 \cos\theta_t} \tag{3.3}$$

Where n is the refractive index of the medium, given by $n = \sqrt{\epsilon_r}$, and R_v and R_p are the reflection coefficients for Vertical and Parallel polarization, respectively [Fra05, Chap 10.4]. Note that the reflection coefficient may well be negative, and what this implies is that the phase of the reflected wave will be shifted by precisely half a period. See Section 3.3.

Polarization As mentioned the reflection coefficients depend on the polarization of the waves. The exact polarization of the waves is not taken into consideration in this thesis beyond the reflection coefficient.

3.2.3 Refraction

Refraction refers to the change in angle to the surface normal that occurs when an EM wave changes medium of propagation, again such as going from air into water. This is what gives the typical broken appearance of a straw when put into a glass of water. The new angle can be expressed in terms of Snell's law [Fra05, Chap 10.4.1]:

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n_2}{n_1} \tag{3.4}$$

$$\theta_2 = \arcsin\frac{n_1 \sin\theta_1}{n_2} \tag{3.5}$$

If the value of $\frac{n_1 \sin\theta_1}{n_2}$ is greater than 1 or less than -1, refraction yields total reflection (meaning the full strength of the wave gets reflected). [Fra05, Chap 10.4.6]

Naturally, unless there is neither specular nor diffuse reflection, the wave will not be as powerful as the incident wave post refraction. If we choose to disregard diffuse reflection altogether, we can define a refraction coefficient (ratio of strength post refraction to pre refraction) as simply being $R_f = 1 - |R|$, by the law of conservation of energy. Where R is the reflection coefficient and R_f is the refraction coefficient.

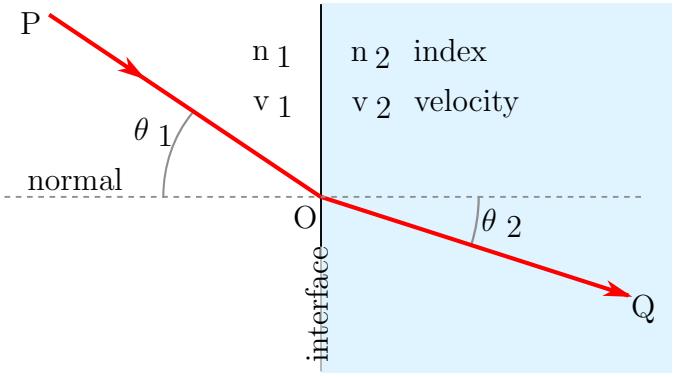


FIGURE 3.2: Refraction of a wave. [Cri06]

3.2.4 Depth of Penetration

Once refracted into a new medium, the strength of an EM wave attenuates at a pace depending on an attenuation rate, which depends on the new medium as well as the frequency of the wave. The *Depth of penetration*, δ of an EM wave is defined as the distance through the medium at which the signal will have fallen to $1/e$ of its original strength, where e is euler's constant. δ can be expressed by the following equation [PS02, Chapter 13.3]:

$$\delta = \frac{1}{\sqrt{\pi f \mu \sigma}} \tag{3.6}$$

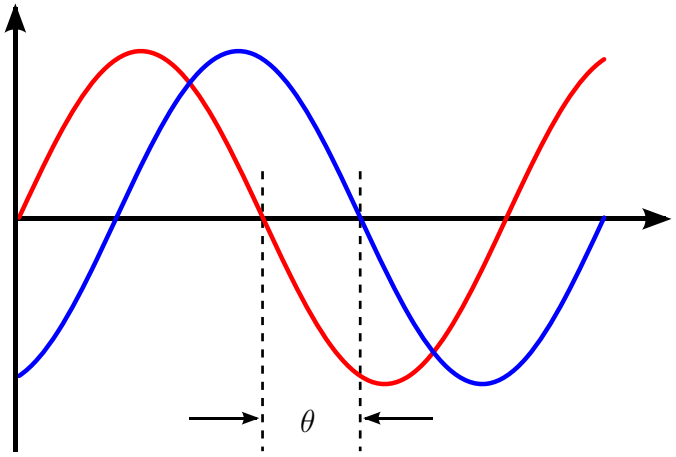


FIGURE 3.3: Two waves slightly out of phase. [Pep09]

Attenuation as the waves passes through a medium is exponential. Meaning that after passing 2δ into the medium, the wave will have attenuated to $1/e^2$ of its original strength. In free space, $\delta = 1/0$, which can be interpreted as not attenuating at all, except for standard free space loss. Attenuation as the wave passes through the medium can thus be modeled as:

$$e^{-d\alpha} \tag{3.7}$$

where $\alpha = \sqrt{\pi f \mu \sigma}$.

3.3 Phase

As is implied by Equation 3.1, electromagnetic waves oscillate over time. A wave with some frequency f has a period of $\frac{1}{f}$ seconds. As waves will oscillate between a positive an a negative value, two waves of the same frequency can cancel each other out or augment each other based on what phases they are in. If they are perfectly in phase, they perfectly augment each other and in the case of equipotent waves the field strength is doubled. Whereas in the opposite case, if they are perfectly out of phase (that is, out of phase by half a period), they cancel each other out, in the case of equipotent waves completely negating each other.

Waves that travel from the same source antenna are naturally perfectly in phase with each other when they leave the antenna. But waves reaching a receiver by different paths will be in different phases at arrival, depending on a variety of factors. The two most important sources of difference in phase are negative reflection coefficients (which shift phase by exactly half a period) and difference in path length. [Cla14]

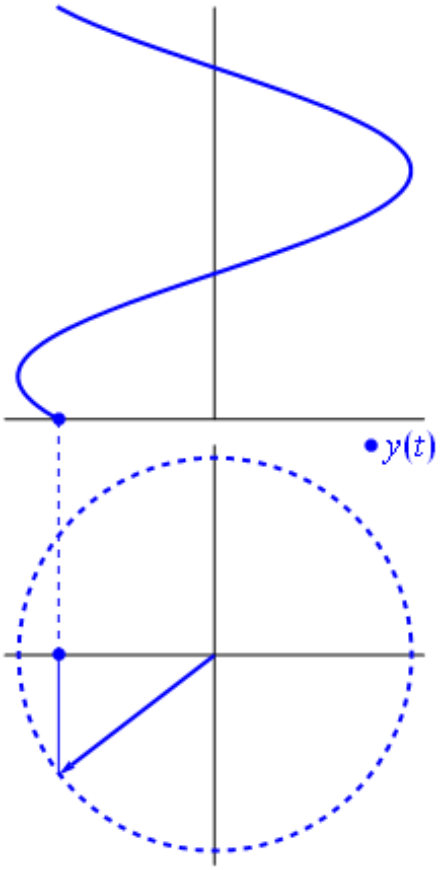


FIGURE 3.4: A wave can be seen as a vector rotating around origin in the complex plane[gon09]

As implied by the complex exponent of Equation 3.1 a wave of strength E and phase φ can be seen as a vector with dimensions $E \cdot (cos(t+\varphi), sin(t+\varphi))$ rotating around origin in the complex plane. When multiple waves are incident at the some receiver, the magnitude of the vector sum of all incident waves thus gives the signal strength, leading to a need to store only a single vector in the buffer accumulating all incident waves.

3.4 An Issue with Spherical Coordinates

In Section 2.3.2, we presented the usage of spherical coordinates to define a radio wave antenna. However, when tracing radio waves, this will lead to a very uneven distribution of waves, as seen in Figure 2.3. Waves will be *much* more dense in the area around poles than around the equator. This is because for every angle θ to the Y-axis the same number of rays will be spawned in a growing circle around the Y-axis. Including the very pole itself which will spawn all rays from the same spot and in the same direction.

This results in uneven accuracy in terms of ray density, forcing the usage of a much higher resolution of rays to achieve the same accuracy than would otherwise have been necessary. In addition, rays can not be given the same amount of power lest the poles be supercharged and the equator barely able to transmit.

To adjust for this, we can either make a much more complex geometric figure such as a massive polyhedron to give an approximately even distribution, spawn each wave at a random point on the sphere, or scale power by the surface area covered by each ray source point in spheric coordinates.

Scaling by surface area will still result in an uneven accuracy, due to uneven distribution of rays, but is easy to implement and still provides a useful simulation. Spawning random rays is not feasible because scaling the power of each ray will not be possible and the resulting random clusters will disturb the results. It is therefore ideal to construct some polyhedron. In this thesis, however, only the approach of scaling by surface area is considered as constructing a polyhedron is a much more complex task.

3.4.1 Determine Source Surface Area

The surface area A covered by a ray source point, can easily be determined by calculating the 4 neighboring source points and constructing a quadrilateral using curved lines that ensures that sides will be shared (so as to avoid subtracting from or adding to total signal strength). If the camera sphere is simplified to having a radius of 1 unit length, this gives a total area of 4π so that once the surface area A covered by the ray source is calculated the ray's strength is simply set to $\frac{A}{4\pi}E$. Where E is the total EM radiation energy produced by the transmitter.

The quadrilateral chosen for every point p is based on observing that the neighbors are two points that are equidistant from p in vertical rotation and two equidistant from p in horizontal rotation. Four lines that follow the sphere are then chosen as the middle ground between these and p. We now have two horizontal lines that both traverse an angle ϕ_1 and two vertical lines that both traverse an angle θ_1 in spherical coordinates. Straightforward integration over Spherical Coordinates can then give us the surface area. A visualization of these quadrilaterals is given in Figure 3.5.

In Spherical coordinates an area element is given by [wik14c]:

$$dA = r^2 \sin\theta d\theta d\phi \tag{3.8}$$

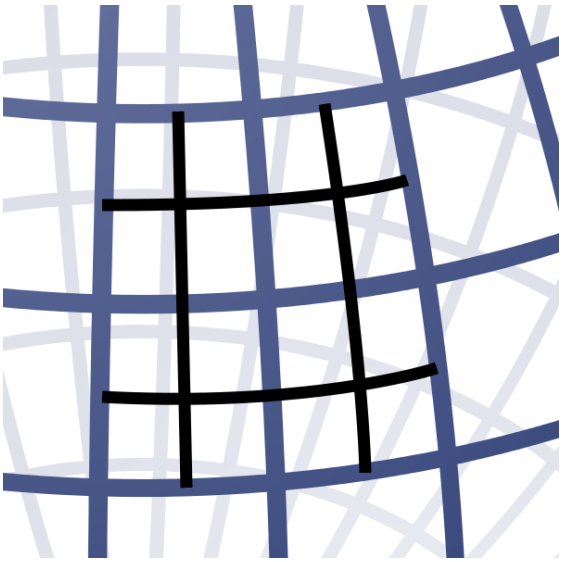


FIGURE 3.5: The surface area covered by a ray. Made by editing [Gee09]

Giving us the formula for surface area as:

$$A = \int_{\phi_0-\frac{\phi_1}{2}}^{\phi_0+\frac{\phi_1}{2}} \int_{\theta_0-\frac{\theta_1}{2}}^{\theta_0+\frac{\theta_1}{2}} r^2 \sin\theta d\theta d\phi \tag{3.9}$$

Where θ_0 and ϕ_0 are the spherical coordinates of p, and r is the radius (1). This integral solves to:

$$A = 2\phi_1 \sin\theta_0 \sin\frac{\theta_1}{2} \tag{3.10}$$

Which gives the poles at $\theta \in \{0, \pi\}$ as expected. However, on the poles the value becomes 0, which can be analytically seen as due to $\sin\theta_0$ changing sign. This can be remedied for the pole $\theta_0 = 0$ (and $\theta_0 = \pi$ by rotation), by taking the absolute value of sin in Equation 3.9 with $\theta_0 = 0$, to get:

$$A = 2\int_{\phi_0-\frac{\phi_1}{2}}^{\phi_0+\frac{\phi_1}{2}} \int_{0}^{\frac{\theta_1}{2}} r^2 \sin\theta d\theta d\phi \tag{3.11}$$

Which solves to:

$$A = 2\phi_1(1 - \cos\frac{\theta_1}{2}) \tag{3.12}$$

Where $\phi_1 = 2\pi$ if we have the poles spawn only one ray each.

3.5 Related Work on Ray Tracing for Signal Transmission

[SR92, Val93, TT95] describe using Ray Tracing to predict signal loss and delay

spread. An antenna is used as a ray source and the paths of rays are traced and recursively reflected and transmitted, until they either miss all geometry or are considered too weak. [SR92, Val93, TT95] do not generate any kind of coverage overview for usage afterwards, but instead register which rays actually hit a specified receiver and use the data to make a delay spread profile as well as to calculate expected signal strength. [TT95] also describes in detail how the dielectric properties of materials are considered, particularly the role of electric permittivity in calculating reflection and refraction coefficies, and the attenuation factor.

[CDB04] performs similar computations on a network of workstations. But in addition to reflection and refraction, also considers diffraction. In a city scene such as the ones used in the article, diffraction between structures would be important to consider.

[SMR07] describes a method of performing the computation efficiently when an assumption can be made that the ground is flat and all buildings perfectly vertical, by tracing rays in 2D and then transforming them to 3D when reflection occurs.

Chapter 4

Radio Signal Simulator Design

In this chapter methods and implementation details for our Radio Signal Simulator are given. We give the values of dielectric properties used, detail the propagation model used, present how the geometry was implemented (including both the terrain itself and the signal buffer which maps onto the terrain), and finally present the architecture of our software.

4.1 Dielectric Properties

Medium	Permittivity ($N \cdot A^{-2}$)	Permeability (F/M)	Conductivity (S/m)
Air	1.00058986	1.00000037	0
Snow	1.533	0.9999972	$1.3x10^{-8}$
Soil	6.0	1.0006	10
Rock	38.0	1.1	1

TABLE 4.1: Relative Permittivity and Permeability of the four mediums considered in this thesis.

Table 4.1 shows the relative permeability, relative permittivity, and conductivity of the four mediums considered in this thesis. The permeability for snow is an interpolation between ice/water and air, as snow is in fact a mixture of ice and air. The interpolation is chosen as 34% ice and 66% air, with the permeability of ice assumed to be close to that of water (0.999991). The remaining values are picked from [Sco04, MB01, Mä96, GAHT09, FFH04, Geo14], as well as Wikipedia articles on permeability, permittivity, and conductivity. Particularly for rock and soil, the values can vary widely and worst case values are picked for these. These values are defined in RayTracerDefines.cuh, and are used in RayTracerSignal.cu to calculate Reflection Coefficients, Refraction Angle, and Attenuation Rate according to Equations 3.2, 3.3, 3.5, and 3.6.

4.2 Propagation Model

After leaving the antenna to propagate towards the receiver, radio waves will travel in all directions from the antenna, enter various mediums, and reflect and refract both specularly and diffusively. All of these modes of propagation will entail both a change in spatial dispersion of waves, and rates of attenuation. Each wave start out from the source with a specific signal strength E.

Line of Sight Naturally, some waves might not intersect any geometry before hitting a receiver. These waves simply have their signal strength added directly to the receiver buffer.

Reflection The next propagation mode to consider is that of multi-path signals, such as by reflection. In this thesis only specular reflection will be considered. When reflected, some signal loss is incurred by part of the signal transmitting into the medium at whose interface reflection occurred, modeled by the reflection coefficient R. After n reflections, remaining signal strength can be expressed as

$$E \prod_i R_i \tag{4.1}$$

Note that the reflection coefficient is sometimes negative. This implies a shift in phase by half a period, which implies reflected waves will often act interferingly to LOS waves.

Refraction Finally, we consider refraction. Like reflection, transmitted waves are affected by a refraction coefficient, which can simply be added to Equation 4.1. However, transmitted waves are also affected by attenuation when transmitted into a lossy medium. From Equation 3.7 we can add attenuation to the equation:

$$E \prod_i R_i \prod_j e^{-d_j \alpha_j} \tag{4.2}$$

Where e is euler's constant, and d_j is the distance traveled through the lossy medium.

Free Space Loss? A reader with experience in the propagation of waves might have noticed that we are ignoring Free Space Loss. To understand why: Consider an antenna transmitting some E per surface unit, each wave transmitted from this antenna should represent the same E per surface unit. If two waves then hit the same receiver by the same path (for example, the line of sight path), the receiver

adds up $\frac{2E}{d^2}$ per surface unit, twice what it should actually receive. A better representation is therefore needed. We therefore set each wave to represent an E_r equal to $E \cdot A$ where A is the surface area represented by the wave on the antenna, calculated by Equations 3.10 and 3.12. We then simply add up the E_r of the wave at the receiver, and scale it down if Ad^2 (where A is the surface represented by a wave after propagating a distance of d) is larger than the surface of the receiver. The latter is not entirely accurate, but gives a decent approximation when the number of waves is large.

To further scale this by $\frac{1}{d^2}$ would no longer give strength per surface unit, but strength per surface area A, which is not a very useful metric and can not even be added up at the receiver as each wave has a different A.

Practical Implementation When implementing Equation 4.2, clearly it is not convenient to keep all the individual values of R_i, d_j, and α_i. The equation is therefore calculated progressively at each intersection as the factors R_i and $e^{-d_j \alpha_j}$ can simply be evaluated at each point, as the equation consists of only multiplication which is both associative and commutative.

4.3 Signal Coverage

Finally, the purpose of everything mentioned earlier in this chapter, is to make radio wave coverage calculation possible. A single antenna is defined by default, its position possible to adjust at run-time, and is then used as the origin for the *antenna* entry program. When an intersection has been made and signal strength should be added, it is added to an n by n buffer that is indexed based on the point of intersection in the XZ-plane. The number of rays to be spawned can be configured at run time, but the size n of the signal coverage buffer must be given before leaving the start menu.

Calculating the indices into the map When each ray makes an intersection, the coordinates of the intersection must be transformed into coordinates into the signal map. Since the terrain is defined to start at XZ coordinates $(0, 0)$, this is a simple matter of scaling by size and then rounding down:

$$\begin{aligned} I_x &= \lfloor \frac{S_x}{T_x} \rfloor R_x \\ I_z &= \lfloor \frac{S_z}{T_z} \rfloor R_z \end{aligned} \tag{4.3}$$

Where S is the dimensions of the signal map, T is the dimensions of the terrain, and R is the intersection coordinates into the heightmap.

Map entries Now that we have indexed our map, we want to store and retrieve signal strength from the buffer. At each index into the buffer we store a vector of two floating point numbers which represent x and y components of the accumulative vector sum of each incident ray (interpreted as a vector based on phase, according to the theory given in Section 3.3). Next we can take the magnitude of the vector to get the signal strength incident to the square.

