Article

Enhancing Visible Light Communication Channel Estimation in Complex 3D Environments: An Open-Source Ray Tracing Simulation Framework

Véronique Georlette [1,*,†], Nicolas Vallois [2], Véronique Moeyaert [1] and Bruno Quoitin [2]

1 Electromagnetism and Telecommunication Department, University of Mons, 7000 Mons, Belgium; veronique.moeyaert@umons.ac.be
2 Department of Computer Science, Faculty of Science, University of Mons, Av. du Champ de Mars, 7000 Mons, Belgium; nicolas.vallois@student.umons.ac.be (N.V.); bruno.quoitin@umons.ac.be (B.Q.)
* Correspondence: verogeorlette@gmail.com
† Current address: Department of Electrical and Computer Engineering, University of Illinois, Chicago, IL 60607, USA.

Abstract: Estimating the optical power distribution in a room in order to assess the performance of a visible light communication (VLC) system is nothing new. It can be estimated using a Monte Carlo optical ray tracing algorithm that sums the contribution of each ray on the reception plane. For now, research has focused on rectangular parallelepipedic rooms with single-textured walls, when studying indoor applications. This article presents a new open-source simulator that answers the case of more complex rooms by analysing them using a 3D STL (stereolithography) model. This paper describes this new tool in detail, with the material used, the software architecture, the ray tracing algorithm, and validates it against the literature and presents new use cases. To the best of our knowledge, this simulator is the only free and open-source ray tracing analysis for complex 3D rooms for VLC research. In particular, this simulator is capable of studying any room shape, such as an octagon or an L-shape. The user has the opportunity to control the number of emitters, their orientation, and especially the number of rays emitted and reflected. The final results are detailed heat maps, enabling the visualization of the optical power distribution across any 3D room. This tool is innovative both visually (using 3D models) and mathematically (estimating the coverage of a VLC system).

Keywords: VLC; ray tracing; open-source; Python; simulator; 3D rooms

Citation: Georlette, V.; Vallois, N.; Moeyaert, V.; Quoitin, B. Enhancing Visible Light Communication Channel Estimation in Complex 3D Environments: An Open-Source Ray Tracing Simulation Framework. *Informatics* **2024**, *11*, 80. https://doi.org/10.3390/informatics11040080

Academic Editor: Alessandro Pozzebon

Received: 30 August 2024
Revised: 23 October 2024
Accepted: 25 October 2024
Published: 31 October 2024

1. Introduction

Visible light communication (VLC) is a communication means that uses visible light emitted by LEDs to transmit information seamlessly. This technology has the capability to use lighting fixtures to send data, while lighting the surrounding environment. When the communication is bidirectional, the technology is called Li-Fi (light fidelity). The word is an analogy to Wi-Fi, which stands for wireless fidelity. To avoid harming the user's eyes, the uplink communication LED is often infrared, which is not visible to the naked eye. Various applications exist where unguided light is used as a communication means. The illustration in Figure 1 shows the three main examples: Li-Fi is the first example (under A), B is the common VLC application related to this paper, and C is optical camera communication (OCC). It worth mentioning that VLC, while being unidirectional, is sometimes used interchangeably with Li-Fi, depending on the authors. The terminology optical camera communication (OCC) is used when the light signal is decoded by a camera (of a smartphone for example). Finally, when lasers are used as the source, free space optics (FSO) is the acronym used, as seen in part D of Figure 1.

Within the scope of this paper, the focus is on VLC systems and the importance of modelling them in complex environment using an ad hoc simulator. The goal is to estimate useful parameters (e.g., reception power, presence of walls and their impact on the overall power, etc.) and, most importantly, their feasibility of implementation and expected performance.

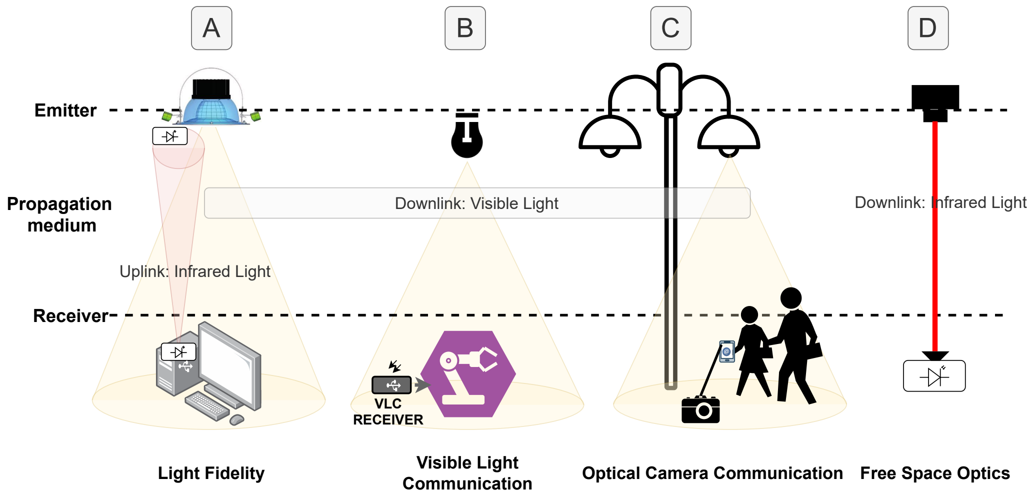

Figure 1. Various applications using unguided light as a communication means (all are referred to as OWC (optical wireless communication)). A: Li-Fi, B: VLC, C: OCC, D: FSO.

When one speaks of modelling a VLC system, first, the environment of the study must be defined. This could be an indoor room, a urban environment, an industrial environment, underwater, etc. Each of these environments modifies the light emitted by the LED due to propagation obstacles like walls (in interiors or outdoor), attenuation due to particles in the air (particularly in urban environments), and the presence of reflective metal walls (as in industrial environments). All these particularities impact the light in such a way as to reduce the quantity that reaches the receiver. As a result, it is important to be able to quantify these losses, to ensure the minimum communication performance that is intrinsically linked to a sufficient power of light reaching the receiver. Many environments can be imagined (space, hospital, mines, etc.) but the focus of the present work and simulator is placed on indoor rooms, as this is the most promising market segment for VLC.

Going back to part B in Figure 1, one can see that a VLC system is composed, like any telecommunication system, of an emitter, a receiver, and a propagation medium. The emitter, in the case studies in this paper, is composed of LEDs. LEDs are used in VLC thanks to their linear optical power response with respect to the current flowing through them. This allows a telecommunications engineer to modify this electric current to the shape of the data signal that is to be transmitted. Once the optical wave is propagated from the optical front-end (the LED), it is important to know how this light wave propagates through the so-called communication channel to reach the photodiode on the reception side. Figure 2 illustrates, for indoor and outdoor environments, a simplified VLC communication system. The aim being that *data*, the message to be transmitted, and *data*′, the message being received, are identical.

Generally speaking, when communication takes place indoors, the LEDs chosen radiate light evenly around the room. The first step in studying the performance of a VLC system in terms of data rate is to quantify the quantity of light that is distributed in the room. This amount of light allows having direct knowledge of the signal's strength at the receiver location, wherever it is in the room.

Quantifying this optical power distribution in a space is the output of channel modelling theory in VLC. Several techniques exist to model the communication channel:

- Ray Tracing: follows the trajectory of each light ray going from the LED until arriving at the receiver. This technique takes into account the reflection properties of the surfaces on the path of the ray and counts the number of rays arriving to calculate the power received [1].

- Geometrical Method: studies how the light is geometrically distributed in the room to estimate the portion of optical power arriving at a reception plane, making it suitable for basic VLC evaluations [2].
- Empirical Measurements: The channel behaviour is studied thanks to real-world data from VLC systems deployed in various environments. These measurements can be performed in the time or frequency domains [3].
- Statistical Channel Models: use probability distributions to represent channel characteristics based on empirical data. These models capture phenomena like path loss and multipath effects, generalizing performance across various environments [4,5].
- Machine Learning Models: leverage historical data to predict VLC channel characteristics by training models on various environmental factors [6,7].

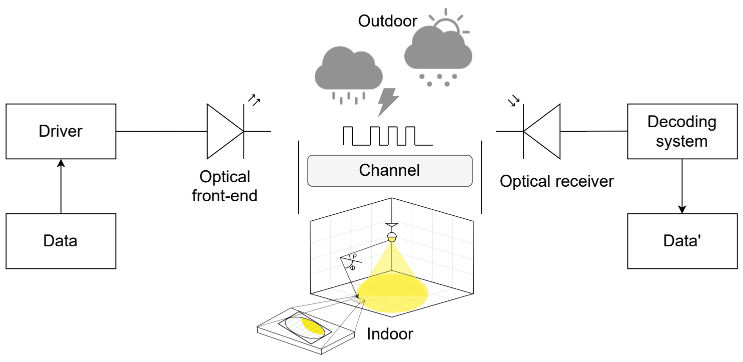

Figure 2. Simple VLC communication system. Illustration for indoor and outdoor environments.

Within the scope of this paper, the focus is put on the two most popular techniques, which are the geometrical and ray tracing approaches. The geometrical method follows how the light is geometrically distributed in the room to estimate the portions of optical power arriving at a reception plane. To achieve this, the light's radiation pattern is generally modeled by a Lambertian emitter whose power consumption is known. A Lambertian emitter is a light source that emits light uniformly in all directions according to an angular distribution, such that the perceived light intensity decreases proportionally to the cosine of the angle between the direction of emission and the normal to the surface of the emitter [8]. In the case of conventional indoor lighting, this hypothesis is valid. The key characteristics of the receiver's photodiode (such as its active reception area, its reception cone or field of view (FOV), and other electronic parameters) are incorporated into the model. Once the relative positions of the emitter and receiver are known, the room is meshed into small surface elements at floor level (the reception plane where the power received is estimated) and at wall level (if any, to estimate the portion of light coming from the lamp that is reflected). The smaller the surface elements, the closer the method becomes to a simplified ray tracing model; however, there are divergences. Nevertheless, if the surface element is small enough in relation to the distance between the emitter and the surface plane or wall, this algorithm can be likened to ray tracing, even if this is not entirely accurate. Figure 3 illustrates this principle. This technique was proposed in Prof Ghassemlooy's book [2]. We implemented it by transforming it into Python 3.9 [9] and applied its various scenarios to estimate communication performance [10–12]. When reflections are studied, it is possible to model a rather specular or diffuse surface (which is the case of a classical plaster wall). In Ghassemlooy's key work and our first simulator, walls had the property of being diffuse rather than specular. If a portion of optical power arrives at an element of the wall surface, this surface is assimilated to a point source, and the portion of optical power reaching it is itself redistributed following a Lambertian. To limit the number of calculations, only the portion of power re-emitted towards the receiving plane is studied in this method [9].

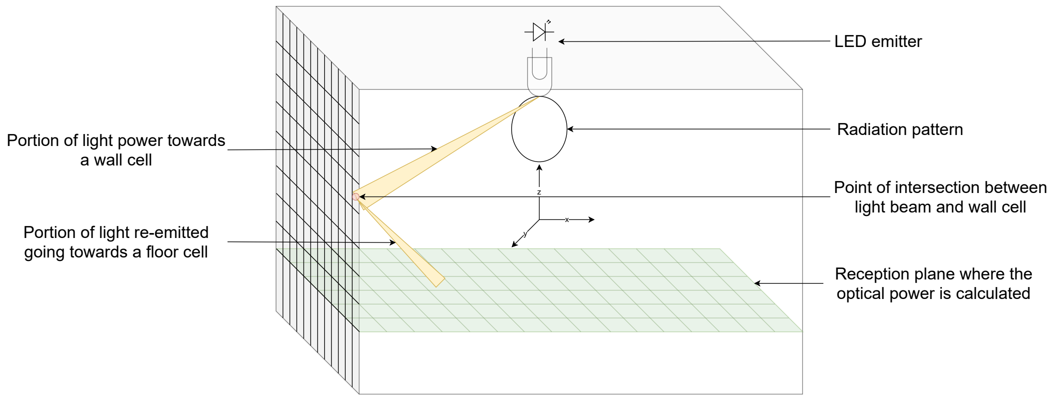
LED emitter
Radiation pattern
Portion of light power towards a wall cell
Point of intersection between light beam and wall cell
Portion of light re-emitted going towards a floor cell
Reception plane where the optical power is calculated
z
y
x

Figure 3. Illustration of our first simulator, where portions of light (cones) are assimilated to rays instead of real rays.

