A Simulation of Neutrino Events in Detectors with Ray Tracing

Santanu Antu

Fermi National Accelerator Laboratory, Batavia, IL 60510

August 12, 2022

Abstract

Neutrinos are the least understood particles of the standard model, and as a result, they have been a central piece of study in particle physics in recent years. Due to its miniscule probability of interacting with material, it is cumbersome to perform any experiment with neutrinos. To resolve this issue, experiments require large complex detector systems. While the experiment has many facets of its own, a computational tool is necessary to simulate the neutrino events and the positions of interactions in a detector. In this paper, we develop a tool using ray tracing to determine the interaction points within the simulated detector. Thus far, only simple shapes, such as a sphere, cube, or cylinder are available.

1 Introduction

The DUNE (Deep Underground Neutrino Experiment) project is one of the largest endeavors so far to understand the nature of neutrinos [1][2]. It will potentially provide us with new hints towards understanding the properties of matter and antimatter, and the evolution of the universe. DUNE will consist of two neutrino detectors, which will detect the most intense neutrino beams produced so far. The first detector, at Fermilab, will record the neutrino interactions near the source of the beam. The second detector at the Sanford Underground Research Laboratory in Lead, South Dakota, will be used to detect the neutrinos 1300 kilometers downstream. One of the key questions this experiment will try to answer is if the matter and antimatter neutrinos oscillate identically [1]. On a basic level, the probability of a neutrino produced in a flavor eigenstate $|\nu_\alpha\rangle$ to oscillate to flavor eigenstate $|\nu_\beta\rangle$ after travelling a distance L is given by [3]:

$$\begin{aligned} P_{\alpha\to\beta} = \delta_{\alpha\beta} - 4\sum_{i<j}^{3} \Re(U_{\alpha i}^* U_{\beta i} U_{\alpha j} U_{\beta j}^*) \sin^2\left(\frac{\Delta m_{ij}^2 L}{4E}\right) \\ +2\sum_{i<j}^{3} \Im(U_{\alpha i}^* U_{\beta i} U_{\alpha j} U_{\beta j}^*) \sin^2\left(\frac{\Delta m_{ij}^2 L}{2E}\right) \end{aligned} \quad (1)$$

Here, U is the PMNS matrix, Δm_{ij} is the difference of masses of the i-th and j-th mass eigenstates and E is the energy of the neutrino. If we get back from the natural units and restore the $\hbar$ and c in the equation, then we get $\frac{\Delta_{jk}(mc^2)^2 L}{4\hbar c E} \approx 1.27 \times \frac{\Delta_{jk} m^2}{\mathrm{eV}^2} \frac{L}{\mathrm{km}} \frac{\mathrm{GeV}}{E}$. The mass differences, $\Delta_{jk} m^2$, are known to be on the order of 10^{-4} eV^2 [3], while the energy E is on the order of MeV or GeV [3]. Hence, a really large distance is needed to have an appreciable

probability of oscillation to a different flavor eigenstate. We need a numerical simulation to see if the experimental outcomes are consistent with the theory. The goal of our project was to build a program that would predict the interaction points of the neutrinos in a detector with various components given the effective cross section of the neutrinos, neutrino beam and detector geometry. The detectors used in these experiments (MicroBooNE, T2K, NOvA, HyperK, and DUNE) can have different shapes [4] [5] [6] [7]. However, in general, they can be approximated with simple shapes such as cuboids, spheres and cylinders. The idea is to develop a simple interface that can take these different geometrical shapes, simulate a detector geometry, provide the probability of interaction of a beam in different sections of the detector, and also produce the interaction points inside the detector. In this tool, we employed ideas from ray tracing, which is well studied in the context of computer graphics [8], and is essential to simulate the interaction points.

2 Basic Ray Tracing

We will discuss the key components of ray tracing relevant to this work in this section. These techniques are well studied in the context of computer graphics [8]. Ray tracing is essentially a rendering algorithm that models the transport of light in an efficient way and produces realistic images [9]. The basic idea behind this algorithm is very simple. In real life, light rays are produced in a source, and spread around in the surrounding until they hit different surfaces and get reflected. These reflected rays goes in all different directions and only a small fraction of the reflected rays eventually reach our camera (or eye/detector). This is why computing the reflections of all the different rays emitted from the source is both computationally expensive and irrelevant. Instead, if we compute only the rays that reach our camera, we save a lot of computational expense, and still produce realistic images.