However, this has one issue: A signal coverage map with lower resolution will have larger individual square and thus receive more signal strength. To adjust for this, the signal strength is divided by the area spanned by a single map entry in the xz plane, in order to get a normalized value (strength per square meter).

Receivers Actual receivers placed in the scene work like the coverage map. Only instead of a 2D buffer, we simply use a 1D buffer that is indexed by the immediately available variable *receiver_id*.

Extracting values from the map We know how to index and read values from the map, but how can the interface allow the user to read out the values from any given buffer? To do this we extend the host-side handling of controls to register left mouse button clicks, write what pixel was clicked to a device-side variable, and the thread corresponding to that pixel then reads out the value to a read-only field in the antTweakBar.

4.4 Defining the geometry

The heightmap defining the terrain of the Snow Simulator (Section 1.3) makes up virtually all geometry to be ray-traced. The vertice encoding of the Snow Simulator is abandoned and only the heightmap (with XZ coordinates left implicit) is used.

4.4.1 Defining the bounding boxes

As mentioned, the map is divided into larger parts that are each given their own bounding box. The bounding boxes are each given their own *bmin* and *bmax* values, which define opposing corners in an *axis aligned bounding box*. Since the box is treated as axis aligned by OptiX, these two corners are enough to define the

entire box. *bmin* defines the corner with the lowest coordinate values, and *bmax* the corner with the highest values. The XZ values of *bmax* and *bmin* are inferred directly from the bounds of the part of the heightmap covered by the box, whereas the Y values are found by taking the max and min values over the entirety of the relevant portion of the heightmap.

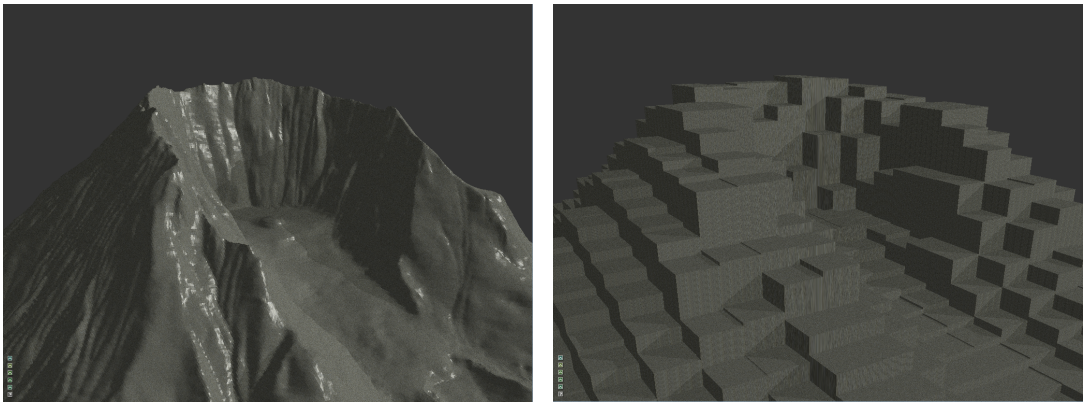


FIGURE 4.1: The bounding volumes cover the terrain.

4.4.2 Intersecting the heightmap

The techniques presented in Section 2.5 are employed here, but with some minor modifications to adjust for the Bounding Boxes: When a bounding box is intersected and the intersection program for the box's geometry is called, it's desirable to only intersect the heightmap contained within that bounding box. To do this the value t of the ray equation at which the ray enters and exits the bounding box are calculated first. Once they are calculated, the intersection loops over the relevant squares. To calculate t cleanly and easily, consider the following equations:

$$\begin{aligned} B_{min} &= R + T \cdot D \\ B_{max} &= R + T \cdot D \end{aligned} \tag{4.4}$$

Naturally, these equations only have an exact solution if the ray intersects the points *bmax* and *bmin* exactly. However, even if they don't, these equations can be used to find the values of t at which the ray will intersect the planes each of the 6 sides of the bounding boxes lie in if solved separately for the individual values X,Y,Z. Each of the planes are parallel to one of the XY, XZ, and YZ planes, giving us two of each, which will be referred to as the *near* and *far* planes by order of intersection. The t value of the first *far* plane intersection is the t value at which the ray leaves the box, and conversely the last intersection of a *near* plane gives the value of t at which the ray enters the box. If the t value of entry is higher than that of exit, then clearly the ray does not intersect.

This assumption is broken by rays moving parallel to planes, and rays are therefore prevented from having directions exactly parallel to any axis plane (by adding a very small value to the angle used). Although OptiX only calls intersection on a bounding box if it is in fact intersected, the latter modification is still important as rays parallel to an axis plane implies one or more 0-valued direction component, giving a division by 0 when solving Equations 4.4.

4.4.3 Compensating for limited floating point precision

Referring back to Section 2.5, in some cases floating point operations do not provide sufficient accuracy to detect an intersection. In this section the adjustments that have been made to compensate for floating point inaccuracies are presented.

Falling off the edge When approaching the edge of a triangle (or any geometry figure), round-off errors will *often* cause an intersection to (incorrectly) not be detected due to roundoff errors following the number of floating point operations required. In this case, the missed intersection is necessarily very close to the edge. Therefore we simply test whether the ray lies on the other side of the surface at the exit point of a square, compared to the entry point, and register an intersection at the exit point.

Near-zero values In Section 2.5 the case where $y_d = 0$ is already handled, but if y_d or the resulting value of a becomes extremely small, then the division when solving the quadratic equation will become highly inaccurate. We therefore switch to the linear equation at some predefined minimum value, rather than only at precisely 0. In both the quadratic and the linear equation, if the ray enters a completely flat or near-flat cell on an y-value that is very close to the surface, or by some other circumstance gets a near-zero value of c, the equation will solve to a potentially even smaller value depending on the values of b and a, and the solution will again become highly inaccurate (on a perfectly flat cell it will solve to 0 no matter how far away from the camera the intersection actually is). The ill effects of this can cause entire cells to vanish, if those cells happen to also define the max y-values for their bounding boxes (at which point the intersection will always have an entry on the surface!). To compensate for this we simply register an intersection at the entry point when c is sufficiently small.

When all else fails Sometimes all of the mentioned measures will still fail to detect an intersection, as witnessed by sections of of the map that are clearly behind blocked paths receiving signals (also when disabling reflection and refraction). A final adjustment to fix even these cases, is to immediately fake an intersection

if the ray is detected to be on the "wrong" side of the heightmap, raising also a flag to indicate to the shader that the intersection is fake. The reason to fake an intersection, rather than terminate that ray or just returning from the intersection, is that intersections are not guaranteed to be made in ascending order of distance. Terminating the ray could therefore prevent an earlier valid intersection from being made, and simply returning would leave no indication for OptiX that all intersections of greater distance are invalid (as OptiX has no option to reduce a ray's maximum distance except by intersection).

4.5 Software implementation

A renderer and a radio signal simulator were implemented using the OptiX framework. This section will detail how this was implemented. Section 4.5.1 summarizes the development environments used, Section 4.5.2 will detail the host-side of the implementation, while Section 4.5.3 will detail the device-side implementation.

4.5.1 Implementation environment

The code was written using KATE, the KDE Advanced Text Editor, and compiled using `gcc` and NVCC. The build environment was changed from the snow simulator's Makefile into a CMake build specified by CMakeList.txt. The code should thus be possible to build on any platform, but only Linux has been tested.

4.5.2 Host-Side

The host-side API of the OptiX framework has both a procedural C API and an object oriented C++ API. Although the snow simulator is written in C++, the procedural C API was used, as this had a far more thorough documentation than the object oriented C++ API, and the author is more experienced writing C than C++.

As a consequence of this, the source file for the host-side implementation is highly procedural and only minor C++ extensions are used. I.e. it is written as C with extensions. The code is not written to allow the existence of multiple ray tracing contexts, or thread safety, and many variables are therefore written as static globals. The code itself is divided into the globally callable initialization function rtInit and the main function rtTrace, as well as several static and some global control functions for interactivity. All functions are prefixed by rt-, same as the OptiX API.

The function rtTrace is called once every frame to perform a new ray trace of the scene, and rtTrace again calls a control function that handles keyboard interactivity (such as moving the camera when certain buttons are pressed), as well as a call to rtTraceSignal. rtTraceSignal will perform the forward-ray-trace to calculate signal coverage, but only if the relevant flag has been set. If the flag is set, a call onwards to rtContextMultiLaunch2D is called. The flag is set when the user clicks the Trace button in the left-hand menu.

The function rtContextMultiLaunch2D is a wrapper function around OptiX's rtContextLaunch2D, which launches only a small portion of the rays at a time, where OptiX's rtContextLaunch2D would launch all at once. The purpose of this function is to give feedback on progress as well as limit the execution time of a single call. The latter prevents the GPU's watchdog from terminating the simulation after 5 seconds when run on a device also used as video-out, which would greatly limit the accuracy obtainable. It also serves to limit the impact on machine interoperability while the call is running.

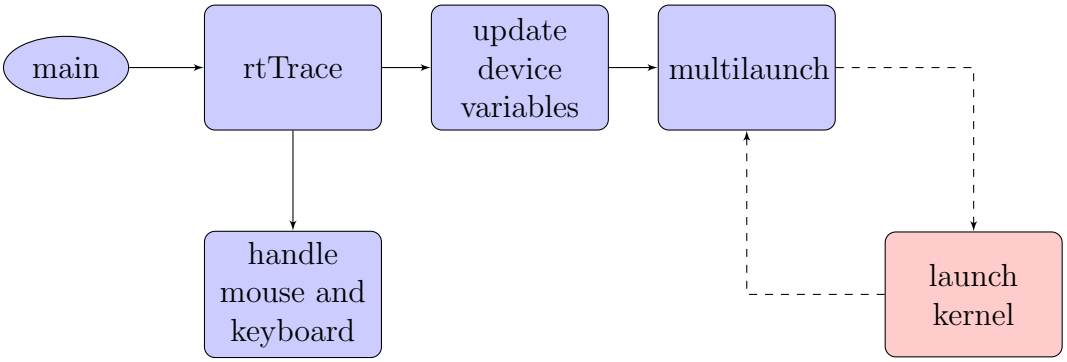


FIGURE 4.2: Rough diagram for the function rtTrace. Cloud indicates outside caller, blue block is host-side and red block is device-side.

Equally important as rtTrace is rtInit. rtInit is responsible for initializing all host and device side variables and buffers, as well as to construct the bounding volume hierarchy mentioned in Sections 2.4.1 and 2.4.2, and to load in the device-side code, build programs from these, and associate them with the relevant materials and geometry. The bounding volume hierarchy constructed is very simple. The landscape is cut into an m by m grid, each square thus making up an n/m by n/m heightmap, and each of these are placed in a bounding box based on the maximum and minimum height. One RTGeometry and RTGeometryinstance pair is created for each of these boxes, and are then used as child nodes for the larger RTGeomtrygroup. As the geometry only needs to be constructed once, the more expensive *sbvh* builder is then used to create the acceleration structure. The receiver sphere is also constructed in this manner, with one RTGeometry and RTGeometryinstance pair added to the RTGeometrygroup.

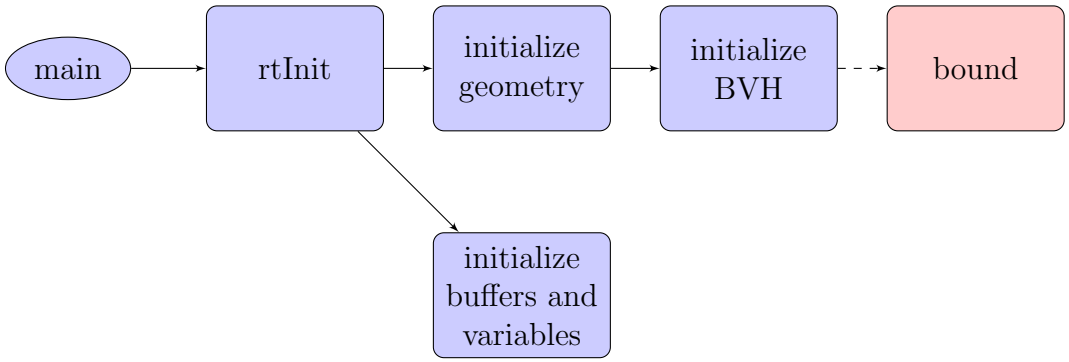


FIGURE 4.3: Rough diagram for the function rtInit. Cloud indicates outside caller, blue block is host-side and red block is device-side.

4.5.3 Device-Side

The entry points to the device-side calculations are the three functions defined in RayTraceCameras.cu. These calculate a ray direction based on launch index and then spawn a ray. The three available entry points are *env*, *pinhole*, and *antenna*. *Pinhole* is naturally akin to a pinhole camera, whereas *env* and *antenna* launch rays in all directions from the origin (using polar coordinates). *Antenna* is the entry point for signal coverage calculation, and uses the variable rtTotal instead of launch dimensions, and adds the variable rtStart to launch index, to cope with the multilaunch approach, and finally calculates each ray's source surface area according to Equations 3.10 and 3.12 and uses these to determine the signal power carried by this specific ray. The variable rtTotal naturally contains the actual total number of rays to be launched over each spherical coordinate, and rtStart the base index to be used in each launch.

These three function spawn two different types of rays: *Env* and *pinhole* spawn *radiance* rays, while *antenna* spawns *radio* waves. Radiance waves contain information about colour and relevancy, which are added to an output buffer once tracing is finished. Radio waves contain propagation data for the wave, such as remaining strength and total distance covered, and is discarded once tracing is finished and is instead added to an output buffer upon each intersection (according to the theory given in Section 3.3).

The next step on the device-side is traversing the bounding volume hierarchy, but this is handled by OptiX. Once OptiX has found volumes that could potentially intersect, a call will be made to the relevant intersect program to intersect the object. In this project there are only the two intersection funtions, one to intersect a heightmap and one to intersect a sphere. The former function simply traverses the heighmap within the bounding box, testing for intersection, applying the theory given in Section 2.5. This function exists in two variants, one where the ray may only exist above surface, and one where it may only exist below. This is made to allow a simple solution to floating point inaccuracies, by reporting leaving a square

in the heighmap on the "wrong" side as an intersection on the edge (see Section 4.4.3). The latter function, for intersecting a sphere, simply solves Equation 2.5.

Once an intersection is found, it is reported and OptiX calls the relevant shader function. For the visualization rays, a very simple shader is called. This shader merely loads a texture, calculates diffused lighting based on the angle between the surface normal and the light source, and then adds shadow waves on top. If signal coverage visualization is enabled, the shader is extended by adding signal strength as extra lumination. An alternative visualization mode, labeled 'simple', uses only signal strength as lumination. For radio waves, the function *signal_shader* is called when the heightmap is intersected, whereas *receiver_shader* is called when a receiver is intersected. *signal_shader* is naturally not an actual shader but only logic to determine reflection, refraction, attenuation, and to add coverage strength to wherever it hit. Reflection, refraction, and attenuation are calculated based on signal frequency and the dielectric properties of the material the wave propagated through. *receiver_shader* is very similar but does not define recursive waves (i.e. does not determine reflection and refraction), and adds signal strength to the receiver's own buffer instead of the signal map.

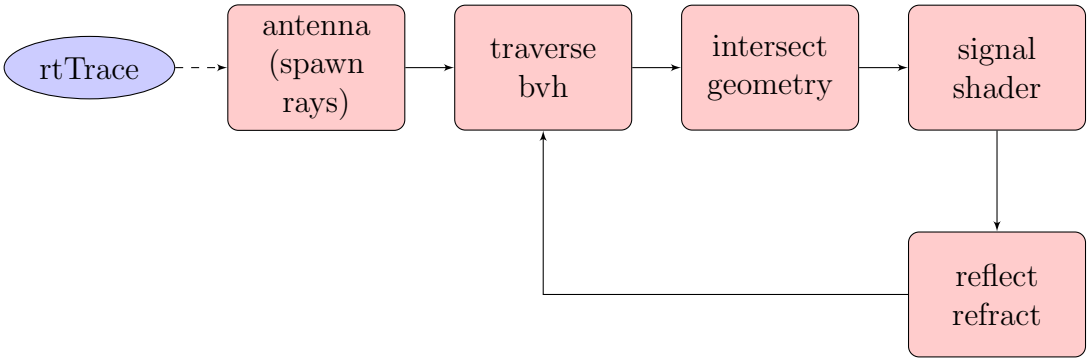


FIGURE 4.4: Rough diagram for the ray tracing kernel. Cloud indicates outside caller, blue block is host-side and red block is device-side.

Chapter 5

Results & Discussion

In this chapter the results of the work done are presented. Screen captures from the application made are analyzed, some verification of the results is given, and performance is briefly evaluated.

5.1 Test scenes

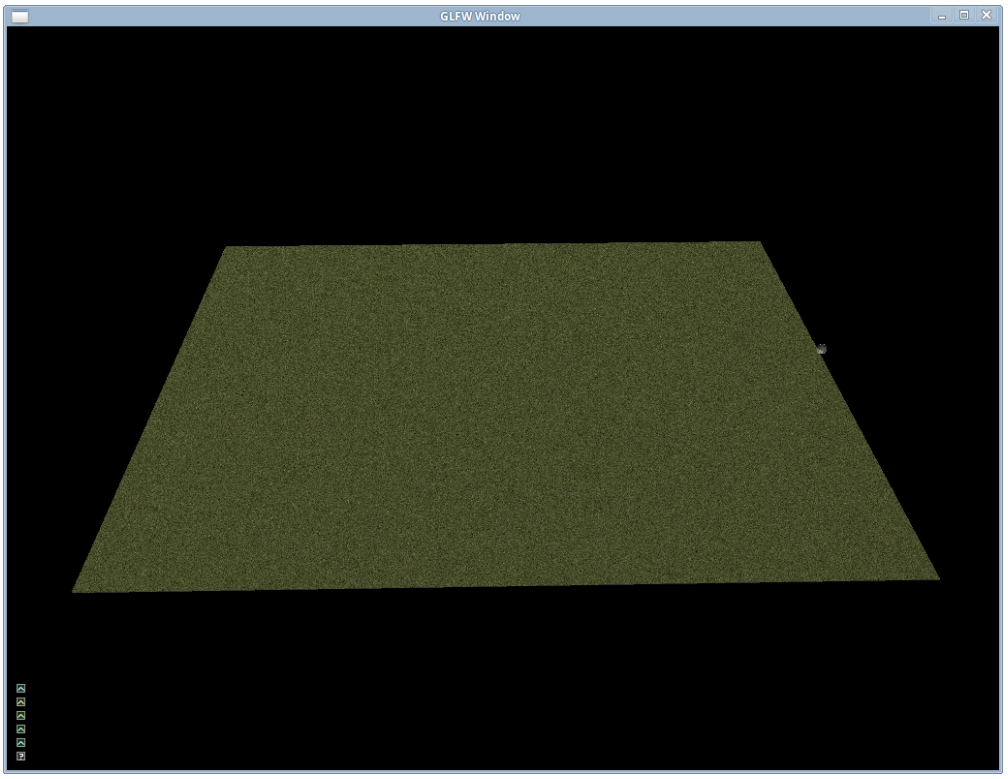
The implementation was tested in four different scenes: Flat ground, flat ground with walls, Mount St Helen, and an unnamed heightmap. The purpose of these scenes is to verify/present the effects of reflection and refraction, and demonstrate the resulting interference, as well as to give a few examples of how the result looks in more natural scenes. In the rest of this section the four scenes will be presented:

Flat ground The natural example to demonstrate line of sight and reflection effects. This scene is nothing more than a completely flat ground of grass. The purpose of this scene is nothing more than to see the result of line of sight signals, as well as the result of adding signals reflected off of the ground to the line of sight signals, and compare this to the results of hand-calculating the results.