In contrast, the Monte Carlo method, which is the most popular ray tracing technique in VLC research, takes a different approach by simulating numerous random scenarios to estimate the light distribution. This method is based on shooting random rays from the light sources, following their trajectories through the room, and taking into account reflections, scattering, and absorption. Figure 4 illustrates random rays taken from an emitter. The light interactions are modeled stochastically, and the results of these simulations are used to obtain a statistical estimate of the light distribution. The method begins by emitting a large number of light rays from the emission source, each launched randomly in different directions. These rays represent the direction of the propagation of light in the room. Each ray is then followed along its path, taking into account interactions with surrounding surfaces, such as walls and objects. At each interaction, the followed ray may be reflected, absorbed, or scattered, depending on the properties of the surfaces it encounters, which is described using models specific to the materials and textures of the surfaces. In the present work, as the assumption is working indoors, plaster walls are assumed. Plaster walls are usually diffuse reflectors. By accumulating the results of numerous ray simulations, the Monte Carlo method provides a statistical estimate of the distribution of light in the room. The resulting statistics can be used to assess the optical power received at different positions of the receiver within the room and to understand the performance of the transmission channel [13–17].

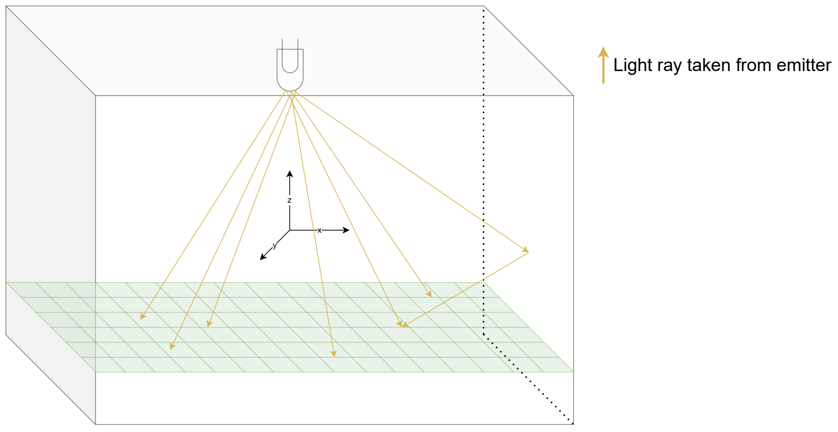

Figure 4. Monte Carlo method. Illustration of random rays taken from the emitter reaching the studied reception plane, with or without first-order reflection.

One of the main advantages of the Monte Carlo method is its flexibility, making it possible to model environments with varied light interactions. It is also accurate, as it directly simulates the trajectories of light rays and their detailed interactions. However, this method can be resource-intensive, requiring a large number of rays to obtain accurate results. This can lead to long calculation times and a high memory consumption. In addition, the quality of the results is highly dependent on the reflection model and surface properties used in the simulations.

In this paper, an open-source simulator is proposed that integrates the generation of random rays from the emitter, as recommended by the Monte Carlo method, giving the possibility to study the light propagation inside complex rooms. Complex rooms refer here to rooms whose walls are not limited to being perpendicular to each other (e.g., square or rectangular) but could be L or even polygonal-shaped (the case of an octagonal room is described further on), thanks to the integration of a 3D model (in STL—stereolithography—format). The aim of this paper is to highlight the computational and hardware advantages of using the ray tracing method over the conventional approach. To the best of our knowledge, this ray tracing VLC open-source simulator is the only one freely available and developed using open-source tools (Python) [18].

This paper first focuses on describing the functionalities, including the innovative functionalities such as the ray tracing algorithm and the 3D room integration, how the reflections are handled, and how the result figures are generated. Afterwards, the structure, material, and algorithms of the simulator are described in detail, and, finally, the last part focuses on how the results were validated and how new use cases can be studied (L or octagonal-shaped rooms).

2. Materials and Methods

This section details the materials and methods used to develop the open-source Python simulator for estimating the optical power distribution in complex 3D rooms. The use of Python was justified by its flexibility, extensibility for scientific computing, and its free-of-charge availability as an open-source environment. The simulator uses a ray-tracing approach to accurately model light interactions within complex 3D environments. First, Section 2.1 presents an overview of the simulator's main functionalities. Then, Section 2.2 describes its core architecture, including its inputs and outputs, and the detailed structure of each source file (see Section 2.2.1). Finally, the key algorithms upon which the simulator is based are thoroughly described, linking the functionalities and their practical implementation (see Section 2.2.2). The algorithms section comprises how the rays are generated, how they are rotated to meet the emitter's normal, how the collisions of the rays with the walls are taken into account, how their respective power is computed, how the reflections are handled, and finally how everything is put together to present the results.

2.1. Functionalities of the Simulator

Table 1 summarizes the functionalities of our ray tracing simulator for estimating the optical power distribution in VLC systems. They are all then detailed in the remainder of this subsection.

Table 1. Summary of the simulator's functionalities.

Functionality	**Description**
Custom 3D Environment Generation	Importation and configuration of custom 3D models, including material properties and light sources.
Ray Generation and Tracing	Calculation of light ray paths within the 3D environment, including reflections.
Communication System modelling	Mathematical models for the VLC channel, including channel attenuation.
Optical Power Distribution Estimation	Estimation of optical power distribution at various points, visualized through color maps and contour plots.

2.1.1. Custom 3D Environment Generation

To describe the geometry of the simulated rooms, the STL file format was chosen. This format describes only the surface geometry of a three-dimensional object. It has the advantage of being an open file format, widely supported by many 3D modelling software packages such as Blender 4.2 [19] and AutoCAD 24.3 [20]. In addition, the integration of the STL format with Python is facilitated by libraries such as numpy-stl. Other file formats exist such as the OBJ format, which includes additional information such as texture positions and vertex normals, which was not necessary for this application. Proprietary formats such as FBX and 3DS were also discarded, due to the constraints associated with their use in open-source projects. Blender was chosen to model the rooms in the present work.

To the best of our knowledge, few papers have integrated STL models into open-source VLC simulators. The most popular paper, written in 2015, used a dedicated software product called Zemax to create the 3D environment to be studied and then extracted the results using MATLAB [21]. Zemax is an optical design software used to model and simulate the propagation of light through complex optical systems, such as those found in VLC [22]. It can be used to create detailed models of light sources, receivers, and surfaces in an indoor environment. Once the simulations have been carried out in Zemax, the results can be exported and analyzed in MATLAB, a programming environment specializing in numerical computation and data analysis [23]. An update from the same team was published in 2020, but the conclusions were similar to the paper they wrote in 2015 [1]. Another paper dealing with more unusual rooms looked at circular and L-shaped rooms, again thanks to Zemax [24]. As both of these software programs are not free of charge, our previous simulator has the advantage of providing similar features, all coded in Python and freely accessible online [9].

In many cases in the scientific literature, the rooms studied in VLC research are rectangular parallelepipedic and assumed to be office spaces, classrooms, or living rooms. However, in certain situations, more complex room shapes have been studied. For example, reference [24] highlighted a relevant study of a hotel room, which are generally L-shaped. This is a point of interest for a VLC simulator. The methodology used in in our paper to deal with such room is to create the room under study using Blender, and then import the generated STL file into our new Python's simulator. The STL format is a type of file mainly used for 3D modelling and 3D printing. It describes the geometry of a 3D object using triangles. Each triangle is defined by its three vertices and a normal that indicates its orientation. In the case of this paper, the 3D object is the room under consideration. The simulator then analyses the file and can display the triangles that make up the room. It then identifies where the light rays arrive on its surfaces and calculates how they are diffused towards the reception plane under study.

Figures 5 and 6 show two STL files of a L-shaped room and an octagonal room, respectively, imported in the Python simulator. The black lines represent the rays emitted by the emitter towards each point of contact (plain yellow circle) with one of the walls. The bold green lines represent a ray emitted from the point of contact with one of the walls, symmetrically with respect to the normal of this wall and pointing inwards. In both illustrations (see Figures 5 and 6), only one ray is re-emitted, but in practice it is the reflective properties of the wall that dictate the nature of this reflection. On the one hand, if a mirror is present, the reflection will be purely specular. Meanwhile, if the wall is made of plaster, for example, the reflection will be diffuse and the point of contact between the beam emitted by the LED and the wall will itself become a diffuse emission point source that will re-emit a quantity of rays in different directions.

When modelling 3D objects in Blender, the surfaces' normals point outwards. However, it is possible to reverse the normal of faces, which is necessary in our ray tracing algorithm when the LED is located inside this volume. Appendix A explains in detail how to generate a room in Blender with the expected normal directions.

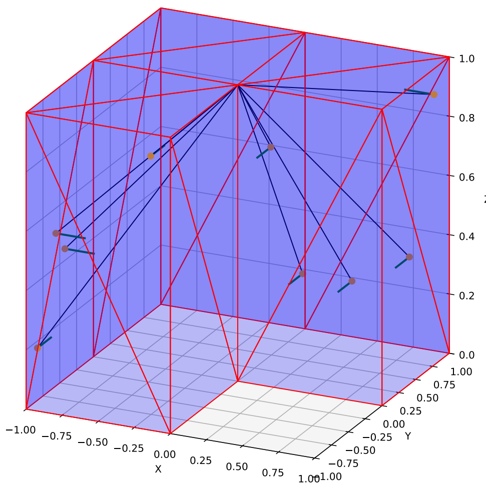

Figure 5. Highlighting of the first reflection with one ray in an L-shaped room.

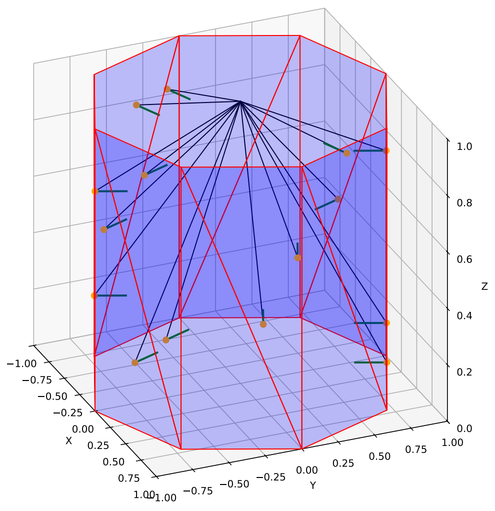

Figure 6. Highlighting of the first reflection with an octagon-shaped room.

2.1.2. Ray Generation and Tracing

The ray tracing algorithm forms the backbone of our simulator. Former VLC open-source simulators approximated a ray using a very narrow cone of light power coming out of the emitter [2]. This was predetermined and proportional to the area of the base of the cone (see Figure 3). When the Monte Carlo method is used, the simulator accuracy depends on the number of rays that the emitter spreads out in the room (see Figure 4). These rays are emitted uniformly in a half sphere located under the emitter. In a classic scenario, the emitter is assumed to be on the ceiling, pointing downwards. However, any orientation can be chosen in our simulator. Figure 7 shows three different orientations of the emitter. As mentioned above, the most common orientation is the emitter facing down, represented by the left sub-figure.

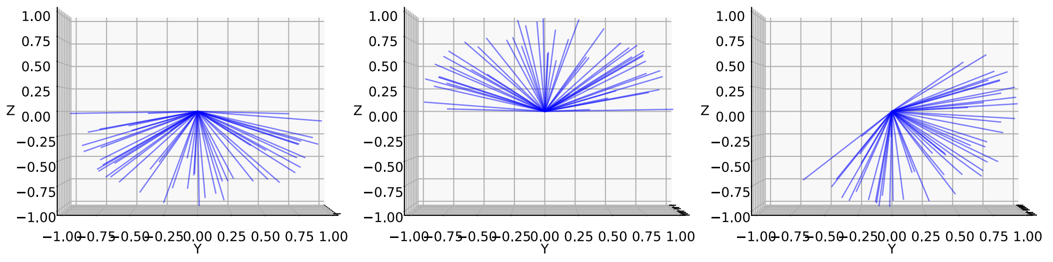

Figure 7. Different orientations of an emitter and the resulting emission of rays.