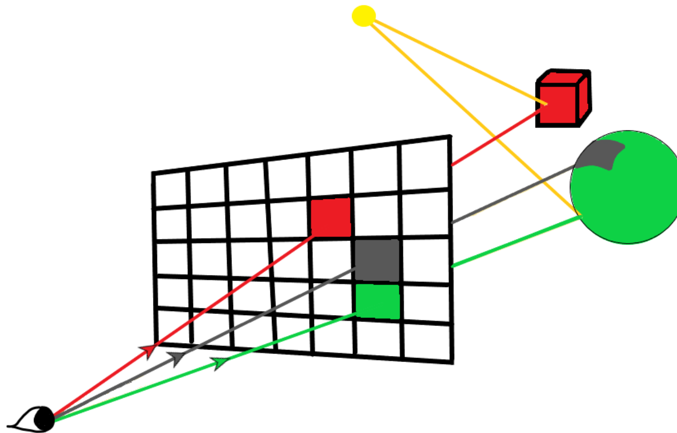


Figure 1: The basic idea of ray tracing. We send imaginary rays from the eye/camera through each pixels, and assign colors to the pixel accordingly.

In terms of programming this, we give ourselves the 3D coordinates of our camera, the position of the screen and the geometrical description of the 3D objects in our scene. For instance, in the case of a sphere, we give ourselves the 3D coordinates of the center of the sphere and its radius. For a cube, we give ourselves the 6 planes whose intersections define the cube. In the case of a cylinder, we need to specify its height, radius and orientation. For the purpose of our project, an efficient code that can handle these three shapes is more than enough.

2.1 Ray tracing with spheres

A sphere is the simplest object in our ray tracing algorithm due to its simplicity in Euclidean space. Assume that the 3D coordinate of the center of our sphere is (m, n, p), and its radius is R. If our camera is at the position (a, b, c) and the screen is the xy-plane, then we can calculate the direction of the rays by subtracting the 3D coordinates of the camera from the 3D coordinates of each of the pixels. Assume that an arbitrary ray from the camera is in the direction $d\hat{x} + e\hat{y} + f\hat{z}$. Then we can parameterize the ray as

$$\vec{r} = (a + dt)\hat{x} + (b + et)\hat{y} + (c + ft)\hat{z} \tag{2}$$

Here $t \in \mathbb{R}$, which we will naturally interpret as a "time" parameter. It is simple to find the times of intersections of this ray with the sphere (which is given by $(x-m)^2 + (y-n)^2 + (z-p)^2 = R^2$) just by replacing the x, y, z components of (2) in the equation of the sphere. This substitution will produce a quadratic equation, whose solutions are given by:

$$t_{1,2} = \frac{1}{2A}(-B \pm \sqrt{\Delta}) \tag{3}$$

Where

$$\begin{aligned}
A &= d^2 + e^2 + f^2 \\
B &= 2ad + 2be + 2cf - 2dm - 2en - 2fp \\
\Delta &= (2ad + 2be + 2cf - 2dm - 2en - 2fp)^2 \\
&\quad - 4\left(d^2 + e^2 + f^2\right)\left(a^2 - 2am + b^2 - 2bn + c^2 - 2cp + m^2 + n^2 + p^2 - R^2\right)
\end{aligned} \tag{4}$$

If $\Delta < 0$, we have no real intersection at all. If $\Delta = 0$, the ray is tangential to the sphere. In either case, we assign black color to the pixel associated with the ray. On the other hand, if $\Delta > 0$, then we get two intersection points. Let the time of intersections be t_1 and t_2. For light rays, we only care about $t_{min} = \min(t_1, t_2)$, as light rays don't penetrate through the sphere. We plug in this minimum time in (2) to calculate the 3D coordinate of the point of intersection.

To find the reflected ray, we use simple geometry and vector algebra. The normal vector to the surface can be found easily by taking the gradient of the equation of the sphere and evaluating at the intersection point. The direction of the reflected ray is then simply given by:

$$\vec{R}_r = \vec{R}_i - 2(\vec{R}_i \cdot \hat{n})\hat{n} \tag{5}$$

There are other graphical details that can be easily accommodated in the program, such as shading, coloring, shininess etc of an object. These can be found in [10]. However, these are not necessary for the purpose of the project. Now we move on to the ray tracing algorithm for a cuboid.

