On building a Python-based Monte Carlo light simulation package for biophotonics

with focus on complex 3-dimensional arbitrary geometries and hardware acceleration.

Mémoire

Marc-André Vigneault

Sous la direction de:

Daniel Côté, directeur de recherche

Résumé

L'essor du Python comme language scientifique et la popularité grandissante du calcul GPGPU constitue un environnement fertile pour le développement de PyTissueOptics, un simulateur de propagation de la lumière dans les tissus, qui est un module entièrement programmé en Python, qui se concentre sur la mise en oeuvre simple et compréhensible des mécanismes de propagation de la lumière, afin de catalyser la démocratisation de cette technique. Dans ce mémoire, nous explorons la théorie derrière le modèle de Monte Carlo, visitons les aspects techniques de la réalisation du projet, expliquons les différentes possibilités d'utilisation et comparons avec des modules connus, tel que MCML et MCX.

Abstract

The rise of Python as a scientific language and the growing popularity of GPGPU computing creates a fertile environment for the development of PyTissueOptics, a light propagation simulator in tissues, which is a module entirely programmed in Python. It focuses on the simple and comprehensible implementation of light propagation mechanisms, aiming to catalyze the democratization of this technique. In this thesis, we explore the theory behind the Monte Carlo model, examine the technical aspects of the project's realization, explain the various usage possibilities, and compare it with known modules, such as MCML and MCX.

Contents

List of Tables

List of Figures

To love and passion.

Practice without theory is blind,
but theory without practice is
just a mere intellectual play.

Immanuel Kant

Acknowledgments

First, I'd like to thank Prof. Daniel Côté as a mentor for his guidance, his listening, his interest and advice-giving, and most especially as a friend, for his understanding, his openness, and his high energy motivation. Second, I'd like to thank Mireille Quemener for her help, her advice, and her extremely pleasant energy. Working with you both has been a charm.

Special thanks to Ludovick Bégin, which has been *the best* coding partner. Working with you made all the difference, it was truly a unique experience.

Thanks to everyone in the laboratory for their high spirits, it was very pleasant working with all of you.

Thanks to my family, mother, father, and brothers for their motivation and support.

Thanks to Françoise Meunier for her incredible support through the dark times of this journey.

Thanks to my close friends for their time and their discussions. I could not have had better encounters, which partially shaped who I am today.

Introduction

This master's thesis introduces the PyTissueOptics[1] Python module, a Monte Carlo-based light propagation simulator written in Python. This work is part of a master's research in biophotonics at the CERVO Brain Research Center, led by Prof. Daniel Côté. The chapter outlines the current issues with Monte Carlo simulations and explains the reasons for this project.

Context of the project

Prof. Daniel Côté introduced the Python module "PyTissueOptics" as an instrument to facilitate Monte Carlo light propagation simulations within tissues. The extensive objective was to enhance the accessibility of Monte Carlo methods by presenting a Python-based simulator that is both comprehensive and modifiable. During optics courses at Université Laval, the module was presented to students as a pedagogical instrument for Monte Carlo simulations and light propagation physics. PyTissueOptics was built upon a C implementation known as Monte Carlo Multi-Layer (MCML) [2], which has remained a benchmark in the domain of Monte Carlo light simulations in tissue for several decades. However, with the rise of Python as a preferred language for scientific endeavors, Prof. Côté initiated the development of PyTissueOptics. Though the initial rendition was somewhat rudimentary, it was apt for answering elementary questions in a fairly efficient timeframe, despite being approximately a 1000 times slower than MCML.

Another module which is in the public domain [3] is the Monte Carlo eXtreme (MCX) [4], originally written in PASCAL. While MCX boasts substantial computational power, its intricate usability and reliance on the less-common PASCAL language present barriers for potential users. As of this date, MCX offers its module in various forms; OpenCL [5] and Matlab compatible modules, cloud-computing architecture [6] and even MCXStudio, a software centralizing all their algorithms. In March 2023, MCX launched the first version of their python module pmcx [7], which is a python wrapper for MCX, thus having the same capabilities, and limitations.

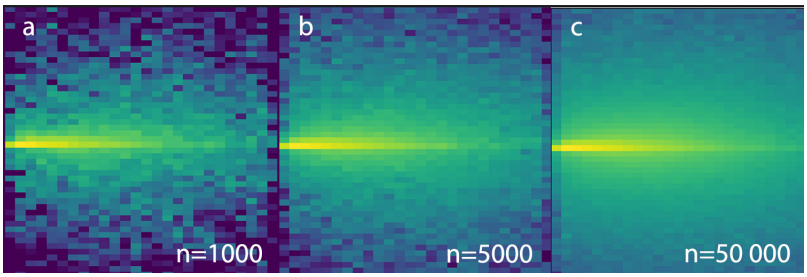


Figure 0.1: Simulation result from the initial PyTissueOptics module in a phantom tissue using a pencil point light source starting on the left end inside the tissue. The heat map represents a binned 2D projection of the average energy deposited after propagating 1000, 5000 and 50 000 photons. Simulating a greater amount of photons increases signal-to-noise ratio and yields a better approximation of the expected spatial energy distribution inside the tissue.

The ease of tool utilization is paramount, particularly in academic research. Many researchers, predominantly students, often lack the luxury of time to grapple with steep learning curves associated with intricate tools or esoteric programming languages. One might argue that while certain packages offer accelerated computational speeds, the true efficiency of a simulation must account for the time needed for tool setup and mastery. With Python emerging as the de facto programming language for STEM academia due to its intuitive nature, there emerges a pressing need for a Python-based tool that encapsulates the virtues of being open-source, user-friendly, and easily modifiable.

It was discerned that PyTissueOptics held significant potential for fostering widespread adoption of the Monte Carlo method. However, to bring its capabilities on par with robust tools like MCX, it necessitated profound enhancements. At its inception, PyTissueOptics was confined to simulating light propagation within rudimentary geometries, as depicted in figure 0.1. Its initial simple algorithm rendered complex simulations with multifaceted objects highly time-consuming. Consequently, two primary enhancements were envisaged: facilitating light propagation in intricate 3D geometries and increasing simulation speeds. These challenges could be addressed via a mesh-based strategy for the former and hardware acceleration for the latter.

Relevance of simulations in biophotonics

Why simulate? Simulations serve as invaluable tools in modern scientific investigations, obviating the need for costly and time-consuming physical experiments. They afford unparalleled visualization capabilities, elevate accuracy, and streamline experiment modifications.

In the realm of biophotonics, simulations hold profound implications. Biophotonics is an

interdisciplinary field leveraging diverse modalities of light propagation. From laser sources to microscopic imaging enabled by intricate optical arrangements, the applications of light in biophotonics are manifold.

Monte Carlo simulations have used in various biophotonic endeavors, such as the design of skin-mimetic optical devices [8], estimation of thermal effects in human tissues [9, 10], and enhancement of hyperspectral imaging systems [11]. These are mere examples, and the democratization of Monte Carlo methods could further amplify its applicability and foster novel innovations.

The problem with other simulation software

Existing software, such as Zemax and SPEOS, offer extensive capabilities for simulating light propagation in tissues. While their prowess is undeniable, they often prove sub-optimal for research contexts due to prohibitive licensing costs, steep learning curves, rigid features, and lack of open-source transparency.

To circumvent these limitations, researchers often gravitate towards open-source alternatives like ValoMC, MCX, and mcxyz. Although these offerings negate the financial constraints and offer extendability, their reliance on intricate programming languages often renders them inaccessible to researchers without extensive coding expertise.

Python as the preferred scientific language

Python stands out as a preferred programming language for scientific pursuits [12] due to its readability and extensive library ecosystem tailored for scientific computations, such as Numpy, Scipy, and Matplotlib. Bolstered by a vast community, it is poised to remain the gold standard for scientific computing. Thus, equipping researchers with Python-centric tools could potentially increase the adoption and efficacy of Monte Carlo techniques.

Objectives of this project

The inaugural rendition of PyTissueOptics lacked modularity and extendability. The primary objective was to overhaul its architecture, ensuring maintainability, readability, and resilience. Subsequent goals encompassed the incorporation of intricate geometries into the simulation paradigm, realization of a simulator grounded in the MCML model, and significant enhancements to its computational efficiency.

Thesis body

Chapter 1 focuses into the theoretical foundations, encompassing the Monte Carlo technique, the radiative transport equation, and the nuances of the MCML model. It further introduces the principles of parallelization in GPU computing and introduces basic theory on ray tracing within 3D environments.

Chapter 2 elaborates on the pragmatic endeavors to refine PyTissueOptics. It begins with a discourse on software development methodologies, followed by a dissection of the architectural challenges and their resolutions. The chapter then transitions to the implementation of 3D support, meshing algorithms, and the propagation model. The concluding segments discuss computational enhancements, focusing on ray-hit search techniques and GPU-accelerated parallelization.

Chapter 3 presents the developed 3D graphic framework and the Monte Carlo Raytracing Simulator. It then validates the simulation results and compares speed with MCML and MCX. Moreover, an array of examples demonstrate its capabilities.

The concluding chapter recapitulates the project's achievements, addresses its limitations, and envisions its future trajectories.

For an exhaustive examination of the codebase, readers are directed to the GitHub repository hosted under the DCCLAB organization: `https://github.com/DCC-Lab/PyTissueOptics`.

Chapter 1

Theoretical aspects of the Monte-Carlo method applied to 3D raytracing

1.1 The Monte Carlo light propagation model

1.1.1 Monte Carlo Technique

The Monte Carlo method is a statistical technique that is used to simulate physical, mathematical, or financial experiments. It is used for problems with various random variables which renders problems usually unsolvable by differential equations. This technique is also useful and very simple since the physics involved is replaced with random sampling.

Brief Monte Carlo History [13].

During the climactic days of World War II in 1945, the genesis of the Monte Carlo simulations coincided with the imminent completion of the world's first Electronic Numerical Integrator and Computer (ENIAC) at the University of Pennsylvania in Philadelphia. Directed by John Mauchly and Presper Eckert, ENIAC promised groundbreaking advancements in computational capabilities.

Recognizing ENIAC's potential, physicists from the Los Alamos National Laboratory (LANL), Nicholas Metropolis and Stan Frankel, embarked on formulating a thermonuclear model apt for the ENIAC. This ambitious project was initiated under the guidance of John von Neumann, a renowned Professor at the Institute for Advanced Study and a consultant for LANL.

However, the cessation of World War II marked a change in the scientific landscape. As

priorities shifted, the team welcomed back physicist Stan Ulam from the University of Southern California's Mathematics Faculty. Ulam, drawing upon his vast knowledge of statistical sampling techniques, envisioned the vast potentials the ENIAC held. His foresight led to the pioneering creation of the first particle transport model rooted in random sampling.

The trio of physicists, having a interest for probabilities and random numbers, named this method "Monte Carlo." This name was inspired by the Monte Carlo Casino, notorious for being the gambling haven of Ulams uncle.

Since its development, the Monte Carlo method has undergone substantial refinement and modification, largely driven by research at LANL. The method's adaptability and reliability have rendered it particularly suitable for detailed statistical physics simulations, with a notable emphasis on particle transport studies. Significant advancement in this method are embodied by the Monte Carlo N-Particle V6 (MCNP6). Developed by LANL, MCNP6 represents a comprehensive tool in the domain of scientific simulations. It has been applied in various fields, including radiation protection, medical physics, nuclear reactor design, and particle accelerator design [14]. A public version of MCNP6 is available for broader scientific use. https://mcnp.lanl.gov/.

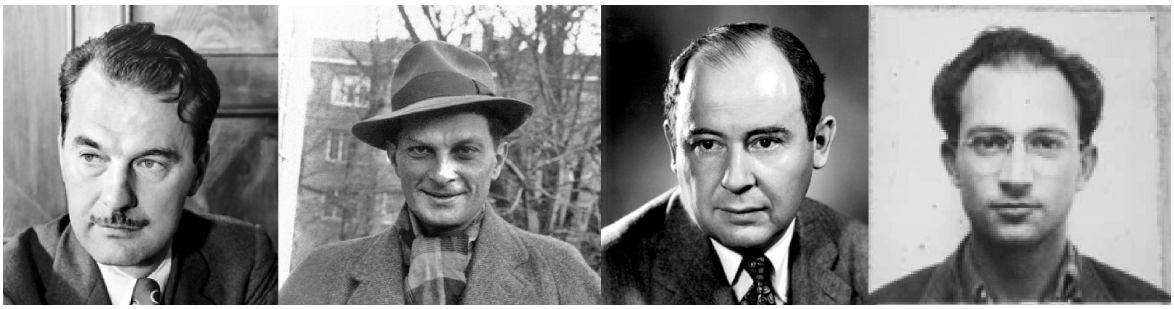


Figure 1.1: Nicholas Metropolis, Stanislaw Ulam, John Von Neumann, Stan Frankel.

Monte Carlo method for physical simulations

Physical processes, especially at the quantum scale, often exhibit probabilistic behavior. Quantum mechanics introduces the notion of probabilistic outcomes for certain measurements, offering a perspective that contrasts with deterministic classical physics. For many complex systems, particularly those involving large collections of small particles, a probabilistic or stochastic approach provides valuable insights. To simulate these probabilistic processes, Monte Carlo methods utilize pseudo-random generators to emulate real-life randomness. By leveraging an extensive sampling size, these simulations can achieve a representative average result with minimal noise in the output.

The efficacy of large sampling is underpinned by the **Central Limit Theorem** (CLT). The CLT posits that under appropriate conditions, the sum of a large number of independent

and identically distributed random variables will be approximately normally distributed. Mathematically, this can be expressed as:

$$f_N(\bar{x}) = \frac{1}{\sqrt{2\pi\sigma_{\bar{x}}^2}} e^{-\frac{(\bar{x}-m_x)^2}{2\sigma_{\bar{x}}^2}}, \quad \text{as } N \to \infty \tag{1.1}$$

One crucial implication of the CLT is that, in a random experiment, as the number of samples (N) increases, the empirical distribution becomes closer to the theoretical distribution. Furthermore, for a given distribution with variance σ_x^2, the variance of the sample mean reduces with increasing sample size as:

$$\sigma_{\bar{x}}^2 = \frac{\sigma_x^2}{N} \tag{1.2}$$

Consequently, the standard deviation of the sample mean is:

$$\sqrt{\sigma_{\bar{x}}^2} = \frac{\sigma_x}{\sqrt{N}} \tag{1.3}$$

Thus, a larger sample size leads to a reduced variance in the sample mean, implying that the sample average more closely approximates the true population average.

For a comprehensive exploration of the mathematical foundations of the Monte Carlo method, including an introduction to probability theory and a complete derivation of its mathematical principles, refer to *Monte Carlo Methods for Particle Transport* [15].

1.1.2 Light Propagation Model

Light, an electromagnetic radiation, can be modeled using various paradigms such as rays, waves, particles, and quantum fields. Each model offers different insights, often increasing in complexity with our understanding of the phenomenon. The Radiative Transfer Equation (RTE) serves as a statistical approach to model light propagation, abstracting complexities tied to electromagnetic or quantum field propagation and light-matter interactions.

Modeling Light Propagation with the RTE

One of the ways that have been used in the past to model light energy transport is using a Radiative Transport Equation (RTE) which is a variation of the Boltzmann Transport Equation (BTE). The Boltzman Transport Equation has been used classically to describe thermodynamic systems or fluid flow; multiple particles bounce on each other and transfer kinetic or thermal energy in the process. Here is the general description of the BTE:

$$\frac{df}{dt} = \left(\frac{\partial f}{\partial t}\right)_{\text{force}} + \left(\frac{\partial f}{\partial t}\right)_{\text{diff}} + \left(\frac{\partial f}{\partial t}\right)_{\text{coll}}, \tag{1.4}$$

where f is a 3-dimensional, momentum dependant and time dependant probability density $f(x, y, z, \rho_x, \rho_y, \rho_z, t)$ of a particle's position. There is a *force* term which takes into account external forces on the particle, a *diffusion* term, which take into account deflections of the particles and a *collision* term which takes into account elastic collisions between particles. This equation has been generalized and is used in various fields, such as general relativity and physical cosmology.

Another variation of this equation, the RTE is used to model the propagation of electromagnetic energy. However, even though it is possible to model light as discrete packets of energy (photons), the RTE doesn't take into account the quantum nature of light and instead acts more like a completely ballistical propagation of light, which is useful but has its limitations. For a long time, it was not clear whether the RTE solutions could give similar results to Maxwell's Equations, but this was mathematically proven that it is not the case [16]. RTE cannot describe interference phenomena and cannot describes situations where two events of scattering affect each other. However, on a macroscopic level, the radiance distribution given by the RTE solutions are valid.

$$\underbrace{(\hat{\mathbf{s}} \cdot \nabla) L(\mathbf{r}, \hat{\mathbf{s}})}_{\substack{\text{Variation of intenity} \\ \text{in path's direction}}} = \underbrace{-\mu_a L(\mathbf{r}, \hat{\mathbf{s}})}_{\substack{\text{Loss by} \\ \text{absorption}}} \underbrace{-\mu_s L(\mathbf{r}, \hat{\mathbf{s}})}_{\substack{\text{Loss by} \\ \text{scattering}}} + \underbrace{\mu_s \int_{4\pi} p\left(\hat{\mathbf{s}}, \hat{\mathbf{s}}'\right) L(\mathbf{r}, \hat{\mathbf{s}}) d\omega'}_{\text{Gain by scattering}} \tag{1.5}$$

Equation 1.5 is a simplification of the complete radiative transfer equation where we neglected the emission from the volume. This simplification was done because for biological tissue, which is mostly made of water, the emissivity is really close to 0, thus the contribution of the emission is negligible.

If we only look at absorption, we get an equation of the form $f' = -a \cdot f$, for which we know the solution is a negative exponential. This results in Equation 1.7, which is Beer-Lambert's law.

$$\int_0^s (\hat{\mathbf{s}} \cdot \nabla) L(\mathbf{r}, \hat{\mathbf{s}}) = \int_0^s -\mu_a L(\mathbf{r}, \hat{\mathbf{s}}) \tag{1.6}$$

$$L(\mathbf{r}, \hat{\mathbf{s}}) = L_0 \cdot e^{-\mu_a \cdot s} \tag{1.7}$$

Thus, from absorption and from scattering, there is a negative exponential variation of the light intensity. For the gain by scattering, which is light coming from volume elements at $\vec{\mathbf{s}}$ into the volume element at $\vec{\mathbf{s}}'$, we are interested in $p\left(\hat{\mathbf{s}}, \hat{\mathbf{s}}'\right)$, which is called the phase scattering

function. This function is dependent on the shape of the scatterer, the position of the incoming light, the scattered light, and sometimes the polarization, and will dictate the directionality of the scattered light.

$$\int_{4\pi} p\left(\hat{\mathbf{s}}, \hat{\mathbf{s}}'\right) dw' = 1 \tag{1.8}$$

Moreover, it is usually correct to assume this function is dependent on the angle between the incident light and the scattered light rather than the whole vectors. This helps to simplify the phase function dependency, as shown in Equation 1.9.

$$p\left(\hat{\mathbf{s}}, \hat{\mathbf{s}}'\right) = p(\cos(\theta)) \tag{1.9}$$

For isotropic scattering, the phase function would simply be $p(\cos(\theta)) = \frac{1}{4\pi}$, but since most material are not isotropic, we will use more complex function to describe scattering.

Another assumption we can make is that the shape of the scatterers are spheres. Since it is not possible to predict the shape of all the different scatterers, this seems like a reasonable assumption. Scattering by spherical objects is often called Mie scattering. However, these scattering functions are still quite complex. A good approximation of these functions is called the modified Henyey-Greenstein function, shown at Equation 1.10, and was found to correctly model light scattering in biological tissues by Steven Jacques [17].

$$p_{\mathrm{m-HG}}(\cos\theta) = \frac{1}{4\pi}\left[\beta + (1-\beta)\frac{1-g_{\mathrm{HG}}^2}{\left(1+g_{\mathrm{HG}}^2 - 2g_{\mathrm{HG}}\cos\theta\right)^{3/2}}\right] \tag{1.10}$$

However, as demonstrated by D. Toublanc [18], the usage of the modified Henyey-Greenstein function over the simpler Henyey-Greenstein has no real interest. Thus it is possible to remove β. It is also easier to characterize materials because the function now only depends on one parameter g, which is the anisotropy parameter and represents the mean cosine of the phase function over all scattering directions. The simplified phase function is as follow:

$$\mathrm{p}(\cos\theta) = \frac{1}{2}\frac{1-\mathrm{g}^2}{\left(1+\mathrm{g}^2 - 2\ \mathrm{g}\cos\theta\right)^{3/2}} \tag{1.11}$$

The influence of g on the phase function can be seen in Figure 1.2.

To summarize, the propagation of light in non-emissive scattering media can be modeled with the RTE. Using approximations, it is possible to make a simple model of light propagation where the absorption of light and the scattering events follow a negative exponential with

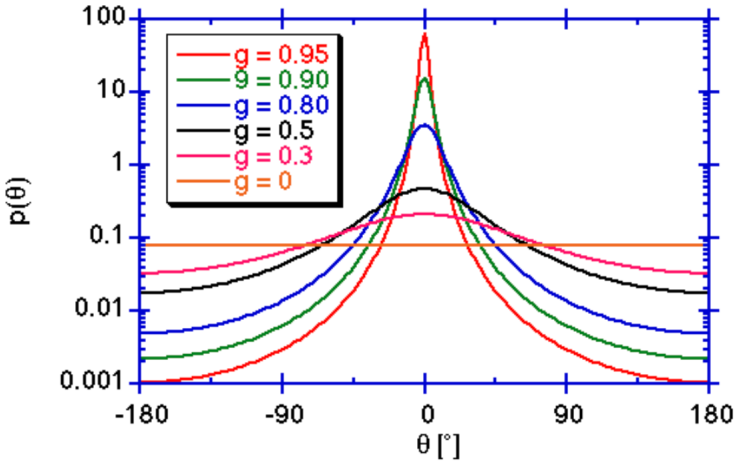


Figure 1.2: The Henyey-Greenstein function which is used to mimic the angular dependence of light scattering by small particles. An anisotropy factor of $g = 0$ leads to a completely isotropic scattering. $g = 1$ will lead to no change in direction [19].

source distance, and the scattering direction is approximated by the Henyey-Greenstein function, where the parameter g denotes the anisotropy of scattering of the material.

Finally, solving this equation is possible, but extremely tedious. The analytical solutions are only known for very simple situations, such as rotationally symmetrical geometries [20]. This is not practical because of the complexity of the equation and because it cannot be used to simulate arbitrary complex 3D geometries with varying materials. Thus, like most statistical physics approaches, it is possible to estimate the solution using a numerical method.

The light propagation model detailed for implementation

The light propagation model used in this work was first presented by S.A. Prahl in his Ph.D. thesis (1988)[21], then further detailed in a comprehensive paper in 1989[22], and later implemented in standard C as Monte Carlo Multi-Layered (MCML) in 1992[2].

The necessary formulas for the implementation are presented below, all of which originate from the work of S.A. Prahl [22]. Figure 1.3 depicts the flow of a photon in a simulation using the Monte Carlo method. The flowchart represents photon physics in the MCML code context, highlighting how photon propagation is modeled using computer code. The actual model used is elaborated upon in Chapter 2.

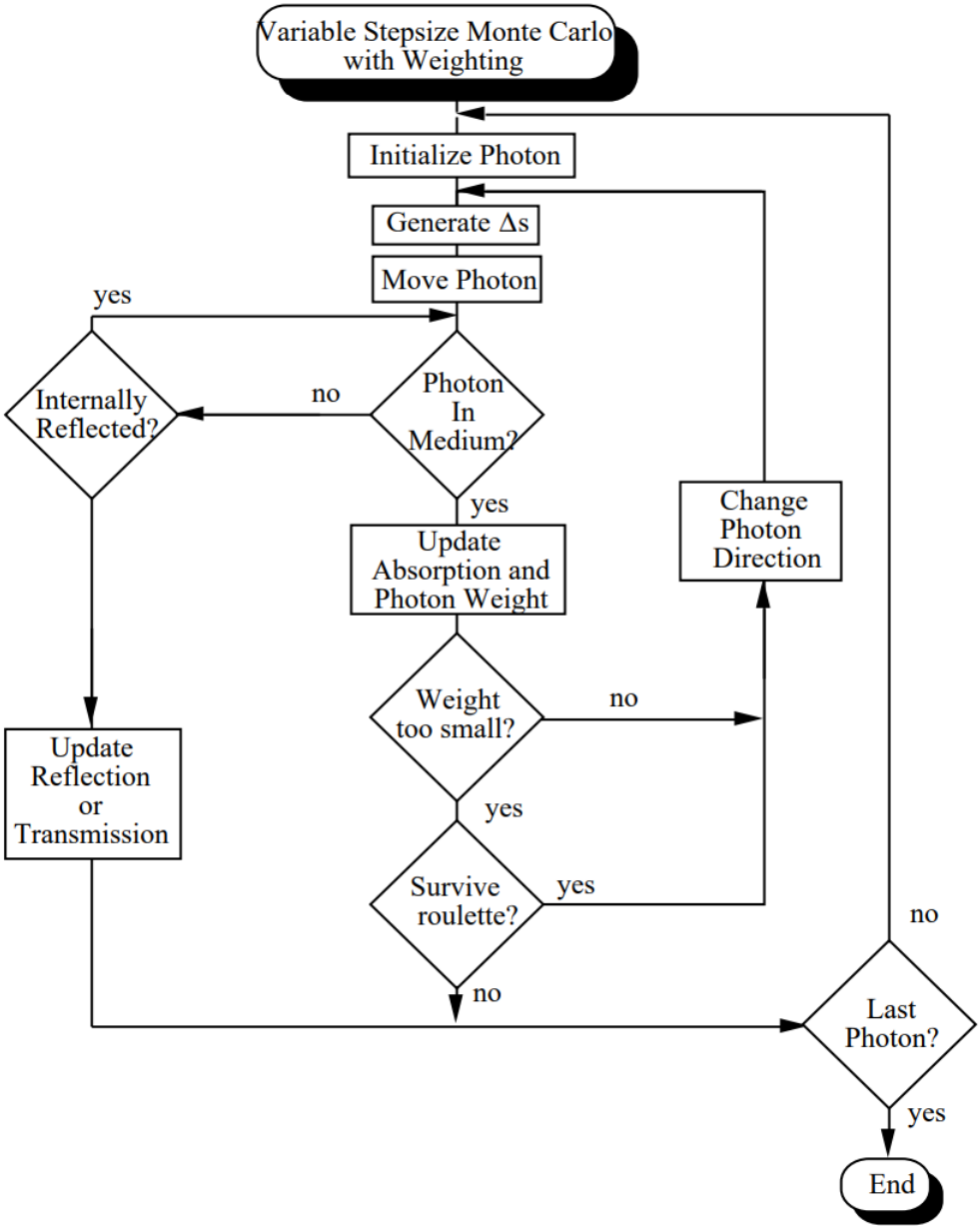


Figure 1.3: Flowchart for the MCML variable stepsize Monte Carlo technique. The photon packet is initialized. The distance to the first interaction event is found and the photon packet is moved. If the photon has left the tissue, the possibility of internal reflection is checked. If the photon is internally reflected then the photon position is adjusted accordingly and the program continues, otherwise, the photon escapes, and the event is recorded. For photons that continue, some fraction of the photon packet (1-a)w will be absorbed each step. This fraction is recorded and the photon weight is adjusted. If the weight is above a minimum, then the rest of the photon packet is scattered in a new direction and the process is repeated. If the weight falls below a minimum, then roulette is played to either extinguish or continue propagating the photon. If the photon does not survive the roulette, a new photon packet is started.