The generation of rays and the tracing algorithm comprise the following successive steps (see Figure 8), which model the path of light rays through the given environment:

1. Ray generation: The algorithm starts by generating rays from each emitter. These rays are emitted in a half sphere.
2. Rotation of the rays: The rays that are emitted in the half sphere undergo a rotation to be in the half-sphere direction in relation to the emitter's normal, this means in the direction of the light emission.
3. Tracing the rays: Once the rays have been emitted, the algorithm traces their unique path through the room. It checks whether the rays collide with any geometric objects (or wall) in the room or with the reception plane.
4. Contribution of the rays to the reception plane: When a ray reaches the reception plane, it contributes to the 2D matrix representing the distribution of the received power (in mW or dBm). This contribution is calculated using the line-of-sight (LoS) propagation model. This model takes into account the characteristics of the ray, as well as the angle of incidence at which it strikes the plane.
5. Reflection of rays off obstacles: If a ray encounters an obstacle before it reaches the reception plane, the algorithm re-emits several new rays from the point of collision, assimilating it to a new point source. These new rays are emitted in random directions relative to the normal vector of the wall. This assumes that the property of the wall makes it a diffuse emitter, which is the case for most indoor walls, as opposed to specular reflection which would be like a mirror.
6. Tracing the reflected rays: The reflected rays are then traced to determine whether they manage to reach the reception plane or not. If a reflected ray reaches the plane, its contribution is calculated using the non-line-of-sight (nLoS) propagation model. If none of the reflected rays reach the reception plane, these rays are discarded and have no impact on the final results.

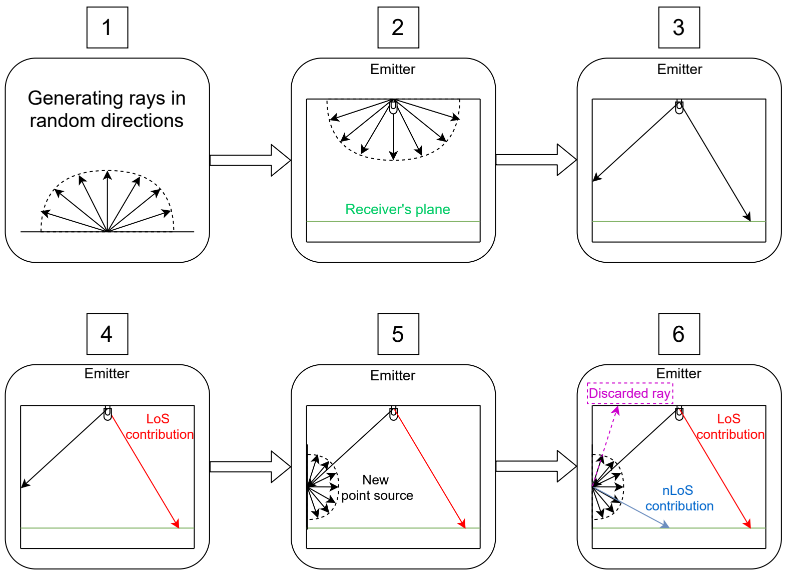

Figure 8. Propagation algorithm representation.

2.1.3. Communication System modelling

To estimate the communication coverage of a VLC system in a room, the simulator models the different elements composing the system, namely, the emitter, the receiver, and the propagation medium. In this work, one or more emitters (LEDs), from which the rays will be emitted, can be integrated into the simulator. The receiver's plane, where the coverage is assessed, is characterized by the properties of a photodiode. The propagation medium, which is the environment in which the emitter and receiver are located can be a room with any geometry. These different elements are modelled mathematically in the following part.

The LED Emitter

In general, LED lighting has a luminance that can be considered identical in all directions per unit solid angle. Elements with this property are called Lambertian emitters. This assumption is used in this work, and it will therefore be assumed that the LEDs used have a Lambertian radiation pattern [25], in which the radiant intensity depends on the irradiance angle θ. The function $R(\theta, \varphi)$ in Equation (1) represents the generalized Lambertian model for energy intensity. In practice, not all LEDs are Lambertian radiators. In urban and industrial environments, like street lighting for example, their radiation patterns are not necessarily symmetrical [10–12]. On the other hand, Lambertian radiators have a uniaxial symmetry to their radiation pattern, which makes it independent from φ (see Figure 9). This leads to Equations (1) and (2).

$$R(\theta, \varphi) = R(\theta) = \frac{m+1}{2\pi} cos^m(\theta) \tag{1}$$

$$m = -\frac{ln(2)}{cos(\theta_{1/2})} \tag{2}$$

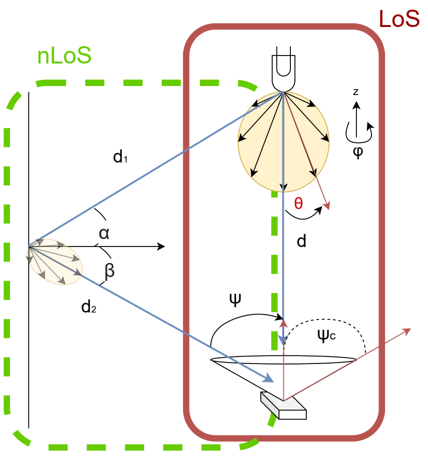

Figure 9. VLC system modelling.

The parameter m (see Equation (2)) represents the mode number, which determines the directivity of the radiation pattern. The half-power angle, $\theta_{1/2}$, also known as the half-power beamwidth, is the angle between the axis of the maximum power and half-power points. Its value is often given in LED data sheets; an ideally Lambertian emitter has a half-power angle of 60 degrees. In addition, there is a parameter to take into account the relative total power of the LED. This means that all the individual rays must have a power that adds up to the total power of the LED. The transmitter is therefore modeled as a set of rays that come from a common origin and have random directions within a hemisphere. The power of each of these rays is therefore the total power of the LED, weighted according to the Lambertian radiation pattern.

The Photodiode Receiver

In order to ensure that the room is adequately covered, a reception plane is drawn to study the performance. The plane is divided into a grid of small cells, and the number of light rays hitting each cell is counted. To take into account the properties of a real VLC receiver, each cell of this grid is modeled as a photodiode, to mimic a conventional receiver that scans the room. A photodiode is a semiconductor device that converts light into an electric current when exposed to light. It generates a current proportional to the power of the light, enabling light to be detected and measured. Its main characteristics are its field of view (FOV), active reception surface, and refractive index. If the angle of incidence of the light beam does not enter the receiver's FOV, then the power received is zero. From these parameters, the gain of the photodiode can be calculated (see Equation (3)).

$$g(n, \psi_c) = \frac{n^2}{sin^2(\psi_c)} \tag{3}$$

In Equation (3), the parameter $g(n, \psi_c)$ represents the optical gain of the photodetector that has a field of view angle $\psi_c < \pi/2$ and a concentrating lens on top with a refractive index n.

The Propagation Medium

Figure 9 shows a VLC system model and the different angles involved in the computation of the power received. Given the information above and the relative position of the emitter with respect to the receiver, the total power received at each point of the receiver's plane can be estimated using both Equations (4) and (5). Equation (4) represents the power of the rays that directly fall into the receiver's field of view when emitted from the LED. They follow a trajectory which has a distance d. This is called line-of-sight (LoS) communication. A_{rx} is the receiving surface of the photodiode and $cos(\psi)$ the inclination between the emitter and the receiver. On the other hand, the reflected rays that contribute to the power received are computed using Equation (5), called the non-line-of-sight (nLoS) contribution. These rays follow a path corresponding to d_1 and d_2. Depending on the type of reflection on the wall, either a specular or diffuse reflection can be considered. In the case of this work, the point where the first ray reaches the wall itself becomes a light source following a Lambertian emission. The reflection coefficient of the wall is ρ_{wall}. Once each contribution has been calculated, the total power on each point of the receiver's plane is the sum of both contributions, as stated in Equation (6). In this work, only the first reflection is taken into account. Considering first-order reflections as sufficient is often a question of a compromise between the accuracy and complexity of the calculation. As additional reflections have an increasingly small effect on received power, this justifies their neglect in practical calculations [2].

$$P_{r-LoS} = P_t H_{LoS}(0) = P_t R(\theta, \varphi) g(n, \psi_c) \frac{A_{rx}}{d^2} cos(\psi) \tag{4}$$

$$P_{r-nLoS} = P_t H_{nLoS}(0) = P_t \frac{R(\theta, \varphi) A_{rx}}{(d_1 d_2)^2} \rho_{wall} cos(\alpha) \cos(\beta) g(n, \psi_c) cos(\psi) \tag{5}$$

$$P_{r-total} = P_t(H_{LoS}(0) + H_{nLoS}(0)) \tag{6}$$

2.1.4. Optical Power Distribution Estimation

Taking into account the quantities estimated in the communication system modelling (Equation (6)), the simulator outputs three-dimensional curves representing the optical power distribution in the room. These 3D curves represent the optical power either in milliwatt (mW) or decibel-milliwatt (dBm), with the relationship between both being written as

$$P_{dBm} = 10log_{10}(P_{mW}) \tag{7}$$

The power curve generation allows the user to move the surface in any direction. The default view is the top view. The abscissa and ordinate values correspond to the dimensions of the room being studied in meters. One unit therefore corresponds to one meter. This is also the case in STL files. It is therefore important to bear this in mind when creating 3D parts with Blender or any other 3D design software.

In order to better comprehend the matrices presenting the values of the reception plane, these are visualized in 3D using the Matplotlib library. In this example (see Figure 10), a Lambertian transmitter is placed in the center of a ceiling in a square room measuring 5 m on each side and with a height of 3 m. The transmitter is placed 2.5 m above the floor, which is the receiving plane that has been divided into small areas where the amount of optical power has been quantified. Table 2 shows the parameters used to generate the 3D visualization of the room in Figure 10. Figures 11 and 12 represent received power distribution examples, to explain how to interpret these curves as well as the output of the simulator.

Table 2. Parameters of the example room.

	Parameter	Value
Environment	Room size	$5 \times 5 \times 3$ m^3
	Reflection coefficient	80%
Emitter	Position	(0; 0; 2.5)
	Orientation	Facing down
	Half Power Angle	60 degrees
	Power	2 W
Receiver	Position	The ground
	Active area	16 mm^2
	Field of View	60 degrees
	Refractive index	1.5

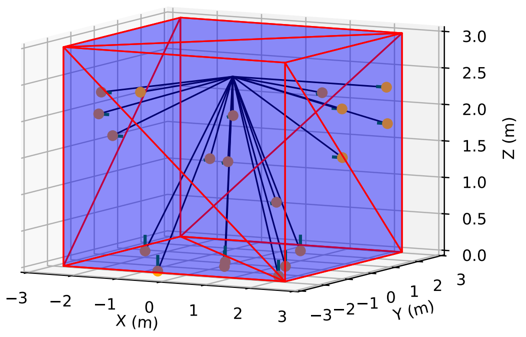

Figure 10. Illustration of the cubic room example.

The left side of Figure 11 represents the optical power distribution in mW considering the LoS rays. In this example, the LED emits 10 batches of 10,000 LoS rays and 100 reflected nLoS rays. The right side of Figure 11 represents the nLoS contributions of optical power. As the emitter is in the center of the ceiling, the curve shows that the reflected rays will be more significant towards the walls than in the center of the room.

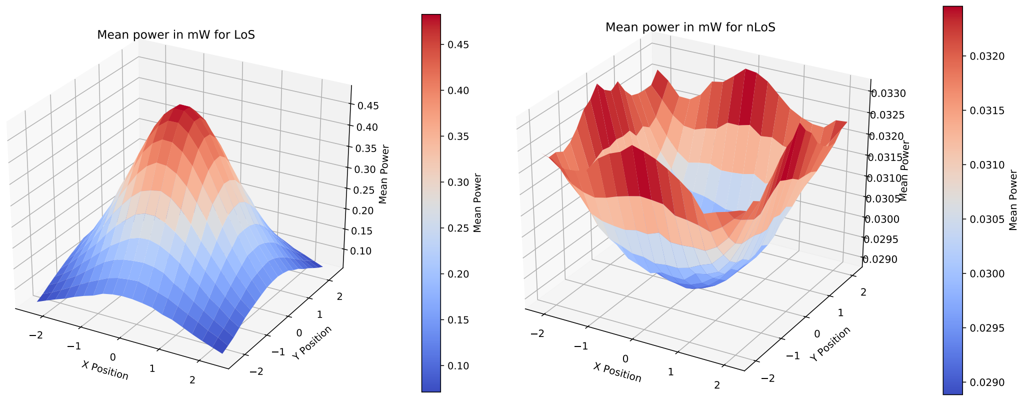

Figure 11. (**left**) LoS ray contribution and (**right**) nLoS ray contributions in mW to total distributed power.

