Simulated Rayleigh Scattering

by

James Nelson

A senior thesis submitted to the faculty of
Brigham Young University - Idaho
in partial fulfillment of the requirements for the degree of

Bachelor of Science

Department of Physics
Brigham Young University - Idaho
July, 2015
DEPARTMENT APPROVAL

of a senior thesis submitted by

James Nelson

This thesis has been reviewed by the research committee, senior thesis coordinator, and department chair and has been found to be satisfactory.

Date Todd Lines, Advisor

Date Evan Hansen, Committee Member

Date Kevin Kelley, Committee Member

Date Stephen McNeil, Department Chair
A simulator was created, which makes use of Monte Carlo to model the behavior of scattered electromagnetic radiation in a given medium. The program provides a flexible framework which can be adapted to explore various scattering phenomena from a semi-classic macroscopic point of view. Proper adaptation of the simulator does require some background knowledge of object-oriented programming. The geometry of the test and the output produced must be configured by the user. A test to simulate Rayleigh scattering, the most dominant type of scattering for visible light in our atmosphere, was implemented and compared against actual measurement of skylight both qualitatively and quantitatively. It was found that the generated data had the characteristics of natural skylight.
Acknowledgements

Thank you to Todd Lines for giving me the idea for this project, in the first place, and giving helpful feedback and insight through out it’s development. And thank you Matthew Brownell for configuring the Romney cpu cluster and providing technical support.
Contents

1 Introduction 1

1.1 Objective ............................................................... 1
1.2 Overview ............................................................... 1
1.3 Introduction to Rayleigh Scattering ......................... 2
1.4 Applications ........................................................... 5

2 Methods 9

2.1 Monte Carlo ............................................................. 9
2.2 System Requirements ............................................... 9
2.3 Simulator Conditions and Assumptions ................... 10
2.4 Modeling ................................................................. 11
2.5 Instructions ............................................................ 13

3 Results 15

3.1 Comparison with Skylight ......................................... 15
3.2 Conclusions ............................................................ 27
3.3 Future Research ........................................................ 27

A Simulator Source Code 31
## List of Figures

1.1  Abstract diagram of the components of the simulator. ........................................... 2
1.2  Class hierarchy of the simulator. .............................................................................. 3
1.3  This figure indicates a full hemisphere view of the direction and magnitude of polar-
    ization induced by Rayleigh scattering, as viewed from an observer looking up with
    the sun directly above them. Adapted from from [6]. ................................................... 4
1.4  The top part of the figure shows the intensity of scattering for the isotropic events.
    The middle section shows the distribution created by the anisotropic events. Every
    Rayleigh scattering event falls into one of these two categories. The last part shows
    the two of them added together. Taken from [5]. ......................................................... 6
3.1  This is a full sky image taken by Dr Shaw in Manoa Loa, Hawaii. Note the band of
    strongly polarized light. ............................................................................................... 16
3.2  This plot (provided by Dr. Shaw) shows Rayleigh scattering vs scattering angle (angle
    between original and scattered directions). Unpolarized light with a wavelength
    of 650 nm, and particles with a 1 nm radius. ............................................................... 17
3.3  Angle from source vs angle of polarization (relative to the z-axis) .......................... 17
3.4  Angle from source vs angle of incidence (relative to the z-axis) .............................. 18
3.5  Angle from source vs frequency ................................................................................. 19
3.6  Wavelength dependence of Rayleigh Scattering (adapted from Dr. Shaw) ............... 20
3.7  Histogram of the number of detected scattered photons sorted by frequency. ........... 21
3.8  Frequency vs angle of polarization (from the Z-axis) ................................................. 22
3.9  Frequency vs angle of incidence (from the Z-axis) ..................................................... 23
3.10 Angle from source vs relative polarization ............................................................... 24
3.11 Frequency vs Angle of Polarization (from the Z-axis) .............................................. 25
3.12 Histogram showing the distribution of relative polarizations detected with an angle
    of incidence of 45 degrees. ...................................................................................... 26
1.1 Objective

The goal was to write a simulator that can accurately model the behavior of electromagnetic radiation traveling through a medium. By implementing known details at the microscopic level, such as calculation of the cross section and the probability distribution of the scattered photons that could be produced, the model should produce distributions of data that have the same macroscopic qualities and dependencies as physically observed in that type of scattering.

1.2 Overview

The program uses very generic and adaptable classes that can be useful in a wide variety of tests. Then, more specific sub-classes were implemented to represent specific, known physical characteristics. The end result of these sub-classes is a simulator that makes use of Monte Carlo to give results that are statistically reliable, if run a sufficient number of times. The program allows the user to specify the conditions and configuration of the test, as well as which resulting data is relevant to include in the output. The simulator was used to model visible light as it passes through the air, for the idealistic case of Rayleigh scattering only, and the generated results were validated against actual measurement.

Figure 1.1 illustrates the basic idea behind the simulator. Photons are emitted from a source, and after being scattered in a medium may or may not be absorbed by the detector. Each simulation
must include the locations and dimensions of the source, medium, and detector class configured. Every detected photon will be included in the output of the program. Photons that are not detected will not be output.

A basic overview of the class structure is provided in figure 1.2. The detector, medium, and source class are the generic parent classes for more specific implementations. Together with the photon class, these make up the main components of a simulator implementation.

### 1.3 Introduction to Rayleigh Scattering

Rayleigh scattering occurs when incident photons interact with spherical or very nearly spherical particles whose radius is much smaller that the wavelength of the incident light. This condition is met in the case of visible light and the main components of our atmosphere (N2, O2, Ar, and CO2). It also happens that visible light and radio waves are the only two ranges in the electromagnetic
Figure 1.2: Class hierarchy of the simulator.

spectrum where absorbance is greatly reduced and nearly all interactions result in a scattered photon of the same energy (inelastic scattering) rather than converting some or all of that energy into thermal motion of the air particles.

Rayleigh scattering was originally discovered and investigated by John William Strutt, the third Lord of Rayleigh, in 1871. He correctly determined that this scattering is inversely proportional to the wavelength raised to the fourth power. The equation to calculate the intensity of an unpolarized beam of Rayleigh scattered light is

\[
I = \frac{I_0}{r^2} \alpha^2 \left( \frac{2\pi}{\lambda} \right)^4 \frac{1 + \cos^2(\theta)}{2},
\]

where \(I_0\) is the original intensity, \(r\) is the distance traveled, \(\alpha\) is a constant related to the polarizability of a particle, \(\lambda\) is the wavelength of the light, and \(\theta\) is the azimuthal angle from the original direction. In other words, the degree that Rayleigh scattering tends to affect light is highly dependent on the wavelength of the light. This make the air somewhat like a big, fuzzy prism.

It also has a tendency to induce a partial polarization in completely unpolarized light. More specifically, light scattered perpendicular to the beam will be strongly polarized in the azimuthal direction compared to that beam as indicated in figure 1.4. That is how anti-glare lenses work.
If the sun is directly above you, most of the light reflecting off the horizon from all directions in vertically polarized. So a lens that filters vertically polarized light can remove most of the glare without impeding visibility. Turn the lens 90 degrees and the glare comes back. Both [1] and [4] give a more thorough treatment of polarization and the scattering of light.

In the context of visible light, Rayleigh scattering is the most dominant type of scattering in our atmosphere. It is responsible for the blue hue that can be observed in the sky on a cloudless day. This happens because photons with higher frequencies (green, blue, and purple) are exponentially more likely to be scattered than lower frequency photons (such as red, orange, and yellow). If you look in any direction other than directly at the sun, you are only seeing photons that have scattered at least once and probably several times, and so the vast majority of them are these higher energy photons that together appear blue. It is for this same reason that we observe red orange and yellow shades at sunset or sunrise. At those angles, the atmosphere that the light must travel through to reach an observer on the ground is much thicker. The higher frequency photons have scattered so many times while traveling that they have become diffuse, and most of them scatter in some
direction other than that of the original beam. Lower frequency photons on the other hand, have
had a chance to scatter just the needed number times without being filtered out.

This phenomena is really the result of a pair of possible scattering events, both of which are
depicted in the figure. One type of event is completely isotropic. It does not affect the photon in
anyway, except to change its trajectory, and the transmitted photon has an even probability of going
in any direction. The other type of event is anisotropic, though it is still azimuthally symmetric
with respect to the original photon. This second type of event strongly favors forward and reverse
scattering, but still does not change the frequency or energy of the photon. The determining factor
between the two types is whether or not the orientation of the oscillation of the photon’s electric
field lies in the plane of scattering. When it does, the event will be anisotropic, and when it is
perpendicular to the plane of scattering the event will be isotropic. For further details see [5] and
[6].

Rayleigh scattering has a fairly strong effect on the physical measurement of skylight. These
effects can cause some degree of systematic error, especially in fields such as meteorology and
astronomy. If these effects can be properly accounted for, such systematic errors can be removed.

1.4 Applications

One of the simpler application of this model is the simulation of a solar eclipse. After configuring
the test for visible light through an atmosphere of sufficient size, simply remove all unscattered
light from the results. A lunar eclipse could be modeled in much the same way, after a lunar light
source is implemented. Given that these events are rare and short lived in any one location, it could
be useful to simulate them in advance in order to investigate what will be relevant to measure and
what you expect to find.

There are other radiation scattering scenarios that could benefit from such a model, such as
laser light or microwave radiation passing through a known medium. It should be noted however,
that fully quantum mechanical phenomena, such as interference through a diffraction grating, are
not present in this framework. It is intended that this program be utilized when macroscopic effects
Figure 1.4: The top part of the figure shows the intensity of scattering for the isotropic events. The middle section shows the distribution created by the anisotropic events. Every Rayleigh scattering event falls into one of these two categories. The last part shows the two of them added together. Taken from [5].
are the focus of the tests. It should be feasible to extend this simulator to account for other types of scattering, such as Mie or Compton.
Chapter 2

Methods

2.1 Monte Carlo

In a Monte Carlo test a simple situation is repeated numerous times in an attempt to draw conclusions or find relationships that would be arduous to calculate analytically. The analytic results, whether known before hand or not, will be asymptotically approached with more and more iterations of the test. A good example of this is found in the calculation of $\pi$. To use Monte Carlo to calculate $\pi$, just take a squared circle, and start picking random points in the square. The ratio of points inside and outside the circle can be used to estimate the value of $\pi$, and the greater the number of points selected, the more closely this ratio will be to the exact value. An example of this test is also included in this project, within the PiApproximation class. Monte Carlo is effective for most highly repetitive processes, such as millions of photons traveling through an approximately uniform medium.

2.2 System Requirements

All of the code in this simulator was written in C++. Some coding is required to configure the test, so it is expected that the user have a working knowledge of C++. This simulator was run in Centos 6. The output files were graphed using gnuplot, and the simulator does make use of the
 GNU scientific library (GSL) both for scientific constants, and random number generation. Both of these packages should be installed before the simulator can be compiled and run.

2.3 Simulator Conditions and Assumptions

This project consists of a simulator class which encapsulates everything else. The user is expected to provide a main method which instantiates and configures this class. The simulator has a source, through which all photons will be generated. The simulator also has a medium through which the photons will scatter and a detector. The number of photons to generate and process must also be set by the user. Note that this number represents the number of photons that will be generated, not the number that will be captured by the detector. When run is called, a file name must be specified to which all of the data will be sent. A good example of this can be found in the ValidationSimulation.cpp file’s format method.

