Informatics

Simulating Windows for Lighting Design Optimization

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Moritz Meier
Matrikelnummer 11908106

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: David Hahn, PhD
Dipl.-Ing. Lukas Lipp

Wien, 20. Dezember 2024

Moritz Meier	Michael Wimmer

Informatics

Simulating Windows for Lighting Design Optimization

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Medical Informatics

by

Moritz Meier
Registration Number 11908106

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: David Hahn, PhD
Dipl.-Ing. Lukas Lipp

Vienna, December 20, 2024

Moritz Meier   Michael Wimmer

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Moritz Meier

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 20. Dezember 2024

Moritz Meier

Danksagung

Zuallererst möchte ich mich bei meinem Vater bedanken, da er mir nicht nur durch seine Unterstützung dieses Studium überhaupt erst ermöglicht hat, sondern auch immer hilfsbereit in allen anderen Bereichen des Lebens ist. Außerdem möchte ich mich bei meiner Freundin bedanken, weil sie mich in der gesamten Studienzeit immer bekräftigt und ermutigt hat. Abschließend möchte ich mich bei meinem Betreuer David Hahn bedanken, da die Zusammenarbeit mit ihm immer sehr angenehm war, und sein konstruktives Feedback und seine Ideen immer sehr hilfreich waren.

Acknowledgements

First of all I would like to thank my father, because not only is it his support that made this course of study possible in the first place, but he is also supportive in all other areas of my life. Additionally I also want to thank my girlfriend, who always reaffirmed and encouraged me throughout my study time. Lastly I want to thank my supervisor David Hahn, as working together was always very pleasant, and the constructive feedback and ideas he provided were always very helpful.

Kurzfassung

Bei der Optimierung des Beleuchtungsdesigns spielen besonders für Büro- und Wohnräume Fenster eine große Rolle, da natürliches Licht wichtig für Produktivität, Fokus und die Stimmung ist. Das Rendering Framework Tamashii, das zur Zeit an der Forschungseinheit für Computer Grafik der TU Wien in Entwicklung ist, bietet eine Funktion automatisch verschiedene Parameter von Lichtquellen an Hand eines Optimierungsziels zu optimieren, beispielsweise die Position, die Intensität oder die Rotation. Das Ziel dieser Arbeit ist die Möglichkeiten die Tamashii zum Beleuchtungsdesign bietet durch die Simulation von Fenstern durch Flächenleuchten zu erweitern. Tamashiis automatische Beleuchtungsoptimierung nutzt Light Tracing, was im Gegensatz zu Path Tracing die Lichtstrahlen von den Lichtquellen anstatt der Kamera verfolgt. Eine Implementierung von Environment Maps im klassischen Sinn wäre daher nicht sinnvoll, denn sonst müssten für jedes Pixel der Environment Map Lichtstrahlen emittiert werden, obwohl davon nur wenige überhaupt das Fenster erreichen würden, was sehr ineffizient wäre.

Wir implementieren einen neuen Lichttyp der Flächenleuchten mit IES Leuchten kombiniert um Fenster zu simulieren. Das IES Dateiformat wird häufig von Leuchtmittelherstellern genutzt, um die physikalischen Eigenschaften von Leuchten darzustellen und diese in Software nutzbar zu machen. Damit unsere neuen Leuchten das Licht das durch Fenster scheint genau simulieren können, wandeln wir HDR Dateien in IES Profile um. Unser neuer Lichttyp kann außerdem an anderen Objekten in der Szene wie Wänden oder Decken befestigt werden, wodurch die manuelle Bewegung auf das verbundene Objekt beschränkt wird, was die Nutzung intuitiver macht. In unseren Tests stellen wir fest, dass unsere Implementierung im Vergleich zu Blenders Path Tracing Renderer Cycles bei der gleichen Kombination an Szenen und HDR Dateien echte Fenster mit wenigen Einschränkungen realistisch simulieren kann.

Damit der Algorithmus zur Beleuchtungsoptimierung ein Fenster nur innerhalb des Objekts bewegen kann an dem es befestigt ist, implementieren wir eine Nebenbedingung die bei jeder Iteration der Optimierung evaluiert wird. Diese Nebenbedingung wurde umgesetzt indem wir mithilfe einer Straffunktion Strafen berechnen wenn das Fenster an den Rand des verbundenen Objekts bewegt wird, wodurch wir verhindern können dass der Algorithmus das Fenster aus dem Objekt hinaus bewegt. Bei der Evaluierung unserer Implementierung stellen wir fest, dass der Algorithmus mithilfe unserer Nebenbedingung gültige Positionen für die Fenster bei der Optimierung findet.

Abstract

For lighting design, optimizing windows plays a major role, especially for office and living spaces, as natural light is important for focus, productivity and also mood. The rendering framework Tamashii, which is currently in development at the research unit of computer graphics at TU Wien, offers a feature to automatically optimize multiple parameters of light sources like the position, intensity or rotation for a predefined lighting target. This thesis aims to expand the possibilities Tamashii offers for lighting design by simulating windows through area lights. Tamashii's automatic light parameter optimization relies on light tracing, which unlike path tracing, casts the light rays from the light sources instead of the camera. This is why implementing environment maps in a classical sense is not feasible, as emitting light rays from each pixel of the environment map only for a small percentage to go through the window is very inefficient.

We implement a new type of light that combines area lights with Illuminating Engineering Society (IES) lights in order to simulate windows. The IES standard is a file format commonly used by luminaire manufacturers to describe the physical properties of a luminaire for simulation in software. To accurately mimic the light that shines through real windows, we convert High Dynamic Range (HDR) files into IES profiles, which our lights can then use. Our new light type can also be attached to models in the scene, such as walls or roofs, which constrains the manual movement of the windows to the connected object and makes their usage more intuitive. In our tests, we find that our implementation is able to realistically simulate real windows when compared to the same combination of scenes and HDR files in Blender's path tracing renderer Cycles.