The left side of Figure 12 represents the sum of both LoS and nLoS contributions presented in Figure 11. One can see that the contribution of the nLoS rays is really small compared to the direct rays. Nevertheless, the study of nLoS rays can be useful in other types of room configurations, where nLoS rays might propagate further than LoS rays. The right side of the figure is the scale conversion from mW to dBm of the sum of both contributions.

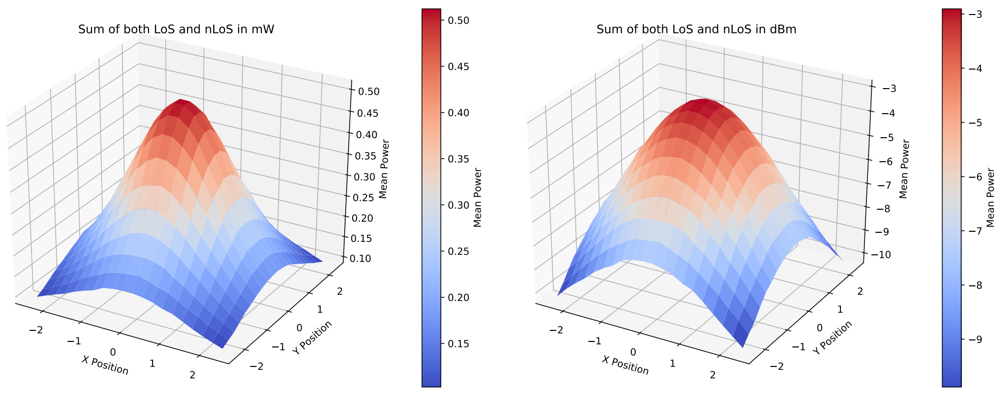

Figure 12. Sum of both LoS contributions in mW and dBm.

2.2. Overview of the Simulator

In this section, the simulator's architecture is described, highlighting the various aspects of its features and operation. First, the main components of the simulator are presented, explaining how data are introduced and processed within the system. In particular, the various inputs to the simulator, such as the model configuration parameters and 3D object specifications, and the outputs produced, such as the results of the optical power distribution simulation, are discussed. The structure of the code is then described in detail, outlining the main modules and their interactions. The aim of this section is to provide an in-depth understanding of how the code is organized, so that how the simulator's various functionalities are integrated and implemented can be understood. The architectural choices and technical considerations that guided the development of the simulator are also discussed. In summary, this section provides a comprehensive overview of the simulator architecture, including both functional and technical aspects, to provide a solid basis for further analysis and discussion.

2.2.1. Simulator's Architecture

Input/Output

Figure 13 presents an overview of the simulator's architecture. The different inputs include the STL file of the 3D room, the intrinsic parameters of the transmitter and receiver and their relative positions, and finally the simulation parameters, such as the number of rays. The output of the simulator is mainly the matrices (or 2D grids) representing the power received in the receiver plane, followed by a 3D representation thanks to Matplotlib.

The detailed inputs to model the propagation of light rays in a given environment are the following:

- Room geometry
  The 3D room's geometry is represented by a list of triangles, each defined by a normal vector and a reflection coefficient ρ. Triangles were chosen to represent the geometry because of their simplicity and flexibility. A triangle is the most basic geometric shape that can be used to model complex surfaces with precision. By subdividing curved or irregular surfaces into a mesh of triangles, it is possible to accurately represent almost any geometric shape, while facilitating ray tracing calculations. These triangles can be created directly in Python's code or imported from an STL file, a format chosen for its compatibility and simplicity.
- Transmitter(s) and receiver plane
  Mandatory information on transmitters includes their number, positions, orientations, and powers, as well as their half-power angles ($\theta_{1/2}$). The latter represents

the angular distribution at which the rays are emitted into the room. The receiver's plane represents the surface where the signal's coverage needs to be assessed. This is characterized by the receiver's FOV angle (ψ_c), the area of the receiver A_{rx}, the refractive index of the receiver n, and the height at which it is located in the room (see Equations (4) and (5)).

- Simulation parameters
  These include the number of rays to be emitted per emitter, the reflections per ray touching a surface, the number of batches used to optimize the simulation, and the desired resolution of the final result matrix. The use of batches resulted from the conclusion that efficient use of NumPy, a Python library that makes it easy to work with arrays of numbers and mathematical calculations, is essential to reduce the computing time and is quite unique to our work. By grouping rays into large matrices, rather than processing them individually in separate Python objects, it was possible to take advantage of NumPy's optimized matrix operations. This approach makes it possible to perform calculations on all the data simultaneously, considerably speeding up the processing.

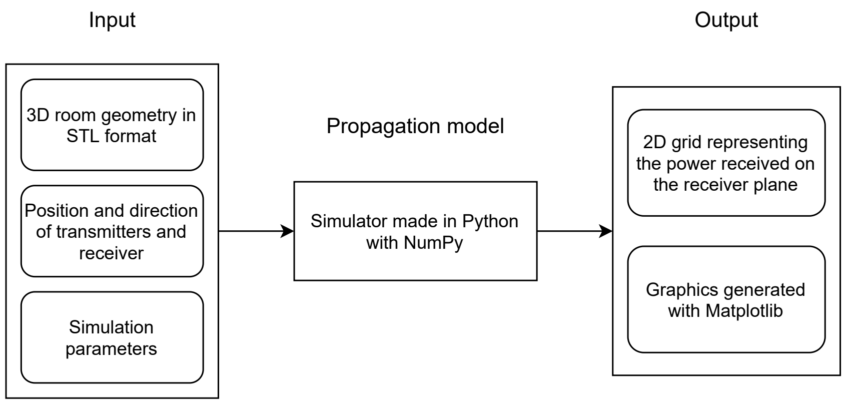
Input
3D room geometry in STL format
Position and direction of transmitters and receiver
Simulation parameters
Propagation model
Simulator made in Python with NumPy
Output
2D grid representing the power received on the receiver plane
Graphics generated with Matplotlib

Figure 13. General architecture of the Python VLC ray tracing simulator.

The propagation model then simulates the trajectory of the rays from the emitters, taking into account the geometry of the room and the defined parameters. The rays passing through the receiver plane are then aggregated into a 2D matrix, where the plane is divided into several squares of equal size. This matrix represents the distribution of the received power and can be visualized as a 3D surface plot, as explained in the previous section.

Python was chosen for its cost-effectiveness, ease of development, and extensive libraries for scientific calculations (like NumPy) and data visualization (like Matplotlib). However, it has performance limitations compared to compiled languages, and dynamic typing can lead to runtime errors. These issues are mitigated through optimized libraries, techniques like type hinting, and thorough documentation for improved code maintenance. All the software material can be found at the following reference [18].

Code Structure

This part of the paper describes the various files that compose the ray tracing simulator code. Each file plays a specific role in the overall functionality of the simulator. Figure 14 illustrates each function or class in the code and how each file contributes to the overall operation of the simulator. There are six files in the simulator: `main`, `utils`, `rays_math`, `simulator`, `random_vector` and `figures`.

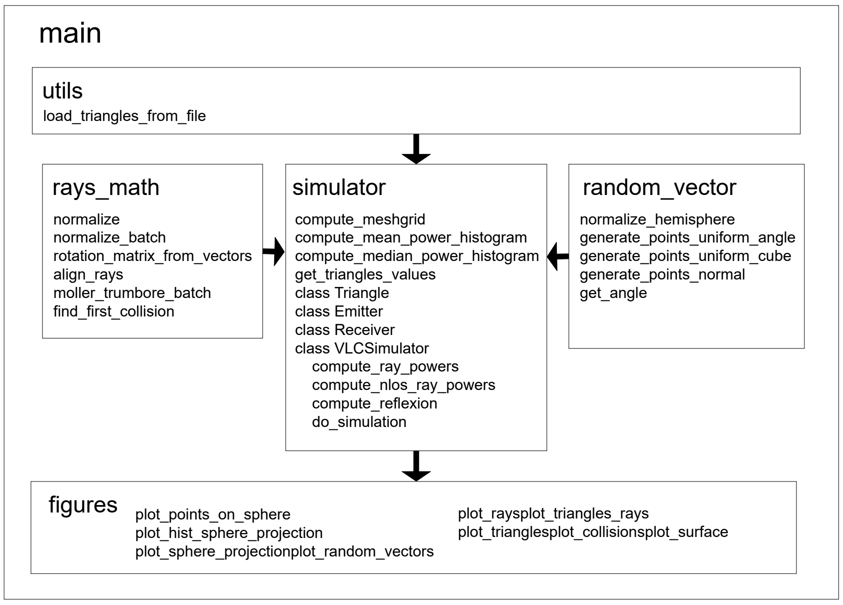

Figure 14. Structure of the simulator's open-source code.

File `utils.py`

The `utils.py` file contains the function for importing an STL file of a 3D room into the Python environment using the function `load_triangles_from_file`. This function retrieves triangles and their normals from the STL file provided. It is essential for loading the geometric data required for the simulation and converting them into objects that can be used by the rest of the code.

File `rays_math.py`

The `rays_math.py` file is dedicated to mathematical calculations related to rays. It includes functions to perform operations such as rotating vectors and detecting collisions between vectors and triangles in geometry. The details of these functions and their implementation are described in depth in Section 2.2.2.

File `random_vector.py`

The `random_vector.py` file includes functions for generating vectors in random directions within a half hemisphere. The detailed operation of these functions is explained in Section 2.2.2, where the specific algorithms used for these generation processes are discussed.

File `simulator.py`

The `simulator.py` file constitutes the core of the simulator code. It contains the main logic and essential classes for the simulator's operation:

- Class `Emitter`: represents the emitters. It consolidates data related to emitters and provides a structure for managing this information, such as the calculation of the Lambertian emission order m (Equation (2)).
- Class `Receiver`: represents the receiving plane and manages data associated with the surface receiving the rays.
- Class `Triangle`: represents triangles within the geometry, storing information such as vertices, normals, and the reflection coefficient ρ associated with each triangle.
- Class `VLCSimulator`: This is the main class of the simulator. Upon instantiation, it handles the emitter objects, the receiver object, and the list of triangles. The method `do_simulation` is responsible for executing the simulation using these objects and the defined simulation parameters. The details of this method's functionality are explored in Section 2.2.2.

File `figures.py`

The `figures.py` file is dedicated to generating various figures for visualizing the simulator's results. It includes functions for creating graphical representations of the simulation results. Additionally, a corresponding Jupyter notebook file named `figures.ipynb` provides code examples and associated results, facilitating the understanding of the simulator's graphical functionalities [18].

2.2.2. Algorithms

The main algorithms used in the simulator are presented in this part of the paper. Each of these algorithms plays a crucial role in the simulation and data processing. The methods used for ray generation, ray tracing, communication modelling, and results visualization are described in detail. First, the algorithm which deals with the methods used to generate the rays needed for the simulation is presented. It defines how the rays are emitted from a source and how they are distributed in space. Second, the ray tracing, a key process that determines the path of the rays through the simulated environment, is presented. This algorithm calculates the intersections between rays and objects, as well as the effects of reflection and transmission. Then, the methods used to model aspects of optical communication in the simulator are explained. These include simulating the interactions between the rays and the receiving surfaces, as well as evaluating the performance of the communication system. Finally, the visualization techniques used to present the simulation results are explained. Notably, this involves how the data are transformed into graphical representations, making it easier to analyze and interpret the results.

Ray Generation

The algorithm begins by iterating over the different emitters, which means that each emitter emits its own rays independently. To generate rays in the simulator, several techniques were considered for creating directions in a half sphere. A first naive way would be to generate spherical coordinates where ϕ and φ follow a uniform law. An illustration of these angles can be found in Figure 15, where ρ is the radial distance, ϕ the colatitude (between 0 and π radians), and φ the longitude (between 0 and 2π radians).

