Development and application of a modular ray tracing framework to multi-scale simulations in photovoltaics

Von der Fakultät für Mathematik und Physik
der Gottfried Wilhelm Leibniz Universität Hannover

zur Erlangung des Grades

Doktor der Naturwissenschaften
Dr. rer. nat.

genehmigte Dissertation
von

Dipl.-Phys. Hendrik Holst
geboren am 18.02.1982 in Hannover

2015

Referent: PD Dr. Pietro Altermatt
Korreferent: Prof. Dr. Detlev Ristau
Tag der Promotion: 19.05.2015

Kurzzusammenfassung

Während optische Simulationen, auf Basis von Raytracing, im Bereich der Solarzellentwicklung bereits seit Jahrzehnten genutzt werden, ist die optische Simulation von Solarmodulen weiterhin eine Herausforderung. Dies hat zwei Gründe. Zum einen werden die optischen Eigenschaften von Modulen durch geometrische Strukturen bestimmt deren Größenordnungen stark variieren. Dies umfasst Größenordnungen im Bereich von Mikrometern für die Pyramidentextur von Solarzellen bis hin zu Abmessungen im Bereich von Zentimetern für die Modulgeometrie. Die daraus folgende Komplexität der Geometrie verhindert die Simulation innerhalb akzeptabler Zeiträume. Zum anderen erschwert die monolithische Struktur vorhandener Simulationssoftware die nötigen Erweiterungen, welche eine Anpassung an die spezifischen Eigenschaften des Problems erlauben und die Simulationszeit senken würden.

In dieser Arbeit wird daher ein neu entwickeltes, modulares Software Framework für optische Simulationen vorgestellt. Sein modularer Aufbau erlaubt dabei den flexiblen Eingriff in den Simulationsprozess und die Integration eigener Effekte. Der Aufbau und die Erweiterbarkeit des Daidalos genannten Frameworks werden detailliert beschrieben.

Mit Daidalos werden die optischen Eigenschaften photovoltaischer Komponenten simuliert, welche bislang nicht oder nicht in dieser Genauigkeit simuliert werden konnten. Erstmals werden Simulation anhand von detaillierten Messungen der texturierten Oberfläche von Solarzellen durchgeführt. Es wird gezeigt, dass solche Simulation die gemessene Reflektivität in guter Übereinstimmung wiedergeben und daher für die Optimierung des Texturierungsprozesses verwendet werden können. Darüber hinaus wird anhand von Simulationen gezeigt, dass der gute Lichteinfang von texturierten Silizium-Wafern dazu führen kann, dass die Reflektivität des Wafers in Messungen unterschätzt wird.

Bedingt durch die optische Wechselwirkung der verwendeten Komponenten innerhalb eines Moduls unterscheidet sich der Lichteinfang einer im Modul verbauten Solarzelle von dem einer Einzelzelle. Durch die komplexe Lichtausbreitung innerhalb des Module sind räumlich getrennte Bereiche optisch miteinander gekoppelt. Dieser Effekt macht optische Simulation auf Basis der lokalen Geometrie unmöglich. Zusätzlich verhindern die stark unterschiedlichen Größenordnungen der Modulkomponenten bislang die Simulation ausgedehnter Bereiche in akzeptabler Zeit. In dieser Arbeit wird ein neuer Ansatz zur Modellierung von Solarmodulen vorgestellt, der die Simulation ganzer Module ermöglicht. Der Ansatz wird genutzt um die Auswirkungen von unterschiedlichen Zellabständen auf den photogenerierten Strom zu untersuchen. Die gezeigten Resultate werden mit Messungen an einem 3×3 Mini-Modul verglichen und der Einfluss auf den generierten Gesamtstrom abgeschätzt.

Neben der Optimierung der optischen Eigenschaften von Modul und Zelle ist auch die konkrete Position eines Solarmoduls auf Hausdächern oder -fassaden von Bedeutung. Daher wird Daidalos verwendet, um den jährlichen Leistungsertrag zu ermitteln, wie er auf gängigen Gebäuden im Umland und der Stadt zu erwarten ist. Dazu wird eine, auf gemessenen Wetterdaten basierende, Tageslichtquelle genutzt, um die Auswirkung lokaler Abschattung auf den zu erwartenden jährlichen Leistungsertrag zu simulieren. Diese Simulationen demonstrieren die Möglichkeit, in Zukunft konkrete Aussagen über

die Nutzbarkeit einzelner Dach- und Fassadenbereichen für die Installation von photovoltaischen Systemen, auch für größere Gebiete, automatisiert zu treffen.

Abstract

While ray tracing of solar cells was established decades ago, ray tracing of entire modules has met obstacles for two main reasons: firstly, the optics of solar cell modules is determined by geometric structures on a wide scale of dimensions, ranging from pyramids on cells in the micrometer range to module geometry in the centimeter range. This leads to high complexity of the simulation. Secondly, although it would be possible to accelerate these simulations by exploiting specific properties, the code of available ray tracing software is rather hard to extend to serve this purpose.

For that reason, this work introduces a newly developed modular software framework for optical simulation. Its modular structure allows for a flexible access to the simulation process and the integration of new effects. The problems that occur using a modular approach are discussed and solutions are presented. The structure and extendibility of the framework, named Daidalos, is presented in detail.

Using Daidalos the optical characteristics of photovoltaic components are investigated by simulations that up to now either hadn't been done at all or not with the presented level of detail. One of the most important properties of a solar cell is the texture on its surface in the range of micrometers, which significantly reduces their reflectivity. For the first time simulations are presented that are based on detailed measurements of the textured surface. It is shown that such simulations of the surface's reflectivity are in good agreement with the measurements and therefore can be used to optimize the texture process. Furthermore, it is shown that the good light-trapping capabilities of textured silicon wafers can result in an error in reflectivity measurements that leads to an underestimation of the measured reflectivity.

Because of the optical interdependencies between the components of a solar cell module, the optical characteristics of a solar cell that is embedded within a module differ from that of a single cell. Due to the complex propagation within the module, spatially separated module regions are optically coupled. Therefore, optical simulations can not be performed on local geometries, but have to include wide regions of the module geometry. Moreover, due to the different length scales of the components used in common solar cell modules, such simulations typically cannot be performed in an acceptable amount of time. In this thesis a new approach for modeling the module geometry is presented that allows for a simulation of the whole module geometry. The presented approach is used to simulate the impact of the inter-cell gap on the photo-generated current. The shown results are compared to measurements on a 3×3 mini-module and the impact on the total photo-generate current is approximated.

Aside from the optimization of the optical characteristics of solar cells and solar cell modules, the particular position on a building's roof or facade is important with respect to the photo-generated current. For this is reason Daidalos is used to simulate the annual power yield as expect for buildings in the city and the urban hinterland. Using a newly developed daylight source, which is based on measured weatherdata, the angular distribution of daylight is considered to evaluate the impact of shadowing on the expected annual power yield.

Keywords: optical simulation, ray tracing, solar cells
Schlagwörter: optische Simulation, Raytracing, Solarzellen

Introduction

Solar cells generate an electric current from the conversion of absorbed sunlight. Therefore, to maximize the efficiency of a solar cell module its ability to absorb light needs to be optimized. Considering solar system installations this optimization process has to take place on different length scales. While commercially available solar cells are textured on a micrometer scale to provide a low reflectivity, solar cell modules are build from components in the range of centimeters to meters.

The paths by which light propagates through an optical system as complex as a solar cell cannot be calculated analytically. Therefore, supported by the rise of desktop computers and their ever increasing computational power, several software applications like RAYSIM [1], TEXTURE [2] and SUNRAYS [3] were developed to allow for a numerical calculation of optical characteristics of solar cells. Most of these applications were developed as monolithic software which provide their users with a fixed set of options to configure their simulations. Due to this limited reconfigurability several tools have to be available to account for different needs during simulation. Moreover, newly discovered effects often can not be integrated into simulations using available tools. In this cases, scientist have to develop there own tools often rewriting parts which are already provided by existing software just to add a small potion of extended functionality. This behavior is prone to errors as well as inefficient.

At the end of the 20th century the complexity of common software has grown to a size which made it cumbersome to develop software as a whole. Instead it was split into smaller parts that allowed for easier development and maintenance. In the year 2000 the *Open Services Gateway initiative* (OSGi) introduced the first release of the OSGi framework [4] which consists of set of specifications of interfaces for the Java to support the development of modular software.

Over the last twenty years several implementations, like *Knopflerfish* or *Apache Equinox*, of the interfaces described by the OSGi framework were developed. Using these implementations it is now possible to create Java applications which are not only developed in an efficient modular manner, but allow the exchange of modules after the application has

been deployed.

Based upon the todays available implementations of the OSGi framework a modular ray tracing simulation can be created. By following a modular approach it is not necessary to rewrite the whole application to add new features. Instead it is possible to develop single modules to extend the functionality where needed but rely on established modules where possible. Such a framework is presented with the following work and used on several problems that can be investigated by ray tracing simulations and that are hardly or not at all possible to implement in other currently available ray tracers

Structure of this thesis

The first part of this work is concerned with an introduction to the Daidalos ray tracing framework and the possibilities by which it can extended by the user. The second part demonstrates the usability of the Daidalos framework and its modular approach by presenting ray tracing simulation on various scales, ranging from wafer optics to the simulation of the facades of buildings.

Chapter 1 gives a short introduction to optics in introduces the concepts of refraction, reflection and absorption.

Chapter 2 presents the necessary background to the programing paradigm referred to as *object orientated programing* (OOP) as needed to understand the further chapters.

Chapter 3 introduces the concept of Monte-Carlo ray tracing. Furthermore, the statistical error which results from the application of this method is discussed.

Chapter 4 presents the newly developed Daidalos ray tracing framework. The first part gives an overview over the fundamental differences between monolithic and modular applications. Moreover, the typical problems which occur when developing modular applications are discussed. This is followed by an overview of the implementation of the Daidalos and its capabilities. The second part of this chapter discusses the possibilities by which user defined modules can interact with the framework to influence the ray tracing process

Chapter 5 demonstrates the use of Daidalos to investigate wafer optics and shows its functionality in comparison with measurements, reference simulations and analytical calculations. An artifact which occurs during the measurement of the reflectivity of wafers with good light-trapping is discussed. Finally, the simulation of the reflectivity of a textured wafer is conducted. For this simulation, the actual surface texture is measured by means of a laser scanning microscope and integrated into the simulation.

Chapter 6 presents a newly developed daylight source which is based upon measured weather data and can be used with Daidalos to simulate real daylight including its

spectral and angular distribution.

Chapter 7 demonstrates the use of DAIDALOS to investigate module optics. A new approach is presented which uses the capabilities of DAIDALOS to allow simulations of the optical characteristics of whole modules. The usability of this approach is demonstrated by simulating the impact of the gap between different solar cells within a solar module. This is compared with a *laser induced current measurement* (LBIC) of a 3×3-cell mini-module.

Chapter 8 uses the previously describes daylight source to simulate the annual irradiation which incidents on the facades of buildings. For these simulations the facades of three buildings are used which were modeled based upon laser scans of real facades and were provided by the *Institute of Cartography and Geoinformatics*(IKG) of the *Leibniz University Hannover*

Chapter 9 summarizes the results of this work

Contents

1 Optics

This chapter serves as an introduction to optics, starting with the common representation of light as an *electromagnetic wave*. In the following this representation is used to describe the propagation of light within matter as well as the concepts of absorption, reflection and refraction.

The text presented within this chapter is based on the work of Reider [5] with other references being marked separately.

1.1. Light as an electromagnetic wave

The fundamental phenomena of classical optics is the existence of the *electromagnetic field* described by the electrical field vector $\vec{E}(\vec{r},t)$ and the magnetic field vector $\vec{H}(\vec{r},t)$. In matter, these vectors are supplemented by the current density $\vec{j}(\vec{r},t)$, the displacement field $\vec{D}(\vec{r},t)$ and the magnetic induction $\vec{B}(\vec{r},t)$ [6]. The relations between these vector quantities are described by Maxwell's equations:

$$\nabla \cdot \vec{D}(\vec{r},t) = \rho, \tag{1.1}$$

$$\nabla \cdot \vec{B}(\vec{r},t) = 0, \tag{1.2}$$

$$\nabla \times \vec{H}(\vec{r},t) = \frac{\partial \vec{D}(\vec{r},t)}{\partial t} + \vec{j}(\vec{r},t), \tag{1.3}$$

$$\nabla \times \vec{E}(\vec{r},t) = -\frac{\partial \vec{B}(\vec{r},t)}{\partial t}, \tag{1.4}$$

where t is the time and ρ is the charge density. For a description of the electromagnetic field within matter Maxwell's equations are accompanied by a set of material equations:

$$\vec{D}(\vec{r},t) = \epsilon_0 \left(\vec{E}(\vec{r},t) + \vec{P}(\vec{r},t)\right) = \epsilon_0 \hat{\epsilon} \vec{E}(\vec{r},t), \tag{1.5}$$

$$\vec{B}(\vec{r},t) = \mu_0 \hat{\mu} (\vec{H} + \vec{M}). \tag{1.6}$$

Here, $\epsilon_0 = 8.854 \times 10^{-12}\,\mathrm{A\,s\,V^{-1}}$ is referred to as dielectric constant of the vacuum and $\mu_0 = 4\pi \times 10^{-7}\,\mathrm{V\,s\,A^{-1}\,m^{-1}}$ is the permeability of vacuum. Additionally, $\hat{\mu}$ and $\hat{\epsilon}$ are the

values for the relative permeability and the relative permittivity which are represented by tensor values in the general case. Furthermore, the polarization vector $\vec{P}$ describes the interaction of the electric field with matter.

1.1.1. The wave equation

Within the frequency range of optics the magnetic induction $\vec{M}$ can be usually considered to be insignificant and is therefore neglected within the further discussion. Furthermore, we are constraining to homogeneous media which are free of charges ($\rho = 0$) and currents ($\vec{j} = 0$).

Under this assumptions the rotation of equation (1.4) is considered as:

$$\begin{aligned} \nabla \times \left(\nabla \times \vec{E}(\vec{r},t) \right) &= \nabla \overbrace{(\nabla \vec{E})}^{=0} - \nabla^2 \vec{E} \\ &= -\frac{\partial}{\partial t} \nabla \times \vec{B}(\vec{r},t). \end{aligned} \tag{1.7}$$

Here, the first equality sign makes use of the identity $\nabla \times (\nabla \times \vec{a}) = \nabla(\nabla \cdot \vec{a}) - \nabla^2 \vec{a}$. Additionally, taking into account that the considered medium is free of charges and the relative permittivity is a scalar, it can be concluded that $\nabla \vec{D} = \epsilon_0 \epsilon \nabla \vec{E} = \rho = 0$. Therefore, using equation (1.3) this results in:

$$\nabla^2 \vec{E} - \frac{\epsilon \mu}{c_0^2} \frac{\partial^2 \vec{E}(\vec{r},t)}{\partial t^2} = 0, \tag{1.8}$$

where $c_0 = \sqrt{\epsilon_0 \mu_0}$ is the velocity of light in vacuum and is defined to be $299\,792\,458\,\mathrm{m\,s^{-1}}$. The above equation is known as the wave equation for the propagation of the electric field. A similar equation can be derived for the magnetic field:

$$\nabla^2 \vec{H} - \frac{\epsilon \mu}{c_0^2} \frac{\partial^2 \vec{H}(\vec{r},t)}{\partial t^2} = 0. \tag{1.9}$$

1.1.2. Harmonic plane waves

While the wave equation for the electric field (1.8) can be solved under a multitude of conditions, the most simple solution is that of a *harmonic plane wave*. This kind of electromagnetic wave is described by[1]:

$$\vec{E}(\vec{r},t) = \Re \left[\widetilde{\vec{E}}(\vec{r},t) \right] = \Re \left[\widetilde{\vec{E}}_0 \cdot \exp\left(-i(\vec{k}\vec{r} - \omega t) \right) \right] \tag{1.10}$$

Here, $\omega = 2\pi\nu$ describes the time harmonic oscillation of a wave with a frequency ν and is referred to as the *circular frequency* of the wave. The vector $\vec{k}$ describes the spatial

[1] While harmonic waves can also be described using trigonometric functions, the representation by an exponential function eases mathematical operations. For example, the application of the operator ∇ is reduced to a multiplication with $-i\vec{k}$.

harmonic oscillation of the wave and points in the direction of propagation. It is known as the *wave vector*. Furthermore, the amplitude vector $\widetilde{\vec{E}}_0(\vec{r},t)$ is in general a complex value which can be represented by:

$$\widetilde{\vec{E}}_0(\vec{r},t) = \begin{pmatrix} E_x \exp(i\varphi_x) \\ E_y \exp(i\varphi_y) \\ E_z \exp(i\varphi_z) \end{pmatrix}, \tag{1.11}$$

where $\phi_{x/y/z}$ are in general arbitrary phase values.

While equation (1.10) takes the real part of the complex wave function, the explicit evaluation of the real part and the explicit highlighting of the complex wave function is omitted in the following. That can be done as Maxwell's equations are linear and the application of the operators can be exchanged with the evaluation of the real part [5]. However, this is not valid anymore when squares of the electric field are considered, in which case the real part evaluation will be done explicitly.

Substituting equation (1.10) into equation (1.8) one can derive the magnitude $k = |\vec{k}|$ as:

$$-k^2\vec{E}(\vec{r},t) - \frac{\omega^2\epsilon\mu}{c_0^2}\vec{E}(\vec{r},t) = 0 \quad \Rightarrow \quad k = \sqrt{\frac{\omega^2\epsilon\mu}{c_0^2}} = \frac{\omega n}{c_0}, \tag{1.12}$$

where $n = \sqrt{\epsilon\mu}$ is referred to as the *index of refraction* of the medium in which the wave propagates. In particular it is $\epsilon = \mu = 1$ in vacuum and therefore $n_{\text{vac}} = 1.0$.

In general ϵ and μ can be complex values[2], in this case the *complex index of refraction* $\hat{n}$ is represented by:

$$\hat{n} = n - i\kappa, \tag{1.13}$$

where κ is referred to as the *extinction coefficient*.

1.1.3. The wavelength

Equation (1.10) introduced the harmonic plane wave, specifying its circular frequency $\omega = 2\pi\nu$ as well as its vacuum velocity c_0. In fact, the velocity c_0 of a plane wave propagating in vacuum and its frequency ν are interrelated by:

$$c_0 = \nu\lambda, \tag{1.14}$$

with λ being the *wavelength* of the wave. When propagating through matter, the propagation velocity of the wave is lower than c_0 by a factor n equal to the index of refraction of the medium. Therefore, equation (1.14) can be written as:

$$c = \frac{c_0}{n} = \nu\lambda \quad \Rightarrow \quad c_0 = \nu\left(\frac{\lambda}{n}\right), \tag{1.15}$$

and is further valid using a reduced wavelength $\lambda_{\text{mat}} = \lambda/n$ within matter.

[2] With respect to the frequencies used in optics μ is near to one for most materials.

1.1.4. Intensity of a plane wave

Following *Pointing's theorem* the energy contained within a volume due to the existence of an electromagnetic field can be written as:

$$\int_A \left[\left(\vec{E} \times \vec{H}\right) \vec{n}\right] dA = -\int_V \left[\frac{\partial}{\partial t}\left(\epsilon_0 \frac{\vec{E}^2}{2} + \mu_0 \frac{\vec{H}^2}{2}\right) + \vec{E}\frac{\partial \vec{P}}{\partial t}\right]. \tag{1.16}$$

Here, the left hand integral is taken over the surface of the volume while the right hand integral is taken over the volume. Additionally, the first term of the integral on the right hand side describes the energy contained within the volume, the second term describes the interaction with the medium and the left hand side represents the energy flux through the bounding faces of the volume. This flux is measured in direction of the normal vector $\vec{n}$ of the particular face and is referred to as the *Pointing vector* $\vec{S}$, with:

$$\vec{S} = \vec{E} \times \vec{H}. \tag{1.17}$$

In the case of plane waves, the Pointing vector is orientated in parallel to the wave vector $\vec{k}$. Assume a charge and current free medium with vanishing magnetization $\vec{M}$ and a relative permeability μ equal to one, then substituting equation (1.10) into (1.4) leads to:

$$\vec{k} \times \vec{E}(\vec{r}, t) = \omega\mu_0 \vec{H}(\vec{r}, t), \tag{1.18}$$

$$\vec{k} \times \vec{H}(\vec{r}, t) = -\omega\epsilon_0\epsilon \vec{E}(\vec{r}, t). \tag{1.19}$$

Using this, the Pointing vector $\vec{S}$ can be represented by:

$$\vec{S} = \vec{E} \times \vec{H} = \frac{1}{\omega\mu_0}\vec{E} \times \left(\vec{k} \times \vec{E}\right), \tag{1.20}$$

and using the orthogonality $\vec{E} \cdot \vec{k} = 0$ this can be further transformed to:

$$\begin{aligned} \vec{S} &= \frac{1}{\omega\mu_0\mu}\vec{E} \times \left(\vec{k} \times \vec{E}\right) \\ &= \frac{1}{\omega\mu_0\mu}\left[\vec{k}\left(\vec{E}(\vec{r}, t) \cdot \vec{E}(\vec{r}, t)\right) + \vec{E}(\vec{r}, t)\left(\vec{E}(\vec{r}, t) \cdot \vec{k}\right)\right] \\ &= \frac{\vec{E}(\vec{r}, t) \cdot \vec{E}(\vec{r}, t)}{\omega\mu_0}\vec{k}. \end{aligned} \tag{1.21}$$

Consequently, for plane waves the Pointing vector and therefore the flux of energy is orientated in the direction of the wave vector $\vec{k}$.

With respect to the high oscillation frequencies considered in optics quantities like the electric and magnetic field vector cannot be measured directly, but only as time averaged values. For this reason, it is the mean value $\left\langle \vec{S} \right\rangle$ which is of interest. This mean value of the energy flux density perpendicular to a plane with normal vector $\vec{f}$ is referred to as the *intensity* I of the wave and can be calculated using equation (1.21) by:

$$I = \left|\left\langle \vec{S} \right\rangle\right| = n\frac{\widetilde{\vec{E}}(\vec{r}, t) \cdot \widetilde{\vec{E}}(\vec{r}, t)^*}{2\mu_0 c_0} = n\frac{|E_0|^2}{2\mu_0 c_0}, \tag{1.22}$$

where n is the index of refraction of the medium. In the case that the normal vector $\vec{f}$ of the plane is not orientated parallel to $\vec{S}$ the value of the intensity is given by:

$$I = \left| \left\langle \vec{S} \right\rangle \cdot \vec{n} \right| = n \frac{|E_0|^2}{2\mu_0 c_0} \cos(\theta), \tag{1.23}$$

where θ is the angle between the direction of the energy flux and the plane's normal vector.

1.1.5. Polarization

Many phenomena in optics, for example the refraction at an interface between two media (see section 1.3), rely on the spatial orientation of the electric field vector. This state of orientation is referred to as the *polarization* of the electromagnetic wave.

With respect to plane waves it can be seen from equation (1.18) that the electrical field is always orientated perpendicular to the wave vector. For this reason, using equation (1.11) and choosing the wave vector $\vec{k}$ to be orientated in the direction of the z-axis, this leads to an $\vec{E}_0$ of:

$$\underset{\sim}{\vec{E}}_0(\vec{r}, t) = \begin{pmatrix} E_\mathrm{x} \exp(i\varphi_\mathrm{x}) \\ E_\mathrm{y} \exp(i\varphi_\mathrm{y}) \\ 0 \end{pmatrix} = \begin{pmatrix} E_\mathrm{x} \exp(i\Delta\varphi) \\ E_\mathrm{y} \\ 0 \end{pmatrix}. \tag{1.24}$$

In the last step, the phase values of x- and y-component are combined within the *relative phase* $\Delta\varphi$ which fully describes the relative orientation of both components.

In many situations (i.e. the evaluation of refraction in section 1.3) it is useful to define a base of two orthogonal polarization states. Due to the linearity of the Maxwell equations this orthogonal base can be used to describe any state of polarization by means of superposition.

1.2. Absorption

A harmonic plane wave which propagates within matter does interact with the medium. To visualize the impact of this interaction consider a plane wave as defined by equation (1.10) but restricted to one dimension. This is described by:

$$E(x, t) = E_0 \cdot \exp(-i(kx - \omega t)). \tag{1.25}$$

Assume the wave propagates a distance Δx from its initial position at $x = 0$. Then the electric field at $x = 0$ and $x = \Delta x$ is described by:

$$E(0, t) = E_0 \cdot \exp(-i\omega t) \tag{1.26}$$

$$E(\Delta x, t) = E_0 \cdot \exp(-i(k\Delta x - \omega t)). \tag{1.27}$$

Making use of equation (1.12), equation (1.27) can be written as:

$$\begin{aligned} E(\Delta x, t) &= E_0 \cdot \exp\left[-i\left(\frac{\omega\hat{n}}{c_0}\Delta x - \omega t\right)\right] \\ &= E_0 \cdot \exp\left[-i\left(\frac{\omega(n-i\kappa)}{c_0}\Delta x - \omega t\right)\right] \\ &= E_0 \cdot \exp\left[-i\left(\frac{\omega n}{c_0}\Delta x - \omega t\right)\right] \exp\left[-\frac{\omega\kappa}{c_0}\Delta x\right], \end{aligned} \tag{1.28}$$

where equation (1.13) was used to consider for the general case of a medium whose relative permittivity is a complex value. The intensities for the initial and the propagated wave can be calculated using equation (1.22). If the propagation of the considered wave occurs in an homogeneous medium then the term $n/(2\mu_0 c_0)$ is the same for $E(0,t)$ and $E(\Delta x, t)$ and the ratio of the intensity at both points can be calculated by:

$$\frac{|E(\Delta x, t)|^2}{|E(0,t)|^2} = \frac{|E_0|^2 \exp\left[-\frac{2\omega\kappa}{c_0}\Delta x\right]}{|E_0|^2} = \exp\left[-\alpha\Delta x\right], \tag{1.29}$$

where $\alpha = \frac{2\omega\kappa}{c_0}$ is referred to as the *absorption coefficient*.

Consequently, equation (1.29) describes an exponential attenuation of a plane harmonic wave which propagates through a medium with a complex index of refraction. This relation is known as the *Lambert-Beer law of absorption*.

1.3. Refraction

With respect to the propagation within matter, the behavior of the electromagnetic wave at an interface between media with different complex index of refraction $\hat{n}_1$ and $\hat{n_2}$ is of particular interest. Such an interface is shown in figure 1.1.

Assume the interface is located at $z = 0$ with the face normal vector $\vec{f}$ orientated parallel to the z-axis. In this case the harmonic plane wave described by equation (1.10) is represented at the point of incidence by:

$$\begin{aligned} \vec{E}_\mathrm{i}(\vec{r}, t) &= \vec{E}_0 \cdot \exp\left(-i(\vec{k}_\mathrm{i}\vec{r} - \omega t)\right) \\ &= \vec{E}_0 \cdot \exp\left(-i(\vec{k}_{\mathrm{i},\parallel}\vec{r} - \omega t)\right), \end{aligned} \tag{1.30}$$

where $\vec{k}_\mathrm{i}$ is the wave vector of the incident wave and $\vec{k}_{\mathrm{i},\parallel}$ is its parallel projection with respect to the interface. This condition must be valid for the incident as well as for the reflected and transmitted wave.

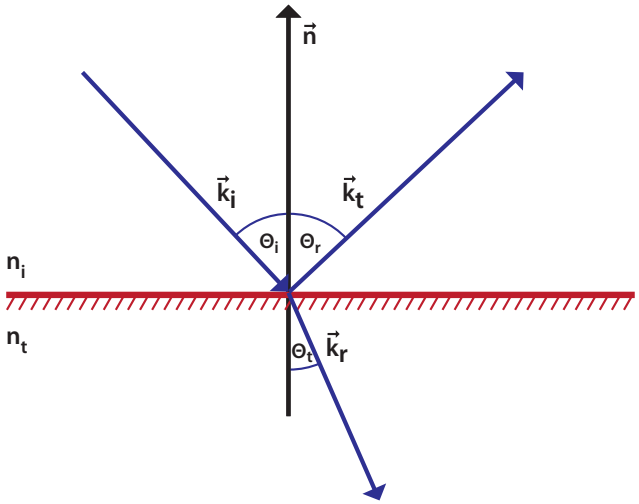


Figure 1.1.: A plane wave with wave vector $\vec{k}_i$ which incidents on an interface between two media with index of refraction n_i and n_t. It is split up into a reflected wave with wave vector $\vec{k}_r$ and a transmitted wave with wave vector $\vec{k}_r$, this is known as refraction. The relations between these three waves are dependent on the index of refraction of the involved media as well as on the angles Θ_i,Θ_r and Θ_i of incidents, reflection and transmission.

Additionally, as the interface is translation invariant the condition must stay valid independent of the particular position vector $\vec{r}$ on the interface. Consequently, the ratio $\frac{\exp\left(-i(\vec{k}_{\mathrm{i},\parallel}\vec{r}-\omega t)\right)}{\exp\left(-i(\vec{k}_{\mathrm{r/t},\parallel}\vec{r}-\omega t)\right)}$ between incident and reflected or transmitted wave, respectively, has to be independent of $\vec{r}$. This can only be true if:

$$\vec{k}_{\mathrm{i},\parallel} = \vec{k}_{\mathrm{t},\parallel} = \vec{k}_{\mathrm{r},\parallel}, \tag{1.31}$$

which is known as the *phase matching condition*. Accordingly, the wave vectors of incident, transmitted and reflected wave span a plane, which is referred to as the *plane of incidence*.

Furthermore, each wave vector has to satisfy the equation (1.12) for the particular medium which leads to:

$$\left|\vec{k}_{\mathrm{i/r/t}}\right| = \frac{\omega}{c_0} n_{\mathrm{i/t}}. \tag{1.32}$$

Introducing the angles $\Theta_\mathrm{i}, \Theta_\mathrm{r}, \Theta_\mathrm{t}$ of incidents, reflection and transmission as shown in figure 1.1, the last two equations can be combined to:

$$\left|\vec{k}_{\mathrm{i},\parallel}\right| = \frac{\omega}{c_0} n_\mathrm{i} \sin(\Theta_\mathrm{i}) = \left|\vec{k}_{\mathrm{r},\parallel}\right| = \frac{\omega}{c_0} n_\mathrm{i} \sin(\Theta_\mathrm{r}) \quad \Rightarrow \sin(\Theta_\mathrm{i}) = \sin(\Theta_\mathrm{r}), \tag{1.33}$$

$$\left|\vec{k}_{\mathrm{i},\parallel}\right| = \frac{\omega}{c_0} n_\mathrm{i} \sin(\Theta_\mathrm{i}) = \left|\vec{k}_{\mathrm{t},\parallel}\right| = \frac{\omega}{c_0} n_\mathrm{t} \sin(\Theta_\mathrm{t}) \quad \Rightarrow n_\mathrm{i} \sin(\Theta_\mathrm{i}) = n_\mathrm{t} \sin(\Theta_\mathrm{t}). \tag{1.34}$$

Here, the first equation is known as the *law of reflection* while the latter equation is referred to as *Snellius law of refraction*.

1.3.1. The Fresnel equations

It follows from Maxwell's equations that the tangential components of the electric field vector $\vec{E}$ and the magnetic field $\vec{H}$ are continuous over the interface. In particular this means that the sum of the tangential components of the fields of transmitted and reflected wave must be equal to the tangential component of the fields of the incident wave.

Consider the previously described interface shown in figure 1.1. The wave vectors $\vec{k}_{\mathrm{i/r/t}}$ of incident, reflected and transmitted wave are located within the plane of incidence (i.e. the x-z-plane) and the interfaces normal vector is orientated in the direction of the z-axis (see figure 1.1). Therefore, the components of the wave vectors are:

$$\vec{k}_{\mathrm{i/r/t}} = \begin{pmatrix} k_{\mathrm{x,i/r/t}} \\ 0 \\ k_{\mathrm{z,i/r/t}} \end{pmatrix}. \tag{1.35}$$

For the following discussion it is helpful to consider two orthogonal states of the electric field of the incident wave, that is two orthogonal states of polarization. In the first state, referred to as the parallel polarized π-state, the electric oscillates within the plane of incidence in the second state, known as the perpendicular polarized σ-state, the electric field vector is perpendicular to the plane of incidence. As both state form a orthogonal base, all other states of polarization can be constructed by superposition.

Assume an incident plane wave whose electric field vector $\vec{E}$ is in the σ-polarized state and is therefore orientated perpendicular (i.e. in y-direction) to the plane of incidence. Following equation (1.18), the magnetic field vector $\vec{H}_\pi$ is orientated perpendicular to $\vec{E}_\sigma$ as well as to $\vec{k}$ and therefore lies within the plane of incidence (i.e. is orientated along the x-axis). The demand for continuity of the tangential components then results in:

$$E_{\mathrm{y,i}} + E_{\mathrm{y,r}} = E_{\mathrm{y,t}}, \tag{1.36}$$

$$H_{\mathrm{x,i}} + H_{\mathrm{x,r}} = H_{\mathrm{x,t}}. \tag{1.37}$$

Using equation (1.18) the equation (1.37) can be written as:

$$(\vec{k} \times \vec{E})_{\mathrm{x,i}} = (\vec{k} \times \vec{E})_{\mathrm{x,r}} + (\vec{k} \times \vec{E})_{\mathrm{x,t}}. \tag{1.38}$$

Furthermore, it has to be considered that the electric field $\vec{E}_{\mathrm{i}}$ is σ-polarized and therefore only the y-component is a non-zero value. This leads to:

$$k_{\mathrm{y,i}} E_{\mathrm{y,i}} = k_{\mathrm{y,r}} E_{\mathrm{y,r}} + k_{\mathrm{y,t}} E_{\mathrm{y,t}}. \tag{1.39}$$

Making use of equation (1.36) this can be resolved for E_{r} and E_{t},

$$E_{\mathrm{r}} = \frac{k_{\mathrm{z,i}} - k_{\mathrm{z,t}}}{k_{\mathrm{z,i}} + k_{\mathrm{z,t}}} E_{\mathrm{i}} = r_\sigma E_{\mathrm{i}}, \tag{1.40}$$

$$E_{\mathrm{t}} = \frac{2k_{\mathrm{z,i}}}{k_{\mathrm{z,i}} + k_{\mathrm{z,t}}} E_{\mathrm{i}} = t_\sigma E_{\mathrm{i}}. \tag{1.41}$$

According to figure 1.1 and equation (1.32), the k_z values can be written as:

$$k_{z,i/r/t} = \left|\vec{k}\right| \frac{\omega}{c_0} n_{i/t} \cos \Theta_{i/r/t}, \tag{1.42}$$

$$\cos(\Theta_t) = \frac{\sqrt{n_t^2 - n_i^2 \sin(\Theta_i)}}{n_t}, \tag{1.43}$$

where the latter equation can be directly derived from Snellius law of refraction (1.34). Substituting equation (1.42) into equation (1.40) and (1.41) leads to:

$$r_\sigma = \frac{n_i \cos(\Theta_i) - n_t \cos(\Theta_t)}{n_i \cos(\Theta_i) + n_t \cos(\Theta_t)}, \tag{1.44}$$

$$t_\sigma = \frac{2n_i \cos(\Theta_i)}{n_i \cos(\Theta_i) + n_t \cos(\Theta_t)}. \tag{1.45}$$

These relations are referred to as the *Fresnel equations* for perpendicular polarized light and r_σ and t_σ are known as the *Fresnel coefficients* of reflection and transmission,respectively.

The derivation of these components can be repeated for an incident parallel polarized plane wave. In this case, the derivation process is similar when electric and magnetic fields are exchanged. This leads to the Fresnel equations for parallel polarized light and the associated Fresnel coefficients, given by:

$$r_\pi = \frac{n_t \cos(\Theta_i) - n_i \cos(\Theta_t)}{n_t \cos(\Theta_i) + n_i \cos(\Theta_t)}, \tag{1.46}$$

$$t_\pi = \frac{2n_i \cos(\Theta_i)}{n_t \cos(\Theta_i) + n_i \cos(\Theta_t)}. \tag{1.47}$$

Energy transfer at the interface

As aforementioned, the electric and magnetic field of electromagnetic waves as used in optics are not directly accessible by measurement. Instead it is the intensity of the electromagnetic wave which is of particular interest.

Consider a plane wave which incidents onto an interface as shown in figure 1.1. The normal vector of the interface does not generally coincident with the wave vectors of the incident, reflected or transmitted wave. For this reason, equation (1.23) has to be used to calculate their intensities with respect to the interface. Due to energy conservation, the total energy of reflected and transmitted wave must be equal to the energy of the incident wave:

$$n_i \frac{|E_{0,i}|^2}{2\mu_0 c_0} \cos(\Theta_i) = n_i \frac{|E_{0,r}|^2}{2\mu_0 c_0} \cos(\Theta_r) + n_t \frac{|E_{0,t}|^2}{2\mu_0 c_0} \cos(\Theta_t). \tag{1.48}$$

Assume the case of a parallel polarized incident plane wave, by making use of equations

(1.46) and (1.47) the energy conservation can be written as:

$$\begin{aligned} |E_{0,\mathrm{i}}|^2 &= |r_\pi|^2|E_{0,\mathrm{i}}|^2 + |t_\pi|^2\left[\frac{n_\mathrm{t}\cos(\Theta_\mathrm{t})}{n_\mathrm{i}\cos(\Theta_\mathrm{i})}|E_{0,\mathrm{i}}|^2\right] \\ &= R_\pi|E_{0,\mathrm{i}}|^2 + T_\pi\left[\frac{n_\mathrm{t}\cos(\Theta_\mathrm{t})}{n_\mathrm{i}\cos(\Theta_\mathrm{i})}|E_{0,\mathrm{i}}|^2\right], \end{aligned} \tag{1.49}$$

where the values R_π and T_π are referred to as the reflectivity and the transmission for parallel polarized light. The corresponding values R_σ and T_σ for a perpendicular polarized wave can be derived analogous.

Any arbitrary incident plane wave can be considered to be a superposition of a parallel polarized and a perpendicular polarized wave. Assume the intensity of the parallel polarized wave to be $I_p \propto |E_{0,p}|^2$ and that of the perpendicular polarized wave to be $I_\sigma \propto |E_{0,\sigma}|^2$, then the total reflected intensity I_refl and transmitted intensity I_trns is given by:

$$I_\mathrm{refl} = R_\pi I_\pi + R_\sigma I_\sigma, \tag{1.50}$$

$$I_\mathrm{trns} = T_\pi I_\pi + T_\sigma I_\sigma. \tag{1.51}$$

1.4. Geometric optics

This section gives a short introduction into the field of geometric optics and is based one the work of Born and Wolf [6] with other references marked as necessary.