MCML Propagation Algorithm Steps

1. Initialize a `Photon` with a position, direction, and weight.
2. Generate a distance Δs.
3. Move `Photon` to a new position.
4. Reflect or Refract the `Photon` if it changes medium.
5. Deposit some of the `Photon` weight.
6. If the weight of the `Photon` becomes insignificant, it goes through a survival roulette.
7. Change the `Photon` direction.
8. Repeat until all the `Photon` weights are 0.

Propagation in free medium

Knowing that the probability density of the absorption and scattering follows Beer's law, from a uniformly distributed random variable ξ, the distance Δs is

$$\Delta s = \frac{-\ln \xi}{\mu_t}, \tag{1.12}$$

μ_t representing the absorption and scattering coefficient $\mu_a + \mu_s$, or the total extinction coefficient.

Because this event is either an absorption event or a scattering event, we will make use of this and reduce the weight by a factor, but still scatter the photon and propagate the photon with the remaining weight. The factor by which we reduce the weight is simply the portion of μ_a on μ_t, or rather the influence that μ_a had on generating a scattering event.

$$\frac{\mu_a}{\mu_a + \mu_s} = 1 - \frac{\mu_s}{\mu_a + \mu_s} = 1 - a \tag{1.13}$$

a is called the albedo and is the fraction of scattered light on the total extinction coefficient. The new weight of the photon after an event just has to be multiplied by the albedo a. The weight loss represents the energy deposited by the photon in the medium.

$$w' = w - w(1 - a) = w - w + wa = wa \tag{1.14}$$

Thus, the value to be logged is $w(1 - a)$ and the new weight for the remaining propagating photon is wa.

Then, the photon has to be redirected. A random angle is generated using the Henyey-Greenstein function and a uniformly distributed variable ξ.

$$\cos\theta = \frac{1}{2g}\left\{1 + g^2 - \left[\frac{1 - g^2}{1 - g + 2g\xi}\right]^2\right\} \tag{1.15}$$

Propagation at an interface

At the interface of changing medium refractive index, the photon will interact with the intersection and either reflect or refract depending on the angle and on the mismatch in the index of refraction. The reflection probability can just be attributed to the Fresnel Coefficient (see Equation 1.16).

$$R(\theta_i) = \frac{1}{2}\left[\frac{\sin^2(\theta_i - \theta_t)}{\sin^2(\theta_i + \theta_t)} + \frac{\tan^2(\theta_i - \theta_t)}{\tan^2(\theta_i + \theta_t)}\right] \tag{1.16}$$

If the photon is transmitted, it will be deflected according to snell's law (see Equation 1.17).

$$n_1 \sin\theta_1 = n_2 \sin\theta_2 \tag{1.17}$$

Also, passed a certain angle, it is possible to get a total internal reflection, which is the point at which all the light is reflected. The angle at which this phenomenon occurs is

$$\theta_c = \arcsin(n_2/n_1) \tag{1.18}$$

Finally, it is possible to model Monte Carlo using some approximation to make our random physical variables easily sampled with a uniformly distributed random variable, which is an easy task for a pseudo-random generator. Using these variables in the model in Figure 1.3 for the distance and the scattering angle should render accurate results of the energy distribution in the scene for biological tissues, which is what has been previously verified [22].

1.2 Parallel Computing and Hardware Acceleration

Monte Carlo simulations are characterized by the independence of each individual simulation. This trait makes them particularly amenable to parallel processing. In situations where large sample sizes are required, as is common with Monte Carlo methods, the computational demands can be significant. The computational cost is amplified when using languages like Python, which, being interpreted, is often slower than its compiled counterparts.

Parallel computing has become a staple in contemporary computational paradigms, especially given the rise of hardware optimized for parallel operations. For instance, Graphical Processing Units (GPUs) are integral in gaming due to their capability to handle numerous parallel computations, offering acceleration factors ranging from 1,000 to 100,000 times with a single GPU.

Several hardware architectures offer parallel processing capabilities, including:

- Multi-core Central Processing Unit (Multi-core CPU)

- Graphical Processing Unit (GPU)
- Application-Specific Integrated Circuit (ASIC)
- Field-Programmable Gate-Array (FPGA)
- Tensor Processing Unit (TPU)
- Accelerated Processing Unit (APU)

General-Purpose computing on Graphics Processing Units (GPGPU) involves leveraging the computational prowess of GPUsoriginally intended for graphics computationsfor more general-purpose tasks. Historically, GPUs were dedicated solely to graphics operations. However, the early 2000s witnessed a surge in GPGPU applications when scientists repurposed GPUs by embedding computational tasks within graphical data, prompting a demand for GPGPU-centric tools and APIs. GPGPU has found applications in diverse fields, including scientific computations, machine learning, and image processing [23].

The parallelism in GPUs stems from their architecture, characterized by numerous small processing cores. These cores operate concurrently, enabling the simultaneous execution of a multitude of instructions. GPUs possess substantial localized memory proximate to their computational units. This design minimizes data retrieval times as opposed to relying on more distant central memory. Figure 1.4 illustrates the architectural distinctions between GPUs and traditional CPUs.

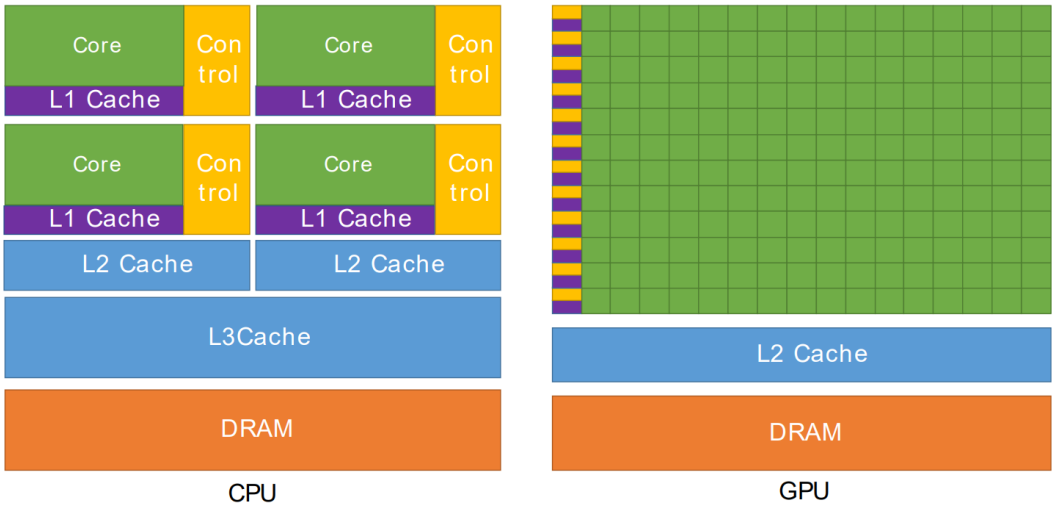


Figure 1.4: This conceptually illustrates the key differences between a CPU and a GPU which is based on resource allocation. The GPU usually has much more cores, but at a lower speed, with memory being shared in core groups. A CPU is usually more oriented toward single-core execution speed [24].

Despite its advantages, GPGPU programming has inherent challenges. The primary hurdle is algorithm design. Algorithms need to be reformulated to make optimal use of the inherently parallel structure of GPUs, which demands extensive code modification and optimization.

1.2.1 OpenCL and PyOpenCL

The main framework chosen for coding the parallelization aspect of this project is OpenCL, justified by its substantial flexibility and extensive cross-platform capabilities, essential for applications intended to run across diverse hardware configurations[25]. Notably, OpenCL facilitates execution across heterogeneous platforms encompassing CPUs, GPUs, and other processors from various manufacturers (see Figure 1.5), thereby significantly broadening hardware compatibility and user reach, which aligns with the objective of this project.

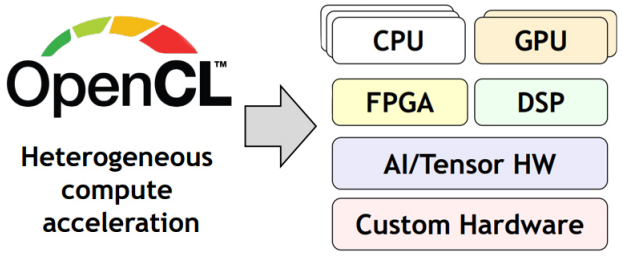


Figure 1.5: Compatibility of OpenCL on many device types. Illustration from OpenCL launch presentation [26].

Additionally, OpenCL has an open-source repository called PyOpenCL[27], which allows to easily access massively parallel compute devices directly from Python. This package was used extensively in the development of PyTissueOptics as the main interface point to control data exchange between GPU and CPU and to launch kernel execution. As a result, it permitted a seamless integration of GPU hardware acceleration for the user.

Many other parallel computing frameworks are available, such as CUDA(NVIDIA), Vulkan(AMD), Metal(Apple), and others. CUDA is a popular alternative because of the user-friendly API it provides as well as many pre-written kernels or tools, such as random number generators. However, CUDA code only supports NVIDIA hardware, which is a major disadvantage when the priority is compatibility and user reach. Metal, for example, is only available for Mac-OS users.

OpenCL Operational Details

OpenCL programs comprise two main components: a host program and a kernel. The host program, usually written in a high-level language like C++, manages the OpenCL platform and oversees the execution of the kernel. In contrast, the kernel, written in a low-level language like OpenCL C, handles the primary computation.

During the execution of an OpenCL program, the host program initializes the OpenCL platform, chooses the appropriate OpenCL device, and then loads and compiles the kernel.

Attribute	SI MD	MIMD
Tasks	1 Task, different parts of data	Multiple tasks, any data
Timing	Synchronous	Asynchronous
Threads	1 Thread/Workgroup	Multiple threads
Speedup	$\propto$ Input size	$\propto$ Independent tasks
Load Balance	Optimum	Depends on scheduling algos

Table 1.1: Differences between data parallelism and task parallelism. Each modality has its purpose, but data parallelism is the most common because it's the easiest to use and the speed increase is higher.

Subsequently, it allocates memory for the data upon which the kernel acts and transfers this data to the OpenCL device. Upon kernel execution completion, the computed results are relayed back to the host. Central to this framework is the Context object, which forms a bridge between the host and the device. All OpenCL commands are invoked via the API, with standard API calls depicted in Figure 1.6 on the Host side. (A detailed example of the Python OpenCL API code, paired with a simple kernel, can be found in Appendix A).

OpenCL, like parallel computing at large, operates under two main paradigms: Single Instruction Multiple Data (SIMD) or data parallelism, and Multiple Instruction Multiple Data (MIMD) or task parallelism. The former, SIMD, involves the repetition of identical operations across cores, albeit on different data segments. The latter, MIMD, is more intricate; each task might vary in execution time, necessitating additional efforts to balance computational loads and maximize efficiency. Distinctive features of both modes are delineated in Table 1.1.

In data parallelism, a practical example is vector addition involving two matrices of dimensions m, n. The goal is to sum corresponding components from both matrices. Each element pair i_1, j_1 and i_2, j_2 gets transferred to private memory and then summed, with results stored in a new buffer. With a GPU having 2000 cores, this matrix addition could be processed nearly 2000 times faster than a single-threaded CPU operation.

In the context of task parallelism, distinct tasks are designated to specific workgroups or individual workitems. Consider a kernel containing the conditional statement: `if $x>3$`, where the action is to subtract vectors; otherwise, the vectors are added. Such tasks, despite being queued asynchronously, can still execute in parallel. However, task parallelism presents optimization challenges, primarily because its efficiency hinges on adept task scheduling and synchronization, both of which demand intricate strategies for optimal execution.

OpenCL gives access to a matrix-like computation platform (ND-Range) where all the workgroups are organized in an N-Dimension grid. This allows to position the workgroups as wanted in any dimension needed to match the size of the incoming data. Each workgroup contains workitems, on which the kernel is executed. This allows for very easy data

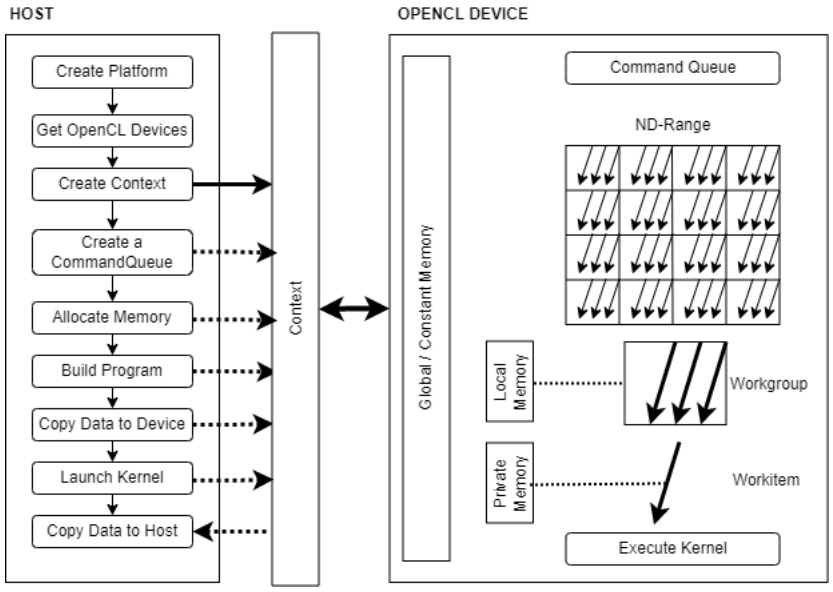


Figure 1.6: Flow of commands and data with the OpenCL platform. Platform control is done through the API on the host device. The kernels are copied to the OpenCL device and executed on each workitem. Global/Constant Memory is shared with all compute units, local memory is exclusive for each workgroup and private memory is exclusive for each workunit.

parallelism when algorithms used to permit this type of computation.

1.3 Ray tracing theory introduction

The Monte Carlo propagation model used in this work treats photons as point-like objects which follow ballistic trajectories and collisions. During a simulation, a line is drawn in the photon's propagation direction. The goal is to know whether the line hits any surface. From that point, we can apply the reflection, and refraction laws and move the photon to the correct position. Figure 1.7 illustrates the problem. This problem is known in the gaming/animation industry as the ray tracing problem. The basic idea behind ray tracing is simple: trace the path of light through a scene, and use that information to determine the color of each pixel in the image. However, actually implementing a ray tracer can be quite complex. For the rest of this work, it can be assumed that we are working in 3D space and that the Cartesian coordinate system is used.

The first step in making a raytracer is to determine how the object's surfaces are defined. This is a domain called digital geometry or the digital 3D modeling process. There are many variations in how computer scientists define geometries on computers. A common technique is called "polygon modeling", and in contrast with other methods, does not require mathematical equations to describe surfaces and is simple to implement. Polygonal modeling is an approach

for modeling objects by representing or approximating their surfaces using polygon meshes (see Figure 1.7). In this method, the basic element is called a vertex, which is a point in 3D space. Connecting 2 vertices generates an edge. Connecting 3 vertices with 3 edge generates a triangle, which is the most basic surface shape. A collection of surfaces that defines a closed region is what defines a solid in the polygon modeling technique. The last sphere in Figure 1.7 depicts a ray hitting a sphere that is composed of many triangular surfaces.

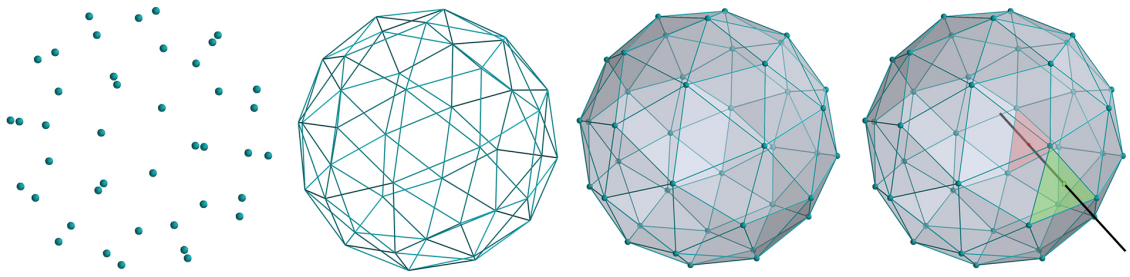


Figure 1.7: Elements necessary to mesh a sphere. Define vertices, connect vertices, form triangles with 3 vertices connections and group them to obtain a fully meshed object. The last sphere shows the ray hit problem where a ray is coming from the right side and going towards the sphere. It first hits the green triangle, which is the entrance hit, and then goes out the sphere from the red triangle in the back.

The naive algorithm would be to go through all the triangles and verify if the line intersects the triangle. However, this algorithm is not practical if we have a large number of triangles, as the time increases linearly with the number of triangles. Moreover, how do we differentiate the first hit, versus the second hit? There are many techniques to solve this problem; a common and efficient solution is to subdivide the 3D-space into non-overlapping boxes, which is a process called space partitioning. The easiest type of subdivision of space is done through axis-aligned bounding boxes (AABB), as is shown in Figure 1.8. This easily defines a region in euclidean space. In Cartesian coordinates, we only need 6 values to define a bounding box: $(x_1, x_2, y_1, y_2, z_1, z_2)$. Confining the objects in a bounding box helps us to know if the ray has a chance to hit the object inside. If a ray does not hit the bounding box, it will surely miss the object. In the case of objects with a high triangle count, it can save a lot of time to be able to discard certain group of triangles. However, with this AABB only encompassing objects, there is improvement if the rays hits the box; all the triangles must be tested for intersection. The idea is to subdivide beyond the object's delimitation to have a better idea of where the ray is going.

The most simple technique is to subdivide all space equally (see Figure 1.9 (a) and (b)). Subdividing space with this method allows for a very simple traversal algorithm: intersect a ray with the next adjacent AABB and test the triangles in that bounding box. This method is practical for scattered polygons or uniformly distributed polygon density through all space. However, this is practically never the the case as 3D model's polygon densities vary in 3D

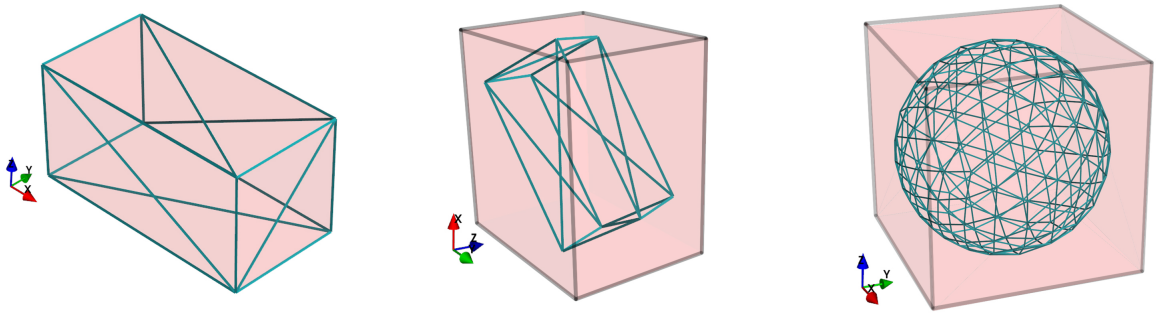


Figure 1.8: Bounding Boxes surrounding meshed objects. The red region represents the bounding box and the blues lines are the mesh. On the left, the rectangle is perfectly aligned with the space axes, this allows for the bounding box to match perfectly. The middle rectangle has been rotated, thus the AABB cannot fit perfectly to capture all the rectangle. A complex object, like a sphere can be delimited in a 3D space by a bounding box.

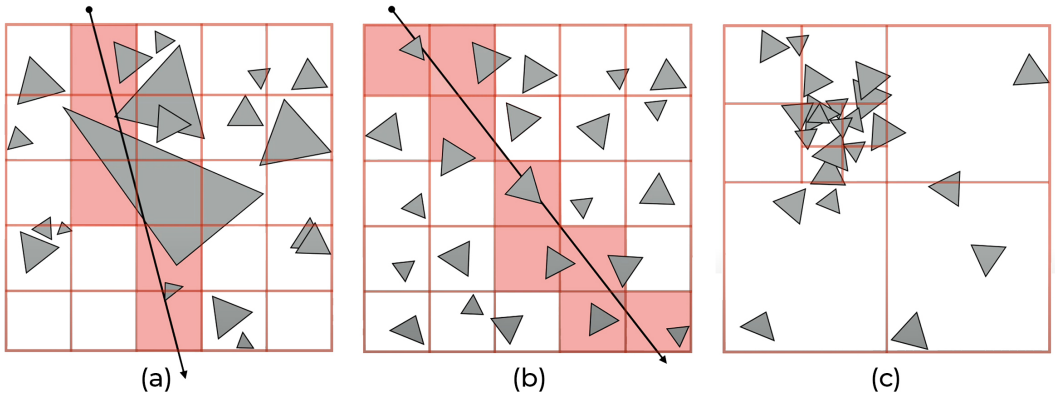


Figure 1.9: Different space subdivision scenarios. (a) shows a uniform grid over overlapping polygons. (b) shows a uniform grid over uniformly distributed polygons. (c) shows quad-subdivision that becomes finer with polygon density. Illustration inspired by Nancy Pollard [28].

space and multiple objects are not uniformly positioned. The solution is to create a smaller grid where the polygon density is higher (see Figure 1.9 (c)). Also, to improve the ray hit search, it is best to organize the subdivision in a tree-like hierarchy.

Space-partitioning systems are often organized in a hierarchical manner, it is quite rare to observe non-hierarchical partitioning like the uniform grid. Preferably, the AABB is recursively divided into sub-regions. Figure 1.10 shows a type of space-partitioning called "binary space-partitioning (BSP)", more specifically a sub-type of BSP called "kd-tree" which will be discussed in details in chapter 2. The BSP technique consists of always splitting the space into 2 sub-regions. Other types of popular splitting techniques exist, such as quad-

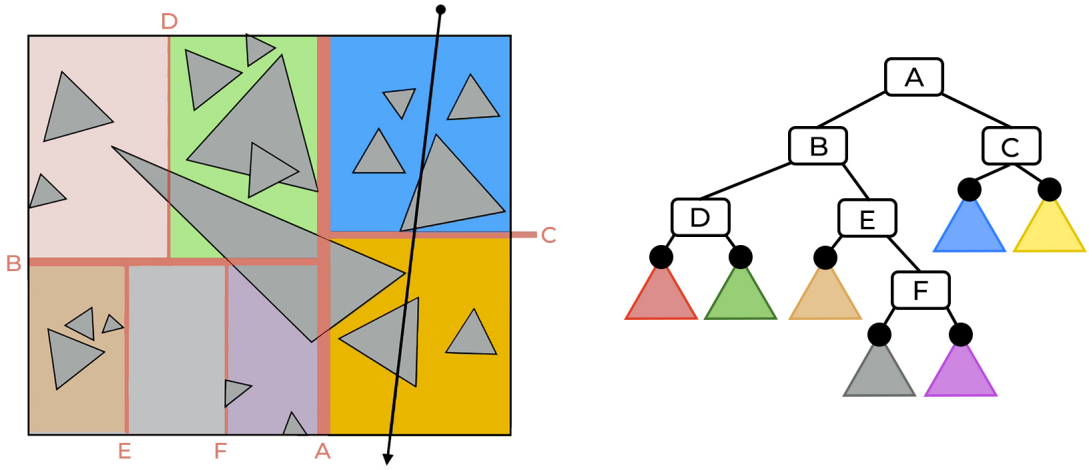


Figure 1.10: Example of a binary space partitioning technique. On the left, the scene contains scattered polygons. The red lines represent the splitting planes and are lettered. On the right, the associated tree is illustrated. The letters in the tree represent the splitting planes and the colored triangles, the end nodes of the tree (called leaves). Each leaf is displayed with a different color. Illustration inspired by [28].

tree and oct-tree, which consist of always splitting the sub-region in 4(quad) or 8(octo). In Figure 1.10, the algorithm starts by selecting a splitting plane (A), which generates 2 sub-regions. These regions are then split again, recursively, until the stop criteria are reached. In the case of Figure 1.10, the ray hits the right of the split plane (A), thus in the tree, it goes to the right. Then, it hits the top region of the split plane (C), thus going to the blue region in the tree. From there, it tries to find a hit with the polygons contained in that region. If it finds a hit, the search can stop. If it does not, the algorithm has to go back into the tree. The specific manner the algorithm searches the tree is called a "traversal algorithm" and is a very important part of the ray-triangle hit problem. One of the main advantages of tree hierarchies is that they are good at early exits, meaning once a hit has been found, the search can almost always stop, which makes these algorithms very fast.

In summary, finding a ray-polygon hit in a scene with many polygons is a hard task. The common method used is to use space partitioning techniques with hierarchical structures. The important steps in solving the ray-hit problem are to carefully select the type of partitioning that is done (binary, quad, octo, etc.), the construction algorithm which selected the splitting planes of the tree, and the traversal algorithm which guides the ray-hit search. Each part has a crucial rule in the overall speed of the ray-hit search. In chapter 2, the selected method will be discussed in detail.

Chapter 2

Methods

The methods section details the decisions made during the development and the technical aspects of design, implementation, and physics. Prior to the beginning of this master's, the PyTissueOptics project already had begun. The capabilities of the module were extremely limited and it had many flaws. Nevertheless, it was functional and simple to use, but the simplicity of use was only one of the criteria that revolved around this package's goals. In Figure 2.1, a code example of the old package is shown with the graph output on the right.

As previously explained in the introduction, the extensibility of the code was limited. This problem slowed down development, and ultimately, completely choked off any attempt to extend the code or improve performance. Moreover, the way the geometry aspect was coded would not have allowed arbitrary 3D support. With that in mind, the work methodology used during this project, described in the next section, is valid for the whole project.

