

## DAIDALOS – A PLUGIN BASED FRAMEWORK FOR EXTENDABLE RAY TRACING

H. Holst,<sup>1</sup> P.P. Altermatt,<sup>2</sup> and R. Brendel<sup>1,2</sup>

<sup>1</sup>Institute for Solar Energy Research Hamelin (ISFH), Am Ohrberg 1, 31860 Emmerthal, Germany

<sup>2</sup>Institute for Solid State Physics, Leibniz Universität Hannover, Appelstr. 2, 30167 Hannover, Germany

**ABSTRACT:** Available ray tracers are usually written as monolithic applications where the user cannot extend the simulation model. Additionally, ray tracers suitable for solar cells, are usually made for classical optical systems, whereas many effects in solar cells cannot be simulated with such systems: for example, luminescent materials require dynamic light sources, whose intensity depends on the earlier event of absorbing a photon. To overcome these commonly experienced limitations, we have developed a ray tracing framework called DAIDALOS, with the major goal of extensibility. Each element of a simulation (e.g. materials, geometries, surface-effects) is represented by a single plugin. Almost arbitrary effects can be added and modeled by user-written plugins without knowing the source-code of the framework.

**Keywords:** Ray Tracing, Modeling, Optical Properties

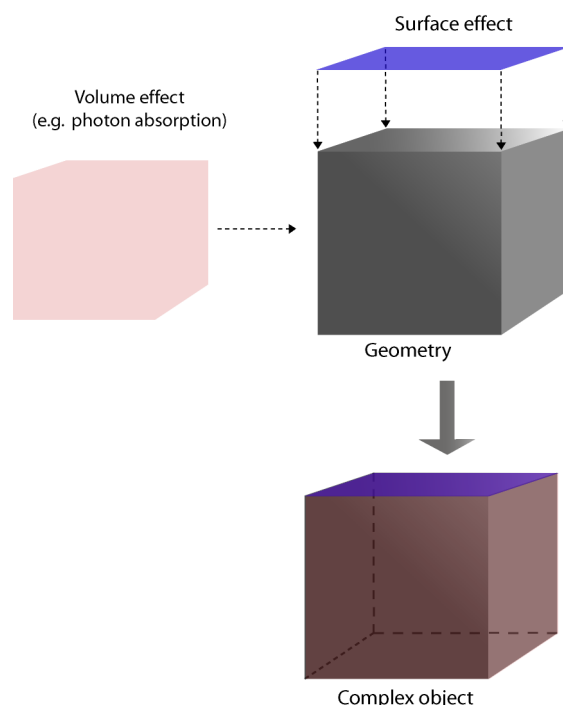
### 1 INTRODUCTION

Ray tracing has been an important part of solar cell modeling. Yet, there is only very few ray tracing software available to specifically simulate the optical properties of solar cells, as for example the freely available software RAYSIM [1]. Ray tracing systems developed in photovoltaic institutions are well suitable, but rarely available outside the particular institution. Most commercially available ray tracers are not designed with solar cells in mind, with the exception of SENTAUROS [2], and it is often difficult to adjust them to solar cells. Additionally, ray tracers suitable for solar cells are usually made for classical optical systems, whereas many effects in solar cells cannot be simulated with such systems: for example, luminescent materials require dynamic light sources, whose intensity depends on the earlier event of absorbing a photon.

To overcome these commonly experienced limitations, we have developed a ray tracing framework that allows the users to extend the model capabilities without knowing or changing the source-code of the framework.

### 2 THE RAY TRACING FRAMEWORK

Common ray tracing software is usually written as monolithic applications. These concepts leave no chance for user-defined extensions to the simulation model. We have adopted a ray tracing framework where each element (e.g. materials, geometries, surface and bulk effects) is represented by a single plugin. This software technology allows the users to write their own plugins and add almost arbitrary effects to the ray tracing model. This is exemplified in Fig. 1: plugins that define a volume or a surface effect (e.g. absorption and scattering) are added to a geometry, which is, in turn, again a plugin. By connecting these plugins, complex objects can be created in a modular way. The definitions of input- and output-data within the tool-chain are accessible through the framework libraries. Therefore, even parts of the tool-chain can be replaced by self-written applications, while staying compatible to the residual tools.