As the frequencies considered within optics are rather high (i.e. around 1×10^{14} Hz to 1×10^{15} Hz) the wavelength as given by equation (1.14) is very small. For this reason, within the field of geometric optics the limit $\lambda \to 0$ is considered and distances in the order of the wavelength are neglected. At first, this seems to be an invalid approach as the magnitude of the wave vector, as defined by equation (1.12), would approach infinity. Nevertheless, it can be shown [6] that a real scalar function Υ can be defined such that the electric and magnetic field are given by:

$$\vec{E}(\vec{r}) = \vec{e}(\vec{r})\exp(ik_0\Upsilon(\vec{r})), \tag{1.52}$$

$$\vec{H}(\vec{r}) = \vec{h}(\vec{r})\exp(ik_0\Upsilon(\vec{r})), \tag{1.53}$$

where $\vec{e}$ and $\vec{h}$ are complex vector functions of the position $\vec{r}$. Substituting these field vectors into the time-free form of the Maxwell equations one can deduce the following relations between $\vec{e}$, $\vec{h}$ and Υ:

$$(\nabla\Upsilon)\times\vec{h} + \epsilon\vec{e} = 0, \tag{1.54}$$

$$(\nabla\Upsilon)\times\vec{e} - \mu\vec{h} = 0, \tag{1.55}$$

$$\vec{e}(\nabla\Upsilon) = 0. \tag{1.56}$$

If equation (1.55) is resolved for $\vec{h}$ and this is substituted into equation (1.54) this results in:

$$(\nabla\Upsilon)\times((\nabla\Upsilon)\times\vec{e}) + \epsilon\mu\vec{e} = 0. \tag{1.57}$$

This can be rewritten using vector identities to:

$$\begin{aligned} \left\{(\nabla\Upsilon)\left[(\nabla\Upsilon)\,\vec{e}\right] - \vec{e}\,(\nabla\Upsilon)^2\right\} + \epsilon\mu\vec{e} &= -\vec{e}\,(\nabla\Upsilon)^2 + \epsilon\mu\vec{e} = 0 \\ &\Rightarrow \quad (\nabla\Upsilon)^2 = \epsilon\mu = n^2, \end{aligned} \tag{1.58}$$

where n is the index of refraction. The function Υ is referred to as the *eiconal* and equation (1.58) is known as the *eiconal equation* which is the base of geometric optics. Additionally, the planes defined by $\Upsilon(\vec{r}) = \text{constant}$ are referred to as the *geometrical wave fronts*.

1.4.1. Energy flux

Based on the eiconal equation (1.58) and the representation of the electric and magnetic fields by equations (1.52) and (1.53) the properties of rays of light can be investigated. With respect to optical simulations it is the average flux of energy through an optical system which is of particular interest. As shown in section 1.1.4 its direction is given by the average direction of the Pointing vector $\vec{S}$, which can be calculated by:

$$\begin{aligned} \left\langle \vec{S} \right\rangle &= \frac{\Re\left[\vec{e}\times\vec{h}^*\right]}{2} \\ &\overset{(1.54)}{=} \frac{\left[\nabla\Upsilon\,(\vec{e}\cdot\vec{e}^*) - \vec{e}^*\,(\vec{e}\cdot\nabla\Upsilon)\right]}{2} \\ &\overset{(1.56)}{=} \frac{\nabla\Upsilon\,(\vec{e}\cdot\vec{e}^*)}{2} = \frac{\nabla\Upsilon\,|\vec{e}|^2}{2}. \end{aligned} \tag{1.59}$$

Accordingly, the direction of energy flux is parallel to the gradient of the eiconal Υ and therefore stands perpendicular on the geometrical wave fronts. Additionally, while $\nabla\Upsilon$ is not necessarily a unit vector, making use of the eiconal equation (1.58), the equation (1.59) can also be written as:

$$\left\langle \vec{S} \right\rangle = \frac{n\,|\vec{e}|^2}{2}\left(\frac{\nabla\Upsilon}{n}\right) = \frac{n\,|\vec{e}|^2}{2}\vec{s}, \tag{1.60}$$

with $\vec{s}$ being the unit vector in the direction of the average energy flux. Consequently, using geometric optics instead of wave optics, the average energy flux through an optical system can be described if the rays of light are orientated perpendicular to the geometric wave fronts.

2

Introduction to object orientated programing

From the most general point of view, any program is essentially made up by two components : *data* and *functions*. While data represent the current state of the program, functions manipulate the data to propagate from one state to another. Following this, the challenge of any program is to start from an initial state, specified by the users input, and propagate towards a final state, which contains a representation of the requested output.

For example, a program which calculates the result of $(a+b)^2$, with a and b being user defined inputs, may proceed as sketched in figure 2.1. First, the initial state is set to the user defined inputs. By applying the +-operator on the variables a and b and assigning the result to the variable c, an internal state is reached[1]. The further application of the $*$-operator leads to the final state, which has the result stored in d.

Although this general description of a program is valid independent of the used programing language the actual structural organization of data and functions during programing does heavily dependent on it. Historically, there were developed different structural styles, so called *programing paradigms*. The DAIDALOS ray tracing framework was developed using the Java® programing language. Therefore, to achieve a deeper understanding of this work a basic knowledge of the programing paradigm used by Java, the so called *object orientated programming* (OOP) [7–11], is advantageous.

While the OOP covers a broad range of different concepts, this chapter will be constraint to the features needed to understand the following chapters. On the one hand, this includes the general concept of classes and objects which are the base of any Java program. On the other hand, the concept of interfaces is heavily used within the DAIDALOS framework. Both concepts are introduced using the example of the aforementioned calculation of the first binomial.

[1] Here, an *internal state* is defined as a state of a program, which is not directly accessible by the user.

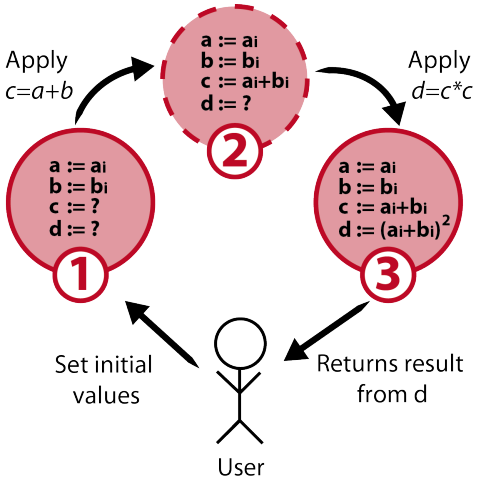


Figure 2.1.: Calculation of $(a + b)^2$ seen as a stepwise modification of an initial state. After the user initializes the first state (1), the execution of the program starts by applying the +-operator. Through this, the state (2) is reached. Further application of the $*$-operator leads to the final state (3), which holds the result of the calculation.

2.1. Classes and objects

The OOP is based upon the concept of *classes* and *objects*. A class is a construct designed by the programmer while writing the program, used to clearly arrange a program into meaningful components. It consists of a set of variables (referred to as *attributes* within the OOP) that represent the data hold by this class and functions (known as *methods* within OOP) that work on this data.

At runtime[2] the class is used as a blueprint to create an object which represents the class in computer memory. The process of deriving an object from a class-blueprint is referred to as *instantiation* and each object is called an *instance* of the class. Multiple instances may be created from one class blueprint. While each of them has its own internal variables and therefore its own state, all objects derived from the same class share the same methods and therefore the same behavior.

For example, the simple calculation shown in figure 2.1 may be implemented as the FirstBinomial-class shown in the class diagram in figure 2.2. Within such a diagram, a class is represented by a rectangular box consisting of three compartments. These hold the class name as well as the attributes of the class and its methods. While some methods have to be accessible from outside the class (e.g. the user has to be able to set the initial values of a and b), other methods only exist for internal purposes (e.g. the add-method). Therefore, the OOP allows to specify access rights for methods and attributes. Within the class diagram these access rights are presented by an access prefix which marks a method or attribute as either public (denoted by a + prefix) or private (denoted by a −

[2]The time interval in which the program is actually executed.

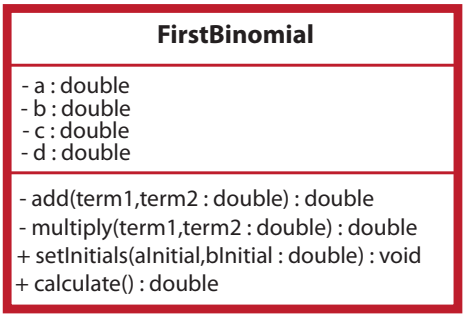


Figure 2.2.: A possible implementation of the first binomial calculation by means of a FirstBinomial class. The class diagram represents the class as a rectangle consisting of three compartments. These hold the name of the class, its attributes and its methods. The prefixes + and − mark an attribute or method as public and private, respectively.

prefix). The declaration of an attribute follows the scheme:

$$\textbf{attribute} \rightarrow [\text{AccessPrefix}][\text{AttributeName}] : [\text{DataType}],$$

where the data type double stands for a floating-point number of double precision. Correspondingly, the declaration of a method follows the similar scheme:

$$\begin{aligned} \textbf{method} &\rightarrow [\text{AccessPrefix}][\text{Name}](\textbf{paramList1}; \textbf{paramList2}; \ldots) : [\text{ReturnType}], \\ \textbf{paramList} &\rightarrow [\text{ParameterName1}], [\text{ParameterName2}], \ldots : [\text{DataType}]. \end{aligned}$$

As the name implies, the return type of a method represents the type of the result that is returned by the method, e.g. a floating-point number with double precision for the shown calculate-method. A return type declared as void specifies a method which doesn't return any value.

2.2. Interfaces

While the use of classes provides an approach for creating a meaningful component structure an application which only relies on them can be hard to maintain. This is due to the case that in order to fulfill the applications task different classes have to work together which inevitable leads to dependencies between them. This inter-class dependency, also known as *coupling*, can result in complications when classes (i.e. their features) should be changed during subsequent maintenance.

For example, consider an application whose single task is the calculation of the first binomial for two user-defined numbers. For the sake of simplicity this application was developed to consist of two classes (see figure 2.3), the FirstBinomial class as shown in figure 2.2 and a UserInterface class. This class provides two methods, namely

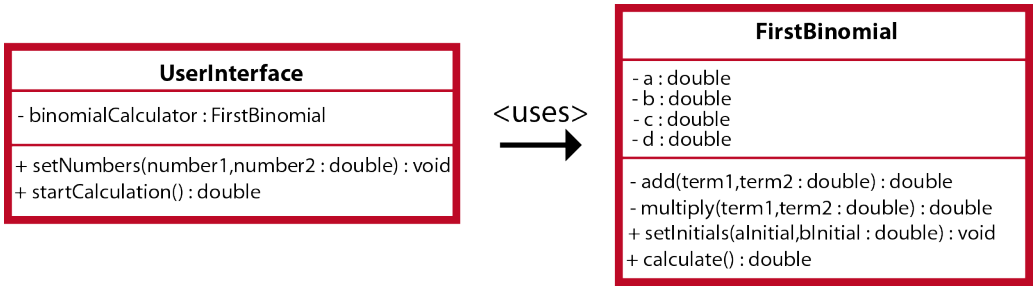


Figure 2.3.: The relationships between the two classes of a hypothetical application used to calculate the first binomial of two user-defined numbers.

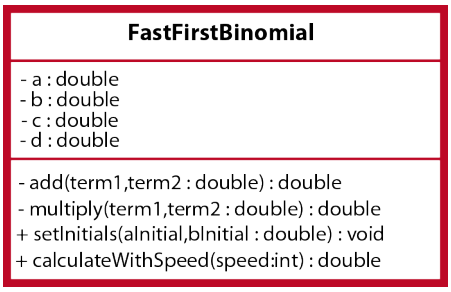


Figure 2.4.: A faster implementation of the first binomial calculation. Compared to the initial FirstBinomial class, the calculate method is changed. Now a speed parameter is required to choose between different speeds of execution.

setUserValues(double,double) and startCalculation(). However, the UserInterface class additionally contains a private attribute of type FirstBinomial which stores an instance of the FirstBinomial class. This instance is used internally by the startCalculation() method to perform the actual calculation. That is, whenever the user starts the calculation by executing startCalculation from the user-interface, this method itself delegates the calculation to the stored instance of FirstBinomial.

While this approach would provide the desired functionality it has one major drawback. By specifying the binomialCalculator attribute to be of the type FirstBinomial, the UserInterface class gets coupled to the FirstBinomial class. To visualize, consider the case that eventually the application should receive an update. With this the FirstBinomial class should be replaced by a more sophisticated FastFirstBinomial class (figure 2.4).

Despite the fact that both classes do the same thing, namely the calculation of the first binomial, their implementation is quite different. While the FirstBinomial-class provides the calculate-method to start the actual calculation, this method is not implemented by the FastFirstBinomial-class. Instead a calculateWithSpeed(int)-method is provided which requires an additional speed parameter. Even if names and parameters were equal, the UserInterface-class explicitly define its binomialCalculator-attribute to be of type FirstBinomial. Due to this the UserInterface-class is strongly coupled to the FirstBinomial-class by explicitly relying on this type as well as on its implementation details (i.e.

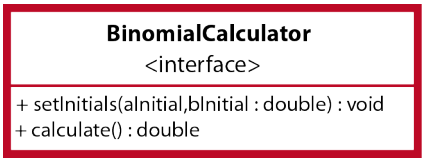


Figure 2.5.: The presentation of the BinomialCalculator-interface. Compared to the presentation of a class, an interface can only consist of public methods (i.e. no attributes). Additionally, the <interface >- keyword is added to distinguish an interface from an equal looking class-definition.

methods and their parameters). Therefore, whenever the coupled component should be replaced or its implementation changes the UserInterface has to be changed accordingly. Considering real-world applications, which can easily consist of hundreds of classes, a strong coupling can lead to *ripple-effects*, where the change of one class propagates through an entire part of the application.

Fortunately, within the OOP their exists a concept, referred to as *interface*, by which the above problems can be solved[3]. An interface can be visualized as a contract by which a class assures the implementation of certain methods. For this, the interface specifies a set of methods but unlike a class it never provides an implementation for them. Instead, the class which implements the interface has to provide an implementation of these methods to satisfy the contract. For example, a BinomialCalculator-interface can be defined as shown in figure 2.5. In addition, the calculator classes FirstBinomial and FastFirstBinomial are required to implement this interface. Within the class diagram (figure 2.6) the implementation of an interface is shown as an arrow consisting of a dashed line and non-filled head. Following the contract specified by the interface, both classes have to provide the methods as specified by the interface. As this is already the case for the FirstBinomial-class no changes have to be made to this class. However, concerning the FastFirstBinomial-class a new calculate-method has to be introduced. One approach to implement this method would be to internally execute the calculateWithSpeed-method, using a default value for the speed-parameter.

Finally, the UserInterface-class is adjusted for defining the binomialCalculator-attribute to be of type BinomialCalculator. By relying on the abstract definition of a binomial calculator rather than on its concrete implementation the coupling between the user-interface and its calculators is reduced with the result of a significantly ease on later changes (e.g. by means of maintenance).

[3]One can argue that the above issues can also be solved by the use of inheritance (e.g. both calculators can be derived from one parent class). While that's true the concept of inheritance (with respect to classes) is not needed to understand subsequent chapters and therefore not part of this short introduction.

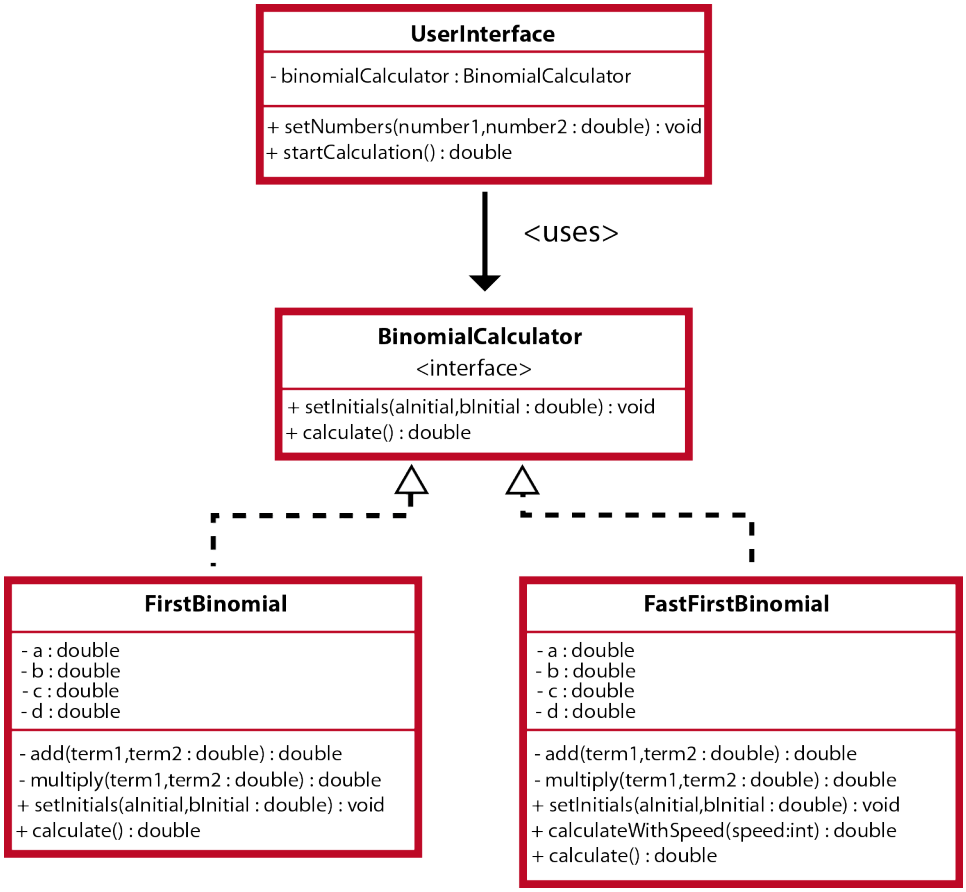


Figure 2.6.: The BinomialCalculator interface introduces an additional level of abstraction. By relying on this abstract definition of a calculator instead of its concrete implementation, the coupling between the UserInterface and the calculator implementations is significantly reduced.

2.2.1. Inheritance with respect to interfaces

While an interface was described to be a contract, which has to be satisfied by an implementing class, it may equally be seen as a kind of *tag* that indicates the membership to a group of similar objects (e.g. the group of binomial calculators). In some cases, such a group should be refined to provide access to the functionality which is exposed only by a part of its members.

To visualize, assume that a subgroup of the available binomial calculators provide a method getSum() to retrieve the sum of the initial values. As the user interface only sees the BinomialCalculator interface it can not make use of methods which are not part of this interface. However, adding the getSum() method to the BinomialCalculator interface would break the contract with those classes which already implement the interface but do not provide the new method.

One approach to solve this problem is by using a OOP concept known as *inheritance*. Following this, a child interface SummingBinomialCalculator can be derived from the BinomialCalculator interface. In this case, the child interface inherits all methods provided by its parent in addition to those defined by itself. Therefore, the child interface can be seen as a specialization of its parent. Within a class diagram the usage of inheritance is indicated by an arrow with a straight line and a non-filled head.

With respect to the above problem, assuming that the FirstBinomial class provides the mentioned getSum() method while the FastFirstBinomial does not, a possible class diagram is shown in Figure 2.7. By implementing the SummingBinomialCalculator interface the FirstBinomial is still of the type BinomialCalculator. During runtime the UserInterface can evaluate the used BinomialCalculator to check whether it also provides the SummingBinomialCalculator interface, in which case it can provide the additional sum-information to the user. Also note that by using this approach the UserInterface class retains its low coupling by still relying only on abstract interfaces instead of concrete implementations.

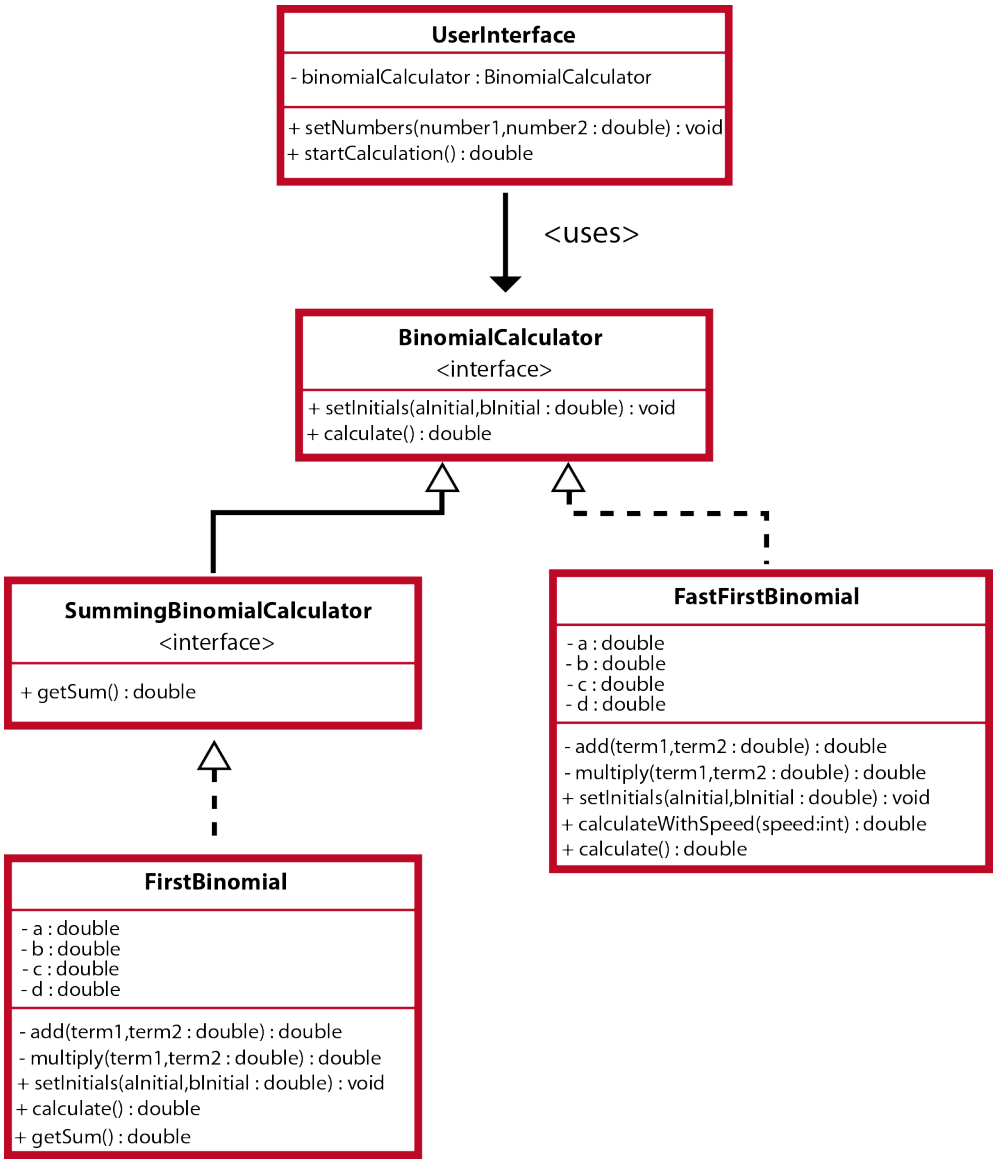


Figure 2.7.: By deriving a new interface (e.g. the SummingBinomialCalculator) from an existing interface (e.g. the BinomialCalculator), a class which implements the derived interface (e.g. FirstBinomial) provides both interfaces. This allows for a group of similar objects, described by the parent interface, to contain subgroups of objects with extended functionality that is provided through the derived interface.

3 Ray tracing

An *optical system*, like a solar cell, usually consists of several components of different material and shape. Each component has an impact on light propagating within the system, leading to refraction, reflection or absorption. Although the optical characteristics of these single components are usually well-known, multiple reflections and scattering on rough surfaces result in a propagation of light, that are not easily described by analytical methods. For this reason, such systems are often investigated by tracing the propagation of light throughout the system. This way, effects like scattering are included in a native way.

In the general case, optical simulations have to consider light as an electromagnetic wave including effects like diffraction and interference [6]. This is done with a standard approach, the *finite-difference time-domain (FDTD)* [12–15] method, or alternatively by other approaches such as the *finite element method (FEM)* [16, 17].

However, in the simulation of solar cells, wave effects can usually be neglected and geometrical optics can be used, which significantly increases performance. This work uses a statistical concept known as *Monte-Carlo particle tracing* [18–20]. In the first part of this chapter the fundamentals of this concept are introduced by describing its application to the events of reflection, transmission and absorption. This is followed by a discussion of the statistical error which is inevitable introduced by any statistical approach.

3.1. The Monte-Carlo method

The *Monte-Carlo* [18] method refers a to category of algorithms that use a stochastic approach to calculate a numerical solution of complex problems by sampling with random numbers.

The *phase space* of a system is a multidimensional space consisting of all possible states the system can be in. With respect to an optical system that is described using geometric optics the phase space can be defined as a seven dimensional space. Every point of this space consists of three values (x, y, z) to specify an initial position of a propagating ray of light, three values $(d_\mathrm{x}, d_\mathrm{y}, d_\mathrm{z})$ to specify its direction of propagation and one value I

to describe its intensity. Any path by which a ray propagates through the system can then be described as a trajectory T within the phase space. Moreover, the stochastic average value $\langle Q \rangle$ of any quantity Q that can be derived for any particular trajectory T can principally be calculated by:

$$\langle Q \rangle = \sum_{\mathrm{T}} \rho(T) Q_T. \tag{3.1}$$

Here, the sum is taken over all possible trajectories in the phase space, Q_T is the value of Q derived for a particular trajectory T and $\rho(T)$ is a normalized weighting factor that describes the probability for a propagation along T. Using the Monte-Carlo approach the value of $\langle Q \rangle$ is approximated by simulating an ensemble composed of N rays of light that propagate along randomly chosen trajectories T_i. For each of these trajectories, the value Q_T is calculated and the average value calculated by:

$$\langle Q \rangle \approx \frac{1}{N_i} \sum_{\mathrm{T}} Q_T. \tag{3.2}$$

This approach provides the advantage that trajectories with high probability $\rho(T)$ and therefore high significance are more frequently considered then trajectories with low significance, this is referred to as *importance sampling*. The calculation of the statistical error that is introduced due to the limited number of considered trajectories depends upon the actual implementation of the Monte-Carlo method. It is discussed in section 3.3 for the approach of Monte-Carlo particle tracing.

3.2. Monte-Carlo particle tracing

This section introduces an implementation of the Monte-Carlo method, referred to as *particle trancing*, which is used for the optical simulations as described within this work. In Monte-Carlo particle tracing, a ray of light is represented by an ensemble of N particles, which are referred to as *simulated photons* or just *photons* in this work. The state of each simulated photon is described by its position vector $\vec{x}$, its propagation direction $\vec{d}$ and a value λ that specifies the simulated wavelength. Within this representation each photon can be considered to transport a fraction γ of the rays intensity I, given by:

$$\gamma = \frac{I}{N}, \tag{3.3}$$

where I is defined by equation (1.23).

Different trajectories are randomly sampled by individually simulating the propagation of these N photons, where each path is followed as illustrated in figure 3.2. First, the simulated photon is created by a light source and initialized with its initial position $\vec{x}$, direction $\vec{d}$ and wavelength in vacuum λ. After its creation the photon is located within a particular medium with an index of refraction n_1 as specified by the simulation domain. Using the photon's position $\vec{x}$ and direction $\vec{d}$, a search for the next hit with an

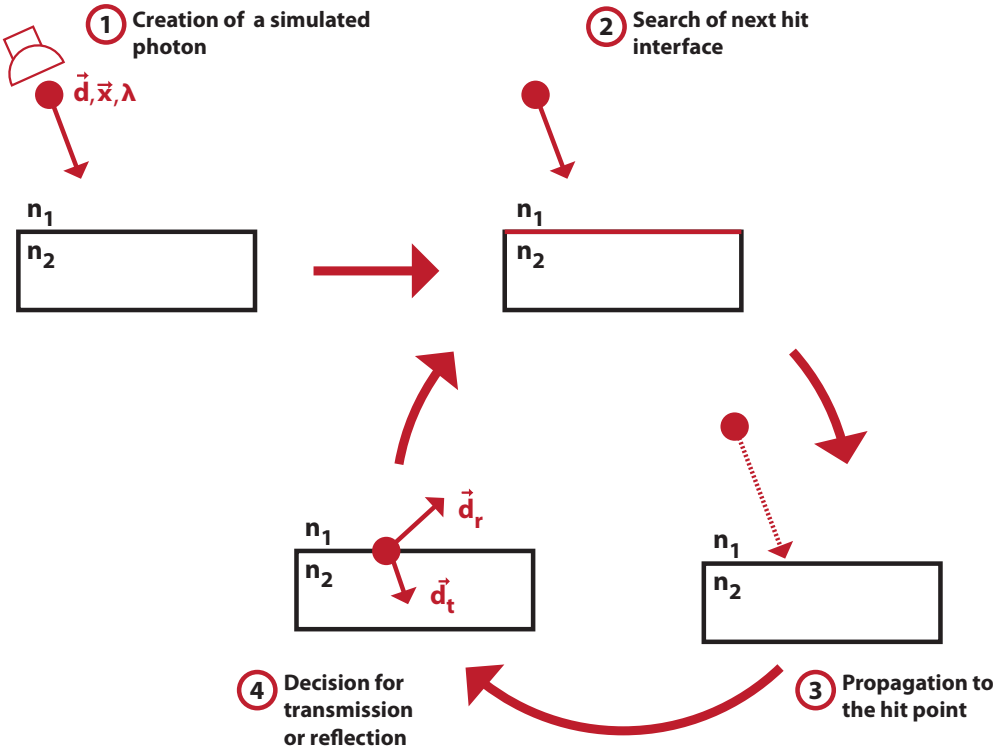


Figure 3.1.: The path of each individual photon is followed in a cyclic manner. In the first step, the photon is created by a light source as defined by the simulation (1). Using its direction of propagation $\vec{d}$ and its position $\vec{x}$ the next hit interface is determined. In case a hit is found the photon is propagated within the current medium to the hit point (3). During this step a possible absorption of the photon by the medium is checked (see text). If not absorbed, the photon reaches the interface. In this case the probabilities for transmission and absorption are calculated (4). Based on this probabilities it is decided whether the photon is reflected by the interface or transmitted through the interface (see text). In any case the direction vector $\vec{d}$ is changed to comply with the law of reflection or Snellius law of refraction. The shown cycle is executed until the photon is either absorbed by the medium or leaves the simulation domain.

interface is conduct, which in the general case separates two media with different index of refraction n_1 and n_2.

If a hit can be found, the photon is propagated along its direction to the hit point on the interface. Depending upon the used material as described by n_1 the photon can get absorbed during this propagation. This absorption process is described by the Lambert-Beer law of absorption as given by equation (1.29). It describes an exponential attenuation by a factor $\Omega(\alpha, \Delta x) = I(\Delta x)/I(0) = \exp(-\alpha \Delta x)$ of a ray's intensity after propagating a distance Δx within a medium with an absorption coefficient α. As each photon transports the same fraction γ of the intensity and all photons are equally handled during simulation, the probability for a single photon to get absorbed is defined by $\Omega(\alpha, \Delta x)$. For this reason, before propagating the photon to the hit point, the distance to the hit point is calculated and $\Omega(\alpha, \Delta x)$ is evaluated. Additionally, a random number m_{abs} in the range from 0 to 1 is generated by means of a pseudo-random number generator. In the case of $m_{\text{abs}} > \Omega(\alpha, \Delta x)$, the photon is absorbed, otherwise it is propagated to the hit point.

For all photons that reach the interface it has to be decided whether they are reflected by the interface or transmitted through the interface. Following a similar reasoning as for the process of absorption, the probability for a photon to get reflected by interface is equal to the interface's reflectivity R and can be calculated by Fresnel theory as introduced in section 1.3. A second pseudo-random number m_{refl} is generated and the photon is reflected by the interface for $m_{\text{refl}} < R$ or transmitted through the interface otherwise. After the decision, the direction vector $\vec{d}$ of the photon is changed to comply with the law of reflection (equation 1.33) or the Snellius of refraction (equation 1.34), respectively.

The cycle of hit search, propagation and refraction calculation is repeated until the photon is either absorbed by a medium or no further hits can be found, which means that the photon leaves the simulation domain.

After conducting a particle tracing simulation using an ensemble of N_{i} incident photons, the optical characteristics of the investigated optical system can be evaluated using equation (3.2). For example, the reflectivity of a silicon wafer for a fixed wavelength λ can be evaluated as the stochastic average of the number of reflected photons N_{r} for an ensemble of N_{i} incident photons. Making use of equation (3.2), this can be written as:

$$R = \langle N_{\text{r}} \rangle \approx \frac{1}{N_{\text{i}}} \sum_{\text{T}} N_{\text{r}}(T) = \frac{N_{\text{r}}}{N_{\text{i}}}, \tag{3.4}$$

where the sum is executed over the trajectories the N_{i} simulated photons and $N_{\text{r}}(T)$ is the number of reflected photons due to particular trajectory T. Other optical characteristics, like transmission and absorption, can be evaluated by a similar calculation.

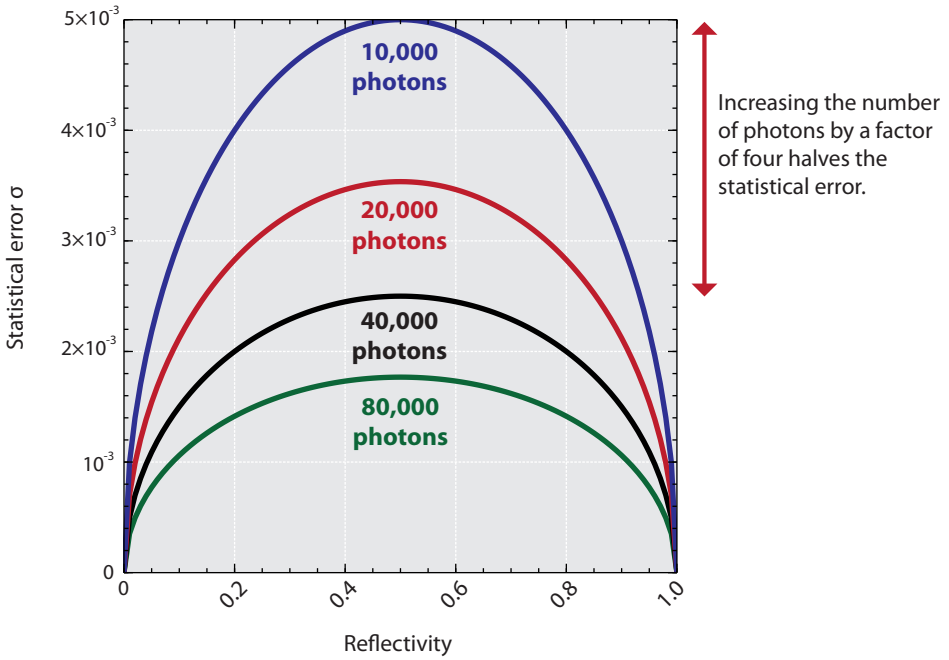


Figure 3.2.: The statistical error σ as expressed by equation (3.9) for different values of the number of incident photons and the value of the reflectivity. Increasing the number of simulated photons by a factor of four reduces the statistical error by 50 %.

3.3. Calculation of the statistical error of Monte-Carlo particle tracing

For most optical systems there is an infinite number of paths. Thus, the number of traced paths (i.e. the number of traced photons N_i) have to be constraint to keep the simulation time reasonable. By doing this using the approach of Monte-Carlo particle tracing, a statistical error is introduced, leading to results that expose a *binomial distribution* with variance σ^2 :

$$\sigma^2 = Npq. \tag{3.5}$$

Here, N is the number of independent events, p is the probability which describes a positive outcome of the event (however this is defined) and $q = (1 - p)$ is the probability for the complimentary result. With this, the standard deviation σ can be calculated by:

$$\sigma = \sqrt{Npq}. \tag{3.6}$$

For example, consider a ray tracing simulation where a set of N_i incident photons results in a number of N_r reflected photons. According to equation (3.4), this leads to a reflectivity value of:

$$R = \frac{N_r}{N_i}. \tag{3.7}$$

As not all possible paths were covered, the value of N_r is afflicted with a statistical error which, following equation (3.6), can be approximated by:

$$\sigma_{N_r} \approx \sqrt{N_i R(1-R)}. \tag{3.8}$$

This leads to a statistical error σ_R for the value of the reflectivity R:

$$\sigma_R \approx \frac{1}{N_i}\sqrt{N_i R(1-R)} = \sqrt{\frac{R(1-R)}{N_i}}. \tag{3.9}$$

This statistical error is shown in figure 3.3 for different values of reflectivity R and simulated photons N_i. As can be seen from the figure and is expressed by equation (3.9), the error is halved when the number of simulated photons is increased by factor of four.

4

Daidalos - A framework for flexible ray tracing

This chapter introduces the concept behind the DAIDALOS framework approach, starting from the most fundamental point, the comparison of monolithic to modular applications [21]. Here, the term *framework* refers to a set of tools or pieces of software that are used to simplify the creation of a second application. This ranges from mathematic frameworks that provide functions for e.g. efficient calculation of matrix products, to user-interface (UI) frameworks that provide common UI elements like buttons or text input-fields. Usually, such frameworks can not be executed as a standalone application, but rely on a second application to use them. This second application will be referred to as *host application*, as it is the host for the used framework, throughout this chapter.

In the later sections, the utilized modularization framework as well as the basic elements provided by DAIDALOS are presented. This is finished by an introduction to the possibilities, namely the types of plugins, which can be developed by any user.

4.1. Monolithic vs. modular applications

Today, any well-designed application is structured into different components with each component being responsible for a fraction of the overall task. The resulting component-structure often is an hierarchic one, where some low-level components (e.g. loading data from disk or doing matrix calculations) are used by some more abstract high-level components. While a component hierarchy is used for monolithic applications as well as for modular ones, there exists a big difference considering the moment in time when the actual hierarchy is established. Regarding a monolithic application (figure 4.1), all components are developed at once, leading to a full knowledge of the actual component hierarchy at the time the application is programed. This fact provides two advantages. First, a developer can rely on the existing components, knowing that each of them will be also existent at runtime. Second, the compiler[1] has access to all parts of the

[1] A *compiler* is an application which translates code written in a programming language (e.g. Java) into the much lesser readable machine code needed for execution on a particular machine (e.g. the Java virtual machine).