There are helper classes provided for matrix rotation in 3D space, and making histograms. The output files were graphed using gnuplot, and the simulator does make use of the GNU scientific library (GSL) both for scientific constants, and random number generation. In all cases SI units were used to keep everything consistent. Also, the program calls for degrees over radians when working with angles.

There are a number of random number generators (RNGs) offered in the GSL. The mt_19937 was the one used in this project. The documentation indicates that this implementation produces numbers quickly, and that these numbers are uncorrelated. This feature was essential because for each scattering event the simulator requires several independent random numbers. The other important factor was that this random number generator has a very long period before repeating numbers for any given seed. The calls to the RNG are in a singleton class, so switching implementations in the future would be practical. This implementation is located in the RandomSingleton class.

Some of the 3D models were implemented as spheres in an effort to make them flexible and intuitive. The medium class is a spherical cloud of whatever material is being represented. By
changing the size and configuration or even using multiple instances, many shapes can be approximated. Detectors, too, are represented with the sub-surfaces of a sphere. Rectangular detectors can be approximately represented by setting the angles of validation to be small and the radius to be large.

2.4 Modeling

When implementing the photon class, it was appropriate to give a semi-classical treatment of photons. The scenarios being modeled are assumed to be larger than a few centimeters. Photons are assumed to travel in a straight Euclidean line at the speed of light. Therefore, there are three coordinates associated with position and velocity (in Cartesian coordinates). The three coordinates associated with velocity allow the user to indicate the direction. The normalization of these coordinates is not important, only the direction indicated by their relative magnitude. Each instance of the photon class also has a frequency, from which wavelength and energy are calculated. There is an orientation variable for keeping track of the direction of the electric oscillation. Circular polarization is also represented, but only as being clockwise, counter-clockwise, or not present, which is the default. Each photon also has three coordinates associated with its point of origin.

Sources are modeled as being points from which photons are generated. Valid angles for emission can easily be set, as well valid ranges of frequencies. The distribution of these quantities will be uniform unless additional implementation is provided in a subclass written by the user.

As mentioned above, detectors are spherical sub-surfaces with a center position, a radius, a full angle phi, and a half angle theta (spherical coordinates) that defines what portion of the surface will detect photons. Currently the detectors are ideal, meaning that incident radiation has a 100% chance of being absorbed and detected. Future work could be done to make models that showed special sensitivity to certain frequencies or angles of incidence.

The medium class has a position, and a radius. It also has functions for calculating the free path distance a given photon will travel through the material before a scattering event. There is a similar function for calculating the free time a photon will travel between scattering events for
added flexibility. It is in the subclasses of the medium class where code to calculate the cross-section for scattering or characteristics of the resulting photon should live, as explained below.

The Rayleigh class has the logic for the individual scattering events. As explained in [3] the mechanism that determines whether a scattering event is isotropic is the difference between the orientation of the electric oscillation and the plane of scattering. When these quantities are perpendicular the event will be isotropic, and when they are parallel the event will be anisotropic. At all angles in between there are components in both directions and proportional chances of each occurrence. It should be noted that the angle of scattering requires an inverse trigonometric function; otherwise the density of selected directions will be higher at the poles than in the azimuthal directions. To achieve the correct distribution for the anisotropic case it is necessary to first apply a normalization function that favors forward and reverse scattering before applying this correction.

The Atmosphere class is a subclass of the Rayleigh class which further specifies how the cross section to interaction is to be found. To be useful, the free distance (or time) a photon travels should not be uniform. In the case of Rayleigh scattering (and several other types) it should be a Boltzmann distribution based on the cross section of interaction. Calculating the cross section of electromagnetic radiation through different materials is an involved process, and techniques will vary in each case. One method of calculation for visible light through air has been detailed in [2]. The article explains that given the index of refraction, the King factor of each gas (which is a term related to calculating depolarization), and a few polynomial approximations, the cross section can be found. By treating each of the four most common gasses in air as partial pressures with their own distributions, four separate values are generated for the free distance from four distributions. The shortest one is selected, and the others discarded.

The simulator class holds everything together. It has a source, a medium, a detector, and a given number of times to generate a photon. Logic is provided to check for detection before, during, and after the photon traverses the medium, thus giving more freedom to the user to use any configuration they wish. There is also a check to discard any photon that leaves the medium on a trajectory that does not go toward the detector. A test then consists of calling the main method, which configures the members of the simulator, instantiates it, and runs it the specified number of
times with a given output file. All detected photons are sent to the format function, and their data is sent to the output file.

2.5 Instructions

In order to utilize this framework, it will probably be necessary to subclass some of these general implementations and a thorough example of how to do that is provided in the SimulateAtmosphere program. In order to validate the model a uniform unpolarized visible light source was written, as well as a detailed Atmosphere class that inherited some scattering logic from the Rayleigh class. The simulator has visible light emitted toward an atmospheric medium with a radius of 100 km toward a large detector.

If a photon is detected, the format function is called, and there is a default implementation to send all of the data that is present in a photon after the last scattering event to the output file. In most cases the user will want to specify what data to send to the file and in what order. In many cases the user will also want to do some pre-processing on the data at this stage, since the pertinent details will vary widely in each case (such as calculating the difference between a photon’s orientation and its angle of incidence and outputting that value instead). This specification is achieved by overriding the format function of the Simulator class to provide the relevant data in the desired format. The scope of this function has access to all members of the Simulator class as well as all values stored within the photon instance. In the atmosphere simulation example, data is formatted according to frequency, various angles, degrees of polarization, and additional data is stored in histograms to be sent to other output files later. These histograms are just objects of the Histogram class that bin data in an array to show frequency. Check the SimulateAtmosphere class in Appendix A for the actual implementation.

Running a simulation of only a million photons took only a few minutes, but in order to have confidence in the results of any Monte Carlo simulator, it should run a very large number of times (1,000,000 times should probably be considered a minimum for most cases). The simulator will periodically indicate what percent it is toward completion, but it may still be desirable to run the
program as a background process for longer tests.

To run the Simulate Atmosphere program, find the directory that contains the executable in a terminal, and type 

```
./SimulateAtmosphere.out <number of photons to generate> <name of test>
```

The main output file will be `<name of test>-<host name of the computer>.out`. There will also be `<name of test>-<host name of the computer>-f.out` and `<name of test>-<host name of the computer>-p.out` files containing histograms of the frequencies and polarizations respectively. This means you can start the program on multiple machines using mpi or a script and not worry about them writing to the same file. You also do not have to worry about them having duplicate seeds, as each execution gets reseeded in the main method based on the time stamp and the host name.
Chapter 3

Results

3.1 Comparison with Skylight

Dr. Joseph A. Shaw is a professor at Montana State University. He has spent many years measuring and analyzing skylight. His findings have provided good data about the specifics of both skylight in general and Rayleigh scattering specifically that can be used to verify the model, as found in [7]. There are two key traits that a Rayleigh scattering model must have for skylight. First, blue (higher frequency) light must scatter more readily and at wider angles than red (lower frequency) light. For all unscattered photons detected, blue should be more abundant at all angles of incidence and especially so as we approach 90 degrees. Second, Dr. Shaw references this equation,

\[
DOLP = \frac{L_s - L_p}{L_s + L_p} = \frac{1 - \cos^2(\theta)}{1 + \cos^2(\theta)} = \frac{\sin^2(\theta)}{1 + \cos^2(\theta)},
\]

which can be used to calculate the degree of polarization of the scattered light as a function of angle of incidence. According to equation 3.1, the scattered light should tend to be polarized in the azimuthal direction, reaching a maximum of 100% at 90 degrees, and 33.3% at 45 degrees (which was the condition this model checked for). Real skylight does not quite reach 100% polarization at 90 degrees of incidence because other types of scattering are present, but it does come fairly close. Figure 3.1 shows an image taken from a cloudless sky in Hawaii. Additional explanation is found
Figure 3.1: This is a full sky image taken by Dr Shaw in Mauna Loa, Hawaii. Note the band of strongly polarized light.

in [7].

Figure 3.2 shows the intensity of scattering at different angles of incidence (Shaw). Note that the vertical axis is logarithmic. Figure 3.3 really serves two purposes. It shows that the scattering is truly azimuthal. It also highlights the angles at which the photons tend to be detected with this geometry.

The configuration used was effective for angles of incidence up to about 50 degrees. Much of the data beyond that point affected by the geometry of the detector. Observe the square boundary in figure 3.4. We can see this square pattern in the second plot at about the 50 degree mark.

The graph in figure 3.5 shows the correlations of frequency with angle of approach relative to an unscattered beam from the source to the point on the detector where the photon made contact. The solid line on the left side represents all of the photons in the test that did not scatter. Note also how the number of scattered photons and the angle from source increase with frequency. Higher frequency photons are considerably more likely to scatter more frequently and at wider angles, just as we would expect. Compare with this diagram provided by Dr. Shaw in figure 3.6.

The number of scattered photons detected at 45 degrees were grouped by frequency in a his-
Figure 3.2: This plot (provided by Dr. Shaw) shows Rayleigh scattering vs scattering angle (angle between original and scattered directions). Unpolarized light with a wavelength of 650 nm, and particles with a 1 nm radius.

Figure 3.3: Angle from source vs angle of polarization (relative to the z-axis)
Figure 3.4: Angle from source vs angle of incidence (relative to the z-axis)
Figure 3.5: Angle from source vs frequency
Figure 3.6: Wavelength dependence of Rayleigh Scattering (adapted from Dr. Shaw)

ogram (figure 3.7). Photons with a higher frequency were much more likely to be detected with a higher degree of incidence and have a higher degree of polarization in the azimuthal direction. I believe that the drop on the right side of the graph is because of the volume and constant density of the medium. When light travels through more atmosphere, the highest frequency light starts to get filtered out. It may also be another manifestation of the geometry of the test. The photons with the very highest degrees of scattering don’t get detected, and these are made up almost entirely of higher frequency photons.

The plots in figures 3.8 and 3.9 show the number of photons detected by frequency for relative polarization and angle of incidence respectively. The main difference is that the second plot groups all unscattered photons in the band at the bottom. So, with the unscattered band, more low frequency photons were detected, but without it mostly high frequency photons were detected. We can therefore infer that the medium appears blue from the perspective of the detector and also at all other angles since blue light is what is escaping from the system.

Figure 3.10 clearly demonstrates a tendency of the photons to be polarized perpendicularly to
Figure 3.7: Histogram of the number of detected scattered photons sorted by frequency.
Figure 3.8: Frequency vs angle of polarization (from the Z-axis)
Figure 3.9: Frequency vs angle of incidence (from the Z-axis)
their angle of approach to the detector at all valid test angles. There are two clouds of photons at
the 90 and 270 degree mark, and relatively lower densities of detected photons at 0 (and 360) and
180 degrees.

Again we can observe the distinct correlation of polarization and azimuthal angle of approach
in this generated plot. The two diagonal bands in figure 3.11 show that as you alter the direction
of approach for the incident photon, there are maximal distributions of photons whose polarization
is perpendicular to that direction and minimal distributions that are aligned or anti-aligned with
it.