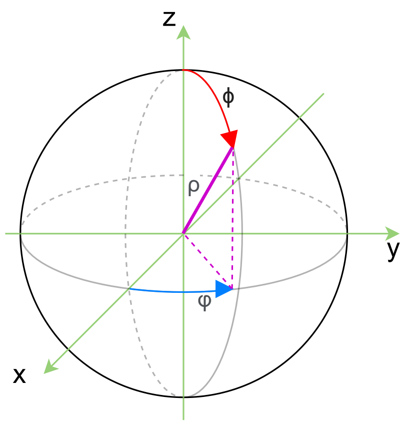

Figure 15. Representation of spherical coordinates.

Generating 10,000 points on a half sphere or rays using this method gives Figure 16. It can be seen that the distribution does not appear to be uniform. Indeed, there is a visibly higher concentration of rays in the center part of the half sphere, as seen on the left side of Figure 16.

The algorithm finally adopted in this work was proposed by [26], which is based on the use of independent coordinates X, Y, and Z generated according to a normal distribution (Equation (8)) centered reduced:

$$X, Y, Z \sim \mathcal{N}(0, 1) \tag{8}$$

After their generation, the X, Y, and Z coordinates are normalized to ensure that the radius has a unit length. This normalization ensures that the ray directions are represented uniformly across the sphere. To ensure that rays are generated only in the half sphere where all directions have a positive Z value, the absolute value of Z is taken. This method ensures that the ray directions are uniformly distributed across the half sphere. The distribution of the points generated thus follows a uniform distribution over this surface, as illustrated in Figure 17, which represents a generation of 10,000 points using this method. Both Figures 16 and 17 were generated using the `generate_points_normal` function within the `random_vector.py` file.

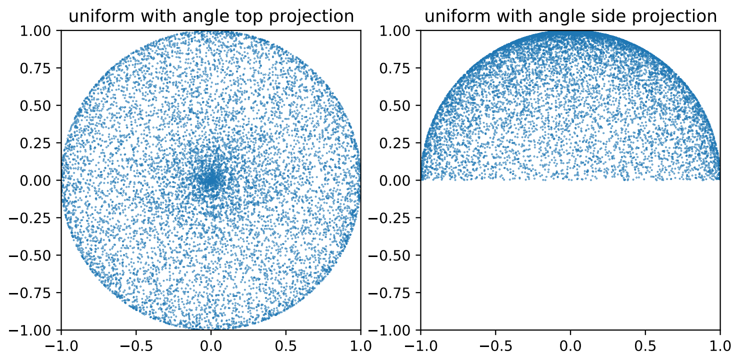

Figure 16. Randomly generated points on a sphere following a uniform distribution using spherical coordinates.

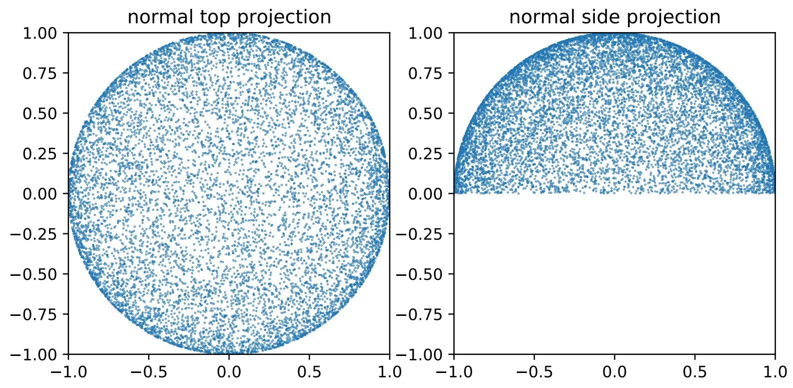

Figure 17. Randomly generated points on a sphere using a normal distribution of X, Y, and Z coordinates.

Rotation of Rays

After generating the rays in a half sphere directed with respect to the vector (0, 0, 1) (facing up), they have to be orientated in the specific direction of the emitter. To that end, the generated rays must be rotated by the angle between the vector (0, 0, 1) and the desired direction of the emitter. The rotation matrix that will be applied to each ray needs to be calculated to complete this step. To begin with, the reference vectors are computed. The v vector is the initial direction of the generated rays. This vector must be aligned with another vector w, representing the direction of the emitter (see Figure 18). To achieve this, these two vectors are normalized to obtain the unit vectors $\hat{v}$ and $\hat{w}$. These unit vectors are obtained by dividing each vector by its norm:

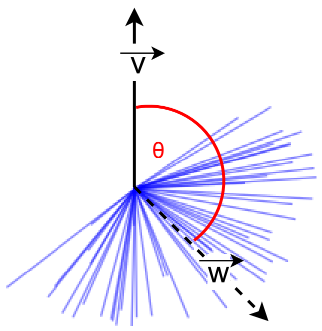

Figure 18. Vectors v and w for ray rotation.

$$\hat{v} = \frac{v}{\|v\|} \tag{9}$$

$$\hat{w} = \frac{w}{\|w\|} \tag{10}$$

Subsequently, the cross product (also known as the vector product) of $\hat{v}$ and $\hat{w}$ is computed to obtain a vector u that defines the axis of rotation. The norm of this cross product provides the sine of the rotation angle θ, as both $\hat{v}$ and $\hat{w}$ vectors are unitary, while the dot product (a.k.a. scalar product) between $\hat{v}$ and $\hat{w}$ yields the cosine of this angle.

$$\sin(\theta) = \|\hat{v} \times \hat{w}\| \tag{11}$$

$$\cos(\theta) = \hat{v} \cdot \hat{w} \tag{12}$$

In the case where the vectors are collinear, two particular situations must be considered. If $\sin(\theta)$ is zero, this indicates that the vectors are either in the same direction, in which case no rotation is required and the rotation matrix is simply the identity matrix I, or in opposite directions, in which case the rotation matrix is the negative of the identity matrix, that is $-I$.

For the other situations, the rotation matrix R is constructed using Rodrigues' formula [27]. This formula involves first creating a square matrix K, which is the antisymmetric matrix (it satisfies the condition $K^T = -K$.) of the rotation vector u:

$$K = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix} \tag{13}$$

The rotation matrix is then calculated by

$$R = I + K\sin(\theta) + K^2(1 - \cos(\theta)) \tag{14}$$

Once the rotation matrix R has been determined, this rotation is applied to the generated rays, to realign them with the direction of the emitter. This application is performed by multiplying each ray by the transpose of the rotation matrix R^T:

$$\text{aligned rays} = \text{rays} \times R^T \tag{15}$$

This algorithm allows the generated rays to be correctly aligned with the direction of the emitter. Two examples, where rays are aligned with a vector that has a direction $(0, 0, -1)$ (facing down) and a vector that has a direction $(0, 1, -1)$ (facing down towards the right), are shown in Figure 7.

The rotation is implemented in the function `align_rays`, which is located in the file `rays_math.py`.

Calculating Collisions with Triangles

Now, the intersections between rays and triangles are needed. To achieve this, the Möller–Trumbore algorithm is used [27,28]. This algorithm is well-suited for ray tracing calculations, as it allows for efficient testing of collisions between rays and triangles in three-dimensional space. This algorithm employs barycentric coordinates. Barycentric coordinates are a coordinate system used to express a point within a triangle relative to its distances from the triangle's vertices. These coordinates are expressed as a triplet (a_1, a_2, a_3), such that $a_1 + a_2 + a_3 = 1$.

The Möller–Trumbore algorithm consists of several steps. Firstly, the coordinates of each triangle's vertices must be extracted. For each triangle, there are three vertices: v_0, v_1, and v_2. The triangle's edge vectors are defined as follows:

$$e_1 = v_1 - v_0 \tag{16}$$

$$e_2 = v_2 - v_0 \tag{17}$$

Next, the cross product between the ray direction and the edge vector e_2 is computed to obtain the vector h. The result of this cross product is used to calculate the determinant a, which is the dot product between h and the edge vector e_1:

$$h = d \times e_2 \tag{18}$$

$$a = e_1 \cdot h \tag{19}$$

If the determinant a is close to zero ($<1\times10^{-8}$), this indicates that the ray is parallel to the triangle. In this case, no intersection is possible, and such cases are marked as parallel and excluded from further calculations.

If a is not close to zero, the calculation continues by determining the factor f, which is the reciprocal of a. This factor is used to normalize the coordinates and is defined as follows:

$$f = \frac{1}{a} \tag{20}$$

Next, the vector s, which is the difference between the ray's origin and the triangle's vertex v_0, is computed. From this vector, the barycentric coordinates u and v of the potential intersection point are determined:

$$s = o - v_0 \tag{21}$$

$$u = f \cdot (s \times e_2) \cdot e_1 \tag{22}$$

$$v = f \cdot (d \times e_1) \cdot s \tag{23}$$

The barycentric coordinates u and v represent the position of the intersection point relative to the triangle's vertices. For the intersection point to lie within the triangle, the barycentric coordinates must satisfy the following conditions: $u \geq 0$, $v \geq 0$, and $u + v \leq 1$. If these conditions are not met, there is no intersection with the triangle.

If the barycentric coordinates are valid, the distance t between the ray's origin and the intersection point is calculated using the factor f:

$$t = f \cdot (e_2 \times e_1) \cdot s \tag{24}$$

This distance t gives the position of the intersection point along the ray. Intersections where the ray does not hit the triangle are set to infinity. This indicates that there is no collision.

For each ray, the closest triangle is identified by comparing the distances calculated for all possible intersections. The method `find_first_collision` (in `rays_math.py` in Figure 14) returns the indices of the nearest triangles, along with the associated distances. The indices of the triangles where no collision occurs are set to -1, and the distances of intersections are set to infinity if no collision is detected. Figure 19 shows an example of the algorithm in action: 20 rays have been emitted towards the top of an octagonal-shaped room. It can be seen that collisions are accurately accounted for, and the normal vectors of the walls that the rays have interacted with are also displayed.

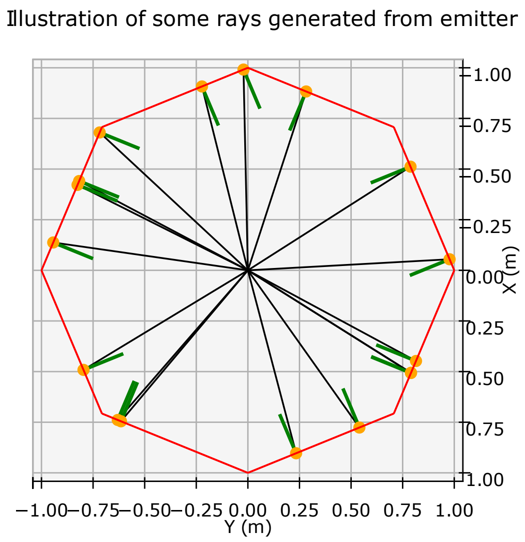

Figure 19. Top view with rays launched into the room from the ceiling center, collision points with surfaces in orange and wall normals in green facing inwards.

Calculating the Power of Direct Rays

Once those collisions have been identified for each ray, they can be categorized into several types, each treated differently:

- Rays that do not intersect any triangle are disregarded and not considered for further calculations.
- Rays that collide with the geometry of the room are retained for reflection calculations.
- Rays that reach the receiver plane require calculation of their power on this plane.

To calculate the power of direct rays when they reach the receiver, the parameters defined in Equation (4) need to be retrieved. Firstly, the knowledge of the cosine of the emission angle θ between the direction of the rays (d_r) and the direction of the emitter (d_e) is necessary. This cosine can be obtained by performing the dot product between these two directions:

$$\cos(\theta) = d_r \cdot d_e \tag{25}$$

Next, the cosine of the angle ψ, which is the angle between the opposite direction of the rays $-d_r$ and the vector $z = [0, 0, 1]$ (oriented along the vertical axis) is needed (see Figure 9). This cosine is computed as follows:

$$\cos(\psi) = -d_r \cdot z \tag{26}$$

The parameters m and P_t are constants of the emitter, provided or calculated during the creation of the object. The values A_{rx} and g are constants of the receiving plane, also determined during the creation of the object. The distance d_1 between the emitter and the point of collision on the receiving plane is obtained from the previous step using the Möller–Trumbore algorithm. It is then necessary to verify that the ray falls within the FOV of the receiver on each cell of the receiver's plane by verifying that the incidence falls within ψ_c.

These calculations are performed in the method `compute_ray_powers`, which is located in the file `simulator.py` (see Figure 14).

Calculation of Reflections

When a ray collides with the geometry, reflections must be considered. The simulator accounts only for diffuse reflections. To model these reflections, a new light source is conceptualized at the point of collision on the surface, from which n_{nr} new rays are emitted. The value of n_{nr} is a parameter of the simulation.