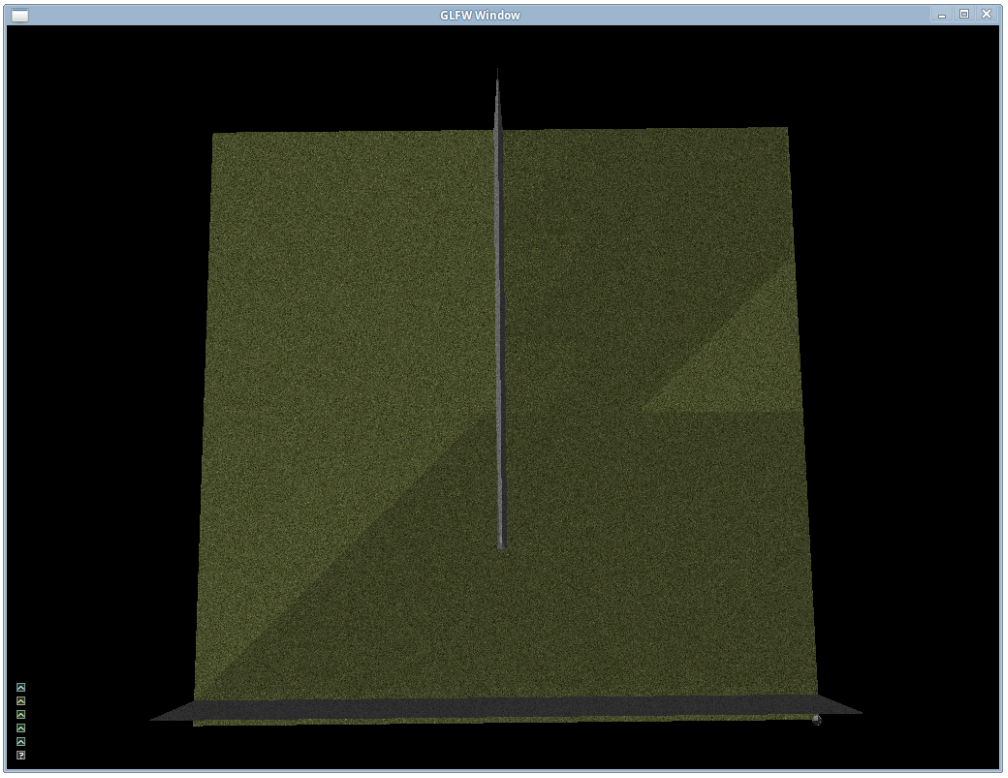
Flag ground with walls This scene is nothing more than the flat ground scene with two walls placed in it. The purpose of this scene, is to verify the results of refraction, and that of interference caused by reflection. For interference, the precise numbers are not verified by calculation as that would require more hand-calculation than the author was willing to do, but is instead verified purely by the visual results (Figure 5.0).

Mount St Helen The default scene for the HPC-Lab Snow-Simulator. This scene is a heightmap resembling Mount St Helen after the eruption of 1980. It consists of a volcano with a massive crater in the center, with the north side completely blown out, as well as parts of some surrounding hills. In this scene it is mainly interesting to demonstrate how the signal behaves in and around the crater, and to demonstrate how difficult it is to give coverage to the whole scene with an antenna placed in, on, or around the mountain.

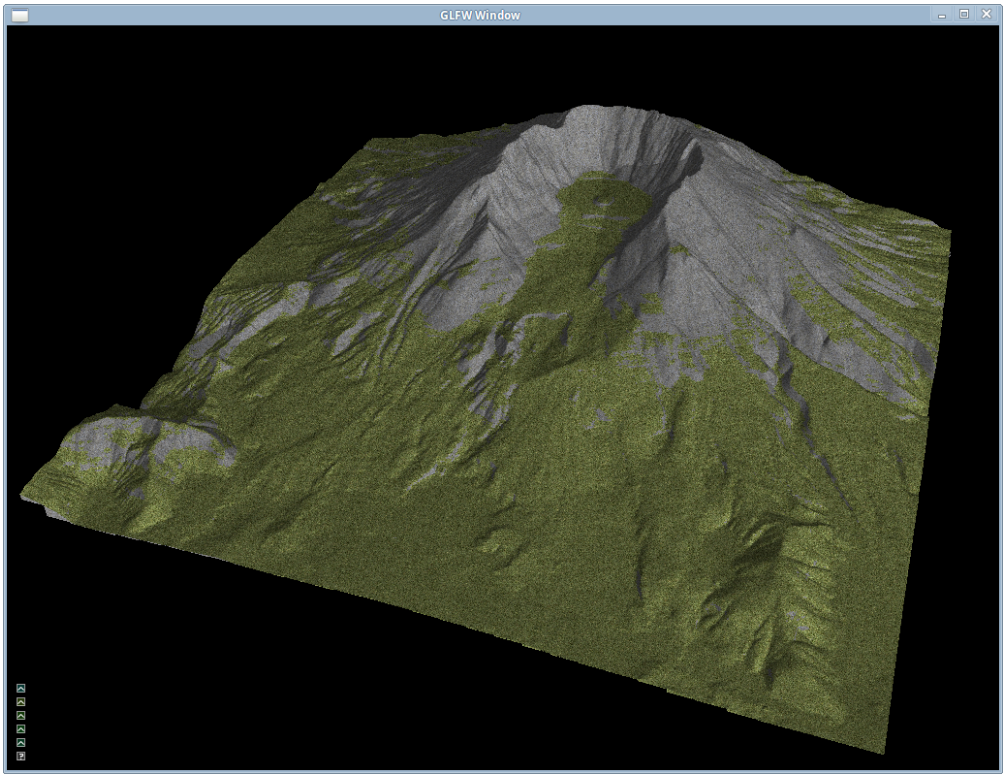
Heightmap This scene is just an unnamed heightmap, henceforth just heightmap. One of the alternative scenes that already existed in the HPC-Lab Snow-Simulator, it represent a hilly terrain with one valley. It is interesting to see the results of varying placements of the antenna in regards to providing signal in the aforementioned valley.



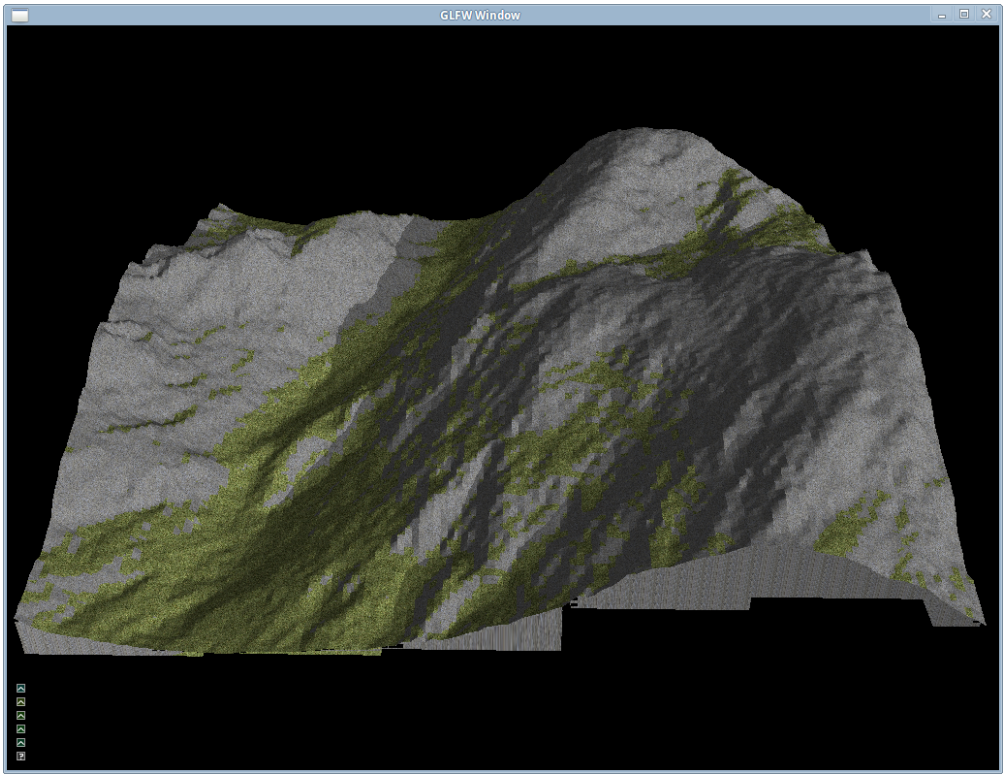
(A) Flat scene



(B) Flat scene with walls



(c) Mount St Helen



(d) Heightmap

Figure 5.0: The four scenes used in this thesis.

Configuration #	Distance	Elevation Tx	Elevation Rx
1	$2000m$	$2m$	$2m$
2	$50m$	$10m$	$10m$
3	$50m$	$10m$	$25m$
4	$200m$	$10m$	$10m$
5	$10m$	$20m$	$20m$
6	$10m$	$20m$	$50m$

TABLE 5.1: The configurations used

Configuration #	Hand calculated Rx	Simulated Rx
1	$4.37493197e^{-5}$	$5.379e^{-5}$
2	6.5003752	6.49765
3	6.879152	6.86112
4	0.174872	0.177014
5	167.788	167.478
6	17.74981	17.3514

TABLE 5.2: Results of hand calculated receiver strength vs. simulated. All simulations were run with 16000 by 16000 waves, and a total sender strength of 65536.

5.2 Reflection

Reflection being very important for signal coverage in some scenes, it is very important to have correct reflection behavior in the simulator. To verify the reflection, the flat scene shown in Figure 5.1a was used, and the antenna and receiver were placed at fixed distances from eachother (ground level) and elevations from the ground. The ground is given the dielectric properties of soil, given in Table 4.1.

Six different configurations were calculated by hand, and then simulated. The six configurations are given in Table 5.1, and the results of the calculations and simulations are given in Table 5.2.

It is also interesting to see the reflection coefficient, to ensure that it becomes close to the expected value (as calculated by hand). To do so, the capability to print directly from inside the kernels is used, and all rays that hit a receiver by indirection print out the reflection coefficient they used. The result is shown in Table 5.3, note that it is not important that reflection coefficients be exactly equal to hand-calculations, as individual rays *should* be incident at slightly different coordinates and angles than the model ray used for hand-calculations. This holds also for the result on configuration 2 where the magnitude of the largest difference is greater than that of the reference reflection coefficient.

Configuration #	Hand calculated R	Largest difference
1	-0.98932227	$2.28541e^{-3}$
2	-0.0084463124	0.0232744
3	0.19704372	0.0109864
4	-0.59802502	0.0348054
5	0.40978258	$2.44144e^{-3}$
6	0.4168122	$9.33448e^{-4}$

TABLE 5.3: Results of hand calculated reflection coefficients vs actual reflection coefficients.

5.3 Refraction

Verifying refraction is somewhat more challenging than verifying reflection. However, as the refraction coefficient is the complement of the reflection coefficient, and a correct reflection coefficient requires a correct angle of refraction. Following that the reflection coefficients appear correct (Table 5.3), we can expect that also the angle of refraction is good as it is part of the calculation of the reflection coefficient. Which was easily verified by simply printing the angle calculated (and based on what data) directly from the kernel.

However, the next step in the algorithm is to actually spawn a new ray in the correct direction (angle of refraction to the surface normal). To rotate the normal in the plane of incidence to get the correct refracted wave, Rodrigues' rotation formula is used (see Section 2.6.3). To verify the implementation of the formula, it suffices to take the dot product between the direction calculated and the surface normal, and verify that this is equal to the cosine of the angle of refraction. This works because the dot product between unit vectors reduces to the cosine of the angle between them. Both automated and manual inspection then agree that the refracted waves are at the expected angle, and the minuscule differences follow only from floating point inaccuracies. To verify, the configurations used in Table 5.1 are re-used and the refraction waves spawned on intersection with the ground are inspected, as can be seen from Table 5.4 floating point inaccuracies are the only source of error.

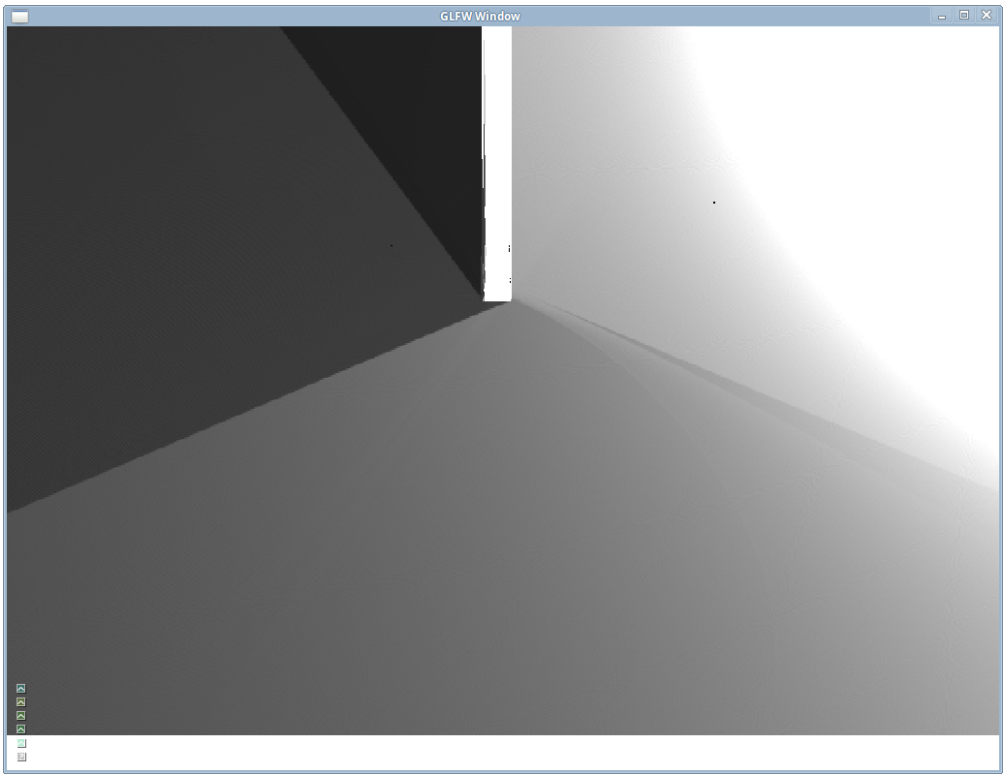
Configuration #	Error
1	$1.78814e-7$
2	$1.78814e-7$
3	$1.78814e-7$
4	$1.78814e-7$
5	$1.78814e-7$
6	$1.78814e-7$

TABLE 5.4: Errors in refraction rotation.

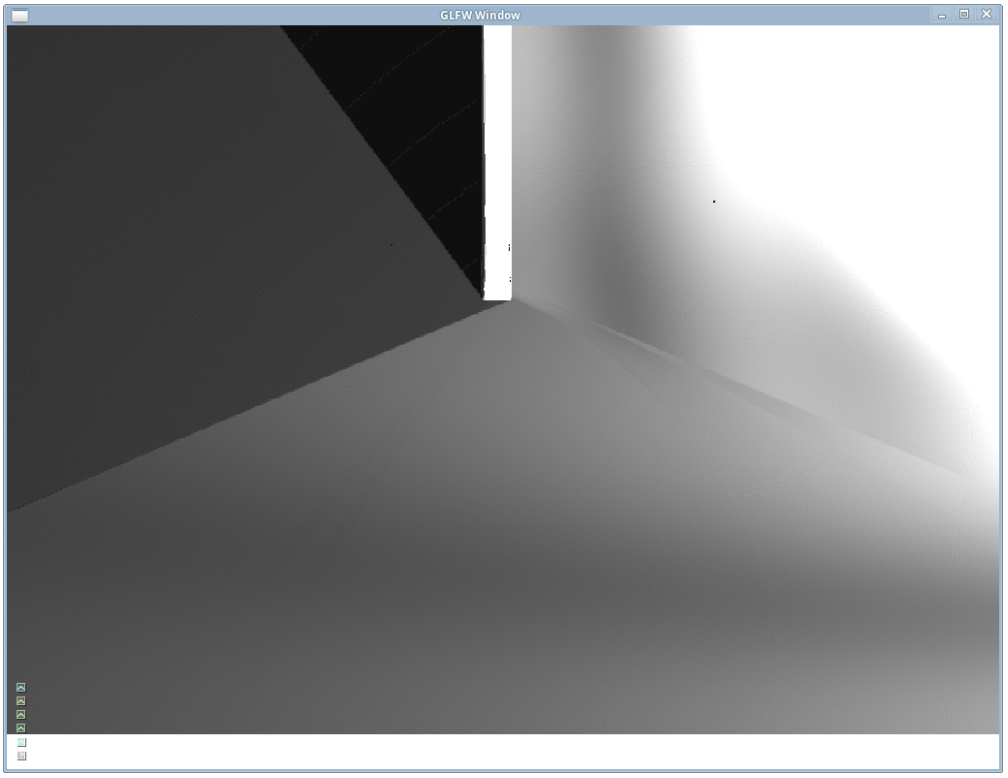
5.4 Interference

Several things cause rays to go out of phase. In this project we have accounted for the reflection coefficient and time delay. Figure 5.0 shows the resulting interference at 4 different frequencies.