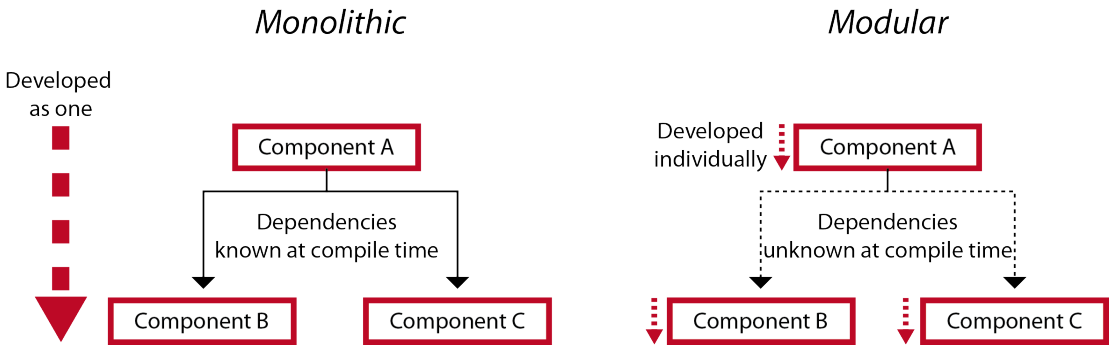


Figure 4.1.: A monolithic application (*left*) is developed as one, knowing the dependencies between different components while it is programed. Contrary to that, the components of a modular application (*right*) are separated into different modules, each individually developed. As the actual components are not loaded until runtime, no assumptions can be made about the existence of particular components during development.

application, which allows him to notify the developer of unsatisfied dependencies (e.g. some components which rely on older, but now removed, components).

There is a significant difference when it comes to a modular application. Here, components or groups of them are distributed within different *modules* (figure 4.1), each of them providing a group of functions to the overall application (e.g. basic math calculations are provided by one module). There exists no strong coupling between the modules, instead a module providing a particular set of functions can be replaced by another one providing the same set. At the moment the host application is started, it chooses the needed modules and establishes the final component hierarchy. Compared to a monolithic application, the modular approach provides greater flexibility, allowing for the replacement of any module at any time even after the compilation of the host application. However, two new tasks have to be tackled if using the modular approach.

First, there has to exist a mechanism to manage the different modules used by the host applications. That means, as the host application is required to choose and load the needed modules[2], there has to be a mechanism which allows for communication between modules and application to exchange information like module type or module version. Furthermore, as the application finally has to establish a valid component hierarchy, the used modularization mechanism has to ensure that the dependencies between all loaded modules can be satisfied. Considering different module versions, where any of them may have other dependencies, this is a rather complex task.

Second, there has to exist a specification which describes the interfaces that allow communication between different modules. That is, component developers have to know which functions their modules should provide and which function they can rely on as

[2]Note, there can always be more than one module providing the required set of functions, so the host application actually has a choice which one to use.

being provided by other modules. Therefore, the types of modules which exist have to be specified as well as the functions they provide.

Considering the DAIDALOS framework, the responsibility for these tasks is split into two parts. The modular DAIDALOS framework specifies the available types of modules, which are also referred to as *plugins*, and the functions they provide. An introduction to its elements is given in section 4.3. The needed modularization mechanism which manages modules as well as their interdependencies is provided by the *OSGi service platform* and is subject of section 4.2.

4.2. The OSGi service platform

The *OSGi service platform* supports the development of modular applications by specifying a fundamental concept which covers the task of defining modules and their dependencies as well as managing (that is, loading, configuring, etc.) them during runtime. Amongst a set of interfaces that allows modules to present their provided features, the service platform includes a bunch of already written code to ease recurring tasks during module management. The term *OSGi* was originally an acronym for the *Open Services Gateway Initiative*. It was dropped after the third release of the OSGi service platform and since then serves as a general trademark for the associated technology [4, 22, 23].

This chapter will introduce the basic mechanisms provided by the OSGi service platform. While OSGi provides a broad set of features to support modularization, this chapter will be constrained to those needed to understand the modularization concept as used by DAIDALOS.

4.2.1. Java archives

Applications written with the Java programming language [24] usually consist of a number of different classes which work together to provide the applications functionality. In order to keep the application well-arranged Java provides a *package*-concept. Any package is comparable to a folder on the disk of a computer and can contain a number of classes as well as sub-packages. During the compilation process, the compiler translates the Java program code of these classes into binary code which could be interpreted by the Java virtual machine (JVM) [25]. As a result, a folder structure is created on disk which reflects the initial package-structure wherein any original class definition is replaced by the binary code outcome of the compilation process.

In the usual compilation process the compiling step is followed by a packaging step. Within this step the created folder structure is compressed into a *Java archive*(.jar) file. Additionally, the compiler adds a *MANIFEST.MF* file which is referred to as *manifest file*. It contains information about the archive and is used by the JVM to execute the contained program.

For example, the manifest includes the name of the class which contains the main()-method of the application. This is the first method that is executed when the application starts and is also referred to as the *entry-point* of the program.

Finally, the created .jar-file can be distributed and executed on other computers.

4.2.2. Bundles

While Java archives allow for an easy distribution of monolithic applications, where the whole program is included within the created .jar file, it is not adequate to develop modular applications. This is mainly due to the fact that modular applications need a sophisticated mechanism for providing access to one part of a module (the public part) while restricting external access to the internal part (the private part). Such separation mechanisms can hardly be implemented with the concepts provided by Java archives.

For this reason, the OSGi service platform uses slightly modified Java archives which are referred to as *bundles*. Additionally to the components of a standard archive file, bundles are extended by an OSGi information file. On the one hand, the existence of this file marks a .jar file as being a bundle. On the other hand, it includes important information about the dependencies of the module. For example, this can include a set of packages which are exported (i.e. made available) to the public (i.e. other modules) and packages which are imported (i.e. required) by the module.

Although it is possible to spread a single module of the application over several OSGi bundles it is common to associate one module with one bundle. Following this one-module-one-bundle approach functionality can be added or removed to or from an application by adding or removing the associated module bundles.

4.2.3. Plugin enviroment

Following the one-module-one-bundle approach mentioned in the last section, a bundle is the representation of a single module on the disk of the computer. It provides basic informations over the bundle's dependencies but is a pure passive object. As mentioned in section 4.1, the module-management, consisting of loading modules in memory and resolving there dependencies, is a rather complex process. In order to simplify the control over this process, OSGi provides an environment which will be referred to as plugin environment within this work. It can be seen as an abstract space into which modules have to be loaded in order to get usable by the application. When properly loaded into the environment any module can access the functionality provided by any other loaded module.

Due to the fact that the plugin environment must exist before the first module is loaded into it, an application designed to use OSGi has to consist of at least two components. First, code responsible for initially setting up the OSGi framework is required. This starter component contains the entry point of the host application and is therefore the first one executed when the user starts the application. Second, there have to be a set of modules which contain the business logic of the application, that is the basic features needed to fulfill the application's task.

During the startup process it is at the responsibility of the starter component to load these base modules into the environment and transfer the execution control to them. Afterwards, the now started application, which is made up of base modules, is in control of the execution process. Up from here it is itself responsible for loading additional modules if needed. Eventually, when the application is shut down, the control of execution is

transfered back to the starter component, which is responsible for properly shutting down the OSGi environment before exiting.

4.2.4. Module lifecycle

Any module runs through a lifecycle which consists of three major states, namely *installed*, *resolved* and *started* [4].

When a module gets loaded by the host application it starts in the *installed* state. Within this state the module's bundle is internally accessible by the host application, but does not provide any functionality yet. Based upon this state an application can execute the *resolve* command. During the resolving process the OSGi framework tries to satisfy the dependencies of the module by those modules which have already been started. In case the dependencies are properly resolved, the module state is changed to *resolved*. At this point the module is not active yet, but can be started by the application executing the *start* command, changing the module state to *started*. When this state is reached the module is successfully loaded into the plugin environment and can provide its functionally to the application.

Eventually, the module is no longer needed, in which case the application executes the *stop* command, which removes the plugin and its functionality from the environment and changes the module state back to *resolved*.

4.2.5. Package exports and services

A module which has reached the *started* state is able to provide its functionality within the plugin environment. This can be done by two approaches, by means of *package exports* or *services* [4].

By using package exports a module defines a subset of the Java packages to be accessible for other modules. However, in order to make use of these exposed classes and interfaces the developers of other modules must know their fully qualified name[3] of the used classes when the modules are developed. For this reason, this approach is most applicable for modules that provide utility libraries (e.g. a math library). Here, the decision about the used classes is made while developing the module, therefore the fully qualified names of each used class is known in advanced.

Using the service-approach, a module registers a set of interfaces with the OSGi platform that represent the services it provides. For example, a module might declare to provide the BinomialCalculator as shown in figure 2.5 to provide the functionality to calculate the first binomial. Other modules can ask the OSGi platform for registered services with a particular interface. While this service query can be made at runtime (contrary to the import of exported packages), both modules (i.e. service provider and requester) have to agree on the interface used for the representation of the service.

Within Daidalos and most of other expandable modular applications a combination of both approaches is used. On the one hand, a publicly available library is released that

[3] The fully qualified name of a Java-class consists of the actual class-name and the name of the package it resides in.

exports packages mainly composed of the interfaces for all services used later on by the application. The definition of those interfaces is one task of the DAIDALOS framework core. On the other hand, any module makes it capabilities available by providing services represented by those interfaces.

4.3. Daidalos framework

This section covers the fundamental principles of the DAIDALOS framework. While the first part is about the underlying concept, the latter part describes the provided elements.

When developing any software framework to support the development of applications of a particular domain (e.g. a framework supporting scientific calculations), the developer has to take care of two issues. First, a concept has to be developed that describes the way in which the framework is meant to be used. This includes the paths (e.g. programming code, additional software tools, etc.) by which support is provided as well as the requirements that have to be satisfied by a host application. Second, the actual framework elements have to be implemented and made available to the using application.

4.3.1. Framework concept

An application providing the possibility of ray tracing has to take care of two tasks. It starts with a pure management task, which includes issues like the presentation of a user interface, the configuration of the elements used during simulation and finally starting the actual simulation process. Afterwards, the task is switched to a simulation orientated one, consisting of elements like the ray tracing algorithm, refraction calculation and result generation[4].

In the case of DAIDALOS, this separation of issues is an important part of the overall concept. To clarify this, DAIDALOS actually separates the simulation into two parts. During the *startup* part, the user starts an application, referred to as *host application*, which provides the ability of ray tracing by means of DAIDALOS and implements the elements described in the *startup concept* discussed later. It is an important point, that there are no other requirements to the host application[5] than being able to execute the startup process. The host application is in charge of the whole startup process, including creation of the user interface and configuration of the used plugins.

When the user starts the simulation process (e.g. by clicking some button in the user interface of the host application), the control is transferred to one of the plugins which were already loaded by the host application. This is the moment, when the actual simulation part, referred to as *runtime* part, begins. During this part, the plugins supported by the DAIDALOS framework control the process of simulation. The elements of this part as well as the process itself is what is described by the *runtime concept*.

[4] There can be an additional task-switch when it comes to displaying the generated results (e.g. in a window of the user-interface).

[5] As DAIDALOS is based on Java programing language its usage is greatly simplified if the host application is also using Java

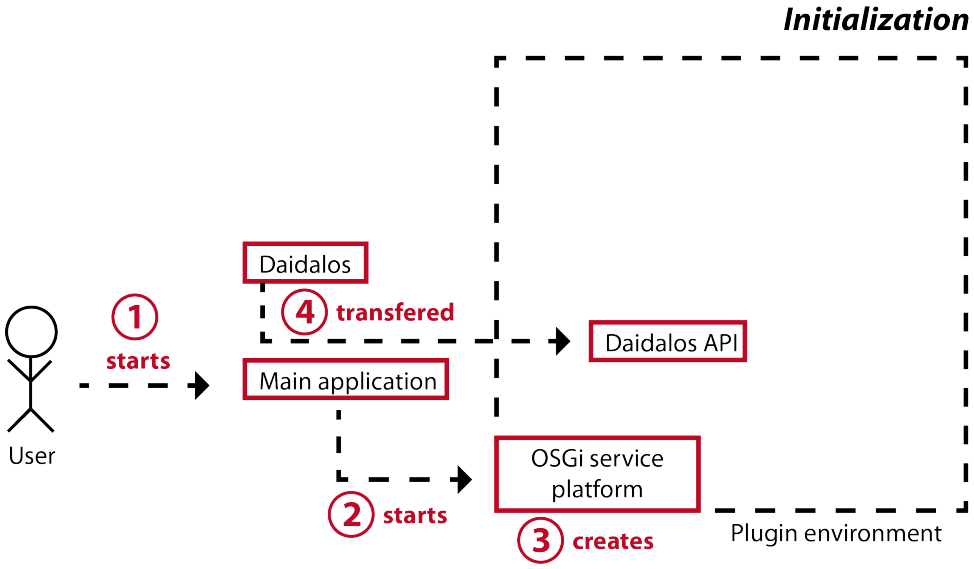


Figure 4.2.: The *initialization* stage is the first stage during startup, triggered by the user when starting the host application (1). During this stage, the host application is responsible for starting the OSGi service platform (2), which itself creates the required plugin environment (3). Afterwards, the host application transfers the DAIDALOS framework into the plugin environment (4), which represents itself by its API.

Startup concept

The startup concept of DAIDALOS is divided into three stages, namely the *initialization*, *configuration* and *finalization* stage.

The initialization stage (figure 4.2) begins with the user starting the host application (1), which is responsible for starting the OSGi service platform (2) that provides the necessary plugin environment (3) needed for proper plugin management. Additionally, the host application loads the DAIDALOS framework which is itself composed of several OSGi modules. After transferring these modules into the plugin environment (4), the framework makes itself available through its *application programming interface (API)* that is discussed in section 4.3.2. Note that the host application resides outside the actual plugin environment and has only minor impact on the later simulation, which executes within the environment.

After the host application is properly started the configuration stage is reached (figure 4.3), it provides an interface[6] to the user, allowing him to configure the simulation to the needs of his problem (5). Based upon the given configuration, the host application chooses the plugins required to fulfill the configured simulation task (6) and transfers them into the plugin environment (7). Each plugin presents itself within the environment

[6]The concept does not make any assumptions about the nature of this interface. For example, it might be a graphical user interface as well as a console based one.

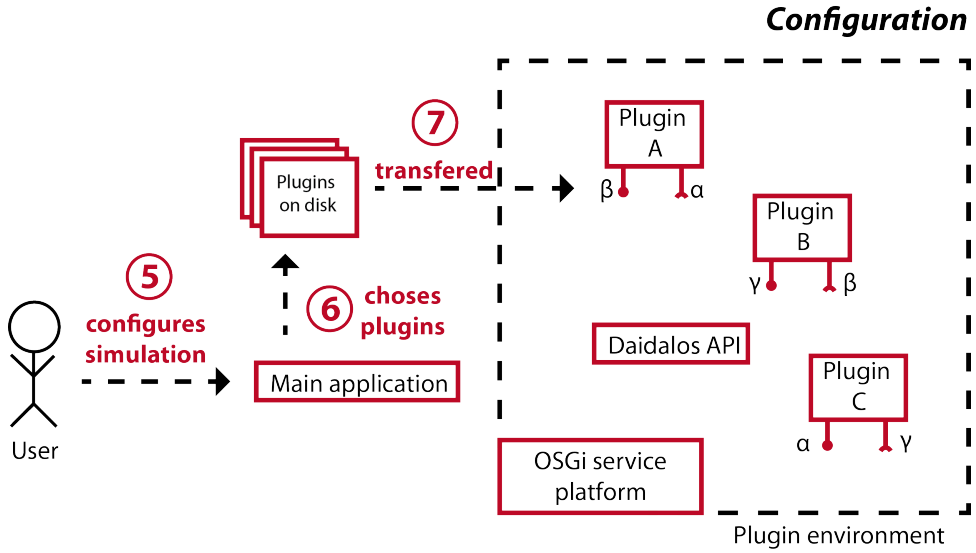


Figure 4.3.: The *configuration* stage is entered when the host application provides a configuration interface to the user. Using this interface the user configures the simulation (5) for his purposes. Based upon this configuration the host application choses the required plugins (6) and transfers them into the plugin environment (7).

by the interfaces (α, β and γ) it provides and needs. Sometime after the configuration has finished, the user starts the actual simulation process (e.g. by using some element of the user interface) and triggers the start of the finalization stage (figure 4.4). During this final stage, the host application finalizes the configuration of the used plugins. This includes the task of creating the interdependencies between different plugins and therefore establishing the final component hierarchy (9). In the last step of the startup process, the host application transfers control to a specific plugin, in this case *Plugin A* (10), that is in charge of the actual simulation process[7]. To satisfy the need for an interface of one plugin (e.g. *Plugin A*) with the one provided by a second plugin (e.g. *Plugin C*), both have to agree with respect to the methods provided by this interface. For this reason, there has to exist an external specification of the interface to which both plugins refer. Providing such interface specifications is one of the responsibilities of the Daidalos framework (see section 4.6).

Runtime concept

While the host application is the leading part during the startup process, this changes as soon as the simulation is switched to the runtime part. This part is controlled by two plugins, which represent themselves as *Tracer* plugin (see section 4.6.1) and *SceneCompiler* plugin (see section4.6.3).

[7]Usually, this will be the plugin providing the Tracer interface that is discussed in section 4.6.1.

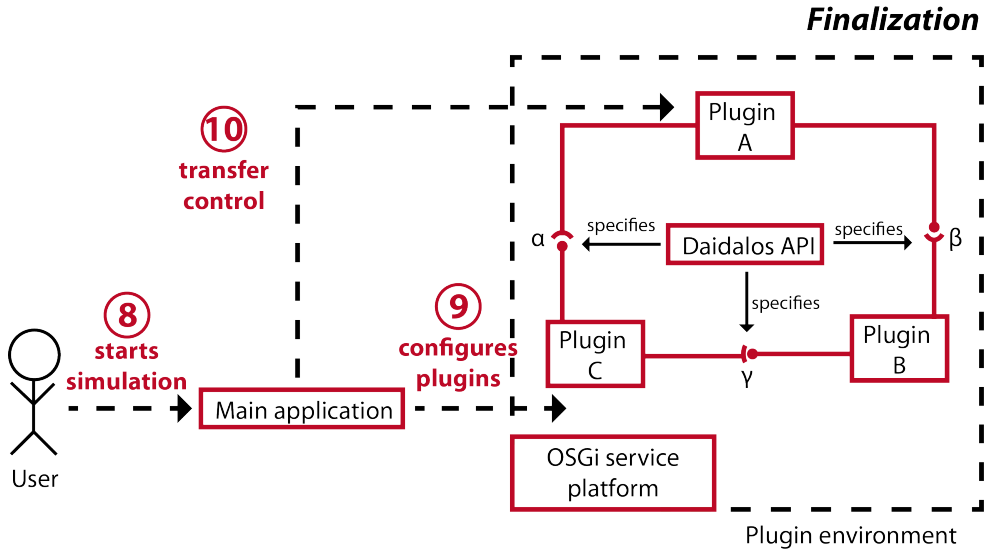


Figure 4.4.: After configuration is done the user starts the actual simulation process (8). This event triggers the *finalization* stage, in which the host application finishes the plugin configuration process and establishes the final component hierarchy (9). As last step, the host application transfers control to the *Plugin A* which controls the simulation process (10).

In DAIDALOS, a simulation consists of *geometry* and *effects*. While geometry specifies the shape, position and orientation of volumes, the effects are software components that can be associated with the geometry. During the tracing process the effects are executed by the Tracer component and can be used to manipulate the photon (e.g. its direction of propagation). They are used to implement physical effects as well as photon detectors (e.g. to count all photons reflected by a particular interface).

The runtime part (see figure 4.5) is specified to begin after the host application has transfered control to the Tracer plugin (marked 10 in figure 4.4). However, there are two steps within the final stage of the startup process which are quite important for the simulation process. First, the host application configures the SceneCompiler plugin by transferring the underlying geometry (figure 4.5a). It is important to note that DAIDALOS makes no assumptions considering the structure by which the geometry is presented. For example, the author of a host application which already has an internal representation of geometry (e.g. by a polygon model) may create a SceneCompiler plugin which accepts this structure and makes it available for simulations with DAIDALOS. Second, the host application registers all effects applied to any part of the geometry as well as geometry independent plugins, e.g. light sources, with the Tracer plugin (b).

Eventually the runtime part is started. At this point the SceneCompiler is responsible for creating a so called *Scene* object (figure 4.5,(1)). The required interface of this object is specified by the DAIDALOS framework and is kept simple to be usable with any type of underlying geometry (see section 4.6.3). After the *Scene* is created, the *Tracer* starts by

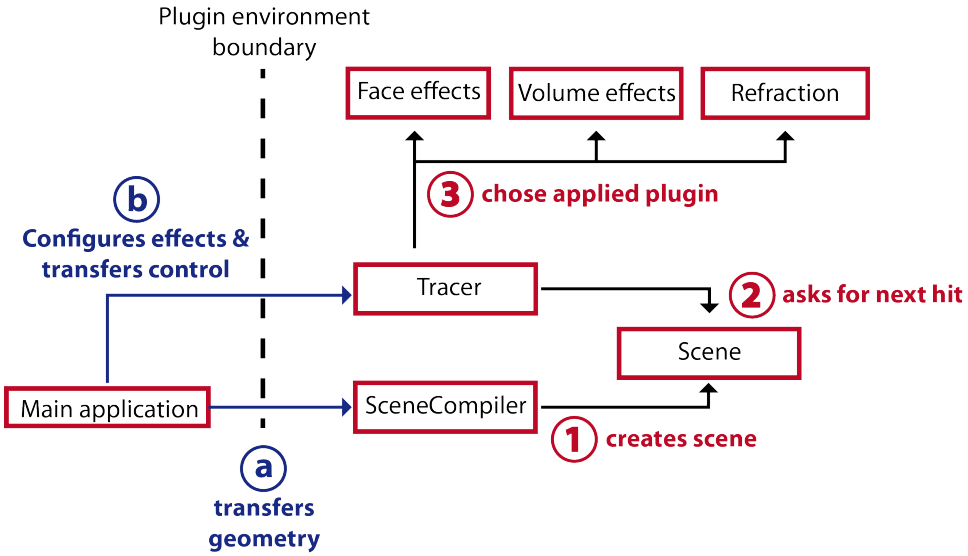


Figure 4.5.: An overview of the runtime process. During the finalization stage of the startup process (figure 4.4), the host application transfers the underlying geometry to the SceneCompiler plugin and registers the used effects with the Tracer plugin. Afterwards, the control of execution is transfered to the Tracer plugin and the actual simulation is started. During the following process, the tracer uses the Scene, created by the SceneCompiler, to retrieve the next interface hit of the photon. According to the simulation, as defined by the user, physical effects which were applied by the user are executed by the tracer.

following a tracing loop, which is discussed in section 4.4. During this loop the Tracer submits the position and direction of the simulated photon to the Scene object and asks for the next hit with an interface (2). Based upon the answer of the Scene object[8], the Tracer decides which effects should be executed (3). This process is periodically repeated until the simulated photon is either absorbed by a media or no next hit is found (i.e. the photon leaves the simulation domain). In such cases, the Tracer either triggers the creation of a new photon or finishes the simulation as no new photons are available.

4.3.2. Framework structure

In order to put the concepts described in the last section in practice, DAIDALOS has to satisfy several requirements. For the startup process elements for communication between host application and plugins are required. This includes recognition of plugin types as well as the transfer of data between host application and plugin. Finally, DAIDALOS should provide support in plugin development as well as in the integration of ray tracing into the host application.

While plugin related elements like associated interfaces and the structure of plugin bundles are separately discussed in section 4.5, this section provides an insight into the remaining elements provided by DAIDALOS to tackle the above issues.

Framework values

Any meaningful communication relies in an agreement between sender and receiver, with respect to the type of data which is exchanged. With regard to scientific communication, this data often is expressed in terms of mathematical expressions, including complex numbers, vectors or matrices. However, although Java provides a broad set of different data types, complex numbers are not part of the natively provided types.

For this reason, DAIDALOS introduces its own set of base types which are referred to as framework values. These types include three tensor types, namely complex numbers, vectors and matrices. Additionally, the Java types Boolean for logical values and String for character strings are also included.

All tensor types provide the methods needed for basic mathematical operations, e.g. matrix-matrix or matrix-vector products. Additionally, all numerical types allow for the definition of a physical unit, referred to as SIUnit. When using the provided mathematical methods these units are taken into account.

Message channels

While the framework values define the content of an exchanged message DAIDALOS provides two ways to transfer them.

[8] There are three possible answers. First, there can be only one next hit, e.g. a photon hitting the side of a cube. Second, there can be multiple hits, e.g. a photon which hits the tip of a pyramid is actually hitting all neighboring faces. Finally, there can be no hits at all, e.g. when a photon leaves the simulated scene.

First, communication can take place by using the predefined plugin interfaces as described in section 4.5. For example, the Tracer plugin (see section 4.6.1) will ask a LightSource plugin (section 4.6.2) for a photon, by calling its createPhoton method which is defined by the associated plugin interface. The advantage of this kind of communication is its well defined nature, reducing the risk of unwanted ambiguities. The main drawback of this approach is the constraint to a number of predefined messages (as defined by the plugin interfaces), leaving no space to introduce new kinds of messages where needed.

Therefore, DAIDALOS supports a second way of communication, based upon so called *message channels*. A *MessageChannel* can be visualized to be similar to a mailing list, identified by a unique name. On the one hand, any plugin can post a message, consisting of an arbitrary *FrameworkValue*, to any message channel. On the other hand, any plugin can register with any message channel as a so called *listener*. In the case, where a message is posted on a channel any listener registered with this channel gets notified of the message. While this approach allows for nearly arbitrary communication between plugins, its main drawback is the undefined nature of messages which are transfered by a message channel. Due to this, any plugin listening to a channel is itself responsible to handle messages which are not matching the expected format (e.g. a plugin expects to receive 3×3 matrices, but another plugin posts a 4×5 matrix).

To visualize the power of the message channel concept, consider the task of information transfer from a plugin to the user. A plugin needed to inform the user, either of a simulation related detail or an occurred error, has no knowledge of the user-interface (e.g. is it a console application or a graphical interface) which is created by the host application. Therefore, DAIDALOS specifies four default channel names, namely *Debug*, *Warning*, *Information* and *Error*. Any plugin wanting to inform the user of any of the above, posts its message on the associated channel. By registering a listener to each of these channels, the host application can receive the messages and decides itself on how to present it to the user.

Utilities

During the process of a simulation, there are recurring tasks which have to be done by most of the used plugins. Therefore, the DAIDALOS framework includes a set of utility classes to ease the development process and avoid redundant code.

The provided classes can be roughly divided into two groups. On the one hand, there is set of classes, e.g. ParameterList and ArgumentSet, which are used during the plugin initialization process (see section 4.5). As all plugins are initialized on a mostly similar way, the provided utility classes are of use for practically every plugin developer. On the other hand, there is set of classes which supports typical tasks done during simulation. For instance, the CubicSpline class is of great use when wavelength-dependent input values (e.g. the material specific index of refraction) should be interpolated.

Scene construction

As it was shown in section 4.3.1, the host application transfers a model of the simulations geometry to a SceneCompiler plugin, which is able to create a Scene from this model (see figure 4.4). There are two ways to conform with this concept.

On the one hand, there are some host applications which already contain an internal geometry model for other purposes and should just be extended to support ray tracing by means of DAIDALOS. For these ones, the easiest approach might be to integrate their own geometry-model into the ray tracing simulation by developing a SceneCompiler plugin that understands the model and creates a Scene from it.

On the other hand, most host applications will not provide their own geometry model or the one provided is to complicated to be easily integrated into a SceneCompiler plugin. For these cases, DAIDALOS provides a set of classes to build geometries based on the approach of *constructive solid geometry* (CSG) [26].

Application programming interface

Programs are seldom static. Their source code changes over time either due to error corrections or due to replacement of deprecated functions by newer ones. Because of this it is necessary to develop with later changes in mind.

For this reason the DAIDALOS framework was split into several modules (see figure 4.6) which can be easily replaced by newer versions. In order to result in a meaningful modularization, two issues have to be considered. First, the classes from one module should, as far as possible, not rely on any class of other modules. This allows for an independent development of single modules and simultaneously limits the impact of possibly introduced errors to the particular module.

Second, the partition of DAIDALOS is hidden from the host application which makes use of it, by a so called *application programming interface (API)*. This is encapsulated within its own module and serves as a kind of additional abstraction level between the host application and DAIDALOS. Whenever the host application wants to use one of the methods provided by DAIDALOS, it calls the associated method of the DAIDALOS API. The underlying logic of the API module knows which class of the framework is responsible for the particular request as well as the module in which the class can be found. After execution of the chosen method, the API logic returns the associated return value to the application. While this approach seems rather complicated, it allows for nearly arbitrary changes on the implementation of the framework modules. For instance, consider moving a class from one framework module to another (e.g. from ValueCreation to Utilities). In the case, the host application or any plugin would directly access the modules this change would lead to broken dependencies[9], i.e. a non-working plugin. By hiding the internal structure of DAIDALOS behind the API, a change in the module structure can be integrated within the underlying logic of the API module. As long as the provided API methods are not changed (i.e. name, parameters and return value stay the same) all plugins will keep working without even noticing the replacement of modules.

[9] That is, the plugin code won't find the required class it depends on (as it has been moved).

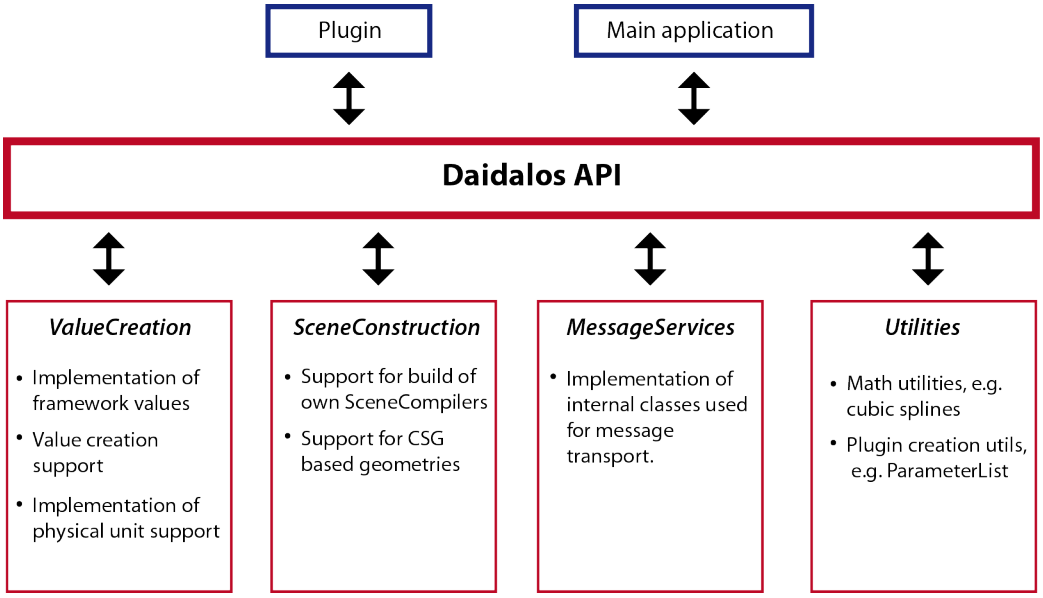


Figure 4.6.: The DAIDALOS framework is distributed as separated modules. External program code, like plugins or the host application, access the framework through an API layer, which hides the internal implementation.

4.4. The tracing loop

This section introduces the concept of a *tracing loop*. While the previous section presented the startup and runtime concepts, which specify the responsibilities of the involved components during the startup and runtime process, a similar concept can be used to specify the responsibilities of the available plugin types during the ray tracing process. This concept is referred to as the *tracing loop* and is shown in figure 4.7.

The specification of the tracing loop by DAIDALOS is necessary for two reasons. First, the tracing loop specifies the information which is available at a given moment during the ray tracing simulation. For example, the search for the next hit with an interface is always executed after the photon creation is done, therefore the information about the photon's propagation direction is available at the moment the hit search plugin is activated. While it is intuitively understandable that this is needed for a meaningful ray tracing simulation an explicit definition of this behavior allows the developers of plugins to rely on the existence of previously calculate information.

The second reason for having specified a tracing loop is to define the point during the ray tracing process when a plugin can influence the simulation. For example, as shown in figure 4.7, a SceneCompiler plugin can change the way by which the next interface hit is calculated, but only a FaceEffect, a RefractionCalculator or a BoundaryCondition plugin can change the way by which the simulated photon interacts with the hit interface. Consequently, the explicit definition of the tracing loop allows developers to choose the appropriate plugin type to influence the ray tracing process in specified manner.

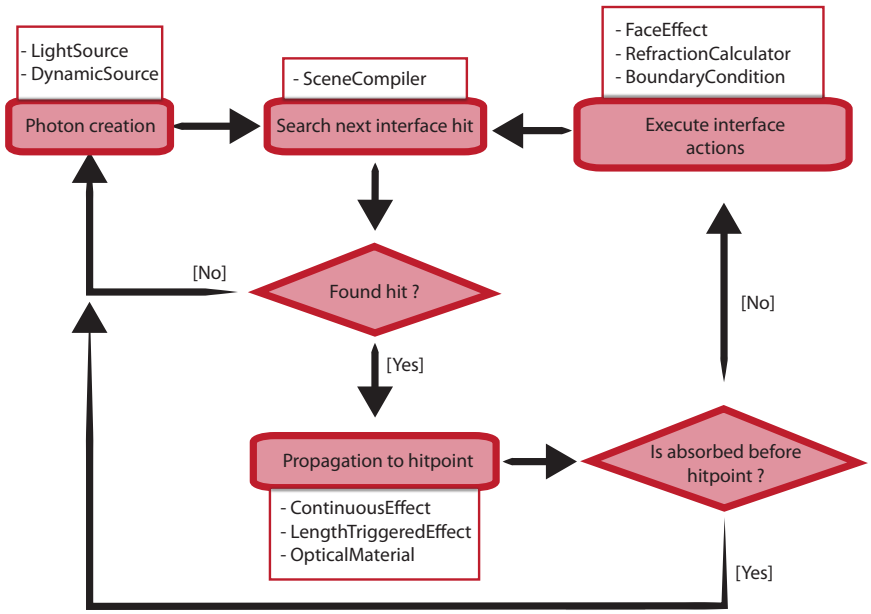


Figure 4.7.: An abstract view of the tracing loop. The actions which have to be executed during ray tracing (rectangles) can all be influenced by particular plugin services. While the tracer is not explicitly shown, it is responsible for actually executing the required actions and making decisions (diamond shapes) based upon their outcome.

Finally, there is one plugin type which is not involved in the tracing loop as shown in figure 4.7, namely the Tracer. As mentioned in the previous section, the plugin of this type is in control of the execution flow during the runtime part of the simulation (see figure 4.5). It is specified by the DAIDALOS framework that a plugin of this type is responsible to ensure that the execution of a ray tracing simulation is done as specified by the tracing loop. This is further discussed in section 4.6.1.

4.5. Plugins

Plugins are the base for every simulation that is made with DAIDALOS. They are part of the startup process, wherein they are loaded and configured by the host application and of the simulation process where they are working together to fulfill the simulation task. This section discusses the two representations of a plugin which result from this twofold appearance.

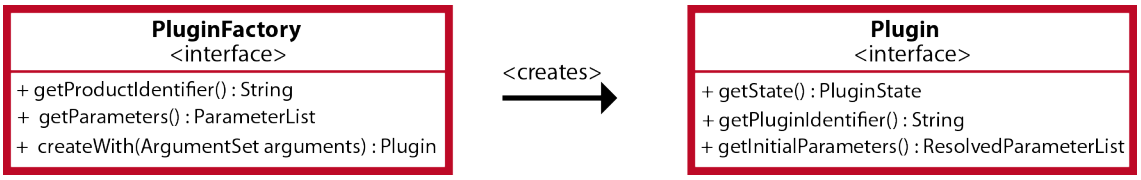


Figure 4.8.: Any plugin presents itself to the host application by means of the PluginFactory interface. By executing the createWith(ArgumentSet) method, the host application triggers the creation of a new Plugin instance.

4.5.1. Plugin bundles

Technically, the format of the binary storage chosen for plugins does not affect the actual simulation process. As the host application is the only component which actually gets in touch with the plugin storage (during the configuration stage, see figure 4.3), a new storage format could easily be developed for any host application

The DAIDALOS framework is developed with the idea of providing the tools to allow any person to develop plugins which than can be used to extend the ray tracing abilities of any host application that supports DAIDALOS. For this reason, DAIDALOS specifies a binary storage format for plugins, referred to as *plugin bundle*.

A plugin bundle is actually a modified OSGi bundle (see section 4.2.2), which allows for using the utilities of the OSGi service platform to load the plugin. The bundle file is extended by a plugin descriptor which is a text-file named *Descriptor.dpd* and is placed within the plugin bundle. It allows a host application to determine the provided features and needed configuration parameters of a plugin without having to load it into computer memory.

4.5.2. Plugin factory

A plugin bundle provides information about the associated plugin, as for example required configuration parameters. It is a pure passive object. The first time the actual plugin comes to life is when the host application transfers it into the OSGi plugin environment. After that, the plugin presents itself to the host application by providing an interface which is referred to as *plugin factory* (figure 4.8) and consists of only three methods.

The host application first executes the getParameters() method to retrieve the plugins configuration parameters[10]. The returned parameter set is designed as a read-only object, that is the host application can read parameters and their default value but cannot change them. In order to change parameter values the parameter set has to be converted into an ArgumentSet object, which is part of the DAIDALOS framework utilities. Finally,

[10]While this set of parameters, which is returned by the getParameters() method, should be equal to the one defined in the plugin descriptor file (see section 4.5.1), there is (at least at the moment) no mechanism which ensures this equality.

the host application executes the createWith(ArgumentSet) method to create the final already configured plugin instance.

The reason for introducing the intermediate plugin factory approach instead of having the host application creating the plugin instance by itself is due to the issue of decoupling (see section 2.2). A host application which should be able to create and configure a plugin needs a deep understanding of the implementation of the particular plugin. Therefore, it would be strongly coupled to the particular implementation of the plugin.[11] By introducing a plugin factory that looks the same for any plugin, a host application doesn't even have to know if it is a light source (section 4.6.2) or a face effect (section 4.6.5) which is created. It uses the methods to specify the configuration while the plugin factory takes care of the correct creation and initialization of the plugin instance.

4.5.3. Plugin service connectors

During the actual simulation process the plugin instance created by the plugin factory presents its capabilities by exposing a set of *service connectors*. For this, the plugin has to implement the Connectable interface (see figure 4.9) which allows for providing a set of ServiceConnector instances. Each connector stores the name of the connector (i.e. its identifier) and the type of capability (e.g. being a light source). In order to clarify the connectors direction, i.e. whether the capability is provided or required, the plugin developer doesn't implement the ServiceConnector interface directly, but one of its child interfaces, namely ServiceProvider and ServiceRequester, respectively.