To ensure that the light parameter optimization algorithm only moves the window lights inside the model its connected to, we implement a constraint that gets evaluated repeatedly while optimizing. We realize this by calculating penalties when the light reaches the edges of the model, in order to encourage the algorithm to keep the window light inside. When evaluating our implementation we find that with the activated constraint, the algorithm is able to find valid positions for the window lights when optimizing.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**

2 Background **5**
2.1 Tamashii 5
2.2 Environment Maps 7
2.3 IES Profiles 9

3 Method **13**
3.1 Utilizing environment maps to simulate windows 13
3.2 Constraints of windows in a scene 17
3.3 Enabling light parameter optimization for windows 19

4 Implementation **23**
4.1 Integration of windows 23
4.2 Conversion Algorithm for HDR files 26
4.3 Attaching windows to models 29
4.4 Movement and scaling constraints for windows 30
4.5 Movement constraint for windows in the optimization process 32
4.6 Summary 34

5 Evaluation **35**
5.1 Overall lighting and histogram comparison 35
5.2 Evaluation of lighting optimization modifications 39
5.3 Lighting configurations 44

6 Conclusion and Future Work **47**

Overview of Generative AI Tools Used **57**

List of Figures **59**

List of Tables **61**

List of Algorithms **61**

Glossary **63**

Acronyms **65**

Bibliography **67**

CHAPTER 1

Introduction

Over the last years the scientific rendering framework Tamashii has been in development at the research unit of computer graphics at TU Wien. The main goal of this framework is to enable developers to implement new ideas without the need to reimplement the underlying technologies of a rendering engine. Two of the methods that have been implemented previously are an adjoint light tracing pipeline and lighting design optimization.

The lighting design optimization works by calculating an objective function value that compares the rendered scene with a predefined lighting target. Furthermore, the calculation of gradients provides information about how the scene parameters that need to be optimized impact the objective function value, which is important to find a local minimum. These two concepts play an essential part in differentiable rendering research. Tamashii's adjoint light tracing pipeline was developed because light tracing is particularly effective for optimizing scene lighting. One of the reasons is that rays get cast from the light sources, so every emitted ray contributes to the scene lighting.

With lighting design optimization, it is possible to optimize the position, rotation, intensity and color of multiple lights in a scene, to optimally light a certain target that was defined previously. Currently, Tamashii supports multiple different light sources like point lights, area lights and spotlights, and also importing and exporting glTF files to seamlessly work with other 3D-rendering programs like Blender. However, the rendering framework does not yet support natural light that passes through windows, doors or skylights, which is why the goal of this thesis is to integrate simulated windows to expand Tamashii's possibilities for lighting design.

To simulate daylight in 3D-rendering, environment maps are usually the easiest way to achieve a realistic result, without having the performance impact of having actual models and light sources in the scene. However, with Tamashii's light tracing pipeline, letting an environment map shine through a normal window in the form of a hole in the wall would be very ineffective, because as opposed to path tracing, where rays get traced from

the camera into the scene, light tracing does the opposite and traces the rays from the lights into the scene. This means that for every pixel on the environment map we would need to trace many rays and only a small fraction of them would go through the window, which would make this approach very inefficient.

Due to this, we chose to convert environment maps into IES profiles, which then are used as an area light. IES profiles are a type of standardized lighting format that describe certain parameters like the dimensions, wattage and emission profile of a luminaire, so lights can be accurately simulated in application software. Because the lighting format is very versatile and Tamashii already supports it, we explore the suitability to simulate the light that would shine through a real window with the conversion of environment maps to IES profiles.

Since Tamashii is designed to work as a lighting design tool, manipulating lights and models in a scene is essential for an interactive workflow. Therefore we need to ensure our window lights are intuitive to work with, especially when it comes to placing and moving them. Unlike other light sources, which can theoretically be placed anywhere in a room, windows can only be installed inside of a wall, so in order to have a realistic representation of how windows can be placed in the real world, we need to bring this constraint over into the rendering framework. This is very important for an intuitive workflow, as having to manually move and align window lights to walls would be very tedious and time-consuming. To realize this, we add a functionality to attach window lights to models like walls or ceilings in the scene, which allows us to verify the position and scaling of the window and ensure that it can not leave the bounds of the connected model.

The movement constraints for manual movement also need to be brought over to the automatic optimization process in order to enable the algorithm to find a valid position for the window light if it is connected to a model. Just as for manual movement, the algorithm is only usable in a beneficial way if it is able to find valid positions for window lights, which is why we manipulate the algorithm's gradient calculation to counteract this.

In chapter 2 we explain the theoretical backgrounds of the underlying technologies and data formats that we utilize. First, we clarify the features of the rendering framework and where exactly we expand the framework for this thesis. After that, we present the structure of environment maps and IES files in detail, as they are essential to this thesis.

In chapter 3 we discuss multiple solutions to the challenges of this thesis, and why we chose our particular method of solving these problems. The challenges we tackle in this chapter include the general integration of windows as light sources into the framework, the conversion of HDR images to IES profiles, constraining the movement of the windows and enabling the light parameter optimization for windows.

In chapter 4 we explain the practical implementation of the solutions presented in chapter 3, and discuss the problems that came up in the process and how we solve them. In particular, we describe our implementation that enables the conversion of HDR files

into IES profiles, how we constrain the movement of windows and the algorithm we use to manipulate how the optimization algorithm moves the lights.

At last in chapter 5, we first evaluate the implementation by comparing it to other 3D-rendering programs like Blender, to see if the approach we apply is feasible for simulating windows by environment maps. After that we evaluate how well the light parameter optimization algorithm is able to optimize the position of window lights with our movement constraint implementation.

CHAPTER 2

Background

In this chapter we will explain important concepts and data formats that are important to understand following chapters of this thesis. Among that are the rendering framework Tamashii, different rendering methods, environment maps and IES profiles.

2.1 Tamashii

Tamashii is a rendering framework which is currently in development at the institute of computer graphics at TU Wien [LHEN+24]. It is used as a research platform to easily experiment with new ideas and approaches without having to implement the underlying technologies every time. To provide an engaging workflow, Tamashii is usable through a command line interface and a graphical user interface, which is powered by the Dear ImGUI library [CCng] and allows the user to inspect the scenes in real time.

Tamashii is implemented in C++ for the CPU-side code, while the code that runs on the Graphics Processing Unit (GPU) is written in OpenGL Shading Language (GLSL) and High-Level Shader Language (HLSL), and CMake manages the build process across multiple platforms. Furthermore, Vulkan is used as the graphics Application Programming Interface (API) because it gives developers a lot of low level control, for example synchronization and explicit memory management of the GPU, which is important for high-performance applications. Tamashii also has three implementations for different rendering methods like a rasterization pipeline, a path-tracing pipeline and an adjoint light tracing pipeline, which we explain in detail in 2.1.1. Currently, Tamashii supports multiple forms of light sources like area lights, spotlights and point lights, and users are also able to create lights using IES profiles. Furthermore, it is possible to import scenes that were created in other programs like Blender, as long as the file has the glTF file format.