**Figure 1:** Plugins can be combined easily to create complex objects

#### 2.1 The concept of the framework

In the plugin concept, every part of a simulation, e.g. light sources, geometries, materials, etc., is represented by a single independent part of code: the plugin. Each plugin can provide three kinds of connections to its environment: properties, connectors and ports:

1. The plugin-properties are defined for allowing the user to control the plugin's representation in the simulated scene. For example, a plugin which provides a cubical geometry may contain properties like its width, height and length.
2. Connectors and ports are used for defining how each plugin relates to the other plugins. Each connector provided by a plugin represents a type of function, like being a light source or geometry, which is supported by this plugin. On the other

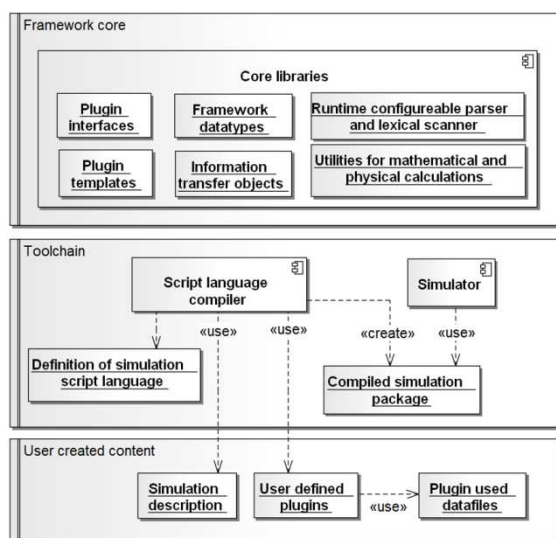
hand, each port used by a plugin requests a function which is needed for proper plugin functionality. For example, a surface effect, like a coating, needs a connection to the associated geometry in order to fulfill its task.

- By creating connections between different plugins, an object can gain additional functionality. For example, a geometry plugin which is connected to a material plugin represents a physical object. Additional connections to surface- and volume-effects can be used for further enhancements as indicated in Fig. 1, e.g. a surface effect representing an anti-reflection coating (ARC) or scattering.

With this concept, a geometric body representing a semiconductor material may the same time be defined as a dynamic light-source to include radiation from radiative recombination. The distribution of excess carriers must be inserted from a device simulator, though, because a ray tracer calculates only the optical, not the electrical properties.

## 2.2 The structure of the framework

The base of the newly developed framework is made up by multiple libraries, the so called core-libraries. They contain definitions of the framework data types as well as classes and interfaces for information exchange between plugins and simulator. For an overview, see Fig. 2. For this reason, each plugin created by a user must include at least one core-library for correct implementation of its interface. We have made efforts in implementing the core-libraries in such a general manner that almost arbitrary effects can be added and modeled.



**Figure 2:** The framework structure consist of three parts: core, tool-chain and user created content

Additionally, several utility classes can be found. For example, a parser as well as a lexical scanner are helpful for reading external input files, which are created with user-friendly input scripts.

Based upon these core-libraries, the framework tool-chain was developed, as indicated in the middle part of Fig. 2. It separates into definitions and applications. The application-part is made up by the compiler- and simulator-application (their usage in the simulation

process will be described in the next section). Although the tool-chain is ready to run, the compiler and simulator applications can be replaced easily if necessary. This possibility is due to the definitions-part of the tool-chain. It consists of definitions of the simulation script language as well as of the structure of a compiled simulation package.

While the core-libraries and tool-chain are important parts of the framework, the biggest part of the ray tracing will be done by user-created content, indicated in the lower part of Fig. 2, like plugins or data files. By keeping the plugin interfaces as small as possible and independent of each other, each plugin stays accessible and reusable. For example, as long as a geometry plugin does not explicitly references a surface effect, surface effects can be changed without side effects to the geometry.