To generate these new rays, the same techniques as those previously described for the emitter are applied. The n_{nr} rays are initially generated randomly within a half hemisphere and then oriented according to the normal to the collision surface by applying an appropriate rotation. Subsequently, the Möller–Trumbore algorithm is again used to detect intersections between these new rays and the geometry triangles, as well as the receiving plane.

Only the rays that collide with the receiving plane are considered; others are ignored. The power of a ray after reflection can then be calculated using Equation (5), where ρ_{wall} is the reflection coefficient of the wall, which should be indicated as a parameter in the code. For d_1 and d_2 (see Figure 9), the results from the two applications of the Möller–Trumbore algorithm are used. The value of ρ (reflection coefficient) is associated with the triangle with which the collision occurred.

The value of $\cos(\alpha)$, which is the angle of incidence between the opposite direction of the incident ray d_1 and the normal to the wall n, is given by

$$\cos(\alpha) = -d_1 \cdot n \tag{27}$$

Next, the cosine of the angle β is calculated. This is the angle between the normal n of the collision surface and the direction of the reflected ray d_2. This cosine is also computed using the dot product:

$$\cos(\beta) = n \cdot d_2 \tag{28}$$

Finally, the cosine of angle ψ, which is the angle between the opposite direction of the reflected ray $-d_2$ and the vector $z = [0, 0, 1]$, representing the vertical axis, is calculated as follows:

$$\cos(\phi) = -d_2 \cdot z \tag{29}$$

Similarly, the angle ψ can be computed, and if angle ψ exceeds the receiver's field of view ψ_c , the power received by this ray is set to zero:

$$P_{\text{received}} = 0 \text{ if } |\psi| > \psi_c \tag{30}$$

These calculations are performed in the method `compute_reflection` and in the method `compute_nlos_ray_powers`, which are found in the file `simulator.py` (see Figure 14).

Aggregation of Results

At this stage, the ray propagation process is complete. A set of points representing collisions between the rays and the receiving plane, each associated with an intensity, are obtained. The ultimate goal of the simulator is to produce a two-dimensional matrix that represents the power density on the surface of the receiving plane. To obtain this matrix, the receiving plane is divided into a grid. For each cell in this grid, the average power of the rays that have landed in that cell are computed.

The function `compute_mean_power_histogram` handles this process by taking as input the impact positions of the rays (x_i, y_i) and the associated powers P_i at these impacts. The grid boundaries are also provided in the form of the arrays `x_edges` and `y_edges`. Firstly, this function calculates a weighted two-dimensional histogram called `total_power_hist`, which accumulates the total power in each cell of the grid. Mathematically, this involves summing the powers of the rays that fall into each cell:

$$\text{total\_power\_hist}(x, y) = \sum_i P_i \quad \text{where } (x_i, y_i) \text{ is in cell } (x, y) \tag{31}$$

Then, a second two-dimensional histogram, `counts_hist`, is calculated. This histogram simply counts the number of rays that have landed in each cell of the grid:

$$\text{counts\_hist}(x, y) = \sum_i 1 \quad \text{where } (x_i, y_i) \text{ is in cell } (x, y) \tag{32}$$

It is important to note that to avoid division by zero (which would occur if a cell contains no rays), zero values in `counts_hist` are replaced with 1. The average power in each cell is then obtained by dividing the `total_power_hist` by the number of rays in that cell:

$$\text{mean\_power\_hist}(x, y) = \frac{\text{total\_power\_hist}(x, y)}{\text{counts\_hist}(x, y)} \tag{33}$$

Finally, this matrix `mean_power_hist`$^{\mathrm{T}}$ represents the average power density on the receiver plane. This result is essential for visualizing and analyzing the distribution of received energy across the plane.

These calculations are performed in the methods located in the file `simulator.py` (see Figure 14).

3. Results

This section of the paper presents the results obtained with the simulator in different scenarios. This includes comparisons with theoretical models, as well as benchmarking against existing simulators. This validates the accuracy and reliability of the output of the simulator. Next, the propagation results in rooms with geometries different from a rectangular parallelepiped room are analyzed. Finally, the computational requirements and efficiency of the simulator are described, this includes the simulator's calculation time in different situations. The simulations were performed using an AMD Ryzen 5 4600H CPU and 16 GB of RAM manufactured by Advanced Micro Devices (SC, USA).

3.1. Validation

To the best of our knowledge, two open-source simulators are available in the research literature. The first was developed in MATLAB 2019 and is available in reference [2]. The second is the simulator we previously implemented in Python based on the MATLAB version but with enhanced features [9,10,12]. Both are based on the methodology presented in Figure 3.

3.1.1. Comparison with a Single Emitter in a LoS Configuration

In this simulation scenario, a square room measuring $5 \times 5 \times 3$ m^3, with a single transmitter placed in the center of the ceiling, was studied in terms of propagation. The parameters related to the pieces of equipment and the room are given in Table 2. Figure 20 shows the simulation results using the code from [9]. Figure 21 shows the results of this simulator developed with 1,000,000 rays (100 batches of 10,000 rays in LoS). It can be seen that the results were very similar.

To quantify the difference between the results obtained by the developed simulator and the reference simulators, the mean square error (MSE) was calculated using the mW values of the power. This error, which measures the mean squared deviation between the simulated and reference received powers, was 0.086 when estimating those two results, indicating a good simulation accuracy.

In addition to this static measurement, it is also interesting to examine how the error varied as a function of the number of rays emitted. Figure 22 shows the evolution of the MSE as a function of the number of rays emitted. To generate this figure, the number of rays was adjusted from 1000 to 50,000 in increments of 1000. At each increment, 100 simulations were carried out, and the average result of these simulations was taken. This is carried out to reduce the impact of random variations and to obtain a more reliable estimate of the average error. It can be seen that the error decreased as the number of rays increased, reflecting a better approximation of light propagation in the simulated environment.

Figure 23, a zoom of the previous graph, also shows that from 35,000 rays onward, the error was close to zero and started to decrease very slightly.

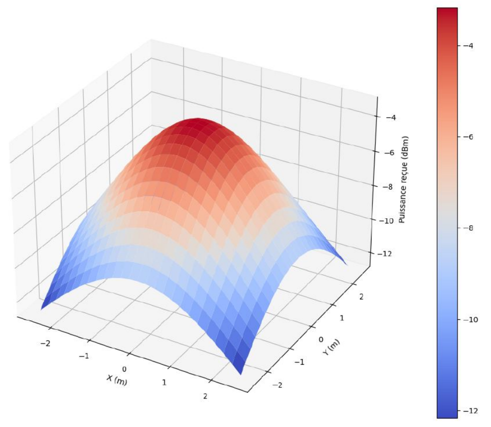

Figure 20. Optical power distribution of one central emitter using the previous simulator.

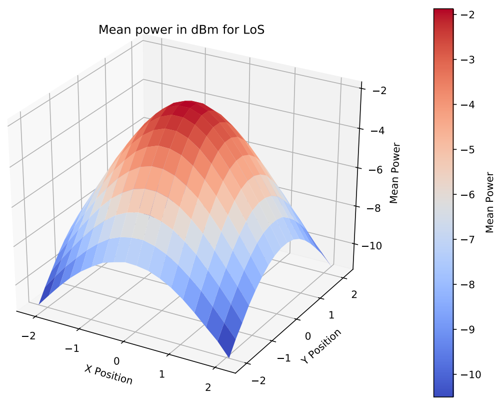

Figure 21. Optical power distribution of one central emitter using the new simulator.

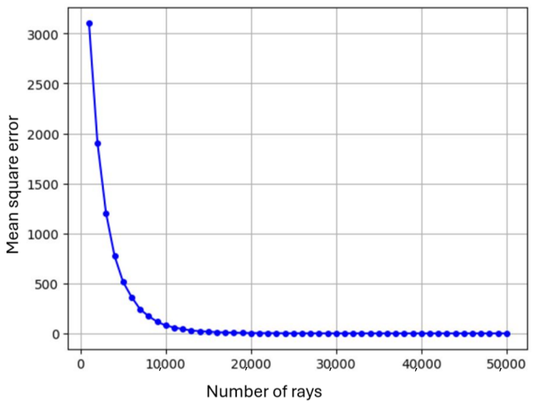

Figure 22. Evolution of the Mean Square Error for 1000 to 50,000 rays by step of 1000 rays—computation on 100 runs.

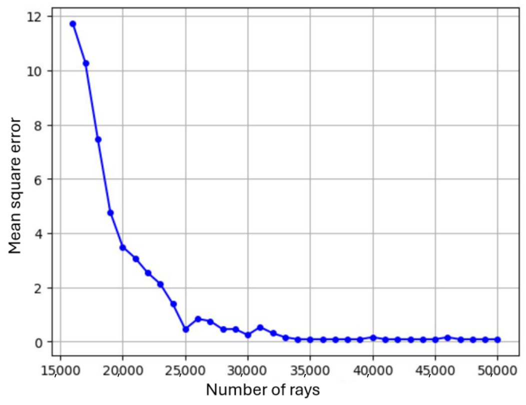

Figure 23. Evolution of the Mean Square Error for 1600 to 50,000 rays by step of 1000 rays—computation on 100 runs.

3.1.2. Comparison with Four Emitters in a LoS Configuration

Another result that is commonly presented in the literature is the case of the same room with four emitters evenly distributed across the ceiling. The positions in meters are given in Table 3. Figure 24 shows the simulation results using the code from [9]. Figure 25 shows the results of the simulator developed in this work with 1,000,000 rays per emitter (i.e., 4,000,000 rays in total). It can be seen that the results were again very similar.

Table 3. Positions of the four transmitters on the ceiling.

Position LED 1	**Position LED 2**
(1.25; 1.25; 3)	(−1.25; 1.25; 3)
Position LED 3	**Position LED 4**
(1.25; −1.25; 3)	(−1.25; −1.25; 3)

The MSE calculated for this simulation was 3.568. Note that this value was higher than for a single transmitter, but was still relatively low. As before, it is interesting to examine the evolution of the error as a function of the number of rays. To generate this figure, the number of rays was adjusted from 1000 to 50,000 in increments of 1000. Given that there were four emitters, this represented a total number of rays ranging from from 4000 to 200,000, as illustrated in Figure 26. The error decreased as the number of rays increased. Figure 27 also shows that from 200,000 rays onward, the quality of the simulation no longer improved significantly, the error value already being 3.568 with 4,000,000 rays. Whether using one emitter or four, it can be seen that the statistical ray tracing method achieved similar results to those obtained using an iterative method (previous simulator). It is important to note that the use of the term 'error' in this context can be a source of confusion. Indeed, the difference between the results of our simulator and those obtained with the reference simulators is what is being measured here. However, this measure of difference does not necessarily reflect an 'error' in the strict sense of the term, but rather a difference that could be due to various factors. It is also possible that the simulator results, even if they show deviations from the references, are closer to physical reality.

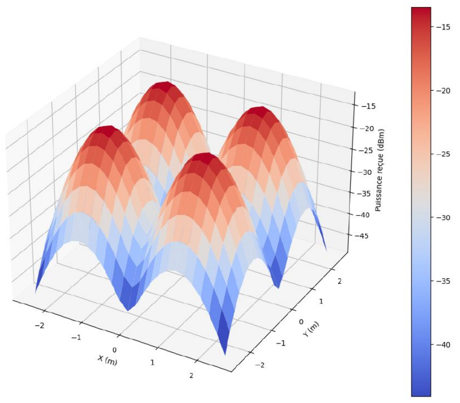

Figure 24. Optical power distribution of four emitters using the previous simulator.

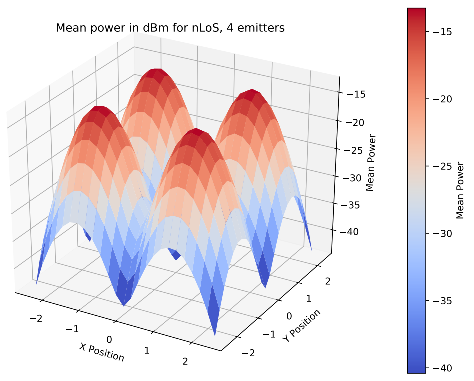

Figure 25. Optical power distribution of four emitters using the new simulator.