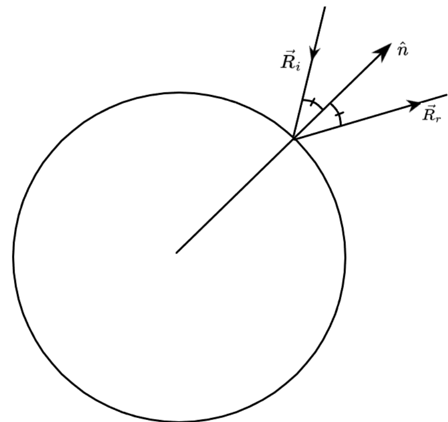


Figure 2: Reflection on a spherical surface

2.2 Ray tracing with rectangular solids

Here we discuss the ray tracing algorithm for a rectangular solid with sides parallel to the coordinate planes. For rectangles with a different orientation, we can transform our ray in the opposite way to compensate for the different orientation. We have this functionality in our final code. So, we only need to consider the basic idea of ray tracing with rectangular solids parallel to the coordinate axes. Suppose that the 6 surfaces of the rectangular solid are given by $x = x_{min}, x = x_{max}, y = y_{min}, y = y_{max}, z = z_{min}, z = z_{max}$.

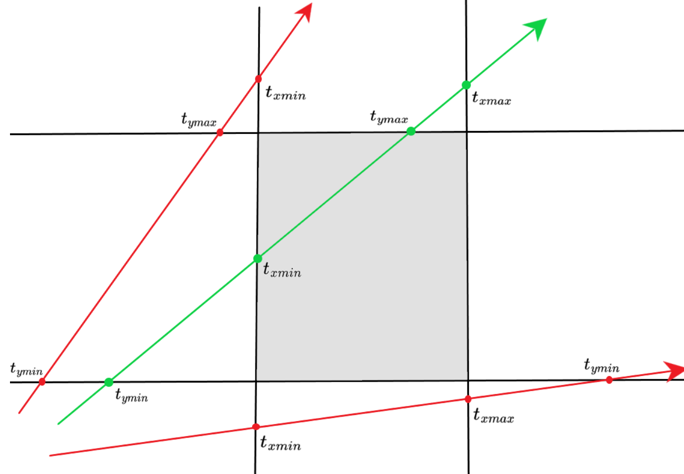


Figure 3: Ray tracing for a 2D rectangle. We don't get an intersections with the a ray if $t_{ymax} < t_{xmin}$ or $t_{xmax} < t_{ymin}$.

If we have a ray given in the parametric form as in (2), we can easily solve the minimum and maximum intersection times with each of the surfaces. The intuitive idea for a 2D rectangle is presented in Figure 3. For the 3D version of it, we present the algorithm to determine if the ray intersects the rectangular solid, and if it does, we calculate the minimum intersection point. We define a function that takes in the origin and direction of the ray (ray_origin and ray_direction) and the minimum and maximum coordinates that define the planes of the rectangular solid. If the ray intersects the rectangular solid, then it stores the minimum intersection time in the variable tmin. Otherwise, it stores infinity as the intersection point. We follow some python and numpy notations in the presented algorithm.

```
def cuboid_intersect(ray_origin, ray_direction, xmin, xmax, ymin, ymax, zmin, zmax):
    t_min=(xmin-ray_origin[0])/ray_direction[0]
    tmax=(xmax-ray_origin[0])/ray_direction[0]
    if tmin>tmax:
        tmin, tmax = tmax, tmin

    tymin=(ymin-ray_origin[1])/ray_direction[1]
    tymax=(ymax-ray_origin[1])/ray_direction[1]
    if tymin>tymax:
        tymin, tymax = tymax, tymin
    if tmin>tymax or tymin>tmax:
        return None
    if tymin>tmin:
        tmin=tymin
    if tymax<tmax:
        tmax=tymax

    tzmin=(zmin-ray_origin[2])/ray_direction[2]
```

```
    tzmax=(zmax-ray_origin[2])/ray_direction[2]
    if tzmin>tzmax:
        tzmin, tzmax = tzmax, tzmin
    if tmin>tzmax or tzmin>tmax:
        return None
    if tzmin>tmin:
        tmin=tzmin
    if tzmax<tmax:
        tmax=tzmax

    if tmin < 0:
        tmin = tmax
    if tmin < 0:
        tmin = numpy.Infinity
    return tmin
```