All plugins are written in the Java programming language. Therefore, they are nearly independent of the particular operating system (OS). For this reason, it is especially suitable for distributed computing environments, where the computing server is running on a UNIX-like OS, while the user works with a Microsoft OS.

## 2.3 The simulation process

To keep the simulation process as flexible as possible, it is split up in three separated steps: simulation description, compilation and simulation.

Each simulation process starts with a description of the scene. During this step, the user defines the used simulation elements, their connections and properties. For this step within the simulation process, the tool-chain offers a simple but powerful script language. By providing a script language instead of a graphical user interface (GUI), the process of defining a simulation stays very flexible.

For example, when running the tracer in a distributed computing environment, users might want to change a simulation on the simulation server instead of transferring it to their local computer. The usage of a script language makes it easy to use Secure Shell (SSH) or other console based applications for executing changes. Furthermore, using a script language approach instead of binary simulation files makes it easy to extend the tool-chain by new tools – last but not least by a GUI application for the scene description!

The second step within the simulation process is the compilation of the created simulation description. During the compilation process, the compiler parses the description and determines the used plugins as well as their properties and connections. Based on the gathered information, the compiler generates a compiled simulation package which contains the used plugins as well as a simplified simulation description for the simulator.

By separating the compilation of a description from the actual process of simulation, it is possible to introduce extended process steps. For example, a compiled simulation package can be easily transferred to other computers in order to perform distributed computations.

As a final step within the simulation process, the simulator application is executed on the compiled simulation package. During simulation, the plugins are loaded into memory and the inter-plugin connections are created. After generating the requested simulation scene, the actual ray tracing process is started. By default, a

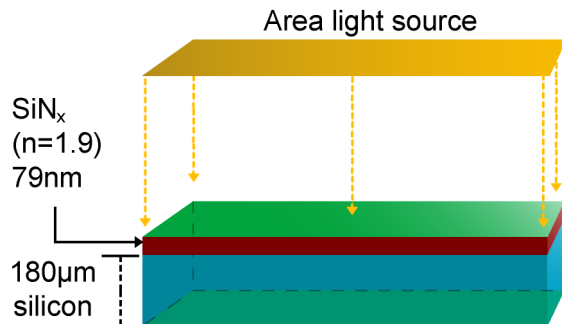
Monte Carlo approach is chosen. However, the core libraries enable the user to create plugins for reverse ray tracing, where rays are started from a detector (e.g. an “eye”), which can be used to visually control the constructed geometries in the scene.

An unlimited number of geometrical objects can be defined, because the tracer is non-sequential, meaning that the ray is able to determine in which body it is momentarily situated. While we constructed the core libraries, various questions emerged: who knows in which body the ray is momentarily situated? The photon, the geometrical body, or an external “authority”? Who knows the photon’s polarization state (if needed)? Who exercises a termination criterion for each ray? Such questions brought us to rather fundamental properties of quantum mechanics, and we always found it most elegant for programming to follow nature’s way of doing things.

The information gathered during tracing is determined by the user in the simulation description, because gathering all the data would very soon exceed the available RAM. A user-defined set of calculated results can be stored in the simulation package and can either be evaluated manually or by user created post-process plugins.