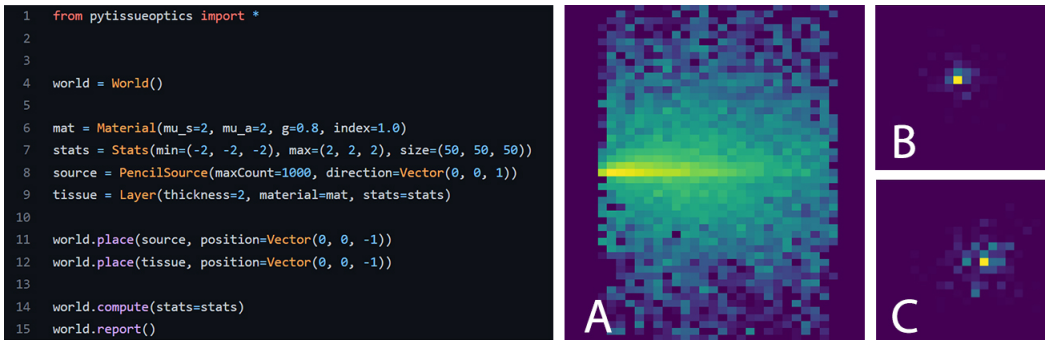


Figure 2.1: Code and graph output from release v1.0.4. (A) represent a cross-section of the energy distribution in the tissue layer. (B) and (C) represent the photon traversal events at the front and back of the tissue.

2.1 On project architecture and quality assurance

The code architecture and quality assurance have been major concerns during the development of this module because of the characteristics required: simplicity, ease of use, ease of modification, and maintainability. Good software architecture requires strong knowledge of architecture techniques, meticulous QA methodology, and development best practices. The PyTissueOptics project had already been initiated, but architecture and code methodology were not following strict sustainable software development rules, which eventually lead to code that was very rigid to modification, hard to maintain, and hard to read, which was against the main objectives of this project. This section explains the software design difficulties encountered and details the selected solutions and their implementation.

2.1.1 Agile software development

Agile software development is the modern standard for developing software. Complex or large-scale projects have to use the right methodology, habits, tools, and workflow to be able to deliver a satisfactory product with high quality. This is not a guide for good software development, but a summary of the techniques used throughout the project, that aim to follow good methodology.

Coding Principles The basic principles of agile software development are often encapsulated in an acronym: SOLID. These 5 design principles are intended to help engineers develop understandable, flexible, and maintainable software [29]. These principles were followed strictly during development. These principles are

- Single-responsibility principle: A class should only have only one job. (SRP)
- Open-closed principle: Entities closed to modifications, but open to extension (OCP)
- Liskov substitution principle: Every subclass should be substitutable by its parent. (LSP)
- Interface segregation principle: Multiple specific interfaces rather than one too general (ISP)
- Dependency inversion principle: Entities should depend on abstractions (DIP)

Workflow Establishing a good workflow and using the right tools is also an important part of agile software development as it helps with problem-solving, issue solving, quick iterations, information keeping, code versions, etc. The GitHub ecosystem was used as it provides the tools for this purpose.

Github has all the following attributes:

- Centered on 'git', which is the most used version control software (VCS).
- Cloud-based code repository, which allows being up to date anywhere at any time without the need to manage hardware components or manually manage a server.
- Provides tools to prevent code conflicts for multiple developers.
- Provides kanban board which helps with feature tracking for fast-paced projects.
- Is the most accessible platform for developers which helps with ease of reach.

2.1.2 Basic UML symbols

In Figure 2.2 the four symbols used are presented to help visualization of the architecture figures. (a) represents inheritance, which is the relation between parent-child. In this case, class B is a child of class A, which means it will inherit all its attributes and methods. (b) represents dependency where B depends on A. An example of dependency is when a method of class B would require an object of class A as an input to correctly function. (c) represents composition where class B must have an attribute which is an instance of A. Class B is composed of class A. (d) represents aggregation, where class B has an instance of class A. However, compared to composition, class B can exist without having that instance of class A. These concepts are key to correctly read the architectural diagrams that are followed.

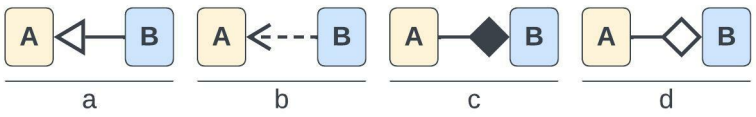


Figure 2.2: List of UML symbols used for the architectural diagrams of this project. (a) is inheritance, (b) is dependency, (c) is composition, (d) is aggregation.

2.1.3 Previous code and architecture

The PyTissueOptics project had begun prior to the beginning of this master's. You can see the old project architecture in Figure 2.3. This architecture posed many challenges to the upgrading of the package. Initial attempts to implement arbitrarily complex geometries and to improve speed failed because mainly of poor architecture design and poor coding choices, which made the code extremely rigid to change.

On the user side, things are not so complicated. This is how the code worked: The user creates a `World` instance. Then, the `Material`, `Stat` and the `Source` which defines the

photon distribution, are created. Then, the geometry object is created; here a `Layer` takes a `Material` and the previously defined `Stats` that will compile the data for this object in particular. Then the objects are positioned in the `World` and the `world.compute` method launches all the calculations. It also takes in a `Stats` object, which in that case, the same object was used for the Layer and the World.

```
from pytissueoptics import *

world = World()
mat = Material(mu_s=2, mu_a=2, g=0.8, index=1.0)
stats = Stats(min=(-2, -2, -2), max=(2, 2, 2), size=(50, 50, 50))
source = PencilSource(maxCount=1000, direction=Vector(0, 0, 1))
tissue = Layer(thickness=2, material=mat, stats=stats)
world.place(source, position=Vector(0, 0, -1))
world.place(tissue, position=Vector(0, 0, -1))
world.compute(stats=stats)
world.report()
```

It is when the world.compute() method is called that things start to get complicated. World.compute() launches its own propagate function to propagate the photons in the vacuum until it reaches a geometry (see Appendix C). It also implements its own function to find an intersection: findNextObstacle. When the photon enters a geometry, the geometry takes over the propagation physics with its own propagate method and own intersection-seeking methods (see Appendix D). This is a problem for multiple reasons:

- The propagation logic is scattered all across the package which is very bad for maintainability
- This is prone to errors and different implementations between the different class implementations.
- The World and Geometry class takes care of logic that should be in the `Photon` class because it is not part of their domain.
- This limits extendability, as a new propagation feature has to modify multiple classes.

Then, the World and Geometry classes also have the job to save the data to the Stats object. This is wrong for the same reasons as previously mentioned, but also because the object that saves the Data and does calculations on the data should be two separate classes. As of this version, saving data, post-simulation data management, and the display are all managed by one object. In more formal terms, these are the problems that are found with the current architecture

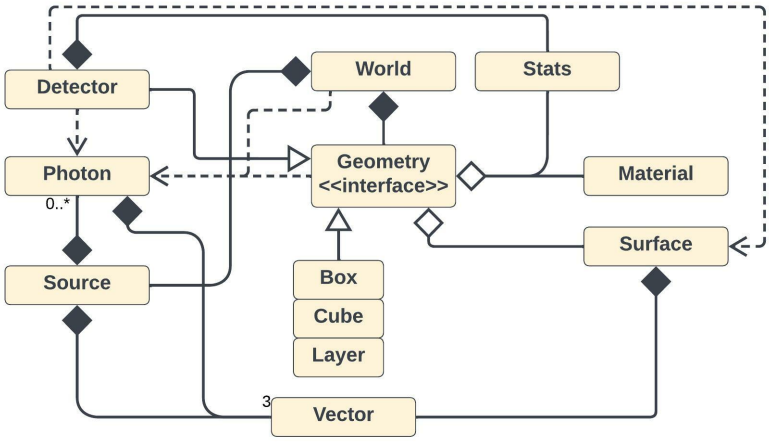


Figure 2.3: PyTissueOptic code architecture at the beginning of the project. The Geometry class is at the center of the whole package

- The `Geometry` class and the `World` class were in charge of propagating the photons and of all the propagation mechanisms. This violates many principles, such as SRP, because the geometry should not manage geometry methods and photon propagation methods, and also DIP because these two domains, which are not related to each other, now depend on one another. The geometry class also violates the OCP because there was no proper interface that allowed for easy extendability.
- The `Stats` object violates SRP by being responsible for logging, statistical calculations, and display. This also creates a dependency on the statistics domain towards the fragile display domain (DIP) and the absence of interfaces limits the extension of the display capabilities (OCP). The strong dependence from both World and Geometry on the Stats object also creates duplicated code and violates DIP because of the low level. The geometry domain should not depend on the high-level statistics domain.

Before the architecture was looked upon, a trial was done to counter the problem of speed. The idea was to vectorize the photon class in order to be able to launch the propagation of several photons "at the same time" through calculations and matrix operations on Numpy or on Cupy (Numpy on GPU). Thus, in Figure 2.4 we can see the Vectors and Scalars classes which were vectorized versions of the `Vector` class, coded in Numpy and Cupy arrays and which allowed doing exactly the same operations. The problem with this technique is that since the geometry class was in charge of propagation, it was necessary to create new methods for multiple photon propagation which greatly congested the class. Before pursuing this idea, speed tests were done with a minimalistic model (see Appendix B) which yielded a speed increase of over 500x. This increase was substantial, but it was without trying to hit any interface (without geometries). When trying to implement the physics, the code became extremely entangled as Geometry and World had to implement new propagation physics to

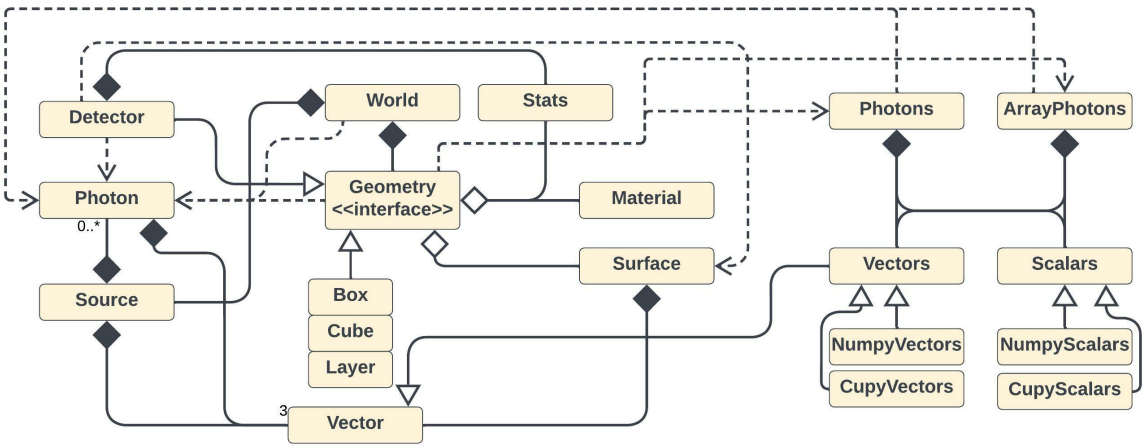


Figure 2.4: PyTissueOptic code architecture after the modifications to vectorize photon propagation.

manage the Photon's propagation. These were called PropagateMany(). A working version was done, but the resultant speed was much slower than anticipated. A decision was made to start from scratch because the attempts to solve the architecture without breaking any code were failing and debugging was becoming extremely tedious. Formally, this attempt added another level of principle-breaking code.

- There is a strong dependence on two different implementations of propagatable objects : the singular `Photon` and the plural `Photons`. Not only do these two objects should be abstracted under a single interface and forced to respect LSP, but clients should not depend that heavily on it. The current code has `Geometry`, `Source` and Material all depending on the implementation details of both `Photon` and Photons, which also exhibits a similar DIP violation to the above case. This results in a lot of duplicated code across clients to handle the case of a singular `Photon` or a batch of `Photons`.

2.1.4 Recent code and architecture

Summary One of the things that are important when developing a large software is to be able to correctly separate domains. In this project, there were 2 very clear separate domains: geometry and physics. The first step is to separate the domains. An attempt had been made to transpose the propagation functions into the `Photon` object, but the interdependence of the two objects and the lack of testing of the module led this experiment to fail. Thus, the first mission was to create an architecture that would allow the separation of the domains (see Figure 2.5).

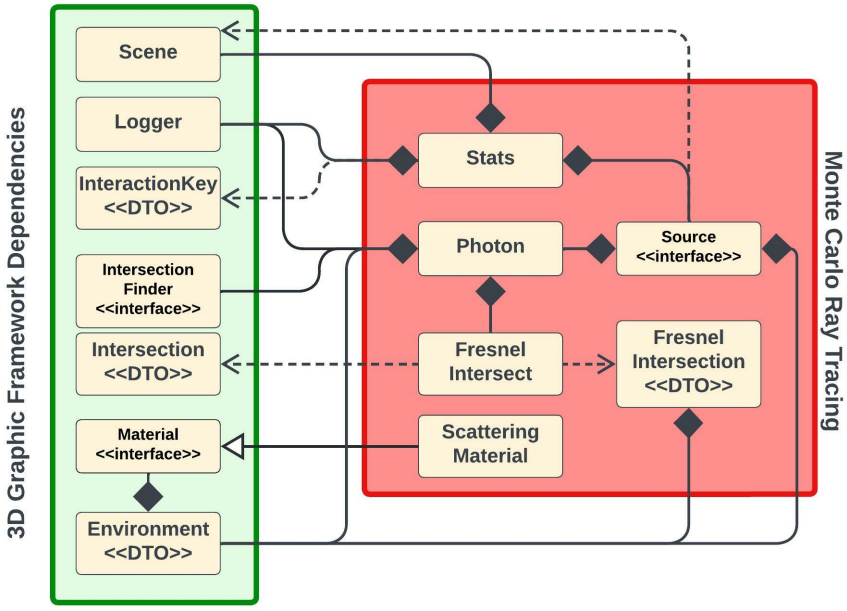


Figure 2.5: Simplification of the project's modules to illustrate dependencies. All the geometry aspects are stored in one module and the physic in another. The dependency is one way: The Photon, Stats, and `Source` require many elements from the geometry module. However, except for the Logger and the Scene, these dependencies are based on abstractions, interfaces, or DTO, which is very good for extendability and maintainability.

The geometry domain was contained in a separate module called 3D Graphic Framework which took care of everything from creating the solid meshes to managing the ray-hit search algorithms.

The Physics domain would take care of the photon creation, the propagation physics, and the data statistics. This separated module would have dependencies on interfaces in the 3D graphic Framework module. This one-way dependency on abstraction solves the DIP problems in the previous architecture.

3D Graphic Framework To be able to manage arbitrarily complex geometries, all the geometry aspect was reviewed. Because of the growing complexity of such a system, it was subdivided into sub-modules: geometry, solids, intersections, viewer, tree, logger, and scene. Everything starts with the Solid. The `Solid` class defines an interface to allow the coding of any meshing type. Basic geometrical objects were pre-created: Ellipsoid, Sphere, Cube, Cylinder, Cone. This `Solid` is dependent on Vertices which defines the points that mesh the solid. These points are grouped in 3 to form triangular surface. The way that these points are grouped depend on the algorithm used. This list of `Triangle` is given to the `Solid` object, which defines its surface. Many solids can be given to a Scene object. This Scene object

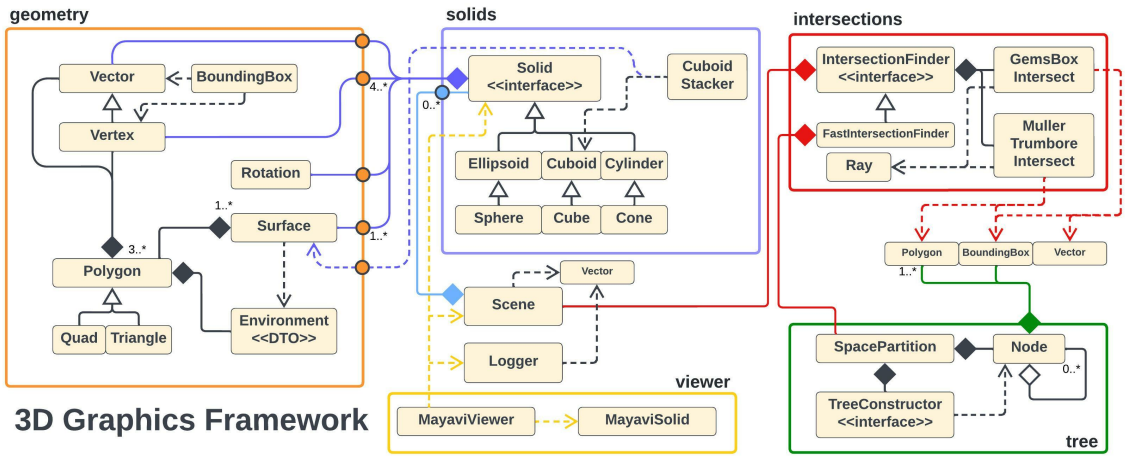


Figure 2.6: The 3D Graphics Framework complete architecture. The separation in sub-modules allows for a clear architecture despite its complexity and inter-dependencies were kept to a minimum.

only manages the solids as a group and can be given Scene parameters. Then, if an external module wants to interact with the solid in the Scene, it can use an IntersectionFinder, which is an interface that allows for objects that look for intersections with objects in the Scene. One realization of the interface is FastIntersectionFinder, which separates the space using the SpacePartition class. The space partition class selects which type of space partitioning it wants to do using a TreeConstructor implementation. Details on the implementation of these classes and interfaces will be given in the following sections.

2.2 On arbitrary geometry support - 3D Graphic Framework

To support arbitrary geometries, the 3D Graphic Framework was coded. This section will dive through all the details relating to the geometrical aspect. First, we will define the geometry basis: Vectors, Polygons, Surfaces, and related objects. Then, these basic objects will be used to build Solids which are at the center of this module. The algorithms to mesh these solids will be explored.

2.2.1 Mesh-based technique

As explained in Section 1.3, the first step to making 3D geometries is to determine the geometric modeling method that will be used to describe surfaces and solids: parametric function, sweeping, voxelization, and surface mesh, are all schemes that allow for solid representation in 3D space. Geometric modeling is a branch of applied mathematics

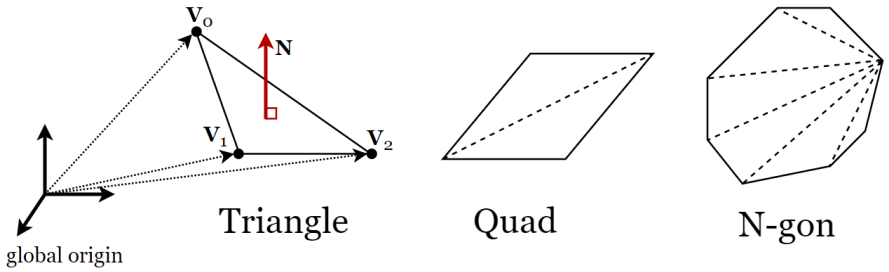


Figure 2.7: The polygons are defined with a set of 3D vertices originating from the global origin. The normal points toward the viewer when the vertices are ordered in a counter-clockwise manner as shown in the triangle example. Any quad or n-gon can be broken down into triangles along the dotted lines.

and computational geometry that studies methods and algorithms for the mathematical description of shapes. It is the process of constructing a complete mathematical description to model a physical entity or system. The surface mesh technique is one of the most used techniques because of its versatility, ease of use, and simplicity. It requires the aggregation of groups of polygons to define a solid, which is essentially an enclosed volume. This enumeration describes the necessary components implemented to correctly describe solids in the 3D graphics framework. See Figure 2.7 for visual aid.

- `Vector`: A class that holds an (x, y, z) component to define its position and on which vector operations can be performed.
- `Vertex`: A child of `Vector` that holds an additional attribute `normal`, which is useful for the smoothing algorithm, discussed in Section 2.4.1.
- `Polygon`: A class that acts as an abstract class for any planar polygon. It holds vertices that are ordered in a counter clock-wise manner, a normal attribute that points in the cross-product direction between edge $(V_1 - V_2) \times (V_2 - V_1)$.
- `Solid`: A class that holds the vertices and the polygons required to define an enclosed volume. Transforms can be applied to this object, such as rotation and translation. It holds a material, which can later describe the behavior of the solid in simulations.

Detailed Solid management

The `Solid` class of the 3D Graphic Framework has some specific features that enable differentiation between polygon groups as faces and the smoothing of these faces.

```python
class Solid:
    def __init__(self, vertices: List[Vertex], position: Vector = Vector(0, 0,
    ↪ 0),
    surfaces: SurfaceCollection = None, material=None, label: str = "solid",
    ↪ smooth: bool = False):
        self._vertices = vertices
        self._surfaces = surfaces
        self._material = material
        self._position = Vector(0, 0, 0)
        self._orientation: Rotation = Rotation()
        self._bbox = None
        self._label = label
        if not self._surfaces:
            self._computeMesh()
        self.translateTo(position)
        self._setInsideEnvironment()
        self._resetBoundingBoxes()
        self._resetPolygonsCentroids()
        if smooth:
            self.smooth()
```

The `Solid` holds a position which is a vector, a list of all its vertices, and a `SurfaceCollection` instance which contains all the associations between faces' labels and the polygons forming that surface. It holds a `BoundingBox` which is an object that holds 6 values $(x_{min}, x_{max}, y_{min}$
$, y_{max}, z_{min}, z_{max})$ of an Axis-Aligned Bounding box (AABB). It defines the boundaries of the `Solid` and helps with the collision detection between solids.

```python
class SurfaceCollection:
    def __init__(self):
        self._surfaces: Dict[str, List[Polygon]] = {}
```

The `SurfaceCollection` acts as a container to group polygons together to describe a face. It links a label with a list of polygons. This allows to only smooth particular groups of polygons on a solid since it is possible to refer to the polygons with only a label, which renders the smoothing process a lot easier. This is useful in the case of curved solids, like a sphere, where the normal should not equate the current triangle's normal, but rather be a weighted normal between all normals. This allows for reflections and refraction which are more accurate, in the sens that it is not rastered. Also, in this association, we can link data to polygons groups using only the label, which is very helpful for any plugin that wants to use surface data, like calculating light fluency through a specific surface.

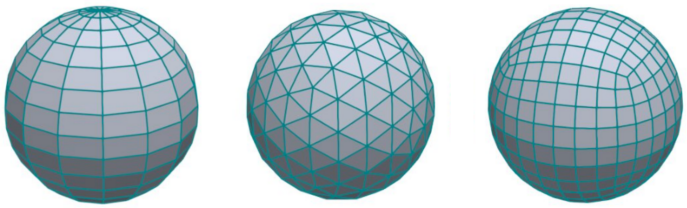


Figure 2.8: Resulting sphere meshes from different meshing algorithms. From left to right, the UV sphere, the icosphere, and the spherified cube. [30]

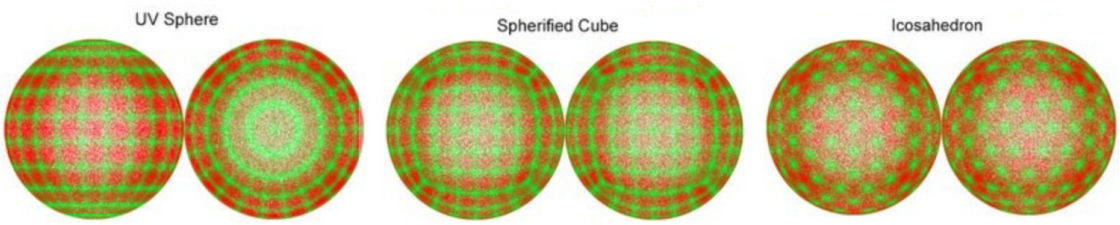


Figure 2.9: Representation of the radius error for different sphere meshing techniques. Green represents a smaller error and red represents a greater error. The green spots represent the vertex points, which are clipped to the unit sphere. [31]

2.2.2 Meshing Algorithms

Sphere There are multiple methods to mesh a sphere, each with its own particularities. As discussed, a mesh will always be an inexact representation of the real object, so the best we can do is to find metrics that quantitatively describe the mesh quality. The most common meshing techniques are *UV-Sphere*, *Normalized-Cube*, *Spherified-Cube*, *Icosahedron*, which resulting meshes can be seen in Figure 2.11. To choose an algorithm, we used the following metrics: Average-Error-to-Unit-Sphere (AETUS), Maximum-Error-To-Unit-Sphere (METUS), and Average-Triangle-Area-Error (ATAE).

As demonstrated by [31], the algorithm which performs the best for all these metrics is the icosahedron algorithm. Furthermore, it also creates the most uniform sphere mesh among all the algorithms. It is illustrated in Figure 2.9 that the radius error on the icosphere is uniform and isotropic. Thus, this algorithm was selected.

The icosahedron algorithm starts with one of the platonic solids, the icosahedron, which is composed of 20 equilateral triangles. This method creates a very uniform arrangement of the vertices where the distance between any two adjacent vertices is always the same and all triangles are equivalent. To get a higher number of triangles we need to subdivide each triangle into four triangles by creating a new vertex at the middle point of each edge, which is then normalized, to make it lie on the unit sphere. The major drawback with this method is that we can only increase the number of faces by four each time.

Figure 2.10: Algorithm steps to increase the icosphere order. 0. Generate the icosahedron (order 0). 1. Select a triangular face. 2. Find the middle of each connecting vertices and set these as new points. 3. Re-mesh the triangle into four triangles. 4. Lift the new vertices to the unit sphere. 5. Repeat these steps for all the triangles. The result will be an icosphere with a higher order.

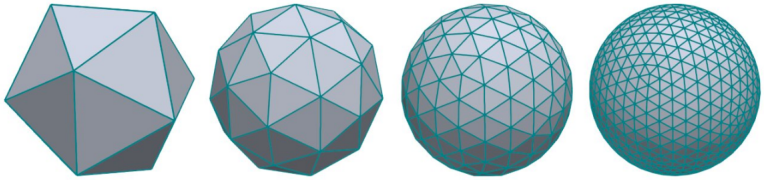


Figure 2.11: Icospheres of order 0 to 3. From left to right, they contain 20, 80, 320, and 1280 triangles. The number of triangles is multiplied by 4 for each order increase.