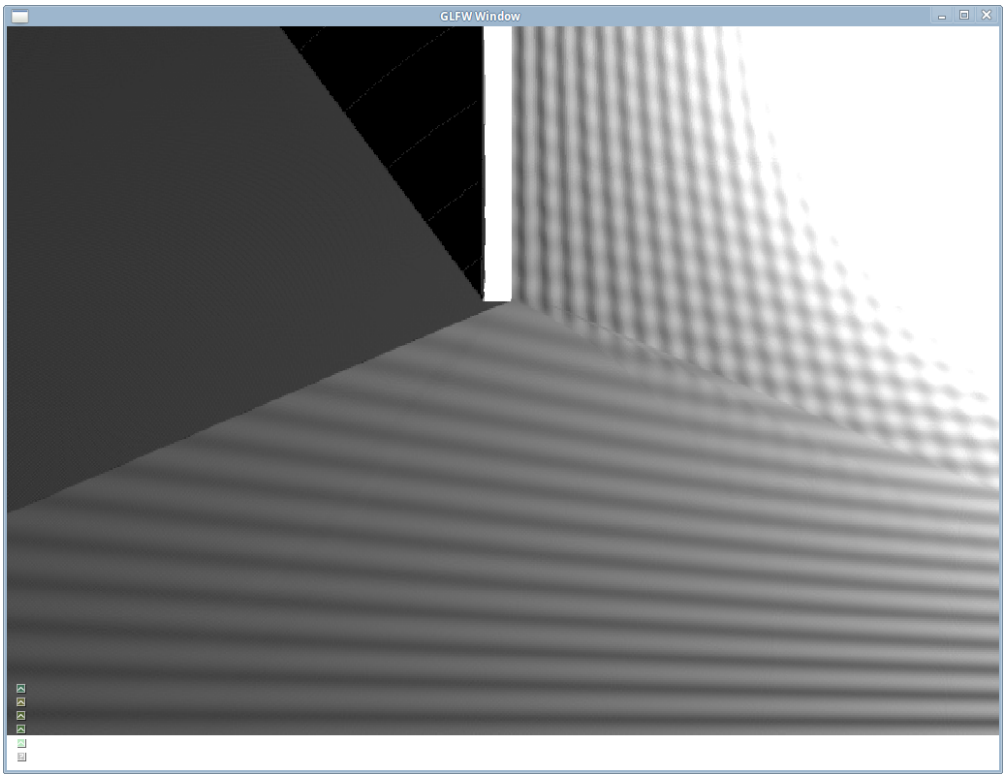
From Figure 5.1b it is visible that the interference is behaving in the manner one might expect. Experience and hand-calculations show the reflection coefficient at interfaces going from low refractive index to higher is positive, retaining the phase, and we get interference primarily only from the time delay for waves reflected off of the two walls visible (white at the bottom and center of the screen) or otherwise taking different paths.



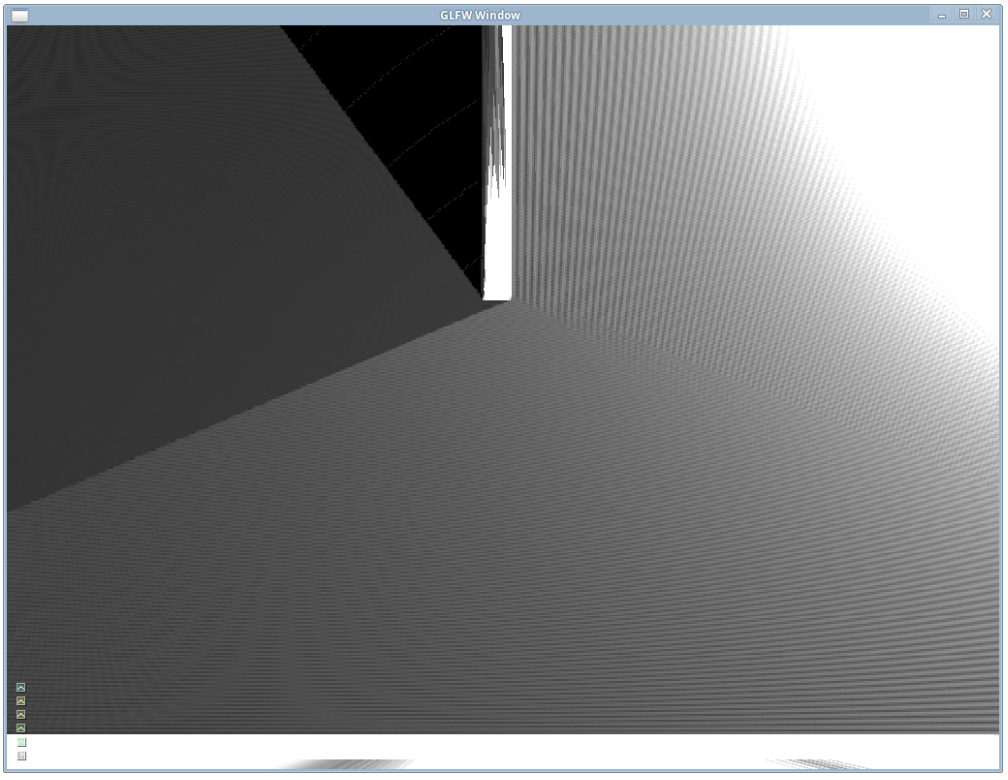
(A) Reflection and interference at 1mhz.



(B) Reflection and interference at 10mhz.



(c) Reflection and interference at 100mhz.



(d) Reflection and interference at 1ghz.

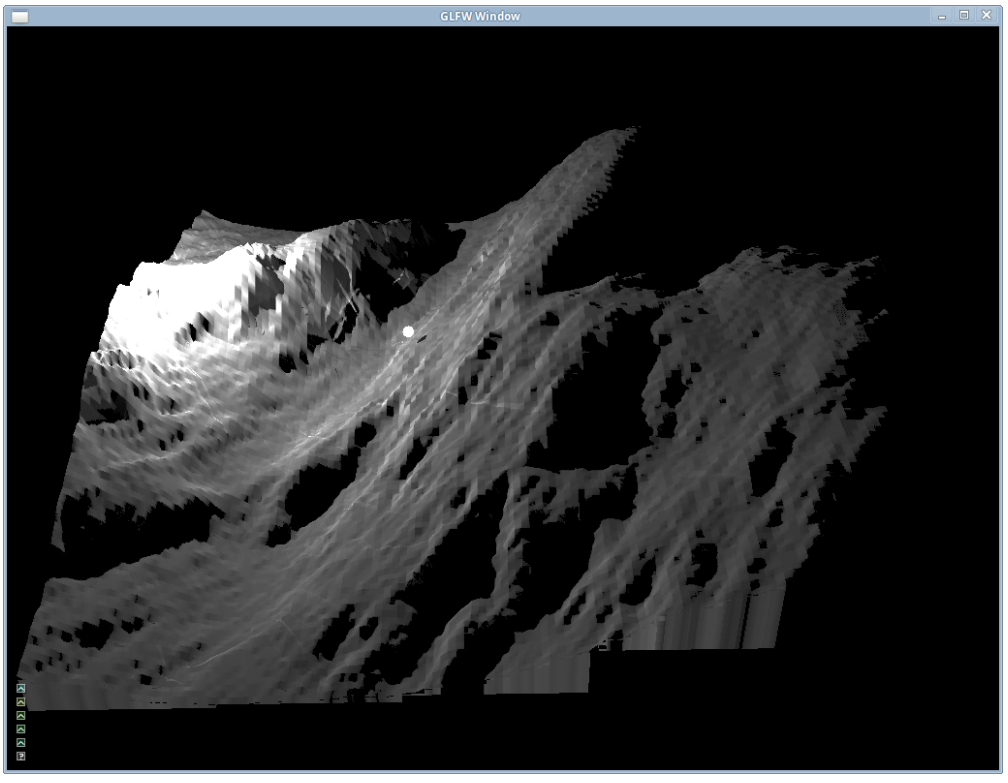
Figure 5.0: Rays reflect off of the walls causing interference. Notice the oscillating strength due to interference, particularly visible in Figure 5.1c. In these caps the antenna is in the upper right corner, while the walls are just under (going up behind) the camera and up in the middle. The ground itself is flat.

5.5 Terrain results

The most interesting result from this thesis, is naturally actual coverage calculation. In this section screencaptures of coverage estimation for Mount St Helen and the heightmap will be presented, with some discussion of the results.

5.5.1 Heightmap

Perhaps the most interesting result to be seen from this heightmap, is that out of these three placements it is neither of the two higher points on the right-hand side of the map that give the best coverage. The highest point, in fact, does not give coverage into the valley on the middle-left at all, whereas both the other placements are able to light up the receiver placed in there.



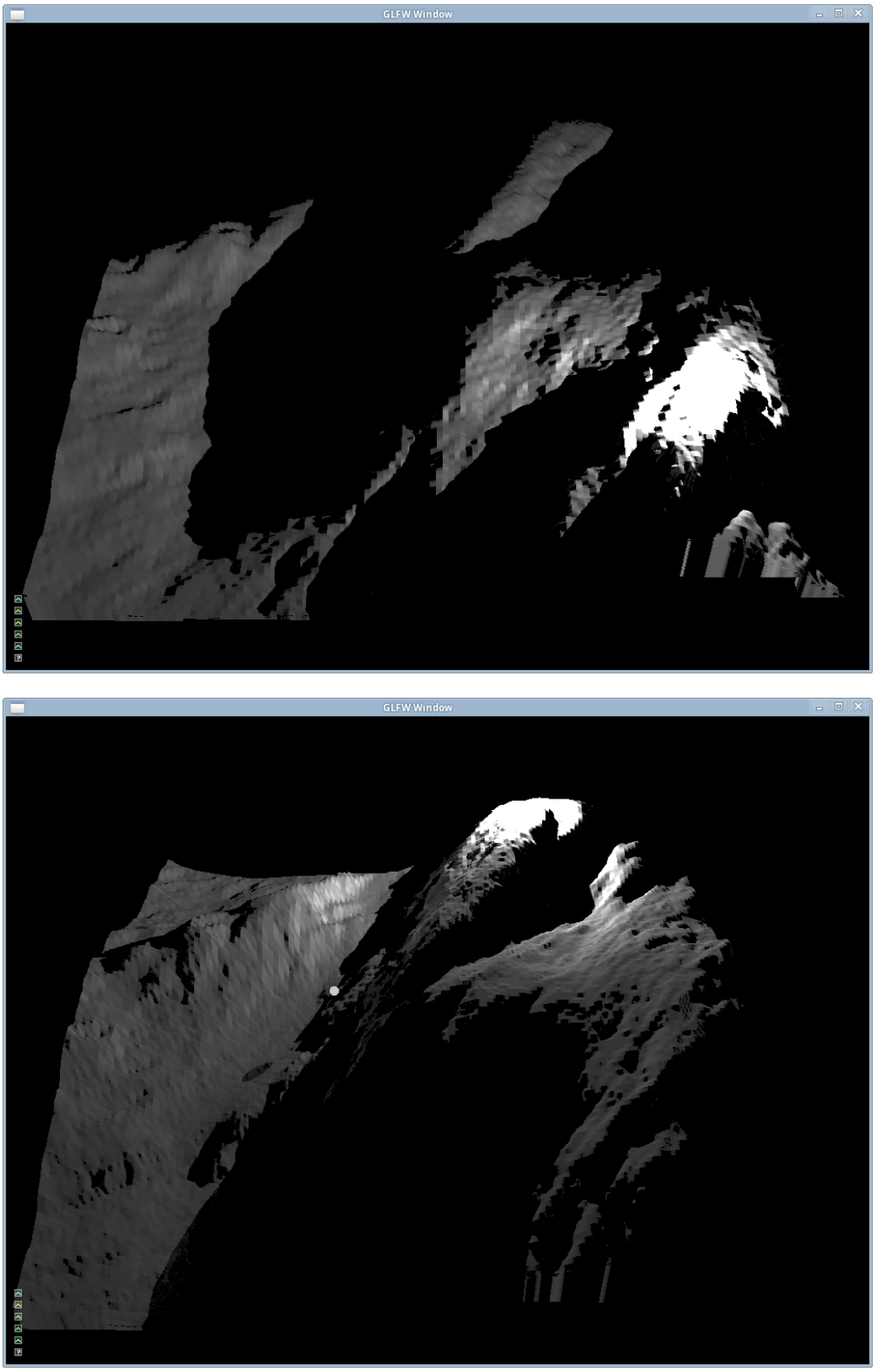
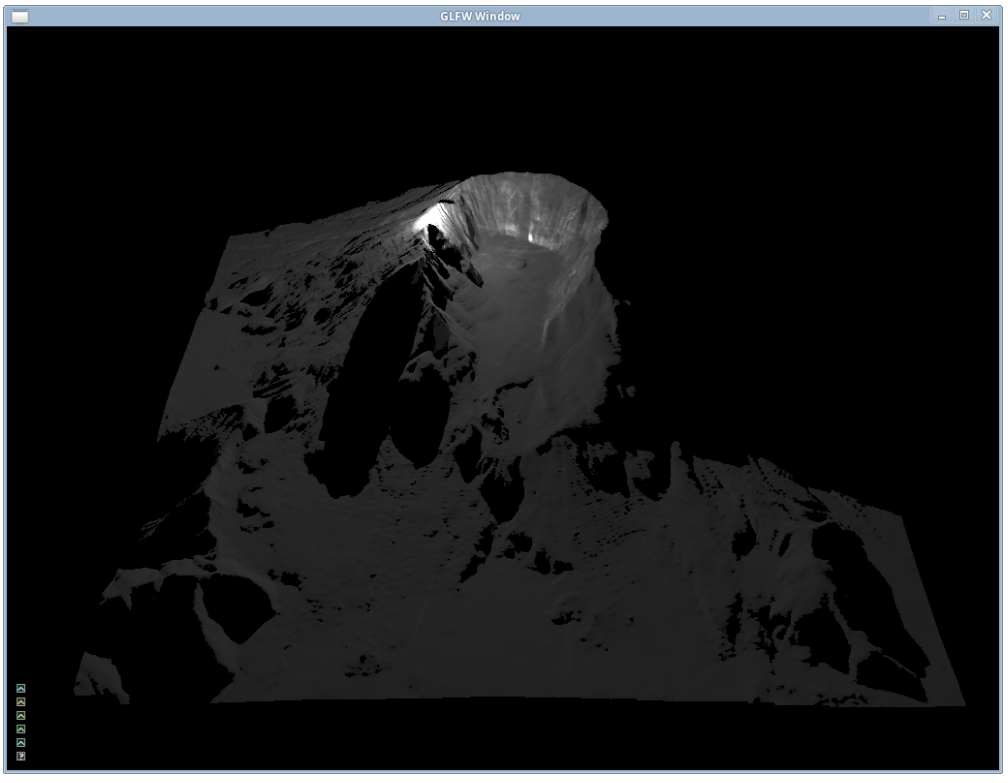
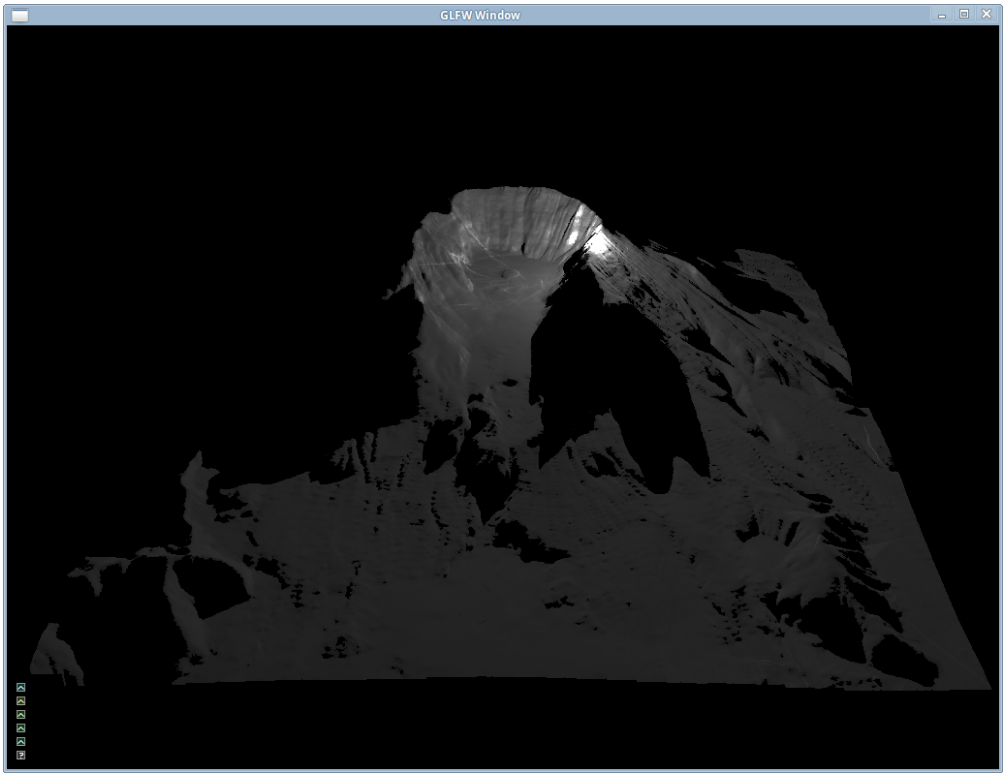


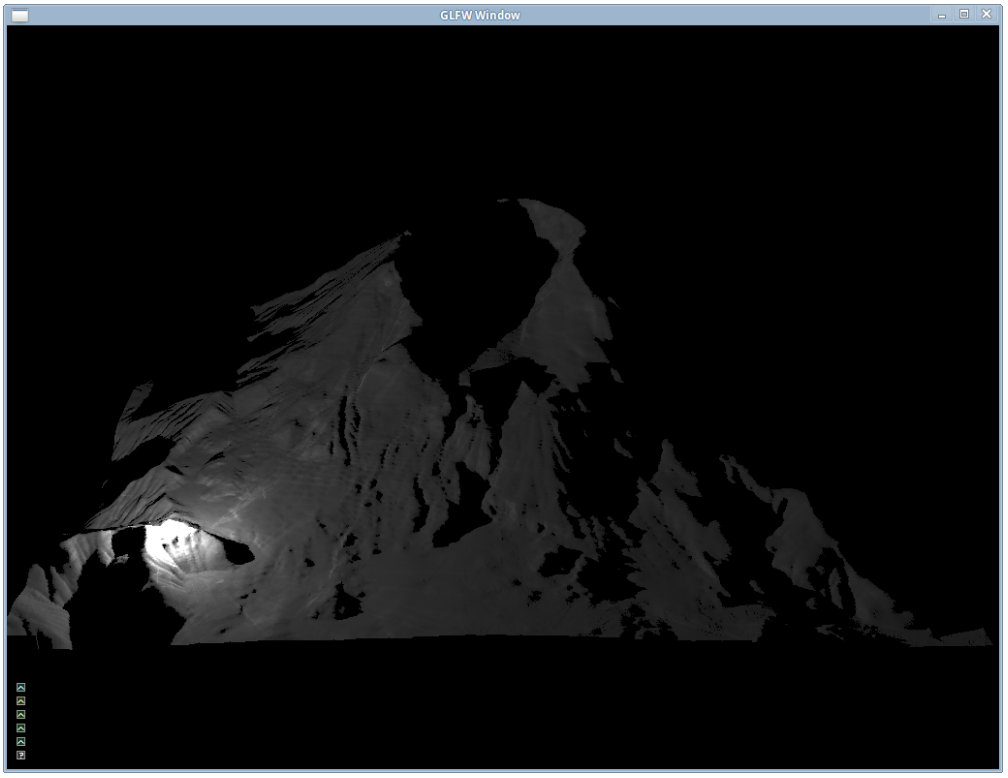
FIGURE 5.0: Three different placements of the antenna in the heightmap.