While the fact of having a plugin requesting a service instead of providing it does not make sense at first, it allows for the development of plugins which act as a filter for the output of other plugins. For instance, consider the task of developing a plugin which introduces the effect of Lambertian reflection. This kind of reflection is used to describe an isotropically distribution of the directions of light reflected on rough surfaces [6]. However, the concept doesn't make any statements about the actual amount (i.e. the intensity) of light reflected by the surface. Instead, this amount is dictated by the particular interface, i.e. the material of the volume or the existence of an anti-reflection coating. While one approach would be to develop a sophisticated Lambertian reflection plugin which provides the ability to describe any useful surface refraction, the possibility for input service connectors allows for a much easier solution. According to this, the plugin is developed to delegate the calculation of refraction to a second plugin, e.g. a Fresnel based calculation of refraction (see section 1.2), which is accepted by an input service connector. When this plugin is done with calculation of probabilities and directions, the Lambertian reflection plugin changes the direction of the reflected light to comply with the desired distribution.

[11] This also implies that a host application has to know the implementation of any plugin it uses. Any change of a plugin (e.g. in a newer version) would force an adaption of the host application.

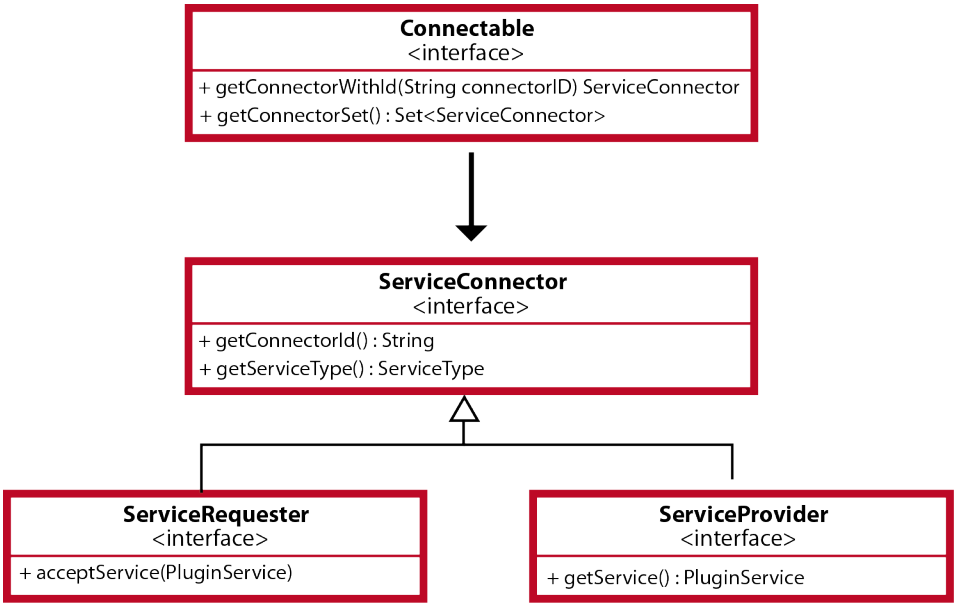


Figure 4.9.: Any plugin which implements the Connectable interface can provide a set of ServiceConnector instances. Each connector represents a capability of the plugin, e.g. a face effect (see section 4.6.5). By implementing either the ServiceRequester or the ServiceProvider interface, the developer decides whether the plugin provides the specified service or requests it.

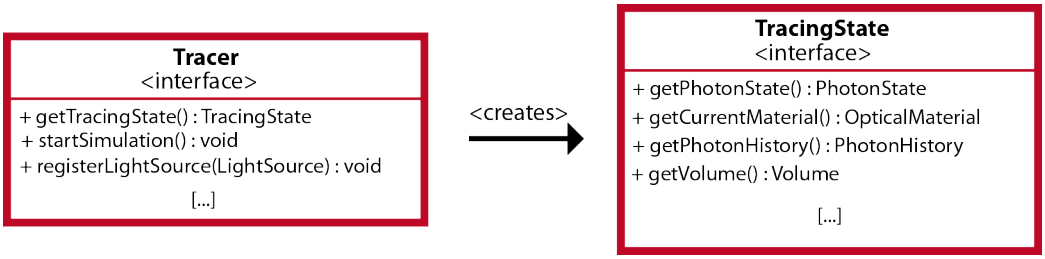


Figure 4.10.: A plugin which should control the tracing process has to implement the Tracer interface. The host application registers all other services with the tracer (here only the registerLightSource method is shown). Additionally, the tracer is responsible for the creation of a TracingState object which is accessible for any plugin within the simulation.

4.6. Available service connectors

This section serves as an overview of the available types of services connectors that can be exposed by a plugin to contribute to the simulation process.

The service connectors exposed by a plugin determine the location within the tracing loop when its effect is executed (see figure 4.7). Furthermore, as discussed later, the choice of the provided connectors determines which properties of a photon (e.g. position, direction, etc.) can be changed by a plugin. For example, while a boundary effect (see section 4.6.7) can change the position of the simulated photon, a refraction calculator (see section 4.6.6) cannot.

4.6.1. The tracer

The tracing loop in figure 4.7 shows the tasks which have to be done during ray tracing. However, this representation of the simulation process omits the point that there has to be program code that controls the execution of this loop. This means, a component which is responsible for deciding when a step is done or which step is taken next, respectively. This issue of flow-control is the responsibility of the Tracer service.

The moment the simulation process is started the control of execution is transfered to a plugin, which is referred to as the tracer. This plugin is required to provide a Tracer service connector (see figure 4.10). The host application registers all used plugins as well as their connections to the elements of the geometry[12] with the tracer (see section 4.3.1). During the simulation the tracer uses the registered plugins (e.g. light sources) to retrieve information (e.g. position, wavelength and propagation direction of a new photon) about the current state of the simulation. Based upon those information the tracer is responsible to decide about the next step within the tracing loop, e.g. whether a photon gets reflected or transmitted at an interface.

Aside from its controlling function, the tracer is responsible for creating and providing

[12]For example, a face effect is always bound to at least one face of the geometry.

an implementation of the TracingState interface (see figure 4.10). This object is distributed at the beginning of each simulation to all involved plugins. The TracingState gives access to the current state of a simulation by providing information about the currently simulated photon, the volume it resides in and a list of events which has led to this state, referred to as PhotonHistory.

The photon history

The existence of a photon is merely a sequence of events, like creation, hitting an interface, getting reflected, and so on. There are several cases in which the action of a plugin might depend on an event which happened in the past of the photon's life. For example, to calculate the distance a photon has propagated within a volume a plugin may ask for the transmission event by which the volume was entered.

For such cases, the TracingState object provides the method getPhotonHistory() which returns a PhotonHistory object. The photon history is a sequential list of all events which led to the actual state of the currently simulated photon:

- **Refraction events**
  These events are associated with each refraction process and can be distinguished with respect to their outcome as transmission and reflection, respectively. Additionally, any refraction event contains information about the interface which led to refraction and the refraction result (see section 4.19).

- **Leaving and entering events**
  These events are generated whenever a photon enters or leaves a volume. They provide information about the involved volumes.

- **Creation-, absorption- and lost-events**
  Events of this kind are generated, when a photon is created by a source or gets lost either by leaving the simulation domain or by getting absorbed by a medium.

4.6.2. Light sources

Light sources are the starting point of any optical simulation. Their is a great variety of different light sources requiring different types of support by the underlying ray tracing framework. For example, light sources as a lamp or the sun emit a steady flux of photons. Contrary to that, the emission of light sources based on photoluminescence depend on the earlier absorption of a photon and therefore may eventually stop emission and resume after the next absorption. With regard to Daidalos a plugin which wants to act as a light source has to provide the LightSource interface as shown in figure 4.11.

When the tracer decides that a new photon is created, it has to choose a light source that should be used for creation. In the case where more than one source is available, the tracer first retrieves the photon flux of any source by calling the getPhotonFlux() method of the LightSource interface. The returned value describes the number of photons per second which are emitted into the full solid angle. The probability by which a

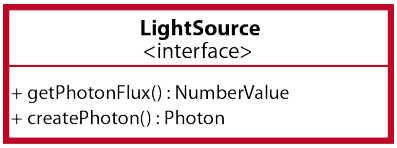


Figure 4.11.: Any plugin exposing the LightSource interface can be used as a light source.

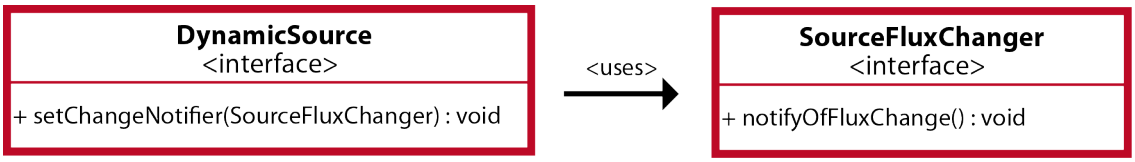


Figure 4.12.: A light source which implements the DynamicSource interface signals that it wants to change its photon flux during simulation. For this, the tracer transfers a SourceFluxChanger object to the source. Any time the light source changes its photon flux, it finalizes the process with executing the notifyOnFluxChange() method of the SourceFluxChanger. This triggers the tracer to re-evaluate the available sources and their fluxes.

particular source creates a photon is then equal to the ratio between its photon flux and the cumulated photon flux of all available sources. After the source is chosen, the tracer executes the createPhoton() method, which returns an initialized photon.

Dynamic sources

Dynamic light sources, as for example luminescence based sources, can stop and resume their emission based upon a set of conditions (e.g. a previously absorbed photon). The associated change in their photon flux value[13] has to be communicated to the tracer. For this, dynamic sources have to implement the DynamicSource interface shown in figure 4.12. When the tracer recognizes this interface it transfers a SourceFluxChanger instance to the source by executing the setChangeNotifier method. By using the methods provided by the SourceFluxChanger instance, a source can communicate changes in its photon flux at any time during the simulation process.

Expansion stages of simulated photons

Simulated photons as created by a light source come in different stages of expansion, which are shown in figure 4.13. The most basic one is the Photon which is initially created by all light sources. At this stage, the photon consists of only three values, namely its wavelength, its current position and the direction of propagation.

[13] For instance, a luminescent source may change its photon flux between zero, when no photon was previously absorbed, and a fixed value for normal emission.

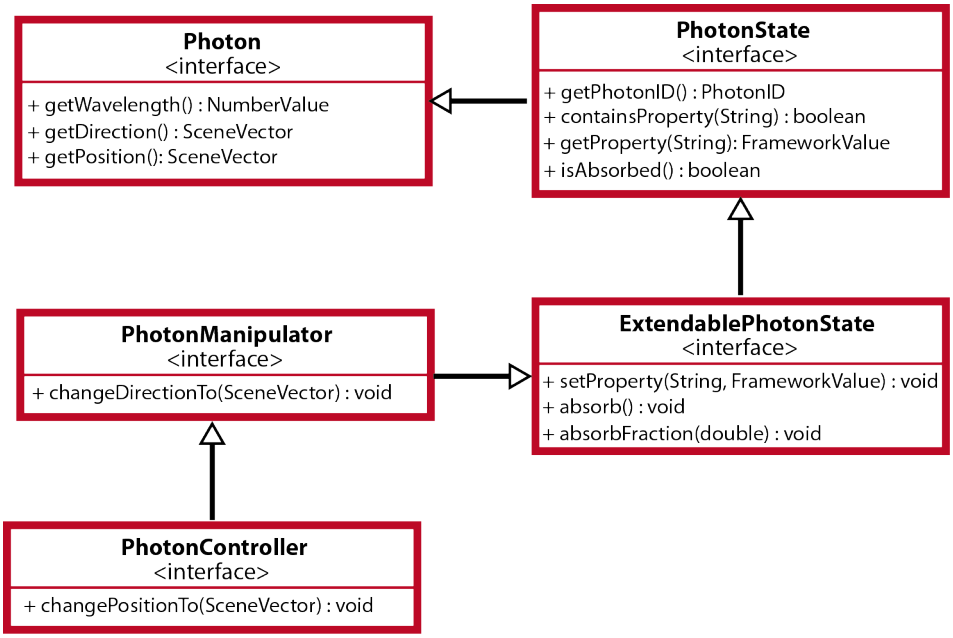


Figure 4.13.: The different expansion stages of simulated photons. Note, each stage extends its successor interface and therefore provides the methods of its parent additionally to the newly introduced ones.

As soon as the tracer gets in touch with the created photon it wraps an implementation of PhotonState around it, which allows for retrieving additional information (e.g. the internal photon identifier). This photon expansion stage is then made available to all involved plugins through the TracingState (see section 4.6.1). The most important feature of a PhotonState object is the ability to request properties previously attached to the photon. These can be set by any plugin that gets access to the ExtendablePhotonState expansion stage.[14] Any property consists of an identification String, which serves as a key by which the associated value can be addressed, and the actual value, which can be any kind of FrameworkValue.

Some plugins, like refraction calculators (see section 4.6.6) or boundary effects (see section 4.6.7) do need the possibility to influence either the direction of propagation of the photon or its current position within the simulation. To support these cases, two additional interfaces exist, namely the PhotonManipulator and the PhotonController.

4.6.3. The scene compiler

Ray tracing simulations based upon Daidalos consist of a geometry, which specifies object shapes and surfaces, as well as plugin-driven effects, which are applied to these objects. There is a broad variety of sources where an actual geometry can come from. This includes different file formats, for example the *standard tesselation language* (STL),

[14] This stage is the common stage transmitted to any involved plugin at its turn within the tracing loop.

Figure 4.14.: The SceneCompiler and its related interfaces are responsible for loading and representing the Scene during the simulation.

as well as different types of geometry descriptions (e.g. representations by polygons or constructive solid geometry).

In order to allow for nearly arbitrary sources of geometry, DAIDALOS breaks the process of providing a geometry into two independent parts. First, a geometry has to be retrieved, that means it has to be loaded from some kind of storage, e.g. a file on disk. Second, the geometry has to be presented within the simulation in a manner that allows for calculating the next hit interfaces for a particular photon. With regard to DAIDALOS, the first part is done by the SceneCompiler and the second by the Scene (see figure 4.14). To visualize the concept, consider a host application that already has a geometry representation by means of STL files, which contain a polygonal description. The developer who wishes to integrate the geometry with DAIDALOS will first develop an implementation of the SceneBlueprint interface. This interface is merely for handling by DAIDALOS, e.g. simplifying the task of informing the user about the loaded scene. Additionally, the developer will implement a plugin which exposes a SceneCompiler service connector to accept the previously implemented blueprint and to create an implementation of the Scene interface when requested. This Scene object represents the loaded geometry during simulation and allows the tracer to execute a search for the next hit interface. As the developer of the SceneCompiler knows the geometry as well as its representation, he can implement the methods used for searching the next interface hit of a photon. The tracer does not have to know the details of the geometry at all. It executes the initiateHitSearch() method of the created Scene object and gets the found hits as result.

At some point of the simulation, e.g. when the photon has been created or its position has been changed by a boundary effect (see section 4.6.7), the tracer needs to determine the current volume in which the photon resides. In such cases, the tracer executes the getEncapsulatingVolumeAt(SceneVector) with the current position of the photon.

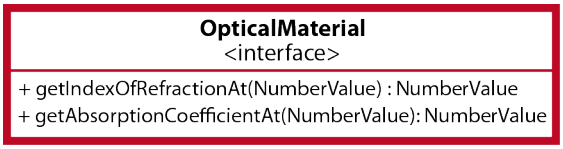


Figure 4.15.: An OpticalMaterial service connector represents a material within the simulation. The getAbsorptionCoefficientAt(NumberValue) returns the real-valued absorption coefficient while the getIndexOfRefractionAt(NumberValue) method retrieves the complex index of refraction at a given wavelength.

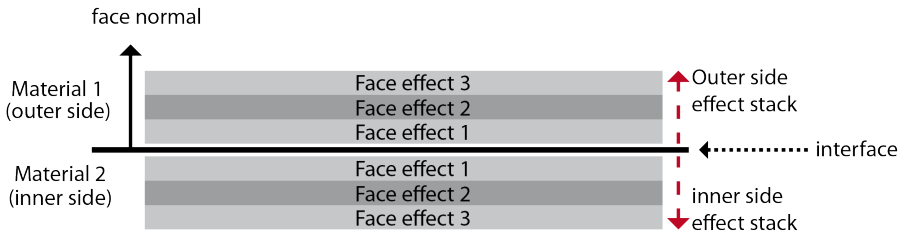


Figure 4.16.: An interface which separates two volumes consists of two sides. The outer side is determined by the direction of the face normal. Both sides expose a separated face effect stack to which face effects can be applied.

4.6.4. Optical materials

In order to allow for the association between volumes (as defined by the geometry) and specific materials, DAIDALOS provides the OpticalMaterial service connector. It consists of two methods, namely the getAbsorptionCoefficientAt(NumberValue), which returns the real-valued absorption coefficient, and the getIndexOfRefractionAt(NumberValue) method, which retrieves the complex index of refraction at a given wavelength.

The tracer is responsible for exposing the current volume (i.e. the volume in which the currently simulated photon is propagating) as well as the associated optical material through its TracingState object (see section 4.6.1). Therefore, any plugin involved with the simulation can access and use the provided values (e.g. for calculating the volume absorption by a volume effect (see section 4.6.8)).

4.6.5. Face effects

Within any meaningful simulation some photons will eventually hit an interface between two volumes of the underlying geometry. Such faces are often used to apply photon detectors, like counters for transmitter or reflected photons. With regard to DAIDALOS any face is associated with a single volume as defined by the geometry. The face's normal vector is specified to be always directed towards the outer side of the volume and is used to distinguish both sides of the face (see figure 4.16).

A plugin that should be able to perform actions on the event of photons hitting a particular face, has to expose the FaceEffect service connector, which is shown in figure

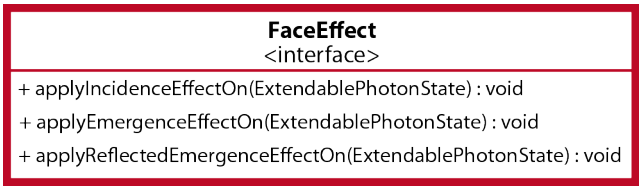


Figure 4.17.: A FaceEffect service connector represents any effect which should be associated with a face of the underlying geometry. During the simulation the tracer calls the method corresponding to the current type of photon-interface contact (see figure 4.18).

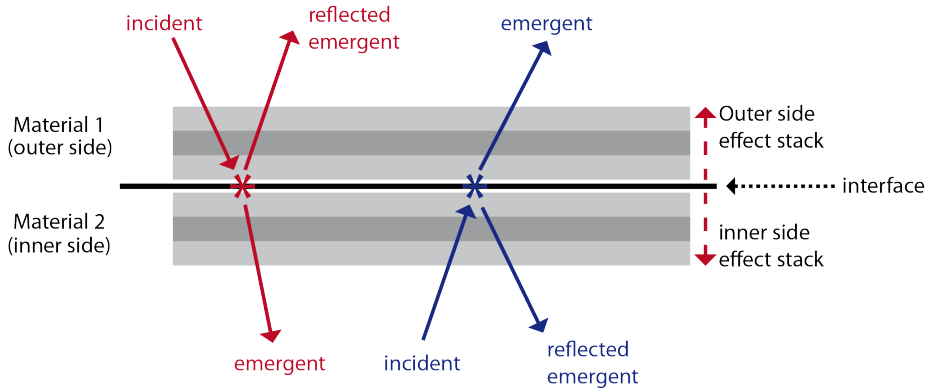


Figure 4.18.: Naming conventions with regard to photons which are propagating to and from a face, respectively. The different names which are associated with different types of a photon-interface contact (i.e. incident, interface hit, reflected or transmitted, respectively) allows for a differentiated reaction of the applied face effects.

4.17. During the configuration of the simulation, the user decides whether a face effect is applied to the inner or outer side face effect stack. Both stacks are growing away from the actual interface, so later applied effects will sit on top of previously applied ones. When the tracer recognizes a photon is hitting a particular face, it is responsible for notifying the associated face effects in the proper order (see figure 4.18). That is, an incident photon will first hit the most outwards effect, triggering the execution of its applyIncidenceEffectOn(ExtendablePhotonState) method. In the case of multiple effects they become active in a decreasing order before the photon actually hits the interface. After the tracer has evaluated the outcome of the refraction process (i.e. the photon gets either reflected or transmitted) either the inner side or the outer side face effect stack is traversed in an increasing order. This propagation than triggers the applyEmergenceEffectOn(ExtendablePhotonState) in the case of transmission or the applyReflectedEmergenceEffectOn(ExtendablePhotonState) for a reflected photon.

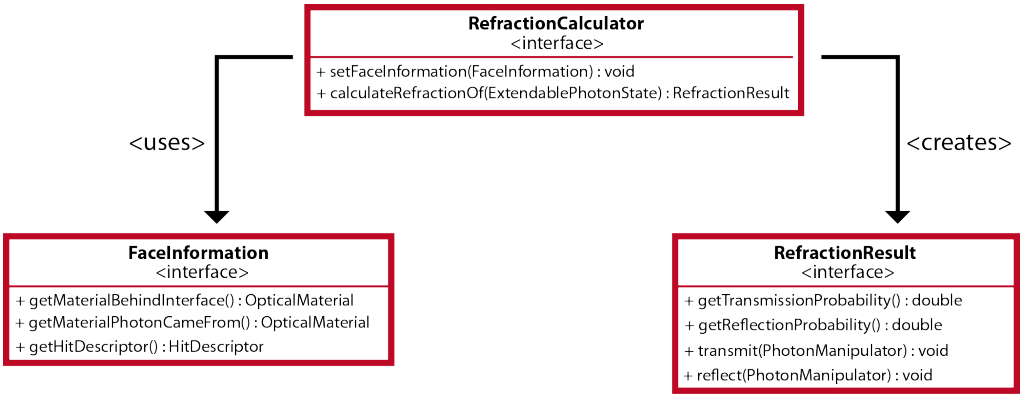


Figure 4.19.: A plugin which is used as refraction calculator has to implement the RefractionCalculator interface. While the tracer specifies the conditions at the hit point by providing an implementation of the FaceInformation interface, the refraction calculator returns an implementation of the RefractionResult interface to provide the probabilities for transmission and reflection.

4.6.6. Refraction calculators

As shown in the last section, face effects are not directly bound to an interface, but are applied to a stack which is located on both sides of the interface (see figure 4.16). The ability to influence the behavior of the actual interface is reserved for two services, namely the RefractionCalculator discussed in this section and the BoundaryEffect which is subject of the following section.

Whenever a photon hits an interface between two materials (i.e. after the face effect stack on the incident side has been traversed) the tracer needs to calculate the properties for transmission and reflection. In most cases this calculation can be done based on the Fresnel equations (see section 1.3). However, some effects like anti-reflection coatings [27] or Lambertian reflection [6] can change the calculated probabilities or the direction of propagation for reflected or transmitted photons. To allow for a broad range of different, exchangeable effects Daidalos associates any face with either a refraction calculator or a boundary effect.

Any plugin that should be used to act as a refraction calculator has to implement the RefractionCalculator interface as shown in figure 4.19. Anytime a photon hits an interface, the tracer is responsible to create a FaceInformation instance, which includes information about the face normal of the interface as well as the materials on both sides. This information is provided to the refraction calculator of the particular interface by calling its setFaceInformation(FaceInformation) method. Afterwards, the tracer calls the calculateRefractionOf(ExtenablePhotonState) method, which is answered by the refraction calculator by returning an implementation of RefractionResult. The refraction result has a twofold function. First, it includes the probabilities for transmission or reflection of the photon, allowing the tracer to decide which of them takes place. Second, after the

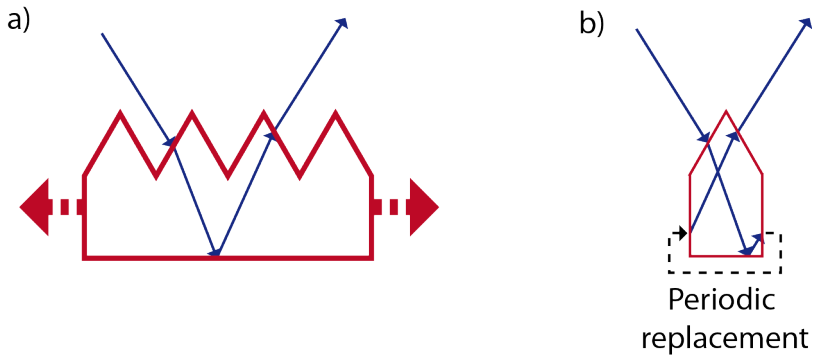


Figure 4.20.: Periodic geometries (a) can often be represented by a repetition of a basic unit cell. By equipping the side boundaries of the unit cell with periodic boundary conditions, the simulation can be constrained to the unit cell (b) while leading to the same simulation result.

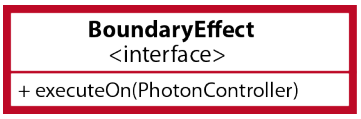


Figure 4.21.: A plugin which should act as boundary effect has to provide the BoundaryEffect service connector.

tracer has made its decision it calls either the transmit or the reflect method, allowing the refraction calculator to choose the direction of the resulting photon. Following this way, the tracer maintains its power to decide what happens within the simulation while the plugin can decide about how it is done.

4.6.7. Boundary effects

Sometimes only changing the direction of a photon is not enough. To visualize, consider the case where a geometry can actually be represented by the repetition of some base element, referred to as *unit cell* (see figure 4.20). In this case, the simulation can be greatly simplified by constraining the path of a photon to this unit cell, applying periodic boundary conditions to the side boundary of the unit cell.

As can be seen from figure 4.19 the reflect and transmit methods of the refraction result are only getting a PhotonManipulator interface (see figure 4.13) and therefore are incapable of changing the photons position (i.e. positioning the photon). For such cases, DAIDALOS specifies the BoundaryEffect interface (see figure 4.21). As discussed in chapter 7, aside from the mentioned unit cell approach, boundary effects can be used to increase the performance in multi-scale simulations[15] by connecting spatially separated simulation domains of different scale.

[15]This means, simulations where the geometry includes different magnitudes of scale (e.g. ranging from m to µm).

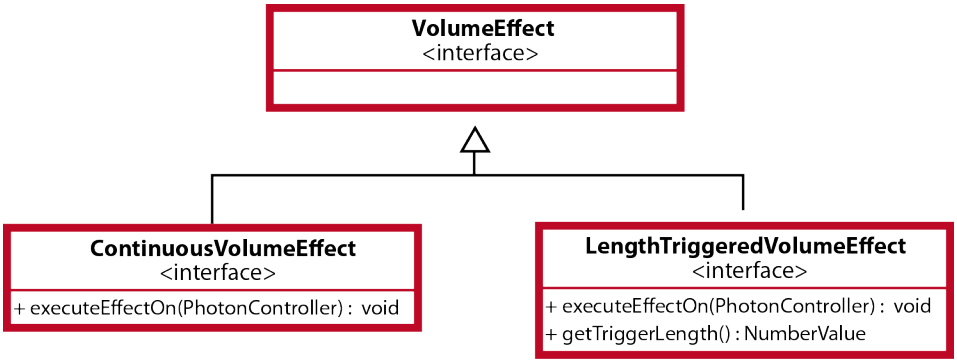


Figure 4.22.: A plugin which should act as boundary effect has to provide the BoundaryEffect service connector.

4.6.8. Volume effects

While previously described services are associated with the faces of a geometric object, DAIDALOS provides two kinds of volume effects, namely the LengthTrigerredEffect and the ContinuousVolumeEffect. Both are used to represent volumetric effects (e.g. absorption of light within volumes), but they differ with respect to the location within the tracing loop where they are executed.

Length-triggered volume effects

A volume effect which is based upon a LengthTriggeredEffect service connector (see figure 4.22) gets executed after a predefined trigger-length. For this, the associated interface specifies the getTriggerLength() method. With respect to the tracing loop (see figure 4.7), the execution of a length-triggered effect takes place between the search for the next hit-point and the actual propagation to it. The tracer compares the trigger length of any length-triggered effect to the distance calculated for the next interface hit. If the trigger-length is lower than the distance to the hit-point, the photon is only moved by the trigger-length and the effect is executed. Otherwise, the photon is moved to the hit-point and the propagated distance is subtracted from the current value of the trigger-length. After any execution of a length-triggered effect, the tracer executes the getTriggerLength() method to ask for a new trigger-length value. Volume effects based upon the LengthTriggeredEffect interface are usable for physical effects which have to be executed at a given location within a volume. For instance, consider the application of an index-gradient lens. Such an optical device impacts the direction of a ray of light through a successive change in its index of refraction. By using a LengthTriggeredEffect the direction of the photon can be changed during its propagation through the volume and allows for an effective simulation of the desired behavior (see figure 4.23(a)).

The main drawback of a length-triggered effect is the interruption of the tracing loop and the resulting increase in simulation time. For example, in order for a precise simulation of an index-gradient lens the trigger-length of the volume effect has to be

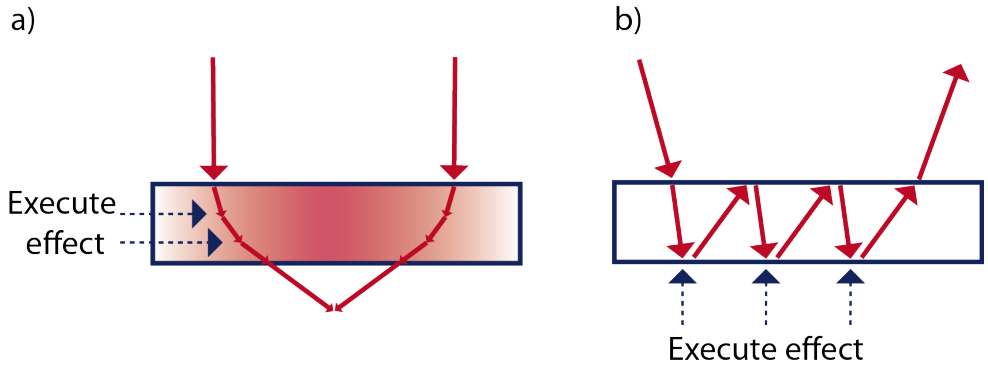


Figure 4.23.: A LengthTriggeredEffect is executed at a location within a volume. It can be used for effects which must be activated within a volume, e.g. the change of propagation direction for an index-gradient lens (a). Contrary, a ContinuousVolumeEffect is executed whenever the photon hits one of the surrounding faces (b). This increases the simulation performance (see text) and should be used for all volume effects which are necessarily executed at a specific location, e.g. general volume absorption.

small enough to follow the underlying gradient of the refractive index. Consequently, the more steep this gradient gets, the smaller the trigger-length has to be. This leads to a significantly increased time for the propagation within the volume and, considering multiple internal reflections, can be the time-dominating part of the simulation.

Continuous volume effects

A volume effect which is based upon the ContinuousVolumeEffect interface (see figure 4.22) gets executed whenever the photon hits a face of the underlying geometry (see figure 4.23). Additionally, if any effect requests execution while propagating through the volume (e.g. a length-triggered effect) the continuous volume effect gets executed before this effect. Due to this behavior, continuous volume effects can be used for any effect which is not directly related to a specific position within a volume. For instance, consider the physical effect of volume absorption as described by the Lambert-Beer law (see section 1.2). This effect doesn't need to be executed at a specific location, but latest before leaving the volume. By using a ContinuousVolumeEffect instead of a length-triggered one, the overall simulation time can be greatly reduced.

5 Wafer optics

In this chapter the ray tracing framework is tested by simulating a simple as possible geometry, the reflectivity of silicon wafers, and by comparing the results to another, established ray tracer, to analytical expressions where possible, and to measurements.

During these tests, it became apparent that the reflectivity measurements are affected by the geometrical configuration of the measurement setup, especially in the wavelength range of weak absorption. Therefore, this chapter starts with a short introduction on reflectivity measurements, done with the Cary UV-VIS-NVIS spectrometer. This is followed by ray tracing simulations of a planar and of a textured silicon wafer. For validation, the gathered simulation results are compared to analytical calculations and reference simulations performed by using the well-known SUNRAYS ray tracing application [3, 28–30]. Additionally, a significant problem associated with reflectivity measurements on wafers with good light-trapping properties is discussed.

In the final section DAIDALOS is used for a detailed simulation of the reflectivity of two textured silicon wafers based upon texture structures measured by laser scanning microscopy.

5.1. Performing reflectivity measurements with the Cary UV-VIS-NVIS spectrometer

This section provides an overview of the reflectivity measurement process with the Cary UV-VIS-NVIS spectrometer which was used to perform the measurements presented in the following two sections. This description is limited to the fundamental measurement procedure as needed to understand the shown reflectivity curves.

With the Cary spectrometer, the reflectivity of a sample is measured using a component referred to as *integrating sphere* or *Ulbricht sphere* as shown in figure 5.1(a). This is a hollow sphere where the inner walls provide ab almost perfect diffuse reflection. Additionally, there are four openings, referred to as *ports*. Two of them are used as inputs, allowing the *reference beam* and the *sample beam* to enter the sphere. Both beams are

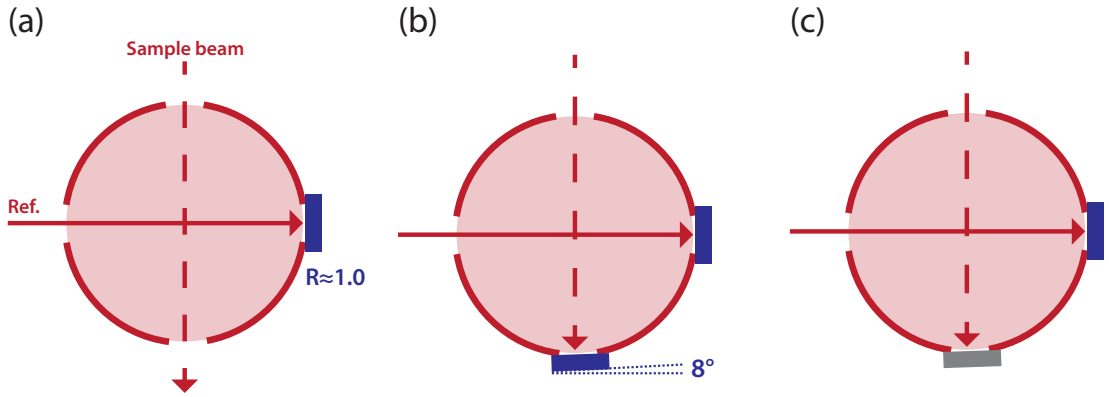
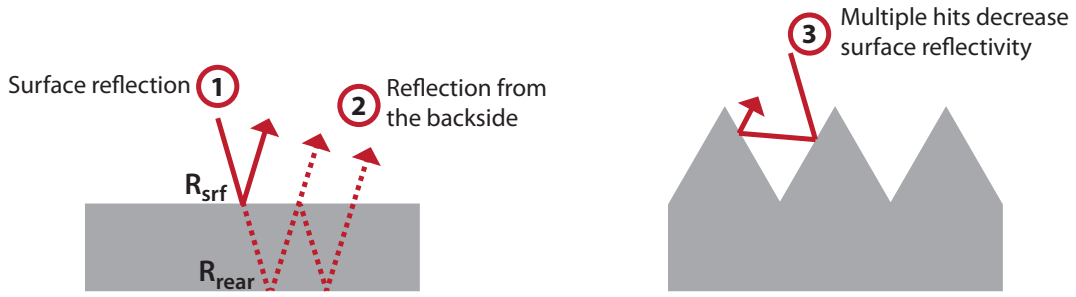


Figure 5.1.: A top view on the integrating sphere. Each reflection measurement with the Cary spectrometer consists of three steps. The spectrometer is calibrated by performing a dark-measurement ($R_{0\%}$) with the sample-port open (a), followed by a light-measurement ($R_{100\%}$) using a reflection standard (b). Afterwards, the actual measurement is executed with the sample (gray rectangle) positioned at the sample-port (c).

generated by the same source and are illuminating the *reference-port* and the *sample-port*, respectively. For the presented measurements, the sample-beam is configured to produce a light-spot with dimensions of 2 mm × 3 mm on the measured sample. The sample is attached to the sample-port with a slope of 8° to avoid that specular reflected light leaves the sphere through the sample-beam input port. Finally, a detector (not shown in the figure) is embedded into the bottom of the integrating sphere to measure the amount of light which is diffusively reflected on its surface.

Each reflection measurement consists of three individual measurements each of them performed for every wavelength λ in the considered wavelength range (usually from 300 nm to 1400 nm). First, the instrument is calibrated by measuring the extreme cases of no-reflection ($R_{0\%}(\lambda)$) and full-reflection ($R_{100\%}(\lambda)$). The first value is received by leaving the sample-port either open or by placing a light-trap at the sample-port (see figure 5.1(a)).[1] The latter value (see figure 5.1(b)) is measured by placing a reflection standard with known reflectivity $R_{\mathrm{std}}(\lambda)$ at the sample-port. It is chosen to best mimic the angular distribution of the reflected light as expected for the actual sample. This means, using a specular reflecting standard for measurements of polished wafers or using a diffuse standard for textured ones. Finally, the measurement of the sample's reflectivity $R_{\mathrm{smp.}}(\lambda)$ is executed as shown in figure 5.1(c), by placing the sample at the sample-port. Based upon the performed measurements the adjusted value of the sample's reflectivity

[1] Most light-traps seem to reflect at least a little amount of the incoming light which leads to an incorrect calibration. This can be avoided by leaving the port open which obviously only makes sense if the environment can be sufficiently shaded.



(a) Paths for reflection

(b) Surface reflection on pyramidal textured wafer

Figure 5.2.: There are two paths by which light can be reflected from a wafer (a). It can either be reflected at the wafer's front surface (1) (reflectivity R_{srf}) or at its rear side (2) (R_{rear}) leaving the wafer through the front (possibly, after some alternating internal reflections). Considering pyramidal surface textures (b), the surface reflectivity is reduced by increasing the average number of hits (3) a ray undergoes before finally being reflected.

$R_{\mathrm{adj.,smp.}}(\lambda)$ is calculated by:

$$R_{\mathrm{adj.,smp.}}(\lambda) = \frac{R_{\mathrm{smp.}}(\lambda) - R_{0\%}(\lambda)}{R_{100\%} - R_{0\%}(\lambda)} \cdot R_{\mathrm{std}}(\lambda). \tag{5.1}$$

5.2. Reflectivity of a planar wafer

The simplest geometry is that of a planar silicon wafer. Considering its optical characteristics two wavelength ranges can be distinguished: where the sample is opaque and where it is transparent. For incident light with wavelengths smaller than about 900 nm all light transmitted into the silicon bulk is absorbed before reaching the wafer's rear surface. Consequently, the reflectivity within this wavelength range is determined only by the reflectivity of the wafer's surface (see figure 5.2(a)).