Ellipsoid Just like the sphere, there are multiple methods to mesh an ellipsoid. The method used consists of clipping the newly generated vertices to an ellipsoid function instead of clipping them on the unit sphere. This essentially elongates the icosphere which generates an ellipsoid-like shape. Knowing the following ellipsoid equation,

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1 \tag{2.1}$$

It is possible to retrieve the Cartesian position of the surface for a given ϕ and θ. For that, we start by parametrizing the ellipsoid equation. It is assumed elongation coefficients a, b, c, are axis-aligned. Then, we apply a coordinate system change from Cartesian to spherical and using the local radius r, we multiply the spherical coordinates to find the actual x, y, and z. This results in the following equations

$$r = \sqrt{x^2 + y^2 + z^2} \tag{2.2}$$

$$x = r\sin(\theta)\cos(\varphi) \tag{2.3}$$

$$y = r\sin(\theta)\sin(\varphi) \tag{2.4}$$

$$z = r\cos(\theta) \tag{2.5}$$

From these equations, we can substitute equations 2.3, 2.4, 2.5 in equation 2.1 and isolate r

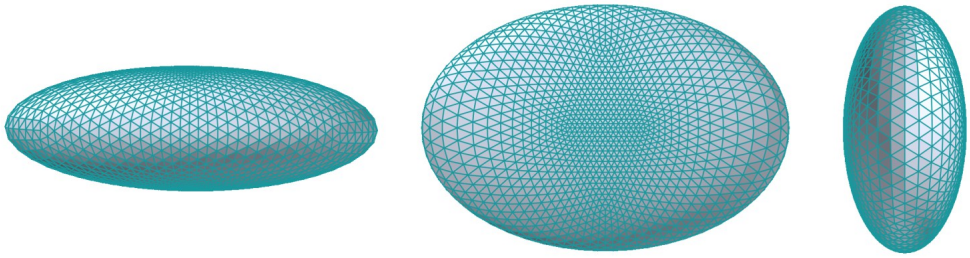


Figure 2.12: Meshed ellipsoid of order 4 based on the proposed algorithm with the parameters ($a = 1.5$, $b = 0.5$, $c = 1$). From left to right, side view, front view, top view. Images generated from our 3D engine.

as a function of θ and ϕ.

$$1 = \frac{r^2 \cos\theta^2 \sin\phi^2}{a^2} + \frac{r^2 \sin\theta^2 \sin\phi^2}{b^2} + \frac{r^2 \cos\phi^2}{c^2} \tag{2.6}$$

$$r^2 = \left[\frac{\cos\theta^2 \sin\phi^2}{a^2} + \frac{\sin\theta^2 \sin\phi^2}{b^2} + \frac{\cos\phi^2}{c^2}\right]^{-1} \tag{2.7}$$

$$r(\theta, \phi) = \sqrt{\left[\frac{\cos\theta^2 \sin\phi^2}{a^2} + \frac{\sin\theta^2 \sin\phi^2}{b^2} + \frac{\cos\phi^2}{c^2}\right]^{-1}} \tag{2.8}$$

Thus, from the meshed-sphere of order x, we check the angle θ and ϕ from each vertex and calculate the corresponding radius from the equation 2.8. We then move the vertex in the direction (θ, ϕ) by the correct scaling factor.

To maintain logical class inheritance, the icosphere algorithm is located inside the `Ellipsoid` class, because the Sphere is just a specific case of an ellipsoid with $a = b = c$. in Figure 2.12 we can see the mesh of different ellipsoids. It is possible to observe that the mesh density is not uniform. This is the major drawback of generating the ellipsoid from an already uniform mesh. An alternative would have been to sample the angle space in a way that would have resulted in a uniform mesh. However, this was not essential for the purpose of this module and could very well be implemented by a third party.

Cylinder The cylinder meshing algorithm is a simple UV algorithm. The two parameters, u and v represent the resolution of the sampling on two model parameters, in this case, the number of divisions around the axis and the number of vertical steps.

From the bottom, we generate a point on the circumference for each angle step ($2\pi/u$). This is done layer-by-layer to the end of the cylinder. To keep the object quite general, it is possible to give an axial profile function that codes for the distance from the center point which is a function of the height of the cylinder. For the cylinder, this function is a constant, thus, the radius never changes. After positioning the points on a layer, the vertical position by a step (height/v). When all the vertices are generated, it is time to connect them - mesh the

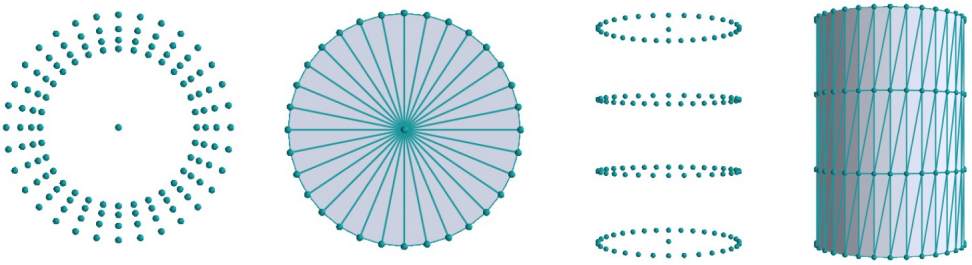


Figure 2.13: Generated vertices and mesh for a normal cylinder. The parameters for this cylinder were $u = 32$, $v = 4$, height= 3, function=1. From left to right, top view vertices, top view mesh, side view vertices, side view mesh.

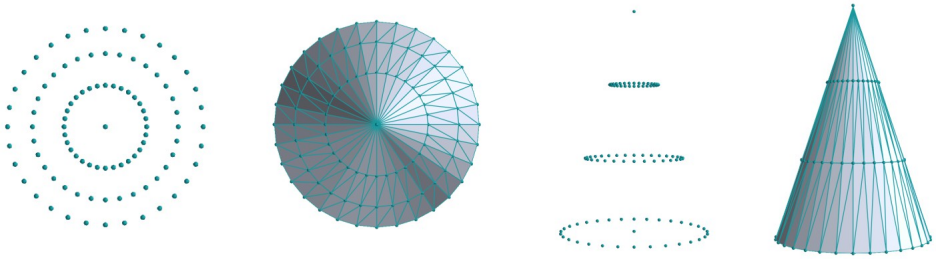


Figure 2.14: Generated vertices and mesh for a cone. The parameters for this cylinder were $u = 32$, $v = 4$, height= 3, function=$(h - y)/h$. From left to right, top view vertices, top view mesh, side view vertices, side view mesh.

solid. The meshing algorithm simply connects 2 vertices on the current layer and the one on the next layer, which if looped through all layers, produces a triangular pattern as seen in Figure 2.13.

Cone The Cone object is based on the Cylinder object. Since the Cylinder algorithm is general, it is possible to give a profile function to make a cone. In this case, the function is

$$(h - y)/h \tag{2.9}$$

where h is the total height along the symmetric axis of the object and y is the current height of the vertex. This produces the mesh as seen in Figure 2.14

Cuboid The cuboid is one of the most basic and simple shapes to mesh. It consists of only 8 vertices and 12 triangles (2 per face). The cuboid parameters a, b, c are defined to be x, y, z, axis-aligned and code for the side length according to each axis. The meshing is done by hand since it is so simple. Despite its simplicity, the cuboid is usually one of the most used shapes and it is the basis for the Stacked-Cuboid object, which is of uppermost importance. in Figure 2.15, we illustrate some possible meshes.

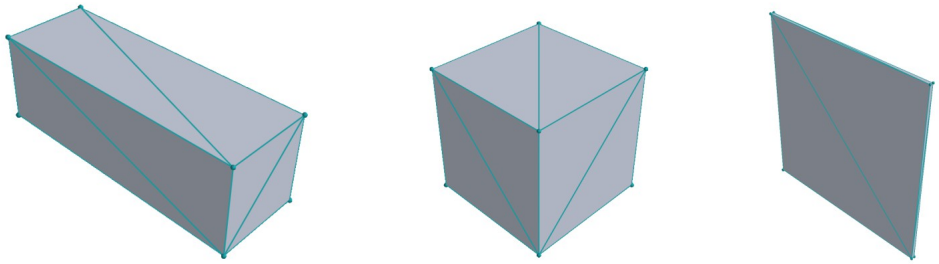


Figure 2.15: Different cuboid meshes. From left to right the parameters are $(a = 1, b = 3, c = 1)$, $(a = 1, b = 1, c = 1)$, $(a = 0.01, b = 1, c = 1)$. It is possible to play with those parameters to make thinner or more elongated cuboids. The special case of $a = b = c$ is the cube.

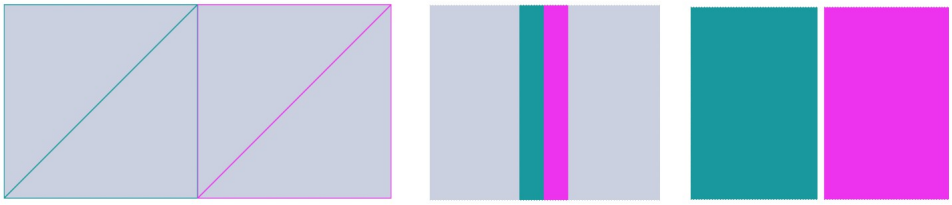


Figure 2.16: Progressive zoom on the interface of two cuboids placed side-to-side so that the right side of the blue cuboid and the left side of the magenta cuboid would merge. When zooming we can see there is a gap between the two, which is due to rounding errors (The gap here is exaggerated for demonstration purposes).

Stacked-cuboid The stacked cuboid is of uppermost importance for this module because it solves a problem for simple multi-layered tissues. When trying to position two cuboids side by side with one interface overlapping, there will probably be some rounding numerical error at some point. The problem is illustrated in Figure 2.16. This can produce many problems: (1) The collision will occur on the second object's interface first, leading to wrong refraction or reflection. (2) A gap will occur so the collision will hit material with the surrounding material instead of the second cuboid material which will again cause wrong reflection and refraction. (3) In most cases, the interfaces don't have knowledge about other interfaces, which will lead to the wrong refractive index being used, thus the interfaces should be merged into one interface, which is never the case if we just put cuboids close to each other. There are many ways to overcome this challenge: One could make an algorithm that checks coplanarity and proximity of polygons to merge the polygons into one, but this would also require an auto-mesh algorithm if it was to be used on anything else than cuboids because all triangles would differ in size and alignment.

The stacked cuboid solves this problem by allowing axis-aligned cuboids to be stacked on each other as long as they have the same length on the stack axis. This implementation requires explicit stacking by the user and is limited to cuboids, which is what was the goal for the first official release.

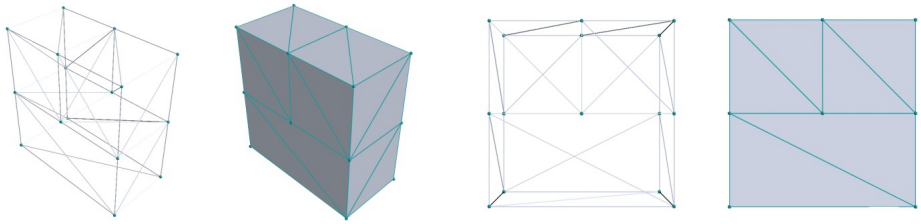


Figure 2.17: Stacked cuboids on 3 different axes. The first cuboid has $(a = 1, b = 1, c = 1)$, the second $(a = 1, b = 1, c = 1)$ which means they can be stacked on any axis. In this case they are stacked on the x axis. The resultant is a cuboid with $(a = 1, b = 2, c = 1)$. The third cuboid has $(a = 1, b = 2, c = 1)$ which means it can be stacked on the previous stack.

First, the stack axis is found and the cuboid's shapes are verified; the 2 cuboids must have matching shapes on the other axes than the stack axis. Second, the second cuboid is moved to the correct position. Third, the interface duplicate is removed and the interface materials are corrected. Finally, the cuboid is re-meshed and given the previously corrected interface at the junction. The 2 cuboids have now become one, this way, it is possible to stack it on already stacked cuboids, as long as the shape matches correctly, as seen in Figure 2.17.

External meshed objects The module needs to be able to support complex 3D geometries, thus the ability to import models from different file formats is necessary. To achieve that, the following structure was implemented: A general object which is in charge of creating the solid from a conventional description of a 3D object's components. This conventional description consists of encapsulating the solid information in a `ParsedObject` Data Object, which contains surfaces described as `ParsedSurface`, which are indices pointing to a list containing the actual 3-d vectors.

```
1  @dataclass
2  class ParsedObject:
3      material: str
4      surfaces: Dict[str, ParsedSurface]
5
6  @dataclass
7  class ParsedSurface:
8      polygons: List[List[int]]
9      normals: List[List[int]]
10     texCoords: List[List[int]]
```

The task of translating from the file to a standard form is given to an interface realization, the `Parser`, which requires implementing a different algorithm for each file format. The result of the `Parser` gives a convenient way of describing objects before the `Loader` converts them into Python objects. The `Parser` also holds lists of all the vertices, normals, and texture coordinates, which are appended to when the file is being parsed.

The task of converting the parsed data into python objects is managed by the `Loader`. The `Loader` automatically selects the `Parser` instance it needs depending on the file extension. After invoking the load method, it returns a list of all the solids in the file. For now, only the wavefront files (.obj) are supported.

First, the conversion algorithm cuts all the N-gons into triangles (see Figure 2.7), in case some polygons were not planar. Then, in a loop, all the vertices and normals are objectified into `Vertex` and `Vector` instantiations. Polygons are created from the indices in the parsed data, referring to the complete vertices list in the Parser instance, which now have a python object equivalent. Afterwards, the `ParsedObject` will be converted to `Solid` objects. First, creating all the `SurfaceCollections` with the polygon object references and the labels given to the surfaces in the `ParsedObject`. Then, creating the `Solid` is easy with all the appropriate input created, the creation is natural. Some converted test models are illustrated in Figure 2.18.

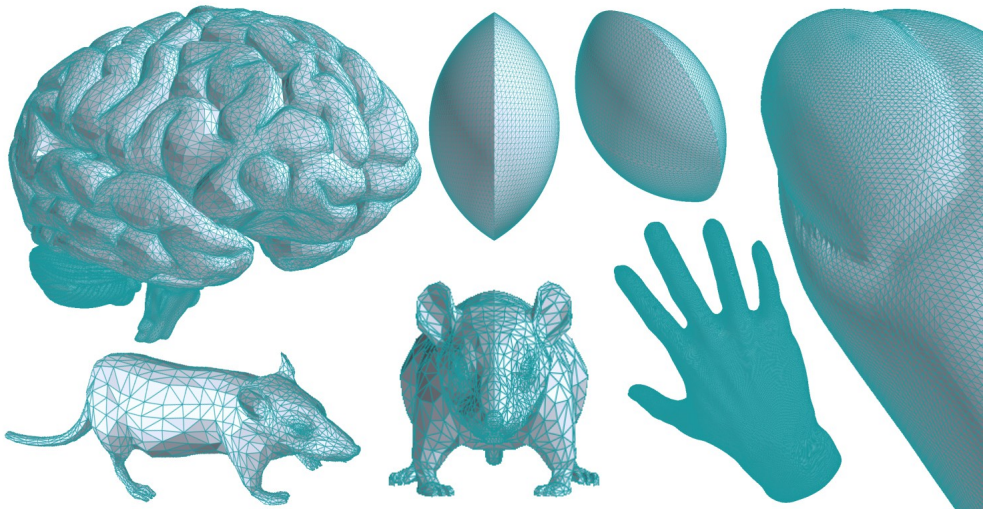


Figure 2.18: Sample of imported complex 3D models. The number of triangles in each model are the following: Brain(385k), Lens(14k), Rat(5k), Hand(741k)

2.3 On Monte Carlo ray tracing implementation

This section is intended to detail the implementation of physics. This relates to the creation of the `Photon` and `Source` objects, as well as the propagation algorithms and the intersection physics (reflection, refraction).

2.3.1 Photon

The `Photon` object is the object that will propagate during our simulation. It is responsible for containing the propagation physics as well as the photon's attributes.

```
1  class Photon:
2      def __init__(self, position: Vector, direction: Vector):
3          self._position = position
4          self._direction = direction
5          self._weight = 1
6          self._environment: Environment = None
7
8          self._er = self._direction.getAnyOrthogonal()
9          self._er.normalize()
10         self._hasContext = False
11         self._fresnelIntersect: FresnelIntersect = None
12
13         self._intersectionFinder: Optional[IntersectionFinder] = None
14         self._logger: Optional[Logger] = None
```

The `Photon` has a position and direction, and a perpendicular direction which are all `Vectors`. The photon has a weight, which starts at 1. The 'hasContext' attribute is useful to dynamically set some photons attribute because the propagation algorithm doesn't allow the photon to propagate if no context was provided. The context consists of an `Environment`, an `IntersectionFinder`, a `Logger` and a `FresnelIntersect`. The `Environment` object is a Data Object (DTO) which simply contains 2 attribute, a `Solid` and a `Material`. This will allow the `Photon` to know in which `Solid` it is and with which material that object is made of, thus the name "environment". Next, the `IntersectionFinder` instance, which is the object that conducts the ray-hit search, will give the ability to the photon to ask to the geometry if it is going to hit an interface, which is crucial for the physics of light propagation. The `Logger` object is the instance in which the photon has to log data when it hits an interface or leave energy after an absorption interaction. The last object that the `Photon` has is the `FresnelIntersect` object which defines the behavior on intersections. When an intersection is found by the `IntersectionFinder`, the `FresnelIntersection` will manage what happens with this information. Separating all tasks into different objects greatly simplifies the code, but also allows for great flexibility if a user would want to modify the intersection physics, one could just use its own implementation of an `FresnelIntersection`.

2.3.2 Source

The `Source` interface is the entity that will guide the creation of the `Photon` instances, as well as launching their propagation methods. Realizations of this interface have the job to give the position and the direction of these photons to simulate different types of sources.

```
class Source:
    def __init__(self, position: Vector, N: int, useHardwareAcceleration: bool
    ↪  = False):
        self._position = position
        self._N = N
        self._photons: Union[List[Photon], CLPhotons] = []
        self._environment = None
        self._loadPhotonsCPU()

    def propagate(self, scene: RayScatteringScene, logger: Logger = None,
    ↪  showProgress: bool = True):
            self._propagateCPU(scene, logger, showProgress)

    def _propagateCPU(self, scene: RayScatteringScene, logger: Logger = None,
    ↪  showProgress: bool = True):
        intersectionFinder = FastIntersectionFinder(scene)
        self._environment = scene.getEnvironmentAt(self._position)
        self._prepareLogger(logger)

        for i in progressBar(range(self._N), desc="Propagating photons",
        ↪  disable=not showProgress):
            self._photons[i].setContext(self._environment,
            ↪  intersectionFinder=intersectionFinder, logger=logger)
            self._photons[i].propagate()

    def getInitialPositionsAndDirections(self) -> Tuple[np.ndarray,
    ↪  np.ndarray]:
        """ To be implemented by subclasses. Needs to return a tuple containing
        ↪  the initial positions and normalized directions of the photons as
        ↪  (N, 3) numpy arrays. """
        raise NotImplementedError

    def _loadPhotonsCPU(self):
        positions, directions = self.getInitialPositionsAndDirections()
        for i in range(self._N):
            self._photons.append(Photon(Vector(*positions[i]),
            ↪  Vector(*directions[i])))
```

2.3.3 FresnelIntersection

The `FresnelIntersection` object is the entity that manages the intersection physics: determining if there is a reflection or refraction and computing the angles at which the photon will be redirected. Its only public method is the compute method which will return a `FresnelIntersection` DTO containing the relevant information to continue the modify the photon propagation direction. From our previously defined `Surface`, it is known that the normal always points outwards of a `Solid`, thus it is possible to verify if the intersection is outgoing or ongoing by using a dot product between the ray and the intersection normal. This will simply allow the object to correctly set the refractive indices for the reflection and refraction computations. Once the `indexIn` and `indexOut` are set, the `getIsReflected` method checks the angle of incidence on the surface and takes into account the mismatch of impedance to determine if the photon is reflected or not. Then, the angle of reflection or refraction is calculated. The interaction physics is explained in section 1.1.2.

```
class FresnelIntersect:
    _indexIn: float
    _indexOut: float
    _thetaIn: float

    def compute(self, rayDirection: Vector, intersection: Intersection) ->
    ↪ FresnelIntersection:
        rayDirection = rayDirection
        normal = intersection.normal.copy()

        goingInside = rayDirection.dot(normal) < 0
        if goingInside:
            normal.multiply(-1)
            self._indexIn = intersection.outsideEnvironment.material.n
            self._indexOut = intersection.insideEnvironment.material.n
            nextEnvironment = intersection.insideEnvironment
        else:
            self._indexIn = intersection.insideEnvironment.material.n
            self._indexOut = intersection.outsideEnvironment.material.n
            nextEnvironment = intersection.outsideEnvironment

        incidencePlane = rayDirection.cross(normal)
        if incidencePlane.getNorm() < 1e-7:
            incidencePlane = rayDirection.getAnyOrthogonal()
        incidencePlane.normalize()

        self._thetaIn = math.acos(normal.dot(rayDirection))

        reflected = self._getIsReflected()
```

```
29
30          if reflected:
31              angleDeflection = self._getReflectionDeflection()
32          else:
33              angleDeflection = self._getRefractionDeflection()
34
35          return FresnelIntersection(nextEnvironment, incidencePlane, reflected,
            ↪  angleDeflection)
```

2.3.4 Propagation algorithm

As discussed in the chapter , the propagation algorithm is based on the MCML model [2]. The essential steps are the same, except for the photon intersection which was modified to fit with the graphics framework and allow a more flexible intersection. First, the `Source` object is the one that creates all the instances of the photons and set their context. Then, in a loop, it will launch the `Photon.propagate()` method, which will propagate using the variable stepsize technique. The way it was implemented, the photon starts with a propagation distance of 0. If it is 0, a stepsize will be generated from the material properties (see 1.12). Knowing the direction of the photon and the stepsize, the photon will use its `IntersectionFinder` to locate the next intersection. If no intersection is found, the photon is moved, then scattered and the `distanceLeft` is set to 0, so that the next loop will generate a new propagating distance. This is done so that, if an intersection is found, the distance to travel after the intersection is saved and the loop now uses this distance instead of generating a new one. With this `distanceLeft`, a new intersection is searched. Without this method, reflections near multiple surfaces, like in a cube corner, could have been reflected only once if the photon had propagated the whole distance without searching for another surface. When the photon scatters, the angle of scattering is generated based on the material's properties and some energy is deposited (as weight loss) and logged in the `Logger` object possessed by the `Photon`. Propagation physics and details are explained at section 1.1.2. Propagation steps are summarized in a flowchart in Figure 2.19.

```
1   def propagate(self):
2       distance = 0
3       while self.isAlive:
4           distance = self.step(distance)
5           self.roulette()
6
7   def step(self, distance=0) -> float:
8       if distance == 0:
9           distance = self.material.getScatteringDistance()
10      intersection = self._getIntersection(distance)
11      if intersection:
```

```
            self.moveBy(intersection.distance)
            distanceLeft = self.reflectOrRefract(intersection)
        else:
            if math.isinf(distance):
                self._weight = 0
                return 0
            self.moveBy(distance)
            distanceLeft = 0
            self.scatter()
        return distanceLeft
```

2.4 On computational speed

Because of the nature of Monte Carlo, the computations have to be extremely fast to render acceptable results in an acceptable time. Two major speed bottlenecks were found, first, the ray-hit search is a particularly hard problem for large amounts of polygons (which is now the case), so finding an efficient method of finding a ray-hit is an important part of speeding up the simulations. Second, parallelizing the propagation physics is a logical step for the Monte Carlo method, as discussed in Chapter 1, which would only limit the speed to the amount of parallel power available, which with modern hardware-accelerators, is very likely to be equal or greater than a factor 100 if implemented correctly.

2.4.1 On fast ray-polygon intersection techniques

The most common way to accelerate ray-hit search in a polygon-meshed 3D scene is to use hierarchical space partitioning systems. As discussed in section 1.3, the 3 important parts are the space partitioning technique, the construction algorithm, and the traversal algorithm.

The way that the space partitioning was implemented allows for a very versatile partitioning. The `SpacePartition` class is composed with a `TreeConstructor` which has different types of implementation, many of which are KD-tree techniques. With this method, it would be possible to easily implement a quad-tree or an oct-tree algorithm just by creating a new `TreeConstructor` realization and changing the instance in our `SpacePartition`. It was implemented this way because no prior knowledge on efficient space partitioning for ray tracing way known. This allowed for extensive testing of the performance of multiple algorithms.

The KD-tree The space partitioning technique that was selected is the "KD-tree" technique which is a sub-type of BSP, with a construction technique called the Surface Area Heuristic (SAH), and a traversal algorithm called sequential traversal. It was selected because of its

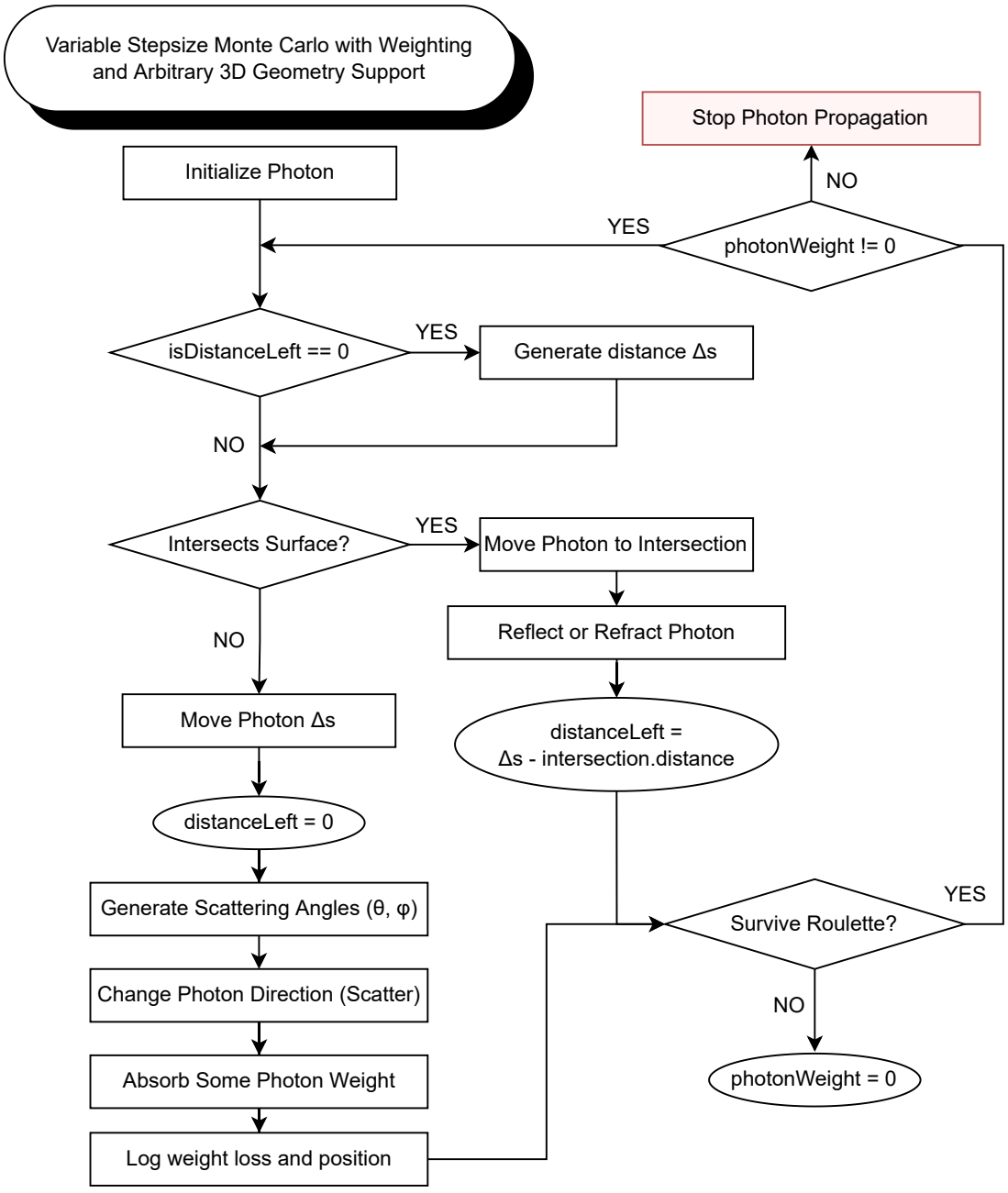


Figure 2.19: Flowchart of the photon propagation physics. This model was inspired by the MCML model with modifications oriented toward arbitrary 3D geometry support. The main difference with the previous model is the intersection management which is done in a completely separated step. Also, each photon launches this loop independently; this helps later with the parallelization of the code.

versatility, performance, and ease of implementation. In the ray-tracing domain, there is often no definitive best technique. Each technique has its own advantages and disadvantage; the selection only depends on the type of scene that the algorithm has to deal with and the expected results. From the literature review ([15, 32–39]) it was concluded that the KD-tree was a good alternative for ray-tracing in a biophotonics context since the scenes are likely to be mesh-dense with varying density and multiple objects, which the KD-tree handles relatively well. The "KD-tree" is short for "k-dimensional tree", which means this technique supports any number of dimensions.