5.5.2 Mount St Helen

Not all terrains can be completely covered by a single antenna. Mount St Helen is by far the highest top in the area, and it is therefore seemingly not possible to place an antenna such that all the area gets coverage, unless an antenna that is suspended in mid-air is an option. Some reflection can also be seen in this scene, as shown in Figure 5.0. When inspecting the are with prominent reflection, the sharp difference between vertical and parallel polarization becomes plainly visible. However, note that the reflection here is very rough and patchy, refer to a discussion about this in Section 5.8.4.




GLFW Window


GLFW Window

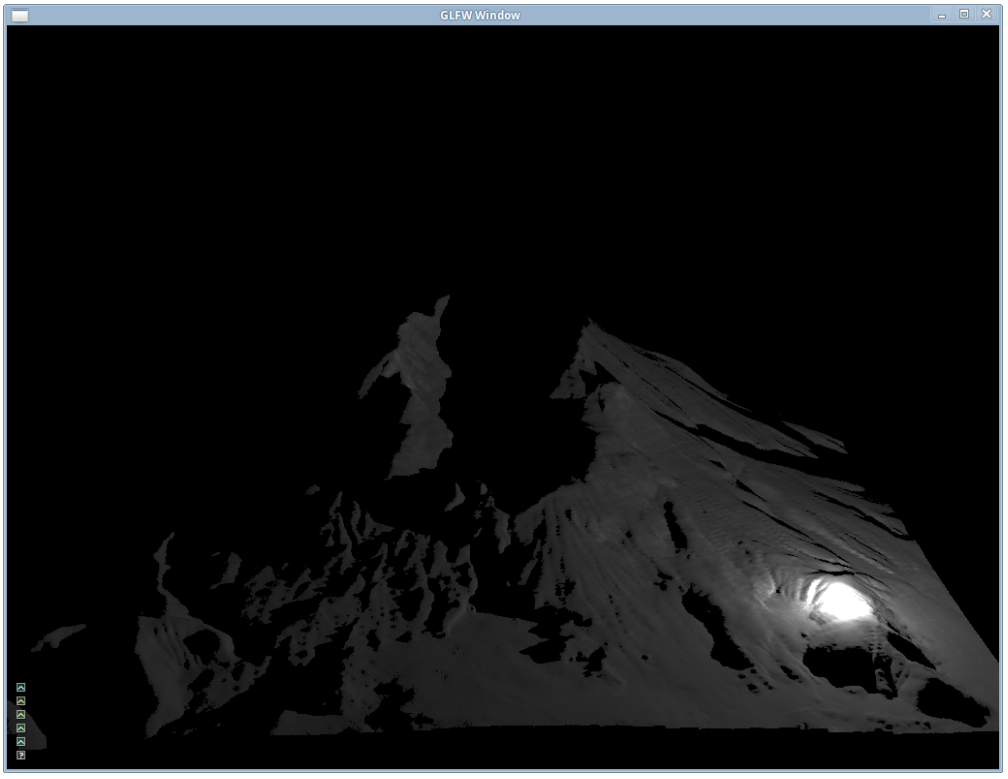
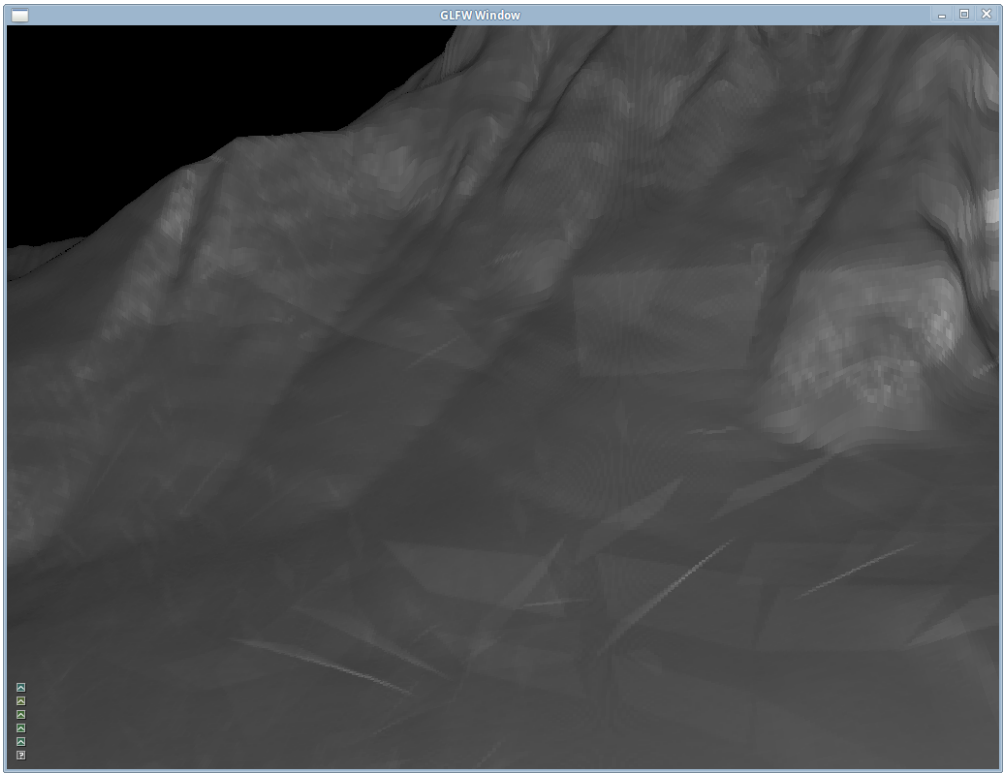
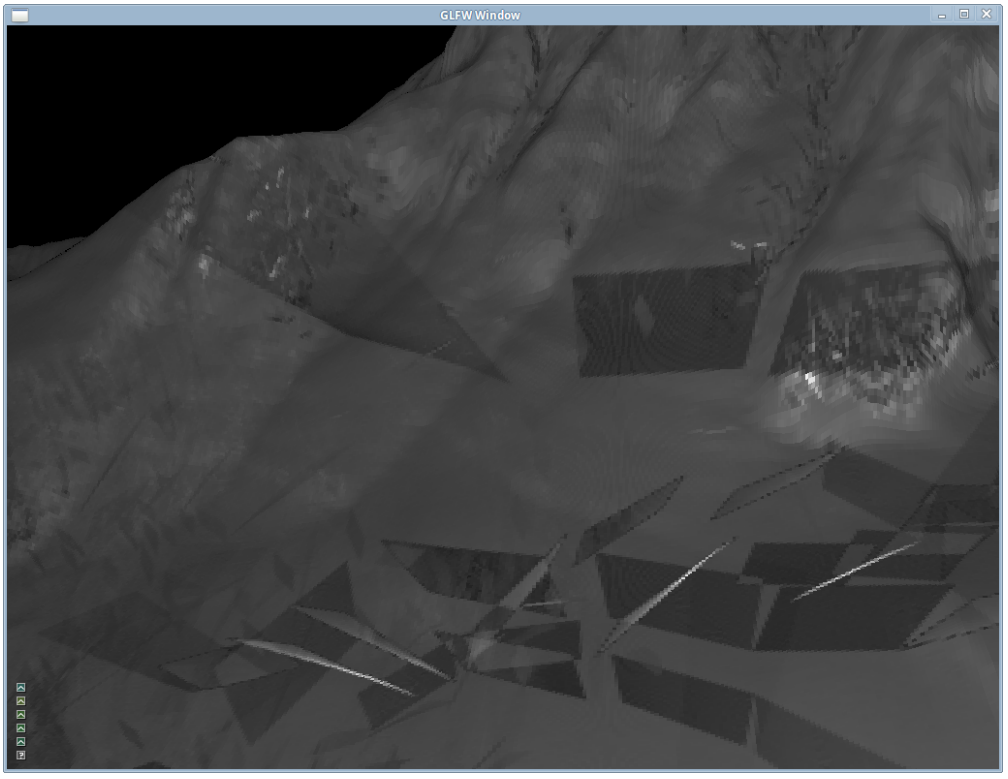


FIGURE 5.-1: Four different placements of the antenna in the Mount St Helen heightmap.



(A) Reflection of Parallel polarization.



(B) Reflection of Vertical polarization.

FIGURE 5.0: Reflection of parallel and vertical polarization in the Mount St Helen crater.

5.6 Cutoff Visualization

An alternative method of visualizing results was implemented. Rather than only lighting the areas with signal, the areas without also fade towards a given color the less signal it has.

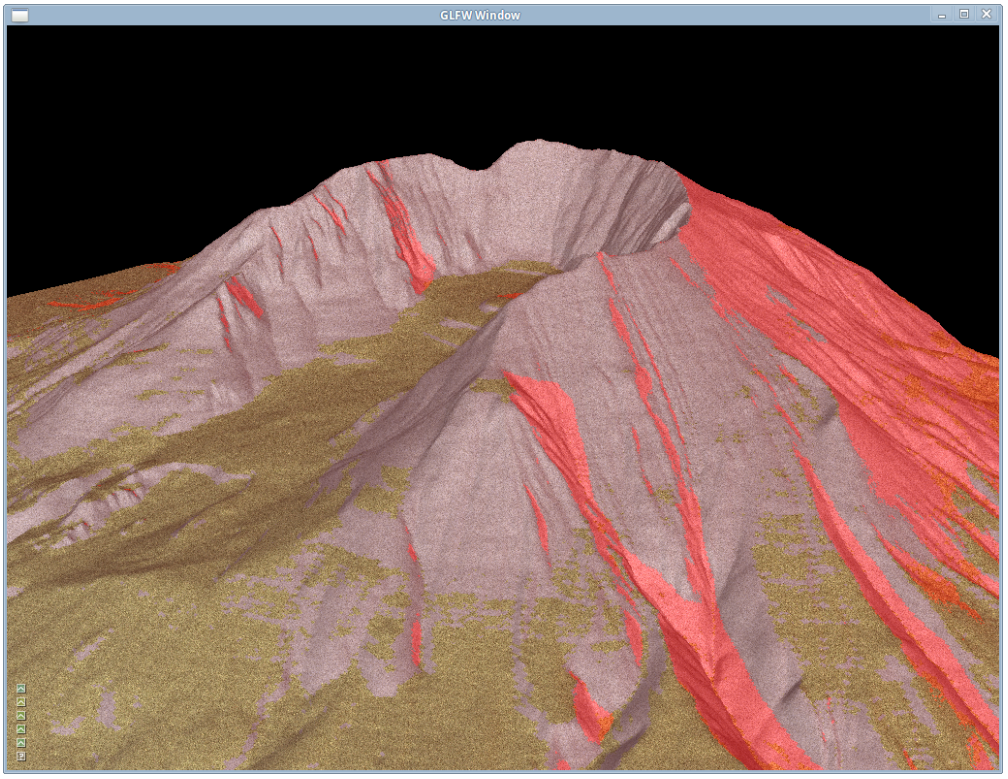


FIGURE 5.1: Cutoff visualization

5.7 Performance

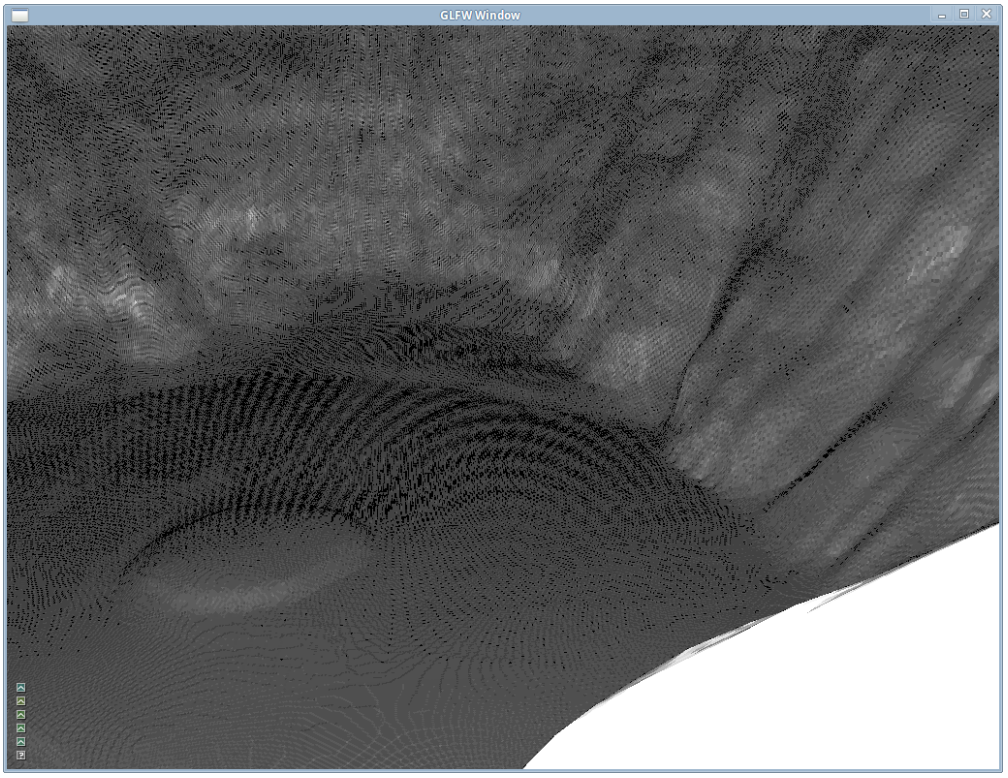
Signal Resolution	Trace time
256	$32ms$
512	$65ms$
1024	$141ms$
2048	$326ms$
4096	$826ms$
8192	$2.328s$
16384	$7.260s$
32768	$24.61s$
65536	$89.33s$
131072	$5m41s$
262144	$22m20s$
524288	$1h29m32s$

TABLE 5.5: Time to perform simulation at select resolutions, using the Mount St Helens map.

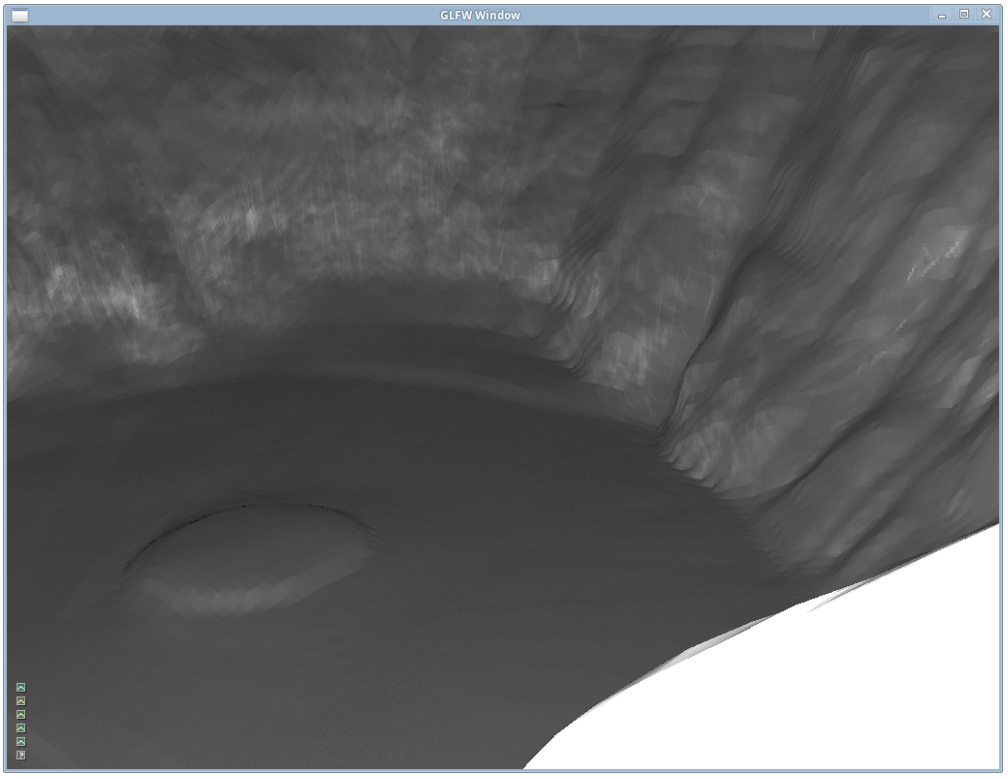
Table 5.5 details the time taken to perform a simulation at select signal resolutions in the Mount St Helens map. Clearly, even at the higher resolutions the simulation times are manageable for offline simulation. However, it might be desirable to try many different spots, at which point attempting to minimize the signal resolution while keeping the accuracy acceptable is the key.

5.7.1 Signal Resolution vs. Signal Map Resolution

As can be seen from Figure 5.2, an important factor to consider when reducing signal resolution is the signal map resolution. If the signal map is not resized along with the signal resolution, then the result will be a signal resolution that is unable to fill the map and the result will lose accuracy disproportionate to the reduction in signal resolution (as missing entire map entries reduces visual recognition, and makes the effect of reflection possibly invisible or deceptive as it might not add to or interfere where it should). A lower resolution map together with a lower resolution signal might be helpful in the initial scouting for antenna locations that deserve a more thorough simulation, the result of one such simulation is shown in Figure 5.3.



(A) Signal resolution of 4096 with a signal map resolution of 4096.



(B) Signal resolution of 16384 with a signal map resolution of 4096.