Knowing the complex index of refraction for silicon, the surface reflectivity can be calculated with the Fresnel theory (see section 1.3). The wafer's surface reflectivity $R_{\mathrm{srf}}(\lambda)$ at a wavelength λ is then given by:

$$R_{\mathrm{srf}}(\lambda) = \alpha \cdot \left| \frac{\widetilde{n}_{\mathrm{air}} \cos(\Theta_i) - \widetilde{n}_{\mathrm{Si}} \cos(\Theta_t)}{\widetilde{n}_{\mathrm{air}} \cos(\Theta_i) + \widetilde{n}_{\mathrm{Si}} \cos(\Theta_t)} \right|^2 + (1-\alpha) \left| \frac{\widetilde{n}_{\mathrm{Si}} \cos(\Theta_i) - \widetilde{n}_{\mathrm{air}} \cos(\Theta_t)}{\widetilde{n}_{\mathrm{Si}} \cos(\Theta_i) + \widetilde{n}_{\mathrm{air}} \cos(\Theta_t)} \right|^2. \tag{5.2}$$

Here, α is the fraction of the light which is perpendicular polarized, Θ_i and Θ_t are the incidence and transmission angle,respectively, and $\widetilde{n}_{\mathrm{air}}$ and $\widetilde{n}_{\mathrm{Si}}$ are the wavelength dependent complex indices of refraction for air and silicon, respectively.

The resulting curve for the parameters $\alpha = 0.5$, $\widetilde{n}_{\mathrm{air}} = 1.0$ and $\Theta_i = 8°$ is shown as black line in figure 5.3 (left). Additionally, two ray tracing simulations were conducted

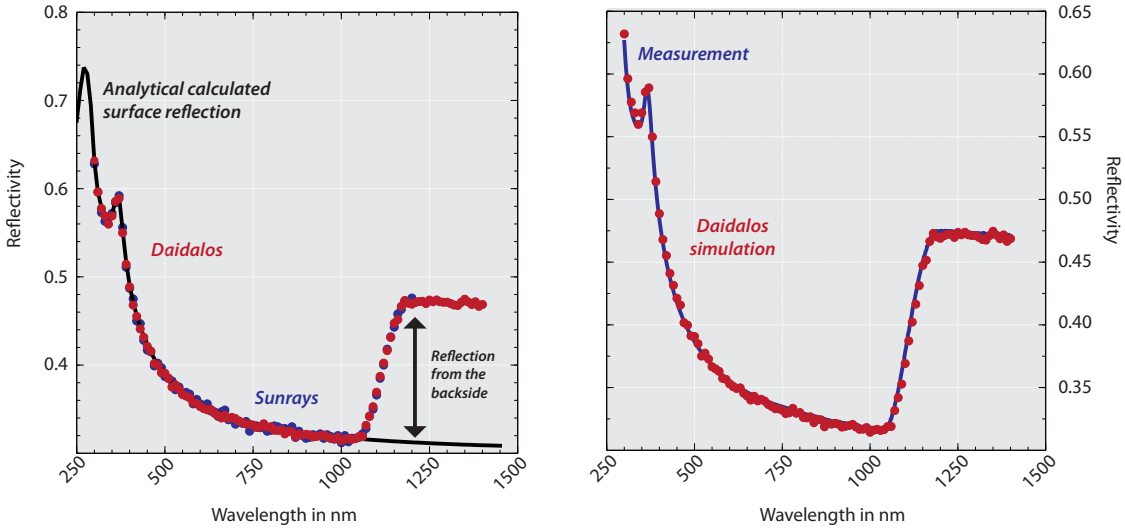


Figure 5.3.: The surface reflectivity of a planar wafer can be calculated analytically with Fresnel theory (black line). Additional simulations of a silicon wafer with a thickness of 1300 µm were executed with DAIDALOS as well as with the well-known ray tracer SUNRAYS (left). Both are in very good agreement with each other as well as with the measurement (right).

for a planar wafer with a thickness $d = 1300\,\mu\text{m}$. One of them with the established ray tracing software SUNRAYS [3], the other using the DAIDALOS framework. Both results are in very good agreement with each other as well as with the Fresnel theory (as long as the wafer is opaque).

Up from a specific threshold wavelength, which depends on wafer material and thickness, some rays reach the wafer's backside. The ones that are reflected towards the front and are leaving the wafer contribute to the measured reflectivity, see figure 5.2(2). This behavior is shown in figure 5.3 (left) and confirmed by the measurement shown in figure 5.3.

5.3. Reflectivity of a pyramidal textured wafer

Rather than being planar, common solar cells provide a pyramidal textured surface as shown in figure 5.2 (b). The investigation of the optical properties of the used texture geometry is a common task that is executed using ray tracing software [29, 31, 32]. The texture increases the average amount of surface hits to which an incident ray of light is exposed. Accordingly, this leads to an increased probability for light being finally transmitted into the solar cell. Considering the optical characteristics, similar wavelength ranges as for planar wafers can be applied. However, due to the multiple interface hits for incident as well as for internally reflected light, an analytical evaluation is rather

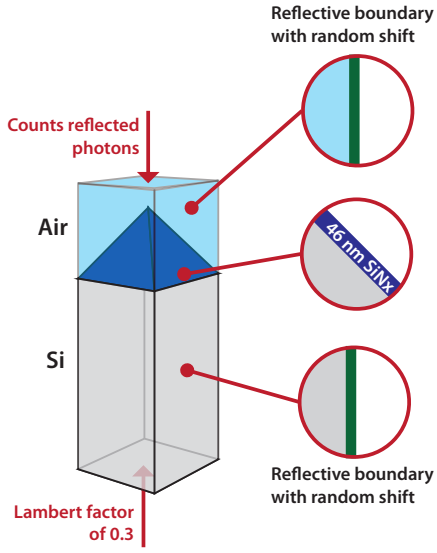


Figure 5.4.: The unit-cell simulation domain used to simulate a pyramidal textured wafer, consists of a single pyramid and the associated substrate. The pyramid is surrounded by a cubic volume of air which counts the reflected photons on its top. Additionally, the pyramid's surface is equipped with a 46 nm silicon nitride and a reflective boundary is applied to the side-boundaries of the domain. The used boundary effect also applies a lateral random shift to the photon which allows for a simulation of random upright pyramids. In order to account for a minor roughness, the substrates back has a Lambertian factor of 0.3.

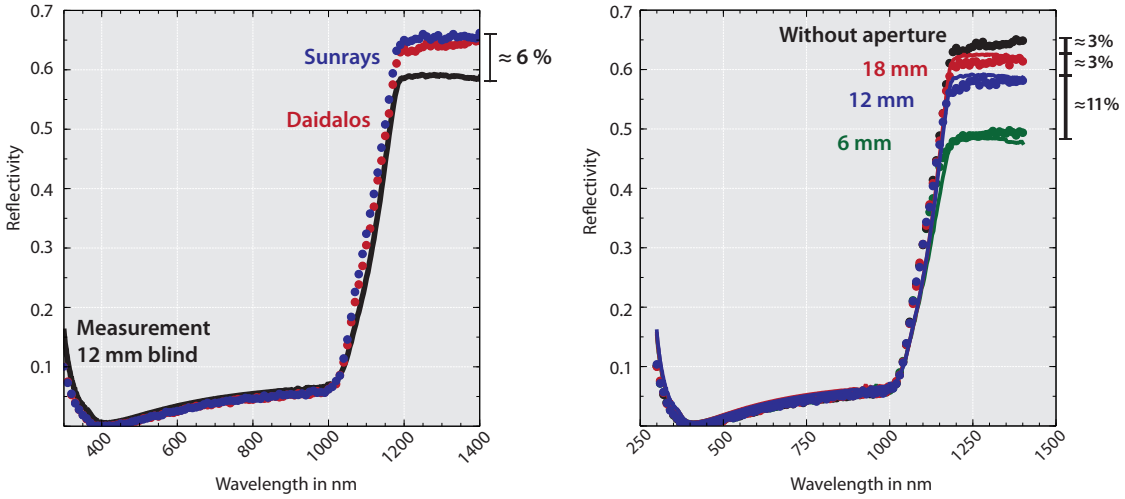
complex. Therefore, for such geometries an optical simulation is the most adequate way to gather insights of the optical characteristics.

In order to validate the results of the ray tracing simulation it is compared to the measurement of a 247 µm thick wafer. It provides a random pyramid texture on its front and a polished rear side. Additionally, a 46 nm silicon nitride (SiNx) coating was applied to its textured front while having no coating on its rear side. The wafer's reflectivity was measured using three different apertures with diameters of 6 mm, 12 mm and 18 mm. They are placed between the wafer's surface and the sample-port of the integrating sphere (see figure 5.1).

For the simulation of this geometry a unit-cell approach was chosen as shown in figure 5.4. The simulation domain consists of a single upright pyramid, at the top surrounded by a cubic volume filled with air and on the bottom continued by the associated substrate volume. The cubic air volume ends 15 µm above the pyramids top to account for a small gap of air between sample and aperture. A FaceEffect (see section 4.6.5) is applied to the outer side of the top of the cubic air volume to count the reflected photons. Within the DAIDALOS simulation, a RefractionCalculator (see section 4.6.6) is used to represent the silicon nitride coating. Additionally, a BoundaryEffect (see section 4.6.7) is applied to the surrounding sides of the substrate and the cubic air volume. This effect acts as a reflective boundary with an additional random shift along the sides of the boundary to allow for a simulation of a random upright pyramid texture. That means, a simulated photon which hits the boundary of the simulation domain is not just reflected, but is shifted by a random amount along the sides of the boundary. Finally, a Lambertian factor of 0.3 is chosen for the substrates rear side to account for a minor roughness. The associated SUNRAYS simulation was configured accordingly.

The results of these simulations are shown in figure 5.5 (a) along with the corresponding measurement using an aperture of 12 mm. As can be seen, while the simulation results produced by SUNRAYS and DAIDALOS are in sufficient agreement, there is a deviation of about 6 % at that plateau for wavelengths in the range of 1200 nm to 1400 nm between simulations and measurement. As can be shown by additional measurements and simulations (see figure 5.5 (b)) this deviation is the result of the good light-trapping ability of a textured wafer in conjunction with the diameter of the apertures used.

To clarify this, during a reflection measurement as described in section 5.1 all light that is not absorbed by the wafer's substrate has six ways to either get transmitted or reflected (see figure 5.6). With respect to a wafer's reflectivity, the incident light gets reflected back into the integrating sphere either from the wafer's surface (1) or after one (3) or several (4) internal reflections. However, note that internal reflections lead to a lateral propagation of the trapped ray within the wafer. While this lateral propagated distance is small for wafers with low light-trapping (e.g. planar wafers) and lower wavelengths which are well absorbed, it can get significant for long wavelengths and good light-trapping as provided by a textured wafer. Depending upon the distance d_{smp} between the sample and the port and the height h_{smp} the lateral propagation of rays which incident near the ports edge can lead to two types of measurement artifacts. First, rays which leave the sample near the ports edge but are still within the port region may propagate through the air slit between port and wafer, finally hitting the wall of the integrating sphere (5).



(a) Simulation of a wafer with pyramidal surface texture

(b) Simulation of the impact of different aperture diameters performed with Daidalos

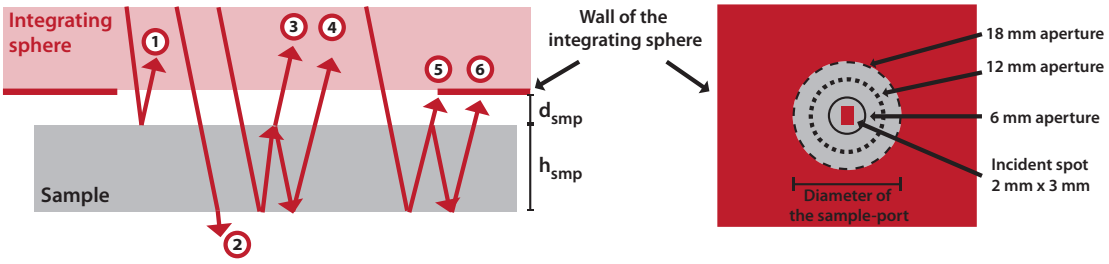
Figure 5.5.: The results produced by Sunrays and Daidalos ray tracing simulations (a) for a pyramidal textured silicon wafer. While both simulations are in great agreement they deviate from the associated measurement (line) by roughly 6 %. This deviation results from light-trapping in the wafer in combination with the aperture used. Further, simulations performed with Daidalos and the comparison with associated measurements (b) confirm this hypothesis.

Second, rays which propagate laterally within the wafer may leave the port region and are, too, hitting the wall of the integrating sphere after leaving the wafer (6).

The impact of these artifacts gets more severe when an aperture is used during measurement, which can be necessary to be able to measure the reflectivity of wafer sample's with dimensions smaller than the sample port of the Cary spectrometer. This is due to the reduced diameter of the opening through which light can be reflected into the integrating sphere (see figure 5.6 (b)). As shown in figure 5.5 (b), using an aperture with a diameter of 6 mm results in loss of a about 17 % of the reflected light. Even using an aperture of 18 mm still results in amount of 3 % of the reflected light getting lost.

As shown in this section Daidalos can be used to perform simulations to investigate the optical characteristics of standard solar cell surface textures. When comparing the simulation results to the measurements it is apparent that good light-trapping capabilities can lead to measurements artifacts that result in an underestimation of a sample's reflectivity. To the authors knowledge this effect and its simulation has not been published before.

Furthermore, it has been shown that Daidalos can be used to simulate the impact of apertures of varying diameter and that these simulations show a sufficient agreement with the made measurements. With respect to the measurement of wafers with good



(a) Path of light during reflection measurement

(b) Wafer surface as seen from the inside of the integrating sphere

Figure 5.6.: During measurements of reflectivity all light which is not absorbed by the wafer's substrate follows one of six paths (a). While most paths (i.e. (1),(3) and (4)) of reflected light lead to correct measurement results, good light-trapping capabilities may lead to light propagating out of the ports region and getting reflected onto the outer walls of the integrating sphere (5,6). If additional apertures are used (b), this effect is increased due to the reduced opening (borders marked by differently dashed black lines) through which light can be reflected into the integrating sphere.

light-trapping it is recommended to avoid any shading aperture. If this is not possible, for example due to the small dimensions of the investigated wafer, the aperture should have a diameter which is significantly larger than the largest dimension of the incident measurement spot. For the shown measurements performed with spot dimensions (see section 5.1) of 3 mm × 2 mm an aperture with diameter of 18 mm reduces the measurement error to about 3 %. In any case a DAIDALOS simulation can be used to estimate the measurement error.

5.4. Complex geometries

This section demonstrates the usability of DAIDALOS for geometries with extended complexity. For this, the reflectivity of a pyramidal textured wafer is simulated, whereby the simulation domain is modeled as large as a scan of a wafer's surface by means of a *laser scanning microscope* (LSM). Thankfully, the processing of the wafers as well as the LSM scans which are shown within this section were executed by Eckard Wefringhaus et al. [33]. The surface texture of two differently etched samples are shown in figure 5.7. Both samples have their front surface as well as their rear surfaces textured with a random pyramid texture. As can be seen, both textures (top) have a similar looking structure. However, the distribution of their pyramid heights (bottom) differs with sample 1 providing pyramid heights centered around 4.5 µm while most pyramids of sample 2 have a height near 6 µm. The measured reflectivity of both samples is provided

in figure 5.8 (left) showing a significant deviation for wavelengths in the range between 400 nm to 1000 nm, with sample 1 showing a reflectivity of up to 1.4 % higher than that of sample 2.

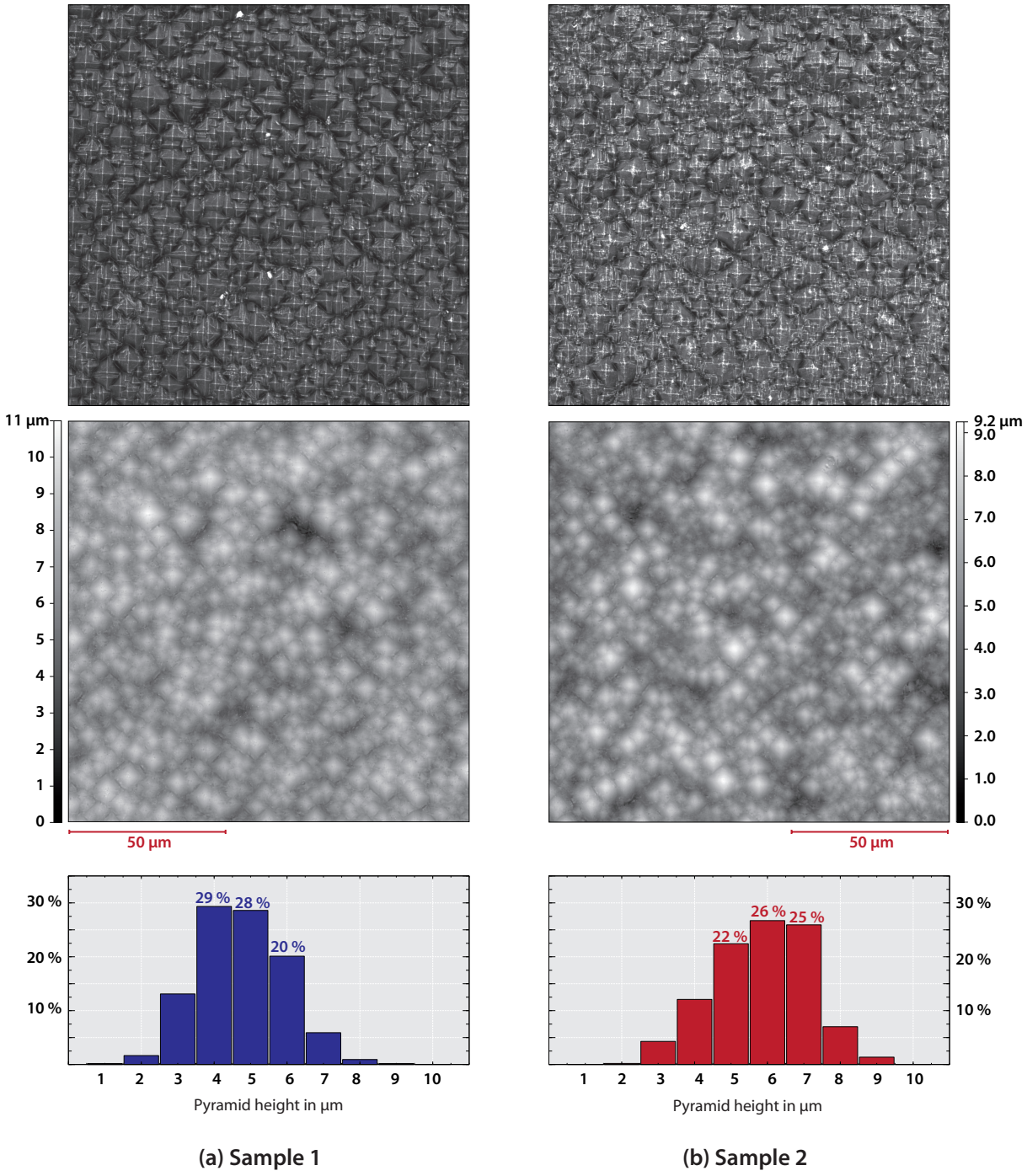


Figure 5.7.: The LSM scans of two differently etched wafers which are referred to as *sample 1* (left) and *sample 2* (right) within the text. The surface structure (top) as well as the height-map (mid) of both samples looks similar. However, the pyramid height distribution (bottom) reveals that while pyramid heights are centered around 4.5 µm for sample 1, most pyramids of sample 2 have a height near 6 µm.

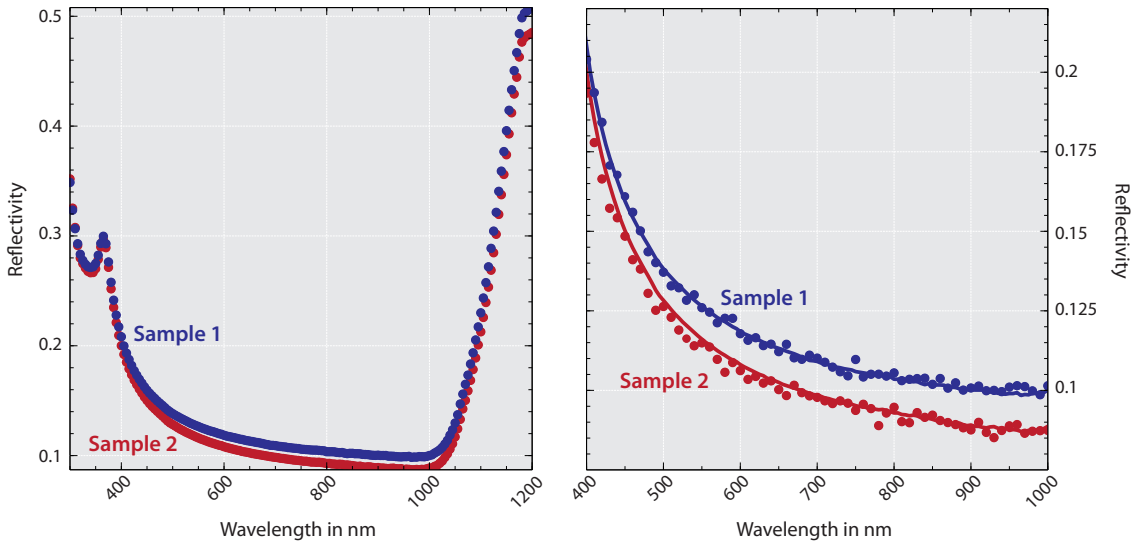


Figure 5.8.: The reflectivity measurements of both samples (left) show significant differences in the wavelength range of 400 nm to 1000 nm. Here, the reflectivity of sample 1 is up to 1.4 % higher than that of sample 2. The simulations which were based upon the LSM scanned surface-textures are in very good agreement with the measurement (b).

In order to reproduce these measurements, a simulation domain was created based upon the gathered LSM scans. For this the coordinates of the pyramid annexes $(x_{\text{anx}}, y_{\text{anx}}, z_{\text{anx}})$ were extracted by Eckard Wefringhaus et al. using a watershed algorithm [33]. It was assumed that all pyramids are ideal (i.e. having a pyramid angle of 54.7°) with their base located at the same height on the 300 µm thick substrate. Due to the former assumption the length of a pyramid's sides can be extrapolated from the calculated height z_{anx} of its annex. These pyramids were than placed on the surface of a 128 µm × 128 µm wide substrate. Finally, the boundaries of the simulation domain were equipped with reflective boundaries and a lambertian factor of 1.0 was chosen for the substrates back.[2] The results of the ray tracing simulations are provided in figure 5.8 (right) and are in good agreement with the made measurements. Nevertheless, it is still a challenge to extract the reasons for the apparent deviation in reflectivity from the shown results. The differences in the distribution of pyramid heights as shown by figure 5.7 (bottom) provides no justification on their own. Instead, it is rather likely that the spatial distribution of pyramids also has to be taken account. For example, high pyramids may lead to a different reflectivity when surrounded by small pyramids than when surrounded by high pyramids. Using

[2]Both sides of the wafer are equipped with a pyramidal texture, but LSM scans only exist for its front. Therefore, a Lambertian factor of 1.0 is an adequate choice. Besides of that, within the considered wavelength range only few rays are reaching the wafer's back, therefore limiting its impact on the optical characteristics.

DAIDALOS it is possible to simulate the optical characteristics of textured wafers and account for variations in the pyramidal texture due to the used etching process. These simulations can be used to optimize the geometry of solar cell textures as well as the associated etching processes. Nevertheless, further investigations are necessary to gain a deeper understanding of the paths that incident light rays take when reflected by the wafer's surface.

6

An advanced light source

This chapter covers the development of a daylight source plugin for DAIDALOS which is based on actual weather data measured at the Institute of Solar Energy Research Hamelin (ISFH) in Germany. In the first part, the global and direct irradiance measurements by means of a pyranometer are introduced. Afterwards, an approach is shown to add additional information to irradiance, namely the wavelength distribution and the angular distribution of light. Finally, the daylight source, as developed by Matthias Winter, is introduced.

6.1. Weather data measurements

The daylight source is based upon irradiation measurements done at the Institute of solar energy research (ISFH) in Hamelin/Germany (52.07° N, 9.35° E) during fourteen years (1992–2005). The measurements were performed using an irradiation measurement device, referred to as *pyranometer*. A schematic of the device's structure is shown in Figure 6.1. Its main components are a thermopile sensor which is calibrated to measure the irradiation between 285 nm to 1300 nm in $\mathrm{W\,m^{-2}}$. A movable light shade is used in two configurations. First, by removing the light shade the thermopile sensor measures the global irradiation on a horizontal plane $I_{\mathrm{glob,hor}}$. Second, by configuring the light shade to block the direct light of the sun during the whole day the sensor measures the horizontal diffuse irradiation $I_{\mathrm{diff,hor}}$. At the ISFH both values where measured during each with a temporal resolution Δt of five minutes. These values incorporate device specific correction factors. For example, a correction is needed due to fact that the light shade does not only block the direct incident irradiation, but also a part of the diffuse irradiation.

Based upon the measured global and diffuse irradiances, the horizontal direct irradiance is calculated by:

$$I_{\mathrm{dir,hor}}(t) = I_{\mathrm{glob,hor}}(t) - I_{\mathrm{diff,hor}}(t). \tag{6.1}$$

This data will be used to derive the angular dependence of sunlight. The sun moves approximately 1.25° in 5 minutes. If we bin the irradiance into 5° × 5° segments,

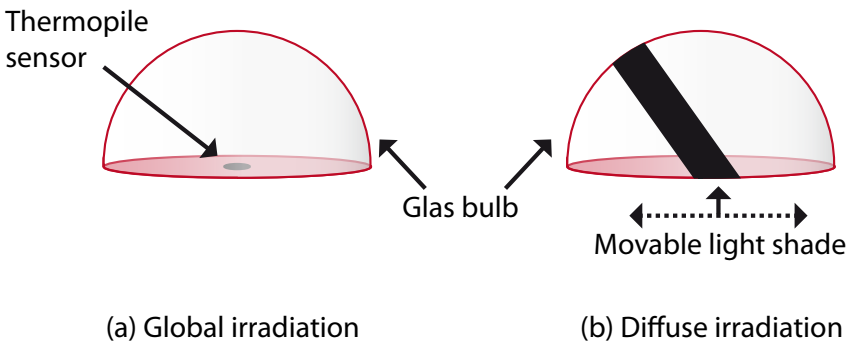


Figure 6.1.: A pyranometer consists of a thermopile sensor which is located under a glass bulb. The thermopile sensor measures the global irradiation (a) in $\mathrm{W\,m^{-2}}$. The configuration can be changed to use a movable light shade which is adjusted to shade the direct light of the sun during the whole day (b). In this configuration, the pyranometer only measures the diffuse portion of the daylight.

considerable binning errors occur. Therefore, these irradiance were linearly interpolated to a temporal resolution of one minute. This means any two consecutive data values $data(t_0)$ and $data(t_0 + 5\,\mathrm{min})$ were interpolated as:

$$data'(t_0) = data(t_0) \tag{6.2}$$
$$data'(t_0 + 1\,\mathrm{min}) = 0.8 \cdot data(t_0) + 0.2 \cdot data(t_0 + 5\,\mathrm{min}) \tag{6.3}$$
$$data'(t_0 + 2\,\mathrm{min}) = 0.6 \cdot data(t_0) + 0.4 \cdot data(t_0 + 5\,\mathrm{min}) \tag{6.4}$$
$$data'(t_0 + 3\,\mathrm{min}) = 0.4 \cdot data(t_0) + 0.6 \cdot data(t_0 + 5\,\mathrm{min}) \tag{6.5}$$
$$data'(t_0 + 4\,\mathrm{min}) = 0.2 \cdot data(t_0) + 0.8 \cdot data(t_0 + 5\,\mathrm{min}). \tag{6.6}$$

As a result, a table is created for each day in the considered interval, consisting of 1440 values of direct, diffuse and global irradiation. Within this chapter, t is a discretized variable which refers to the minutes $t \epsilon [0 \ldots 1439]$ of a particular day for which interpolated measurement data exists. For example, figure 6.2 shows a comparison between the fourteen year averaged global irradiance for the 15th of June (red) and the 15th of December (blue).

6.2. Generating the spectral distribution using SMARTS

The following section describes the solar spectra were calculated using the SMARTS software. Furthermore, an approach is presented for normalizing these spectrally resolved values with the measured irradiances.

As a pyranometer measures the cumulative irradiance there is no information about the prevailing spectra of daylight at the time the irradiances were measured. For that

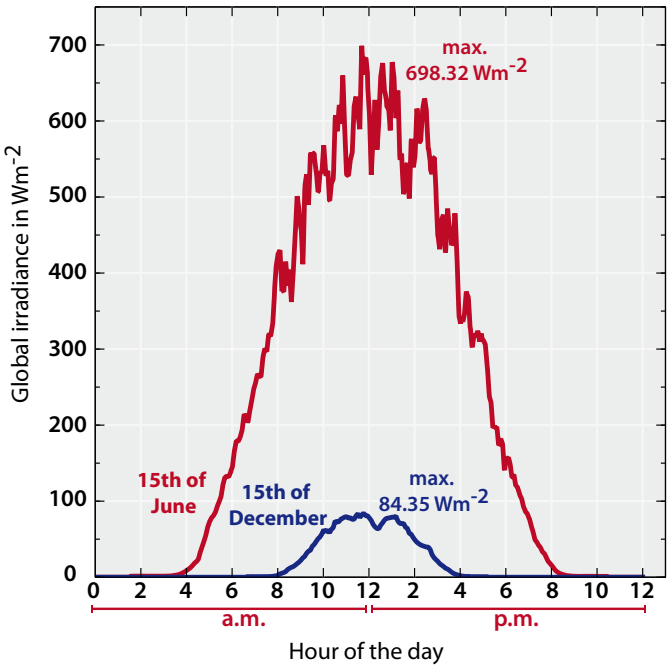


Figure 6.2.: The fourteen year average of the global irradiance for the 15th of December (blue) and the 15th of June (red). During summer the average global irradiance is about eight times higher than in winter.

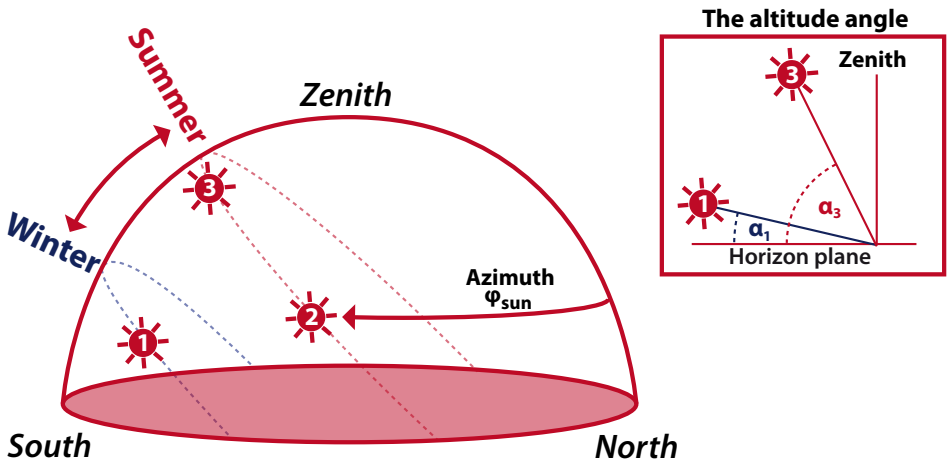


Figure 6.3.: During each day the sun travels over the celestial hemisphere on an circular path (shown as dashed line) which changes over seasons. Using the horizontal coordinate system, the sun's position is described by an azimuthal angle φ measured from the north over east and an altitude angle α measured against the horizon plane (see outtake at the right).

reason, the spectral distributions are simulated with the SMARTS model [34, 35]. It is a standard in photovoltaics. For example, SMARTS was used to calculate the standard AM1.5g spectrum from the measured extraterrestrial AM0 spectrum.

6.2.1. Simulated spectral values

During each day the sun travels along a circular path on the hemisphere, shown in figure 6.3. With respect to the *horizontal coordinate system* the sun's position on the hemisphere is described at any time t by an *azimuth angle* $\varphi_{\text{sun}}(t)$ measured from the north via east and an *altitude angle* $\alpha_{\text{sun}}(t)$ measured from the horizontal plane towards the zenith.

As light propagates through the atmosphere it is scattered or absorbed by particles or molecules. The impact of both processes increases with the amount of traversed air which is described by a value referred to as *air mass* m. For the sun located at the zenith the air mass is defined to be 1.0 and increases with decreasing altitude angle $\alpha_{\text{sun}}(t)$ approximatively as [36]:

$$m(\alpha_{\text{sun}}(t)) = \frac{1}{\sin(\alpha_{\text{sun}}(t))}. \tag{6.7}$$

In order to resolve this altitude-dependent effect the spectral calculations are executed at several altitude angles α_{sun} between 0° to 62.9°[1] at a step-size of 0.1°. The SMARTS

[1]The altitude angle of 62.9° is the maximum altitude reached at the ISFH, where measurements were taken (see section 6.1).

software uses a far more sophisticated approach than equation (6.7) to calculate the influence of the air mass on the solar spectrum. Each SMARTS simulation provides the solar spectrum at ground level for a different solar altitude under standard-atmospheric conditions. The calculations include 91 wavelengths between 300 nm to 1200 nm with a resolution of 10 nm. At each wavelength the following values are calculated:

- **Diffuse horizontal irradiance**
  The diffuse horizontal irradiance $I_{\mathrm{SMARTS,diff,hor}}(\alpha_{\mathrm{sun}})$ is the irradiance by diffuse light measured in $\mathrm{W\,m^{-2}}$ on a horizontal plane.

- **Global horizontal irradiance**
  The global horizontal irradiance $I_{\mathrm{SMARTS,glob,hor}}(\alpha_{\mathrm{sun}})$ is the irradiance by all incident light, measured in $\mathrm{W\,m^{-2}}$ on a horizontal plane.

- **Direct horizontal irradiance**
  The direct horizontal irradiance $I_{\mathrm{SMARTS,dir,hor}}(\alpha_{\mathrm{sun}})$ accounts for the direct part of the incident light, measured in $\mathrm{W\,m^{-2}}$ on a horizontal plane.

- **Direct normal photon flux**
  The direct normal photon flux $F_{\mathrm{SMARTS,dir,norm}}(\lambda, \alpha_{\mathrm{sun}})$ is the photon flux, measured in $\mathrm{cm^{-2}\,s^{-1}\,nm^{-1}}$, as received on a plane which is orientated perpendicular to the direction of incidence.

- **Diffuse horizontal photon flux**
  The diffuse horizontal photon flux $F_{\mathrm{SMARTS,diff,hor}}(\lambda, \alpha_{\mathrm{sun}})$ is the photon flux, measured in $\mathrm{cm^{-2}\,s^{-1}\,nm^{-1}}$, as received on a horizontal plane.

6.2.2. Matching simulated spectra with measurements

The SMARTS-Model allows for a broad set of options to configure the spectra simulations to comply with the conditions of the actual measurement. Nevertheless, an additional scaling procedure is necessary to normalize the simulations to the measured irradiances.

To visualize, consider a partly clouded hemisphere where several white clouds are covering the blue sky. These clouds are effectively masking the diffuse spectrum of the sky and instead are reflecting the direct spectrum of the sun in a diffuse manner. Consequently, the diffuse irradiation as measured by the pyranometer is a superposition of the diffuse spectrum of the blue sky and the diffusively reflected direct spectrum of the sun. Therefore, within this section an approach is presented to estimate the degree of cloudiness from the measured irradiation data and to use it to achieve a good matching between simulation and measurement. The matching procedure can be split up into several steps which are outlined in figure 6.5.

Cloud opacity

The following spectra calculations are made under the assumption that the radiance of a partly clouded sky can be constructed from the weighted sum of the radiances of the

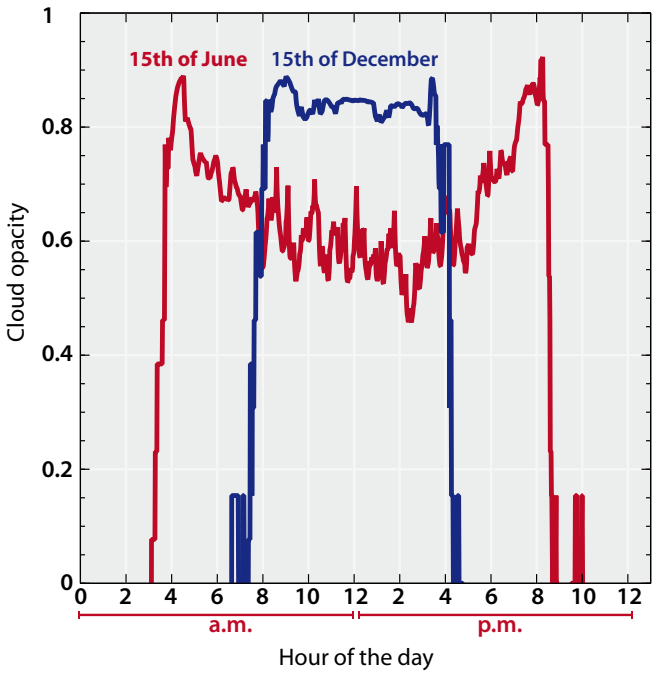


Figure 6.4.: A comparison of the fourteen year average of the cloud opacity calculated for the 15th of December (blue) and the 15th of June (red). Low light conditions in the morning and evening hours lead to measurement artifacts which result into fluctuation of the cloud opacity around the zero-line.

clear sky and the overcast sky. The weighting factor used for this is the *cloud opacity* which is defined to be zero for a blue sky and one for an overcast sky.

According to the semi-empirical model of Gueymard [37] the cloud opacity can be calculated from the values of diffuse and global irradiance. However, during our experiments with the daylight source we found that the evaluation of the cloud opacity according to [37] leads to an overestimation of the light emitted in the ultra-violet regime. Therefore, we used an empirical formula derived by Matthias Winter which is presented in the following. The calculation of the cloud opacity is done during step (3) in figure 6.5 and consists of two parts. First, the ratio of diffuse and global irradiance is calculate. This is done for the measured irradiances as well as for the irradiances simulated by the SMARTS software:

$$\begin{aligned} r_{\mathrm{diff/glob,hor}}(t) &= \frac{I_{\mathrm{diff,hor}}(t)}{I_{\mathrm{glob,hor}}(t)}, \\ r_{\mathrm{SMARTS,diff/glob,hor}}(t) &= \frac{I_{\mathrm{SMARTS,diff,hor}}(\alpha_{\mathrm{sun}}(t))}{I_{\mathrm{SMARTS,glob,hor}}(\alpha_{\mathrm{sun}}(t))}. \end{aligned} \tag{6.8}$$

Second, the cloud opacity is calculated by:

$$c_{\text{opac.}}(t) = \frac{r_{\text{diff/glob,hor}}(t) - r_{\text{SMARTS,diff/glob,hor}}(t)}{1 - r_{\text{SMARTS,diff/glob,hor}}(t)} \tag{6.9}$$

Finally, the value of $c_{\text{opac.}}$ is clamped to the interval of 0 to 1:

$$c_{\text{opac.}}(t) < 0 \quad \Rightarrow c_{\text{opac.}}(t) = 0, \tag{6.10}$$

$$\wedge \quad c_{\text{opac.}}(t) > 1 \quad \Rightarrow c_{\text{opac.}}(t) = 1. \tag{6.11}$$

To visualize, figure 6.4 shows the cloud opacity as calculated for the 15th of December (blue) and the 15th of June (red). The global irradiance for both days were also shown in figure 6.2.