As mentioned in the chapter above, Tamashii also has a feature to automatically optimize the position, intensity and color of light sources in the scene in a view-independent way [LHEN+24]. The optimization works by predefining one or multiple targets in the scene, for which the optimization algorithms try to optimize the lights in order to optimally light these targets. The underlying technologies to this feature are explained in further detail in subsection 2.1.1.

2.1.1 Rendering Methods

In this subchapter we briefly explain relevant rendering techniques including light tracing, path tracing and differentiable rendering and how they are implemented in the rendering framework Tamashii.

Path tracing

Path tracing is a specific type of rendering algorithm and a form of ray tracing. The algorithm applies the Monte Carlo method, which means it randomly samples as many light paths possible to compute an image [Kaj86]. The algorithm traces the rays from the camera through each pixel and follows their paths through the scene while they bounce off of objects while being either reflected or refracted. Because in practice scenes often contain point lights, which are impossible to hit by a ray, techniques like next event estimation are used to directly determine whether a light source is visible or not. The tracing of rays continues until they either hit their limit of recursive bounces, or their contribution to the final image gets negligible [Vea97]. This recursive tracing of rays enables a path tracing algorithm to simulate global illumination.

Because path tracing is very computationally expensive, it is often not possible to sample enough rays per camera pixel to get a noise free image, which is why there are denoising algorithms that help with getting a smoother image. Despite its computational cost, path tracing is widely used in film production and photorealistic rendering when rendering time is not as big of a concern.

Tamashii implemented a real time path tracing pipeline that is highly customizable via the user interface. It is possible to choose different pixel filters, use tone mapping, change ray settings, enable accumulation of samples and many more settings.

Light tracing

Light tracing is a global illumination algorithm used in computer graphics, which like path tracing is also a form of ray tracing. The major difference is the direction in which the rays are traced, as light tracing starts tracing rays from the light sources instead of the camera, which is why it is also called forward ray tracing [LRS97]. This means that after casting a ray from then light sources, it bounces around the scene until it is either absorbed or hits the camera. Light tracing can simulate global illumination through direct and indirect lighting, which means that object get either illuminated by the light

directly or illuminated by light reflecting or refracting off of other objects. This means that light tracing is suitable to be used for complex lighting effects like caustics.

Tamashii has an implementation of an adjoint light tracing pipeline with different parameters that can be manipulated in the User Interface (UI), like the culling mode, the amount of rays emitted and the number of allowed indirect ray bounces. Unlike traditional rendering, the pipeline not only renders an image, but also calculates the lighting for the whole scene, which provides information that can be used by the automatic light parameter optimization to find a local minimum.

Differentiable rendering

Unlike normal rendering that computes an image from the scene parameters, differentiable rendering is a technique that calculates both the image and the derivatives. This can either be done directly with respect to a specific scene parameter, or more commonly with an objective function that measures how close the image is to a defined target with multiple scene parameters [LADL18]. Through the calculation of the gradients, it is possible to use gradient-based optimization techniques that are often used in machine learning, to optimize certain aspects of the scene.

Tamashii combines differentiable rendering with light tracing, to enable the usage of optimization techniques in a view-independent and camera free way to optimize the position, color, intensity and rotation of luminaires in the scene according to a defined lighting target. With the gradient and an objective function value, the optimization algorithms are able to find local minima for the light sources in the scene. There are multiple optimization algorithms implemented in Tamashii, for instance L-BFGS [Noc80], and ADAM [KB17].

2.2 Environment Maps

Environment maps are one or multiple images used in 3D-graphics to simulate the realistic reflection and illumination of objects or a scene by a surrounding environment, ideally covering all possible viewing directions to ensure accurate lighting. It was first introduced as environment or reflection mapping by James F. Blinn and Martin E. Newell in their scientific paper called “Texture and Reflection in Computer Generated Images” in 1976 [BN76]. The goal of this technique is to simulate reflections on shiny objects like water or metallic surfaces, without having to rely on raytracing, which is still very computationally expensive. After that in 1984, Gene Miller and Robert Hoffman invented an illumination model based on reflection mapping, which builds on the paper of Blinn and Newell [MSH84]. The evolution of this technique was published by Ned Greene in 1986, who proposed cube mapping as an alternate way to represent the environment as six images, which makes reflections more accurate [Gre86]. Today, environment maps are mostly used in video games, movies and virtual reality applications, because they can save processing power by simulating complex reflections and lighting without having to compute the surroundings in real time.

As mentioned above, there are multiple different ways to implement environment mapping [Zim99]. Cube environment maps represent the environment by six independent perspectives that surround the object or scene like a cube. Spherical environment maps on the other hand simulate the environment as a sphere around the scene. A less common way to represent the environment is parabolic mapping, where the environment is split into two perspectives for each hemisphere.

Capturing realistic representations of an environment for the use in 3D-graphics often involve special techniques. With the use of a 360-degree camera, a spherical image of the environment can be taken in a single shot, which can then be processed into either a cube or spherical environment map. Other methods involve capturing images of a mirrored ball, which is also called a light probe, or panoramic photography, where multiple overlapping pictures are used to create a 360-degree panoramic image.

Figure 2.1: Example of an environment map called Rosendal Plains [DS24a].

There are also different approaches to storing the environment maps. For cube environment maps, the data can be stored as independent images for each perspective of the six faces of the cube. For spherical environment maps on the other hand equirectangular projection can be used, where the spherical data is projected onto a rectangle in a 2:1 aspect ratio, which means they map a 360° horizontal and 180° vertical field of view onto the rectangle. An example of an equirectangular representation of an environment map can be seen in figure 2.1. Equirectangular environment maps can have many different resolutions, while most of them are anywhere between 1024×512 to 8192×4096 pixels.