FIGURE 5.2: A comparison of the result of higher and lower resolutions of signal in the simulation.

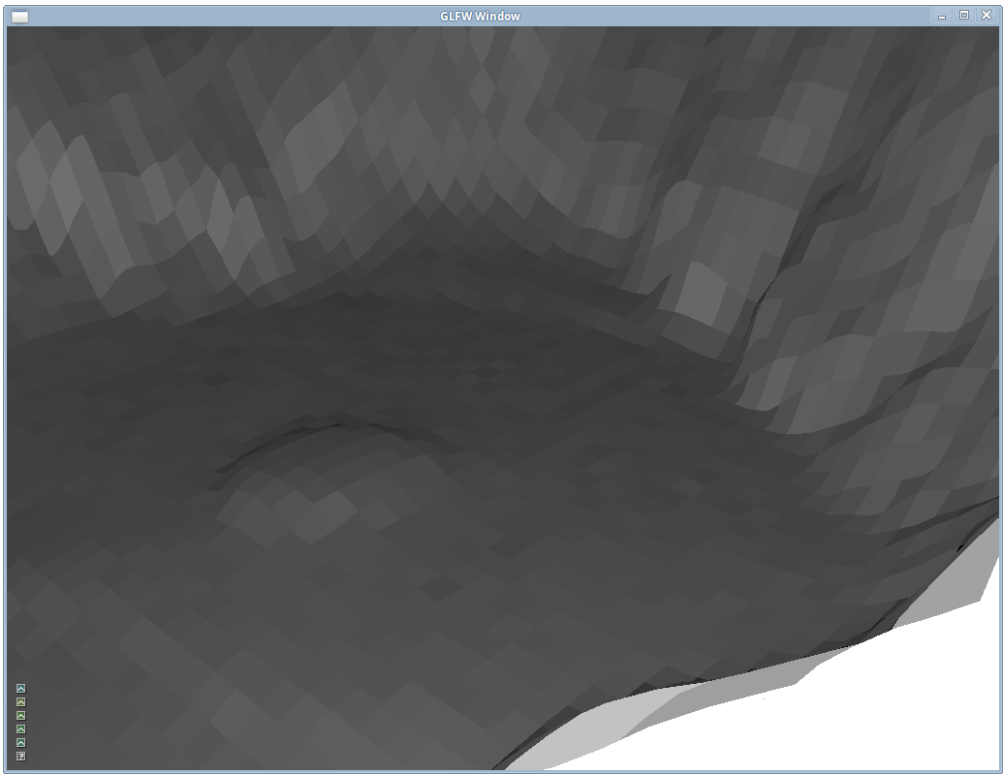


FIGURE 5.3: The visual result of a simulation run at a signal resolution of 1024 and a map resolution of 256. The simulation took 141ms.

5.8 Discussion

5.8.1 Floating point inaccuracies

The inaccuracies caused by floating point numbers with intersecting geometry is generally possible to overcome by some additional checks. However, the inaccuracies inherent in sending a very large number of rays is more difficult to handle. The primary issues stemming from this, are splitting the total ray strength into many smaller parts (one per ray), adding up the signal strength from very many small rays in the signal buffer, and sending the rays in each their own direction with an ever smaller difference in direction.

All of these issues could be greatly remedied by using higher precision floating point numbers. For handling signal strength, double precision floating point numbers are therefore employed to the greatest extent found possible, however hampered at some points by a bug in NVIDIA's OptiX' internal compiler: When using the double precision trigonometric functions, in some contexts OptiX will fail to parse the generated .ptx file, complaining about an unexpected character (which, upon inspection, is not actually present in the .ptx file at the line indicated by OptiX). Sometimes the double precision functions can work without problems, but othertimes it will refuse to compile and what precisely in the context causes

the failure was not possible to track down, and so single-precision functions were used. The precise code affected by this error is when spawning the rays, the surface area of each ray must be calculated using single precision, which means that as the number of rays grows large the surface area will grow inaccurate, causing an overall inaccuracy in the result.

In addition, NVIDIA's CUDA does not support atomic add operations for double precision. This could be implemented manually, but at a potentially very steep cost of performance. Although spread out over a large signal map, each warp is likely to have a very high rate of internal correlation in which buffer entry they access, so the performance cost is expected to be notable also for this project. However, the exact cost of switching to double precision has not been explored in this thesis. Since precision is already reduced to floating point levels due to the aforementioned bug, further exploration was not given priority.

Finally, when run on compute devices with a higher number of double precision floating point arithmetic units, the problem with accuracy in both direction and geometric intersection could be greatly mitigated by swiching to double precision entirely, eliminating all usage of single precision. However, optix carries no support for this, and as such the ray-tracing engine would have to be re-implemented to utilize double-precision, which is beyond the scope of this thesis.

5.8.2 Mapping signal buffer to terrain

Referring back to Section 4.3, as-is the coverage map cares only about the xz coordinates, and ignores elevation. This represents a challenge when storing signal strength for very steep terrain. Refer, for instance, to Figure 5.0. The walls are brightly white from high signal strength, even though the ground underneath is not (even at peak values). This is because no effort is made to map the buffer onto the heightmap, only the xz plane is mapped. Not only does this interfere with accuracy, as steep hills will look like it has the same signal coverage all the way up the hill, it also makes the method of normalization of signal strength used in Section 4.3 inaccurate. For example, the aforementioned walls turn brightly white because they actually cover a *much* larger area than just that in the xz plane.

5.8.3 Dielectric properties

The ground in our simulation does not consist of any specific composition of rock, or soil, and the exact interpolation for snow chosen in Section 4.1 are not chosen with any accuracy based on proper insight into the topic. Therefore all the values, except air and vacuum, of Table 4.1 do not reflect any real values, but are chosen

arbitrarily among the ones interpreted as possible for soil and rock when reading [Sco04, MB01, GAHT09, Geo14]. This of course, comes from the fact that the snow simulator does not carry any exact information about the exact composition of the soil, the rock, or any other matter.

5.8.4 Patchy reflection

Referring back to Figure 5.0 in Section 5.5.2. The reflection achieved in actual terrain is very rough. This is because the heightmap approach used in the HPC-Lab Snow Simulator does not lend itself very well to creating completely smooth terrain. This leads to the surface frequently having a snap in its curvature on the transition between cells, giving a very patchy look to both surface (when seen up close) and reflection, as if it had a discontinuous gradient.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, a ray-tracing based radio coverage simulator was developed and implemented based on the OptiX framework. The simulator was developed within the already existing HPC-Lap Snow Simulator framework which let it take advantage of some of the existing modules, such as the terrain heightmap and the graphical user interface.

First the he exact physical interactions between radiowaves and matter were considered, including using the dielectric properties of matter and exact angle of incidence. This allowed the simulation to give realistic reflection and refraction, where the actual reflection and refraction coefficients and angle of refraction could be calculated.

Subsequently, ourimplementation was extended to map a buffer onto the terrain that stores the received signal strength at each point on the ground. This allowed the simulation to not only calculate the strength of the signal received by some receiver placed in the scene, but also to generate a map that shows the geographic coverage given by a transmitter at a given position. The resolution of this map can be increased or decreased at will, which combined with varying the number of rays to be traced allows the simulation to change between preferring speed over accuracy or accuracy over speed. This allows a (very rough) simulation in real-time, which can be used to find locations that show promise faster, and then perform a more accurate simulation.

Finally, the implementation was extended by accounting for phase differences, and the interferent nature of waves in different phases. This allowed the simulation to show all reflections and refractions as not invariably positive to the outcome, but rather how it will vary based on exact distance between the reflection surface and

the path of other waves, and the reflection coefficient. The latter, the reflection coefficient, was also extended to see the waves as either vertically or parallelly polarized, which greatly influences the exact result obtained via reflection.

The work done in this thesis demonstrates how accurately signal coverage can be calculated with the aid of the acceleration of modern-day hardware. Simulations can be run at such resolutions that floating point inaccuracy is a much larger problem than execution time, and as such allows for much better and faster results than those of older CPU-based radio signal simulators. However, faster is not real-time, and as such the effects of winds on high antennas and other artifacts that could be caused by wind in the terrain, which were considered in the original problem description have not been explored.

Note, however, that a high resolution in the simulation does not necessarily mean accurate results. No effort was made to explore the effects of diffraction or atmospheric conditions on radio wave propagation.

6.2 Future Work

Folowing are suggestions for several extensions of this work:

6.2.1 Extending the simulation

As mentioned in our conclusion, no effort was made to simulate the effects of diffraction and atmospheric conditions on radio waves. It follows then that the simulation likely does not reflect reality. A natural next step of this work would hence be to attempt to incorporate these effects into the propagation model used. Air should not be considered enitrely uniform, leading to some bending of waves as they pass upwards through the atmosphere, and waves should diffract as they pass near objects.

Additional points at which the simulation can and should be extended, lie in the exact physical properties of the various mediums used. In our simulation, only a vague resemblance to soil and rock is used. A more real simulation should contain scenes with an abundance of matter, with real dielectric values.

6.2.2 Complex values

In none of the theory given in Chapter 3 are complex values considered. In reality, many of the values given (such as permittivity) and calculated (the reflection coefficient) have complex valued solutions. According to [Cla14], these lead to complex

results for the reflection and refraction coefficient, where the complex-valued solution can be seen as a phase shift. These were not taking into consideration in this thesis, and might represent an important point to increase the accuracy of the simulation.

6.2.3 Signal map

Referring to Section 5.8.2. Adjusting for any of this has not been given priority, and the issue has been summarily ignored by the author. An obvious idea for modifying the normalization of signal strength to adjust for this issue, is to calculate the exact area covered by each entry into the heightmap. To better accomodate this, it might be appropriate to require the resolution of the signal map to be an exact multiple of that of the heightmap. This would allow the signal map logic to immediately index into the heightmap, and use some formula or logic to calculate or better approximate the area spanned by that cell.

6.2.4 Multiple GPUs

OptiX has native support for multiple GPUs. In regular visualization ray tracing, this should work well under the assumption that only the actual output buffer is written to and in an orderly fashion where each ray writes exclusively to its own index. However, for this thesis, the signal map used as the target buffer is written to with no regard for which ray writes where or in what order. This makes atomic add operations and the need to keep the buffers up to date on both GPUs all the time extremely slow and using a single GPU is far more efficient. As-is, the implementation picks whichever GPU comes first in OptiX's option to list devices, as it appears to put the video-out GPU last in the list. This allows long simulations to be run without hogging the rest of the system. Although the user will need to modify the code to use a different GPU.

As attempting to use the native support for multiple GPUs gave rather discouraging results, the potential usage of multiple GPUs has not been considered in this thesis. Clearly, using multiple GPUs could greatly improve the simulation speed for higher resolutions. As mentioned the main issue is the need for atomic access, where no spatial ordering of access busts OptiX's automatic support for multiple GPUs. An idea for adjusting for this is to rewrite the host-side of the application to allow multiple contexts, create one context per GPU, and manually create a simulation on each GPU. Each GPU can then spawn its own range of rays, and add signal strength to their own buffers, and a reduction can then be performed on those buffers.

Appendix A

User Guide

A.1 Installation

This section will detail compilation from source. Prerequisites are

- Access to the HPC-Lab Snow Simulator's "Raytrace" branch.
- The Nvidia OptiX SDK version 3.0.1.
- The Nvidia CUDA SDK version 5.0.
- CMake version 2.8.12 or later.

CMake: Earlier versions of CMake do not have finder-scripts for all packages used by the Snow Simulator. On Ubuntu 12.04 it was necessary to compile a newer version of CMake from source as the repository version was unable to find GLEW. To install an appropriate version, simply download the source for the latest version of cmake from http://www.cmake.org/, extract it, and compile as usual:

```
./configure
make
%This might be necessary to run as superuser:
make install
```

Snow Simulator: The HPC-Lab Snow Simulator should be cloned into a folder of the user's choice, whereupon the user should switch to the Raytrace branch. The build may be performed in-tree or out-of-tree. If building out-of-tree, *the executable file and all .ptx files must be manually copied to the top folder of the snow-simulator source and run from there.* To build in-tree, navigate to the source folder and run:

```
cmake .
```

To build out-of-tree, create a new folder to build in and then run cmake using the path to the source directory:

```
mkdir build
cd build
% evt. replace .. with path to source directory
cmake ..
```

If you run cmake on the source now, the compilation will likely fail. Either by error of some missing package or not being able to find package configuration files for OptiX. First and foremost, install all missing packages.

OptiX and CUDA: The application has been tested using version 3.0.1 and 3.5.1 of the NVIDIA OptiX library. Version 3.0.1 supports only CUDA 5.0, whereas version 3.5.1 should support both CUDA 5.0 and 5.5. However attempts at running with the configuration OptiX version 3.5.1 and CUDA 5.5 failed, so the author recommends using OptiX version 3.5.1 with CUDA 5.0.

CUDA can be installed as usual, and CMake should be able to find it. If multiple versions of CUDA is installed, the usage of a UI wrapper such as cmake-curses-gui is recommended to simplify specifying which of the CUDA installs to use. Alternatively the symbolic link /usr/local/cuda can be changed to point at the desired version.

```
sudo apt-get install cmake-curses-gui
% cmake-curses-gui is run the same way one runs cmake, only with ccmake
ccmake .
```

Note that the repository version of cmake-curses-gui may be unable to find GLEW for the same reason the repository version of cmake might not. However, if cmake-curses-gui stops with an error saying it cannot find GLEW, it suffices to "quit without generating" and then running the updated cmake.

OptiX should only be extracted, but can be extracted to any folder of the user's choice. To allow CMake to find OptiX, one can choose between specifying the folder manually using cmake-curses-gui or exporting the environment variable "OptiX_INSTALL_DIR" to that directory.

```
% example export of OptiX_INSTALL_DIR
export OptiX_INSTALL_DIR=~/Downloads/NVIDIA-OptiX-SDK-3.0.1-linux64
```

Once the environment variable has been exported, it is necessary to clean up all temporary files generated by CMake from the build directory if any unsuccessful attempts were made, before trying again. The author also found that it was necessary to clean all temporary files generated by CMake in an in-tree build, even when trying to build out-of-tree.

The OptiX SDK should then be found by CMake. The only step that remains is installing any other missing packages. The HPC-Lab Snow Simulator code has no wonky dependencies and can safely be compiled with multiple threads ("make -j") for faster compilation.

A.2 Usage

A.2.1 Startup menu

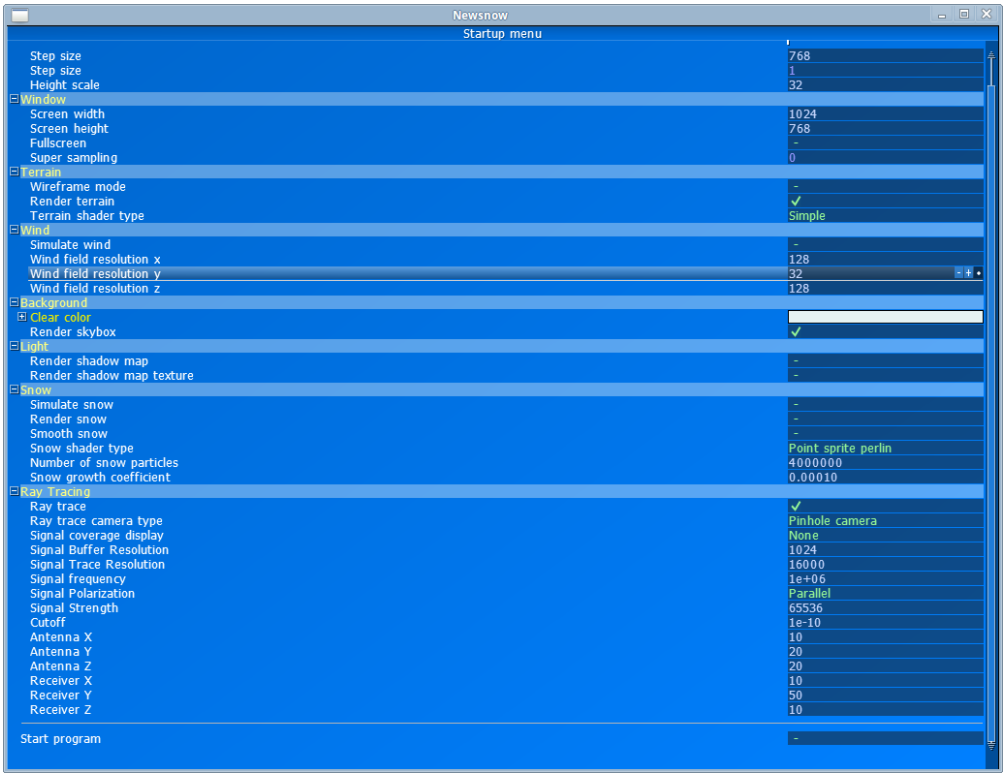


FIGURE A.1: Screen capture of the startup menu.

Figure A.1 shows the first screen that greets you upon launching the application. In this section the menu options related to the radio simulation and ray-tracing components of the application will be described. Note that only *Signal Buffer Resolution* must be set at this point in the program, all other options can be changed at runtime.

- **Ray Trace:** Checking this field enables ray-tracing visualization. Required for all functionality related to ray-tracing.
- **Ray Trace Camera Type:** Which type of camera to use: Pinhole or Environment (spheric)
- **Signal Coverage Display:** What type of visualization of coverage to use.
- **Signal Buffer Resolution:** The resolution to use on the signal buffer.
- **Signal Trace Resolution:** The resolution of rays sent from the antenna.
- **Signal Frequency:** Frequency of the waves transmitted from the antenna.
- **Signal Polarization:** Type of polarization used: Parallel or Vertical.
- **Signal Strength:** Total power transmitted from the antenna.
- **Antenna XYZ:** Coordinates of the antenna location.
- **Receiver XYZ:** Coordinates of the receiver location.

At this point it is most important just to set the signal buffer size before clicking "Start program".

A.2.2 Main menu

Figure A.2 shows a capture of the interface with menu after starting in the Mount St Helen map. Notice that most of the options from the main menu are here as well. However, there are a few new fields to introduce:

- **Cutoff:** The minimum power you expect a receiver will be able to pick up.
- **Cutoff color:** Color to fade towards if cutoff visualization is enabled.
- **Visualized cutoff:** Enables cutoff visualization.
- **Move antenna:** Moves the antenna to the current location of the camera.
- **Move receiver:** Moves the receiver to the current location of the camera. Note that you wil be blinded by this as the receiver will cover the camera. Simply move the camera a short distance to correct this.
- **Received strength:** The power incident at the buffer entry at the spot you last clicked on. Left click at any spot and this field will update itself while any signal coverage display is enabled. Note that the spot moves with the camera so after moving the camera you will need to click again.