Lastly the histogram in figure 3.12 was for the detected photons with an angle of incidence
of 45 degrees. It shows a clearer picture of these bands, and was used to estimate the degree of
Figure 3.11: Frequency vs Angle of Polarization (from the Z-axis)
polarization. We expect that the degree of polarization will increase from 0 to 90 degrees. We can use this data to infer the degree of polarization. The area under curve at angles with a positive bias minus the area under the angles with a negative bias divided by the total area is roughly equal to

\[ P = \frac{(220 \text{ photons} \times 180 \text{ degrees}) - (110 \text{ photons} \times 180 \text{ degrees})}{(220 \text{ photons} + 110 \text{ photons}) \times 180 \text{ degrees}} = 33.3\%. \] (3.2)

This is the degree of polarization predicted by equation 3.1 for 45 degrees of incidence.
3.2 Conclusions

We can clearly observe the first macroscopic characteristic we were looking for. As the wavelength of an incident beam increases, it becomes less likely to scatter by that same amount raised to the fourth power. And indeed, our results show a line very similar to the one provided by Dr. Shaw up to about 700 THz (remember that the generated graph is reversed since the x axis in frequency rather than wavelength). If the test were run again with a thinner atmosphere perhaps the correlation would be even stronger. The correlation of wavelength and intensity of scattering is affected by the change in the magnitude of the cross section rather than being inherent in any given scattering event. In other words, the scatter function in the file Rayleigh.cpp affects all photons the same, regardless of their frequency (or wavelength). The getFreeDistance function on the other hand, gives much lower values for photons with higher frequency because the cross section used to calculate this distance is highly frequency dependent.

It is actually the selection mechanism for isotropic scattering in individual events that causes the azimuthal polarization pattern. At no point in the code do we cause a biased change to a photon’s polarization. In the Rayleigh class the scattering function does selectively change the linear polarization on a photon, but only by giving it a new evenly distributed random value. In earlier iterations of this model various biased changes were tested, but the results always tended to yield either too much or the wrong kind of linear polarization. By randomly changing the orientation of a photon for anisotropic scattering and leaving unchanged for isotropic scattering, a very close match to the predicted degree of linear polarization at 45 degrees of incidence is achieved, and that polarization is in exactly the right direction.

3.3 Future Research

This framework can be used to estimate the effects of Rayleigh scattering for other radiation related projects. If the distribution of generated light and the variable density of the atmosphere were added then a simulation could be made to give quantitative measurements of the sky, such as what is the exact color we see when the sun reaches a certain angle above the horizon. If the type and
thickness of the medium are known, as well as the exact distribution of incoming light, accurate predictions can be made about how the Rayleigh scattered light will be reflected. It may also be useful to make predictions about the prevailing conditions in the upcoming solar eclipse. With a little work this model can be extended to model other types of scattering as well.

This simulator could be made to run in parallel on the Linux cluster in the Romney building by making use of the message passing interface (mpi). This would allow detailed tests to be run on the scale of hundreds of millions of photons in a manageable amount of time. It could also be improved by creating an interface that would allow the user to configure and run a test without having to code a main method, even if it came at the cost of a little flexibility.
Bibliography


Appendix A

Simulator Source Code

Atmosphere.h

/*******************************************************************************
 * James Nelson
 * Atmosphere Header
 * 1-9-2015
*******************************************************************************
#ifndef ATMOSPHERE_H
#define ATMOSPHERE_H

#include <gsl/gsl_const_num.h>
#include "Rayleigh.h"
#include "Photon.h"

/*********************************************************
 * Atmosphere Class
 * A medium through which photons may pass.
***********************************************************/
class Atmosphere : public Rayleigh
{
private:
    enum Gas {N2, O2, AR, CO2};
    // Percient composition by volume
    const static double COMPOSITION[4];
    // Density by mole/m^3
    const static double DENSITY = 22414;
    const static double NA = GSL_CONST_NUM_AVOGADRO;

double getCrossSection(const Photon &photon, Gas gas) const;
double getKingFactor(const Photon &photon, Gas gas) const;
double getDensity(Gas gas) const;
double getRefractiveIndex(const Photon &photon) const;

public:
    Atmosphere() : Rayleigh() {}
    Atmosphere(Point point) : Rayleigh(point) {}
    Atmosphere(double * coordinate) : Rayleigh(coordinate) {}
    Atmosphere(double radius) : Rayleigh(radius) {}
    Atmosphere(Point point, double radius) : Rayleigh(point, radius) {}
    Atmosphere(double * coordinate, double radius) :
        Rayleigh(coordinate, radius) {}
    Atmosphere(const Medium & medium) : Rayleigh(medium) {}
    virtual double getFreeDistance(const Photon &photon) const;
    virtual double getFreeTime(const Photon &photon) const;
};
#endif
Atmosphere.cpp

/***************************************************************************/
* James Nelson
* Atmosphere Class
* 1–9–2015
* For more information see:
***************************************************************************/
#include <iostream>
#include <math.h>
#include "Atmosphere.h"
#include "Photon.h"
using namespace std;

const double Atmosphere::COMPOSITION[4] = {0.78084, 0.20946, 0.00934, 0.00036};

/***************************************************************************/
* Get Density
* Get the density of the gas (% by volume * mole/m^3 * NA).
***************************************************************************/
double Atmosphere::getDensity(Gas gas) const
{
    return COMPOSITION[gas] * DENSITY * NA;
}

/***************************************************************************/
* Get Cross Section
* The cross section is used in determining how far a photon travels before being scattered.
***************************************************************************/
double Atmosphere::getCrossSection(const Photon &photon, Gas gas) const
{
    double ri = getRefractiveIndex(photon);
    double ri2 = pow(ri, 2); // ri^2
    double density = getDensity(gas); // molecules per m^3
    double kf = getKingFactor(photon, gas);
    return 24 * pow(M_PI, 3) * pow(ri2 - 1, 2) / (pow(photon.getWavelength(), 4)
        * pow(density, 2) * pow(ri2 + 2, 2)) * kf;
}
* Get King Factor
* The king factor is related to the depolarization constant
* of a gas, and is need to calculate the cross section. It
* varies from one gas to another.

```cpp
double Atmosphere::getKingFactor(const Photon &photon, Gas gas) const
{
    double kingFactor = 0;
    switch (gas)
    {
        case N2:
            kingFactor = 1.034 + .000317 / pow(photon.getWavelength(), 2);
            break;
        case O2:
            kingFactor = 1.096 + .001385 / pow(photon.getWavelength(), 2) +
                         .0001448 / pow(photon.getWavelength(), 4);
            break;
        case AR:
            kingFactor = 1.00;
            break;
        case CO2:
            break;
    }
    return kingFactor;
}
```

* Get Refractive Index
* The refractive index of air for a given frequency.

```cpp
double Atmosphere::getRefractiveIndex(const Photon &photon) const
{
    double one_over_lambda2 = pow(photon.getWavelength(), -2);
    return (8.6051 + 2480990 / (132.274 - one_over_lambda2) + 17455.7 /
             (39.32957 - one_over_lambda2)) * .00000001 + 1;
}
```

* Get Free Distance
* Return the distance a photon will travel before an
* interaction (does not need to be the same each time).

```cpp
double Atmosphere::getFreeDistance(const Photon &photon) const
```
// calculate the distance the photon will travel because of each partial pressure of gas
double distance[4] = {0, 0, 0, 0};
// based on a correct random distribution that is calculated independently from the other distances
distance[0] = - (log(1 - getRS().rand())) / (getDensity(N2) *
    getCrossSection(photon, N2));
distance[1] = - (log(1 - getRS().rand())) / (getDensity(O2) *
    getCrossSection(photon, N2));
distance[2] = - (log(1 - getRS().rand())) / (getDensity(Ar) *
    getCrossSection(photon, N2));
distance[3] = - (log(1 - getRS().rand())) / (getDensity(CO2) *
    getCrossSection(photon, N2));
int n = 0;
int i;
// select the shortest of these distances
for (i = 1; i < 4; i++)
{
    if (distance[i] < distance[n])
    {
        n = i;
    }
}
return distance[n];

/*****************************/
* Get Free Time
* Return the time a photon will travel before an interaction (does not need to be the same each time).
*****************************/
double Atmosphere::getFreeTime(const Photon &photon) const
{
    return getFreeDistance(photon) / Photon::getC();
}
Detector.h

/******************************************************************************
 * James Nelson
 * Detector Header
 * 1–23–2015
******************************************************************************

#ifndef DETECTOR_H
#define DETECTOR_H

#include "Photon.h"
#include "Point.h"
#include "SolidAngle.h"

/******************************************************************************
 * Detector Class
 * This class simulates light detector.
******************************************************************************

class Detector
{
private:
  Point location;
  Point lastContact;  // Last registered point of contact
  SolidAngle theta;   // 0 to pi
  SolidAngle phi;     // 0 to 2 pi
  double radius;
  double numDetected;
  bool isNarrowing(const Point & original, const Point & next) const;
  double getMinDistance(const Photon & photon, double & time) const;
  Point getContact(const Photon & photon, double & minTime);

public:
  Detector() : theta(false), phi(true) { location; lastContact; radius = 1;
    numDetected = 0;};
  Detector(double * coordinate, double thetaMin, double thetaMax,
            double phiMin, double phiMax, double radius);
  Detector(const Detector & detector) : location(detector.getLocation()),
    lastContact(detector.getLastContact()), theta(detector.getTheta()),
    phi(detector.getPhi())
    {radius = detector.getRadius(); numDetected = 0;};
  Point getLocation() const { return location;};  // get the location
  SolidAngle getTheta() const { return theta;};  // get the vertical angle
  SolidAngle getPhi() const { return phi;};  // get the horizontal angle
  double getRadius() const { return radius;};
  double getNumDetected() const { return numDetected;};
};
Point getLastContact() const { return lastContact; };
void setLocation(int min, double max) { theta.setMin(min);
    theta.setMax(max); }
void setTheta(double min, double max) { theta.setMin(min);
    theta.setMax(max); }
void setPhi(double min, double max) { phi.setMin(min); phi.setMax(max); }
void setRadius(double r) { if (r >= 0) radius = r; }
void setNumDetected(double n) { if (n >= 0) numDetected = n; }
virtual bool detected(const Photon &photon, double amount,
    bool isTime = true);
# include <iostream>
# include <cmath>
# include "Detector.h"
using namespace std;

Detector::Detector(double *coordinate, double thetaMin, double thetaMax,
double phiMin, double phiMax, double radius) : lastContact(),
theta(thetaMin, thetaMax, false), phi(phiMin, phiMax, true)
{
    for (int i = 0; i < Point::SIZE; i++)
    {
        location.setCoordinate(i, coordinate[i]);
    }
    this->radius = radius;
    numDetected = 0;
}

bool Detector::isNarrowing(const Point &original, const Point &next) const
double dOriginal = getLocation().getDistance(original);
double dNext = getLocation().getDistance(next);
double dCheck = original.getDistance(next);
return pow(dOriginal, 2) + pow(dCheck, 2) >= pow(dNext, 2);