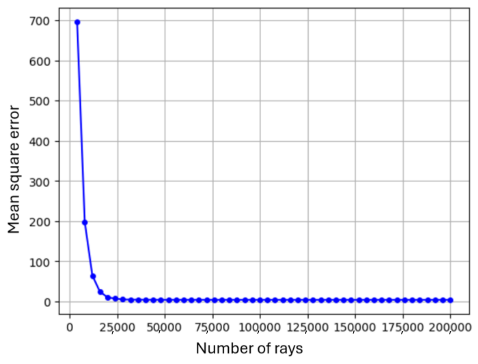

Figure 26. Mean Square Error for 4000 to 200,000 for four emitters.

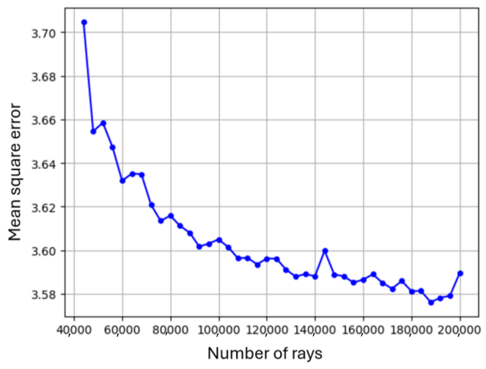

Figure 27. Mean Square Error for 40,000 to 200,000 for four emitters.

3.1.3. Comparison of the NLoS Contributions for One Emitter

This study used the same parameters as those presented in Table 2. This simulation was carried out with 1,000,000 rays sent by the emitter and 100 rays emitted for each reflection on a wall. The contribution of reflections to received power was relatively small compared to LoS power. To better illustrate this contribution, Figures 28 and 29 only show the received power due to reflections. Although the general shape of the distributions was comparable, minor differences were observed closer to the walls. Our hypothesis is that this difference was due to the more accurate results offered by the ray tracing algorithm for the behaviour of light during reflections, whereas the geometric approach used in our old simulator simplified the interactions, and even predicted them, and therefore gave a smoother but less accurate distribution result. However, it can be seen that the general trend was really close. The power received was greater in the middle of the walls and less in the center of the room in both cases. The MSE calculated for this simulation was 0.17.

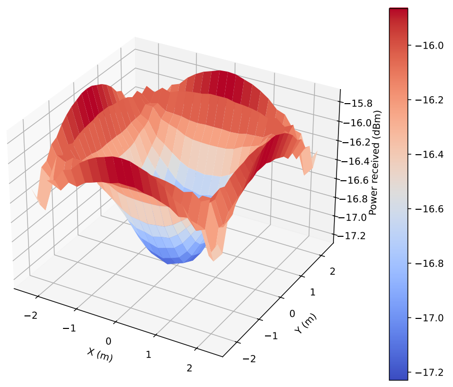

Figure 28. Optical power distribution of first order reflections for one emitter using the previous simulator.

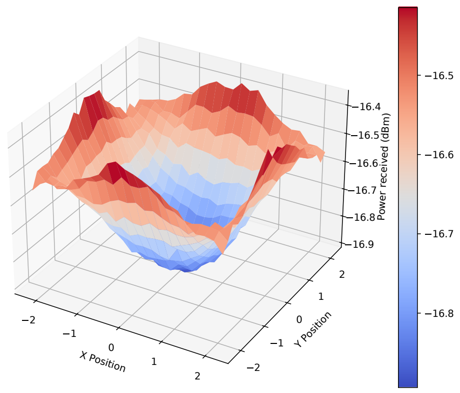

Figure 29. Optical power distribution of first order reflections for one emitter using this simulator.

3.1.4. Comparison of the NLoS Contributions for Four Emitters

This simulation was carried out with 1,000,000 rays sent for each emitter and 100 rays for the reflections. The situation was the same as described in Section 3.1.2 and Table 3. The results were very close to those obtained from the references. Looking at the received power from reflections alone, Figures 30 and 31 show that the general shape and values were similar. Once again, the reflected rays were more powerful at the edges of the walls than in the center of the room. The mean square error calculated for this simulation was 3.84, only similar to that obtained for LoS in the case of four transmitters.

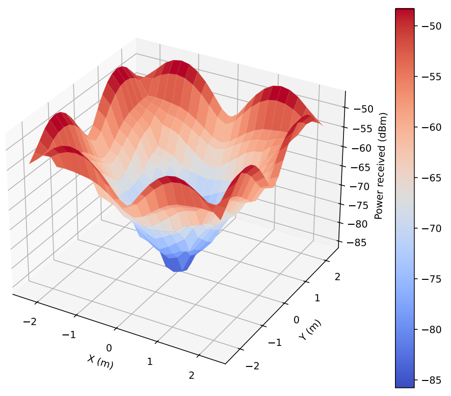

Figure 30. Optical power distribution of first order reflections for four emitters using the previous simulator.

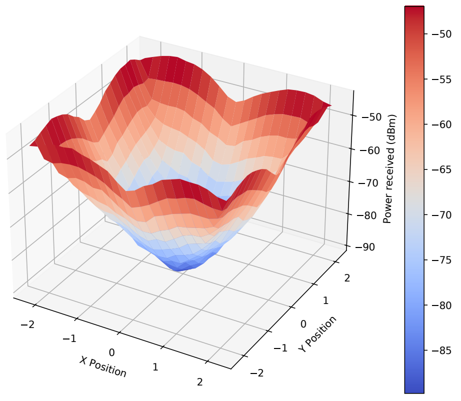

Figure 31. Optical power distribution of first order reflections for four emitters using the new simulator.

3.2. *Propagation in Complex Geometry Rooms*

Having been validated, the key to this new simulator, and what sets it apart from anything else available free and open-source, is that it is capable of handling the propagation of one or more LED emitters in rooms with a complex geometry. To demonstrate this, an octagonal room and an L-shaped (corner) room were studied. Both rooms were created using Blender and exported into a STL file. They were then imported into the simulator. All simulation presented in the following sections used 1,000,000 rays for the emitter and 100 rays per reflection point.

3.2.1. Octagonal-Shaped Room

The simulation parameters used in this scenario can be found in Table 4 and the geometry of the room can be found in Figure 32. The width of each wall was one meter, as was the height of the room. Note that the user can select larger rooms, in the present case, its dimensions were taken only as an example.

The results of the simulation with the octagonal room are shown in Figures 33 and 34. Both figures show graphs of the received power, with a top view and a side view. It can be seen that the simulator correctly took into account the geometry of the octagonal room. The rays remained confined within the geometry, which indicates that the propagation of the rays was correctly modeled, with no leakage.

Table 4. Parameters of the octagonal room.

	Parameter	Value
Environment	Room type	Octagonal
	Wall width	1 m
	Wall height	1 m
	Reflection coefficient	80%
Emitter	Position	(0; 0; 1)
	Orientation	Facing down
	Half Power Angle	12.5 degrees
	Power	2 W
Receiver	Position	The ground
	Active area	16 mm^2
	Field of View	60 degrees
	Refractive index	1.5

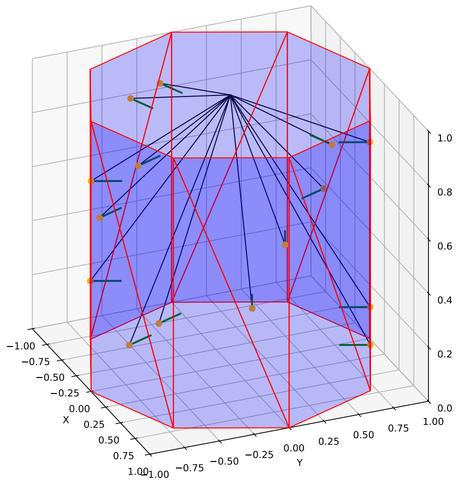

Figure 32. Illustration of the octagonal room.

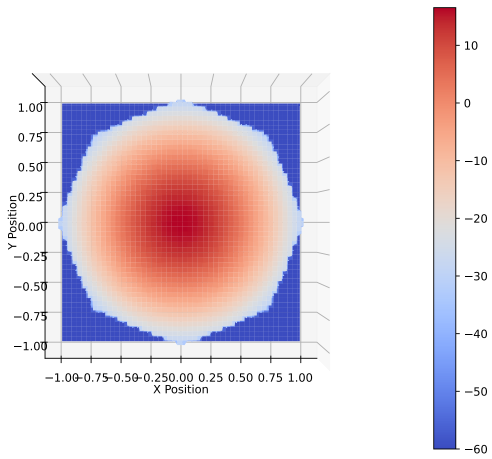

Figure 33. Top view of the optical power distribution of an LED in an octagonal room.

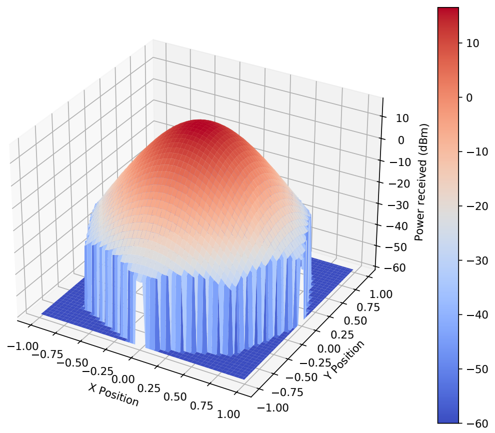

Figure 34. Side view of the optical power distribution of an LED in an octagonal room.

3.2.2. L-Shaped Room

An L-shaped room, as shown in Figure 35, was also simulated. This configuration was used to verify how the simulator handled angles and reflections in confined spaces. The parameters of this simulation are shown in Table 5. The room dimensions were two meters for the larger outer walls and one meter for the inner walls, and the emitter was decentralized in one of the corners of the room.

Table 5. Parameters of the corner room.

	Parameter	Value
Environment	Room type	corner room
	Inner walls	1 m
	Outer walls	2 m
	Wall height	1 m
	Reflection coefficient	80%
Emitter	Position	(−0.5; −0.5; 1)
	Orientation	Facing down
	Half Power Angle	40 degrees
	Power	2 W
Receiver	Position	The ground
	Active area	16 mm²
	Field of View	90 degrees
	Refractive index	1.5

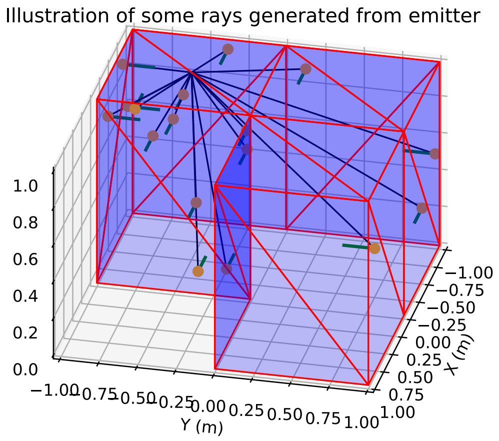

Figure 35. Illustration of the L-shaped room.

Figure 36 shows the results of the simulation in the room with a corner, without taking the reflections into account. It can be seen that the light only propagated in the areas accessible in a direct line-of-sight from the transmitter, while the area on the right remained in shadow. Figure 37, on the other hand, shows the simulation taking reflections into account. It can be seen that the light now reached the right-hand side thanks to the reflections. This shows that the reflections were correctly integrated and that the simulator was capable of modelling light scattering.

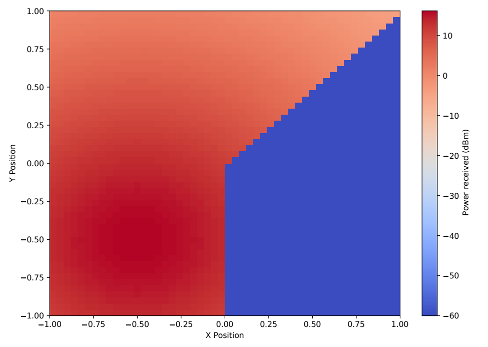

Figure 36. Optical power distribution in a corner room without reflections.

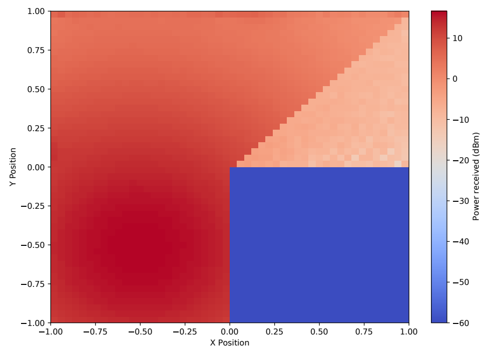

Figure 37. Optical power distribution in a corner room including reflections.