Construction Algorithm The preferred construction technique for this work is based on the Surface Area Heuristic (SAH), a technique popularized in 1990 [37]. Canonically, the splitting axis rotates through the space dimensions with node depth, but for our purposes, this was not the most efficient technique, because the canonical algorithm generates a balanced tree. Balanced trees are good at splitting space equally, thus rendering a very consistent ray-hit hit search, because each branch of the tree has the same depth, thus reaching leaf nodes will always take the same amount of time. However, the tree generated with SAH is very good at delimiting and isolating multiple objects (see Figure 2.22), which is the main feature that was attractive. Equation 2.10 is the SAH cost function. It is this equation that will guide the positioning of the plane: the plane is translated into an axis until the cost function finds the global minimum. The implementation of the construction algorithm is extremely large and should be viewed on the GitHub repository.

$$C(A,B) = t_{\text{traversal}} + p_A \sum_{i=1}^{N_A} t_{\text{intersect}}\ (a_i) + p_B \sum_{i=1}^{N_B} t_{\text{intersect}}\ (b_i) \tag{2.10}$$

- $t_{traversal}$ represents the cost of traversing the node. It is usually a constant cost that has to be determined empirically.
- N_A and N_B are the number of triangles in regions A and B.
- a_i and b_i are the ith triangle of region A and B.
- $t_{intersect}$ represents the cost of intersecting with a polygon.
- p_A and p_B represent the probability of hitting a box's surface of volume A that is contained in space C. To compute, sum all the surface area of the volume A and divide by the area of the faces of volume C : $p_A = \frac{\sum \text{faces area}_A}{\sum \text{faces area}_C}$.

On the example shown in Figure 2.20, it is demonstrated the best way to minimize the cost function is to maximize the empty space and isolate groups of polygons. When the groups of polygons are all isolated, the algorithm continues to split the space with the same algorithm

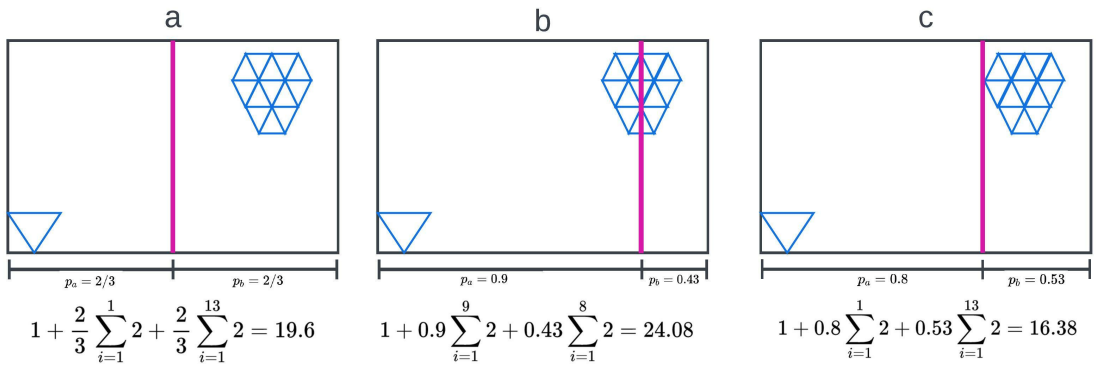


Figure 2.20: Comparison of different splitting planes with the SAH cost equation. (a) shows the basic binary split where space is divided into two sections. (b) shows the canonical KD-tree split where the split is positioned at the median of all polygons. (c) shows the optimal split that minimizes the SAH cost function. This illustrates that SAH tries to maximize empty space and confine groups of polygons.

until it reaches a stop condition. The PyTissueOptic's algorithm does a split plane search in 3 dimensions each time, thus also finding the best split axis as well.

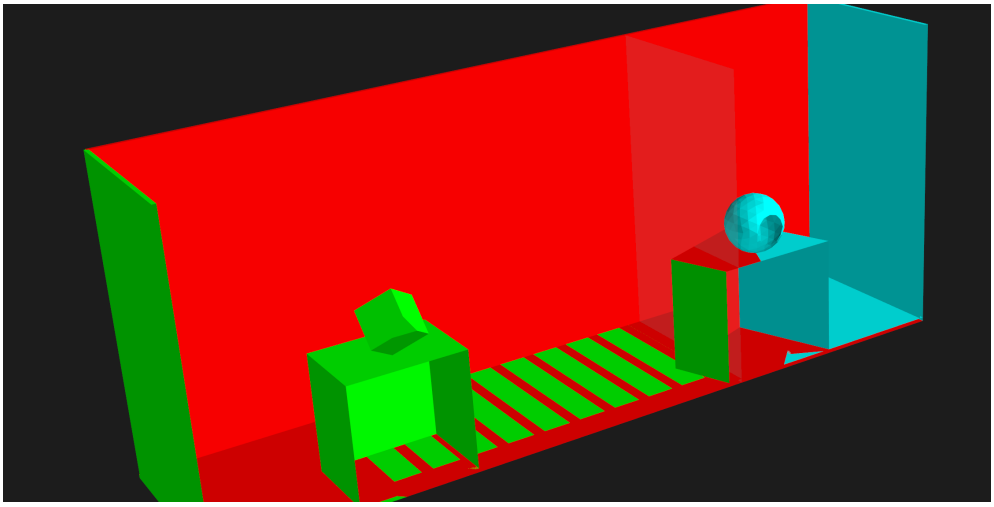


Figure 2.21: Splitting plane demonstration in 3-dimension. The splitting plane is the vaguely opaque plane. Green: left side of plane, goes to left node. Blue: right side of plane goes to right node. Red: touches the split plane, goes in both nodes.

Figure 2.21 illustrates a split plane search in a part of a 3D scene, rather than 2D. An example of a fully constructed 3D KD-tree with the PyTissueOptics algorithm is presented in Figure 2.22. You can observe that the axis-aligned objects are very well segmented by the SAH algorithm. The algorithm is also capable of splitting dense polygon mesh. in Figure 2.23, we observe the fine partitioning that occurs in a dense polygon-mesh. This is wanted behavior because even though the algorithm is good at isolating clusters of polygons if a ray hits the sphere, it still would be a computationally intense task to verify all those polygons. Thus, there is a need to partition finer. The stop condition of the splitting occurs when the cost

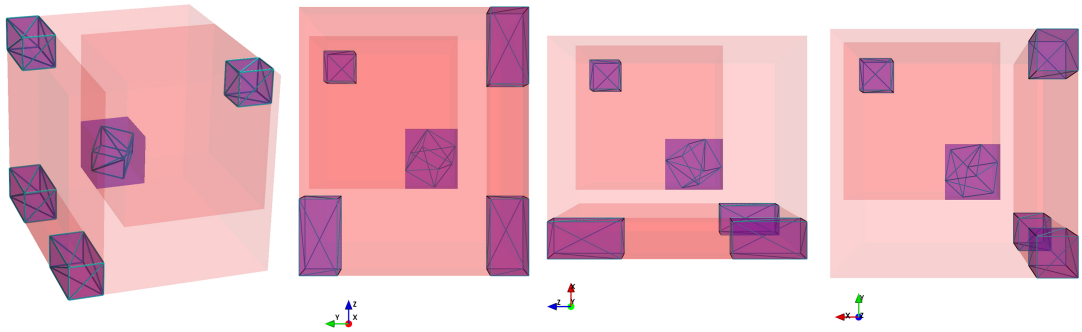


Figure 2.22: Representation of a partitioned 3D space as a KD-tree. Pink denotes the limits of a node's bounding box. The darker the color, the deeper the node is in the tree. The purple bounding box represents the leaf nodes. This is a simple scene to partition, but it illustrates clearly the isolation capabilities of the SAH technique.

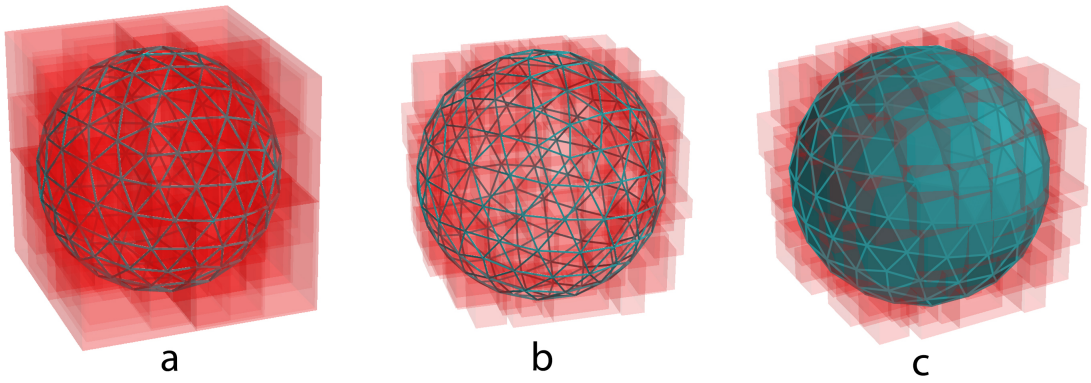


Figure 2.23: KD-tree of different scene complexities. (a) represents the complete KD-tree of an icosphere. The darker the color, the deeper the nodes are. (b) and (c) shows only the leaf nodes of the tree. (c) shows a solid sphere to enhance the contrast with the node bounding boxes.

function of splitting further is greater than the traversal cost of all the triangles in the node.

Many parameters have to be manually "fine-tuned". That is unfortunately common practice to avoid too complex algorithms. Here are parameters that have been determined empirically,

- Starting split axis is the widest side.
- 20 split plane tries for each axis
- Intersection cost was manually found to be 0.5 to give good results.
- maximum tree depth is 20 to avoid too long construction time
- minimum node polygon count is set to 6. If it reaches 6, the node will node split further.

Traversal Algorithm The traversal algorithm used is a sequential traversal algorithm, meaning it traverses the tree from the root, traveling up to the leaves. This code is a

recursive function that finds the closest intersection of a ray with a given node. If the node is a leaf, it will find the closest intersection of the ray with any of the polygons in the node. The `findClosestPolygonIntersection()` method uses a ray-polygon algorithm called MöllerTrumbore, which is detailed later on. If the node is not a leaf, it will check if it is worth exploring the node's children before doing so by looking at the intersection of the ray and the child node bounding box. This intersection check uses a fast ray-box algorithm which is detailed later as well. If the ray hits the child node, the algorithm will explore it. If an intersection is found, it will compare it to the closest intersection found so far and return the closest one.

```
1  def _findIntersection(self, ray: Ray, node: Node, closestDistance=sys.maxsize)
   ↪  -> Optional[Intersection]:
2      if node.isLeaf:
3          intersection = self._findClosestPolygonIntersection(ray, node.polygons)
4          return intersection
5
6      if not self._nodeIsWorthExploring(ray, node, closestDistance):
7          return None
8
9      closestIntersection = None
10     for child in node.children:
11         intersection = self._findIntersection(ray, child, closestDistance)
12         if intersection is None:
13             continue
14         if intersection.distance < closestDistance:
15             closestDistance = intersection.distance
16             closestIntersection = intersection
17
18     return closestIntersection
```

Ray-Box hit algorithm The ray-box intersection algorithm is used to find an intersection of a node's bounding box. This implementation of a fast ray-box algorithm was written by Andrew Woo [40] and is still known to be a very fast algorithm today. The algorithm is designed to quickly calculate whether or not a ray intersects with a given axis-aligned bounding box (AABB). The code first checks if the ray origin is inside the AABB. If so, it returns true and the hit point is set to the ray origin. Otherwise, the code calculates the T distances to the candidate planes. The candidate planes are the planes that define the edges of the AABB. The code then gets the largest of the T distances and uses that to calculate the final hit point. Finally, the code checks if the hit point is actually inside the AABB. If so, it returns true. Otherwise, it returns false.

```
def getIntersection(self, ray: Ray, bbox: BoundingBox) -> Union[Vector, None]:
    minCorner = [bbox.xMin, bbox.yMin, bbox.zMin]
    maxCorner = [bbox.xMax, bbox.yMax, bbox.zMax]
    origin = ray.origin.array
    direction = ray.direction.array
    # Find candidate planes (only depends on ray's origin)
    quadrant = [None, None, None]
    candidatePlanes = [None, None, None]
    inside = True
    for i in range(3):
        if origin[i] < minCorner[i]:
            quadrant[i] = self.LEFT
            candidatePlanes[i] = minCorner[i]
            inside = False
        elif origin[i] > maxCorner[i]:
            quadrant[i] = self.RIGHT
            candidatePlanes[i] = maxCorner[i]
            inside = False
        else:
            quadrant[i] = self.MIDDLE
    if inside:
        return ray.origin
    # Calculate distances to candidate planes
    maxT = []
    for i in range(3):
        if quadrant[i] != self.MIDDLE and direction[i] != 0:
            maxT.append((candidatePlanes[i] - origin[i]) / direction[i])
        else:
            maxT.append(-1)
    # Set plane as the one with largest distance.
    plane = maxT.index(max(maxT))
    # Check final candidate is inside box and construct intersection point
    hitPoint = [None, None, None]
    if maxT[plane] < 0:
        return None
    if ray.length and maxT[plane] > ray.length:
        return None
    for i in range(3):
        if i != plane:
            hitPoint[i] = origin[i] + maxT[plane] * direction[i]
            if hitPoint[i] < minCorner[i] or hitPoint[i] > maxCorner[i]:
                return None
        else:
            hitPoint[i] = candidatePlanes[i]
    return Vector(*hitPoint)
```

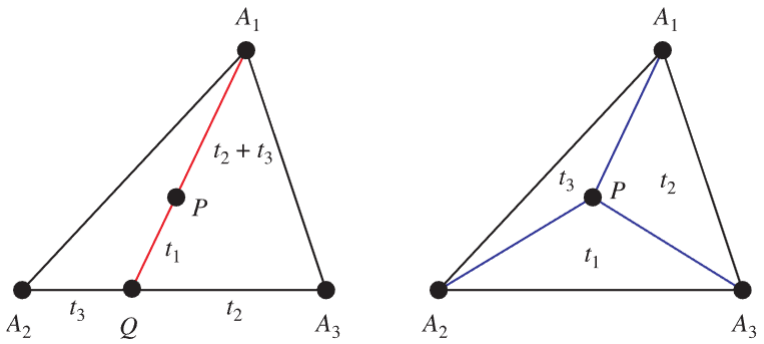


Figure 2.24: To find the barycentric coordinates for an arbitrary point P, find t_2 and t_3 from the point Q at the intersection of the line A_1P with the side A_2A_3, and then determine t_1 as the mass at A_1 that will balance a mass $t_2 + t_3$ at Q, thus making P the centroid (left figure). Furthermore, the areas of the triangles (right figure) ΔA_1A_2P, ΔA_1A_3P, and ΔA_2A_3P are proportional to the barycentric coordinates t_3, t_2, and t_1 of P [43].

Ray-Triangle hit algorithm The MöllerTrumbore algorithm [41] is an efficient method for calculating the intersection of a ray with a triangle. It uses a few optimizations to avoid having to calculate the intersection of the ray with the triangle's plane. Instead, it uses the triangle's vertices and the ray's direction to calculate the intersection.

A ray is usually defined as $\vec{R}(t) = \vec{O} + w\vec{D}$, where O is the origin vector, D is the direction vector and w is the scalar that scales D. As described in the section 2.2.1, a triangle is defined by three vertices A, B and C. The goal is to determine whether the ray hits inside the triangle defined by the three vertices. A point on a triangle can be represented by $P = t_1A + t_2B + t_3C$, where t_1, t_2, t_3 are the barycentric coordinates and A, B, C are triangle's vertices. This system was discovered by Mobius (1827) [42]. To have a point inside the triangle, the conditions $t_1 \geq 0, t_2 \geq 0$, $t_3 \geq 0$ and $t_1 + t_2 + t_3 = 1$ must be fulfilled. The coordinate value on a certain axis is related to the proximity of that point to the vertex compared to the other vertices. As seen in Figure 2.24, these value are proportional to the areas created by joining the vertices to the point P.

The intersection point of the ray and of the triangle can be written in barycentric form. Knowing $t_1 + t_2 + t_3 = 1$, we can substitute $t1 = 1 - t_2 - t_3$

$$O + tD = (1 - t_2 - t_3)A + t_2B + t_3C \tag{2.11}$$

This leads to the following equation written in matrix form:

$$\begin{bmatrix} -D, & B - A, & C - A \end{bmatrix} \begin{bmatrix} w \\ t_2 \\ t_3 \end{bmatrix} = O - A \tag{2.12}$$

Möller and Trumbore explain that the term $O - A$ can be looked at as a transformation moving the triangle from its original world space position to the origin (the first vertex of the

triangle coincides with the origin). The other side of the equation has for effect to transform the intersection point from x, y, z space to "w, t_1, t_2 space". Considering A, B, C, D has vectors (3 dimensions), we have a 3x3 matrix, which gives us a system of 3 equations and 3 vectors unknowns. The steps to solve this system of equations are detailed in [41].

The code below is an implementation of the MöllerTrumbore ray-triangle 3D intersection algorithm. The algorithm works by first calculating the determinant of a matrix made up of the triangle's vertices and the ray's direction. If the determinant is less than a certain threshold (EPSILON), then the ray is considered to be parallel to the triangle and no intersection is possible.

```
def _getTriangleIntersection(self, ray: Ray, triangle: Triangle) -> Optional[Vector]:
    """ MöllerTrumbore ray-triangle 3D intersection algorithm. """

    v1, v2, v3 = triangle.vertices
    edgeA = v2 - v1
    edgeB = v3 - v1
    pVector = ray.direction.cross(edgeB)
    determinant = edgeA.dot(pVector)

    rayIsParallel = abs(determinant) < self.EPSILON
    if rayIsParallel:
        return None

    inverseDeterminant = 1. / determinant
    tVector = ray.origin - v1
    u = tVector.dot(pVector) * inverseDeterminant
    if u < 0. or u > 1.:
        return None

    qVector = tVector.cross(edgeA)
    v = ray.direction.dot(qVector) * inverseDeterminant
    if v < 0. or u + v > 1.:
        return None

    t = edgeB.dot(qVector) * inverseDeterminant
    lineIntersection = t < self.EPSILON
    if lineIntersection:
        return None

    if ray.length and t > ray.length:
        return None

    return ray.origin + ray.direction * t
```

If the determinant is greater than the threshold, then the inverse of the determinant is calculated. This inverse is used to calculate the values of two vectors, t_1, t_2 (u, v in the code). These vectors represent the point of intersection between the ray and the triangle.

If either u or v is less than 0 or greater than 1, then the ray does not intersect the triangle. If both u and v are between 0 and 1, then the ray intersects the triangle. The value of t is then calculated. This value represents the distance from the ray's origin to the point of intersection.

If 't' is less than the EPSILON threshold, then the ray intersects the triangle's plane but not the triangle itself. If t is greater than the ray's length, then the ray does not intersect the triangle. Otherwise, the point of intersection is returned.

Intersection smoothing Because the 3D objects are defined as polygon meshes, the angle of interaction with an object's region is only dependent on the polygon's orientation. A given ray that hits the same polygon, but on different parts of the polygon, will have the same angle of incidence. This is called *flat shading*. Shading is the process of computing the interaction on the surface of an object depending on, but not limited to, the distance of the light source, the color of the light source, the material properties, the Bidirectional Reflectance Distribution Function (BRDF), and possibly other parameters depending on the implementation. Most real-life objects, even more so biological entities, tend not to have sharp edges. Thus, polygon meshing causes the problem of not correctly modeling the surface of a smooth object.

To circumvent this, it is possible to use interpolation functions to find a smoother incidence plane. If the intersecting polygon was prepared for smoothing (ie. it has vertex normals), the barycentric normal interpolation is used to correct the polygon normal depending on the ray hit position on the polygon. Barycentric coordinates have been defined in the last section as a weighted position that's dependent on the point's distance to the vertices. The smoothing algorithm first calculates the 3 barycentric weights for each vertex. Then, each vertex's normal is multiplied by the vertex's weight and then all the weighted normal are summed together. This resultant vector is the interpolated normal, which is normalized and then returned by the algorithm. Figure 2.25 illustrates the differences between flat shading and barycentric shading for an icosphere. Another advantage of using smoothing interpolation is the ability to reduce the number of polygons on an object without compromising the smoothing quality. As seen in Figure 2.25, the smoothing stays similar for every sphere order. Thus, knowing it is computationally non-intensive to calculate the barycentric normal interpolation, it is much better to use a low-order sphere to save on the computing time of the construction and traversal of the space partition complexity generated by more polygons.

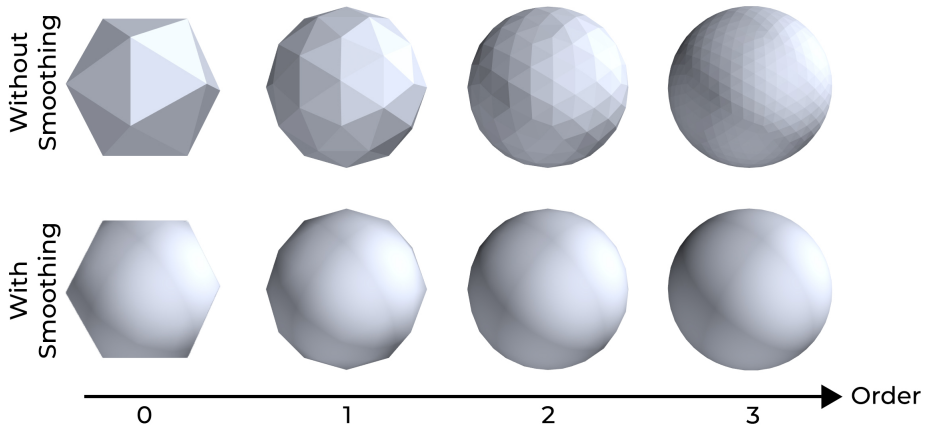


Figure 2.25: Different shading algorithms on an icosphere of increasing order. Flat shading (without smoothing) does not interpolate the polygon normal. Barycentric shading (with smoothing) uses barycentric interpolation to correct the polygon's normal at the point of intersection. Even with low-order icospheres, the barycentric shading creates a smooth sphere, only the contour's distance to the center is the sphere is ill-defined.

```
def getSmoothNormal(polygon: Polygon, position: Vector) -> Vector:
    if not polygon.toSmooth:
        return polygon.normal
    weights = _getBarycentricWeights(polygon.vertices, position)
    smoothNormal = Vector(0, 0, 0)
    for weight, vertex in zip(weights, polygon.vertices):
        smoothNormal += vertex.normal * weight
    smoothNormal.normalize()
    return smoothNormal

def _getBarycentricWeights(vertices: List[Vertex], position: Vector) ->
↪  List[float]:
    weights = []
    n = len(vertices)
    for i, vertex in enumerate(vertices):
        prevVertex = vertices[(i - 1) % n]
        nextVertex = vertices[(i + 1) % n]
        w = (_cotangent(position, vertex, prevVertex) +
             _cotangent(position, vertex, nextVertex)) / (position -
             ↪  vertex).getNorm() ** 2
        weights.append(w)
    return [w / sum(weights) for w in weights]
```

2.4.2 On photon propagation parallelization

To increase speed on calculations such as Monte Carlo, where every calculation is completely independent, an efficient method is to use parallelization. This requires multiple computational cores, which are usually found on multi-core CPU or GPU. It is common to use GPU as a hardware acceleration platform because of its very large number of cores. There are many specialized languages used to code parallel software, but the most versatile is OpenCL. In this section, the steps towards the parallelization of the photon's propagation algorithm are detailed and the code is explained.

2.4.3 Pseudo-random generator

Monte Carlo simulations require a high level of randomness to be able to work properly; that's how Monte Carlo works. Thus, in computer science, since everything is deterministic, there is a need to create random values from a deterministic process, and because of their deterministic nature and their repeatability, these algorithms are called "pseudo-random". In the OpenCL language, as of version 2.0, no pseudo-random generators are available. Thus, it was necessary to do the appropriate research to code one.

A simple pseudo-random algorithm was found called the Wang Hash-Function, which generates a random integer from an integer input.

```
1  uint wangHash(uint seed){
2      seed = (seed ^ 61) ^ (seed >> 16);
3      seed *= 9;
4      seed = seed ^ (seed >> 4);
5      seed *= 0x27d4eb2d;
6      seed = seed ^ (seed >> 15);
7      return seed;}
8  float getRandomFloatValue(__global unsigned int *seedBuffer, unsigned int id){
9       float result = 0.0f;
10      while(result == 0.0f){
11          uint rnd_seed = wangHash(seedBuffer[id]);
12          seedBuffer[id] = rnd_seed;
13          result = (float)rnd_seed / (float)UINT_MAX;
14          }
15      return result;
16      }
17   __kernel void fillRandomFloatBuffer(__global unsigned int *seedBuffer,
     ↪  __global float *randomFloatBuffer){
18     int id = get_global_id(0);
19     randomFloatBuffer[id] = getRandomFloatValue(seedBuffer, id);}
```

The code above is a kernel function to fill a buffer with random float values. The function takes two arguments - a seed buffer and a float buffer. The seed buffer is used to generate the random float values. The floating buffer is where the generated float values are stored. The kernel function works by first getting the id of the current work-item. It then uses the id value to generate a random float value using the Wang Hash function. The id is the id number of the thread The generated float value is then stored in the float buffer. The randomly generated seed then replaces the old value in the seed buffer, so that the next value is randomly chosen as well, not simply sequentially.