The most popular data formats to store environment maps are HDR formats, because other than Low Dynamic Range (LDR) formats, HDR formats can store a wide range of brightness values, which is especially important for physically based rendering (PBR) pipelines. While there are multiple HDR image formats like OpenEXR or TIFF, this

thesis focuses only on the Radiance RGBE format, which was invented by Greg Ward and published in the book Graphics Gems II [War91]. This format stores the data as 32 bits per pixel, meaning 8 bits for each color channel and one shared exponent, and uses run-length encoding. That means each color channel can have a value from 0 to 255, while the exponent can range anywhere from -128 to 127, which leads to a theoretical max value of 255×2^{127} for each channel. Although such high values would probably never be reached in practice, it allows HDR images to capture very bright light sources which would be impossible in standard LDR images, where the individual color channels only range from 0 to 255.

2.3 IES Profiles

IES profiles are a type of data format which describe certain physical properties and the distribution of light from a light source, and were created and standardized by the Illuminating Engineering Society by the name IES LM-63-1986 so it can be used in different software [Ill19]. These profiles are used to realistically simulate the behavior of real lights, which is why they are utilized especially in architecture or manufacturing but also in video games or 3D-visualizations.

```
1  IESNA:LM-63-1995
2  [TEST]
3  [MANUFAC] BEGA
4  [MORE] Copyright LUMCat V
5  [LUMCAT]
6  [LUMINAIRE] 50975.6K3 (Preliminary)
7  [LAMPCAT] LED 7,9W
8  [LAMP] 321 lm,9 W
9  TILT=NONE
10 1 -1 1.0 73 1 1 2 -0.080 0.000 0.000
11 1.0 1.0 9
12 0.0  2.5  5.0  7.5  10.0  12.5  15.0  17.5  20.0  22.5  25.0
      27.5  30.0 32.5  35.0  37.5  40.0  42.5  45.0  47.5  50.0
      52.5  55.0  57.5  60.0  62.5 65.0  67.5  70.0  72.5  75.0
      77.5  80.0  82.5  85.0  87.5  90.0  ...  180.0
13 0.0
14 330.8  333.2  335.2  336.2  336.8  337.1  337.2  336.7  331.4
      302.1  238.1  159.9  90.7  51.3  38.9  36.0 34.8  34.2  33.5
        32.7  31.6  29.3  25.2  19.8  15.4  12.3  9.9  8.1  6.4
      5.1  3.9  3.0  2.1  1.4  0.9  0.4  0.1  0.1  ...  0.0
```

Listing 2.1: Shortened IES File Data Example [ies].

The data for these profiles is often recorded by photometric testing laboratories where the intensity of the light at different angles is captured [Lab]. Most IES profiles contain various metadata about the light source like the manufacturer, description of the light source and the laboratory and date it was tested. They also contain physical properties of the light like the wattage and lumen values. What they do not contain however is color information in any form, so when using IES profiles for lighting simulation the color information has to be assigned separately. An example of an IES file for a LED light from BEGA, which was shortened and formatted for clarity, can be seen in figure 2.1.

Arguably the most important data in IES files are the emission angles which are split into vertical and horizontal angles and the corresponding light intensity values. The vertical angles represent the angle from the nadir to the zenith and range from 0 to 180 degrees, which means on a hanging light source, 0 degrees corresponds to directly below the light, 90 degrees corresponds to the horizon and 180 degrees corresponds to directly upwards. On the other hand, horizontal angles represent the horizontal plane, ranging from 0 to 360 degrees. A visualization of these angles taken from the official IES specification can be seen in figure 2.2.

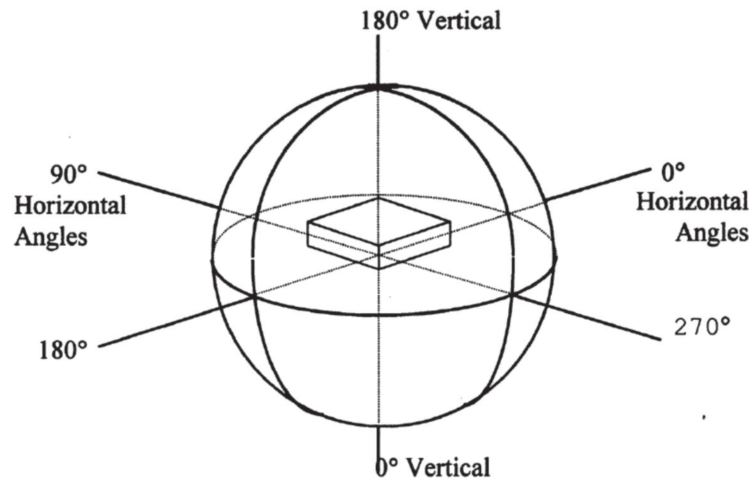


Figure 2.2: Visualization of the angles in an IES file [Ill19].

Typically, the intensity values are arranged line by line for each horizontal angle, so one line contains the intensity values of one horizontal angle in combination with all vertical angles. As an example in figure 2.1, line 12 represents the vertical angles at which the intensity was measured, while line 13 specifies the horizontal angle. This particular example has 0 degrees as it's only horizontal angle, which means that the luminaire is rotationally symmetrical. Below that, in line 14, the file specifies the intensity values of the light source, which are measured in candela (cd).

In figure 2.3, we can see the light distribution of the IES profile shown in listing 2.1.