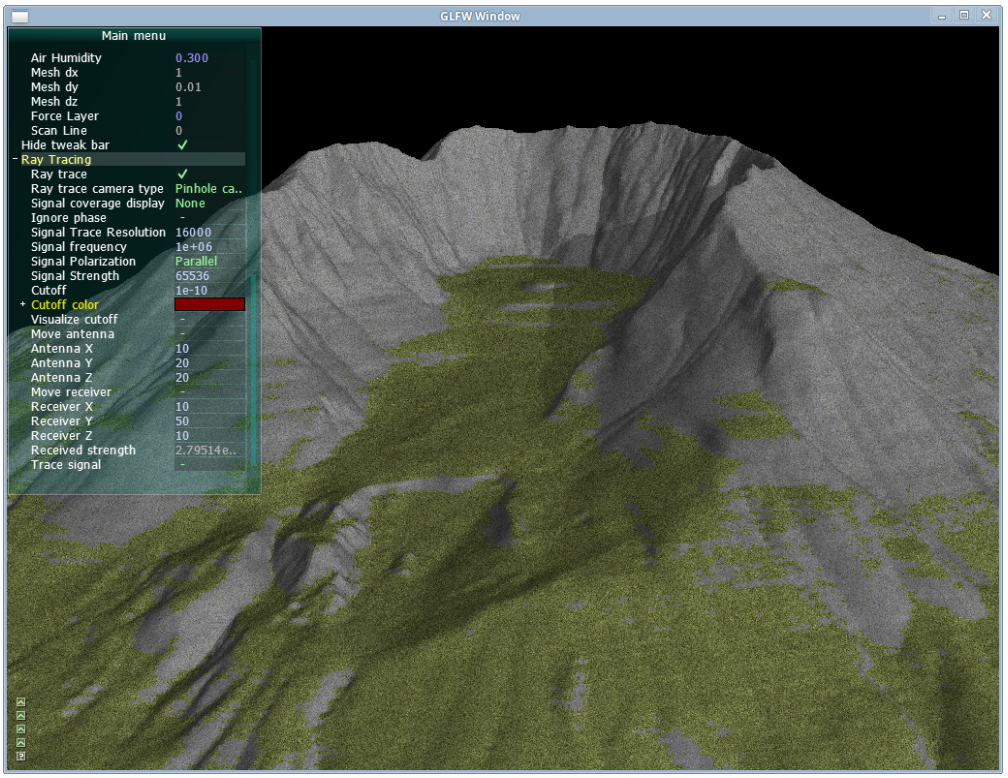


FIGURE A.2: Screen capture of the main menu.

- **Trace signal** Requests a simulation of radio signal with the current configuration.

Appendix B

Selected Codelets

B.1 Geometry - Bounding Boxes

```
// Generate the bounding box for a sphere
rtDeclareVariable(float3, center, , );
rtDeclareVariable(float, radius, , );
RT_PROGRAM void bound( int idx, float result[6] )
{
  result[0] = center.x - radius;
  result[1] = center.y - radius;
  result[2] = center.z - radius;
  result[3] = center.x + radius;
  result[4] = center.y + radius;
  result[5] = center.z + radius;
}
```

```
// Generate the bounding box for a portion of the heightmap
rtDeclareVariable(float3,   bmin, , );
rtDeclareVariable(float3,   bmax, , );
RT_PROGRAM void bound( int primIdx, float result[6] )
{
  result[0] = bmin.x;
  result[1] = bmin.y;
  result[2] = bmin.z;
  result[3] = bmax.x;
  result[4] = bmax.y;
  result[5] = bmax.z;
}
```

B.2 Geometry - Intersection

```
// Intersection of a sphere
RT_PROGRAM void intersect( int primIdx )
{
```

```
  float b,c;
  //Move sphere to origin
  float3 o = (ray.origin - center);

  //Solve quadratic equation a*t^2 + b*t + c
  b = optix::dot(2.f * o , ray.direction);
  c = optix::dot(o, o) - radius*radius;

  //t = (-b +- sqrt(b^2 - 4ac))/2a
  float discriminant = b*b - 4*c;
  if(discriminant<0.f) return; //No real roots, miss
  discriminant = sqrtf(discriminant);
  float t1 = (-b + discriminant) / (2.f);
  float t0 = (-b - discriminant) / (2.f);


  if(t0>0.f)
  {
    if(rtPotentialIntersection( t0 ))
    {
      invalid_hit = 0;
      receiver_id = sphere_id;
      float3 hit = ray.origin + ray.direction*t0;
      geometric_normal = optix::normalize(center - hit);
      if(rtReportIntersection(0)){
        return;
      }
    }
  }
  if(t1>0.f)
  {
    if(rtPotentialIntersection( t1 ))
    {
      invalid_hit = 0;
      receiver_id = sphere_id;
      float3 hit = ray.origin + ray.direction*t1;
      geometric_normal = optix::normalize(center - hit);
      if(rtReportIntersection(0)){
        return;
      }
    }
  }
  return;
}
```

```
// Quadratic interpolation of the heightmap
// t_near is the t value at which the ray enters the square.
// (Lu, Lv) are the x,z coordinates of the cell in the heightmap
float3 EC = (ray.origin + t_near * ray.direction - gmin) * inv_cellsize -
    make_float3(Lu, 0.0f, Lv);
EC.y = ray.origin.y + t_near * ray.direction.y;

float ya = d00;
float yb = d10-d00;
float yc = d01-d00;
float yd = d11-d10-d01+d00;
float a = D.x*D.z*yd;
float b = -D.y + D.x*yb + D.z*yc + (EC.x*D.z + EC.z*D.x)*yd;
float c = ya - EC.y + EC.x*yb + EC.z*yc + EC.x*EC.z*yd;

if(abs(c) < 1.e-6f)
{
  // Register as intersection.
```

```
  if( rtPotentialIntersection( t_near ) )
  {
    computeNormal(Lu, Lv, ray.origin+t_near*ray.direction, ya, yb, yc, yd);
    invalid_hit = 0;
    if(rtReportIntersection(0))
      return;
  }
}
else if(abs(a) < 2.e-5f)
{
  // Drop quadratic term and solve linear
  float t_cell = -fdividef(c, b);
  float t = t_near + t_cell;
  if(t_cell > 0.0f && t < t_exit + 2.e-5f)
  {
    t = fmin( t, t_exit );
    if(rtPotentialIntersection( t ))
    {
      computeNormal(Lu, Lv, ray.origin+t*ray.direction, ya, yb, yc, yd);
      invalid_hit = 0;
      if(rtReportIntersection(0))
        return;
    }
  }
}
else
{
  // Solve quadatric
  b = -0.5f * b;
  float disc = b*b-a*c;
  if(disc > 0.0f){
    float root = sqrtf(disc);
    float t_cell = fdividef(b + root, a);
    float t = t_near + t_cell;
    bool done = false;
    if(t_cell >= 0.0f && t <= t_exit + 2.e-5f)
    {
      t = fmin( t, t_exit );
      if( rtPotentialIntersection( t ) )
      {
        computeNormal(Lu, Lv, ray.origin+t*ray.direction, ya, yb, yc, yd);
        invalid_hit = 0;
        if(rtReportIntersection(0))
          done = true;
      }
    }
    t_cell = fdividef(b - root, a);
    t = t_near + t_cell;
    if( t_cell >= 0.0f && t <= t_exit + 2.e-5f)
    {
      t = fmin( t, t_exit );
      if( rtPotentialIntersection( t ) ) {
        computeNormal(Lu, Lv, ray.origin+t*ray.direction, ya, yb, yc, yd);
        invalid_hit = 0;
        if(rtReportIntersection(0))
          done = true;
      }
    }
    if(done)
      return;
  }
}
```

B.3 Signal Map

```
//Indexing into the map
static __device__ __inline__ size_t2 get_map_coord()
{
  // Get exactly where the ray hit
  float3 hit_point = line(t_hit);

  // Find the hit location relative to the heightfield
  float3 field_point = hit_point - gmin;
  // Scale down/up to a value between 0.0 and 1.0
  float3 relative_position = field_point / (gmax - gmin);
  // Find coordinates in the signal map
  size_t2 nnodes = signal_map.size();
  double2 map_coordinatesf = make_double2(make_float2(relative_position.x,
    relative_position.z)
              * make_float2(nnodes.x, nnodes.y));
  size_t2 map_coordinatesi = make_size_t2(map_coordinatesf.x, map_coordinatesf.y)
    ;


  // Optix/cuda doesn't have min(2) for size_t...
  map_coordinatesi.x = map_coordinatesi.x > nnodes.x-1 ? nnodes.x-1 :
    map_coordinatesi.x;
  map_coordinatesi.y = map_coordinatesi.y > nnodes.y-1 ? nnodes.y-1 :
    map_coordinatesi.y;

  return make_size_t2( map_coordinatesi.x    , map_coordinatesi.y     );
}
```

```
//Retrieving signal strength
static __device__ __forceinline__ double pointStrength(float2 p)
{
  return sqrt( p.x*p.x + p.y*p.y );
}

static __device__ double get_strength()
{
  size_t2 coord = get_map_coord();
  size_t2 nnodes = signal_map.size();
  float3 fsizes = (gmax - gmin) / (make_float3(static_cast<float>(nnodes.x-1),
    1.0f, static_cast<float>(nnodes.y-1)));
  double2 dsizes = make_double2(fsizes.x, fsizes.z);
  double area = dsizes.x*dsizes.y;
  return fabs ( pointStrength(signal_map[coord]) / area );

}

static __device__ double getReceiverStrength()
{
  return fabs( pointStrength( receiver_buffer[ receiver_id ] ) );
}
```

```
// Handle intersection at a receiver
RT_PROGRAM void receiver_shader()
{
  // Get exactly where the ray hit
  float3 hit_point = line(t_hit);

```

```
  // Calculate ray strength at hit point
  //Free space loss
  prd_radio.d += static_cast<double>(t_hit);

  //Medium loss, if applicable
  double atr = attenuationRate( prd_radio.medium, prd_radio.frequency );
  prd_radio.strength /= exp(static_cast<double>(t_hit)*atr);

  prd_radio.time += timeTaken( t_hit, prd_radio.medium );

  double strength = prd_radio.strength;
  double projectedA = prd_radio.A*prd_radio.d*prd_radio.d;
  if(projectedA > M_PI)strength /= (projectedA/M_PI);
  // Account for phase shift
  if(!ignore_phase)
  {
    double phase_shift = phase( prd_radio.time, prd_radio.frequency );
    double x = strength*cosf(2.*M_PI*phase_shift);
    double y = strength*sinf(2.*M_PI*phase_shift);
    atomicAdd( &receiver_buffer[ receiver_id ].x, x );
    atomicAdd( &receiver_buffer[ receiver_id ].y, y );

  }
  else
  {
    atomicAdd( &receiver_buffer[ receiver_id ].x, fabs(strength) );
  }
}
```

```
// Handle intersection in the heightmap
// Shader specific declarations
RT_PROGRAM void signal_shader()
{
  // Workaround for not being able to signal to OptiX that it can/should narrow
    the t_range
  // without also registering an intersection.
  if(invalid_hit)return;

  // Get exactly where the ray hit
  float3 hit_point = line(t_hit);


  // Calculate ray strength at hit point
  //Free space loss
  prd_radio.d += static_cast<double>(t_hit);

  //Medium loss, if applicable
  double atr = attenuationRate( prd_radio.medium, prd_radio.frequency );


  prd_radio.strength /= exp(static_cast<double>(t_hit)*atr);
  prd_radio.time += timeTaken( t_hit, prd_radio.medium );

  double strength = prd_radio.strength;

  // Account for phase shift
  size_t2 coord = get_map_coord();
  if(!ignore_phase)
  {
    double phase_shift = phase( prd_radio.time, prd_radio.frequency );
    double x = strength*cosf(2.*M_PI*phase_shift);
    double y = strength*sinf(2.*M_PI*phase_shift);
    atomicAdd( &signal_map[ coord ].x, x );
```

```
34      atomicAdd( &signal_map[ coord ].y, y );

36    }
      else
38    {
        atomicAdd( &signal_map[ coord ].x, fabs(strength) );
40    }

42    // Do not continue totally insignificant rays
      if(fabs(strength) < 1.e-20)
44    {
        return;
46    }
      float3 normal = faceforward( geometric_normal, -ray.direction, geometric_normal
         );
48
      TerrainType newType = terrainType(normal);
50    double incident = acos( abs(optix::dot(ray.direction,normal)) / optix::length(
        ray.direction) / optix::length(normal) );

52    double3 data = reflectRefract(prd_radio.medium, newType, incident);

54
      // reflection ray
56    if( prd_radio.depth < max_depth) {
        float3 R = optix::reflect( ray.direction, normal );
58      RadioPL new_rpl = prd_radio;
        new_rpl.depth++;
60      new_rpl.strength *= data.x;
        new_rpl.R = data.x;
62      if( fabs(R.x) < RT_MIN_ANGLE ) R.x = RT_MIN_ANGLE;
        if( fabs(R.y) < RT_MIN_ANGLE ) R.y = RT_MIN_ANGLE;
64      if( fabs(R.z) < RT_MIN_ANGLE ) R.z = RT_MIN_ANGLE;

66      optix::Ray refl_ray = optix::make_Ray( hit_point, R, 2, scene_epsilon,
        RT_DEFAULT_MAX );
        switch(new_rpl.medium)
68      {
          case FREE:
70        case AIR:
            rtTrace(scene, refl_ray, new_rpl);
72          break;
          default:
74          rtTrace(underworld, refl_ray, new_rpl);
        }
76    }

78    //Refraction ray
      if( prd_radio.depth < max_depth) {
80      float3 R = refract( ray.direction, -normal, data.z);
        float diff = abs( cosf( data.z ) - optix::dot( R, -normal ) );
82      if(diff > 1.19210e-7)rtPrintf( "< %g >\n", diff );
        RadioPL new_rpl = prd_radio;
84      new_rpl.depth++;
        new_rpl.medium = newType;
86      new_rpl.strength *= data.y;
        new_rpl.R = data.y;
88      if( fabs(R.x) < RT_MIN_ANGLE ) R.x = RT_MIN_ANGLE;
        if( fabs(R.y) < RT_MIN_ANGLE ) R.y = RT_MIN_ANGLE;
90      if( fabs(R.z) < RT_MIN_ANGLE ) R.z = RT_MIN_ANGLE;

92      optix::Ray refr_ray = optix::make_Ray( hit_point, R, 2, scene_epsilon,
        RT_DEFAULT_MAX );
        switch(new_rpl.medium)
94      {
```

```
      case FREE:
96    case AIR:
        rtTrace(scene, refr_ray, new_rpl);
98      break;
      default:
100     rtTrace(underworld, refr_ray, new_rpl);
    }
102
  }
104 }
```

B.4 Radiowave Physics Functions

```
//Convenience function that calculates reflection index,
2 //refraction index, and refraction angle and returns it as
//one double3.
4 static __device__ __forceinline__ double3 reflectRefract(
                  TerrainType m1,
6                 TerrainType m2,
                  double incident)
8 {
  //Get refraction angle
10 double refraction = refractionAgle(m1, m2, incident);
  //Total reflection
12 if(refraction < 0.) return make_double3(1., 0., 0.);
  //Normal reflection
14 double n1 = sqrt(getPermittivity(m1) / VACUUM_PERMITTIVITY);
  double n2 = sqrt(getPermittivity(m2) / VACUUM_PERMITTIVITY);
16 double cosin = cos(incident);
  double cosre = cos(refraction);
18 double reflect;
  if(polarization == POLARIZATION_PARALLEL)
20 {
    reflect = (n2 * cosin - n1 * cosre)/(n2 * cosin + n1 * cosre);
22 }
  else
24 {
    reflect = (n1 * cosin - n2 * cosre)/(n1 * cosin + n2 * cosre);
26 }
  return make_double3(reflect, 1. - abs( reflect ), refraction);
28 }
```

```
static __device__ __forceinline__ float3 rotateAxis( float3 v,
2                       float3 axis,
                        double angle)
4 {
  //Rodrigues' rotation formula.
6 float3 axv = optix::cross(axis,v);
  float adv = optix::dot(axis,v);
8 double cs = cos(angle);
  double sn = sin(angle);
10 return (v * cs) + (axv * sn) + (axis * adv * (1 - cs));
}
12
static __device__ __forceinline__ float3 refract( float3 v1,
14                      float3 v2,
```

```
                        double angle)
{
  float3 axis = optix::normalize(optix::cross(v2, v1));
  float3 result = optix::normalize(rotateAxis(v2, axis, angle));
  return result;
}
```

```
// Find current phase offset from 0 as a normalized value.
static __device__ __forceinline__ double phase( double time, double frequency )
{
  double period = 1000000000. / frequency;
  double periods = time / period;
  return periods - floor(periods);
}
```

```
static __device__ __forceinline__ double speedInMedium( TerrainType m )
{
  return 1.0 / ( sqrt( getPermeability( m ) ) * sqrt( getPermittivity( m ) ) );
}

static __device__ __forceinline__ double timeTaken( double meters, TerrainType m
    )
{
  double time_seconds = meters / speedInMedium( m );
  double time_nano = time_seconds * 1000000000.;
  return time_nano;
}
```

```
static __device__ __forceinline__ double attenuationRate( TerrainType m,
                        double f)
{
  double conductivity = getConductivity(m);
  double permeability = getPermeability(m);
  if(conductivity > 0.001){
    return sqrt(permeability*M_PI*conductivity)*sqrt(f);
  }
  else{
    return 0.;
  }
}
```

```
static __device__ double getPermittivity(TerrainType medium)
{
  switch(medium)
  {
    case FREE:
      return VACUUM_PERMITTIVITY;
    case AIR:
      return AIR_PERMITTIVITY;
    case GRASS:
      return GRASS_PERMITTIVITY;
    case ROCK:
      return ROCK_PERMITTIVITY;
    case SNOW:
      return SNOW_PERMITTIVITY;
```

```
  }
  return 1.;
}

static __device__ double getPermeability(TerrainType medium)
{
  switch(medium)
  {
    case FREE:
      return VACUUM_PERMEABILITY;
    case AIR:
      return AIR_PERMEABILITY;
    case GRASS:
      return GRASS_PERMEABILITY;
    case ROCK:
      return ROCK_PERMEABILITY;
    case SNOW:
      return SNOW_PERMEABILITY;
  }
  return 0.;
}

static __device__ double getConductivity(TerrainType medium)
{
  switch(medium)
  {
    case FREE:
      return VACUUM_CONDUCTIVITY;
    case AIR:
      return AIR_CONDUCTIVITY;
    case GRASS:
      return GRASS_CONDUCTIVITY;
    case ROCK:
      return ROCK_CONDUCTIVITY;
    case SNOW:
      return SNOW_CONDUCTIVITY;
  }
  return 0.;
}
```