// Get Min Distance
Get the minimum distance from the center of this
detector that a photon will be at any given point along
the path is will traverse in the given time. Update the
amount of time it took to get there. If 0 or the full time
are the optimal values in the given range, return those
values.

double Detector::getMinDistance(const Photon & photon, double & time) const
{
    Photon p = photon; // since we are not going to change photon
    p.advance(time); // move the photon the full time along its trajectory
    double dInitial = photon.getLocation().getDistance(location); // 0 time
    double dFinal = p.getLocation().getDistance(location); // full time
    if (dInitial < dFinal && !isNarrowing(photon.getLocation(), p.getLocation()))
    {
        // the closest point is behind the photon
        time = 0;
        return dInitial;
    }
    else if (dInitial > dFinal && !isNarrowing(p.getLocation(),
        photon.getLocation()))
    {
        // the closest point is farther ahead than the photon had time to travel to
        return dFinal;
    }
    // normalization constant for the componenets of the velocity of the photon
    double n = p.normalize();
    double vx = photon.getVelocity()[0] * n;
    double vy = photon.getVelocity()[1] * n;
    double vz = photon.getVelocity()[2] * n;
    double px = photon.getLocation()[0];
    double py = photon.getLocation()[1];
    double pz = photon.getLocation()[2];
    double lx = getLocation()[0];
    double ly = getLocation()[1];
}
double lz = getLocation()[2];
// this equation is derived by realizing the dot product of the velocity
// of the photon and the vector from the photon to the detector’s location
// must be zero at the point we are seeking
    time = (vx * (lx - px) + vy * (ly - py) + vz * (lz - pz)) / (pow(vx, 2) +
            pow(vy, 2) + pow(vz, 2));
p = photon;
p.advance(time);
    return p.getLocation().getDistance(getLocation());
}

/****************************
* Get Contact
* Calculate the point of contact the photon will make.
* Assume that the minTime is the time to get to the closest
* point along the path, and that the photon will be at that
* point be within the radius. Set the calculated point as
* the last contact registered to this detector.
****************************/
Point Detector::getContact(const Photon & photon, double & minTime)
{
    Photon p = photon;
    int n = 1;
    double t = minTime / pow(2, n++);
    double tPrev = 0;
    double d = 0;
    double dPrev = photon.getLocation().getDistance(location);
    // binary search
    while (d < radius && t != tPrev)
    {
        p = photon;
p.advance(t);
        d = p.getLocation().getDistance(location);
        t = tPrev;
        if (d < radius) // we are inside the radius, t should be smaller
        {
            t -= minTime / pow(2, n++);
            dPrev = d;
        }
        else if (d > radius) // we are outside the radius, t should be bigger
        {
            t += minTime / pow(2, n++);
            d = dPrev;
        }
    }
lastContact = p.getLocation(); // keep track of this point
return lastContact;

/**
 * Detected
 * Determine whether the given photon will be absorbed by
 * the detector. Note: if this method returns true then the
 * point of last contact will have been reset.
 */
bool Detector::detected(const Photon & photon, double amount, bool isTime)
{
    if (photon.getLocation().getDistance(location) < radius)
    {
        // already past the detector
        return false;
    }
    double time;
    if (isTime) // they gave us time to travel
    {
        time = amount;
    }
    else // they gave us distance to travel
    {
        time = amount / Photon::getC();
    }
    double minTime = time;
    double minDistance = getMinDistance(photon, minTime);
    if (minDistance > radius) // we never get close enough
    {
        return false;
    }
    Point contact = getContact(photon, minTime); // get where we hit
    double r = sqrt(pow(contact.getCoordinate(0), 2) +
                     pow(contact.getCoordinate(1), 2)); // sqrt(x^2 + y^2)
    // angles of point of contact from the detectors location
    double anglePhi = atan2(contact.getCoordinate(1) - location.getCoordinate(1),
                             contact.getCoordinate(0) - location.getCoordinate(0)) * 180 / M_PI;
    double angleTheta = atan2(r, contact.getCoordinate(2) -
                              location.getCoordinate(2)) * 180 / M_PI;
    // true if the point was in valid range
    return phi.isInRange(anglePhi) && theta.isInRange(angleTheta);
#ifndef HISTOGRAM_H
#define HISTOGRAM_H

#include <iostream>
#include <string>

using namespace std;

/* ***********************
 * James Nelson
 * Histogram Header
 * 3-25-2015
***************************/

class Histogram
{
    protected:
        int* bins;
        unsigned int numBins;
        double binSize;
        double initial;
    public:
        Histogram(unsigned int numBins, double binSize, double initial);
        Histogram(const Histogram& h);
        ~Histogram()
        {
            delete[] bins;
        }
        int* getBins() const { return bins; }
        unsigned int getNumBins() const { return numBins; }
        double getBinSize() const { return binSize; }
        double getInitial() const { return initial; }
        string toString() const;
        double operator[](int i) const
        {
            if (i >= 0 && i < numBins) return bins[i];
            else return 0;
        }
        double operator[](double n) const
        {
            int i = (int)((n - initial) / binSize);
            if (i >= 0 && i < numBins) return bins[i];
            else return 0;
        }
        void operator <<(double n)
        {
            int i = (int)((n - initial) / binSize);
            if (i >= 0 && i < numBins) bins[i]++;
        }
};
#endif
#include <sstream>
#include "Histogram.h"

using namespace std;

Histogram::Histogram(unsigned int numBins, double binSize, double initial)
{
    this->numBins = numBins;
    this->binSize = binSize;
    this->initial = initial;
    bins = new int[numBins];
    for (int i = 0; i < numBins; i++)
    {
        bins[i] = 0;
    }
}

Histogram::Histogram(const Histogram & h)
{
    this->numBins = h.numBins;
    this->binSize = h.binSize;
    this->initial = h.initial;
    bins = new int[h.numBins];
    for (int i = 0; i < h.numBins; i++)
    {
        this->bins[i] = h.bins[i];
    }
}
/***************************************************************
 * To String
 * Return "bin-initial bin-final bin-count\n" for all bins.
***************************************************************/
string Histogram::toString() const
{
    stringstream stream;
    for (int i = 0; i < numBins; i++)
    {
        stream << (initial + i * binSize) << " " << (initial + (i + 1) * binSize)
                << " " << bins[i] << "\n";
    }
    return stream.str();
}
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# S c a t t e r i n g
# James Nelson
# 1−8−15
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
all: PiApproximation.out TestDetector.out TestSource.out
   TestVisibleSource.out TestPhoton.out TestMedium.out
   TestRayleigh.out TestPoint.out TestSolidAngle.out
   TestRotation.out TestRandomSingleton.out TestAtmosphere.out
   TestSimulation.out TestHistogram.out SimulateAtmosphere.out
   echo "Done"

PiApproximation: PiApproximation.out
   ./PiApproximation.out
TestDetector: TestDetector.out
   ./TestDetector.out
TestSource: TestSource.out
   ./TestSource.out
TestVisibleSource: TestVisibleSource.out
   ./TestVisibleSource.out
TestPhoton: TestPhoton.out
   ./TestPhoton.out
TestMedium: TestMedium.out
   ./TestMedium.out
TestRayleigh: TestRayleigh.out
   ./TestRayleigh.out
TestPoint: TestPoint.out
   ./TestPoint.out
TestSolidAngle: TestSolidAngle.out
   ./TestSolidAngle.out
TestRotation: TestRotation.out
   ./TestRotation.out
TestRandomSingleton: TestRandomSingleton.out
   ./TestRandomSingleton.out
TestAtmosphere: TestAtmosphere.out
   ./TestAtmosphere.out
TestSimulation: TestSimulation.out
   ./TestSimulation.out
TestHistogram: TestHistogram.out
   ./TestHistogram.out
SimulateAtmosphere: SimulateAtmosphere.out
   ./SimulateAtmosphere.out

45
PiApproximation.out: PiApproximation.o Source.o Detector.o SolidAngle.o Photon.o Point.o RandomSingleton.o
    g++ -o PiApproximation.out PiApproximation.o Source.o Detector.o SolidAngle.o Photon.o Point.o RandomSingleton.o -lgs1 -lgs1cblas
TestDetector.out: TestDetector.o Detector.o SolidAngle.o Photon.o Point.o
    g++ -o TestDetector.out TestDetector.o Detector.o SolidAngle.o Photon.o Point.o RandomSingleton.o
    -lgs1 -lgs1cblas
TestSource.out: TestSource.o Source.o SolidAngle.o Photon.o Point.o RandomSingleton.o
    g++ -o TestSource.out TestSource.o Source.o SolidAngle.o Photon.o Point.o RandomSingleton.o
    -lgs1 -lgs1cblas
TestVisibleSource.out: TestVisibleSource.o VisibleSource.o Source.o SolidAngle.o Photon.o Point.o RandomSingleton.o
    g++ -o TestVisibleSource.out TestVisibleSource.o VisibleSource.o Source.o SolidAngle.o Photon.o Point.o RandomSingleton.o
TestPhoton.out: TestPhoton.o Photon.o Point.o
    g++ -o TestPhoton.out TestPhoton.o Photon.o Point.o
TestMedium.out: TestMedium.o Photon.o Medium.o Point.o Rotation.o RandomSingleton.o
    g++ -o TestMedium.out TestMedium.o Photon.o Medium.o Point.o Rotation.o RandomSingleton.o
    -lgs1 -lgs1cblas
TestRayleigh.out: TestRayleigh.o Photon.o Rayleigh.o Point.o Medium.o Rotation.o RandomSingleton.o
    g++ -o TestRayleigh.out TestRayleigh.o Photon.o Rayleigh.o Point.o Medium.o Rotation.o RandomSingleton.o
    -lgs1 -lgs1cblas
TestAtmosphere.out: TestAtmosphere.o Photon.o Rayleigh.o Point.o Medium.o Rotation.o RandomSingleton.o Atmosphere.o
    g++ -o TestAtmosphere.out TestAtmosphere.o Photon.o Rayleigh.o Point.o Medium.o Rotation.o RandomSingleton.o Atmosphere.o
    -lgs1 -lgs1cblas
TestPoint.out: TestPoint.o Point.o
    g++ -o TestPoint.out TestPoint.o Point.o
TestSolidAngle.out: TestSolidAngle.o SolidAngle.o
    g++ -o TestSolidAngle.out TestSolidAngle.o SolidAngle.o
TestRotation.out: TestRotation.o Rotation.o
    g++ -o TestRotation.out TestRotation.o Rotation.o
TestRandomSingleton.out: TestRandomSingleton.o RandomSingleton.o
    g++ -o TestRandomSingleton.out TestRandomSingleton.o RandomSingleton.o
    -lgs1 -lgs1cblas
46
TestSimulation.o: TestSimulation.o Simulation.o Medium.o Rayleigh.o Atmosphere.o Source.o VisibleSource.o Detector.o Photon.o Point.o SolidAngle.o Rotation.o RandomSingleton.o
TestHistogram.out: TestHistogram.o Histogram.o
g++ -o TestHistogram.out TestHistogram.o Histogram.o
PiApproximation.o: PiApproximation.cpp
g++ -c PiApproximation.cpp
Detector.o: Detector.h Detector.cpp SolidAngle.h SolidAngle.cpp Point.h Point.cpp Photon.h Photon.cpp
g++ -c Detector.cpp
TestDetector.o: Detector.h TestDetector.cpp Photon.h Photon.cpp
g++ -c TestDetector.cpp
Source.o: Source.h Source.cpp SolidAngle.h SolidAngle.cpp Point.h Point.cpp Photon.h Photon.cpp RandomSingleton.h
RandomSingleton.cpp
g++ -c Source.cpp
TestSource.o: Source.h TestSource.cpp Photon.h Photon.cpp
g++ -c TestSource.cpp
VisibleSource.o: VisibleSource.h VisibleSource.cpp Source.h Source.cpp SolidAngle.h SolidAngle.cpp Point.h Point.cpp Photon.h Photon.cpp RandomSingleton.h RandomSingleton.cpp
g++ -c VisibleSource.cpp
TestVisibleSource.o: VisibleSource.h TestVisibleSource.cpp Source.h Source.cpp Photon.h Photon.cpp
g++ -c TestVisibleSource.cpp
Photon.o: Photon.h Photon.cpp
g++ -c Photon.cpp
TestPhoton.o: Photon.h TestPhoton.cpp Point.h Point.cpp
g++ -c TestPhoton.cpp
Medium.o: Medium.h Medium.cpp Photon.h Point.h Rotation.h Rotation.cpp
g++ -c Medium.cpp
TestMedium.o: Medium.h TestMedium.cpp Photon.h Point.h  
Rotation.h Rotation.cpp
  g++ -c TestMedium.cpp
Rayleigh.o: Rayleigh.h Rayleigh.cpp Medium.h Medium.cpp
  g++ -c Rayleigh.cpp
Photon.h Point.h
  g++ -c TestRayleigh.cpp
Point.o: Point.h Point.cpp
  g++ -c Point.cpp
TestPoint.o: Point.h TestPoint.cpp
  g++ -c TestPoint.cpp
SolidAngle.o: SolidAngle.h SolidAngle.cpp
  g++ -c SolidAngle.cpp
TestSolidAngle.o: SolidAngle.h TestSolidAngle.cpp
  g++ -c TestSolidAngle.cpp
Rotation.o: Rotation.h Rotation.cpp
  g++ -c Rotation.cpp
TestRotation.o: Rotation.h TestRotation.cpp
  g++ -c TestRotation.cpp
RandomSingleton.o: RandomSingleton.h RandomSingleton.cpp
  g++ -c RandomSingleton.cpp
TestRandomSingleton.o: RandomSingleton.h
  TestRandomSingleton.cpp
  g++ -c TestRandomSingleton.cpp
Atmosphere.o: Atmosphere.h Atmosphere.cpp
  g++ -c Atmosphere.cpp
TestAtmosphere.o: Atmosphere.h TestAtmosphere.cpp
  g++ -c TestAtmosphere.cpp
Simulation.o: Simulation.h Simulation.cpp
  g++ -c Simulation.cpp
TestSimulation.o: Simulation.h TestSimulation.cpp
  g++ -c TestSimulation.cpp
Histogram.o: Histogram.h Histogram.cpp
  g++ -c Histogram.cpp
TestHistogram.o: Histogram.h TestHistogram.cpp
  g++ -c TestHistogram.cpp
ValidationSimulation.o: ValidationSimulation.h
  ValidationSimulation.cpp Simulation.h Simulation.cpp
  g++ -c ValidationSimulation.cpp
SimulateAtmosphere.o: ValidationSimulation.h Simulation.h
Histogram.h SimulateAtmosphere.cpp