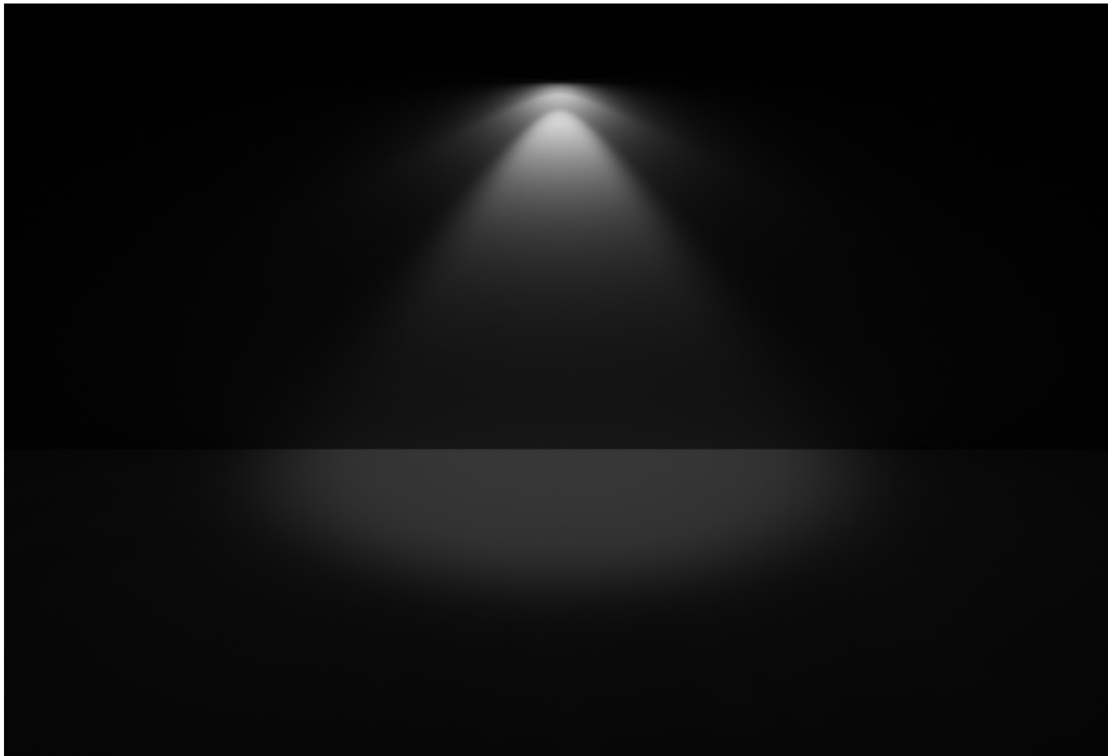


Figure 2.3: Light distribution of the unshortened version of Listing 2.1.

CHAPTER 3

Method

In this chapter multiple solutions for the various challenges we face are discussed. First of all, we need a performant and realistic way to simulate the light that shines through a window. Why we chose environment maps for this and what considerations we need to make to solve this problem is discussed in section 3.1. Furthermore we need to ensure that our implementation of windows offers intuitive controls, since Tamashii aims to work as a lighting design tool. In section 3.2 we explain how certain movement constraints we impose on windows help with the usability of our implementation. At last, we need to enable the light optimization algorithm to work correctly with our implementation of windows, since this is one of the core features of Tamashii. The iterations we go through while solving this challenge will be explained in section 3.3.

3.1 Utilizing environment maps to simulate windows

For the simulation of windows in Tamashii we chose to use environment maps for several reasons. First of all, environment maps offer a realistic and performant way to simulate the surroundings of a scene, which eliminates the need to create a complex environment by hand. Furthermore, since environment maps are just images, they can be quickly swapped to experiment with different lighting configurations. This is important because as a lighting design tool, Tamashii's usability is heavily dependent on good performance and an interactive workflow that allows users to quickly experiment with different lighting setups. Now we will go over two different ways we can utilize the environment maps to implement our windows into the rendering framework, what the upsides and downsides are, and which way we ultimately chose.

The first option for implementing windows is to utilize environment maps in a way multiple 3D-rendering programs such as Blender or 3ds Max have done, so that the environment map is infinitely far away and illuminates everything in the scene. To implement this into both the adjoint light tracing pipeline and the path tracing pipeline, we would

have to majorly rewrite the shader code to allow for global illumination by environment maps. This would also come with a need to implement an option to manipulate the rotation and intensity of an already loaded environment map. As already mentioned in subsection 2.1.1, since light tracing casts rays from the lights into the scene, calculating light rays for a whole environment map only for a small fraction of them making it through windows would be a very inefficient approach without additional techniques such as importance sampling.

Another consideration to make is Tamashii's usability as a lighting design tool. With this approach, we would need to implement a workflow to manipulate geometry directly in the rendering framework in order to quickly add movable windows to walls or ceilings without needing to rely on another program. Right now, manipulating geometry is not possible in the rendering framework as it is not trying to be a 3D-modeling program, so this would also be an enormous amount of work to implement. Furthermore, working with an environment map and a window in the form of a hole in the wall is not as intuitive as working with a single light source.

The second way to implement windows into Tamashii is to convert the environment maps to IES profiles, and to use them with area lights. In other applications, it might not make a lot of sense to implement the usage of IES profiles just to convert HDR files into them, but since Tamashii can already utilize IES profiles, this approach is definitely feasible. The approximation of windows using IES area lights is possible because with IES profiles, we can control the intensity of the light at different emission directions, which is needed to accurately simulate the varying lighting conditions of real windows. This allows us to simulate directional lighting, which mostly happens when the light shines unobstructed from the sun through a window, and diffuse lighting, which happens when the sunlight gets scattered in clouds or fog, and lighting conditions in between. The effect the different emission angles have can be seen in figure 3.1, where the left area light has a spread of 1° and the right area light has a spread of 45°.

For a realistic depiction of a window however, we also need information about the surroundings, which is where we can utilize environment maps. By sampling the environment maps, which offer a realistic representation of the environment, we can get both the intensity values from different directions for the IES profile, as well as color information. Through the combination of IES profiles with environment maps, we can approximate windows by only calculating the light propagation from the window itself, without needing a complex simulation of the environment.

However, this approach comes with some limitations we also need to discuss. First of all, we cannot simulate occlusions that can happen when objects block the light from the outside, like trees or buildings that are near the window, because an environment map does not contain any depth information that would make it possible to calculate occlusion effects. Another simplification is that unlike real windows, where the light distribution varies at different points of the window depending on the surroundings, the light distribution is the same on every point of the simulated windows because we only use one IES profile for the whole light. Theoretically it would be possible to use multiple

Figure 3.1: Two area lights with an emission angle of 1° and 45° respectively.

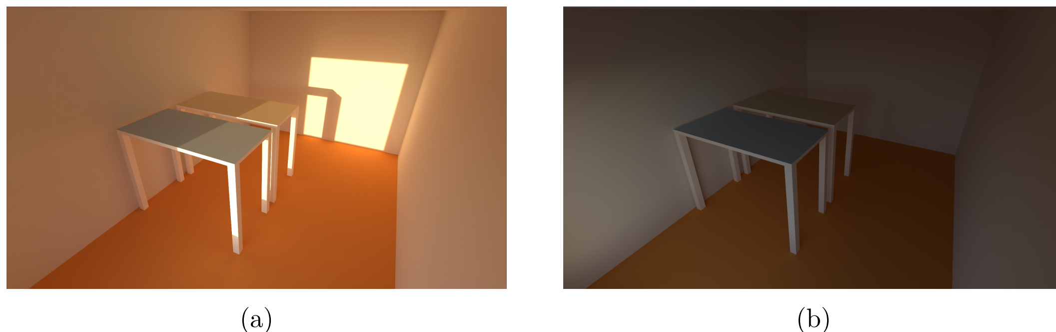
IES profiles for the simulated windows, but the visual difference between one and multiple IES profiles would likely be negligible for a lighting design tool. Furthermore, reflection or refraction effects caused by the light traveling through the glass layers of a window can also not be simulated with our approach.