B.5 Cameras

```
/*
 * Pinhole camera.
 * Eye => Position of Camera "Eye"
 * W => Vector from Eye to center of Camera Pinhole.
 * U => Vector from Eye to side of the camera Pinhole.
 * V => Vector from Eye to top of the camera Pinhole.
 */
RT_PROGRAM void pinhole()
{
  float2 d = make_float2(launch_index) / make_float2(launch_dim) * 2.f - 1.f;;
  float3 ray_origin = eye;// + d.x*U + d.y*V - W;
  float3 ray_direction = normalize(W + d.x*U + d.y*V);
  if( fabs(ray_direction.x) < RT_MIN_ANGLE ) ray_direction.x = RT_MIN_ANGLE;
  if( fabs(ray_direction.y) < RT_MIN_ANGLE ) ray_direction.y = RT_MIN_ANGLE;
  if( fabs(ray_direction.z) < RT_MIN_ANGLE ) ray_direction.z = RT_MIN_ANGLE;
  optix::Ray ray = optix::make_Ray(ray_origin, ray_direction, 0, 0.f,
    RT_DEFAULT_MAX);
```

```
17
  PerRayData_radience prd_radience;
19  prd_radience.importance = 1.f;
  prd_radience.depth = 0;
21
  rtTrace(scene, ray, prd_radience);
23
  picture[launch_index] = make_color( prd_radience.result );
25 }
```

```
1 /*
 * Radiowave antenna
3 */
RT_PROGRAM void antenna()
5 {
  for(int i = 0; i < antennas.size(); i++)
7  {
    //a will be the number of radians spanned by emission in each (spheric)
    dimension
9   //.tr is the "top right" while .bl is the "bottom left" spheric coordinates
    used
    float2 a = antennas[i].tr - antennas[i].bl;
11  //d will be indexed into those radians based on launch index
    float2 d = make_float2(launch_index + rtStart) / make_float2(rtTotal) * a;
13  d += antennas[i].bl;
    //radians spanned by each ray's emission surface
15  double theta_1 = (1./static_cast<double>(rtTotal.y)) * static_cast<double>(a.
    y);
    double phi_1 = (1./static_cast<double>(rtTotal.x)) * static_cast<double>(a.x)
    ;
17  float3 angle;
    double A;
19  //Calculate relative area of emission
    if(launch_index.y + rtStart.y == 0)
21  {
      //Pole
23    if(launch_index.x == 0){
        double temp = cosf(theta_1 / 2.);
25      A = 4. * M_PI * (1. - temp);
      }
27    else return;
    }
29  else
    {
31    A = 2. * phi_1 * sin(d.y) * sinf(theta_1 / 2.);
    }
33


35  //Get angle of emission
    angle = make_float3(cosf(d.x) * sinf(d.y), cosf(d.y), sinf(d.x) * sinf(d.y));
37  //Extremely small angles will affect floating point precision
    if( fabs(angle.x) < RT_MIN_ANGLE ) angle.x = RT_MIN_ANGLE;
39  if( fabs(angle.y) < RT_MIN_ANGLE ) angle.y = RT_MIN_ANGLE;
    if( fabs(angle.z) < RT_MIN_ANGLE ) angle.z = RT_MIN_ANGLE;
41  //Set up payload
    RadioPL prd_radio;
43  prd_radio.strength = antennas[i].strength*A / (4.*M_PI);
    prd_radio.depth = 0;
45  prd_radio.d = 0.0;
    prd_radio.medium = AIR;
47  prd_radio.max = RT_DEFAULT_MAX;
    prd_radio.frequency = frequency;
49  prd_radio.time = 0.;
```

```
    prd_radio.A = A;
    prd_radio.origin = antennas[i].position;
    prd_radio.phase_shift = 0.;
    float3 ray_origin = antennas[i].position;
    float3 ray_direction = normalize(angle);
    //Raytracing
    optix::Ray ray = optix::make_Ray(ray_origin, ray_direction, 2, 0.,
    RT_DEFAULT_MAX);

    rtTrace(scene, ray, prd_radio);
  }
}
```

B.6 OptiX Variables

```
// Declaring the variables device-side

// The global bounding box
rtDeclareVariable(float3,   gmin, , );
rtDeclareVariable(float3,   gmax, , );

rtTextureSampler<float4, 2> grass;
rtTextureSampler<float4, 2> rock;
rtTextureSampler<float4, 2> snow;

rtBuffer<BasicAntenna> antennas;
rtBuffer<float2, 2> signal_map;
rtBuffer<float2, 1> receiver_buffer;
rtBuffer<float, 1> signalread_buffer;
rtDeclareVariable( RadioPL, prd_radio, rtPayload, );

rtDeclareVariable( float3,   geometric_normal, attribute geometric_normal, );
rtDeclareVariable( unsigned, invalid_hit, attribute invalid_hit, );
rtDeclareVariable( unsigned, receiver_id, attribute receiver_id , );
```

```
// Declaring OptiX variables host-side

static RTcontext context;

...

static inline void declareVariable( const char* name )
{
  RTvariable variable;
  rtContextDeclareVariable( context, name, &variable );
}

...

void rtInit(Terrain* terrain)
{
  ...
  declareVariable( "rtStart" );
  declareVariable( "rtTotal" );
  ...
}
```

```

static RTcontext context;

...

// Setting Variables host-side
// Equivalent functions were made for all other types than uint2 used in OptiX as
    well
static inline void setVariable2ui( const char* name, unsigned arg1, unsigned arg2
    )
{
  RTvariable variable;
  rtContextQueryVariable( context, name, &variable );
  rtVariableSet2ui( variable, arg1, arg2 );
}

...
void rtInit(Terrain* terrain)
{
  ...
  setVariable2ui( "rtStart", 0, 0 );
  setVariable2ui( "rtTotal", 0, 0 );
  ...
}
```

Appendix C

Poster

HPC-Lab
Computer & Info. Science
Norwegian University of Science and Technology
http://research.idi.ntnu.no/hpc-lab

GPU-Accelerated Radio Wave Coverage Prediction

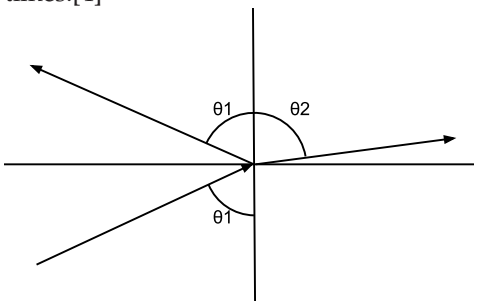
Mads Buvik Sandvei (Master Student) & Dr. Anne C. Elster (Advisor)
Dept. of Computer and Information Science, NTNU

Our Work

Accurately predicting the coverage of radiowave transmitters is an interesting and challenging problem in both civilian and military research. In order to accurately calculate coverage our work makes use of the technique of ray tracing on GPUs, augmented by considering electromagnetic wave physics to achieve accurate reflection coefficients, angles of refraction, attenuation rates, and interference, to make a physically accurate simulation of the propagation of radio waves. We use NVIDIA's OptiX[3] library to build a GPU-accelerated ray tracing application within the already existing HPC-Lab snow simulator, reusing terrain and user interface components already present in the snow simulator.

Ray Tracing

The technique of ray tracing[1] makes use of simulating the physical propagation of light to generate better pictures than those of rasterization. Ray Tracing sends rays from a source and then intersects them with a scene obtaining exact points of intersection, whereupon we recursively spawn reflection and refraction waves. This requires an enormous amount of computational power, however owing to the parallel nature of ray tracing we can use GPUs to accelerate the method and obtain acceptable simulation times.[4]



Reflection and Refraction

References

[1] Turner Whitted. An improved illumination model for shaded display. 1980

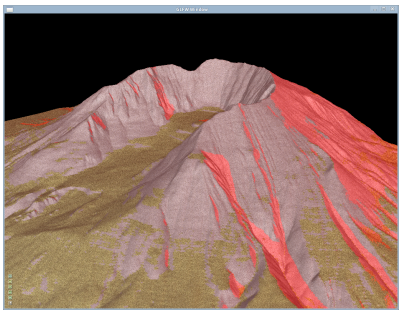
[2] Gonfer. Unfasor. 2009. Available from http://en.wikipedia.org/wiki/File:Unfasor.gif

[3] NVIDIA. NVIDIA OptiX Ray Tracing Engine Programming Guide 2012

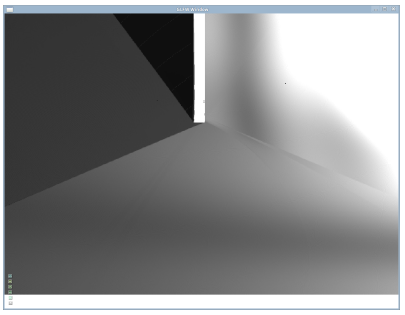
[4] Aile T. and Laine S. Understanding the efficiency of ray traversal on gpus. 2009

Signal Map

Cutoff visualization



Simple visualization



The map is implemented as a buffer of two-dimensional vectors. Three means of visualization was implemented: Simple visualization using signal strength as lone shading, using signal strength as lumination onto the textured terrain, and cutoff visualization using signal strength as lumination while fading into some color where signal strength is low.

EM Waves

Radio waves are electromagnetic waves, as is light. Ray tracing is therefore a suitable technique for simulating radio waves as well. Electromagnetic waves are waves of radiant energy that propagate through space, characterizable by three vector values: Direction, electric field, and magnetic field. The latter two can be expressed as:

$$E(x,t) = E_0 e^{-i\omega t}$$
$$B(x,t) = B_0 e^{-i\omega t} \quad (1)$$

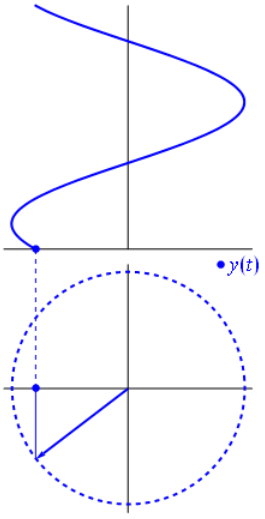
Whenever passing through or changing medium, the fields of electromagnetic waves will interact with the material, depending on its own frequency, polarization, and the dielectric properties of the mediums. The three most important properties are magnetic permeability μ, electric permittivity ϵ, and conductivityσ. To achieve accurate reflection coefficients, angles of refraction, and attenuation rates we use appropriate equations that use the dielectric properties:

$$R_v = \frac{n_1 \cos\theta_i - n_2 \cos\theta_t}{n_1 \cos\theta_i + n_2 \cos\theta_t}$$
$$R_p = \frac{n_2 \cos\theta_i - n_1 \cos\theta_t}{n_2 \cos\theta_i + n_1 \cos\theta_t} \quad (2)$$
$$\theta_2 = \arcsin \frac{n_1 \sin\theta_1}{n_2}$$
$$e^{-d\sqrt{\pi f \mu \sigma}}$$

Where n is the refractive index of the medium, given by $n = \sqrt{\epsilon_r}$.

Signal Representation

By taking the trigonometric representation of the complex exponent in Equation 1, a wave of strength E and phase φ can be seen as a vector with dimensions $E \cdot (cos(t+\varphi), sin(t+\varphi))$ rotating around origin in the complex plane. When multiple waves are incident at the some receiver, the magnitude of the vector sum of all incident waves thus gives the signal strength, leading to a need to store only a single vector in the buffer accumulating all incident waves.



A wave ondulating in the complex plane.[2]

Acknowledgement

We would like to thank NTNU and NVIDIA throught their CUDA Research Center and CUDA Teaching Center Programs for supporting the NTNU/IDI HPC-Lab led by Dr. Anne C. Elster, and Kongsberg Gruppen through Alexander Gosling for inspiring the topic.

Bibliography

[AMD10] AMD. *AMD CAL Programming Guide v2.0*, December 2010.

[App68] Arthur Appel. Some techniques for shading machine renderings of solids. 1968.

[Arv05] Johan Arvelius. Reflection angles, September 2005. URL: http://en.wikipedia.org/wiki/File:Reflection_angles.svg.

[Bab11] Kjetil Babington. Real-time ray tracing for the hpc-lab snow simulator. 2011.

[Bab12] Kjetil Babington. Terrain rendering techniques for the hpc-lab snow simulator. June 2012.

[CDB04] Zongqiang Chen, Alex Delis, and Henry K. Bertoni. Radio-wave propagation prediction using ray-tracing techniques on a network of workstations. 2004.

[Cla14] Lloyd D. Clark. Personal communication, 2014.

[Cri06] Cristian. Snell's law, November 2006. URL: http://en.wikipedia.org/wiki/File:Snells_law.svg.

[Eid09] Robin Eidissen. Utilizing gpus for real-time visualization of snow. February 2009.

[FFBE13] Thomas L. Falch, Jostein Bø Fløystad, Dag W. Breiby, and Anne C. Elster. Gpu-accelerated visualization of scattered point data. June 2013.

[FFH04] M. Farzaneh, I. Fofana, and H. Hemmatjou. Electrical properties of snow. 2004.

[Fra05] Jerrold Franklin. *Classical Electromagnetism*. Addison-Wesley, Temple University, 2005.

[GAHT09] Rober Grisso, Mark Alley, David Holshouser, and Wade Thomason. Soil electrical conductivity. 2009.

[Gee09] Geek3. Vector graphics of a sphere, October 2009. URL: http://en.wikipedia.org/wiki/File:Sphere_wireframe_10deg_6r.svg.

[Geo14] Electrical resistivity of rocks, 2014. URL: http://www.ualberta.ca/~unsworth/UA-classes/424/notes424-2014.html.

[Gje09] Aleksander Gjermundsen. Lbm vs. sor solvers on gpus and real-time snow simulations. December 2009.

[gon09] gonfer. Unfasor, February 2009. URL: http://en.wikipedia.org/wiki/File:Unfasor.gif.

[Lud09] Holger Ludvigsen. Ray tracing using nvidia optix. 2009.

[Lud10] H. Ludvigsen. Real-time ray tracing using nvidia optix. 2010.

[MB01] Alex Martinez and Alan P. Bymes. Modeling dielectric-constant values of geologic materials: An aid to ground-penetrating radar data collection and interpretation. December 2001.

[Mik13] Magnus Alvestad Mikalsen. Openacc-based snow simulation. June 2013.

[Mä96] Christian Mätzler. Microwave permittivity of dry snow. March 1996.

[Nor12] Andreas Nordahl. Enhancing the hpc-lab snow simulator with more realistic terrains and other interactive features. June 2012.

[Nor13] Lars Espen Strand Nordhus. Ray tracing for simulation of wireless networks in 3d scenes. June 2013.

[NVI12] NVIDIA. *NVIDIA OptiX Ray Tracing Engine Programming Guide*, November 2012.

[NVI14a] NVIDIA. *CUDA C Programming Guide*, February 2014.

[NVI14b] NVIDIA. Optix, 2014. URL: http://www.nvidia.com/object/optix.html.

[Pbr08] Pbroks13. Vector graphics of a pinhole camera, May 2008. URL: http://en.wikipedia.org/wiki/File:Pinhole-camera.svg.

[Ped11] Stian Aaraas Pedersen. Progressive photon mapping on gpus. 2011.

[Ped12] Stian Aaraas Pedersen. Ray traacing for solar system analysis on gpu. 2012.

[Pep09] Peppergrower. Phase shift, February 2009. URL: http://en.wikipedia.org/wiki/File:Phase_shift.svg.

[PS02] Pollack and Stump. *Electromagnetism*. Addison-Wesley, Michigan State University, 2002.

[RLNB65] P.L. Rice, A.G. Longley, K.A. Norton, and A.P. Barsis. Transmission loss predictions for tropospheric communication circuits. 1965.

[Sal06a] Ingar Saltvik. Parallel methods for real-time visualization of snow. June 2006.

[Sal06b] Ingar Saltvik. Parallel visualization of snow. June 2006.

[San13] Mads Buvik Sandvei. Evaluating wind field solvers for the ntnu hpc-lab's snow simulator. December 2013.

[Sch11] Schreiberx. Bounding volume hierarchy, December 2011. URL: http://en.wikipedia.org/wiki/File:Glasses_800_edit.png.

[Sco04] James H. Scott. Electrical and magnetic properties of rock and soil. August 2004.

[SMR07] Daniela Naufel Schettino, Fernando J. S. Moreira, and Cássio G. Rego. Efficient ray tracing for radio channel characterization of urban scenarios. 2007.

[SR92] Scott Y. Seidel and Theodore S. Rappaport. A ray tracing technique to predict path loss and delay spread inside buildings. 1992.

[Sun14] Dan Sunday. Intersection of rays and triangles (3d), 2014. URL: http://geomalgorithms.com/a06-_intersect-2.html.

[Tje13] J.S. Tjetjen. Anita longley's legacy: The longley-rice model - still going strong after almost 50 years [historical corner]. 2013.

[Tra09] Gilles Tran. Realistic ray-tracing image, October 2009. URL: http://en.wikipedia.org/wiki/File:Glasses_800_edit.png.

[TS09] Aila T. and Laine S. Understanding the efficiency of ray traversal on gpus. 2009.

[TT95] W.K. Tam and V.N. Tran. Propagation modelling for indoor wireless communication. 1995.

[Val93] Reinaldo A. Valenzuela. A ray tracing approach to prediction indoor wireless transmission. 1993.

[Ves12] Frederik Magnus Johansen Vestre. Enhancing and porting the hpc-lab snow simulator to opencl on mobile platforms. June 2012.

[Vik03] Torbjœrn Vik. Real-time visual simulation of smoke. June 2003.

[Whi80] Turner Whitted. An improved illumination model for shaded display. 1980.

[wik14a] Wikipedia's article on electromagnetic spectrum, 2014. URL: http://en.wikipedia.org/wiki/Electromagnetic_spectrum#Regions_of_the_spectrum.

[wik14b] Wikipedia's article on rodrigues' rotation formula, 2014. URL: http://en.wikipedia.org/wiki/Rodrigues%27_rotation_formula.

[wik14c] Wikipedia's article on spheres, 2014. URL: http://en.wikipedia.org/wiki/Sphere.