    g++ -c SimulateAtmosphere.cpp

copy:
    cp *.cpp *.h makefile ../Backup/.

clean:
    rm *.o *.out
Medium.h

/* *****************************************************
 * James Nelson
 * Medium Header
 * 1-8-2015
 *****************************************************/

#ifndef MEDIUM_H
#define MEDIUM_H

#include "Photon.h"
#include "Point.h"
#include "Rotation.h"
#include "RandomSingleton.h"

/******************************************************************************
 * Medium Class
 * This class simulates a medium for photons to pass
 * through. Children classes can absorb and scatter them as
 * desired.
 ******************************************************************************/
class Medium {

protected:
  Point location;
  Rotation rotater;
  double radius;

public:
  Medium() : rotater(1, 0, 0) {radius = 1.0;};
  Medium(Point point) : location(point), rotater(1, 0, 0) {radius = 1.0;};
  Medium(double * coordinate) : location(coordinate), rotater(1, 0, 0)
    {radius = 1.0;};
  Medium(double radius) : rotater(1, 0, 0) {this->radius = radius;};
  Medium(Point point, double radius) : location(point), rotater(1, 0, 0)
    {this->radius = radius;};
  Medium(double * coordinate, double radius) : location(coordinate),
    rotater(1, 0, 0) {this->radius = radius;};
  Medium(const Medium & medium) : location(medium.getLocation()),
    rotater(1, 0, 0) {this->radius = medium.getRadius();};
  Point getLocation() const {return location;}; // get the current location
  double getRadius() const {return radius;};
  Rotation getRotater() const {return rotater;};
  virtual double getFreeDistance(const Photon &photon) const;
  virtual double getFreeTime(const Photon &photon) const;
};
static RandomSingleton & getRS() { return RandomSingleton::getInstance(); }
void setLocation(int i, double n);
void setRotator(double x, double y, double z) {
    rotater.setAxis(0, x);
    rotater.setAxis(1, y);
    rotater.setAxis(2, z);
}
void setRadius(double r) {
    if (r >= 0) radius = r;
}
bool isInMedium(const Point &point) const {
    return point.getDistance(location) <= radius;
}
virtual void scatter(Photon &photon);
#else

#endif
# Medium.cpp

/**
 * James Nelson
 * Medium Class
 * 1-8-2015
 */
#include <cstdlib>
#include "Medium.h"
using namespace std;

/**
 * Set Location
 * Set a specific coordinate of the location.
 */
void Medium::setLocation(int i, double n)
{
    if (i >= 0 && i < Point::SIZE)
        location.setCoordinate(i, n);
}

/**
 * Get Free Distance
 * Return the distance a photon will travel before an
 * interaction (does not need to be the same each time).
 */
double Medium::getFreeDistance(const Photon &photon) const
{
    return rand(); // rewrite this for children classes
}

/**
 * Get Free Time
 * Return the time a photon will travel before an
 * interaction (does not need to be the same each time).
 */
double Medium::getFreeTime(const Photon &photon) const
{
    return rand(); // rewrite this for children classes
}

/**
 * Scatter
 * This is the basic function to change the velocity, and
possibly the frequency and polarization of a given photon. Should call increment on the photon.

```cpp
void Medium::scatter(Photon &photon)
{
    if (isInMedium(photon.getLocation()))
    {
        for (int i = 0; i < Point::SIZE; i++)
        {
            photon.setVelocity(i, rand() % 3 - 1);
        }
        photon.increment();
    }
}
```
Photonic

#include <string>
#include "Point.h"
#include <gsl/gsl_const_mksa.h>
using namespace std;

/* Photon Class
 * This class simulates a photon and keeps track of
 * relevant data. All distances are in meters, all times are
 * in seconds, all angles are in degrees, and all energies
 * are in Joules.
 */
class Photon {
private:
  Point location; // where the photon is
  Point origin; // where the photon started
  Point velocity; // which way the photon is going
  double frequency; // keeps track of wavelength and energy also
                   // angle of E field oscillation to arbitrary axis,
                   // as viewed from the source
  double orientation;
  double phase; // phase between E and B maxima
  int circularPolarization; //1 for ccw from source, -1 for cw, 0 for neither
  unsigned int numInteractions; // number of interactions

public:
  Photon(); // defaults are 0 or random
  Photon(double frequency); // set the frequency
  Photon(double * origin, double * velocity); // set the origin
              // set the origin and the frequency
  Photon(double * origin, double * velocity, double frequency); // set everything
Photon (double origin, double velocity, double frequency,
    double orientation, double phase, int circularPolarization);
Photon (Point origin, Point velocity, double frequency, double orientation
    double phase, int circularPolarization); // set everything
Photon (const Photon & photon) {*this = photon;}; // copy photon
~Photon () {};
Point getLocation () const {return location;}; // get the current location
Point getOrigin () const {return origin;}; // get the original location
Point getVelocity () const {return velocity;}; // get the velocity
static double getG() {return GSLCONSTMKSA_SPEED_OF_LIGHT;};
static double getH() {return GSLCONSTMKSA_PLANCKS_CONSTANT_H;};
double getFrequency () const {return frequency;};
double getWavelength () const {return getC() / frequency;}; // l=c/f
double getEnergy () const {return getH() * frequency;}; // E=hf
double getOrientation () const;
double getPhase () const {return phase;};
int getCircularPolarization () const {return circularPolarization;};
bool isCCW () const {return circularPolarization == 1;}
bool isCW () const {return circularPolarization == -1;}
bool isLinear () const {return circularPolarization == 0;}
    // distance between coordinates, not total distance traveled
double getDistanceFromOrigin () const;
unsigned int getNumInteractions () const {return numInteractions;};
void setLocation (int i, double n);
void setOrigin (int i, double n);
void setVelocity (int i, double n);
void setFrequency (double frequency)
    {if (frequency >= 0) this->frequency = frequency;};
void setWavelength (double wavelength)
    {if (wavelength > 0) frequency = getG() / wavelength;};
void setEnergy (double energy)
    {if (energy > 0) frequency = energy / getH();};
void setOrientation (double orientation);
void setPhase (double phase);
int setCircularPolarization (int n)
    {if (n >= -1 && n <= 1) circularPolarization = n;};
bool setCCW () {circularPolarization = 1;}
bool setCW () {circularPolarization = -1;}
bool setLinear () {circularPolarization = 0;}
void setNumInteractions (unsigned int n) {numInteractions = n;};
double normalize ();
void increment () {numInteractions++;};
void advance (double amount, bool isTime = true);
string toString () const;
void print() const;
void operator=(const Photon & photon);
};
#endif
/*******************************
* James Nelson
* Photon Class
* 1–6–2015
*******************************/
#include <iostream>
#include <cmath>
#include <cstdlib>
#include <sstream>
#include "Point.h"
#include "Photon.h"
using namespace std;

/*******************************
* Photon
* Set everything to 0, velocity along the x1 axis, and the
* frequency to 1.
*******************************/
Photon::Photon()
{
  initialize();
}

/*******************************
* Photon
* Set the frequency.
*******************************/
Photon::Photon(double frequency)
{
  initialize();
  setFrequency(frequency);
}

/*******************************
* Photon
* Set the coordinates and origin to the given point. Also
* set the velocity.
*******************************/
Photon::Photon(double *origin, double *velocity)
{
  initialize();
  for (int i = 0; i < Point::SIZE; i++)
  {
     (*origin++) = (*velocity++);
  }
}
Photon::Photon(double *origin, double *velocity, double frequency)
{
    initialize();
    for (int i = 0; i < Point::SIZE; i++)
    {
        setLocation(i, origin[i]);
        setOrigin(i, origin[i]);
        setVelocity(i, velocity[i]);
    }
    setFrequency(frequency);
}