Despite all these limitations, this approach offers a considerably realistic way to simulate windows, which is also performant enough to be used with the light tracing pipeline and the automatic lighting design optimization.

Ultimately we choose the second approach, which is why there are more considerations to think of on how to accurately convert HDR files, especially because the IES profiles only contain emission angles and intensity values. Since environment maps are images, they do not contain any emission angles, which means we need to either calculate them or let the user configure them in the UI.

Calculating them correctly would be very difficult, because even with a single environment map, it is possible to have two completely different illumination patterns, as seen in figure 3.2. For this reason, we decide to set the emission angles to a baseline value, and let the user decide what kind of lighting configuration they want.

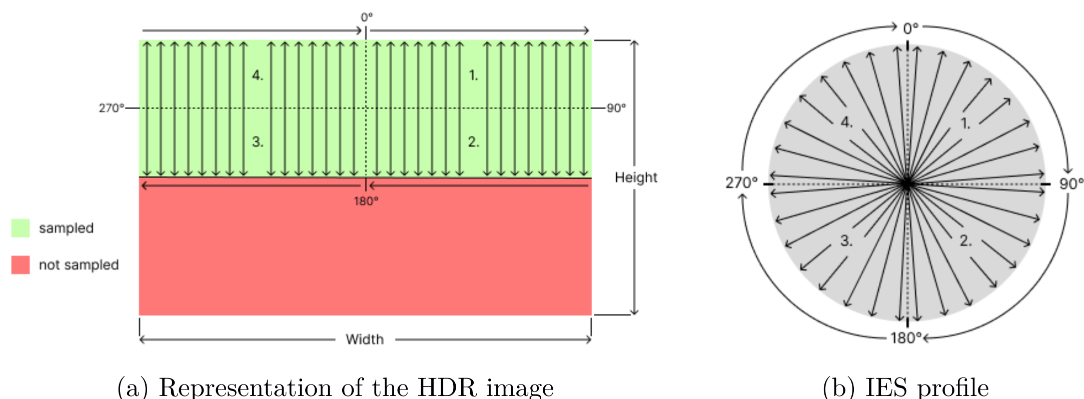
As already mentioned in section 2.2, most environment maps have a resolution between 1024×512 and 8192×4096, and the IES texture we convert the environment map into will have a size of 256×256 pixels. We select a texture size of 256×256 pixels because the resolution is high enough to capture the varying intensity values of the environment map while avoiding long sampling times that would come with a higher texture resolution.



(a) (b)

Figure 3.2: Image (a) shows the small office test scene in Blender with the The Sky Is On Fire environment map [GZnd]. Image (b) shows the same scene, with the same environment map rotated by 100° around the Z axis.

To get the intensity values, we decide to only sample the upper half of the HDR image, because on most HDR images the lower half is the ground and therefore very dark, which would not be realistic because the ground rarely reflects enough light into a window. The sampling itself works in a circular pattern, because we want to convert the hdr file as accurately as possible to the IES profile. We do this by sampling in four quadrants, starting with the first quadrant, then the fourth, after that the third and at last the second. In each quadrant, 64 rows are sampled, with 256 sampling points per row, which results in 256×256 sampling points in total. The rows are always sampled from the center of the vertical sampling range outwards, which ensures, that the center of the sampling area is also in the center on the IES texture. A sketch of the sampling can be seen in figure 3.3, where the arrows indicate how the rows get sampled for each quadrant, and how the sampled rows will be end up on the IES profile. The angle values that are present in both subfigure (a) and (b) refer to the horizontal angles, that were explained earlier in section 2.3.



(a) Representation of the HDR image (b) IES profile

Figure 3.3: This is a sketch on how we convert HDR files to IES profiles.

Because the environment maps have a much higher resolution than the resulting IES

texture, we choose to sample 4 pixels of the environment map for each sampling point, and bilinearly interpolating the 4 pixels of the environment map. Problems that normally arise when applying bilinear interpolation like blurring do not matter here, since we only want to create an IES texture and not a picture, where small pixel-sized details do not matter. After the sampling is done, we need to convert the interpolated pixels into luminance values: $0.2126 * R + 0.7152 * G + 0.0722 * B$, where R, G and B are the values of the individual color channels [Uni15]. The luminance values are then used to create the IES texture, where each value in the texture corresponds to one combination of vertical and horizontal emission angles.

3.2 Constraints of windows in a scene

Certain constraints need to be implemented for window lights to accurately simulate the behavior of real windows. The next subsections will go into detail which constraints need to be imposed and how they can be realized.

3.2.1 Attaching windows to models

Because windows can only exist on walls and sometimes ceilings in the real world, this constraint also needs to be brought over to the window lights in the rendering framework. This means that we need to add a functionality to attach window lights to models when creating a new window light. Considering that there can be many different models in a single scene, we need to restrict the attachment of window lights to models that are not a wall or a ceiling. We can do this by allowing the attachment only for models that are flat, which means one of the model's dimensions is zero. The attachment process also needs to be intuitive for the user, which means the model the window light is attached to should be changeable in the UI without needing to create a new window light.

For the attachment to work, we need to calculate both a new position and direction for the window light, so it is aligned with the model. The position can be just copied over from the model to the window light, but for the direction we need to calculate a rotation based on the initial direction the window light faces and the new direction it should face. The rotation angle can be calculated with the arc cosine of the dot product of both directions: $\theta = \arccos(\mathbf{v}_1 \cdot \mathbf{v}_2)$, where θ is the resulting angle and v_1 and v_2 are the initial direction and the target direction of the window light respectively. The axis we need to rotate the window light around to align it with the new model should be an orthogonal vector so it can be calculated with the cross product: $\mathbf{v}_3 = (\mathbf{v}_1 \times \mathbf{v}_2)$, where v_1 and v_2 are the initial direction and the target direction again and v_3 is the resulting rotation axis.