2.4.4 OpenCL C propagation algorithm

The propagation algorithm in OpenCL C[1] is almost the same as the Python algorithm. Here are some key differences:

- Because the intersections are not implemented yet, there is no need to check for a `distanceLeft` as the algorithm will generate a new propagation distance for each iteration.

```
1  __kernel void propagate(uint dataSize, float weightThreshold, __global
   ↪  photonStruct *photons,
2  __constant materialStruct *materials, __global loggerStruct *logger, __global
   ↪  float *randomNums,
3  __global uint *seedBuffer){
4      uint gid = get_global_id(0);
5      uint stepIndex = 0;
6      uint logIndex = 0;
7      float g = materials[0].g;
8      float mu_t = materials[0].mu_t;
9      float4 er = getAnyOrthogonalGlobal(&photons[gid].direction);
10     photons[gid].er = er;
11
12     while (photons[gid].weight >= weightThreshold){
13         logIndex = gid + stepIndex * dataSize;
14         randomNums[gid] = getRandomFloatValue(seedBuffer, gid);
15         float distance = getScatteringDistance(randomNums, mu_t, gid);
16         moveBy(photons, distance, gid);
17         randomNums[gid] = getRandomFloatValue(seedBuffer, gid);
18         float phi = getScatteringAnglePhi(randomNums, gid);
19         randomNums[gid] = getRandomFloatValue(seedBuffer, gid);
20         float g = materials[0].g;
21         float theta = getScatteringAngleTheta(randomNums, g, gid);
22         scatterBy(photons, phi, theta, gid);
23         interact(photons, materials, logger, gid, logIndex);
24         stepIndex++;}}
```

[1] All the functions and kernels used by the `propagate` kernel can be found in Appendix E and F.

- There is no object in OpenCL C. The closest that can be done to mimic a class acting as a state container is to use structs. In order to reduce complexity, all the photons state were compiled into a `photonStruct` type. The applies for `materialStruct` and `loggerStruct`.
- Because of the way OpenCL C works, and in order to take advantage of the GPU parallelism without the need to micro manage memory, global memory blocs are allocated. The prefix `__global`, when placed before an input variable, serves as an indicator to OpenCL to go search in the global memory. Here, the memory for the photon's state (`photons`, the materials properties (`materials`), the logging data (`logger`), the random numbers (`randomNums`) and the seeds for the random generator (`seedBuffer`)

2.4.5 PyOpenCL Python API management

The PyOpenCL API controls most of the workflow of the parallelization. This encapsulates the creation of the custom struct types, the allocation of the memory blocks, the values initializations, the data copy to and from the GPU, and the control of the kernel's execution.

In order to provide a seamless interface to the user, the hardware acceleration is hidden in a `useHardwareAcceleration` attribute of the `Source` class, initialized at the object's construction. The `Source` realizations return photon's positions and directions as Numpy arrays, which are used to fill the photonStruct. The object `CLPhotons` is the entity in charge of the PyOpenCL API workflow. If the `useHardwareAcceleration` is set to `True`, the methods used by the `Source` to create the photons and launch propagation are dynamically changed to `loadPhotonsOpenCL` and `propagateOpenCL`.

```
1 class Source:
2     def propagate(self, scene: RayScatteringScene, logger: Logger = None,
      ↪  showProgress: bool = True):
3         if self._useHardwareAcceleration:
4             self._propagateOpenCL(scene, logger)
5 
6     def _loadPhotons(self):
7         if self._useHardwareAcceleration:
8             self._loadPhotonsOpenCL()
9 
10    def _propagateOpenCL(self, scene: RayScatteringScene, logger: Logger = None):
11        self._photons.prepareAndPropagate(scene, logger)
12 
13    def _loadPhotonsOpenCL(self):
14        positions, directions = self.getInitialPositionsAndDirections()
15        self._photons = CLPhotons(positions, directions, self._N)
```

When the `prepareAndPropagate` method of the CLPhotons instance is launched, all the API workflow is in motion. Here are the key steps:

1. At `CLPhotons` initialization, the OpenCL `Context` and `CommandQueue` are created and the appropriate device is selected. Then, the new struct types are created. Here's the photonStruct type creation:

```
def makePhotonType(device: 'cl.Device'):
    photonStruct = np.dtype(
        [("position", cl.cltypes.float4),
         ("direction", cl.cltypes.float4),
         ("er", cl.cltypes.float4),
         ("weight", cl.cltypes.float),
         ("material_id", cl.cltypes.uint)])
    name = "photonStruct"
    photonStruct, c_decl_photon = cl.tools.match_dtype_to_c_struct(device,
    ↪  name, photonStruct)
    photon_dtype = cl.tools.get_or_register_dtype(name, photonStruct)
    return photon_dtype, c_decl_photon
```

2. Then, the memory is allocated for all the variables of the simulation, and empty memory blocks are allocated for all the results to be written. Again, here is an example of the photons. The same steps are performed for the material properties, the logger, and the random numbers.

```
def _makeBuffers(self):
    self._makePhotonsBuffer()
    self._makeMaterialsBuffer()
    self._makeLoggerBuffer()
    self._makeRandomBuffer()

def _makePhotonsBuffer(self):
    photonsPrototype = np.zeros(self._N, dtype=self._photon_dtype)
    photonsPrototype = rfn.structured_to_unstructured(photonsPrototype)
    photonsPrototype[:, 0:3] = self._positions[:, ::]
    photonsPrototype[:, 4:7] = self._directions[:, ::]
    photonsPrototype[:, 12] = 1.0
    photonsPrototype[:, 13] = 0
    self._HOST_photons = rfn.unstructured_to_structured(photonsPrototype,
    ↪  self._photon_dtype)
    self._DEVICE_photons = cl.Buffer(self._context,
    cl.mem_flags.READ_WRITE | cl.mem_flags.COPY_HOST_PTR,
    hostbuf=self._HOST_photons)
```

3. Third, the OpenCL C code has to be compiled and the new types have to be declared to the OpenCL Context.

```
def _buildProgram(self):
    randomSource = open(os.path.join(self._sourceFolderPath,
    ↪  "random.c")).read()
    vectorSource = open(os.path.join(self._sourceFolderPath,
    ↪  "vectorOperators.c")).read()
    propagationSource = open(os.path.join(self._sourceFolderPath,
    ↪  "propagation.c")).read()

    self._program = cl.Program(self._context, self._c_decl_photon +
    ↪  self._c_decl_mat +
    self._c_decl_logger + randomSource + vectorSource +
    ↪  propagationSource).build()
```

4. Finally, propagation is launched and the memory has to be copied to the computing device (GPU), and post execution of the kernels, the memory has to be copied back to the CPU. The logging data is then sent to a Logger object so that the Stats object can carry on normally with its tasks.

```
def _propagate(self):
    datasize = np.uint32(len(self._HOST_photons))
    self._program.propagate(self._mainQueue, self._HOST_photons.shape, None,
    ↪  datasize,
    self._weightThreshold, self._DEVICE_photons, self._DEVICE_material,
    self._DEVICE_logger, self._DEVICE_randomFloat, self._DEVICE_randomSeed)
    self._mainQueue.finish()
    cl.enqueue_copy(self._mainQueue, dest=self._HOST_logger,
    ↪  src=self._DEVICE_logger)
    log = rfn.structured_to_unstructured(self._HOST_logger)
    self._logger.logDataPointArray(log, InteractionKey("universe", None))
```

Memory management and saturation Compared with the python version, the increase of performance is substantial for the same scene configurations by a factor of close to 500x on a *GTX 1060 3GB*, which is, as of today, considered a very deprecated GPU. Modern GPU hardware has between 12GB and 24GB of VRAM and the latest consumer-grade RTX 3090Ti 24GB has 10752 cores compared with 1280 cores for the former. Because of the need to pre-allocate memory for the logging data, this results in a rapid saturation of the memory. When memory is saturated, it has to be copied to the CPU and emptied. Then, another *batch* of photons can be launched. This is called batching and it is a bottleneck in the simulation speed as making memory copy from GPU to CPU creates a considerable delay (usually a couple of *ms*). If one has to make many batches, this copy delay can become quite considerable, so a bigger memory is an advantage. With the RTX 3090Ti, we could estimate the propagation speedup to be at least 10x faster than with a GTX 1060.

Chapter 3

Results

3.1 Python 3D Graphics Framework

The geometry domain has been contained in a separate package called "3D Graphic Framework" which allows to create 3 dimensional arbitrarily complex objects and scenes. Basic geometrical shapes are provided as well as the ability to upload 3D models from external files, such as .obj. The package is built with the intent of logging data on different surfaces or volumes for scientific purposes, so the surface and solid definition allows labels to be given to groups of polygons, objects, or groups of objects. Data related to these different objects can be logged with the Logger object. It also includes a high-performance space-partitioning algorithm and intersection finder which can be used for any polygon-intersection task. This section describes in detail the use of the most important parts of the framework and shows some usage examples.

3.1.1 Overview of the package

- Create `Solids`: `Cube`, `Cuboid`, `Sphere`, `Ellispoid`, `Cylinder`, `Cone`.
- Transforms `Solids`: `translateTo`, `translateBy`, `rotate`, `scale`
- Stack `Cuboids` together on an axis with a similar width.
- Allow full confinement of `Solids` within `Solids`.
- Group `Solids` in a `Scene` with a common ambient `Material`.
- Ask to find `intersection` between `Ray` and any `Polygon` in the scene using a fast KD-tree partitioning method.
- Log, save and retrieve data (point, scalar, vector) and associate it with a specific solid and specific surface within that solid with `Logger` object.
- View `Solid` models and logged data in 3D space with `MayaviViewer` with a GUI and fully customizable visualization.

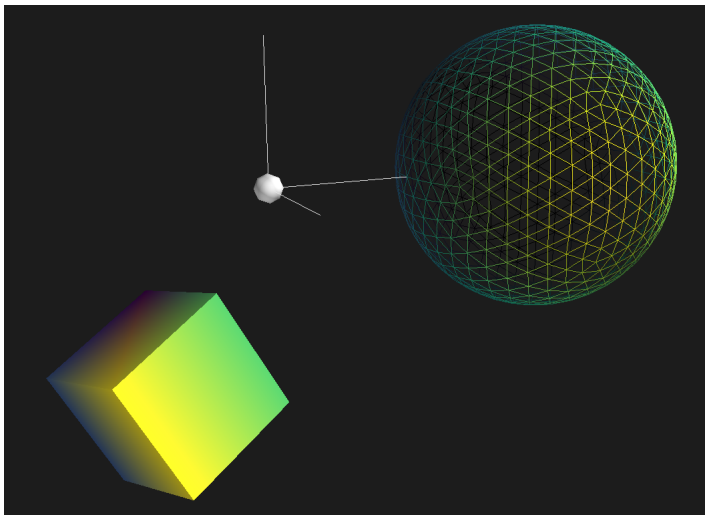


Figure 3.1: Basic scene demonstration for the 3D Graphic Framework. A sphere, a rotating cube, and the X, Y, and Z axes are displayed using the MayaviViewer class. The sphere is displayed as a wireframe and the cube as a solid surface.

3.1.2 Basic usage

A basic example is explained in detail to grasp the general usage of the package. The code below and Figure 3.1 depict a usual use of the package.

```
1  from pytissueoptics.scene import Cuboid, Sphere, Vector, MayaviViewer
2  import numpy as np
3
4  sphere = Sphere(radius=1, position=Vector(2, 0, 0))
5  cube = Cuboid(a=1, b=1, c=1, position=Vector(-1, -1, -1))
6  cube.rotate(0, 20, 30, rotationCenter=Vector(0, 0, 0))
7  viewer = MayaviViewer()
8  viewer.add(sphere, representation="wireframe", lineWidth=1, showNormals=False)
9  viewer.add(cube, representation="surface", lineWidth=5, showNormals=False)
10 viewer.addPoints(np.array([[0, 0, 0]]), scale=0.2)
11 viewer.addSegments(np.array([[0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 1, 0], [0, 0, 0,
   ↪  0, 0, 1]]))
12 viewer.show()
```

First, importing the necessary objects, `Sphere` and `Cuboid` which are 3D meshed objects, the `Vector` class which is used to point in 3D space in the scene, `MayaviViewer` which is used to display the 3D objects and data. First, the creation of the meshed objects requires some attributes such as dimensions and position. It is possible to apply transformations, such as rotation. Then, the objects can be added to the `MayaviViewer` with the `add` method, providing the display details such as the representation type, linewidth, etc. Adding points or

segments in 3D space is also simple with the `addPoints` and `addSegments`. Finally, invoke the Mayavi display with the `show` method.

This summarizes the usual usage of the graphic framework to create scenes. More specialized features will be discussed in the following sections, such as 3D model import, using the `IntersectionFinder`, logging data.

3.1.3 External 3D model import

To load external 3D files, the user must invoke the `loadSolid` function and provide a wavefront `filepath` at the object's creation. The `loadSolid` uses a file parser to get the relevant information from the file and a `Loader` object converts this information to a `Solid` objects. For now, the only available parser is the wavefront files (.obj) parser, but with this architecture, any other file parser could be added as an extension. The resultant object is a `Solid`, which means it is possible to give a `Material`, to set the smoothing on imported objects, and to apply any transform.

```
1  from pytissueoptics.scene import loadSolid, MayaviViewer
2  solid = loadSolid("rat.obj")
3  viewer = MayaviViewer()
4  viewer.add(solid, representation="surface")
5  viewer.show()
```

The shown code loads a rat wavefront file and displays it using the Mayavi viewer. The representation mode is set to 'surface'. This results in the Figure 3.2.

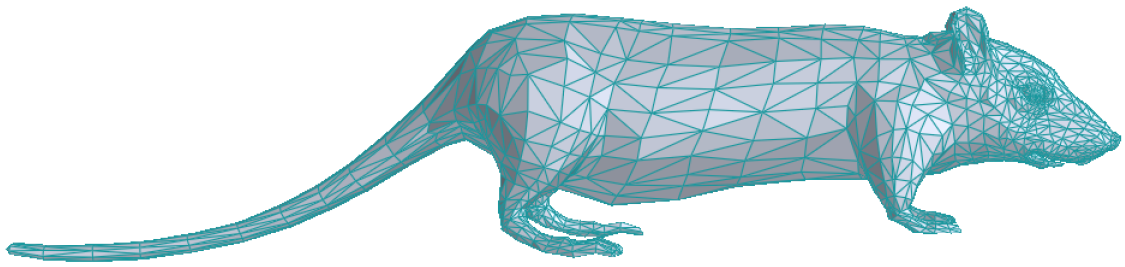


Figure 3.2: Rat 3D model imported from a wavefront object (.obj).

3.1.4 Data Viewer

With the `MayaviViewer` object it is possible to display 3D objects from the Graphics Framework. Here are the main functionalities: the `.add` to add solid instances, `.addScene` to add whole scenes, `.addLogger` to add the data of a `Logger` object, `.show` to open the Mayavi display. It is possible also to directly influence the Mayavi properties in python by using the object returned by the viewer when using the add method. Modification of the Mayavi pipeline is shown in the code example below, which results in Figure 3.3.

```
1  from pytissueoptics.scene import Ellipsoid, MayaviViewer
2
3  solid = Ellipsoid(3, 1, 1, smooth=True)
4  viewer = MayaviViewer()
5  s1 = viewer.add(solid, representation="surface")[0]
6  scene = s1.parent.parent.parent.parent
7  scene.scene.background = (0.4, 1, 1)
8  s1.actor.property.color = (0.8, 0.9, 0.9)
9  s1.actor.property.edge_visibility = True
10 s1.actor.property.line_width = 0.5
11 s1.actor.property.edge_color = (250 / 255, 2 / 255, 15 / 255)
12 s1.actor.property.interpolation = "gouraud"
13 viewer.show()
```

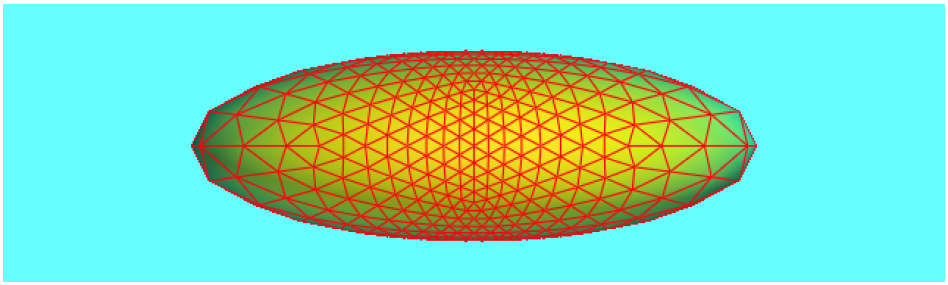


Figure 3.3: Ellipsoid viewed with Mayavi with modified display parameters. The color of the surfaces is yellow, the edges are red and the background is magenta.

3.1.5 Data Logging

Data logging is an important part of the Graphic Framework. The way it was implemented allows the user to select an `InteractionKey` to tie the data to a solid or a surface. It will then be easier to ask for the data that is tagged by the specific label when calculating metrics in any simulation in later plugins that use this package.

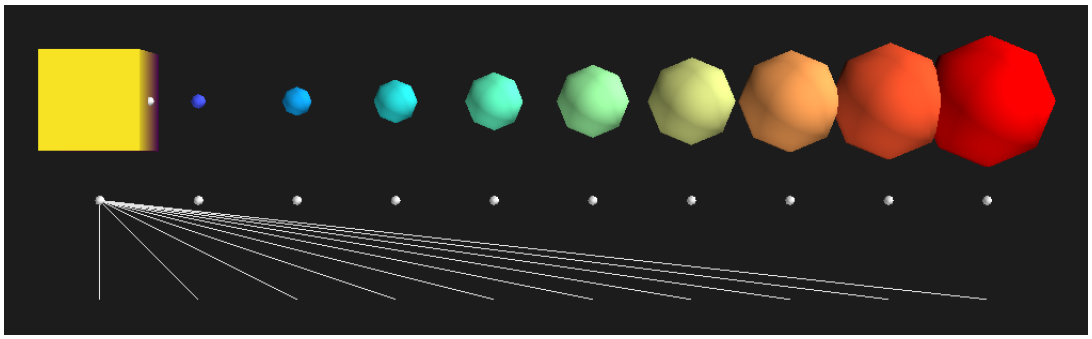


Figure 3.4: Showcase of the data logging possibilities. The white dot on the cube is linked to the solid via the use of a solid label while logging. The colored balls are scalar data saved at a position. Size is set to scale with value. The white points are just position data without any scalar value attached. The lines show segments logged which require an origin and an endpoint.

```
from pytissueoptics.scene import Cuboid, Vector, Scene, Logger, MayaviViewer
from pytissueoptics.scene.logger import InteractionKey

cube = Cuboid(a=1, b=1, c=1, position=Vector(0, 0, 0))
myScene = Scene([cube])
myLogger = Logger()
myLogger.logPoint(Vector(0.5, 0, 0),
↪  key=InteractionKey(solidLabel=cube.getLabel()))
for i in range(10):
    myLogger.logDataPoint(i, Vector(i, 0, 0), key=None)
    myLogger.logPoint(Vector(i, -1, 0), key=None)
    myLogger.logSegment(Vector(0, -1, 0), Vector(i, -2, 0), key=None)

viewer = MayaviViewer()
viewer.addScene(myScene, representation="surface", lineWidth=1,
↪  showNormals=False)
viewer.addLogger(myLogger, pointScale=0.1)
viewer.show()
```

3.1.6 Intersection Finder Example

The `IntersectionFinder` is an entity that requires a Scene as an input to dissect. According to the implementation used, the Scene will be dissected differently. The class only possesses a single public method which is `findIntersection` and requires to be given a Ray. It will then return an `Intersection` DTO containing the intersection information, such as the angle of incidence, the polygon reference, the normal, the distance left to hit, etc. `IntersectionFinder` can be instantiated from the complex 3D scene. Figure 3.6 shows the

segmentation of a complex 3D environment of non-homogeneous polygon density, achieved by the `FastIntersectionFinder`, which is the KD-tree based implementation.

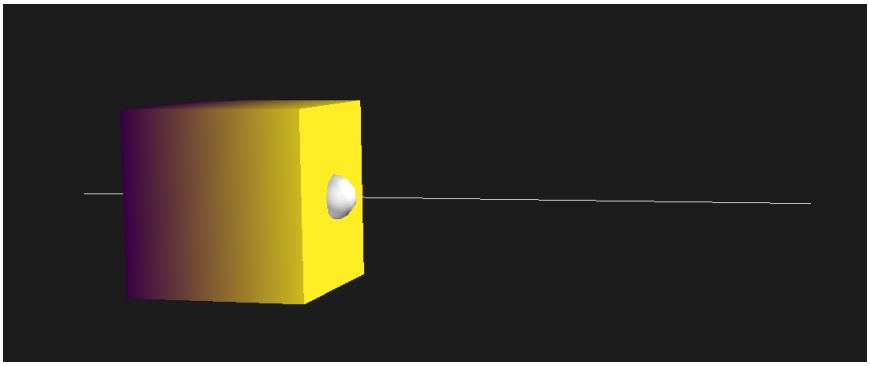


Figure 3.5: A cube is it by a ray shown as a line and its intersection point the one of the cube's polygon is displayed as a sphere.

Here is an example of the intersection finder usage.

```
1  from pytissueoptics.scene import Cuboid, Vector, Scene, Logger, MayaviViewer
2  from pytissueoptics.scene.intersection import FastIntersectionFinder, Ray
3  from pytissueoptics.scene.logger import InteractionKey
4
5  cube = Cuboid(a=1, b=1, c=1, position=Vector(0, 0, 0))
6
7  myScene = Scene([cube])
8  myLogger = Logger()
9  myIntersectionFinder = FastIntersectionFinder(myScene)
10
11 myLonelyRay = Ray(origin=Vector(0, 0, 3), direction=Vector(0, 0, -1), length=4)
12 foundIntersection = myIntersectionFinder.findIntersection(myLonelyRay)
13 myLogger.logSegment(myLonelyRay.origin, myLonelyRay.origin+
14 myLonelyRay.direction*myLonelyRay.length, key=InteractionKey(None, None))
15 myLogger.logPoint(foundIntersection.position,
   ↪  key=InteractionKey(solidLabel=cube.getLabel(),
16  surfaceLabel=foundIntersection.surfaceLabel))
17
18 viewer = MayaviViewer()
19 viewer.addScene(myScene, representation="surface", lineWidth=1,
   ↪  showNormals=False)
20 viewer.addLogger(myLogger, pointScale=0.25)
21 viewer.show()
```

First, the example imports the necessary components to create the required entities. The cuboid is created and is passed in a Scene. With this scene, it is possible to create an `IntersectionFinder`. To interact with it, a `Ray` pointing towards the cube is created.

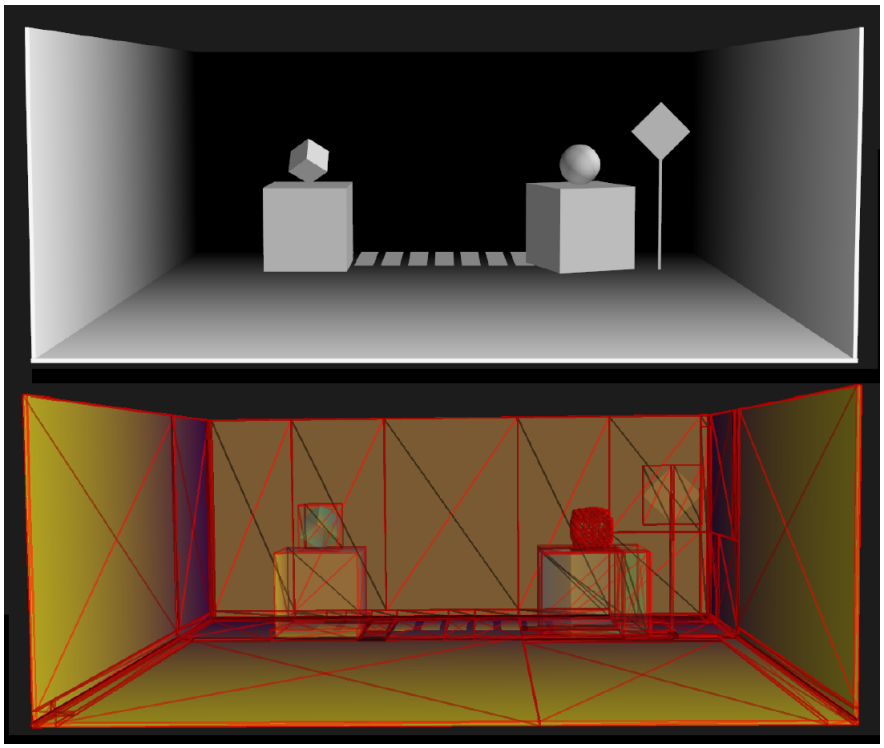


Figure 3.6: Display of a complex 3D scene with different levels of mesh complexity and of its KD-tree structure overlapped with the scene. The KD-tree has a higher node density in the regions with high mesh density, like around the sphere, and very large nodes for large objects, like walls. The objects are also well segmented, this means this KD-tree will probably be swift at computing a ray-hit search.

Then it is asked to find if the given ray hit a polygon. It does so it returns an `Intersection`. In order to verify the hit point is correct, a viewer is created and the ray, the solids, and the hit point are displayed as seen in Figure 3.5.

3.2 PyTissueOptics Light Simulation Plugin

The physics domain has been contained in a separate package called "rayscattering" which allows to propagate of light in 3D scenes using the Monte Carlo technique. The package allows the user to define tissues and propagate in them. The user can define scattering materials and apply the material to the 3D objects created. The package has different types of sources and allows the user to create its own sources. Finally, it manages the display of the simulation results as text, 2d graphs, and 3D data.

3.2.1 Basic usage

A basic example is explained in detail to grasp the general usage of the package. The code below and Figure 3.7 depict a usual use of the package. Here is simulated the propagation of a pencil source through a custom tissue called `PhantomTissue`, which is a `RayScatteringScene` child. This tissue is composed of a stacked cuboid with 3 layers of different materials. The `Logger` object serves to log the data during the simulation. Then, the source is selected. Here a `PencilPointSource` is used, which is a source where all the photons are propagated in the same direction and from the same point. Then, photons from the source are propagated in the tissue and the energy deposition is logged. A `Stats` object is created which will use the logger data to display information and calculate metrics. A report from the `Stats` object is asked. The distribution of the energy in the tissue is shown in 3D. In these real examples, the import statement will be removed as it is always the same.

```
from pytissueoptics import *

logger = Logger()
tissue = tissues.PhantomTissue()
source = PencilPointSource(position=Vector(0, 0, -1), direction=Vector(0, 0,
↪  1), N=2000)

source.propagate(tissue, logger=logger)

stats = Stats(logger, source, tissue)
stats.report()
stats.showEnergy3D()
```

3.2.2 RayScatteringScene

The `RayScatteringScene` object is a child of the `Scene` object from the graphics framework. Its purpose is to hold information on how the user wants to display the scene.