Photon::Photon(double *origin, double *velocity, double frequency, double orientation, double phase, int circularPolarization)
{
    initialize();
    for (int i = 0; i < Point::SIZE; i++)
    {
        setLocation(i, origin[i]);
        setOrigin(i, origin[i]);
        setVelocity(i, velocity[i]);
    }
    setFrequency(frequency);
    setOrientation(orientation);
    setPhase(phase);
    setCircularPolarization(circularPolarization);
}
Photon
* Set the coordinates and origin to the given point. Set
* the velocity, frequency, orientation, phase, and circular
* polarization.

```
/* Photon */
/* Set the coordinates and origin to the given point. Set */
/* the velocity, frequency, orientation, phase, and circular */
/* polarization. */

Photon::Photon(Point origin, Point velocity, double frequency,
                double orientation, double phase, int circularPolarization)
{
    initialize();
    for (int i = 0; i < Point::SIZE; i++)
    {
        setLocation(i, origin.getCoordinate(i));
        setOrigin(i, origin.getCoordinate(i));
        setVelocity(i, velocity.getCoordinate(i));
    }
    setFrequency(frequency);
    setOrientation(orientation);
    setPhase(phase);
    setCircularPolarization(circularPolarization);
}
```

```
/* Get Orientation */
/* Get the current orientation. For circular polarization */
/* the orientation variable need not change, but the phase */
/* must also be considered. */

double Photon::getOrientation() const
{
    double currentOrientation = orientation + circularPolarization * phase;
    if (currentOrientation >= 360)
    {
        currentOrientation -= 360;
    }
    else if (currentOrientation < 0)
    {
        currentOrientation += 360;
    }
    return currentOrientation;
}
```
/*********************************************************
* Get Distance From Origin
* Calculate the distance between the current location and
* the origin.
***********************************************************/
double Photon::getDistanceFromOrigin() const
{
    return location.getDistance(origin);
}

/*********************************************************
* Set Location
* Set a specific coordinate of the location.
***********************************************************/
void Photon::setLocation(int i, double n)
{
    if (i >= 0 && i < Point::SIZE)
        location.setCoordinate(i, n);
}

/*********************************************************
* Set Origin
* Set a specific coordinate of the origin.
***********************************************************/
void Photon::setOrigin(int i, double n)
{
    if (i >= 0 && i < Point::SIZE)
        origin.setCoordinate(i, n);
}

/*********************************************************
* Set Velocity
* Set a specific coordinate of the velocity.
***********************************************************/
void Photon::setVelocity(int i, double n)
{
    if (i >= 0 && i < Point::SIZE)
        velocity.setCoordinate(i, n);
}

/*********************************************************
* Set Orientation
* Set the orientation of the E field oscillation to some
* angle between 0 and 360 degrees.
***********************************************************/
void Photon::setOrientation(double orientation)
{
    while (orientation < 0)
    {
        orientation += 360;
    }
    while (orientation >= 360)
    {
        orientation -= 360;
    }
    this->orientation = orientation;
}

void Photon::setPhase(double phase)
{
    while (phase < 0)
    {
        phase += 360;
    }
    while (phase >= 360)
    {
        phase -= 360;
    }
    this->phase = phase;
}

void Photon::initialize()
{
    for (int i = 0; i < Point::SIZE; i++)
    {
        setLocation(i, 0);
        setOrigin(i, 0);
        setVelocity(i, 0);
    }
}
setVelocity(0, 1);
setFrequency(1.0);
setOrientation(0);
setPhase(0);
setCircularPolarization(0);
setNumInteractions(0);

/** Normalize
 * Return the normalization constant for the relative
 * values of the components of the velocity such that the
 * magnitude would be the speed of light.
 */
double Photon::normalize()
{
    int square = 0; // the sum of each component squared
    for (int i = 0; i < Point::SIZE; i++)
    {
        square += pow(velocity[i], 2);
    }
    if (square == 0) // this is a problem, no direction is specified
    {
        setVelocity(0, 1.0);
        square += 1.0;
    }
    return getC() / sqrt(square);
}

/** Advance
 * Change the current coordinates of the photon by advancing
 * along the direction of the velocity by the amount, which
 * is time in seconds by default (but may also be the
 * distance in meters) assuming the photon travels
 * with velocity c. Adjust the phase.
 */
void Photon::advance(double amount, bool isTime)
{
    double time;
    if (isTime) // they gave us time to travel
    {
        time = amount;
    }
}
else // they gave us distance to travel
{
    time = amount / Photon::getC();
}

double normal = normalize(); // get the constant of normalization
for (int i = 0; i < Point::SIZE; i++)
{
    setLocation(i, getLocation()[i] + getVelocity()[i] * normal * time);
}

double periods = frequency * time; // the number of periods that went by
// only the decimal part will affect the phase
double phaseChange = 360 * (periods - (long) periods);
phase += phaseChange;
if (phase > 360)
{
    phase -= 360;
}

上来一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵一阵阵
/***********************************************************/
* Print
* Output all of the values.
***********************************************************/
void Photon::print(const)
{
    cout << "(" << getLocation().getCoordinate(0);
    for (int i = 1; i < Point::SIZE; i++)
    {
        cout << "," << getLocation().getCoordinate(i);
    }
    cout << ") , [" << getOrigin().getCoordinate(0);
    for (int i = 1; i < Point::SIZE; i++)
    {
        cout << "," << getOrigin().getCoordinate(i);
    }
    cout << "] , <" << getVelocity().getCoordinate(0);
    for (int i = 1; i < Point::SIZE; i++)
    {
        cout << "," << getVelocity().getCoordinate(i);
    }
    cout << "> , " << frequency << " Hz, " << orientation << ", ", "
    << phase << ", #" << numInteractions << endl;
}

/***********************************************************/
* Equals
* Copy the given photon.
***********************************************************/
void Photon::operator=(const Photon & photon)
{
    for (int i = 0; i < Point::SIZE; i++)
    {
        setLocation(i, photon.getLocation()[i]);
        setOrigin(i, photon.getOrigin()[i]);
        setVelocity(i, photon.getVelocity()[i]);
    }
    setFrequency(photon.getFrequency());
    setOrientation(photon.getOrientation());
    setPhase(photon.getPhase());
    setNumInteractions(photon.getNumInteractions());
    setCircularPolarization(photon.getCircularPolarization());
PiApproximation.cpp

/**********************************************************************************
 * James Nelson
 * Pi Approximation
 * 1-21-2015
**********************************************************************************/
#include <iostream>
#include <cmath>
#include "Source.h"
#include "Detector.h"
#include "Photon.h"
using namespace std;

/**********************************************************************************
 * Main
 * For testing purposes. Should give an answer about equal
 * to pi.
**********************************************************************************/
int main()
{
    long n = 1000;
    double d0[3] = {1000, 0, 0};
    double s0[3] = {0, 0, 0};
    double a = atan2(1, 1000) * 180 / M_PI;
    Detector d(d0, 0, 180, 90, 270, 1.0);
    Source s(s0, (90 - a), (90 + a), 360 - a, a);
    for (int i = 0; i < 4 * n; i++)
    {
        cout << i << endl;
        Photon p = s.emit();
        if (d.detected(p, 1001, false))
        {
            d.setNumDetected(d.getNumDetected() + 1.0 / n);
        }
        cout << d.getNumDetected() << endl;
        return 0;
    }
}
Point.h

/***************************************************************************/
* James Nelson
* Point Header
* 1-21-2015
***************************************************************************/
#ifndef POINT_H
#define POINT_H

/***************************************************************************/
* Point Class
* Cartesian space point.
***************************************************************************/
class Point {
public:
  static const int SIZE = 3; // dimensions of space
private:
  double coordinate[SIZE];
public:
  Point();
  Point(double * c);
  Point(const Point & p);
  double getCoordinate(int i) const {return (i >= 0 && i < SIZE) ? coordinate[i] : 0;};  // get coordinate value
  void setCoordinate(int i, double n) {if (i >= 0 && i < SIZE)
   coordinate[i] = n;};  // set coordinate value
  double getDistance(const Point & p) const;
  double calculateAngle(const Point & p1, const Point & p2) const;
  void print() const;
  double operator[](int i) {return coordinate[i];};
  void operator=(const Point & p);
};
#endif
Point.cpp

/******************************
* James Nelson
* Point Class
* 1–21–2015
******************************/
#include <iostream>
#include <cmath>
#include "Point.h"
using namespace std;

/******************************
* Point
* Set every coordinate to 0.
******************************/
Point::Point()
{
    for (int i = 0; i < SIZE; i++)
    {
        setCoordinate(i, 0);
    }
}

/******************************
* Point
* Set every coordinate to match array c.
******************************/
Point::Point(double * c)
{
    for (int i = 0; i < SIZE; i++)
    {
        setCoordinate(i, c[i]);
    }
}

/******************************
* Point
* Copy point p.
******************************/
Point::Point(const Point & p)
{
    for (int i = 0; i < SIZE; i++)
    {

setCoordinate(i, p.getCoordinate(i));
}

/**
* Get Distance
* Get the distance between the two points.
******************************************************************************/
double Point::getDistance(const Point & p) const
{
    double d = 0;
    for (int i = 0; i < SIZE; i++)
    {
        d += pow((this->getCoordinate(i) - p.getCoordinate(i)), 2);
    }
    return sqrt(d);
}

/**
* Calculate Angle
* Get the angle in degrees made by the two given points with this point as the vertex. Use the law of cosines.
******************************************************************************/
double Point::calculateAngle(const Point & p1, const Point & p2) const
{
    double a = p1.getDistance(p2);
    double b = this->getDistance(p1);
    double c = this->getDistance(p2);
    if (a == 0 || b == 0 || c == 0)
    {
        return 0;
    }
    double ratio = (pow(b, 2) + pow(c, 2) - pow(a, 2)) / (2 * b * c);
    if (ratio > 1)
    {
        ratio = 1.0;
    }
    else if (ratio < -1)
    {
        ratio = -1.0;
    }
    return acos(ratio) * 180 / M_PI;
}
/*******************************************************************************/
/* Print */
/* Print the values of the coordinates. */
*******************************************************************************/
void Point::print() const
{
    cout << "(" << getCoordinate(0);
    for (int i = 1; i < SIZE; i++)
    {
        cout << "," << getCoordinate(i);
    }
    cout << ")";
}

/******************************************************************************/
/* Equals */
/* Copy point p. */
*******************************************************************************/
void Point::operator=(const Point & p)
{
    for (int i = 0; i < SIZE; i++)
    {
        setCoordinate(i, p.getCoordinate(i));
    }
}
RandomSingleton.h

/**********************************************************
* James Nelson
* Point Header
* 2–12–2015
***********************************************************/
#ifndef RANDOM_SINGLETON_H
#define RANDOM_SINGLETON_H
#include <gsl/gsl_rng.h>
using namespace std;

/**********************************************************
* Random Singleton Class
* Gives everyone reliable random numbers.
***********************************************************/
class RandomSingleton
{
    private:
        gsl_rng * rng;
        RandomSingleton();
    public:
        RandomSingleton();
        static RandomSingleton & getInstance() { static RandomSingleton instance;
            return instance;};
        double rand();
        double rand(double max);
        double rand(double min, double max);
        void seed(unsigned long int seed);
};
#endif
RandomSingleton.cpp

#include <sys/time.h>
#include <gsl/gsl_rng.h>
#include "RandomSingleton.h"
using namespace std;

RandomSingleton::RandomSingleton()
{
    rng = gsl_rng_alloc(gsl_rng_mt19937);
    timeval tv;
    gettimeofday(&tv, NULL);
    unsigned long int seed = (tv.tv_sec * 1000) + (tv.tv_usec / 1000);
    gsl_rng_set(rng, seed);
}

RandomSingleton::~RandomSingleton()
{
    gsl_rng_free(rng);
}

double RandomSingleton::rand()
{
    return gsl_rng_uniform(rng);
}
Rand

* Generate random number \([0, \text{max})\).