Matching the diffuse and direct photon fluxes

The diffuse spectrum as seen by the pyranometer is modeled as a superposition of the simulated spectra for diffuse and direct light. The particular contributions are weighted by the calculated cloud opacity (see equation 6.9). This weighting-procedure is done during step (7) shown in figure 6.5 and is subdivided into several internal steps which are described in the following.

As preliminary calculation, the direct photon flux $F_{\text{SMARTS,dir,norm}}$ simulated using SMARTS is converted from the normal flux value to the horizontal flux value by:

$$F_{\text{SMARTS,dir,hor}}(\lambda, \alpha_{\text{sun}}) = F_{\text{SMARTS,dir,norm}}(\lambda, \alpha_{\text{sun}}) \cdot \sin(\alpha_{\text{sun}}), \tag{6.12}$$

where α_{sun} is the sun's altitude angle, which can be calculated for any particular geographical location, date and time by means of Ref. [38]. Based upon the simulated values a scaling factor s_{diff} is derived by:

$$\begin{aligned} s_{\text{diff}}(\lambda, t) = \\ \left(\frac{F_{\text{SMARTS,diff,hor}}(\lambda, \alpha_{\text{sun}}(t)) \cdot (1 - c_{\text{opac.}}(t))}{I_{\text{SMARTS,diff,hor}}(\alpha_{\text{sun}}(t))} + \right. \\ \left. \frac{[F_{\text{SMARTS,dir,hor}}(\lambda, \alpha_{\text{sun}}(t)) + F_{\text{SMARTS,diff,hor}}(\lambda, \alpha_{\text{sun}}(t))] \cdot c_{\text{opac.}}(t)}{I_{\text{SMARTS,glob,hor}}(\alpha_{\text{sun}}(t))} \right). \end{aligned} \tag{6.13}$$

Here, the left term describes the contribution of the diffuse irradiance of the blue sky, which is dominant in the clear sky case (i.e. $c_{\text{opac.}} = 0$), and the right term the contribution of reflected direct light, which dominates under overcast conditions.

Finally, the diffuse photon flux is retrieved by multiplication of the scaling factor with the measured diffuse irradiance:

$$F_{\text{diff,hor}}(\lambda, t) = s_{\text{diff}}(\lambda, t) \cdot I_{\text{diff,hor}}(t) \tag{6.14}$$

Similar to that, the direct horizontal photon flux as given by equation 6.12 is matched to the measured direct irradiance I_{dir} (step (8) in figure 6.5) by:

$$F_{\text{dir,hor}}(\lambda, t) = \frac{F_{\text{SMARTS,dir,hor}}(\lambda, \alpha_{\text{sun}}(t))}{I_{\text{SMARTS,dir,hor}}(\alpha_{\text{sun}}(t))} \cdot I_{\text{dir,hor}}(t). \tag{6.15}$$

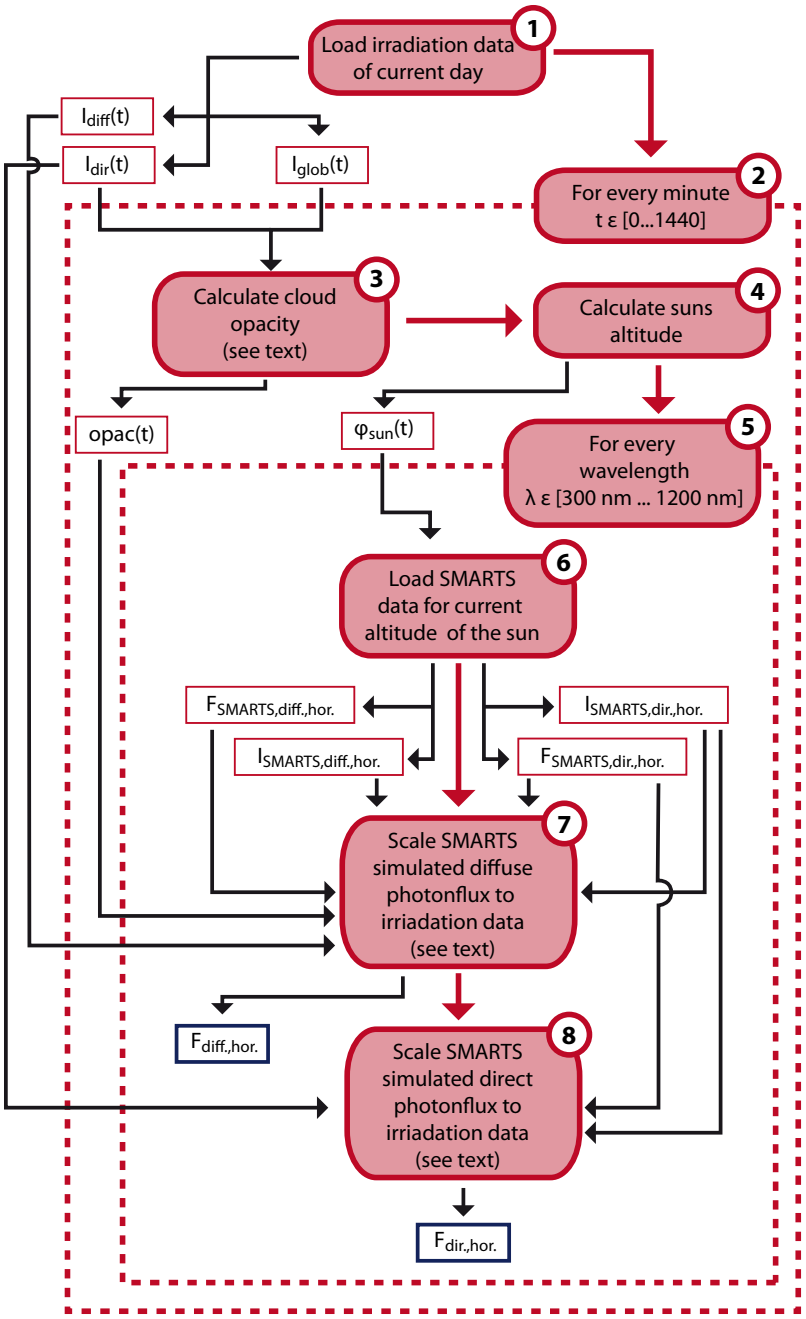


Figure 6.5.: Several steps (rounded rectangles) are necessary to scale the photon fluxes as calculated by SMARTS to the measured irradiances. The procedure consists of two loops (loop-bodies in dashed rectangles) in which the final photon flux values $F_{\mathrm{diff,hor}}$ and $F_{\mathrm{dir,hor}}$ (shown in blue rectangles) are calculated. In the figure the flow of execution is shown by red arrows, while the flow of data is presented by black arrows.

6.3. Generating the angular distribution

In order to perform a meaningful ray tracing simulation the angular distribution of the incident light has to be taken into account, too, not just the specular distribution. That means the photon fluxes of direct and diffuse light, as calculated in the last section, have to be distributed onto different directional segments.

The angular distribution of daylight was investigated and parametrized in detail by Gueymard [37]. This parametrization was incorporated into the in-house *SunCalculator* software developed by Marco Ernst, which was used here to simulate the angular distribution of the calculated photon fluxes and spectra. This section covers the formulas used within the *SunCalculator* software. Additionally, the bin-model used by the daylight source, developed here, is introduced.

Within this section most equations are time-dependent. However, this is not indicated explicitly to keep the shown formulas simple. That means, anytime a value explicitly or implicitly depends upon the measured irradiances $I_{\mathrm{dir,hor}}$, $I_{\mathrm{glob,hor}}$ or $I_{\mathrm{diff,hor}}$, this value actually must be evaluated for any time t within the measurement interval. However, this time dependence is taken into account when describing the final weather data table within section 6.4.

6.3.1. Bin-Model of the direction of radiation

While the sky's hemisphere is a continuous area in the real-world, it has to be discretized for usage within a computer simulation. This discretization is done by partitioning the hemispherical area into a set of bins (see figure 6.6).

In order to comply with the formulas described within this section each bin is addressed by its midpoint, specified by a polar angle θ measured from the zenith and the azimuthal angle φ measured from north over east. The angular size of the bins, which determines the resolution of the resulting grid, is chosen to be $\Delta\varphi = \Delta\theta = 5°$.

6.3.2. Direct irradiation

According to Gueymard [37] the horizontal direct irradiation on an arbitrary orientated plane can be calculated by:

$$I_{\mathrm{dir,hor}}(\beta, \alpha_{\mathrm{sun}}) = \max\left[\frac{\cos(\beta)}{\sin(\alpha_{\mathrm{sun}})}, 0\right] I_{\mathrm{dir,norm.}} \sin(\alpha_{\mathrm{sun}}). \tag{6.16}$$

Here, β is the angle of incidence with respect to the planes normal, $\alpha_{\mathrm{sun}} = 90° - \theta_{\mathrm{sun}}$ is the sun's altitude angle and $I_{\mathrm{dir,norm.}}$ is the value of the measured direct normal irradiation.

With respect to the developed daylight source, the irradiation is always considered to be incident onto the horizontal plane. Therefore, β can be replaced by the sun's altitude angle α_{sun} for the horizon plane, leading to the expression:

$$\begin{aligned} I_{\mathrm{dir,hor}}(\alpha_{\mathrm{sun}}) &= \max\left[\frac{\cos(\alpha_{\mathrm{sun}})}{\sin(\alpha_{\mathrm{sun}})}, 0\right] I_{\mathrm{dir,norm.}} \\ &= \max\left[\frac{1}{\sin(\alpha_{\mathrm{sun}})}, 0\right] I_{\mathrm{dir,hor}} \sin(\alpha_{\mathrm{sun}}). \end{aligned} \tag{6.17}$$

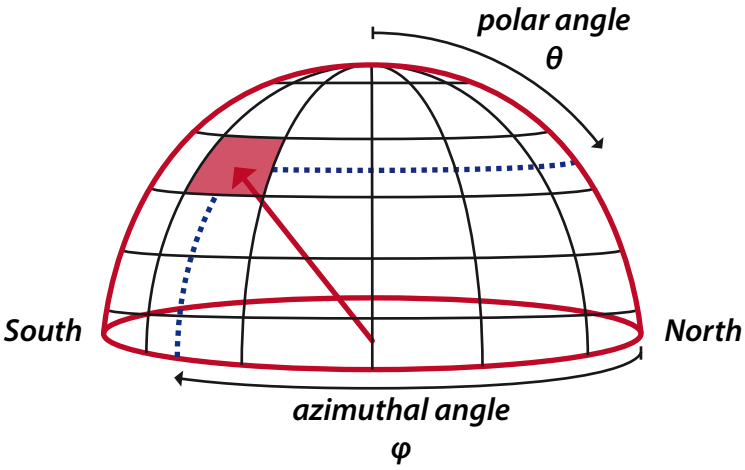


Figure 6.6.: The hemisphere is partitioned into distinct bins. Each bin is associated with its midpoint, specified by a zenith angle θ and an azimuthal angle φ. The resolution of the partition depends upon the angular sizes of the bins which are set to $\Delta\varphi = \Delta\theta = 5°$ for calculation of the daylight source.

Furthermore, the cosine factor is removed from the max-function to convert the given direct irradiation value measured under normal incidence into the corresponding horizontal direct irradiation as measured by the pyranometer.

In order to comply with the coordinate system of the hemisphere's bin-model (see section 6.3.1), the sun's altitude angle α_{sun} is replaced by the corresponding zenith angle θ_{sun}:

$$I_{\text{dir,hor}}(\theta_{\text{sun}}) = \max\left[\frac{1}{\cos(\theta_{\text{sun}})}, 0\right] I_{\text{dir,hor}} \cos(\theta_{\text{sun}}). \tag{6.18}$$

This equation allows the evaluation of the direct irradiance emitted by any bin of the hemisphere depending upon the bins midpoint-coordinate (θ, φ) and the horizontal direct irradiance $I_{\text{dir,hor}}$ as measured by the pyranometer.

6.3.3. Diffuse irradiation

Contrary to the direct irradiation, which impinges only from the direction of the sun, the diffuse irradiation is incident from any direction. Its angular distribution be given by $D(\theta, \varphi)$, which is normalized over all possible angles of incidence [37]:

$$\int_0^{2\pi} d\varphi \int_0^{\pi} D(\theta, \varphi) \cos(\theta) \sin(\theta) d\theta = 1. \tag{6.19}$$

For a spatially discretized hemisphere, these integrals can be reduced to sums. Considering a partition into bins with midpoint-coordinates at (θ_i, φ_i) and a angular resolution of

$\Delta\varphi$ and $\Delta\theta$, respectively, this leads to:

$$\sum_{\varphi_i}\sum_{\theta_i} D(\theta_i,\varphi_i)\cos(\theta_i)\sin(\theta_i)\Delta\varphi\Delta\theta = 1. \tag{6.20}$$

Consequently, the angular distribution of the diffuse horizontal irradiation can be described by:

$$I_{\mathrm{diff,hor}}(\theta_i,\varphi_i) = \overbrace{D(\theta_i,\varphi_i)\cos(\theta_i)\sin(\theta_i)\Delta\varphi\Delta\theta}^{R(\theta_i,\varphi_i)} I_{\mathrm{diff,hor}}, \tag{6.21}$$

where R is a spatial distribution factor. According to Gueymard [37], R depends upon the particular cloud conditions and can be generally expressed by:

$$R = (1 - c_{\mathrm{opac.}})R_{\mathrm{cs.}} + c_{\mathrm{opac.}}R_{\mathrm{ov.}}. \tag{6.22}$$

Here, $c_{\mathrm{opac.}}$ is the cloud opacity factor (see equation 6.9), $R_{\mathrm{cs.}}$ is the distribution factor for clear sky conditions and $R_{\mathrm{ov.}}$ describes the case of an overcast sky. The remaining part of this section will cover the theory needed to calculate the values of $R_{\mathrm{cs.}}$ and $R_{\mathrm{ov.}}$.

Overcast conditions

Following [37, 39–41], the angular distribution $D(\theta,\varphi)$ of the diffuse radiation of an overcast sky only depends upon the zenith angle θ and can be described by:

$$\frac{D(\theta)}{D(0)} = \frac{1 + b\cos(\theta)}{1 + b}. \tag{6.23}$$

According to Gueymard [37], most theoretical and experimental determinations for an overcast sky suggest a value b in the range of 1.0 to 2.0. While the median value of $b = 1.5$ would be a good choice for overcast conditions, experimental observations for a partly clouded sky show a decreases of b with cloudiness. In order to take the latter effect into account Gueymard assumes that b is a linear function of the cloud opacity, given by:

$$b = 0.5 + c_{\mathrm{opac.}} \tag{6.24}$$

Substituting the equation (6.23) into equation (6.20), this results in:

$$D(0)\sum_{\varphi_i}\sum_{\theta_i}\frac{1 + b\cos(\theta)}{1 + b}\cos(\theta_i)\sin(\theta_i)\Delta\varphi\Delta\theta = 1. \tag{6.25}$$

Consequently, $D(0)$ can be expressed by:

$$D(0) = \frac{1}{\sum\limits_{\varphi_i}\sum\limits_{\theta_i}\frac{1+b\cos(\theta)}{1+b}\cos(\theta_i)\sin(\theta_i)\Delta\varphi\Delta\theta}. \tag{6.26}$$

As b can be calculated using the equations (6.24) and (6.9), the value of $R_{\text{ov.}}(\theta_k, \varphi_k)$ for a bin with midpoint-coordinates (θ_k, φ_k) can be calculated by:

$$\begin{aligned} R_{\text{ov.}}(\theta_k, \varphi_k) &\overset{(6.21)}{=} D(\theta_k, \varphi_k)\cos(\theta_k)\sin(\theta_k)\Delta\varphi\Delta\theta \\ &\overset{(6.23)}{=} D(0)\frac{1 + b\cos(\theta_k)}{1 + b}\cos(\theta_k)\sin(\theta_k)\Delta\varphi\Delta\theta \\ &\overset{(6.26)}{=} \frac{(1 + b\cos(\theta_k))\cos(\theta_k)\sin(\theta_k)}{\sum_{\varphi_i}\sum_{\theta_i}(1 + b\cos(\theta))\cos(\theta_i)\sin(\theta_i)} \end{aligned} \tag{6.27}$$

Clear sky conditions

The diffuse light is not distributed homogeneously over the hemisphere. The main reason is Mie scattering at particles comparable in size to the wavelength of the light. Mie scattering is stronger in forward direction than in side direction. This is the main reason why the blue sky near the sun looks whiter than at a large angle away from the sun.

Therefore, with respect to clear sky conditions, the hemisphere is divided into three zones around the sun which are treated by different parameterizations for their diffuse irradiance [42]. The membership of a particular point (θ, φ) of the hemisphere to any of these zones is defined by the angle γ between the point's position-vector and the position-vector of the sun. For values of γ lower than 20°, the point is considered to be part of the *circumsolar zone* which describes the sky's appearance near the sun. This zone is itself subdivided into the C1-zone ($\gamma \leq 3°$) and the C2-zone ($3° < \gamma \leq 20°$). A value of γ bigger than 20° associates a point with the *hemispherical zone* which describes the remaining blue part of the sky.

The C1-Zone ($\gamma \leq 3°$)

The innermost zone is investigated most thoroughly, because it constitutes an important contribution to the sunlight impinging on concentrator systems. According to an analysis of the *Lawrence Berkley Laboratory (LBL)*, conducted between 1975 and 1979, the luminance $N(\gamma)$ in the zone C1 is mainly linear to γ, if presented on a log-log scale, and can therefore be described by [42]:

$$N_{C1}(\gamma) = \gamma^{-\tau} N_{C1}(1), \tag{6.28}$$

where τ is a term which depends on the particular turbidity conditions.

Following Gueymard [43] [42] t can be parametrized by the air mass m and a factor β which can be described by:

$$\begin{aligned} m\beta \leq 1 \quad &\Rightarrow \quad \tau = 6.556(m\beta)^{0.5} - 3.346m\beta \\ 1 \leq m\beta \leq 3 \quad &\Rightarrow \quad \tau = 3.210 + 0.200(m\beta - 1)^{0.5}. \end{aligned}$$

The D-value as described by equation (6.20) is a normalized value with respect to the diffuse horizontal irradiation. Therefore, in order to retrieve the D_{C1} for the zone C1, a

normalization has to be applied to the N_{C1} calculated by equation 6.28. According to Gueymard [42] such a normalization is described by:

$$D_{C1} = \frac{N_{C1}(\gamma)}{N_{C1}(1)} \cdot \frac{N_{C1}(1)}{E_n} \cdot \frac{E_n}{E_d}, \tag{6.29}$$

where E_n and E_d represent the normal and horizontal luminance, respectively. A further evaluation is possible by considering each product term on its own.

The first term of the product can be derived from equations (6.28) and (6.29). The second term is associated with the case of Rayleigh scattering and can be described for $\beta = 0$ and $\gamma \approx 0°$ as [44]:

$$\frac{N_{Rlgh}}{E_n} = \frac{3}{8}\pi m\tau_R. \tag{6.30}$$

Here, τ_R is the optical thickness in the atmosphere associated with the molecules which are the source of the Rayleigh scattering. Following Gueymard [43], the factor $m\tau_R$ can be expressed the parametrization

$$m\tau_R = 0.11008 + 0.07260\ln(m) + 0.04077\ln(m)^2,$$

as long as $1 \leq m \leq 20$ stays valid.

While (6.30) holds for $\beta = 0$, an additional factor $F(m, \beta)$ is needed to describe $\frac{N_{Rlgh}}{E_n}$ for $\beta \neq 0$. Regarding the analysis done by the LBL, $F(m, \beta)$ can be expressed [42] by

$$F(m, \beta) = \exp(A\beta^B).$$

Here, the factors A and B can be further parametrized by m as [42]

$$\begin{aligned} m \leq 2.4 \quad &\Rightarrow \quad A = 8.03 + 0.931m - 0.179m^2 \\ &\wedge \quad B = 0.243 - 0.023m \\ m > 2.4 \quad &\Rightarrow \quad A = 8.093 + m^{0.15} \\ &\wedge \quad B = 0.14 + \exp(-2.211 - 0.342m). \end{aligned}$$

The third term of the product in (6.29) is a function of the turbidity and the mass of the atmosphere. Following Gueymard [42], it can be parametrized by

$$\frac{E_n}{E_d} = \frac{A_0 + A_1\beta}{1 + A_2\beta}.$$

Here, A_0 to A_2 are given by [42]

$$\begin{aligned} A_0 &= 3.17 - 1.98m + 0.059m^2 + 11.36m^{0.5} \\ A_1 &= 3.93 - 3.29m + 0.061m^2 - 8.62m^{0.5} \\ A_2 &= \exp(2.780 + 0.121m - 0.233m^{0.5}) \end{aligned}$$

The C2-zone ($3° < \gamma \leq 20°$)

Within the circumsolar zone C2, Gueymard [42] suggest a parametrization of D_{C2} by

$$D_{C2} = (b_0 + b_1\theta_{\text{sun},deg} + b_2\theta^2_{\text{sun},deg}) \exp(b_3 + b_4\gamma), \tag{6.31}$$

where $\theta_{\text{sun},deg}$ is the zenith angle of the sun in degree and

$$\begin{aligned} b_0 &= 0.109 + 0.029\gamma + 0.005 \exp(-0.015\theta_{\text{sun,deg}} + 1.07 \cdot 10^{-5}\theta^3_{\text{sun,deg}}), \\ b_1 &= 0.02 - 6 \cdot 10^{-4}\gamma, \\ b_2 &= 6.8 \cdot 10^{-5}, \\ b_3 &= 0.24, \\ b_4 &= -0.054. \end{aligned}$$

Hemispherical zone ($\gamma > 20°$)

Considering the hemispherical zone, the luminance does only depend on the altitude but not on the angular distance to the sun. In his paper [42], Gueymard suggest a parametrization of $D_{hem}(\theta)$ by

$$D_{\text{hem.}}(\theta) = (1 + 0.01\theta_{\text{sun,deg}})(0.275 - 0.395\cos(\theta) + 0.170\cos(\theta)^2). \tag{6.32}$$

Complete model of D_{cs}

In order to create the final model of D_{cs}, a renormalization factor K_N has to be introduced, for equation (6.19) to stay valid. According to Gueymard [42], K_N is parametrized by

$$K_N = a_0 + a_1\theta'_{\text{sun,deg}} + a_2\theta'^2_{\text{sun,deg}} + a_3\theta'^3_{\text{sun,deg}},$$

where

$$\begin{aligned} \theta'_{\text{sun},deg} &= 0.01\theta_{\text{sun},deg}, \\ a_0 &= 1.0156 + 0.0907\beta - 0.8644\beta^2, \\ a_1 &= -0.1966 + .15843\beta - 3.8185\beta^2, \\ a_2 &= 0.3651 - 4.8270\beta + 7.9650\beta^2, \\ a_3 &= -0.0113 + 2.010\beta - 2.950\beta^2. \end{aligned}$$

Using K_N, the final D_{cs} is expressed by:

$$D_{cs} = K_N \left(\max\left(D_{C1}, D_{C2}\right) + D_{hem}\right). \tag{6.33}$$

In conjunction with equation (6.21), this can be written as spatial distribution factor:

$$R_{\text{cs.}} = K_N \left(\max\left(D_{C1}, D_{C2}\right) + D_{hem}\right) \cos(\theta_k) \sin(\theta_k) \Delta\varphi\Delta\theta. \tag{6.34}$$

6.4. The final weather data

This section covers the overall process by which the previously presented calculations are executed to generate the final weather data table as used by further simulations, e.g. as input for the developed daylight source plugin (see section 6.5). It stores the data by means of a bin-model presented in section 6.3.1. Therefore, considering an angular resolution of 5° for the altitude angle as well as for the azimuthal angle, the resulting data table consists of $(360/5) \cdot (180/5) = 2592$ rows. Each row consists of several columns which are described in the following text.

Column 1. to 2.: Bin center coordinates

These are the coordinates that describe the center coordinates of the particular bin given as altitude angle α and azimuthal angle φ. Note, that the bins within the final data table are addressed by their altitude angle α instead of their polar angle $\theta = 90.0° - \alpha$.

Column 3.: Directly irradiated energy

This is the energy density of the direct irradiation which incidents from the solid angle represented by the particular bin during the time Δt. It is given in $\mathrm{W\,m^{-2}} \cdot \Delta t$ where Δt is given in seconds. The value is calculated based on equation (6.18) which is summed over all days D of the measurement interval using time steps of $\Delta t = 60\,\mathrm{s}$ by:

$$E_{\mathrm{dir,hor}}(\alpha, \varphi) = \sum_{D} \sum_{t=0}^{1439} \mathrm{isSunPosition}(\alpha, \varphi, t) \cdot I_{\mathrm{dir,hor}}(t, \alpha_{\mathrm{sun}}(t)) \Delta t. \tag{6.35}$$

Here, the sun's altitude angle $\alpha_{\mathrm{sun}}(t)$ for a specific data can be calculated by well-known formulas [38]. The function $\mathrm{isSunPosition}(\alpha, \varphi, t)$ evaluates to one if the sun's position $(\alpha_{\mathrm{sun}}(t), \varphi_{\mathrm{sun}}(t))$ at the time t lies within the bin with center coordinates (α, φ), otherwise it is zero. Accordingly, the direct irradiated energy measured at a time t is always associated with the bin which currently incorporates the sun.

Column 4.: Diffusely irradiated energy

This is the energy density of the diffuse irradiation which incidents from the solid angle represented by the particular bin during the time Δt. It is given in $\mathrm{W\,m^{-2}} \cdot \Delta t$ where Δt is given in seconds. The value is calculated based on the equation (6.22) which is summed over all days D of the measurement interval using time steps of $\Delta t = 60\,\mathrm{s}$ by:

$$E_{\mathrm{diff,hor}}(\alpha, \varphi) = \sum_{D} \sum_{t=0}^{1439} R(90° - \alpha, \varphi, t) \cdot I_{\mathrm{dir,hor}}(t) \cdot \Delta t. \tag{6.36}$$

Columns 5. to 95.: Directly irradiated photon density

These columns contain the spectral photon density (in $\mathrm{nm\,m^{-2}} \cdot \Delta t$) associated with the measured direct irradiance. It is given for wavelengths of 300 nm to 1200 nm with a resolution of 10 nm.

The values are calculated at any time t of any day D within the measurement interval using equation 6.15 and associated with the bin which currently incorporates the sun:

$$F_{\text{dir,hor}}(\alpha, \varphi, \lambda) = \sum_{D} \sum_{t=0}^{1439} \text{isSunPosition}(\alpha, \varphi, t) \cdot F_{\text{dir,hor}}(\lambda, t). \tag{6.37}$$

Here, λ is the wavelength for which the direct photon flux is calculated and the function isSunPosition(α, φ, t) evaluates to one if the sun's position $(\alpha_{\text{sun}}(t), \varphi_{\text{sun}}(t))$ at the time t lies within the bin with center coordinates (α, φ), otherwise it is zero.

Columns 96. to 186.: Diffusely irradiated photon density

These columns contain the spectral photon flux (in $\text{nm}\,\text{m}^{-2} \cdot \Delta t$) associated with the measured diffuse irradiance. It is given for wavelengths of 300 nm to 1200 nm with a resolution of 10 nm.

The values are calculated at any time t of any day D within the measurement interval using equation 6.14. Afterwards they are distributed into the available bins by means of the spatial distribution factor R described by equation (6.22):

$$F_{\text{dif.,hor}}(\alpha, \varphi, \lambda) = \sum_{D} \sum_{t=0}^{1439} R(90^\circ - \alpha, \varphi, t) \cdot F_{\text{dir,hor}}(\lambda, t) \tag{6.38}$$

6.5. The daylight source plugin

This section gives an overview of the implementation of the daylight source as a DAIDALOS plugin, developed by Matthias Winter. As described in section 4.6.2, such a light source plugin has to implement the associated LightSource interface shown in figure 4.11. According to this, any source has to be able to create a Photon (see figure 4.13) providing its wavelength λ, its initial position $\vec{x}$, and direction $\vec{d}$.[2]

Considering the daylight source plugin, its implementation supports two different modes for its appearance during simulation, namely the spherical mode and the box mode. Both modes are based on the final weather data as described in section 6.4 as well as some mode specific, user defined values.

6.5.1. Spherical mode

Using the spherical mode the appearance of the daylight source is most similar to that of the celestial hemisphere. That means the simulated photons are emitted from the inner surface of a sphere with a user defined radius r.

The main challenge of the light source plugin is to calculated the values needed to initialize the simulated photon. These are the photons wavelength λ, its initial position $\vec{x}$, and its direction of propagation $\vec{d}$. They are derived in a two-step process.

[2] Note, the LightSource-interface also provides a photon flux value. However, in most cases (especially if only one source is used for simulation) this value can be any positive number. Therefore, it is neglected in the following discussion.

In the first step, the propagation direction $\vec{d}$ and the wavelength λ is determined based upon the given angular distribution (see section 6.3). For the direction, a solid angle bin (see section 6.3.1) with center coordinates (α, φ) is randomly chosen. The probability $P_{\text{sph}}(\alpha, \varphi, \lambda)$ for a bin with center coordinates (α, φ) of getting chosen is based on the photon flux emitted by the bin and defined by:

$$P_{\text{sph}}(\alpha, \varphi, \lambda) = \frac{(F_{\text{dir,hor}}(\alpha, \varphi, \lambda) + F_{\text{dif.,hor}}(\alpha, \varphi, \lambda))}{\sum\limits_{\alpha_B, \varphi_B} \sum\limits_{\lambda} (F_{\text{dir,hor}}(\alpha_B, \varphi_B, \lambda) + F_{\text{dif.,hor}}(\alpha_B, \varphi_B, \lambda))}. \tag{6.39}$$

Here, the sum over α_B, φ_B represents the sum over all solid angle bins. Each bin has an angular extend of $\Delta\psi$ and $\Delta\theta$, therefore covering a solid angle ranging from $(\theta - \Delta\theta, \varphi - \Delta\varphi)$ to $(\theta + \Delta\theta, \varphi + \Delta\varphi)$. To select a specific direction of propagation within this solid angle, a pair of random offsets:

$$(\delta\theta_{\text{rand.}}, \delta\varphi_{\text{rand.}}) \qquad \text{, where} \quad \delta\theta_{\text{rand.}} \epsilon\, [-\Delta\theta, \Delta\theta)\,, \quad \delta\theta_{\text{rand.}} \epsilon\, [-\Delta\varphi, \Delta\varphi)\,, \tag{6.40}$$

is calculated. This leads to a direction vector of:

$$\vec{d}_{\text{phot.}} = \begin{pmatrix} -\sin(\theta + \delta\theta_{\text{rand.}})\cos(\varphi + \delta\varphi_{\text{rand.}}) \\ -\sin(\theta + \delta\theta_{\text{rand.}})\sin(\varphi + \delta\varphi_{\text{rand.}}) \\ -\cos(\varphi + \delta\varphi_{\text{rand.}}) \end{pmatrix}. \tag{6.41}$$

In the second step, the initial position $\vec{x}$ of the photon is determined. If any obstacles are neglected the light within the sphere must be homogeneous as shown in figure 6.7 (1). That means that light which incidents from a particular direction is visible from any point within the sphere. This formulation is equivalent to the representation where the incident light does homogeneously irradiate the area of the sphere projected on a plane perpendicular to the light's direction (2). For this reason, the initial position on the sphere's surface is determined by randomly choosing a point within the projected circle (3) and projecting this point back onto the hemisphere (4).

Finally, the resulting position is rotated to match the direction chosen and shifted to match the position of the sphere as defined by the user (figure 6.7).

6.5.2. Box source

While the spherical mode of the day light source is most similar to the common observation of a hemispherical sky, there are some situations where a box shaped source is preferable. For example, consider the simulation of a common solar cell module of rectangular shape. When covering the module with a hemispherical source, a significant amount of photons won't strike the module at all, but pass without any impact to the simulations result. By switching to a box shaped representation of the light source the modules outline can be tightly surrounded by the source. Accordingly, the same statistical error (see equation 3.9) can be reached using a decreased number of simulated photons.

The internal working within box-mode is more complex than that of the spherical-mode. It can best be visualized by thinking of the user defined box to be located within an

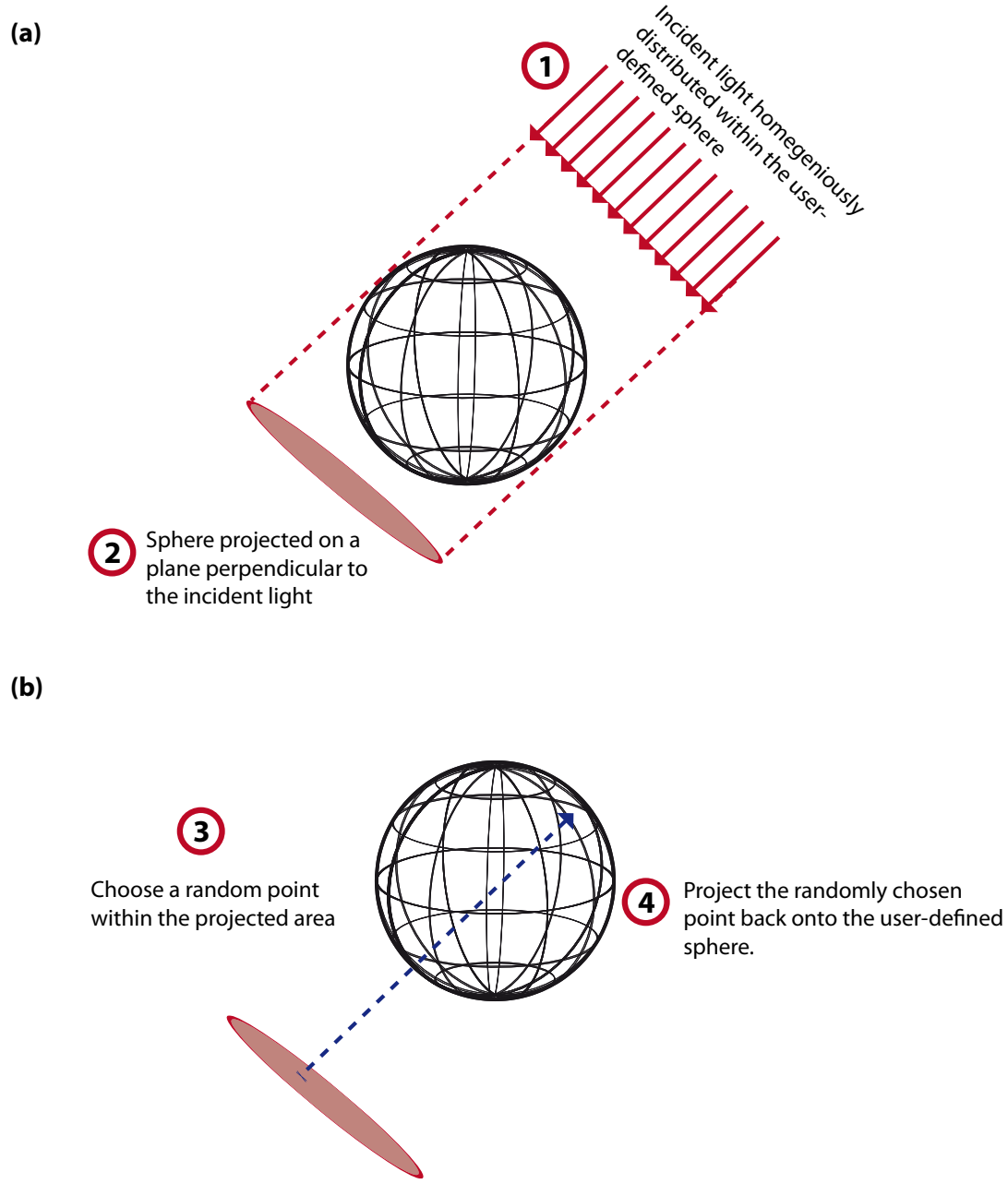


Figure 6.7.: In the second step, the initial position of the photon is calculated. The distribution of light is homogeneous within the sphere (a). That means if all obstacles are neglected the light incident from a particular direction can be seen from any location within the sphere. Therefore, it is uniformly distributed on the circle which represents the area of the sphere, projected perpendicularly to the light's direction (2). The initial position is then determined (b) by choosing a random point on the projected area (3) and calculating the intersection with the hemisphere along the direction of the incident light (4).

irradiating sphere. As preliminary step, the parallel projection of each of the six box faces is calculated for any solid angle bin. Therefore, given a bin with the center coordinate (α, φ) the area A_i of the i-th box face is projected onto an area $A(\alpha, \varphi, i)$ as seen by the bin. In order to resolve possibly rapid changes in the area seen by the bins[3], the available bins are previously subdivide to increase their angular resolution to exceed a user-defined threshold, which defaults to $1^\circ \times 1^\circ = 3.046 \times 10^{-4}$.

This preliminary process is followed by the first step which consists of choosing the propagation direction of the emitted photon. As for the hemispherical mode, this is done by selecting a random solid angle and wavelength. However, due to the dimensions of the box, different bins are seeing differently sized projections of the box faces $A(\alpha, \varphi, i)$ that have to be taken into account. For this reason, the probability of a bin to getting selected is defined by:

$$P_{\text{box}}(\alpha, \varphi, \lambda) = P_{\text{sph}}(\alpha, \varphi, \lambda) \cdot \frac{\sum\limits_{i} A(\alpha, \varphi, i)}{\sum\limits_{\alpha_B, \varphi_B} \sum\limits_{i} A(\alpha, \varphi, i)}, \tag{6.42}$$

where $P_{\text{sph}}(\alpha, \varphi, \lambda)$ is the corresponding selection probability of the spherical-mode as given by equation 6.39.

At last, the initial position of the photon, located onto one of the box faces, is determined. This is done, by first selecting a random box-face which is seen by the emitting bin. Here, the probability $P_f(\alpha, \varphi, i)$ of a particular face i of the box for getting selected by a bin with center coordinates (α, φ) is given by:

$$P_f(\alpha, \varphi, i) = \frac{A(\alpha, \varphi, i)}{\sum\limits_{i} A(\alpha, \varphi, i)}. \tag{6.43}$$

Finally, the initial position of the photon is randomly selected (uniformly distributed) on the previously chosen face.