We also need to pay attention to two edge cases, which happen when the dot product of the two vectors is either 1 or -1, because that means that the initial direction and the target directional are parallel or antiparallel respectively. When the vectors are parallel, we need no rotation at all, so we can just update the position of the window light. When

the vectors are antiparallel on the other hand, we need a new rotation axis because the cross product of two antiparallel vector will result in the zero vector. In this case we can just choose the tangent of the window light, which solves our problem.

3.2.2 Manual transformation constraints

In Tamashii, objects can be manipulated either in world space or in object space by a gizmo. World space is the global coordinate system of the entire scene, where all objects are transformed relative to a fixed origin and coordinate axes. Local space, or also referred to as object space, is the coordinate system of a single object, which can be used for transformations relative to its own origin.

Translations can be done on multiple axes at once, and it is possible to rotate and scale along one of the axes. Furthermore, it is possible to perform transformations on objects by changing the values directly inside the model matrix. The goal of constraining movement of window lights is to not allow the lights to leave the attached model, because as previously stated this also would not be possible in the real world, and would also not be very user-friendly. To achieve this, every translation of the window light needs to be checked whether its bounding box is still inside the connected model's bounding box. If the light's bounding box reaches the edge of its connected model, the translation needs to be reduced by the exact amount the light would be outside the model. Since Tamashii has the possibility to transform objects in local space, we chose to automatically switch to local space translations when selecting a window that is connected to an object and disable the translation axis that aligns with the objects normal vector. This way, the translations can only happen on the two-dimensional plane of the object it is attached to, which makes bounding box checks easier. We have two choices for the bounding box checks, as they can either be done in world space or in object space.

The calculations in world space would be fairly easy if the objects would always be axis aligned because it would be possible to simply compare the coordinates, but since this is not always the case, this cannot be done. Because the window can only move on a two-dimensional plane in the three-dimensional world, it is possible to define the bounding edges of the object the light is connected to and then check if the corner points of the light are still inside. With two edge vectors, any point that lies on the plane can be described as a linear combination of these edges: $P = P_0 + u \cdot \text{Edge}_1 + v \cdot \text{Edge}_2$. Here, P is the corner point of the light we want check whether it is inside the connected object, and Edge_1 and Edge_2 are the two edge vectors originating from the corner point P_0 of the connected object: The equation can be solved for u and v, and if both of the values are between 0 and 1, the corner point lies inside the plane. If the values are either below 0 or above 1 it is outside.

If we do the calculations in object space on the other hand, the problem of constraining the movement gets a little bit more straightforward. If one of the objects gets transformed into the other object's local space, there is no need to account for rotations that are present in world space. This way only the bounding boxes need to be calculated with

the dimensions and position of the objects and checked whether one object is still inside the other by comparing the coordinate values of the window light and the connected object. The transformation needed to transform one object into the other object's local space is done by multiplying one object by the other object's inverse model matrix: $L_{\text{object}} = M_{\text{world}}^{-1} \cdot L_{\text{world}}$. In this example, L_{world} represents the light in world space, M_{world}^{-1} the inverse of the model matrix in world space and L_{object} the transformed light in the model's object space. The downside of this approach is that the constant conversions and transformations between local space and world space can have a performance impact, since this calculation happens every frame while manually moving the window light.

Ultimately we chose to do the bounding box checks in object space, because despite the performance impact, it is easier when we do not need to pay attention to any rotations that are present in world space. Rotation of windows that are attached to a model is prohibited entirely, because rotating the windows without its attached model would not make sense, and could potentially cause problems when done accidentally. It would make sense to implement a functionality to rotate both the light and the connected model at once, but since this is not a priority and does not hinder the other functionalities, we chose to leave this direction for future work. Scaling also needs to be limited to be inside the bounds of the object the light is connected to.

3.3 Enabling light parameter optimization for windows

As briefly mentioned in section 2.1, Tamashii has a feature to automatically optimize multiple parameters of light sources in a scene, including the position, intensity and the color. The optimization algorithm works by calculating an objective function value ϕ that compares the rendered scene with a predefined lighting target. The value ϕ gets calculated on every iteration of the optimization process, and is the main value that determines a good solution. Furthermore, the calculation of gradients help the optimization algorithm by providing information about how the parameters of the lights impact the objective function value. Tamashii also has implementations for constraints, which get evaluated by the optimization algorithm on each iteration, and add a penalty value to the objective function value when the constraint is not fulfilled. An example for this is a constraint that is aimed at hindering the optimization algorithm from making the lights too bright while optimizing, which can be helpful when energy-efficient solutions are needed.

To enable the light optimization algorithm to optimize our window lights in a correct way, the same constraints that apply to manual movement of the window lights should also be brought over for the light parameter optimization. Otherwise, the algorithm would move window light around in the whole scene and break the attachment between a window light and its model.

The most reasonable way to do this is to create an additional constraint that is specific to window lights, that hinders the algorithm from moving the lights away from the model they are attached to. The optimization algorithm uses the gradients to calculate the correct movement in order to find a local minimum, so by adding penalties to the

calculated gradient, the movement can be manipulated. We also want to calculate the gradient penalties in the connected model's object space as we did with the translation check for manual movement in subsection 3.2.2. The advantage of doing the calculation in object space is that we do not have to worry about the rotation of the objects in world space, which makes the calculations of the penalties easier, but it could have a performance impact since these calculations have to be done on every iteration of the optimization algorithm.

We can transform both the gradients of the window light and the window light itself by multiplying them with the inverse of the model's model matrix: $L_{\text{object}} = M_{\text{world}}^{-1} \cdot L_{\text{world}}$. In this equation L_{world} represents the light's model matrix in world space, M_{world}^{-1} the inverse of the model's model matrix and L_{object} the transformed light in the model's object space, but the calculation for the gradients is the same. After these transformations, we need to calculate the bounding boxes for both the window light and the model for the penalty calculation.