`double RandomSingleton::rand(double max)`

```cpp
return max * gsl_rng_uniform(rng);
```

Rand

* Generate random number \([\text{min}, \text{max})\).

`double RandomSingleton::rand(double min, double max)`

```cpp
return (max - min) * gsl_rng_uniform(rng) + min;
```

Seed

* Seed using the given number.

`void RandomSingleton::seed(unsigned long int seed)`

```cpp
rng = gsl_rng_alloc(gsl_rng_mt19937);
gsl_rng_set(rng, seed);
```
Rayleigh.h

 /******************************************************************************
 * James Nelson
 * Rayleigh Header
 * 2–5–2015
 ******************************************************************************/

#ifndef RAYLEIGH_H
#define RAYLEIGH_H

#include "Medium.h"

/******************************************************************************
 * Rayleigh Class
 * This class simulates a medium for photons to pass
 * through. The photons undergo Rayleigh scattering.
 ******************************************************************************/

class Rayleigh : public Medium
{

private:
    double distribute(double angle) const;

public:
    Rayleigh() : Medium() {}
    Rayleigh(Point point) : Medium(point) {}
    Rayleigh(double * coordinate) : Medium(coordinate) {}
    Rayleigh(double radius) : Medium(radius) {}
    Rayleigh(Point point, double radius) : Medium(point, radius) {}
    Rayleigh(double * coordinate, double radius) : Medium(coordinate, radius) {}
    Rayleigh(const Medium & medium) : Medium(medium) {};
    virtual void scatter(Photon &photon);
};

#endif
```cpp
#include <iostream>
#include <cmath>
#include "Rayleigh.h"
using namespace std;

double Rayleigh::distribute(double angle) const
{
  // if a random number [0,1) is less than 2*cos(angle)^2, redistribute angle
  if (getRS().rand() < 2 * pow(cos(angle), 2))
  {
    if (angle > M_PI / 2.0)
      angle += 2 * (3.0 * M_PI / 4.0 - angle);
    else
      angle -= 2 * (angle - M_PI / 4.0);
  }
  return angle;
}

void Rayleigh::scatter(Photon &photon)
{
  if (isInMedium(photon.getLocation()))
```
{
  // we need the velocity of the photon
  double vx = photon.getVelocity().getCoordinate(0);
  double vy = photon.getVelocity().getCoordinate(1);
  double vz = photon.getVelocity().getCoordinate(2);
  // angle that this event will rotate and scatter the photon
  // perpendicular to the direction of travel, measured ccw
  // from the z-axis from the point of view of the source of the photon
  double phi = getRS().rand() * 2 * M_PI; // 0–2PI
  // relative angle
  double alpha = photon.getOrientation() * M_PI / 180 - phi;
  if (alpha < 0)
  {
    alpha += 2 * M_PI;
  }
  // theta will either be isotropic or not
  bool isotropic = getRS().rand() < (-cos(2 * alpha) / 2 + .5);
  // take acos of this angle as a scalar [-1, 1] to get theta
  double beta = getRS().rand() * M_PI;
  if (!isotropic)
  {
    beta = distribute(beta);
    photon.setOrientation(getRS().rand(0, 360));
    // this line becomes important in the cases of circular polarization
    photon.setPhase(0);
  }
  double theta = acos(2 * beta / M_PI - 1); // 0–PI and not uniform
  // taking the acos is necessary even in the isotropic case, because
  // we are using spherical coordinates and we want the distribution
  // at the poles to have the same density and the distribution at
  // the equator
  double a = vx; // will be rotated to become the new vx
  double b = vy; // will be rotated to become the new vy
  double c = vz; // will be rotated to become the new vz
  double d = -vy; // component of axis in the x–y plane
  double e = vx; // component of axis in the x–y plane
  if (vx == 0 && vy == 0) // special case when v is only in the z direction
  {
    d = vz / abs(vz);
  }
  setRotater(d, e, 0); // set an axis in the x–y plane, perpendicular to v
  rotater.rotate(a, b, c, theta * 180 / M_PI); // rotate by theta degrees
  setRotater(vx, vy, vz); // set v as the axis of rotation
  rotater.rotate(a, b, c, phi * 180 / M_PI); // rotate by phi degrees
}
photon.setVelocity(0, a); // new vx
photon.setVelocity(1, b); // new vy
photon.setVelocity(2, c); // new vz
// number of scattering events for the photon should be incremented
// by one
photon.increment();
}
/*******************************************************************************************************************************************/
* James Nelson
* Simulate Atmosphere
* 3−6−2015
*******************************************************************************************************************************************/
#include <iostream>
#include <stdlib.h>
#include <cmath>
#include <sys/time.h>
#include "ValidationSimulation.h"
#include "Atmosphere.h"
#include "VisibleSource.h"
#include "Detector.h"
using namespace std;

/*******************************************************************************************************************************************/
* Main
* For testing purposes.
*******************************************************************************************************************************************/
int main(int argc, char *argv[])
{
  double r = 100000;
  double d = r * sqrt(3);
  double ps[3] = {−d, 0, 0};
  double pa[3] = {0, 0, 0};
  double pd[3] = {d, 0, 0};
  VisibleSource vs(ps, 60.0, 120.0, 330.0, 30.0);
  Atmosphere atm(pa, r);
  Detector detector(pd, 60.0, 120.0, 150.0, 210.0, r);
  int num = 1000;
  string hostname(getenv("HOSTNAME"));
  if (argc > 1)
  {
    num = atoi(argv[1]);
  }
  ofstream fout, f, p;
  if (argc > 2)
  {
    string name(argv[2]);
    fout.open((name + "−" + hostname + ".out").c_str());
    f.open((name + "−" + hostname + ".f.out").c_str());
    p.open((name + "−" + hostname + ".p.out").c_str());
  }
else {
    fout.open((hostname + " .out ").c_str());
    f.open((hostname + "-f.out ").c_str());
    p.open((hostname + "-p.out ").c_str());
}
ValidationSimulation sim(vs, detector, atm, num);
timeval tv;
gettimeofday(&tv, NULL);
unsigned long int seed = (tv.tv_sec * 1000) + (tv.tv_usec / 1000);
int offset = 0;
for (int i = 0; i < hostname.size(); i++)
{
    offset += pow(10, i) * hostname[i];
}
sim.getRS().seed(seed + offset);
sim.run(fout);
f << sim.getFrequencies().toString();
p << sim.getPolarizations().toString();
fout.close();
f.close();
p.close();
return 0;
}
Simulation.h

/****************************************************************************
 * James Nelson
 * Simulation Header
 * 1-8-2015
/****************************************************************************
#define SIMULATION_H

#include <fstream>
#include "Source.h"
#include "Detector.h"
#include "Photon.h"
#include "Medium.h"
#include "RandomSingleton.h"

/****************************************************************************
 * Simulation Class
 * This class runs a Monte Carlo simulation. The source
 * be pointed at or in the medium.
/****************************************************************************
class Simulation
{
 protected:
  Source * source; //Source of photons
  Detector * detector; //Detector of photons
  Medium * medium;
  unsigned long int numIterations;
  string header;
  virtual string format(const Photon & photon);
 public:
  Simulation(Source & source, Detector & detector, Medium & medium,
             unsigned long int numIterations, string header = "")
  {this->source = &source; this->detector = &detector;
   this->medium = &medium; this->numIterations = numIterations;
    this->header = header;};
  Source & getSource() {return *source;};
  Detector & getDetector() {return *detector;};
  Medium & getMedium() {return *medium;};
  unsigned long int getNumIterations() {return numIterations;};
  static RandomSingleton & getRS()
  {return RandomSingleton::getInstance();};
  void setNumIterations(unsigned long int numIterations)
{this->numIterations = numIterations;}
void run(ofstream & fout);
}
#endif
/********************************************
* James Nelson
* Simulation Class
* 2-27-2015
********************************************/
#include <iostream>
#include <cmath>
#include <sstream>
#include "Simulation.h"
using namespace std;

/********************************************
* Format
* Format the data for gnuplot. In this scope you have
* access to the photon before it was detected, the
* detector, medium, and source.
*********************************************/
string Simulation::format(const Photon & photon)
{
    return photon.toString();
}

/********************************************
* Run
* Run the simulation.
*********************************************/
void Simulation::run(ofstream & fout)
{
    Photon photon;
    double distance;
    double freeDistance;
    long double percent = numIterations / 99.0;
    int x = -1;
    fout << header << endl;
    for (int i = 0; i < numIterations; i++)
    {
        photon = getSource().emit(); // new photon
        distance = photon.getLocation().getDistance(
            getMedium().getLocation()) + .01;
        while (!getMedium().isInMedium(photon.getLocation()) &&
            photon.getLocation().getDistance(
                getMedium().getLocation()) < distance)
            .
{
    // get to the edge of the medium
    distance = photon.getLocation().getDistance(
        getMedium().getLocation());
    photon.advance(
        distance - getMedium().getRadius() * .9999, false);
}
while (getMedium().isInMedium(photon.getLocation()))
{
    // while we are in the medium
    // get free distance
    freeDistance = getMedium().getFreeDistance(photon);
    // photon detected
    if (getDetector().detected(photon, freeDistance, false))
    {
        fout << format(photon) << endl; // format and output the result
        // end the while loop
        photon.advance(getMedium().getRadius() * 2, false);
    }
    else // photon not detected
    {
        photon.advance(freeDistance, false); // advance photon
        // still in medium
        if (getMedium().isInMedium(photon.getLocation()))
        {
            getMedium().scatter(photon); // scatter photon
        }
        else if (getDetector().detected(photon, photon.getLocation() .getDistance(getDetector().getLocation()), false))
        {
            // check for the case that the detector
            // is outside the medium but that the
            // photon is headed straight towards it
            // format and output the result
            fout << format(photon) << endl;
        }
    }
}
if (((double) i / percent > x)) // output status
{
    while (((double) i / percent > x))
    {
        x++;
    }
}
cout << x << "% complete\n";

if (x != 100)
{
  cout << "100% complete\n";
}
SolidAngle.h

/**
 * James Nelson
 * Solid Angle Header
 * 1–21–2015
 */

#ifndef SOLID_ANGLE_H
#define SOLID_ANGLE_H

/**
 * Solid Angle Class
 * An angle with an upper and lower range. Everything is in
 * degrees. If isFullAngle is set to false then that means
 * that the ranges are all bound [0,180). If min is greater
 * numerically than max then the valid range wraps around
 * from min to max.
 */

class SolidAngle {

private:
    static const double FULL = 360.0;
    static const double HALF = 180.0;
    bool isFullAngle;
    double min;
    double max;

public:
    SolidAngle() {
        isFullAngle = true; setMin(0); setMax(.999999 * FULL);}
    SolidAngle(bool isFullAngle) {
        this->isFullAngle = isFullAngle; setMin(0);
        setMax((isFullAngle) ? .999999 * FULL : .999999 * HALF);}
    SolidAngle(double min, double max) {
        isFullAngle = true; setMin(min); setMax(max);}
    SolidAngle(double min, double max, bool isFullAngle) {
        this->isFullAngle = isFullAngle; setMin(min); setMax(max);}
    SolidAngle(const SolidAngle & solidAngle) {
        isFullAngle = solidAngle.isFullAngle;
        setMin(solidAngle.getMin()); setMax(solidAngle.getMax());}
    double getMin() const {return min;}
    double getMax() const {return max;}
    double getRange() const;
    void setMin(double n);
void setMax(double n);
bool isInRange(double n) const;
void print() const;
} #endif
SolidAngle.cpp