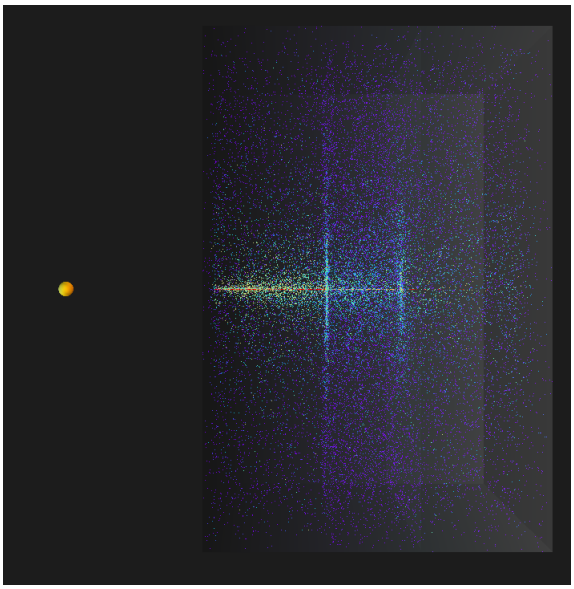


Figure 3.7: 3D representation of the energy deposition of a propagated PencilSource in a 3-layered tissue. The higher energy deposition is represented by the red dots and the less energy by the purple dots.

Also, it serves as an abstract class so that users can create children with their corresponding geometries and materials. A user could take a `Scene` object and it would work the same way, but the `Stats` object requires a `RayScatteringScene` because it requires a `addToDisplay` method, which is what tells the stat object how the objects are to be displayed (color, edges, representation, etc.).

3.2.3 Sources

The `Source` is an interface that is in charge of the creation and propagation of the photons. Different sources have been implemented as default to allow the user to simulate different types of illumination. Position and direction have to be presented as `Vector`. These sources are listed below with a short description (see Figure 3.8):

- `IsotropicSource`: Photons originate from the same position, direction is random on a 4π solid angle.
- `PointPencilSource`: Photons originate from a single point, all photons point in the same direction.
- `DirectionalSource`: Photon's positions are evenly distributed on a circle of a certain diameter and they all point to the same direction.
- `DivergentSource`: Photon's positions are evenly distributed on a circle of a certain diameter and their directions are evenly distributed in a cone of angle. The maximum angle is expressed in radian.

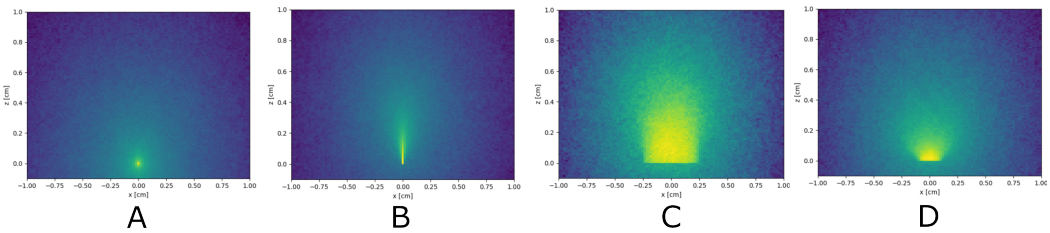


Figure 3.8: Source types available in the PyTissueOptics package. Sources of 20k photons are propagated in a tissue with optical properties (μs = 30, μa = 0.1, g = 0.9) with no interfaces. A) Isotropic source, B) Point Pencil source, C) Directional source, D) Divergent source.

At initialization, the user has to choose if the `Source` will be hardware accelerated. This is passed as an argument `useHardwareAcceleration`. The `Source` only has `.propagate()` as a public method, which requires a `RayScatteringScene` to propagate into and a `Logger` to save the propagation progress.

3.2.4 Stats

The `Stats` object is the entity responsible to take the logging data and produce human-readable data. It is also responsible to summon the `Viewer` class from the graphics framework to display the data to the user. This class is a little bit difficult to understand as it has to deal with the data of each solid and of each surface independently. The `Stats` object first need a `Logger` instance and a `RayScatteringScene` instance. In the logger, the data is stored and each data point has a `solidLabel` and a `surfaceLabel` which can be None. This way, it is possible to link parts of data to a specific solid and a specific surface or group of polygons. Thus, it is much easier to retrieve the data on a single instance or compute statistics for only a surface or a solid. On Figure 3.9 is showcased different modalities of representation. Here is a summary of the possible display methods of the `Stats` object.

- `.showEnergy3DOfSolids()`: takes no argument, shows everything.
- `.showEnergy3DOfSurfaces()`: takes a `solidLabel` as an argument. Will show all the surface data of that specific object.
- `.showEnergy3D()`: takes a `solidLabel, surfaceLabel` as arguments. If None is provided, it will 3D show all the solids and surfaces.
- `.showEnergy2D()`: integrates data along the provided `projection` axis to display a 2d representation of the energy deposition. The user can provide a `bin` number to bin the data, `limits` to change the region of the displayed data, `logScale` to show energy deposition on a logarithmic scale.

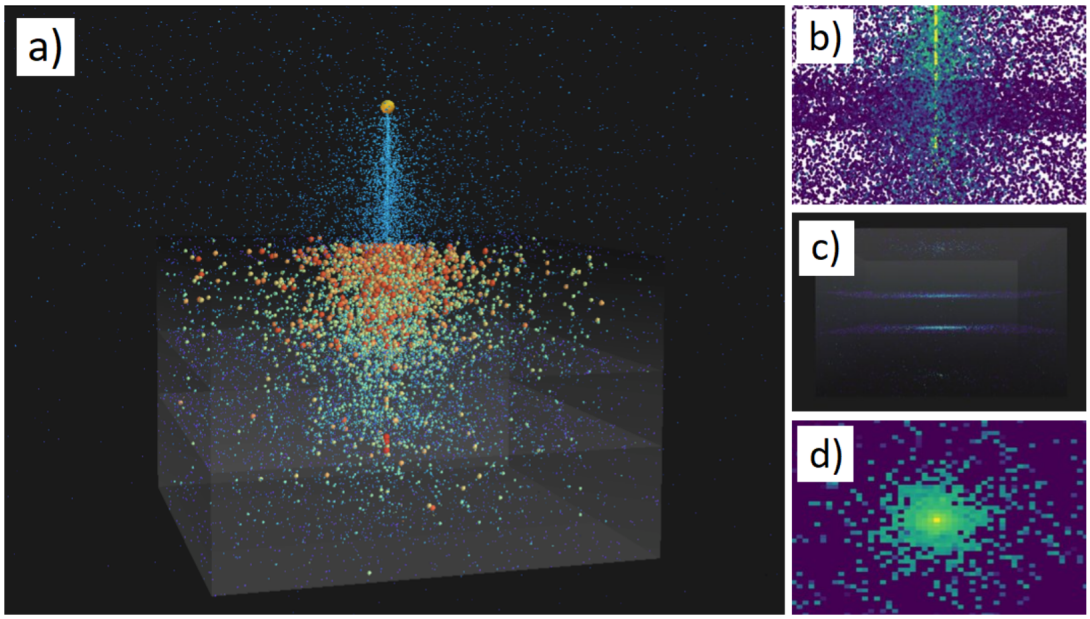


Figure 3.9: Possible visualizations given by the Stats object. (a) 3D display of all data and solids, (b) a 2D projection of the whole tissue stack, (c) a 3D display of surface intersections, and (d) a 2D binned display of intersections at the first interface.

- `.showEnergy1D()`: integrates data on the provided `along` axis. Data in the plane formed by the 2 perpendicular vectors to the axis is summed. This results in a 1-dimensional graph of the energy deposited along the selected axis.
- `.report()`: display a report of transmittance, reflectance, and absorption of all objects in the scene.

3.3 Validation of the propagation model

For this work, the validation of the model will focus on its MCML capabilities, which are designed for simulating light transport in multi-layered tissue structures. To evaluate the model, we will replicate scenarios in MCML, MCX, and PyTissueOptics (PTO), allowing us to compare our results not only to the base model but also to other modern open-source tools.

Scene 1						**Absorption (%)**		
Layer #	ua (cm-1)	us (cm-1)	g	n	width (cm)	MCML	MCX	PTO
Layer 1	0.1	20	0.7	1.3	1.0	24.78	25.19	24.87
Layer 2	10	2	0.9	1.4	0.5	19.32	19.56	22.32
Layer 3	1	200	0.95	1.5	1.0	0.03	0.00	0.04
Scene 2								
Layer 1	0.5	1.0	0.7	1.3	1.0	49.5	50.29	49.42
Layer 2	0.8	5.0	0.8	1.4	1.0	37.34	37.95	37.26
Layer 3	2.5	50	0.9	1.5	1.0	8.51	8.67	8.59
Scene 3								
Layer 1	1	100	0.9	1.37	0.1	26.12	26.85	25.86
Layer 2	1	10	0	1.37	0.1	14.86	15.25	15.09
Layer 3	2	10	0.7	1.37	0.2	23.13	23.70	24.21
Scene 4								
Layer 1	1	100	0.9	1.37	0.1	19.30	19.82	19.19
Layer 2	100	10	0.9	1.39	0.01	49.73	50.91	49.75
Layer 3	2	10	0.9	1.35	0.1	5.18	5.31	5.59

Table 3.1: Description of the scenes used for the validation and results of the mean absorption percentages for each algorithm at each layer. The standard deviation for each absorption value is $2\sigma < 0.1\%$ (N=10). A 1M photons count was used to ensure low standard deviation with different seeds for each algorithm.

These scenes were designed to encompass a broad range of μ_a, μ_s, g, and n values typical of biological tissues, according to Steven L Jacques' review [44]. While optical properties change with wavelength, the ranges provided here apply specifically to the 500 nm wavelength, which is well-documented in the review. Value of μ_a range from [0.001 100], μ_s from [5, 100], g from [0.1, 1] but is mostly concentrated over 0.5 and n from [1.33, 1.514]. These ranges are approximation derived from this review. The objective is to obtain a general portrait of the validity over a broad range; the exactitude of the range is not paramount.

The difference observed between the MCML module and MCX versus PyTissueOptics are within the same order of magnitude. The largest deviation from the MCML algorithm comes from the PyTissueOptics results on Scene 1, layer 2, which shows a 2.76% difference with the

reference model. The second highest deviation is from MCX, on Scene 4, layer 2, which shows a 1.18% difference. These results highlight that the models used are not perfectly equal and inherently have some degree of incertitude. This can be explained by differences in the basis of their algorithms and by the details in the implementation, like rounding error, parameters and thresholds used.

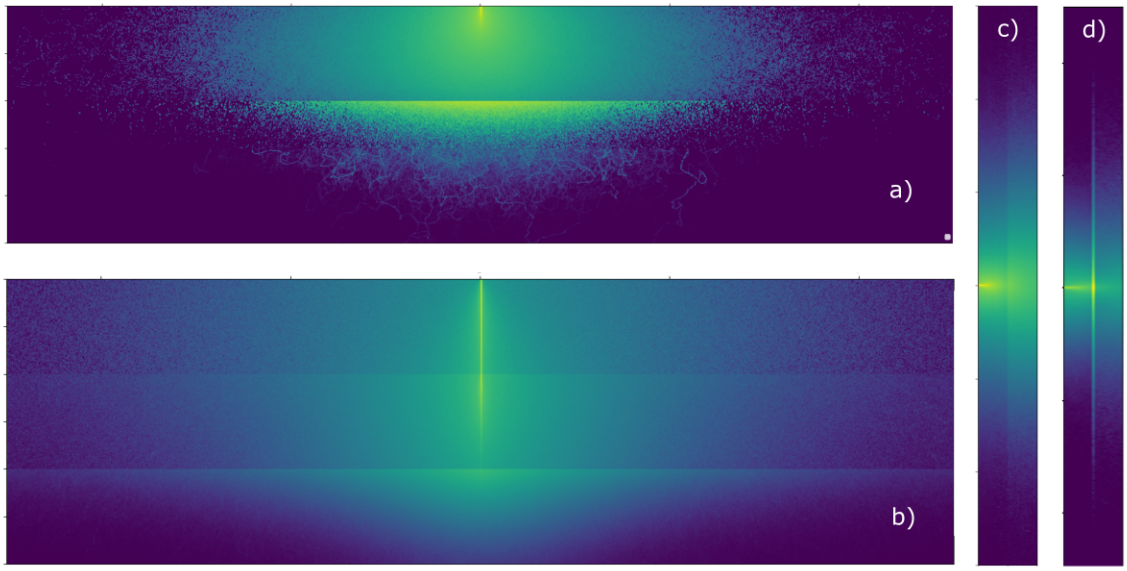


Figure 3.10: Results of the PyTissueOptics simulations when requesting X-projection graphs for the validation scenes. In order: a)-d) representing scenes 1-4. The axes of propagation (Z-axis) are along the shortest sides. The Z axis of scene 3 and 4 is stretched to show the smaller layers.

The level of validation presented is aimed at displaying the overall accuracy for simple scenes, like 3 layered stacked tissues (see results on Figure 3.10). More complex validations will be required when comparing complex scenes, as the PyTissueOptics tool is developed and improved. However, this is beyond the scope of this work. The codes used to obtain those numbers for MCML, MCX and PyTissueOptics are available in the GitHub[1] repository of the project.

3.4 Performance Analysis

Performance analysis is based on the scenarios described in Table 3.1. We recorded simulation times for each run with each algorithm, averaging over 10 runs (N=10). These times include phases like initialization, data transfer, kernel execution, and data saving. As the MCML package does not detail separate phase times, only total computation time is compared.

The execution of the MCML simulations take place on a Ryzen 5 5600X 6 core

processor clocked@3.7GHz, while the execution for the GPU-capable packages MCX and PyTissueOptics take place on a GTX1060 3GB 1152 cores clocked@1.506MHz.

Scene	MCML	MCX	PTO
Scene 1	25	303	133
Scene 2	22	1098	120
Scene 3	32	3815	148
Scene 4	58	1150	245

Table 3.2: Average computation speed reported in photons/ms for validation scenes (N=10).

The results, shown in Table 3.2, indicate the total computational speed in photons/ms. This measurement encompasses all execution phases including kernel runtime. The reported value represents the average speed from 10 sample runs. These results demonstrate that the PTO's GPU-based computations are significantly faster than MCML's CPU-based processes by approximately an order of magnitude. MCX also shows a considerable speed advantage over PTO for simple three-layer scenarios.

However, each module's performance can be influenced by various factors such as hardware specifications and software optimizations. Although these results were obtained on the same hardware setup, it's essential to consider that actual performance may vary with different hardware configurations. The parameter settings for each scenario were chosen based on available documentation and user understanding, suggesting that further optimizations for MCML and MCX could be possible.

For MCX specifically, scene dimensional parameters had to be tuned in order to allow each simulation to run correctly and output the same result than MCML. In the case of smaller layers of scene 2 and 3, scene dimension were multiplied by 10 and optical parameters divided by 10 in order for the package to understand smaller layers. This an example of parameter that could have affected the overall speed of the simulation.

Thus, it is possible that further improvements of the execution speed could be achieved for the MCML and MCX package. One must keep in mind that the exact execution speed is dependent on a number of parameters, such as scene composition itself. Nonetheless, these comparisons provide a general overview of each package's performance, offering insights into their practical usability and relative efficiency.

3.5 Practical examples for biophotonics

3.5.1 Propagation in Infinite Medium

Here we simulate the propagation of a divergent uniform source through an `InfiniteTissue`. We give the Tissue a material to propagate through and we use the useHardwareAcceleration flag to propagate the source with the advantages of the OpenCL propagation code. By default, this flag is set to True. Usually, it is not possible to view the data in 3D without binning as this generates too much data. 2D-Projections are recommended. This type of simulation is quite simple and faster than when there are objects or boundaries in the scene, since each photon does not have to test the triangle hit conditions.

```
myMaterial = ScatteringMaterial(mu_s=30.0, mu_a=0.1, g=0.9)
tissue = samples.InfiniteTissue(myMaterial)

logger = EnergyLogger(tissue)
source = DivergentSource(position=Vector(0, 0, 0), direction=Vector(0, 0, 1),
    N=50000,
diameter=0.2, divergence=math.pi/4)

source.propagate(tissue, logger=logger)

viewer = Viewer(tissue, source, logger)
viewer.reportStats()
viewer.show2D(View2DProjectionX())
viewer.show2D(View2DProjectionX(limits=((0, 2), (-1, 1))))
```

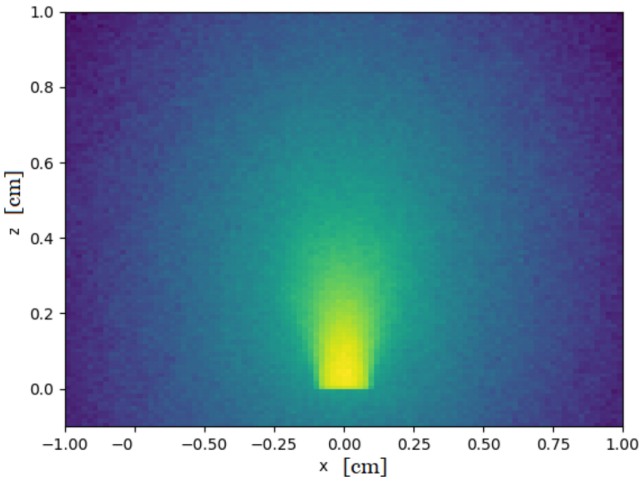


Figure 3.11: Close-up view of a 2D binned projection of the energy deposited when propagating 50k photons with hardware acceleration in a highly scattering infinite tissue (μs = 30, μa = 0.1, g = 0.9) with a `DivergentSource`.

3.5.2 Propagation in a custom layer stack

This example shows how to make a custom layer stack. Three `Cuboids` are created and stacked one after the other in a layer stack. It is possible to do this using `cuboid.stack()`. The cuboid sides on the stacked face have to have the same dimensions on both cuboids.

```
N = 5000000 if hardwareAccelerationIsAvailable() else 500
materialLayer1 = ScatteringMaterial(mu_s=2, mu_a=0.5, g=0.7, n=1.3)
materialLayer2 = ScatteringMaterial(mu_s=5, mu_a=0.8, g=0.8, n=1.4)
materialLayer3 = ScatteringMaterial(mu_s=50, mu_a=2.5, g=0.9, n=1.5)

layer1 = Cuboid(a=10, b=10, c=1, position=Vector(0, 0, 0), material=materialLayer1)
layer2 = Cuboid(a=10, b=10, c=1, position=Vector(0, 0, 0), material=materialLayer2)
layer3 = Cuboid(a=10, b=10, c=1, position=Vector(0, 0, 0), material=materialLayer3)
stack1 = layer1.stack(layer2, "back")
stackedTissue = stack1.stack(layer3, "back")
tissue = ScatteringScene([stackedTissue])

logger = EnergyLogger(tissue)
source = PencilPointSource(position=Vector(0, 0, -2), direction=Vector(0, 0, 1), N=N)
source.propagate(tissue, logger)

viewer = Viewer(tissue, source, logger)
viewer.reportStats()
viewer.show2D(View2DProjectionX())
```

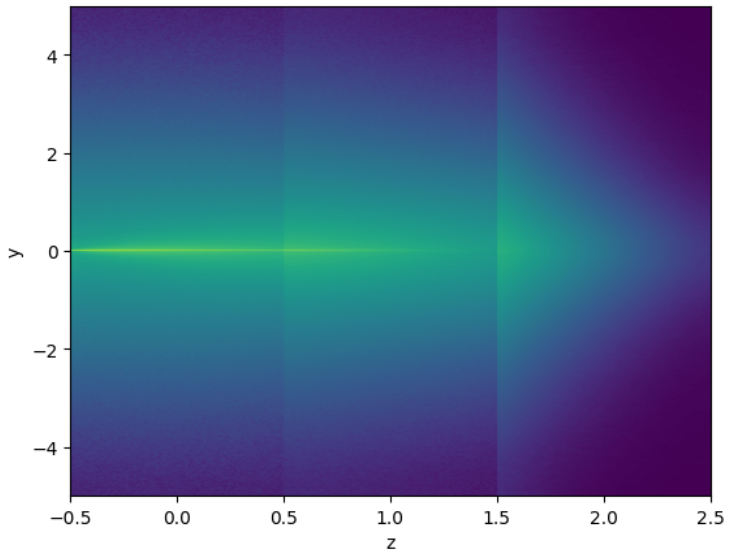


Figure 3.12: 2D binned projection of the energy deposited in three layers of different scattering properties and index with a PencilSource. Photon Count = 50M

3.5.3 Propagation in a complex scene

Propagation of a directional light source through a highly scattering medium scene composed of a sphere inside a cube. This type of scene is considered complex, as it has multiple objects, some of which with high triangle count, and objects are within each other. This requires management of the interfaces correctly to ensure that during a Fresnel intersection the proper refraction indices are used.

```
N = 1000000 if hardwareAccelerationIsAvailable() else 200

material1 = ScatteringMaterial(mu_s=20, mu_a=0.1, g=0.9, n=1.4)
material2 = ScatteringMaterial(mu_s=30, mu_a=0.2, g=0.9, n=1.7)
cube = Cuboid(a=3, b=3, c=3, position=Vector(0, 0, 0), material=material1)
sphere = Sphere(radius=1, order=3, position=Vector(0, 0, 0), material=material2,
↪  smooth=True)
scene = ScatteringScene([cube, sphere])

logger = EnergyLogger(scene)
source = DirectionalSource(position=Vector(0, 0, -2), direction=Vector(0, 0, 1), N=N,
↪  diameter=0.5, displaySize=0.25)

source.propagate(scene, logger)

viewer = Viewer(scene, source, logger)
viewer.reportStats()
viewer.show2D(View2DProjectionY())
```

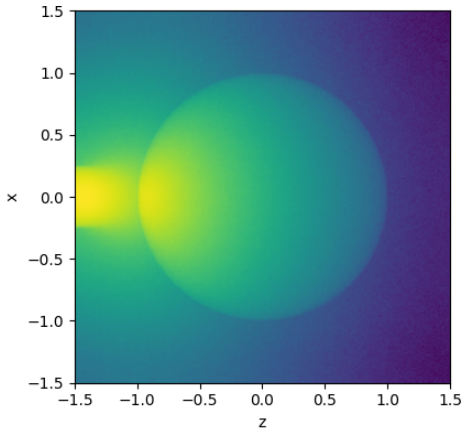


Figure 3.13: 2D binned Y projection of the energy deposited in a highly scattering sphere within a highly scattering cube. Photon Count = 1M

3.5.4 Propagation through optical elements

Thin Cuboid solids are used as screens for visualization, and a SymmetricLens() is used as a lens. They all go into a RayScatteringScene which takes a list of solid. It is possible to display our scene before propagation. It is possible to experiment with different focal lengths and material. The symmetric thick lens will automatically compute the required surface curvatures to achieve the desired focal length.

```
N = 1000000 if hardwareAccelerationIsAvailable() else 2000
glassMaterial = ScatteringMaterial(n=1.50)
vac = ScatteringMaterial()
s1 = Cuboid(a=40, b=40, c=0.1, position=Vector(0, 0, 30), material=vac, label="1")
s2 = Cuboid(a=40, b=40, c=0.1, position=Vector(0, 0, 70), material=vac, label="2")
s3 = Cuboid(a=40, b=40, c=0.1, position=Vector(0, 0, 99), material=vac, label="3")

lens = SymmetricLens(f=100, diameter=25.4, thickness=3.6, material=glassMaterial,
↪  position=Vector(0, 0, 0))
myCustomScene = ScatteringScene([screen1, screen2, screen3, lens])
source = DirectionalSource(position=Vector(0, 0, -20), direction=Vector(0, 0, 1),
↪  diameter=20.0, N=N, displaySize=5)

logger = EnergyLogger(myCustomScene, "ex03.log")
source.propagate(myCustomScene, logger)

viewer = Viewer(myCustomScene, source, logger)
viewer.reportStats()
viewer.show3D()
viewer.show2D(View2DSurfaceZ("1", "front", surfaceEnergyLeaving=False))
viewer.show2D(View2DSurfaceZ("2", "front", surfaceEnergyLeaving=False))
viewer.show2D(View2DSurfaceZ("3", "front", surfaceEnergyLeaving=False))
```

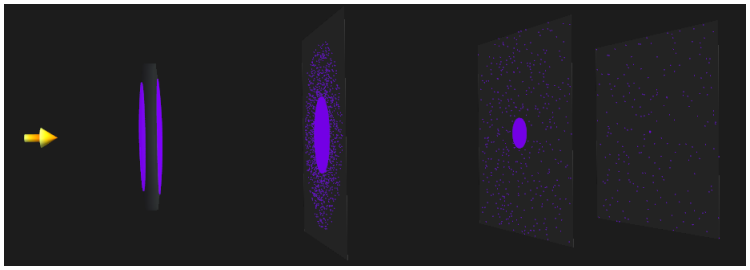


Figure 3.14: 3D visualization of the surface interactions (purple) on a lens and on multiple screens. The circular spot on the screen reduces in size with distance. The last screen on the right is positioned at the focal distance of the lens, thus its small size.

Discussion

This project was focused on the development of a Monte Carlo light propagation simulator in python called PyTissueOptics[1]. The propagation physics was inspired by [2], which is the propagation model of the MCML code. To carry out this project, an independent 3D graphics framework was built and a ray light propagation simulation module was built on top of it.

Achievement of objectives

The first objective is the correction of the rigid software architecture and the quality assurance of the submitted code. The software architecture has been redesigned and standardized with the terms and concepts that belong to the software engineering community. This restructuring, the application of the SOLID principles, and the extensive testing of the package helped to allow more extensibility, readability, and robustness to the code. The centralization of the propagation physics in the Photon object solved the major SRP problem. The major DIP problem was solved by separating geometry from physics completely. The resultant code and architecture satisfy the quality assurance criteria enough to be able to say that the first objective was achieved.

The second objective is the improvement of the versatility of the geometric domain to allow arbitrary complex 3D geometries. To achieve this, the geometric domain was completely redesigned and isolated in its own package, which lead to the 3D Graphic Framework. The polygon-mesh technique was selected because of its ease of implementation and versatility. The import of external 3D geometry is now supported. It converts 3D files to python objects which can be used as any 3D object in the package. It also includes an intersection finder which allows the user to search for collisions between segments and polygons in the scene. Labels can be given to solid, surface, or groups of polygons, to easily identify and link data to these objects. A 3D viewer has been integrated into the package in order to better visualize 3D environments and provide a better experience in general for the user. Overall, the second objective was exceeded because arbitrarily complex 3D geometries are now supported and because the geometry domain has been packaged in an independent entity with supplementary

features such as a viewer, a logger, and an intersection finder.