Then we compute the reflected rays and other graphical elements in a similar way as in the case of the sphere, but these are not strictly necessary for this project. Someone enthusiastic about these details can learn more about this in [8]. A basic image with some of the techniques presented in [8] is presented in Figure 4

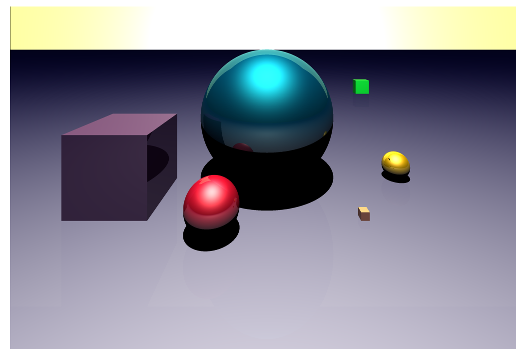


Figure 4: An image produced using basic ray tracing and Blinn-Phong illumination method.

2.3 Modification for Neutrino

In the case of neutrino ray tracing, we need to add a few modifications. First of all, we don't need to consider any kind of reflections from the detector surfaces. Neutrinos only interact with the weak force, and so the probability of interaction with the detector surface is zero for all practical purpose [3]. So, instead of stopping our computation at the minimum interaction point and computing the reflection, we propagate the ray through the bulk of the detector, and find the intersection points with the surface of the different detector components. There might be different materials inside each of the detector components, each having different probabilities of interaction with the neutrino ray. In our program, we consider these different probabilities of interaction in different materials and generate the interaction points in the detector for a given neutrino beam.

3 Overview of the Code

3.1 Setting up the Detector

The first ingredient that we need to generate the interaction points in the detector is the geometry of the detector. As mentioned before, we only need to consider simple shapes such as spheres, boxes, and in some situations cylinders. The user of the program inputs the defining geometric characters of these shapes. For instance, the radius and coordinate of the center for a sphere, the six defining planes for a box, or the radius, height and axis for a cylinder. A large box/sphere/cylinder that encloses all the other shapes is what we call the *world*. All the interactions that we are concerned with happens inside the world. The user can define different

detector components with different geometries and filled with different materials inside the world. In our program, we also have to import the cross sections of neutrinos with the different materials composing the detector. This cross section is analogous to the effective cross sectional area of a spherical atom that's often introduced in calculations related to statistical mechanics. Depending on the cross sections of neutrinos with different materials, we can calculate the mean free paths in each of the materials.

3.2 Determining the Interaction Points

If we send in a ray of a large number of neutrinos inside a material with mean free path λ, the surviving number of neutrinos falls off proportional to $\exp(-x/\lambda)$, where x is the distance travelled by the ray. Hence, the probability for a neutrino interacting with the detector after it has travelled a distance x inside the material can be modelled by:

$$P(x) \propto 1 - e^{-x/\lambda} \tag{6}$$

Given the geometry of the different detector components, we normalize this probability distribution to make sure the probability of interaction inside the world is 1. Now as we shoot a neutrino beam inside the detector, a neutrino ray can intersect the detector components at several points based on the origin and direction of the ray. We choose a random number (α) between 0 and 1, and a random direction within some range of spread of the neutrino beam. For an arbitrary neutrino, which intersects the surfaces of the detector components at n different points, we have $(n+1)$ different segments of the line inside the detector components. We use (6) to compute the probabilities ($P(s_i)$) of interaction in each of the segments (s_i). Now, we calculate the first segment (say s_r) such that $\sum_{i=1}^{r} P(i) > \alpha$. This is the segment where the interaction happens. Let $\vec{R_{r0}}$ and $\vec{R_{r1}}$ be the first and last point of intersection of this segment, and l_r be the length of this segment. Then we can obtain the approximate point of interaction using linear interpolation. Let $\beta = \sum_{i=1}^{r} P(i) - \alpha$. Then, the interaction point is simply given by:

$$\vec{R} = \vec{R_{r0}} + (\vec{R_{r1}} - \vec{R_{r0}}) \left(\frac{\alpha + P(s_r) - \sum_{i=1}^{r-1} P(s_i)}{P(s_r)} \right) \tag{7}$$

The user can select the number of neutrinos in the beam, and the program will use this method to compute that many interaction points.