[3] For example, one bin might just barely see the top-face of the box while this is completely hidden for its neighboring bin.

7 Simulating module optics

This chapter covers the simulation of the optical features of solar cell modules. It starts with an overview of the components of a solar cell module. This is followed by the introduction of the *Laser beam induced current* (LBIC)-method [45, 46] which allows for an investigation of the optical impact of the different module components.

While ray tracing simulations of module optics were done before, these simulations were either done on simplified module geometries [47–49] or by using analytical calculations [50–52]. Contrary to that, within this chapter a multi-domain approach is presented which allows for the simulation of widely extended areas of solar modules within acceptable simulation time. Finally, based upon the presented approach the impact of the inter-cell distance is discussed.

7.1. Module optics

A common solar cell module is composed of several components as shown in figure 7.1. The actual solar cells are arranged into a grid and interconnect by metal-connectors. This solar cell grid is embedded into an encapsulation layer which is commonly made of either ethyl-vinyl-acetate (EVA) or silicone.[1] Finally, the encapsulated cells are placed within an enclosure made of a backsheet behind the cells, a glass pane at the front and a metal frame which supports stability. Each of these components has an impact on the optical characteristics of the final solar cell module and increases the number of possible paths of light which have to be taken into account during a ray tracing simulation (see figure 7.2).

Following the path of the incident light, as shown in figure 7.2, the top glass is the first component which is hit by the incoming rays. Commonly, this is made of glass with a low iron content to reduce the amount of absorbed light (case 2). Additionally, an anti-reflection coating can be applied to the glass outer surface in order to reduce its reflectivity (1). Those rays which are not lost by these two processes reach the interface

[1] While silicone shows lower parasitic absorption, EVA is the cheaper material and therefore the prevalent encapsulation material within commercially available solar cell modules.

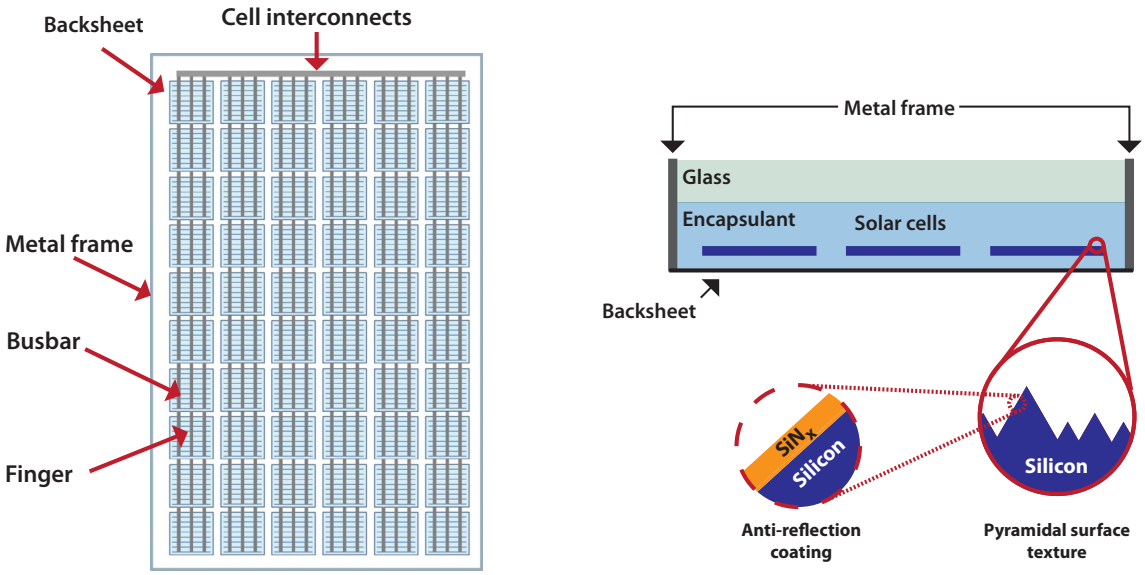


Figure 7.1.: A solar cell module (left) is composed of several components. The actual solar cells are arranged into a grid-layout, interconnected by metal ribbons and embedded into an encapsulation material (see text). This is enclosed by a rear-backsheet, a front-glass and a metal frame. Each solar cell is equipped with a pyramidal surface texture and an anti-reflection coating to reduce its reflectivity (right).

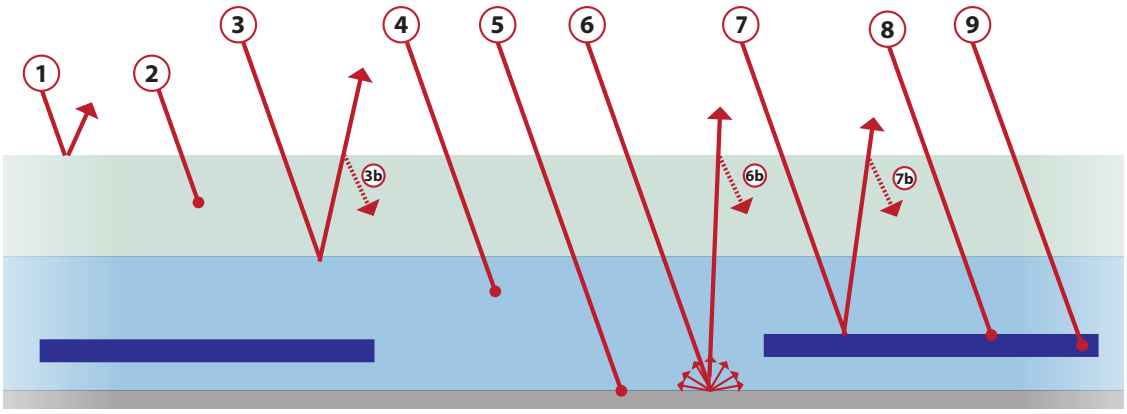


Figure 7.2.: There are several path by which a photon can propagate through a solar cell module (see text). Most of these paths are not leading to any contact with a solar cell.

between the top glass and the encapsulating material. Depending upon the encapsulation material used and the angle of incidence, this interface can reflect (3) some amount of light. Furthermore, light can get trapped within the glass layer undergoing multiple alternating reflections at the glass/encapsulation and the glass/air interface (3b)(6b)(7b). Considering the low absorbance of the used glass, trapped light can propagate large lateral distances (in the range of centimeters) in the glass layer before eventually leaving it through one of its interfaces.

All rays which are not reflected at the glass/encapsulation interface are entering the encapsulation material. In the commonly used EVA, most rays of wavelengths between 320 nm to 360 nm are absorbed within this layer (4) before reaching the actual solar cells. Considering the amount of space (i.e. the gap) between neighboring solar cells and between solar cells and metal frame (see figure 7.1), even the rays which are not absorbed by the EVA have a good chance to hit the backsheet instead of the actual solar cell area.[2] Here, a small part of the light gets absorbed (5) while the major part is reflected in a diffuse manner (6). A fraction of the incident rays eventually hits the surface of a solar cell. While some of these rays are reflected (7) either by the solar cells metalization (i.e. electrical contacts) or the pyramidal textured cell surface (see figure 7.1) others are possibly absorbed (8) by the applied anti-reflection coating. Finally, the remaining rays which enter the solar cell can be absorbed within the cell's body (9) and contribute to the generated current. Rays entering the solar cell may leave the cell without being absorbed, as discussed in chapter 5.

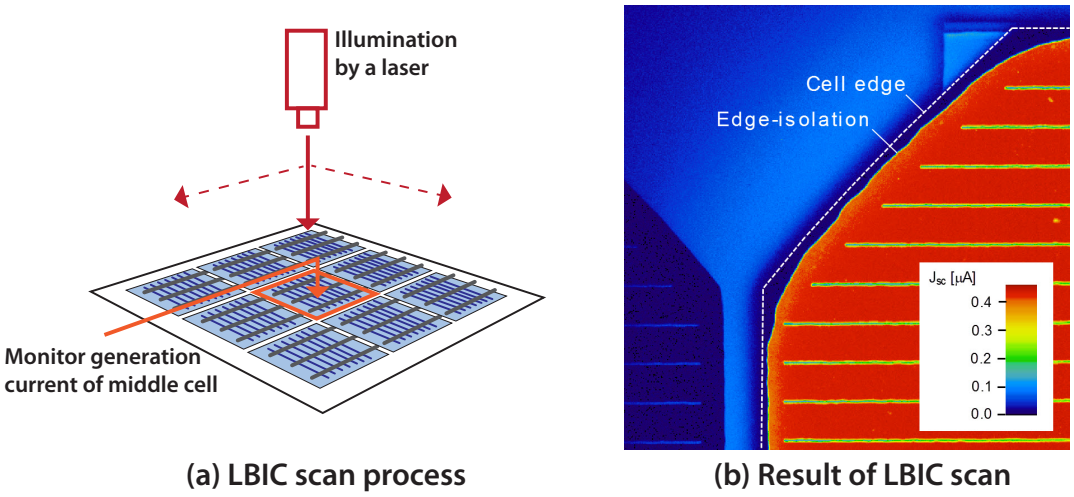
Due to the shown processes, which each have their impact on the optical characteristics of the final solar cell module, it is not possible to directly deduce the module's optical properties from the characteristics of its components. In order to investigate the optical impact which results from a change in any of these components, the full module has to be considered within a simulation to account for the interdependencies with other components.

7.1.1. Laser beam induced current

Within this section the *Laser beam induced current* (LBIC) method is described that allows for an experimental investigation of the optical impact of the module components. Using this method, a single solar cell of a module is contacted separately and the generated current is measured. A laser beam is then moved over the module's surface illuminating it at predefined points. Finally, each irradiation point is associated with the current generated within the contacted solar cell while that point was irradiated.

As an example, a 3x3 solar cell mini-module was investigated by an LBIC scan with the central cell contacted and its generation current monitored, see figure 7.3(a). A cutout of the resulting bitmap is shown in figure 7.3(b). The pixel colors are representing the generation current as measured for the solar cell shown on the right. It's no surprise that most current is generated when the solar cell surface is directly illuminated by the laser beam. In this case, the main current loss is due to the reflection by the metalized areas

[2] The backsheet makes up for roughly 10% of the module area seen by perpendicular incident light.



(a) LBIC scan process (b) Result of LBIC scan

Figure 7.3.: The result of an LBIC scan. The solar cell on the right is contacted and the generated current is measured. Each pixel's color maps to the generation current in the contacted cell when that pixel was irradiated. While most current is generated when the cell is directly illuminated, even the illumination of the backsheet and the metallization of the neighboring cell lead to significant current generation.

of the cell (i.e. the fingers and the busbar; see figure 7.1). However, some fraction of the light first reflected by the fingers is subsequently reflected back onto the cell, hence the metalized area still generates a current of about 0.2 µA. This effectively reduces the total shading area of the metalization by a factor which is referred to as *shading factor*. Furthermore, looking at the edge of the contacted cell as marked in figure 7.3(b), a small region at the cells edge has no electrical contact (i.e. cannot contribute to the generated current). This is due to an edge isolation process during solar cell production.

However, the most important point to be extracted from the shown LBIC result is the fact that the backsheet and even the metalized areas of the neighbor cell measurably contribute to the monitored generation current. These contributions result from reflected light which is reflected at the front side of the glass, as shown by the paths (6b) and (7b) in figure 7.2. This effect leads to a long-range optical coupling between laterally separated parts of a module. Furthermore, it eliminates the possibility to calculate the module's optical characteristics by combining the results of several separately executed simulations of small-sized parts of the module geometry. Instead, each simulation has to take into account a simulation-domain which spans at least the distance a photon can travel due to multiple reflections at the EVA/glass or glass/air interface, respectively.[3]

[3]The actual distance depends upon the used glass and EVA thickness and lies in the range of centimeters for common solar cell modules.

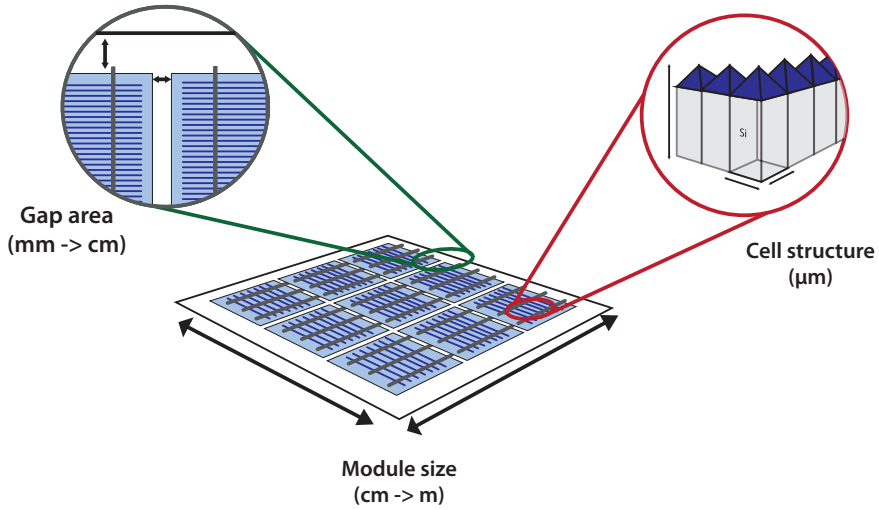


Figure 7.4.: The geometry of a module can be represented by gap-domain (right) representing the inter-cell and cell-to-frame gap, as well as a cell-domain modeling the inner cell structure made of periodically repeating unit-cells (left).

7.2. Modeling the module geometry

Within this section an approach is presented to model the geometry of a solar cell module. As shown in the previous section, there exists an optical long-range coupling between laterally separated parts of a module. For this reason, it is not sufficient to perform ray tracing simulations on separated parts and to expect that the combination of the gained result will represent the actual optical characteristics of the module.

The major problem in modeling a module geometry is the large difference in scale of the composing parts. While the size of a usual solar cell module is in the range of meters, the dimensions of the individual solar cells are in the range of centimeters. Furthermore, each solar cell provides a pyramidal surface texture (see figure 7.1) with dimensions in the range of micrometers. This results in an amount of roughly 900 million pyramids to be modeled for each solar cell. Consequently, a ray tracing simulation which incorporates all geometrical details of a full solar cell module cannot be performed within acceptable time on todays computers.

7.2.1. A multi-domain approach

Instead of modeling of a full solar cell module as a whole, a multi-domain approach is used. For this, the module geometry is represented by only two simulation domains, as shown in figure 7.4. While the inter-cell and the cell-to-frame gap is represented by the gap-domain (left), the inner area (i.e. not directly adjacent to a gap) of a solar cell is represented by the cell-domain (right). As aforementioned, these two domains cannot be simulated separately in order to account for any long-range coupling effects.

To visualize, consider a ray of light which illuminates the surface of the cell, as shown in figure 7.5(a). First, it propagates through the cell's interior before leaving it through an edge orientated towards the inter-cell gap. It travels through the gap by alternating reflections at the glass/air-interface and the backsheet; and finally leaves the module just above the neighboring cell.

Using DAIDALOS, such long-range propagations can be easily taken into account by connecting both simulation domains during the ray tracing process. For this, a BoundaryEffect (see section 4.6.7) is applied to each boundary face of those domains. The ray tracing simulation starts by generating a new photon above the inner-cell domain (1). The photon propagates through this domain, until it hits one of its boundaries. In this case, the applied BoundaryEffect evaluates whether the photon resides in the cell-domain (2) or should be transfered to the gap-domain (3). In the later case, the photon further propagates through the gap-domain until it is transfered back to the cell-domain (4).

It should be highlighted that this approach only required the development of one extra plugin, namely the used BoundaryEffect. All other functionality, like creation of geometry, representation of materials and the ray tracer itself are features which are either provided by the DAIDALOS framework utilities (see section 4.3.2) or can be reused from earlier simulations (e.g. the ray tracer or the plugin to represent the cell's nitride coating). Furthermore, using additional simulation domains, the level of detail of the ray tracing simulation is almost arbitrary expendable. An example for such an extension is reference [53].

7.3. Simulation of the optical impact of the gap-distance

Within this section the usability of the presented modeling approach is demonstrated by investigating the optical impact of the inter-cell gap distance. This is done by reproducing an LBIC scan through two differently sized inter-cell gaps.

As shown in section 7.1.1, light which impinges on the area of the backsheet can be reflected back onto nearby cells. This effect can be interpreted as an effective increase of the light collecting area of the cell. However, as can be seen from figure 7.3, this light harvesting effect quickly decreases with increasing distance to the cells edge. For this reason it makes sense to evaluate the impact of the particular gap distance on the generated current.

7.3.1. Simulation model and materials

Using the multi-domain approach introduced in section 7.2.1, both gap simulations were execute with the dimensions shown in figure 7.6 (i.e. simulating a 2 mm narrow-gap and a 25 mm wide-gap.).

The wavelength-dependent refractive index n and extinction coefficient k for crystalline silicon were taken from [54]. Additionally, a 76 nm thick SiNx layer was applied as anti-reflection coating on the pyramids of the solar cells front-texture. The refractive index was modeled based on in-house measurements and the results are tabulated in the appendix. The optical data for the EVA encapsulation material were taken from [54] while

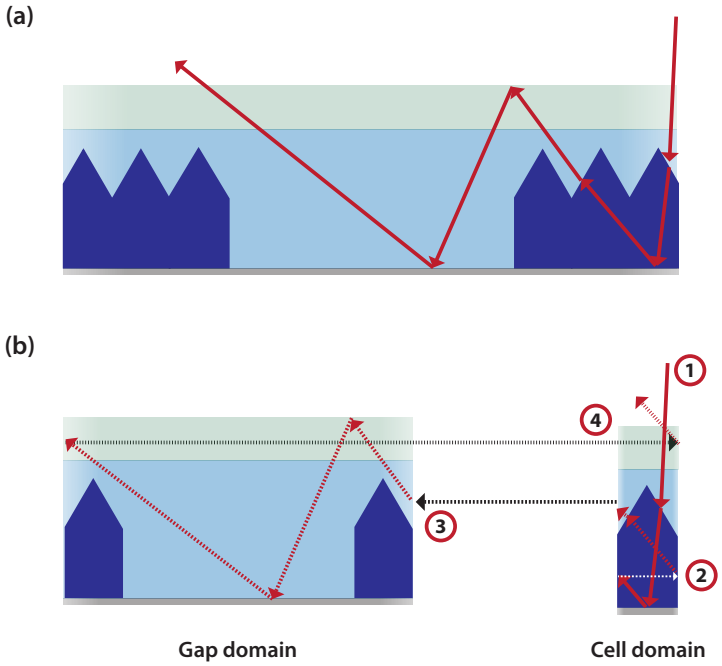


Figure 7.5.: A photon which travels through the module (a) can cover the cell as well as the gap domain. In the simulation a BoundaryEffect connects the two domains. It tracks the photons overall position and decides whether the photon is shifted within the current domain (2) or is switched from one domain to the other (3+4).

the reflectivity of the backsheet is set to $R = 0.85$.[4] Furthermore, a RefractionCalculator-service (see section 4.6.6) was developed and applied to the face representing the backsheet to force Lambertian reflection.

An additional absorption free anti-reflection coating is applied to the top of the front glass, providing $k = 0$ and n equal to a factor of 0.9 times the n of glass. These fictive n values come close to those values we measured in-house from commercial manufacturers.

7.3.2. Simulation results

As aforementioned, the ray tracing simulation is executed to reproduce an LBIC scan through a gap. For this reason, a point source is used as LightSource (see section 4.6.2) and moved starting from the cell towards the gap.

A high spatial resolution (i.e. small step size of the moving point source) is used to evaluate the near-edge region for the small-gap simulation, with the result shown in figure 7.7(a).[5] As can be seen, a direct illumination of the solar cell surface results in a

[4] During the simulations, this value has proven to best represent the backsheet of the measured mini-module (see figure 7.3)

[5] While a gap of 2 mm was simulated, here only the first 400 µm are shown as all effects are visible within

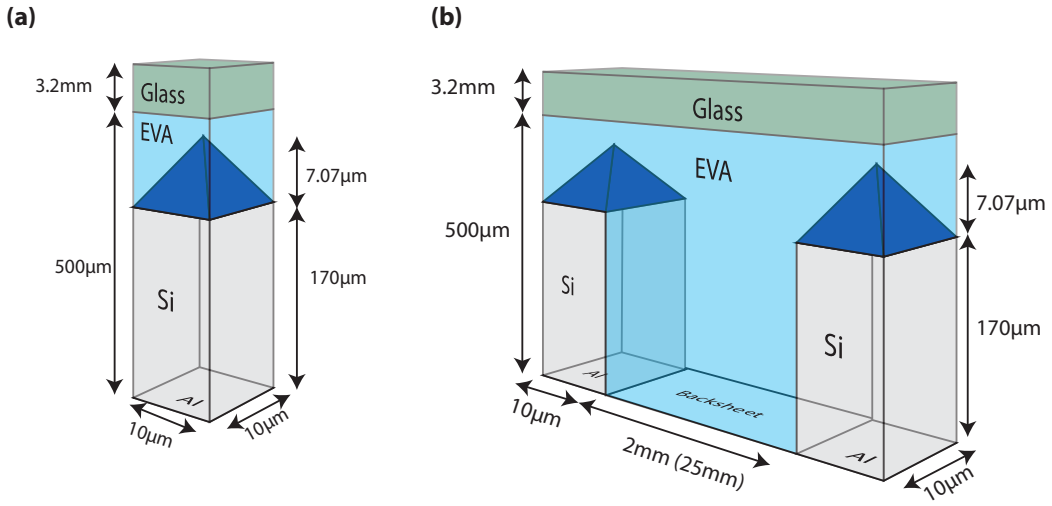


Figure 7.6.: The models (not to scale) of cell-domain (a) and the gap-domain (b) as used the simulations. While the backsides of the cells is considered to be fully metalized by aluminum, the backside of the gap is considered to be filled with the backsheet. Note, while the gap is 2 mm wide for the narrow gap simulation this is changed to 25 mm for the wide gap case.

photo-generated current of about 40 mA cm^{-2}. This drops to half as soon as the LBIC beam enters the gap and decreases further while moving farther away from the cell edge, see (2) in figure 7.7. This behavior can be explained as shown in figure 7.8. A ray of light hitting the backsheet is reflected into all directions due to the backsheet's Lambertian reflection. This means, when being near the cell edge only the half of the incident rays are reflected towards the cell and can actually be absorbed. Consequently, the generated current is reduced by 50 %. When moving farther away from the cells edge, an increasing part of those rays which are reflected towards the cell hit the glass/air interface. Total reflection occurs only for incident angles greater than roughly 40°. For a fully Lambertian backsheet a fraction of about 35 % of the reflected rays escapes from the module.

Especially for wide gaps as shown in figure 7.7(b), yet another effect has to be taken into account. Consider an illumination of the gap at a distance x measured from the illuminated location to the nearest cell. Assume further that the ray reflected by the backsheet incidents on the glass/air interface with an angle of incidence $\gamma = 40°$ which just allows for total reflection. By relying on geometrics, the lateral distance which the ray travels by this single reflection can be calculated by:

$$d_{\min} = 2h \tan(40°) \approx 1.68h. \tag{7.1}$$

Here, $h = h_{\mathrm{EVA}} + h_{\mathrm{glass}}$ is the vertical distance from the backsheet to the top of the front-glass. That means if the distance x between the illumination point on the backsheet

this range.

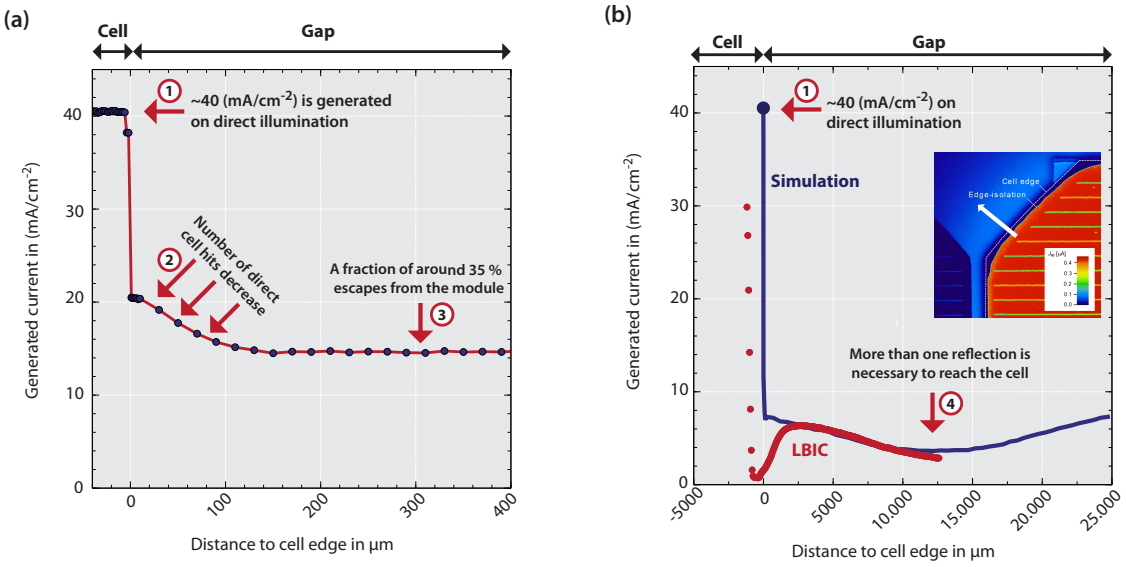


Figure 7.7.: The result of a ray tracing simulation of a 2 mm wide inter-cell gap (a) shows a generated current of about 40 mA cm^{-2} for a directly illuminated cell (1). This is reduced to the half as soon as the cells edge is reached (2) and decreases further when moving farther away (3). Simulating a gap of 25 mm (b) width it can be seen that up from a specific distance some rays need more than one reflection at the glass/air interface to reach the solar cell (4). As shown, the course of the simulated curve is in good agreement with the one measured in the 3x3 mini-module. Note that the measured curve is scaled here to ease comparison.

and the nearest cell edge is greater than $d_{\min}$ some of the reflected rays need subsequent reflections at the glass/air interface to reach the cell (see figure 7.8(4)).

This effect was investigated by performing a ray tracing simulating of a 25 mm wide gap. The results are shown in figure 7.7(b). The simulation result was compared to a line-scan (shown on the top-right) taken from the LBIC scan shown in figure 7.3. The shown measurement result is scaled to ease the comparison between both curves. There exist some differences near the cell edge which result from two features not included in the simulation. First, there is the edge isolation as aforementioned in section 7.1.1. And second, the solar cells within the measured mini-module are not placed directly on the backsheet as it is the case in the simulation, but are placed with a little offset. For this reason, rays which are incident on the backsheet near the cells edge (see figure 7.8(2)) are reflected under the cell and can not contribute to the generated current.

Beside these differences there is a good agreement between both curves within the gap region. Furthermore, the expected current loss can be seen, reaching its maximum at the

gap midpoint.

7.3.3. Increase in photo-generation current for full square solar cells

Within this section the photo-generation current $I_{\mathrm{gap}}(x)$ as simulated for a location within gap with a distance x to the next cell edge is used to approximate the additional current generated within a full square solar cell. The presented approximation neglects the electrical properties (e.g resistance of the cell interconnection) of the final module as well as those of the solar cell (e.g. lifetime of the charge carriers).

The illumination of a small inter-cell gap of 2 mm leads to an additional photo-generation current $I_{2\mathrm{mm}}(x)$ which depends upon the distance x between the illuminated point of the backsheet and the nearest cell edge. It is shown in figure 7.7(a). Within the shown simulation n illumination points x_n were placed with a spacing Δx within the gap. Therefore, the photo-generation current coming from a narrow line through the gap can be calculate by:

$$I_{\mathrm{1D,2mmGap}} = \sum_{i=1}^{n} I_{2\mathrm{mm}} \cdot \Delta x \approx 1.5\,\mathrm{mA\,cm^{-1}}. \tag{7.2}$$

Common full square solar cells have a side-length of 15.6 cm in both directions. Therefore, the additional photo-generation current generated by a 2 mm gap evaluates to 23.4 mA. Executing these calculations for the 25 mm wide gap results in an additional generated current of about 100.57 mA.

A common full square solar cell has a surface area of $(15.6 \times 15.6)\mathrm{cm}^2 = 243.36\mathrm{cm}^2$. As shown in figure 7.7 a direct illumination of the cell results in a photo-generation current of 40 $(\mathrm{mA/c^2m})$. Therefore, a current of about 9734.4 mA is expected for a homogeneous illumination of the cell's surface area.

To conclude, a solar cell module as shown in figure 7.1 consists of 60 solar cells. From these, 24 solar cells are located at the module edge and being surrounded by three small inter-cell gaps and one large cell-to-metal gap. Therefore, those cells generate an additional current of $3 \times 23.4\,\mathrm{mA} + 100.57\,\mathrm{mA} = 123.97\,\mathrm{mA}$ due to the light reflected from the gap. Doing this calculation for the 4 solar cells at the corners and the 32 inner solar cells this results in additional photo-generation currents of 247.94 mA and 93.6 mA,respectively. Finally, this results in an additional photo-generation current of 6.96 A from the gap compared to a current of $60 \times 9734.4\,\mathrm{mA} = 584.064\,\mathrm{A}$ as generated by direct illumination[6]. Consequently, the gap is responsible for roughly 1 % of the total photo-generation current.

[6] This amount of photo-generation current would only be available from the module if all solar cells were connected in parallel. As can be seen from figure 7.1 this is usually not the case, but was chosen here to ease the comparison.

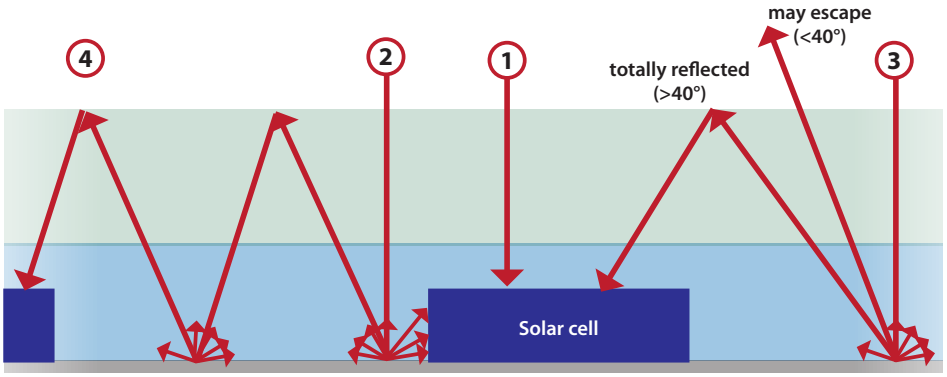


Figure 7.8.: The amount of light actually hitting the solar cell decreases if moving away from direct illumination (1). While being near the cell roughly 50 % of the diffusively reflected rays directly hits the cell (2). This further decreases (3) when moving farther away as around 35 % of the reflected rays are not subject to total reflection at the glass/air interface. Finally, for wide gaps, often one or more reflections at the glass/air interfaces are needed to reach the cell surface (4).

8

Simulations of facades

This chapter covers the simulation of the annual irradiation which incidents on the facades of chosen buildings. For this, the day light's spectral and angular distribution is considered by using the developed daylight source (see Chapter 6).

8.1. Facade vs. rooftop installations

Facade installations of either photovoltaic or photothermal systems is nothing new, but is used and discussed since more than a decade [55–57]. While most solar systems today are installed on roofs, facade installations can provide some advantages when considering the large-scale supply with solar generated energy [58]:

- **Availability of space**
  Todays cities provide large areas of facades available for solar system installations possibly outnumbering the usable space for roof top installations. This larger area may compensate for the less received power on the vertical facades. In section 8.4 the simulation of a common city building is presented to demonstrate the effect of vertical facades as well as shadowing due to nearby neighbor buildings.

- **Broadening of the peak-power interval**
  Common roof top installations reach their maximum performance around midday when the suns light incidents almost perpendicular onto the roof. However, considering the further growth of solar system installations it might be favorable to spread the power peak towards the morning and evening hours.

  As most buildings have several differently orientated facades usable for solar system installations, usually facades can be found either orientated towards the rising (i.e. morning) sun or toward the sunset. Consequently, using those areas for the installation of solar cell systems can support available rooftop installations by broadening the peak-power interval.

- **Decreased sensitivity for dirt**
  For the reason of their large exposed area solar cell modules are prone to become soiled either by men-made dust (i.e. by cars or nearby industry) or environmental conditions (i.e. tree pollen). This soiling leads to a shadowing effect which decreases the solar systems efficiency. Due to their large installation angle, facade mounted solar systems are less prone to gather significant amounts of dirt and are easily cleaned; i.e. by rain shower. Furthermore, the heavily inclined installation avoids the accumulation of snow in the winter, therefore reducing the mechanical load which have to be taken into account for roof top installations.

Therefore, the usage of

8.2. Simulation process

This section covers the multi-step process (see figure 8.1) from acquiring the geometry of a given building to evaluating the power yield due to irradiation of its facade- and rooftop-areas.

In the first step, geometrical information about a building's facade were collected by means of a mobile laser scanning system which was done by the *Institute of Cartography and Geoinformatics*(IKG) of the *Leibniz University Hannover*. The result of such a scan is a point cloud as shown in figure 8.1(1). Using a sophisticate algorithm [59], the IKG created a *constructive solid geometry* (CSG) model which represents the building's geometry (see figure 8.1(2)).

8.2.1. Simulation and model-triangulation

In the second step, these CSG descriptions are integrated into a DAIDALOS simulation whose structure is shown in figure 8.2. Here, the daylight source (see section 6.5) is used in box-mode to tightly surround the building. An AbsorbEverything-plugin is registered as a face effect (see section 4.6.5) to the outer side of any face of the building's geometry. The only effect of this plugin is to absorb every photon hitting a face, thus avoiding any further interaction of this photon. This implies that reflection of light at walls and windows are neglected. Additionally, a PhotonPositionSaver-plugin is registered to the same FaceEffect stack (see figure 4.16) and stores the position of any incoming photon prior to the absorption by the AbsorbEverything-plugin. As result of the simulation a table is produced which contains the hitpoint of any photon (i.e. a point cloud shown in figure 8.1(3a)).

Unfortunately, using CSG, large areas of the building are represented by single faces of the underlying CSG geometry. Consequently, calculating the power yield based upon those faces would effectively lead to averaging over the faces area, blurring the optical impact of local structures of the building. For this reason, the CSG geometry is triangulated using the software *Blender*[1] to create a mesh of sufficiently small triangles (see figure

[1]One may argue, that *Blender*, as used for graphical ray tracing, is not the most useful software to

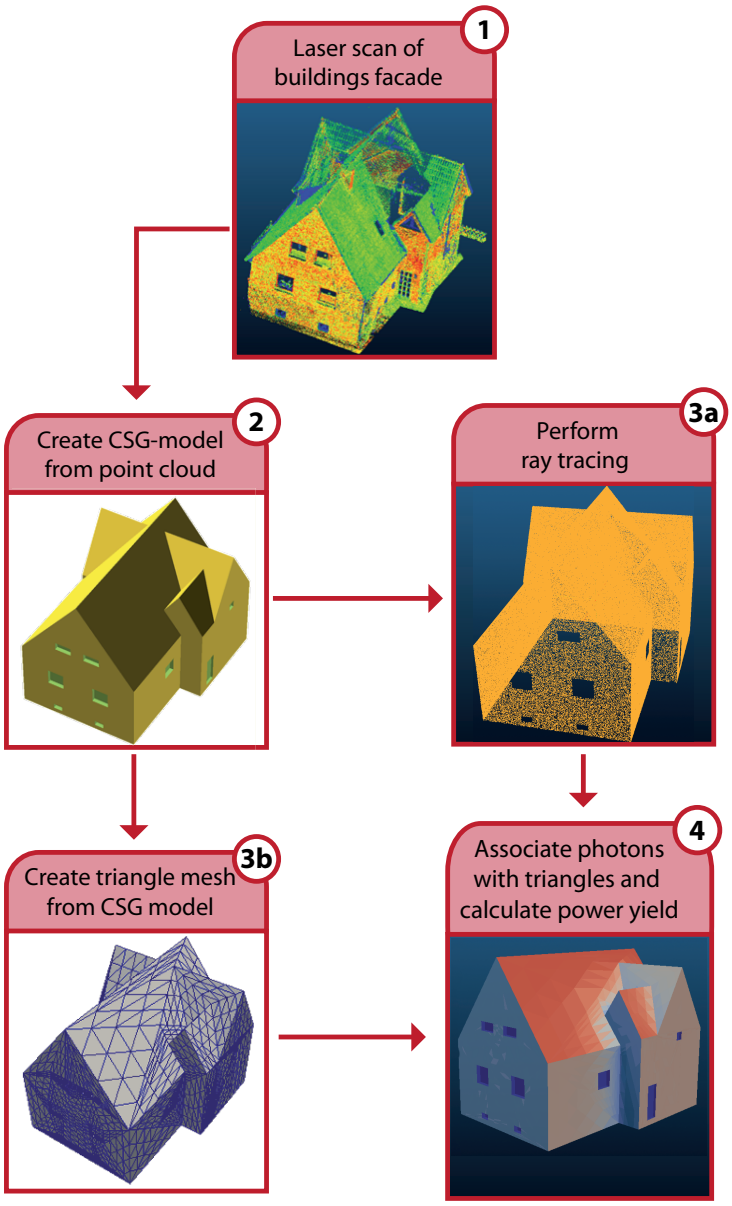


Figure 8.1.: The facade simulation process consists of several steps. First, the building is scanned with a laser resulting in a point cloud which represents the building's facade (1). This is used to create a CSG model of the facade (2) on which a DAIDALOS based ray tracing simulation determines the hitpoints of the simulated photons on the facade (3a). These hitpoints are associated with a triangle mesh created from the CSG model (3b) to finally determine the expected power yield on the facade (4).

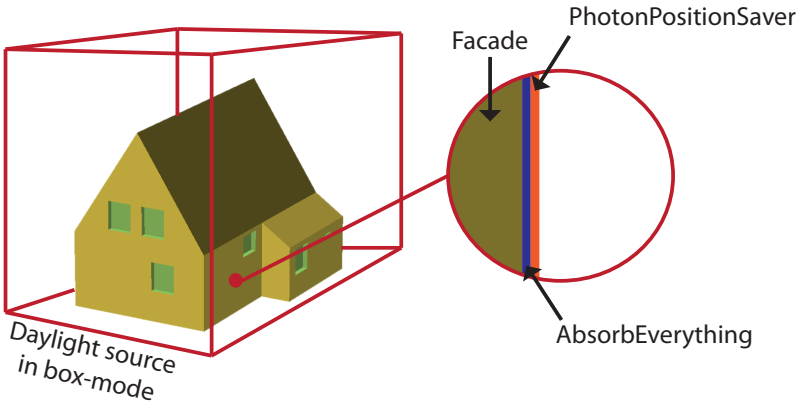


Figure 8.2.: Any facade simulation consists of a CSG model of the building's facade which is tightly embedded into a daylight source used in box-mode. Each face of the building is equipped with a face effect provided by the AbsorbEverything plugin, absorbing all incoming photons. Additionally, each face except of the windows stores the hitpoints of incident photons by means of a PhotonPositionSaver-plugin.