The third objective is the implementation of a functional light propagation simulator based on the Monte Carlo method that is simple of understanding, simple to use, and easily modifiable. To achieve this, the physics domain was completely encapsulated in a package that is dependent on the 3D Graphics Framework. The physics flow was adapted to the Python language and modified to be easily parallelizable. The results of this simulator have been verified to give consistent results with MCML on 1-layer and 2-layered scenarios. Simulations can be started with as little as 7 lines of code, which are all evocative, and the code itself follows strict QA and clean code principles, which helps a lot for the maintainability. Moreover, the validation section shows that results for 3-layers stacks are within an acceptable order of magnitude, when compared to the reference MCML code and to MCX, a more advanced package. Thus, the objective was achieved as the package provides valid simulation data, ease of use and maintainability, which all have been concerns through the development of the package.

The last objective is to increase the speed of execution of the light propagation, in order to obtain a speed of execution comparable to that of MCML for similar geometries. Speed was tackled on two fronts, the physics propagation speed, and the intersection search speed. Both ways were explored independently. The intersection search speed was improved using a space partitioning technique called the SAH-based KD-tree partitioning system. This allowed dividing the space automatically where polygon density is higher and allowed to maximize the exclusion of empty space. This resulted in a large improvement in speed for particular sets of scenes. The speed improvement depends on the configuration of the scene (the polygon distribution, number of objects and the materials). PyOpenCL has been selected to parallelize propagation because of its cross-platform compatibility and performance with multi-core hardware. Overall the objective was achieved because the GPU-Accelerated algorithm of PyTissueOptics can excess the speed of the MCML code.

Perspectives

Current limitations and future features

Design and feature choices have been made throughout the project, sometimes in order to reduce the scope of the project, which obviously is in a state of continuous improvements. First, regarding photon interaction with interfaces, only specular reflections have been implemented. However, many other models of interaction exist, such as Lambertian reflections or Bidirectional Reflectance Distribution Function (BRDF), which are two popular models for reflection[45]. Of course, one could easily modify the code to implement these models

themselves, but in the future, these could be implemented by default and given as a choice to the user.

Second, regarding the physics aspect, the polarization state of the photons is a feature that could easily be added in the future. It would open the door for bulk polarization simulation, like in raytracing, where the interference phenomenon is not very important. The Monte Carlo model of light propagation does not work with interference, because all the scattering events are randomly distributed, which averages out the interference. However, apart from interference, bulk polarization can still reveal some interesting information in a simulation [46]. Chromaticity could also be given another important parameter of the simulations. Tissues properties (absorption, scattering) have often a very strong dependence on the energy of the photon. Implementing a wavelength parameter to the photon, and providing wavelength-dependent functions as optical properties could reveal more accurate information.

Speed of execution has been improved due to the efficient intersection finding algorithm and because of the OpenCL implementation of the propagation physics. In the hardware-accelerated simulations, almost half of the execution time comes from the data transfer rather than the kernel computation. This proves to be a bottleneck that could be revisited if other techniques of data management were used. Further improvements are also required on the efficiency of the kernel in OpenCL.

Metrics usually used by the users of Monte Carlo simulation software include absorbance, reflectance, and transmittance, which are available metrics in PyTissueOptics. Fluence is also an important metric that is not yet implemented. PyTissueOptics could also directly estimate the thermal rise in a particular tissue due to light absorption, which would simplify greatly this problem, which is a frequent question in biophotonic, eg [47], [48], [49].

This project being an Open-Source project, it is expected that external developers give their contribution to the package if the guidelines are well made and the package well documented. A Kanban board has been put up on the official GitHub repository (see Appendix G) This lists the future features in order of importance for the project. The DCCLab group will ensure future developments and moderation regarding the PyTissueOptics package. This Kanban will serve as a guide for future developers and as a reminder of the features that should be developed first.

In research and scientific applications

The field of biophotonics is the field that is the most related to this project. Biophotonics is the study of biological entities with the use of light as a tool, so naturally, this light simulator in diffuse media targets this field in particular. However, because 3D models can be imported and smoothed with this package, optical system simulations could also be possible. It also

is planned as a future feature to allow exact lens profiles to be added onto 3D objects to replace the polygon mesh. Because the development of PyTissueOptic is still young, it could easily show applications in other fields, such as snow optics simulation. After discussions with Takuvik students, it has been observed that the field of snow optics has similar problems to the field of light propagation in tissue, as snow acts mostly like a tissue. This could lead to some interesting use of the package, or even in the addition of unplanned features by the community.

The DCCLab group has already published a raytracing module entirely coded in Python [50]. This work could be merged with the raytracing module to allow Monte Carlo based raytracing, which would result in a more complete tool for light simulation.

PyTissueOptics has also good potential to become a popular tool to teach the Monte Carlo technique, light propagation physics, and teach robust software development. The package achieved its objective of being extremely easy to understand, use and modify, which are good attributes of a teaching tool. As PyTissueOptics was previously used in Prof. Daniel Côté's class on light propagation in tissue, it is believed this improved version will be taught to his future students.

Summary

In summary, PyTissueOptics is the first Open-Source Python-based light propagation simulation package in diffuse tissue known to be developed. It also includes one of the first Python 3D graphics frameworks. This framework provides great support for light simulation in arbitrarily complex 3D tissues and environments. The raw simulation speed, while slower than its MCX counterpart by a factor of 10-1000x, is shown to be sufficient for research purposes since when combined with its easy learning curve and extensibility, PyTissueOptics is the fastest solution overall. Also, the current deployed version allows hardware-accelerated simulations in complex geometries, a considerable advantage over MCML. This project is still in its development phase, but the working module that is presented now is a huge step for the Python-based Monte Carlo light simulation alternatives and sets the scene for the democratization of the use of this method in biophotonics.

Appendix

A PyOpenCL example

```
#!/usr/bin/env python

import numpy as np
import pyopencl as cl

a_np = np.random.rand(50000).astype(np.float32)
b_np = np.random.rand(50000).astype(np.float32)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_g = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a_np)
b_g = cl.Buffer(ctx, mf.REAFD_ONLY | mf.COPY_HOST_PTR, hostbuf=b_np)

prg = cl.Program(ctx, """
__kernel void sum(
    __global const float *a_g, __global const float *b_g, __global float *res_g)
{
  int gid = get_global_id(0);
  res_g[gid] = a_g[gid] + b_g[gid];
}
""").build()

res_g = cl.Buffer(ctx, mf.WRITE_ONLY, a_np.nbytes)
knl = prg.sum  # Use this Kernel object for repeated calls
knl(queue, a_np.shape, None, a_g, b_g, res_g)

res_np = np.empty_like(a_np)
cl.enqueue_copy(queue, res_np, res_g)

# Check on CPU with Numpy:
print(res_np - (a_np + b_np))
print(np.linalg.norm(res_np - (a_np + b_np)))
assert np.allclose(res_np, a_np + b_np)
```

B Demo of Vectorized Photon Propagation

```
from pytissueoptics import *
from time import time_ns

time0 = time_ns()
batches = 6
N = 50000
position = NumpyVectors(N=N)
direction = NumpyVectors([[0, 0, 1]]*N)
er = NumpyVectors([[1, 0, 0]]*N)
weight = NumpyScalars([1.0]*N)
isAlive = True
time1 = time_ns()
print(f"Initialization   ::  {(time1-time0)/1000000000}s")
time2 = time_ns()

for i in range(batches):

    time3 = time_ns()
    count = 0
    while isAlive:
        count += 1

        theta = NumpyScalars.random(N=N) * 2 * np.pi
        phi = NumpyScalars.random(N=N) * 2 * np.pi
        d = NumpyScalars.random(N=N)

        er.rotateAround(direction, phi)
        direction.rotateAround(er, theta)
        position = position + direction*d
        weight *= 0.9
        isAlive = (weight.v > 0.001).any()

    position = NumpyVectors(N=N)
    direction = NumpyVectors([[0, 0, 1]] * N)
    er = NumpyVectors([[1, 0, 0]] * N)
    weight = NumpyScalars([1.0] * N)
    isAlive = True

    time4 = time_ns()
    print(f"Batch #{i}::{(time4-time3)/1000000000}s::{(i+1)*N/1000000}M
    ↪  photons/{N*batches/1000000}M::({int((i+1)*N*100/(N*batches))})")
time5 = time_ns()
print(f"Calculations Only::{N*batches/1000000}M
↪  photons::{(time5-time2)/1000000000}s")
```

```
43    print(f"Complete Process::{N*batches/1000000}M
      ↪  photons::{(time5-time0)/1000000000}s")
44    print(f"Performances::{((time5-time0)/1000)/(N*batches)} us/photon")
```

C World propagation method

```
1   def compute(self, graphs):
2       self.startCalculation()
3       N = 0
4       for source in self.sources:
5           N += source.maxCount
6
7           for i, photon in enumerate(source):
8               currentGeometry = self.contains(photon.globalPosition)
9               while photon.isAlive:
10                  if currentGeometry is not None:
11                      # We are in an object, propagate in it
12                      currentGeometry.propagate(photon)
13                      # Then check if we are in another adjacent object
14                      currentGeometry = self.contains(photon.globalPosition)
15                  else:
16                      # We are in free space (World). Find next object
17                      intersection = self.nextObstacle(photon)
18                      if intersection is not None:
19                          # We are hitting something, moving to surface
20                          photon.moveBy(intersection.distance)
21                          # At surface, determine if reflected or not
22                          if intersection.isReflected():
23                              # reflect photon and keep propagating
24                              photon.reflect(intersection)
25                              # Move away from surface to avoid getting stuck
                                ↪ there
26                              photon.moveBy(d=1e-3)
27                          else:
28                              # transmit, score, and enter (at top of this loop)
29                              photon.refract(intersection)
30                              intersection.geometry.scoreWhenEntering(photon,
                                ↪ intersection.surface)
31                              # Move away from surface to avoid getting stuck
                                ↪ there
32                              photon.moveBy(d=1e-3)
33                              currentGeometry = intersection.geometry
34                      else:
35                          photon.weight = 0
36              self.showProgress(i + 1, maxCount=source.maxCount, graphs=graphs)
37
38      duration = self.completeCalculation()
39      print("{0:.1f} ms per photon\n".format(duration * 1000 / N))
```

D Geometry propagation method

```
def propagate(self, photon):
    photon.transformToLocalCoordinates(self.origin)
    self.scoreWhenStarting(photon)
    d = 0
    while photon.isAlive and self.contains(photon.r):
        # Pick distance to scattering point
        if d <= 0:
            d = self.material.getScatteringDistance(photon)

        intersection = self.nextExitInterface(photon.r, photon.ez, d)

        if intersection is None:
            # If the scattering point is still inside, we simply move
            # Default is simply photon.moveBy(d)
            photon.moveBy(d)
            d = 0
            delta = photon.weight * self.material.albedo
            photon.decreaseWeightBy(delta)
            self.scoreInVolume(photon, delta)

            # Scatter within volume
            theta, phi = self.material.getScatteringAngles(photon)
            photon.scatterBy(theta, phi)
        else:
            # If the photon crosses an interface, we move to the surface
            photon.moveBy(d=intersection.distance)

            # Determine if reflected or not with Fresnel coefficients
            if intersection.isReflected():
                # reflect photon and keep propagating
                photon.reflect(intersection)
                photon.moveBy(d=1e-3)  # Move away from surface
                d -= intersection.distance
            else:
                # transmit, score, and leave
                photon.refract(intersection)
                self.scoreWhenExiting(photon, intersection.surface)
                photon.moveBy(d=1e-3)  # We make sure we are out
                break

        # And go again
        photon.roulette()
    self.scoreWhenFinal(photon)
    photon.transformFromLocalCoordinates(self.origin)
```

E Propagation functions in OpenCL C

```
1
2
3   void decreaseWeightBy(__global photonStruct *photons, float delta_weight, uint
    ↪  gid){
4       photons[gid].weight -= delta_weight;}
5
6   void interact(__global photonStruct *photons, __constant materialStruct
    ↪  *materials,
7   __global loggerStruct *logger, uint gid, uint logIndex){
8       float delta_weight = photons[gid].weight *
        ↪  materials[photons[gid].material_id].albedo;
9       decreaseWeightBy(photons, delta_weight, gid);
10      logger[logIndex].x = photons[gid].position.x;
11      logger[logIndex].y = photons[gid].position.y;
12      logger[logIndex].z = photons[gid].position.z;
13      logger[logIndex].delta_weight = delta_weight;}
14
15  void moveBy(__global photonStruct *photons, float distance, uint gid){
16      photons[gid].position += (distance * photons[gid].direction);}
17
18  float getScatteringDistance(__global float * randomNums, float mu_t, uint gid){
19      return -log(randomNums[gid]) / mu_t;}
20
21  float getScatteringAnglePhi(__global float * randomNums, uint gid){
22      float phi = 2.0f * M_PI * randomNums[gid];
23      return phi;}
24
25  float getScatteringAngleTheta(__global float * randomNums, float g, uint gid){
26      if (g == 0){
27          return acos(2.0f * randomNums[gid] - 1.0f);
28      }
29      else{
30          float temp = (1.0f - g * g) / (1 - g + 2 * g * randomNums[gid]);
31          return acos((1.0f + g * g - temp * temp) / (2 * g));}}
32
33  void scatterBy(__global photonStruct *photons, float phi, float theta, uint
    ↪  gid){
34      rotateAroundAxisGlobal(&photons[gid].er, &photons[gid].direction, phi);
35      rotateAroundAxisGlobal(&photons[gid].direction, &photons[gid].er, theta);}
36
37  void roulette(__global photonStruct *photons, __global uint * randomSeedBuffer,
    ↪  uint gid){
38      float randomFloat = getRandomFloatValue(randomSeedBuffer, gid);
39      if (randomFloat < 0.1f){
40          photons[gid].weight /= 0.1f;}
```

```
41          else{
42              photons[gid].weight = 0.0f;}}
```

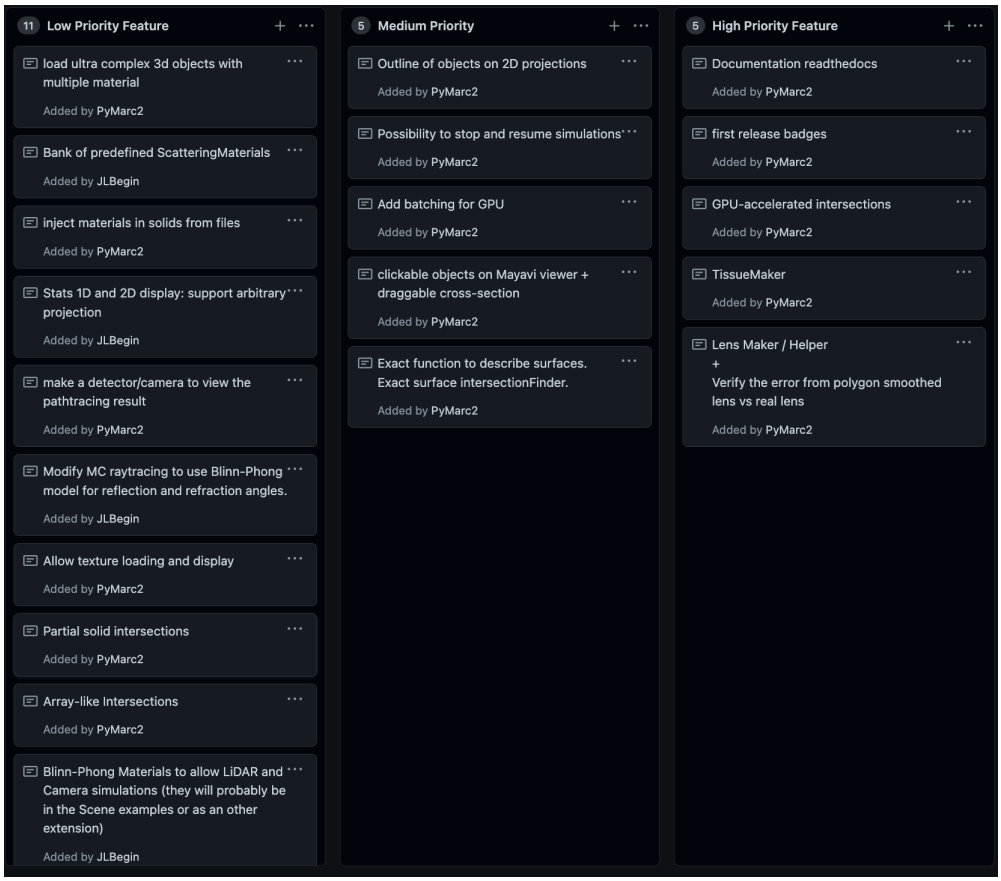
F Propagation kernel in OpenCL C

```
__kernel void getScatteringDistanceKernel(__global float * distanceBuffer,
__global float * randomNums, float mu_t){
    uint gid = get_global_id(0);
    distanceBuffer[gid] = getScatteringDistance(randomNums, mu_t, gid);}

__kernel void getScatteringAnglePhiKernel(__global float * angleBuffer,
__global float * randomNums){
    uint gid = get_global_id(0);
    angleBuffer[gid] = getScatteringAnglePhi(randomNums, gid);}

__kernel void getScatteringAngleThetaKernel(__global float * angleBuffer,
__global float * randomNums, float g){
    uint gid = get_global_id(0);
    angleBuffer[gid] = getScatteringAngleTheta(randomNums, g, gid);}
```

G Kanban Board



Kanban board of the planned features ordered by importance.

Bibliography

[1]DCC-Lab, *Pytissueoptics*, (2024) https://github.com/DCC-Lab/PyTissueOptics (visited on 05/05/2024).

[2]J. L. S. Wang Lihong, *Monte carlo modeling of light transport in multi-layered tissues in standard C*, https://omlc.org/software/mc/man_mcml.pdf.

[3]Q. Fang and al., *Mcx code base*, Accessed: 2023-09-30.

[4]Q. Fang and D. A. Boas, "Monte carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units", en, Opt. Express **17**, 20178–20190 (2009).

[5]L. Yu, F. Nina-Paravecino, D. Kaeli, and Q. Fang, "Scalable and massively parallel monte carlo photon transport simulations for heterogeneous computing platforms", en, J. Biomed. Opt. **23**, 1–4 (2018).

[6]Q. Fang and S. Yan, "MCX cloud-a modern, scalable, high-performance and in-browser monte carlo simulation platform with cloud computing", en, J. Biomed. Opt. **27**, 10.1117/1.JBO.27.8.083008 (2022).

[7]Q. Fang, *Pmcx*, Accessed: 2023-09-30.

[8]H. Li, C. Zhang, and X. Feng, "Monte carlo simulation of light scattering in tissue for the design of skin-like optical devices", en, Biomed. Opt. Express **10**, 868–878 (2019).

[9]S. I. Rivera-Manrique, M Brio-Perez, J. A. Trejo-Sanchez, H. L. Offerhaus, J. R. Ek-Ek, and J. A. Alvarez-Chavez, *Thermal diffusion and specular reflection, monte carlo-based study on human skin via pulsed fiber laser energy*, https://ris.utwente.nl/ws/portalfiles/portal/113723089/RiveraManrique.pdf, Accessed: 2022-7-16.

[10]G. Arias-Gil, F. W. Ohl, K. Takagaki, and M. T. Lippert, "Measurement, modeling, and prediction of temperature rise due to optogenetic brain stimulation", en, Neurophotonics **3**, 045007 (2016).

[11]B. H. Herrmann and C. Hornberger, "Monte-Carlo simulation of light tissue interaction in medical hyperspectral imaging applications", en, Current Directions in Biomedical Engineering **4**, 275–278 (2018).

[12]*Tiobe index*, Accessed: 2023-09-30.

[13]N Metropolis, *THE BEGINNING of the MONTE CARLO METHOD*, https://library.lanl.gov/cgi-bin/getfile?00326866.pdf, Accessed: 2022-6-28, 1987.

[14]*Academic reports using mcnp*, Accessed: 2023-09-30.

[15]A. Haghighat, *Monte carlo methods for particle transport; second edition*, http://dx.doi.org/9780429584107.

[16]M. I. Mishchenko, "Radiative transfer theory: from maxwell's equations to practical applications", in *Wave scattering in complex media: from theory to applications* (Springer Netherlands, Dordrecht, 2003), pp. 366–414.

[17]Jacques, Alter, and Prahl, "Angular dependence of HeNe laser light scattering by human dermis", Lasers Life Sci.

[18]D Toublanc, "Henyey-Greenstein and mie phase functions in monte carlo radiative transfer computations", en, Appl. Opt. **35**, 3270–3274 (1996).

[19]S. P. Steven Jacques, *Introduction to biomedical optics*, 2002.

[20]A. Liemert, D. Reitzle, and A. Kienle, "Analytical solutions of the radiative transport equation for turbid and fluorescent layered media", en, Sci. Rep. **7**, 3819 (2017).

[21]S. A. Prahl, "Light transport in tissue", PhD thesis (The University of Texas at Austin).

[22]S. A. Prahl, "A monte carlo model of light propagation in tissue", in Dosimetry of laser radiation in medicine and biology, edited by G. J. Mueller, D. H. Sliney, and R. F. Potter (Jan. 1989).

[23]M. H. C. Technologist and G. P. U. Computing, *A BRIEF HISTORY OF GPGPU*, https://www.cs.unc.edu/xcms/wpfiles/50th-symp/Harris.pdf, Accessed: 2022-07-06.

[24]CUDA Working Group, *CUDA c++ programming guide*, https://docs.nvidia.com/cuda/archive/11.2.0/pdf/CUDA_C_Programming_Guide.pdf.

[25]"Programming in opencl and its advantages in a gpu framework", International Journal For Science Technology And Engineering **10**, 3739–3743 (2022).

[26]Khronos OpenCL Working Group, *OpenCL Presentation*, https://www.khronos.org/assets/uploads/apis/OpenCL-3.0-Launch-Apr20.pdf, Accessed: 2022-07-06.

[27]A. Kloeckner, Y. Yu, M. Wala, I. Fernando, M. Bencun, K. Kulkarni, M. Diener, H. Gao, A. Fikl, Z. Weiner, M. Weigert, R. Palmer, S. Latham, G. Magno, H. Fuller, J. Mackenzie, S. Niarchos, S. Gill, C. Gohlke, A. Bhosale, A. Rothberg, E. Ey, H. Rapp, S. van der Walt, G. Thalhammer, J. Kieffer, N. Poliarnyi, D. Bollinger, A. Nitz, and G. Bokota, *Pyopencl*, version v2024.1, Jan. 2024.

[28]N. Pollard, *Computer graphics cmu 15-462/662, lecture 13, spatial data structures*, 2022.

[29]R. C. Martin, "Design principles and design patterns", Object Mentor **1**, 597 (2000).

[30]Daniel Sieger, *Generating meshes of a sphere.* https://www.danielsieger.com/blog/2021/03/27/generating-spheres.html, Accessed: 2022-06-08, 2021.

[31]Oscar Sebio Cajaraville, *Four ways to create a mesh for a sphere*, https://medium.com/@oscarsc/four-ways-to-create-a-mesh-for-a-sphere-d7956b825db4, Accessed: 2022-06-08, 2015.

[32]C. Benthin, I. Wald, and P. Slusallek, "A scalable approach to interactive global illumination", en, Comput. Graph. Forum **22**, 621–630 (2003).

[33]T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer", in Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on graphics hardware - HWWS '05 (2005).

[34]M Hapala and V Havran, "Review: kd-tree traversal algorithms for ray tracing", en, Comput. Graph. Forum **30**, 199–213 (2011).

[35]D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing", in Proceedings of the 2007 symposium on interactive 3D graphics and games - I3D '07 (2007).

[36]O. Gervasi, D. Russo, and F. Vella, "The AES implantation based on OpenCL for multi/many core architecture", in The international conference on computational science and its applications - ICCSA 2010 (Mar. 2010).

[37]J. D. MacDonald and K. S. Booth, "Heuristics for ray tracing using space subdivision", Vis. Comput. **6**, 153–166 (1990).

[38]A. Soupikov, M. Shevtsov, and A. Kapustin, "Improving kd-tree quality at a reasonable construction cost", in Interactive ray tracing, 2008. RT 2008. IEEE symposium on (Sept. 2008), pp. 67–72.

[39]I. Wald and V. Havran, "On building fast kd-trees for ray tracing, and on doing that in O(N log n)", in 2006 IEEE symposium on interactive ray tracing (Sept. 2006), pp. 61–69.

[40]A. Woo, "Fast ray-box intersection", in *Graphics gems* (Academic Press Professional, Inc., USA, Aug. 1990), pp. 395–396.

[41]T. M. Oller Prosolvia, C. Ab, and B. Trumbore, *Fast, minimum storage ray/triangle intersection*, Accessed: 2022-7-24.

[42]A. F. Mobius, *Der barycentrische calcul ein neues hülfsmittel zur analytischen behandlung der geometrie dargestellt und insbesondere auf die bildung neuer classen von aufgaben und die entwickelung mehrerer eigenschaften der kegelschnitte angewendet von august ferdinand mobius professor der astronomie zu leipzig*, de (verlag von Johann Ambrosius Barth, 1827).

[43]E. W. Weisstein, *Barycentric coordinates*, https://mathworld.wolfram.com/BarycentricCoordinates.html, Accessed: 2022-7-24.

[44]S. Jacques, "Optical properties of biological tissues: a review", Physics in medicine and biology **58**, R37–R61 (2013).

[45]*A survey of brdf representation for computer graphics*, (1997) `https://www.cs.princeton.edu/~smr/cs348c-97/surveypaper.html` (visited on 02/01/2024).

[46]T. Yun, W. Li, X. Jiang, and H. Ma, "MONTE CARLO SIMULATION OF POLARIZED LIGHT SCATTERING IN TISSUES", in *Advances in biomedical photonics and imaging* (WORLD SCIENTIFIC, Sept. 2008), pp. 57–61.

[47]K. Podgorski and G. Ranganathan, "Brain heating induced by near-infrared lasers during multiphoton microscopy", en, J. Neurophysiol. **116**, 1012–1023 (2016).

[48]D. F. Cardozo Pinto and S. Lammel, "Hot topic in optogenetics: new implications of in vivo tissue heating", en, Nat. Neurosci. **22**, 1039–1041 (2019).

[49]M. Ghanbari and G. Rezazadeh, "Thermo-vibrational analyses of skin tissue subjected to laser heating source in thermal therapy", en, Sci. Rep. **11**, 22633 (2021).

[50]V. P. Noël, S. Masoumi, E. Parham, G. Genest, L. Bégin, M.-A. Vigneault, and D. C. Côté, "Tools and tutorial on practical ray tracing for microscopy", en, NPh **8**, 010801 (2021).