4 Results

We applied the methods discussed above and set up a sample detector with 3 boxes with some rotations applied to them as shown in figure 6. Each box is filled with materials of different mean free paths. We then shot a beam of 200 neutrinos which spreads out as a 45° cone around the z-axis from the point (0,0,-2), and plotted the XY, YZ and ZX projections of the interaction points. We have also tested the code by sending 10,000 neutrinos in the $+z$ direction from the point $(0, 0, -2)$. As expected, the neutrino ray intersects the boxes at 4 points, and has 5 different line segments. Comparing the probability calculation using (6), and frequency of interaction points of 10,000 rays, we are getting a precise agreement as shown in figure 5.

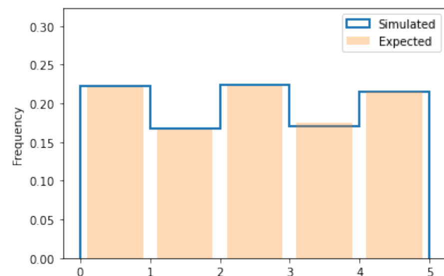
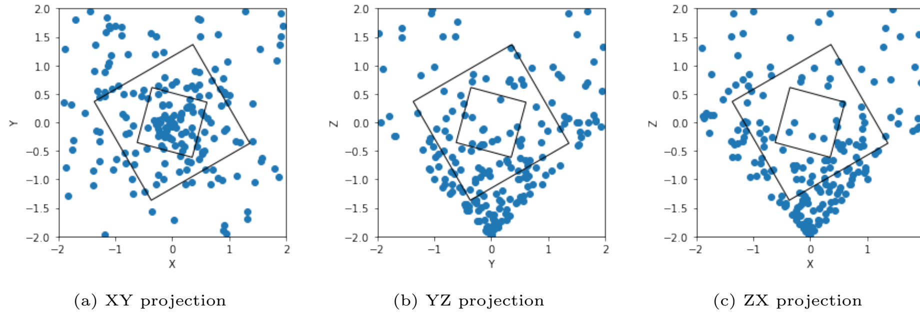


Figure 5: The histogram of probability of interaction in different segments of the ray (orange) and the normalized frequency of the interaction using the simulation (blue).



(a) XY projection (b) YZ projection (c) ZX projection

Figure 6: The three different coordinate plane projections of neutrino interaction points.

5 Conclusions and Future Steps

With our code, we could successfully generate the interaction points of the neutrinos in the detector. However, we still need a user-friendly interface which is more accessible to the people not familiar with the technical details of the code. Therefore, the next step extending project is to create an interface where a user can input the details of the rays and the geometry of the detector manually or import them from a file.

References

[1] Abi, B., Acciarri, R., Acero, M. A., Adamov, G., Adams, D., Adinolfi, M., Ahmad, Z., Ahmed, J., Alion, T., Monsalve, S. A., Alt, C., Anderson, J., Andreopoulos, C., Andrews, M., Andrianala, F., Andringa, S., Ankowski, A., Antonova, M., Antusch, S., . . . Zwaska, R. (2020). Volume I. Introduction to DUNE. Journal of Instrumentation, 15(8), T08008–T08008. https://doi.org/10.1088/1748-0221/15/08/t08008.

[2] A. Abed Abud *et al.* [DUNE], Instruments **5**, no.4, 31 (2021) doi:10.3390/instruments5040031 [arXiv:2103.13910 [physics.ins-det]].

[3] V. Barger, D. Marfatia, and K.L. Whisnant, The Physics of Neutrinos (Princeton University Press, Princeton, 2012), pp. 11-43.

[4] B. Fleming [MicroBooNE], doi:10.2172/1333130

[5] K. Abe *et al.* [T2K], [arXiv:1901.03750 [physics.ins-det]].

[6] D. S. Ayres *et al.* [NOvA], doi:10.2172/935497

[7] K. Abe *et al.* [Hyper-Kamiokande], [arXiv:1805.04163 [physics.ins-det]].

[8] P. Shirley, Ray Tracing in One Weekend (2020).

[9] K. Suffern, Ray Tracing from the Ground Up (Chapman and Hall/CRC, Natick, 2016), pp. xiii-xviii.

[10] B.T. Phong, Communications of the ACM 18, 311 (1975).