8.1(3b)). The point cloud which resulted from the DAIDALOS ray tracing process is then projected onto the triangle mesh, associating each triangle i with a number of photons N_i received on the triangles area A_i.

8.2.2. Calculation of the power yield

In the final step, the expected power yield has to be calculated for each triangle of the generated mesh. As the number N_i of incident photons is already known for each triangle, this calculation consists of two parts. First, the irradiation power transported by a single simulated photon must is derived. Second, based upon assumptions about the efficiency of the used solar system installation the expected power yield is calculated.

Transported power per photon

If used in box-mode, the daylight source distributes the total irradiated power equally onto all simulated photons. Consequently, to derive the power which is carried by each photon, the overall power on all faces of the box has to be calculated first.

As described in section 6.4, the final weather data associates the horizontal irradiance as measured by the pyranometer with the bins of the underlying bin model. Accordingly,

generate meshes for scientific simulations (i.e. leading to triangles of different sizes). With respect to the facade simulations presented within this chapter, the created meshes haven proven to be sufficient for an estimation of the expected power yield and the optical impact of men-made and natural obstacles. However, more advanced meshing may be useful for future simulations.

knowing the irradiance $I_{\text{hor.}}(\varphi, \theta)$ of any bin with center coordinates (φ, θ), as shown in figure 6.6, the overall irradiated power is computed by the following steps:

1.) The normal irradiance $I_{\text{norm.}}(\varphi, \theta)$ is calculated by:

$$I_{\text{norm.}}(\varphi, \theta) = I_{\text{hor.}}(\varphi, \theta) \cos(\theta). \tag{8.1}$$

2.) This normal irradiance $I_{\text{norm.}}(\varphi, \theta)$ is distributed onto the side faces of the box. Usually, the box orientation will be aligned with the axes of the coordinate system, leading to the projection:

$$\begin{aligned}
I_{+x}(\varphi, \theta) &= I_{\text{norm.}}(\varphi, \theta) \sin(\theta) \max\left[-\sin(\varphi), 0\right], && (8.2)\\
I_{-x}(\varphi, \theta) &= I_{\text{norm.}}(\varphi, \theta) \sin(\theta) \max\left[\sin(\varphi), 0\right], && (8.3)\\
I_{+y}(\varphi, \theta) &= I_{\text{norm.}}(\varphi, \theta) \sin(\theta) \max\left[\cos(\varphi), 0\right], && (8.4)\\
I_{-y}(\varphi, \theta) &= I_{\text{norm.}}(\varphi, \theta) \sin(\theta) \max\left[-\cos(\varphi), 0\right], && (8.5)\\
I_{+z}(\varphi, \theta) &= I_{\text{hor.}}(\varphi, \theta), && (8.6)\\
I_{-z}(\varphi, \theta) &= 0. && (8.7)
\end{aligned}$$

Here, I_{+x} is the irradiance coming from the box-face located in direction of the positive x-axis. The values of I_{-x}, I_{+y}, I_{-y}, I_{+z} and I_{-z} are representing the other sides of the box. The remaining non-directional fraction of the irradiation is associated with the top-face of the box.

3.) As the irradiance measured by the pyranometer were derived for an area of $1\,\text{m}^2$, the results of the previous step have to be scaled with the actual dimensions of the box which were configured by the user. The result of this step is the power P_i (in $\text{kW}\,\text{h}$) irradiated by any of the i box-sides.

Performing these steps for any bin of the hemisphere and summing over the power irradiated by all box-sides, the total amount $P_{\text{box.,total}}$ as emitted by the box-mode daylight source is derived. Consequently, using a number N of simulated photons the average power $\overline{P}_{\text{phot.}}$ carried by a single photon is given by:

$$\overline{P}_{\text{phot.}} = \frac{P_{\text{box.,total}}}{N}. \tag{8.8}$$

Power yield per triangle

Based upon the average power per photon (see Equation 8.8), the irradiance received on a triangle i of the mesh is calculated by:

$$I_{\text{triangle},i} = \frac{N_i \cdot \overline{P}_{\text{phot.}}}{A_i}. \tag{8.9}$$

Here, N_i are the number of photons associated with the triangle and A_i is its area in m^2. In order to deduce a power yield based upon these irradiance values, an assumption about

the installed solar system has to be made. Therefore, in the remaining part of this chapter a solar cell module efficiency of $\sigma = 0.16$ is assumed to account for any losses due to conversion of sun light to electrical current. Additionally, an overall system performance ratio of $r_p = 0.85$ is used to consider for system dependent losses (e.g. efficiency of the used inverted rectifier). Based on these assumptions, the yearly power yield of a triangle mesh element is calculated by:

$$E_i \left(\frac{\mathrm{kWh}}{\mathrm{m}^2} \right) = \sigma \cdot r_p \cdot I_{\mathrm{triangle},i}. \tag{8.10}$$

In the last step, the triangle area is colored based upon the associated power yield and represents the final result of the building simulation process (see figure 8.1(4)).

8.3. Buildings in the urban hinterland

This section investigates the annual irradiation which can be expected for houses in the urban hinterland. Most buildings in this region are small compared to common city buildings and consists of usually two to three floors. Accordingly, men-made or natural structures like neighbor buildings or trees can lead to significant shadowing of the available facade area. Furthermore, as those buildings are commonly built to meet individual needs (in contrary to companies) these buildings provide a much greater diversity in structural appearance. This often includes several annexes or other building extensions which can lead to additional shadowing.

Two examples of a building in the urban hinterland are shown in figure 8.3. The first one shown in figure 8.3(top) provides a simple facade without much detail or extensions. If any shadowing obstacles, like neighbors or trees, are neglected the expected power yield is simulated as shown in figure 8.4. While the rooftop receives the maximum power yield of 130 kWh/m^2/year to 140 kWh/m^2/year, the facades also receive a significant power of around 100 kWh/m^2/year. An integration over all faces of this building, considering their dimensions, leads to the result that the rooftop has the potential of roughly 54% of the yearly power yield, while the facades account for 41%.

Most buildings found in the urban hinterland are not standing isolated, but are either placed nearby a shadowing neighbor or a have gardens with natural structures like trees. Therefore, the impact of trees and neighboring buildings was considered by two additional simulations as shown in figure 8.5. It can be seen, that such structures may result in severe shadowing of extended parts of the facade which is usually taken into account when planing the installation of photovoltaic system.

Finally, the impact of structural extensions or annexes were considered by simulating the second building, shown at the bottom of figure 8.3. It is obvious from figure 8.6 that such structures are complicating the task of finding usable areas for solar system installations. This is due to two reasons. On the one hand, these structures serve as obstacles for light leading to wide areas of the roof being shadowed. On the other hand, by partitioning the rooftop into several, small faces of possible different slope the installation process gets increasingly difficult (possibly some of the smallest faces cannot be used for

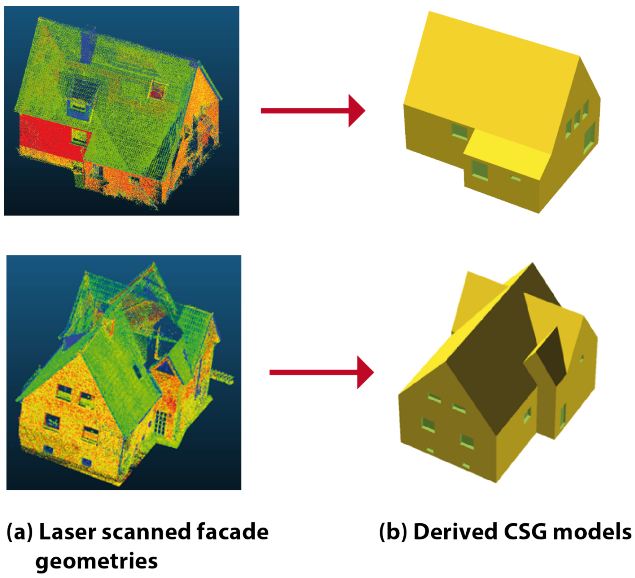


Figure 8.3.: The facades of two buildings in the urban hinterland were scanned by a laser (a). Afterwards, their facade geometries were derived from the laser scans and converted into CSG models (b) which are used in the ray tracing simulations.

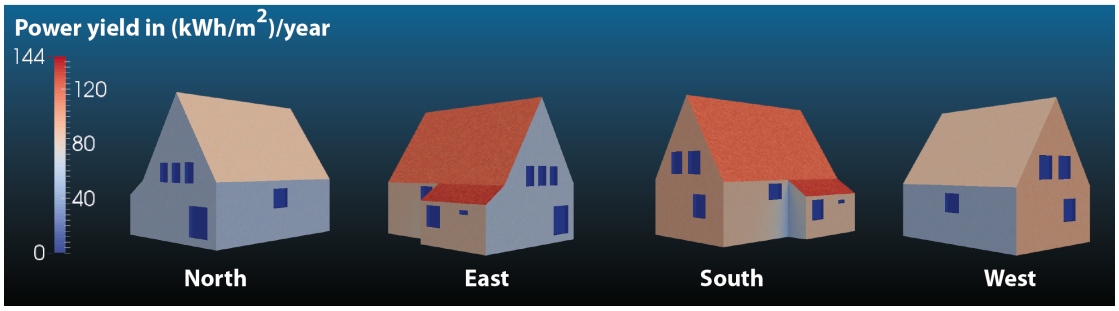


Figure 8.4.: The simulation of the facade of a non-shadowed building in the urban hinterland. The receivable power yield of the facade is around 100 kWh/m^2/year while being 130 kWh/m^2/year to 140 kWh/m^2/year for the roof.

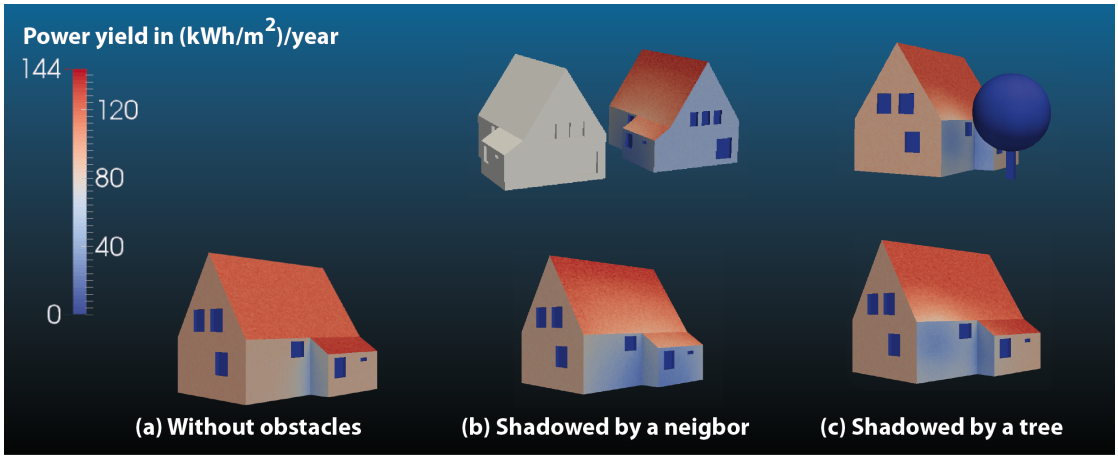


Figure 8.5.: Simulation of a typical building in the urban hinterland. South-east and south-west orientated facades may receive power yields of up to 100 kWh/m^2/year if not shadowed by any obstacles (a). This is significantly reduced when nearby neighbors (b) or natural structures like trees (c) are taken into account.

solar system installation at all). However, as can be seen by comparing figure 8.4 and figure 8.6, there is no impact on the facades which still receive a power yield of about 100 kWh/m^2/year.

8.4. City buildings

Contrary to buildings in the urban hinterland, the common city building consists of several floors and provide large facade areas. For this reason, natural structures like trees have only limited impact on the received overall irradiation. However, due to space considerations, city buildings are often surrounded by nearby neighbor buildings which can lead to severe shadowing of the facades. Additional shadowing may originate from building extensions often found in the form of balconies, especially on the buildings with a facaded orientated towards south.

The laser scan of an exemplary city building is shown in figure 8.7(a). Considering the amount of details, like windows and balconies, this model is the most complex one within this work. Therefore, in order to avoid an unnecessary decrease in simulation performance, subtle details like rooftop extensions are removed when creating the CSG model (see figure 8.7(b)).

Using this model, we considered two different situations. First, the most ideal case is simulated considering the city building to stand far away from any other obstacles. The result of this simulation is shown in figure 8.8, viewing the building from the south-direction. It is no surprise that the roof gets the biggest amount of irradiation, leading to a power yield of around 160 kWh/m^2/year. However, the facades area also

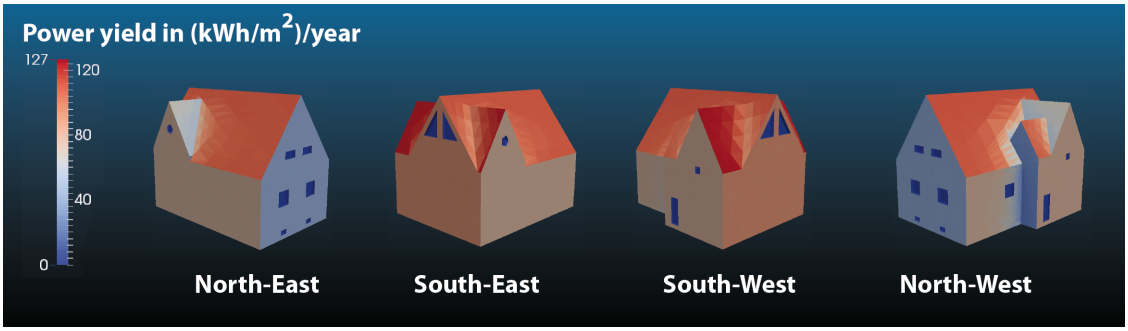


Figure 8.6.: Simulation of an urban building with structural extensions; e.g. annexes. The shadowing resulted from these structures are causing a severe decrease of the roofs power yield.

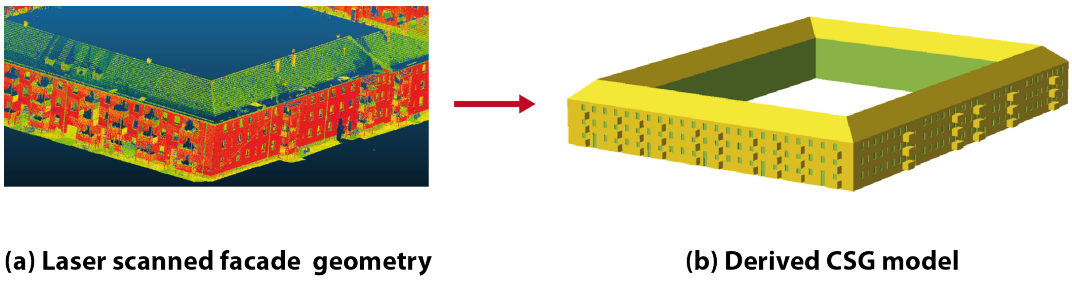


Figure 8.7.: A laser scan of a city building (cutout shown in (a)) is used to generate the CSG model (b) used for simulations. Note, with respect to the laser scan, some subtle details like rooftop extensions were removed in the CSG model to increase simulation performance.

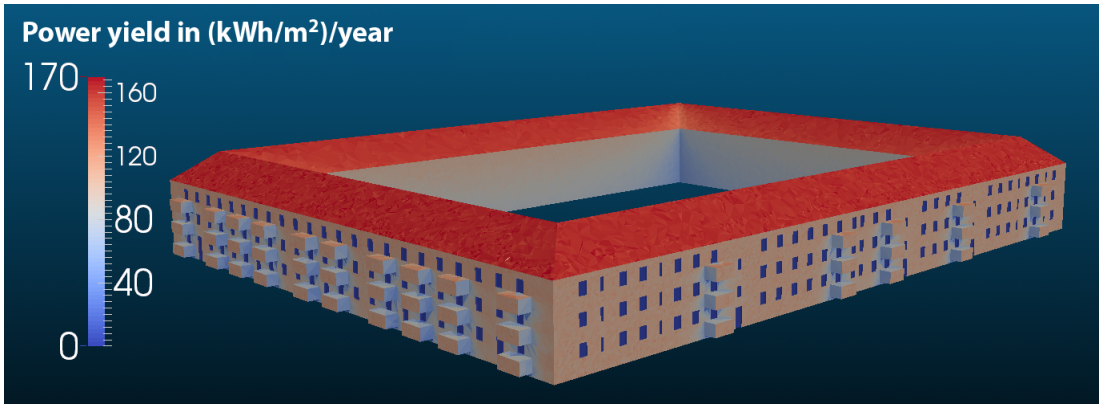


Figure 8.8.: The simulation of a city building which stands far away from any shadowing obstacles.

shows a significant amount of irradiation of around 100 kWh/m^2/year. Even if the severe shadowing in areas near the balconies is taken into account. Considering the large area of the facade, facade-installations seem to be a serious competitor for rooftop systems. Unfortunately, most city buildings are not standing alone but are confined by several neighbor buildings of nearly equal size. Due to space considerations, the streets between are usually narrow leading to even more shadowing by neighbor buildings. Therefore, we simulated a scene shown in figure 8.9 including four neighbor buildings which are surrounding the middle building. As can be seen, the resulting shadowing is severe to a point where the facades of the lower floors are nearly unusable for photovoltaic installations. Nevertheless, the upper floors facade remains usable with power yields of around 80kWh/m^2/year.

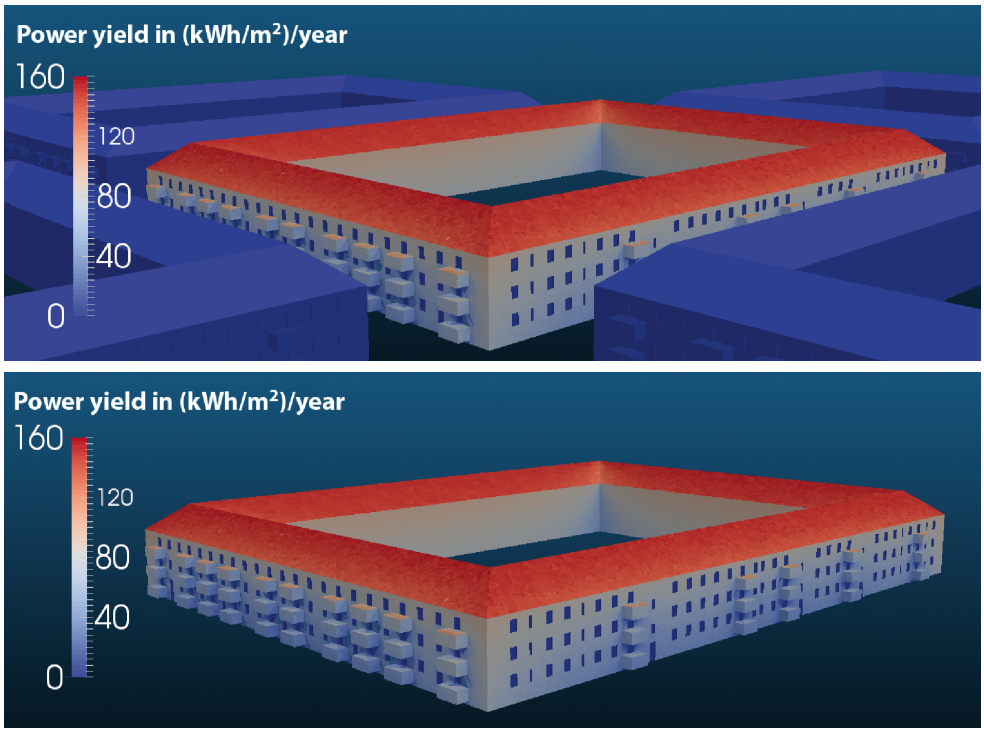


Figure 8.9.: The simulation of a city building including nearby neighbors (top) shows significant impact of shadowing for the facades of the lower floors (bottom). Nevertheless, the most upper floor still receives a power yield of around 80kWh/m^2/year and may be considered for solar system installations.

9 Summary

Within this work we presented the newly developed DAIDALOS ray tracing framework. The DAIDALOS framework follows a modular approach and allows users to change the ray tracing process with self-written plugins. This way ray tracing simulations can include new effects while relying on existing functionality for everything else.

This ray tracing framework was applied to reflectivity measurements on textured wafers using an integrating sphere. It could be shown that the good light-trapping capabilities of such wafers can lead to severe underestimation of the reflectivity in measurements. This is due to light which propagates within the wafer by alternating internal reflections at the wafers front and rear surface. This effect gets even more severe when using an aperture to measure the reflectivity of small wafers. With the simulations this effect could be quantified for the first time.

Using DAIDALOS simulations the reflectivity of textured wafers could be calculated in detail based on measured texture geometries for the first time. The texture structure was derived by laser scanning microscopy (LSM). Differently etched surface textures show deviations in their measured reflectivity of up to 1.4 % in the wavelength range from 400 nm to 1000 nm. The optical simulations conducted with DAIDALOS agree with the measurements.

With DAIDALOS ray tracing simulations on entire solar cell modules become possible. It was shown that the complex propagation of light within the module leads to an optical coupling between spatially separated parts of the module. For this reason, it is not valid to assume that the module's optical characteristics can be derived from a set of separately conducted simulations of small regions of the module geometry. We provided a new approach that uses DAIDALOS capabilities to connect different simulation domains during simulation. The usage of this approach was demonstrated by simulating the impact of the length of the inter-cell gap to the photogenerated current of the module. It was shown that the simulation conducted using DAIDALOS are in good agreement with the laser beam induced current (LBIC) measurement of a 3×3-cell mini-module. It was deduced that the light reflected from the gap is responsible for an additional amount of 6.96 A of photo-generated current or roughly 1 % of the modules total current. The

presented approach is nearly arbitrary expendable and was also used for sophisticated analyses of optical losses in modules [53, 60].

We showed that DAIDALOS can be used to simulate the annual irradiation which incidents on the facades of buildings. For this simulation we used the facades of real building whose facade geometry was laser scanned and modeled by the Institute of Cartography and Geoinformatics (IKG) of the Leibniz University Hannover [59]. Additionally, we presented a newly developed daylight source which was derived from measured weather data and allows the simulation of real daylight by considering its angular and spectral distribution. Using this source we simulated the annual power which irradiates on the facades of buildings in the urban region as well as in the city. Hereby we considered for shading obstacles like facade extensions, neighbor buildings and trees. Simulation times between 5 min to 12 min for each building would allow an efficient evaluation of the solar potential of buildings and their facades even on larger scale.

A

Material parameters of silicon nitride (SiNx) as derived by in-house measurements

The values of silicon nitride with an index of refraction of $n = 2.05$ at a wavelength of 610 nm were derived by Pietro Altermatt from ellipsometric measurements, conducted by Andreas Wolf at the Institute of solar energy research (ISFH) in Hamelin/Germany.

Wavelength in nm λ	Index of refraction n	Extinction coefficient κ
310	2.281	$1.021 \cdot 10^{-01}$
320	2.264	$8.652 \cdot 10^{-02}$
330	2.247	$7.308 \cdot 10^{-02}$
340	2.232	$6.155 \cdot 10^{-02}$
350	2.217	$5.176 \cdot 10^{-02}$
360	2.204	$4.355 \cdot 10^{-02}$
370	2.191	$3.672 \cdot 10^{-02}$
380	2.180	$3.069 \cdot 10^{-02}$
390	2.169	$2.525 \cdot 10^{-02}$
400	2.159	$2.041 \cdot 10^{-02}$
410	2.150	$1.613 \cdot 10^{-02}$
420	2.141	$1.239 \cdot 10^{-02}$
430	2.133	$9.177 \cdot 10^{-03}$
440	2.124	$6.471 \cdot 10^{-03}$
450	2.117	$4.258 \cdot 10^{-03}$
460	2.110	$2.522 \cdot 10^{-03}$
470	2.103	$1.250 \cdot 10^{-03}$
480	2.096	$4.255 \cdot 10^{-04}$
490	2.091	$3.671 \cdot 10^{-05}$
500	2.085	$4.790 \cdot 10^{-10}$
510	2.081	0.000
520	2.077	0.000

Wavelength in nm λ	Index of refraction n	Extinction coefficient κ
530	2.073	0.000
540	2.069	0.000
550	2.066	0.000
560	2.063	0.000
570	2.060	0.000
580	2.058	0.000
590	2.055	0.000
600	2.053	0.000
610	2.051	0.000
620	2.049	0.000
630	2.047	0.000
640	2.045	0.000
650	2.044	0.000
660	2.042	0.000
670	2.040	0.000
680	2.039	0.000
690	2.038	0.000
700	2.036	0.000
710	2.035	0.000
720	2.034	0.000
730	2.033	0.000
740	2.031	0.000
750	2.031	0.000
760	2.030	0.000
770	2.029	0.000
780	2.028	0.000
790	2.027	0.000
800	2.026	0.000
810	2.025	0.000
820	2.024	0.000
830	2.024	0.000
840	2.023	0.000
850	2.022	0.000
860	2.022	0.000
870	2.021	0.000
880	2.020	0.000
890	2.020	0.000
900	2.019	0.000
910	2.019	0.000
920	2.018	0.000
930	2.018	0.000
940	2.017	0.000

Wavelength in nm λ	Index of refraction n	Extinction coefficient κ
950	2.017	0.000
960	2.016	0.000
970	2.016	0.000
980	2.015	0.000
990	2.015	0.000
1 000	2.015	0.000
1 010	2.014	0.000
1 020	2.014	0.000
1 030	2.013	0.000
1 040	2.013	0.000
1 050	2.013	0.000
1 060	2.012	0.000
1 070	2.012	0.000
1 080	2.012	0.000
1 090	2.011	0.000
1 100	2.011	0.000
1 110	2.011	0.000
1 120	2.011	0.000
1 130	2.010	0.000
1 140	2.010	0.000
1 150	2.010	0.000
1 160	2.010	0.000
1 170	2.009	0.000
1 180	2.009	0.000
1 190	2.009	0.000
1 200	2.009	0.000
1 400	2.009	0.000

Bibliography

[1] J.E. Cotter. Raysim 6.0: a free geometrical ray tracing program for silicon solar cells. In *Photovoltaic Specialists Conference, 2005. Conference Record of the Thirty-first IEEE*, pages 1165–1168, Jan 2005.

[2] A.W. Smith, A. Rohatgi, and S.C. Neel. Texture: a ray tracing program for the photovoltaic community. In *Photovoltaic Specialists Conference, 1990., Conference Record of the Twenty First IEEE*, pages 426–431 vol.1, May 1990.

[3] R Brendel. Sunrays: a versatile ray tracing program for the photovoltaic community. In *Proceedings of the 12th European Photovoltaic Solar Energy Conference*, volume 1994, 1994.

[4] Richard S. Hall, Karls Pauls, Stuart McCulloch, and David Savage. *OSGi in Action: Creating modular applications in Java*. Manning Publications Co., 2011.

[5] Georg A. Reider. *Photonik*. Springer Wien New York, 2005.

[6] Max Born and Emil Wolf. *Priciples of Optics*, volume 7th. Cambridge University Press, 2005.

[7] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Longman, 1993.

[8] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1998.

[9] Timothy Budd. *An Introduction to Object-Oriented Programming*. Addison-Wesley Educational Publishers Inc, 2001.

[10] Michael Blaha and James Rumbaugh. *Object-Oriented Modeling and Design with UML*. Prentice Hall International, 2004.

[11] Bruce Eckel. *Thinking in Java: The definitive introduction to object-oriented programming in the language of the world wide web.* Prentice Hall, 2006.

[12] Karl S. Kunz and Raymond J. Luebbers. *The finite difference time domain method for electrodynamics.* Crc Pr Inc, 1993.

[13] Kane Yee. Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media. *Antennas and Propagation, IEEE Transactions on*, 14(3):302–307, May 1966.

[14] Allen Taflove and Susan C. Hagness. *Computational Electrodynamics: The finite-difference time-domain method.* Artech House Inc, 2005.

[15] Umran S. Inan and Robert A. Marshal. *Numerical Electromagnetics: The FDTD Method.* Cambridge University Press, 2011.

[16] Niels Sadrje Ottonsen and Hans Petersson. *Introduction to the finite element methods.* Prentice Hall, 1992.

[17] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals.* Butterworth-Heinemann, 2005.

[18] Nicholas Metropolis and S. Ulam. The monte carlo method. *Journal of the American Statistical Association*, 44(247):335–341, September 1949.

[19] Andrew Glassner. *An Introduction to Ray Tracing.* Morgan Kaufmann Publishers, 1989.

[20] Kevin Suffern. *Ray Tracing from the Ground Up.* A K Peters, 2007.

[21] Craig Walls. *Modular Java.* Pragmatic Programmers, 2009.

[22] Alexandre de Castro Alves. *Osgi in depth.* Manning, 2011.

[23] Kirk Knoernschild. *Java Application Architecture: Modularity Patterns with Examples Using OSGi: A Roadmap for Enterprise Development.* Addison-Wesley, 2012.

[24] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification, Java SE 8 Edition.* Addison Wesley, 2014.

[25] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition.* Prentice Hall, 2013.

[26] Sherif Ghali. Constructive solid geometry. In *Introduction to Geometric Computing*, pages 277–283. Springer London, 2008.

[27] Ronald R. Willey. *Field Guide to Optical Thin Films.* SPIE - The International Society for Optical Engineering, 2006.

[28] R. Brendel. Simple prism pyramids: a new light trapping texture for silicon solar cells. In *Photovoltaic Specialists Conference, 1993., Conference Record of the Twenty Third IEEE*, pages 252–255, May 1993.

[29] R. Brendel. Coupling of light into mechanically textured silicon solar cells: A ray tracing study. *Progress in Photovoltaics: Research and Applications*, 3(1):25–38, 1995.

[30] R. Brendel and D. Scholten. Modeling light trapping and electronic transport of waffle-shaped crystalline thin-film si solar cells. *Applied Physics A*, 69(2):201–213, 1999.

[31] A.W. Smith and A. Rohatgi. Ray tracing analysis of the inverted pyramid texturing geometry for high efficiency silicon solar cells. *Solar Energy Materials and Solar Cells*, 29(1):37 – 49, 1993.

[32] Toshiki Yagi, Yukiharu Uraoka, and Takashi Fuyuki. Ray-trace simulation of light trapping in silicon solar cell with texture structures. *Solar Energy Materials and Solar Cells*, 90(16):2647 – 2656, 2006.

[33] Mirko Loehmann and Eckard Wefringhaus. Microscopic parameters to describe homogeneity of alkaline texture on si-wafers. *Energy Procedia*, 38(0):849 – 854, 2013. Proceedings of the 3rd International Conference on Crystalline Silicon Photovoltaics (SiliconPV 2013).

[34] Christian Gueymard. Smarts2, a simple model of the atmospheric radiative transfer of sunshine: Algorithms and performance assessment. Technical report, Florida Solar Energy Center, 1995.

[35] Christian A. Gueymard. Interdisciplinary applications of a versatile spectral solar irradiance model: A review. *Energy*, 30(9):1551 – 1576, 2005. Measurement and Modelling of Solar Radiation and Daylight- Challenges for the 21st Century.

[36] Fritz Kasten and Andrew T. Young. Revised optical air mass tables and approximation formula. *Appl. Opt.*, 28(22):4735–4738, Nov 1989.

[37] Christian Gueymard. An anisotropic solar irradiance model for tilted surfaces and its comparison with selected engineering algorithms. *Solar Energy*, 38(5):367 – 386, 1987.

[38] P. Duffett-Smith and J. Zwart. *Practical Astronomy with your Calculator or Spreadsheet*, volume 4. Cambridge University Press, 2011.

[39] K Kondratyev. Ya.(1969): Radiation in the atmosphere. *Albedo of the underlying surface and clouds. Academic Press, New York-London*, 12:411–452, 1969.

[40] Sigmund Fritz. Illuminance and luminance under overcast skies. *J. Opt. Soc. Am.*, 45(10):820–825, Oct 1955.

[41] M. D. Steven and M. H. Unsworth. The angular distribution and interception of diffuse solar radiation below overcast skies. *Quarterly Journal of the Royal Meteorological Society*, 106(447):57–61, 1980.

[42] Christian Gueymard. Une paramétrisation de la luminance énergétique du ciel clair en fonction de la turbidité. *Atmosphere-Ocean*, 24:1–15, 1986.

[43] Christian Gueymard. Modélisation physique du rayonnement solaire basée sur les observations météorologiques horaires. *Cooll. Int. Météorologie et Énergies Renouvelables*, 03:291–302, 1984.

[44] K. Bullrich. Scattered radiation in the atmisphere and the natural aerosol. *Advanced Geophysics*, 10:99–260, 1964.

[45] J. Carstensen, G. Popkirov, J.Bahr, and H.Foell. Cello: an advanced {LBIC} measurement technique for solar cell local characterization. *Solar Energy Materials and Solar Cells*, 76(4):599 – 611, 2003. Photovoltaics and photoactive materials - properties, technology and applications.

[46] O. Breitenstein, J. P. Rakotoniaina, M. H. Al Rifai, and M. Werner. Shunt types in crystalline silicon solar cells. *Progress in Photovoltaics: Research and Applications*, 12(7):529–538, 2004.

[47] Keith R. McIntosh, Richard M. Swanson, and Jeffrey E. Cotter. A simple ray tracer to compute the optical concentration of photovoltaic modules. *Progress in Photovoltaics: Research and Applications*, 14(2):167–177, 2006.

[48] K.R. McIntosh, J.N. Cotsell, J.S. Cumpston, A.W. Norris, N.E. Powell, and B.M. Ketola. An optical comparison of silicone and eva encapsulants for conventional silicon pv modules: A ray-tracing study. In *Photovoltaic Specialists Conference (PVSC), 2009 34th IEEE*, pages 000544–000549, June 2009.

[49] Jo Gjessing and Erik S. Marstein. Optical performance of solar modules. *Energy Procedia*, 38:348 – 354, 2013. Proceedings of the 3rd International Conference on Crystalline Silicon Photovoltaics (SiliconPV 2013).

[50] G. Lifante, F. Cusso, F. Meseguer, and F. Jaque. Solar concentrators using total internal reflection. *Appl. Opt.*, 22(24):3966–3970, Dec 1983.

[51] Greg Smestad and Patrick Hamill. Concentration of solar radiation by white backed photovoltaic panels. *Appl. Opt.*, 23(23):4394–4402, Dec 1984.

[52] P. Grunow and S. Krauter. Modelling of the encapsulation factors for photovoltaic modules. In *Photovoltaic Energy Conversion, Conference Record of the 2006 IEEE 4th World Conference on*, volume 2, pages 2152–2155, May 2006.

[53] Matthias Winter, Malte R. Vogt, Hendrik Holst, and Pietro P. Altermatt. Combining structures on different length scales in ray tracing: analysis of optical losses in solar cell modules. *Optical and Quantum Electronics*, pages 1–7, 2014.

[54] Martin A. Green. Self-consistent optical parameters of intrinsic silicon at 300 k including temperature coefficients. *Solar Energy Materials and Solar Cells*, 92(11):1305 – 1310, 2008.

[55] J.A. Clarke, J.W. Hand, C.M. Johnstone, N. Kelly, and P.A. Strachan. Photovoltaic-integrated building facades. *Renewable Energy*, 8(1-4):475 – 479, 1996. Special Issue World Renewable Energy Congress Renewable Energy, Energy Efficiency and the Environment.

[56] Stefan Krauter, Rodrigo Guido Araujo, Sandra Schroer, Rolf Hanitsch, Mohammed J Salhi, Clemens Triebel, and Reiner Lemoine. Combined photovoltaic and solar thermal systems for facade integration and building insulation. *Solar Energy*, 67(4-6):239 – 248, 1999.

[57] T.T. Chow, W. He, and J. Ji. An experimental study of facade-integrated photovoltaic/water-heating system. *Applied Thermal Engineering*, 27(1):37 – 45, 2007.

[58] P. Redweik, C. Catita, and M. Brito. Solar energy potential on roofs and facades in an urban landscape. *Solar Energy*, 97(0):332 – 341, 2013.

[59] Hai Huang, Claus Brenner, and Monika Sester. A generative statistical approach to automatic 3d building roof reconstruction from laser scanning data. *{ISPRS} Journal of Photogrammetry and Remote Sensing*, 79(0):29 – 43, 2013.

[60] Malte R. Vogt, Hendrik Holst, Matthias Winter, Shebnem Knoc, A. Ruppenthal, Marc Koentges, Rolf Brendel, and P. P Altermatt. Optical loss analysis of colored pv modules using comprehensive ray tracing. In *Proceedings WCPEC*, volume 6, November 2014.

List of publications

Publications arising from the work in this thesis:

1. H. Holst, P.P. Altermatt and R.Brendel, Daidalos - A plugin based framework for extendable ray tracing, *Proceedings of the 25th EUPVSEC*, 2150 (2010).

2. H. Holst, M. Winter, M.R. Vogt, K. Bothe, M. Köntges, R. Brendel, and P.P. Altermatt, Application of a new ray tracing framework to the analysis of extended regions in Si solar cell modules, *Energy Procedia* **38**, 86 (2013).

3. M. Winter, H. Holst and P.P. Altermatt, Prediction of a double-antireflection coating made solely with SiNx in a single, directional deposition step, *Energy Procedia* **38**, 895 (2013).

4. M.R. Vogt, M. Winter, H. Holst, S. Knoc, A. Ruppenthal, M. Köntges, R. Brendel and P.P. Altermatt, Optical loss analysis of colored PV modules using comprehensive ray tracing, *Proceedings of the 6th WCPEC* **6**, (2014).

Curriculum vitae

Name	Hendrik Holst
Anschrift	Kaiserstraße 49, 31785 Hameln
Geburtsdatum	18. Februar 1982
Nationalität	deutsch
Familienstand	ledig

Ausbildung

September 1994 - Juni 2001	Besuch des Georg-Büchner-Gymnasiums in Letter/Seelze Abschluss der allgemeinen Hochschulreife im Juni 2001
Oktober 2002 - Dezember 2008	Universität Hannover Studium der Physik Diplomprüfung im Dezember 2008

Anstellung

seit November 2008	Institut für Solarenergieforschung GmbH (ISFH) Hameln/Emmerthal Wissenschaftlicher Mitarbeiter im Bereich optische Simulationen