### 3 COMPARISON TO AN ESTABLISHED TRACER

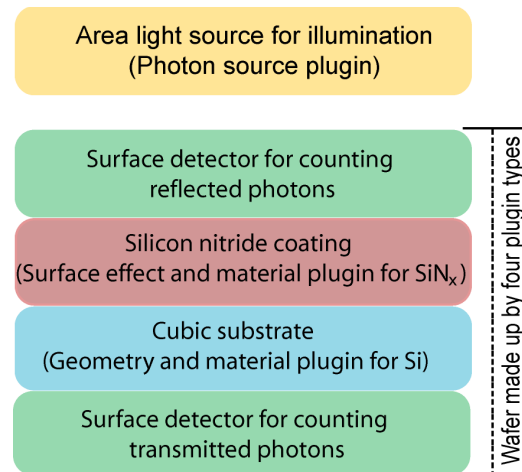
In order to give a proof-of-concept and to verify the results created by the simulator, a simple scene shown in Fig. 4 is simulated. It consists of a wafer with a thickness of 180  $\mu\text{m}$ . The wafer has a 79 nm thin silicon nitride coating on its illuminated side with refractive index  $n = 1.9$ . An area-shaped light source is used for perpendicular illumination with unpolarized light. This scene is built up using five kinds of plugins as indicated in Fig. 5.



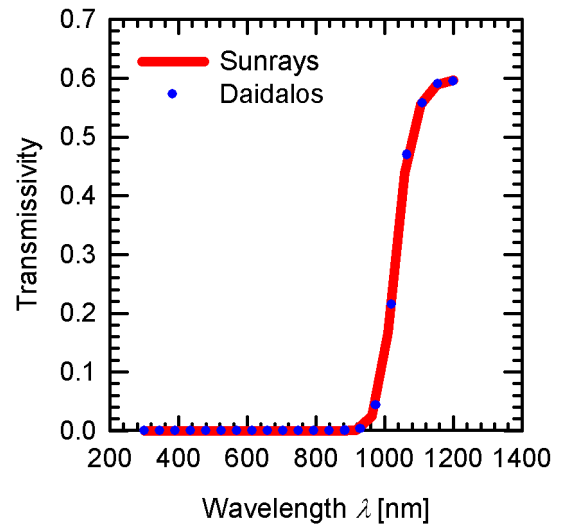
**Figure 4:** Scene used for comparison. The used area light source emits unpolarized photons. Each photon hits the wafer in perpendicular direction. See Fig. 5 for further description of colors.

As reference, we choose the well-known ray tracer SUNRAYS [3]. Due to the fact that SUNRAYS was exclusively developed for the simulation of solar cells, it is an especially well suited reference.

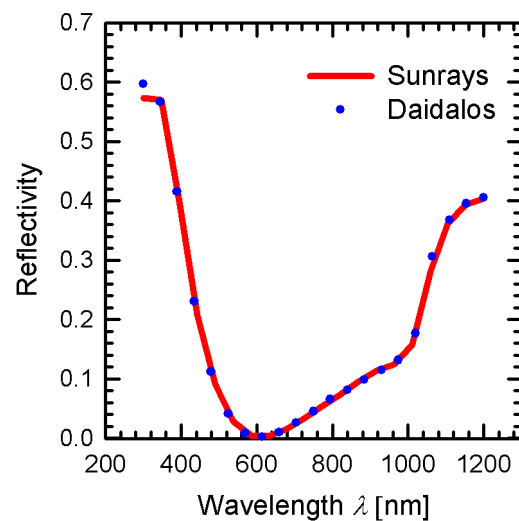
Figs. 6 and 7 show that an excellent agreement is found between DAIDALOS and SUNRAYS.



**Figure 5:** Five kinds of plugin are used for creating the desired scene.



**Figure 6:** Comparison of simulated transmissivity



**Figure 7:** Comparison of simulated reflectivity

#### 4 CONCLUSION

A plugin-based framework for extendable ray tracing is presented. It satisfies the special needs of solar cell development by a highly flexible implementation. Beside the provided basic environment, nearly arbitrary extensions can be added by writing new plugins.

The proof of concept is shown by comparison with the established ray tracing software SUNRAYS. An excellent agreement has been found.

- 
- [1] J. Cotter, 31st IEEE PV Specialists Conference 2005, p. 1165
  - [2] Sentaurus, Manual Version 2009.06, Synopsys Inc., Mountain View, CA, 2009.
  - [3] R. Brendel, 12th EU PV Solar Energy Conf. 1994, p. 1339.