3.3. Computational Considerations

The computation time required to run the simulations using the simulator was examined. Particular attention was given to how the execution time varied with the number of rays processed per simulation. To assess this performance, the time taken to perform simulations with different numbers of rays, ranging from 1 to 300,000 in increments of 10,000 rays, was measured. This relationship is illustrated in Figure 38. For each ray count value, 10 distinct simulations were conducted, and the average execution time of these simulations was calculated. The execution time was measured using Python's `time` library. Only the time spent calling the `do_simulation` function was taken into account, while the time needed to initialize the objects used by the simulator was excluded, to only concentrate on the simulation part.

Figure 38 shows that the execution time seemed to increase linearly with the number of rays. This trend was to be expected, as the rays were processed independently of each other, implying that the computation time is proportional to the total number of rays to be simulated. Furthermore, for this number of rays, the simulation time was very short compared with the human reference point.

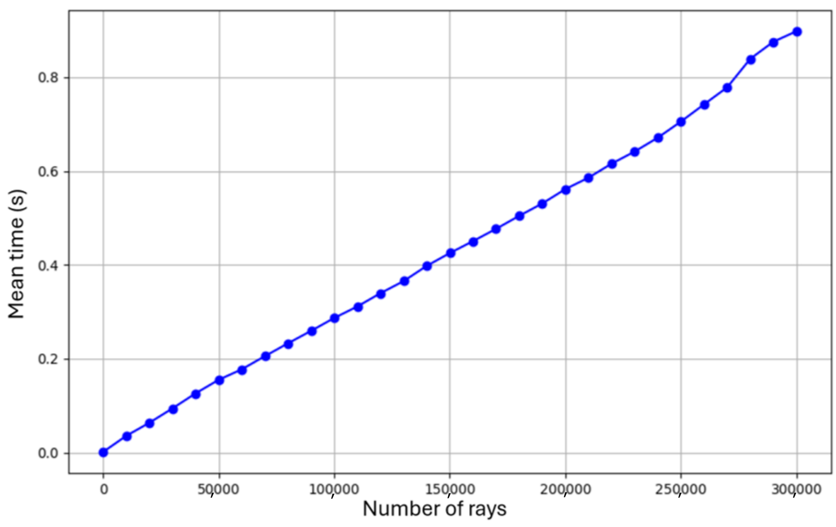

Figure 38. Evolution of simulation time depending on number of rays to be generated.

4. Conclusions

This simulator represents a noteworthy advancement in open-source VLC channel modelling, offering enhancements over existing models in the literature. To the best of our knowledge, no other open-source and free of charge ray-tracing simulator is available that can handle complex 3D room environments with a similar capability. Developed using Python and accessible via GitHub's free version [18], this work introduces several key improvements over prior implementations. Notably, it supports the importing of intricate 3D room models and employs a ray-tracing algorithm to estimate light propagation and optical communication coverage with increased precision. The simulator provides flexibility in the emitter positioning, quantity, and orientation. The use of batches for the generated and reflected rays to diminish the simulation time is a key novelty unique to this paper.

The paper details the simulator's architecture, and describes the critical algorithms utilized. The simulator's outputs were validated against the current literature, demonstrating its reliability. It also makes it easier for the user to define multiple emitters and enhances the analysis of reflected rays, which is crucial for VLC system design. Unlike previous models that primarily focused on rectangular parallelepiped rooms, this simulator has expanded capabilities to include any room geometry, as demonstrated with the L-shaped and octagonal-shaped room examples, showcasing its versatility in handling various room geometries.

While future research may explore the integration of real radiation patterns for different LED types and advanced reflection models, the current version of the simulator already represents a significant step forward in simulating complex optical environments, demonstrating its ability to model complex room shapes and accurately analyze light propagation and coverage in the field of VLC modelling.

Author Contributions: Conceptualization, V.M. and B.Q.; methodology, V.G., V.M. and B.Q.; software, N.V.; validation, V.G. and N.V.; formal analysis, V.G., N.V., V.M. and B.Q.; investigation, N.V.; resources, V.G. and N.V.; data curation, N.V.; writing—original draft preparation, V.G. and N.V.; writing—review and editing, V.G. and V.M.; visualization, V.G. and N.V.; supervision, V.G., V.M. and B.Q.; project administration, V.M. and B.Q.; funding acquisition, V.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially funded by the BAEF (Belgian American Educational Foundation) and the Rotary Club of Mons (BE).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original data presented in the study are openly available in GitHub at https://doi.org/10.5281/zenodo.13516110.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analysis, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

FoV	Field of View
FSO	Free Space Optics
LED	Light Emitting Diode
Li-Fi	Light Fidelity
LoS	Line-of-Sight
MSE	Mean Square Error
nLoS	non-Line-of-Sight
OBJ	Object file
OCC	Optical Camera Communication
OWC	Optical Wireless Communication
STL	Stereolithography
VLC	Visible Light Communication

Appendix A

To create a cube with inward-facing normals in Blender and export this model as an STL file, follow these steps:

1. Create a cube Open Blender :
    - Launch Blender. By default, a scene with a cube is usually already present.
    - Select the cube: Click on the cube to select it, if it is already present in the scene. Otherwise, add a cube by going to Add > Mesh > Cube.
2. Invert the Normals
    - Switch to Edit Mode: Press Tab to switch to Edit Mode. Select All Sides: Press A to select all sides of the cube.
    - Flip Normals: Press Alt + N to open the normals menu, then select Flip Normals.
3. Check and clean Normals
    - Check Normals: Switch to front view of normals by pressing Alt + N and select Show Normals > Visible. This will allow you to see if the normals are correctly oriented inwards.
    - Manual Correction: If some of the normals are incorrect or you want better control, you can manually select the faces and press Alt + N > Reverse Normals to correct them.
4. Export to STL
    - Return to Object Mode: Press Tab to return to Object Mode.
    - Export the Model: Go to File > Export > STL.
    - Configure Settings: Choose where you want to save the file, give it a name, and make sure the export options are set to your requirements.
    - Save the File: Click on Export STL to save the file.

References

1. Miramirkhani, F.; Uysal, M. Channel modelling for indoor visible light communications. *Philos. Trans. R. Soc. A* **2020**, *378*, 20190187. [CrossRef] [PubMed]
2. Ghassemlooy, Z.; Popoola, W.; Rajbhandari, S. *Optical Wireless Communications: System and Channel Modelling with Matlab*$^{®}$; CRC Press: Boca Raton, FL, USA, 2019.

3. Al-Kinani, A.; Wang, C.X.; Zhou, L.; Zhang, W. Optical wireless communication channel measurements and models. *IEEE Commun. Surv. Tutor.* **2018**, *20*, 1939–1962. [CrossRef]
4. Tang, P.; Yin, Y.; Tong, Y.; Liu, S.; Li, L.; Jiang, T.; Wang, Q.; Chen, M. Channel Characterization and modelling for VLC-IoE Applications in 6G: A Survey. *IEEE Internet Things J.* **2024**, *11*, 34872–34895. [CrossRef]
5. Yahia, S.; Meraihi, Y.; Ramdane-Cherif, A.; Gabis, A.B.; Acheli, D.; Guan, H. A survey of channel modelling techniques for visible light communications. *J. Netw. Comput. Appl.* **2021**, *194*, 103206. [CrossRef]
6. Turan, B.; Coleri, S. Machine learning based channel modelling for vehicular visible light communication. *IEEE Trans. Veh. Technol.* **2021**, *70*, 9659–9672. [CrossRef]
7. Li, Z.; Shi, J.; Zhao, Y.; Li, G.; Chen, J.; Zhang, J.; Chi, N. Deep learning based end-to-end visible light communication with an in-band channel modelling strategy. *Opt. Express* **2022**, *30*, 28905–28921. [CrossRef] [PubMed]
8. Bhalerao, M.V.; Sumathi, M.; Sonavane, S. Line of sight model for visible light communication using Lambertian radiation pattern of LED. *Int. J. Commun. Syst.* **2017**, *30*, e3250. [CrossRef]
9. Georlette, V. *veroniquegeorlette/VLC_channel_modelling_python: Opensource VLC Channel Simulator in Python*, Version v1.0 ; Zenodo: Geneva, Switzerland, 2023. [CrossRef]
10. Georlette, V.; Bette, S.; Brohez, S.; Pérez-Jiménez, R.; Point, N.; Moeyaert, V. Outdoor visible light communication channel modelling under smoke conditions and analogy with fog conditions. *Optics* **2020**, *1*, 259–281. [CrossRef]
11. Georlette, V.; Melgarejo, J.S.; Bette, S.; Point, N.; Moeyaert, V. Potential and challenges of visible light communication for industrial assembly lines with mobile workstations. In Proceedings of the 2021 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT), Bandung, Indonesia, 27–28 July 2021 ; IEEE: Piscataway, NJ, USA, 2021; pp. 228–234.
12. Georlette, V.; Honfoga, A.C.; Dossou, M.; Moeyaert, V. Exploring Universal Filtered Multi Carrier Waveform for Last Meter Connectivity in 6G: A Street-Lighting-Driven Approach with Enhanced Simulator for IoT Application Dimensioning. *Future Internet* **2024**, *16*, 112. [CrossRef]
13. Lee, J.H.; Ko, M.S.S. Monte Carlo Simulation for Indoor Optical Wireless Communications. *IEEE Trans. Commun.* **2006**, *54*, 1862–1872.
14. Rajagopalan, R.; Tsoi, S.C.L. Monte Carlo Based Channel modelling for VLC Systems. *J. Opt. Commun. Netw.* **2010**, *2*, 302–311.
15. Chaudhry, A.; Sadiq, M.Z.K.B. Monte Carlo Simulation for Visible Light Communication in Indoor Environments. *IEEE Access* **2015**, *3*, 2365–2376.
16. Li, X.; Xu, Y. Monte Carlo Simulation for Light Propagation in Visible Light Communication Systems. *Appl. Opt.* **2017**, *56*, 6276–6283.
17. Zhang, L.; Wang, Y.; Zhao, H. Monte Carlo Simulation for Channel Estimation in Visible Light Communication Systems with Dynamic Lighting Conditions. *IEEE Trans. Commun.* **2023**, *71*, 1245–1255.
18. Vallois, N. *MokonaNico/vlc-simulation-raytracing: v1.0.0—VLC Simulation Based on Raytracing*, Version v1.0.0 ; Zenodo: Geneva, Switzerland, 2024. [CrossRef]
19. Blender. BLENDER Downloads. 2024. Available online: https://www.blender.org/download/ (accessed on 24 August 2024).
20. Autodesk. AutoCAD Downloads. 2024. Available online: https://www.autodesk.com/products/autocad/free-trial (accessed on 24 August 2024).
21. Miramirkhani, F.; Uysal, M. Channel modelling and characterization for visible light communications. *IEEE Photonics J.* **2015**, *7*, 1–16. [CrossRef]
22. ZEMAX LLC. ZEMAX. 2023. Available online: https://www.zemax.com/ (accessed on 24 August 2024).
23. Mathworks. MATLAB. 2024. Available online: https://www.mathworks.com (accessed on 24 August 2024).
24. Sarbazi, E.; Uysal, M.; Abdallah, M.; Qaraqe, K. Indoor channel modelling and characterization for visible light communications. In Proceedings of the 2014 16th International Conference on Transparent Optical Networks (ICTON), Graz, Austria, 6–10 July 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 1–4.
25. Komine, T.; Nakagawa, M. Fundamental analysis for visible-light communication system using LED lights. *IEEE Trans. Consum. Electron.* **2004**, *50*, 100–107. [CrossRef]
26. Muller, M.E. A note on a method for generating points uniformly on n-dimensional spheres. *Commun. ACM* **1959**, *2*, 19–20. [CrossRef]
27. Hughes, J.F. *Computer Graphics: Principles and Practice*; Pearson Education: London, UK, 2014.
28. Möller, T.; Trumbore, B. Fast, minimum storage ray/triangle intersection. In *ACM SIGGRAPH 2005 Courses*; Association for Computing Machinery: New York, NY, USA, 2005; pp. 7–es.