/******************************************************************************************
* James Nelson
* Solid Angle Class
* 1-21-2015
*******************************************************************************************/
#include <iostream>
#include "SolidAngle.h"
using namespace std;

 /******************************************************************************************
* Reduce
* Put the angle in the range [0-360).
*******************************************************************************************/
double SolidAngle::reduce(double n) const
{
    while (n < 0)
    {
        n += FULL;
    }
    while (n >= FULL)
    {
        n -= FULL;
    }
    return n;
}

 /******************************************************************************************
* Get Range
* Get the magnitude of the valid range.
*******************************************************************************************/
double SolidAngle::getRange() const
{
    if (min <= max)
    {
        return max - min;
    }
    else
    {
        return (isFullAngle) ? (FULL - min + max) : (HALF - min + max);
    }
}
/******************************************************************************************
* Set Min
* Set minimum angle (does not have to be less than max angle).
*******************************************************************************************/
void SolidAngle::setMin(double n)
{
    n = reduce(n);
    if (isFullAngle || n < HALF)
    {
        min = n;
    }
    else
    {
        min = FULL - n;
    }
}

/******************************************************************************************
* Set Max
* Set maximum angle (does not have to be more than min angle).
*******************************************************************************************/
void SolidAngle::setMax(double n)
{
    n = reduce(n);
    if (isFullAngle || n < HALF)
    {
        max = n;
    }
    else
    {
        max = FULL - n;
    }
}

/******************************************************************************************
* Is In Range
* Return true is the given angle is within the solid angle.
*******************************************************************************************/
bool SolidAngle::isInRange(double n) const
{

n = reduce(n);
if (min <= max)
{
    return n >= min && n <= max;
}
el if (!isFullAngle && n >= HALF)
{
    n = FULL - n;
}
return !(n > min && n < max);

/***************************************************************************/
/* Print */
/* Print the values of the coordinates. */
/***************************************************************************/
void SolidAngle::print() const
{
    cout << "[" << min << "," << max << "]";
}
Source.h

/********************************************************
* James Nelson
* Source Header
* 1-21-2015
********************************************************/
#ifndef SOURCE_H
#define SOURCE_H

#include "Photon.h"
#include "Point.h"
#include "SolidAngle.h"
#include "RandomSingleton.h"

/********************************************************
* Source Class
* This class simulates a source of light.
********************************************************/
class Source
{
  private:
    Point location;
    SolidAngle theta; //0 to pi
    SolidAngle phi; //0 to 2 pi
  public:
    Source() : location(), theta(false), phi(true) {}
    Source(double * coordinate, double thetaMin, double thetaMax,
           double phiMin, double phiMax);
    Source(const Source & source) : location(source.getLocation()),
    theta(source.getTheta()), phi(source.getPhi()) {}
    Point getLocation() const {return location;} // get the location
    SolidAngle getTheta() const {return theta;} // get the vertical angle
    SolidAngle getPhi() const {return phi;} // get the horizontal angle
    Point getVelocity();
  static RandomSingleton & getRS()
    {return RandomSingleton::getInstance();}
    void setLocation(int min, double max) {theta.setMin(min);
    theta.setMax(max);};
    void setTheta(double min, double max) {theta.setMin(min);
    theta.setMax(max);};
    void setPhi(double min, double max) {phi.setMin(min); phi.setMax(max);};
    virtual Photon emit();
};
#endif
Source.cpp

#include <iostream>
#include <cmath>
#include "Source.h"
using namespace std;

Source::Source(double *coordinate, double thetaMin, double thetaMax, double phiMin, double phiMax) : theta(thetaMin, thetaMax, false), phi(phiMin, phiMax, true)
{
    for (int i = 0; i < Point::SIZE; i++)
    {
        location.setCoordinate(i, coordinate[i]);
    }
}

Point Source::getVelocity()
{
    Point velocity;
    double t, p;
    if (theta.getMin() <= theta.getMax())
    {
        t = theta.getMin() + theta.getRange() * getRS().rand();
    }
    else
    {
        t = theta.getMin() - theta.getRange() * getRS().rand();
    }
    p = phi.getMin() + phi.getRange() * getRS().rand();
}
velocity.setCoordinate(0, cos(p * M_PI / 180) * sin(t * M_PI / 180));
velocity.setCoordinate(1, sin(p * M_PI / 180) * sin(t * M_PI / 180));
velocity.setCoordinate(2, cos(t * M_PI / 180));
return velocity;

/**
 * Emit
 * Generate a photon with all of the right characteristics.
 */
Photon Source::emit()
{
    Photon photon(getLocation(), getVelocity(), 100, getRS().rand(360),
                   getRS().rand(360), getRS().rand(360));
    return photon;
}
ValidationSimulation.h

/***************************************************************************/
* James Nelson
* Validation Simulation Header
* 3-18-2015
***************************************************************************/
#ifndef VALIDATION_SIMULATION_H
#define VALIDATION_SIMULATION_H

#include "Simulation.h"
#include "Histogram.h"

/***************************************************************************/
* Validation Simulation Class
* This class runs a Monte Carlo simulation. It validates
* the model for simple rayleigh scattering in
* the atmosphere.
***************************************************************************/
class ValidationSimulation : public Simulation
{
    protected:
        Histogram frequencies;
        Histogram polarizations;
        virtual string format(const Photon & photon);
    public:
        ValidationSimulation(Source & source, Detector & detector,
                            Medium & medium, unsigned long int numIterations):
            Simulation(source, detector, medium, numIterations),
            frequencies(1000, 3.89e11, 400.0e12), polarizations(360, 1, 0)
            {header = "Source-angle Frequency" +
             " Relative-Polarization Z-approach Z-linear-polarization";};
        Histogram getFrequencies() const {return frequencies;};
        Histogram getPolarizations() const {return polarizations;};
};
#endif
ValidationSimulation.cpp

/**
 * James Nelson
 * Validation Simulation Class
 * 3–18–2015
 */
#include <iostream>
#include <cmath>
#include <sstream>
#include "ValidationSimulation.h"
using namespace std;

/**
 * Format
 * Format the data for gnuplot. Should output "Source-angle
 * Frequency Relative-Polarization Z-approach
 * Z-linear-polarization"
 */
string ValidationSimulation::format(const Photon & photon) 
{
    stringstream data;
    Point p = photon.getLocation(); // photon
    Point c = getDetector().getLastContact(); // contact
    Point s = getSource().getLocation(); // source
    double angleFromSource = c.calculateAngle(p, s); // angle pcs
    double angleFromZ;
    if (angleFromSource < 0.1) // case of all unscattered photons
    {
        angleFromZ = 0;
    }
    else
    {
        // from s to c
        double sc[3] = {c.getCoordinate(0) - s.getCoordinate(0),
                        c.getCoordinate(1) - s.getCoordinate(1),
                        c.getCoordinate(2) - s.getCoordinate(2)};
        Rotation r = getMedium().getRotater();
        // set axis of rotation to z-axis
        r.setAxis(0, 0);
        r.setAxis(1, 0);
        r.setAxis(2, 1);
        // from s to p
        double sp[3] = {p.getCoordinate(0) - s.getCoordinate(0),
                        p.getCoordinate(1) - s.getCoordinate(1),
                        p.getCoordinate(2) - s.getCoordinate(2)};
        data << "Source-angle Frequency Relative-Polarization Z-approach Z-linear-polarization";
    }
}
\( p . \text{getCoordinate}(1) - s . \text{getCoordinate}(1), \)
\( p . \text{getCoordinate}(2) - s . \text{getCoordinate}(2) \};

// for calculating the angles to rotate sp such that sc would be
// on the x-axis
double d = \sqrt(\text{pow}(sc[0], 2) + \text{pow}(sc[1], 2));

// z-axis rotation
r . rotate(sp[0], sp[1], sp[2], -atan2(sc[1], sc[0]) * 180 / M_PI);
// set axis of rotation to y-axis
r . setAxis(0, 0);
r . setAxis(1, 1);
r . setAxis(2, 0);

// y-axis rotation
r . rotate(sp[0], sp[1], sp[2], atan2(sc[2], d) * 180 / M_PI);
angleFromZ = (atan2(-sp[1], sp[2]) * 180 / M_PI + 180);

// orientation - angle of incidence from z-axis
double relativePolarization = photon . getOrientation() - angleFromZ;
if (relativePolarization < 0)
{
  relativePolarization += 360;
}
if (angleFromSource >= 0.1)
{
  frequencies <<= photon . getFrequency();
}
if (angleFromSource >= 45 && angleFromSource < 46)
{
  polarizations <<= relativePolarization;
}
data <<= angleFromSource <<= " " <<= photon . getFrequency() <<= " "
 <<= relativePolarization <<= " " <<= angleFromZ <<= " "
 <<= photon . getOrientation();
return data . str();
}
VisibleSource.h

/**************************************************************************
 * James Nelson
 * Visible Source Header
 * 3-2-2015
 **************************************************************************/
#ifndef VISIBLE_SOURCE_H
#define VISIBLE_SOURCE_H

#include "Source.h"

/**************************************************************************
 * Visible Source Class
 * This class simulates a source of visible light. The
 * emitted photons are incoherent, have random (linear)
 * polarization, and a evenly distributed random frequency
 * [400 THz, 789 THz).
 **************************************************************************/
class VisibleSource : public Source
{
    public:
        VisibleSource() {}
        VisibleSource(double * coordinate, double thetaMin, double thetaMax,
                       double phiMin, double phiMax) : Source(coordinate, thetaMin,
                       thetaMax, phiMin, phiMax) {}
        VisibleSource(const Source & source) : Source(source) {}
        virtual Photon emit();
};
#endif
VisibleSource.cpp

/*!*******************************************************************/
* James Nelson
* Visible Source Class
* 3–2–2015
/*!*******************************************************************/
#include "VisibleSource.h"
using namespace std;

/*!*******************************************************************/
* Emit
* Generate a photon with all of the right characteristics.
/*!*******************************************************************/
Photon VisibleSource::emit()
{
    Photon photon(getLocation(), getVelocity(), getRS()
        .rand(400.0e12, 789.0e12), getRS().rand(360),
        getRS().rand(360), 0);
    return photon;
}