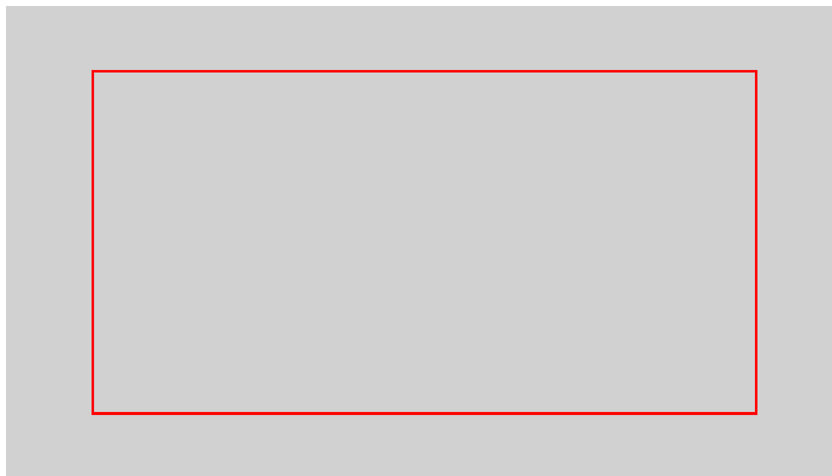


Figure 3.4: A sketch of a wall, where the red lines indicate the threshold from which penalties get added to the gradient.

For the penalty calculation we decide to start adding penalties to the gradients even before the algorithm moves the window light over the edge of the wall, to discourage the algorithm from finding a solution that is very close to the edges, because most windows in real life have a central position in a wall. A sketch of the threshold from which penalties are added can be seen in figure 3.4. The optimization algorithm calculates gradients for parameters independently, which means we can manipulate the movement the algorithm chooses on each axis individually. The gradient for the axis that aligns with the normal vector of the model always has to be zero, as otherwise the attachment between the window light and the model would break. The penalties for the remaining two axes need

to be calculated individually, and the penalty value for each axis represents the exact amount the window light's bounds exceed the threshold of the model. After the penalties are calculated, they get added to the gradient and also to the objective function value ϕ.

If we only manipulated the gradients without the objective function value, the algorithm could still find local minima where the window light would be outside the bounds of the connected model, as the main indicator of a good solution is the objective function value and not the gradients. On the other hand, if we only manipulate the objective function value, the algorithm would break the attachment with the wall as there are no hard constraints implemented that hinder the algorithm from moving the window light away from the connected model. Through the manipulation of both, we can ensure that the optimization algorithm finds valid positions for our window lights.

CHAPTER 4

Implementation

In this chapter we explain the practical implementation of the solutions presented in chapter 3, and discuss the problems that came up in the process and how we solved them. Among that is the integration of the new `WindowLight` class, the implementation used to convert HDR files into IES profiles, how we solved constraining the movement of windows and the implementation used to manipulate how the optimization algorithm moves the lights.

4.1 Integration of windows

Before implementing the solutions we discussed in chapter 3, we first need to implement the `WindowLight` class, which will be used on the Central Processing Unit (CPU) sided part of the rendering framework. On the CPU, the light types all have specialized classes with specific attributes that derive from a superclass which includes attributes that all lights have in common, like the color, intensity and direction. The light superclass can be seen in listing 4.1.

These specialized classes for every light type are necessary for differentiating between lights in the editing UI, the light parameter optimization and the overall background logic of Tamashii. This is why a new type of light on the CPU side is definitely necessary, and also because the window lights need extra attributes for certain functionalities that would not be beneficial to have in an area light, for example connecting the light with an object to constrain its movement.

The new `WindowLight` class can be seen in listing 4.2, where we combined attributes from area lights with attributes from IES lights. The shape and dimensions are important to have a rectangular representation of a window in the rendering framework, and the horizontal and vertical angles together with the light texture enable our window to utilize IES profiles either directly or as a conversion from a HDR file. Furthermore, we added

the mConnectedModel attribute for the attachment of window light to models, which will be explained in the next subsection 4.3.

```
class Light : public Asset {
public:
    enum class Type
    {
       DIRECTIONAL, POINT, SPOT, SURFACE, IES, WINDOW
    };

    // Getter and setter methods removed for compactness

    virtual Light_s                  getRawData() const = 0;
                                     ~Light() override = default;
protected:
                                     Light(Type aLightType, glm::vec3 aDirection, glm
                                         ::vec3 aTangent);

    Type                             mLightType;
    glm::vec3                        mColor;
    float                            mIntensity;
    glm::vec3                        mDirection;
    glm::vec3                        mTangent;
};
```

Listing 4.1: Implementation of the Light superclass other classes derive from.

When lights get moved over to the GPU, every light no matter which type gets packaged into a struct, which is possible because the GPU only needs the data that is important for the rendering. The struct can be seen in in listing 4.3.

```
class WindowLight final : public Light {
public:
    enum class Shape
    {
       SQUARE, RECTANGLE
    };
                                     WindowLight(): Light(Type::WINDOW, { 0,0,-1 }, {
                                         1,0,0 }), mShape(Shape::SQUARE), mDimension
                                         (1) {}
    Light_s                          getRawData() const override;

    // Getter and setter methods removed for compactness

private:
    Shape                            mShape;
    glm::vec3                        mDimension;

    std::vector<float>               mVerticalAngles;
    std::vector<float>               mHorizontalAngles;

    Texture*                         mCandelaTexture;
    std::shared_ptr<Model>           mConnectedModel;
};
```

Listing 4.2: Implementation of the WindowLight class.

Because the `type` attribute still exists, it would be possible to also implement our window light as a new type in the shaders, but that would mean that we would have to implement the light propagation logic from scratch. But since the windows have a lot in common with the already implemented area lights, this approach would lead to lots of redundant code.

```
struct Light_s {
    glm::vec3               color;
    float                   intensity;
    glm::vec4               pos_ws;
    glm::vec4               n_ws_norm;
    glm::vec4               t_ws_norm;
    glm::vec3               dimensions;
    uint32_t                double_sided;
    float                   range;
    float                   light_offset;
    float                   inner_angle;
    float                   outer_angle;
    float                   light_angle_scale;
    float                   light_angle_offset;
    float                   min_vertical_angle;
    float                   max_vertical_angle;
    float                   min_horizontal_angle;
    float                   max_horizontal_angle;
    int                     texture_index;
    uint32_t                triangle_count;
    int                     id;
    uint32_t                index_buffer_offset;
    uint32_t                vertex_buffer_offset;
    uint32_t                type;
};
```

Listing 4.3: Implementation of the struct that is used for lights in the shaders.

Because of that we choose to treat the window lights as area lights on the GPU, which means we need to change the shader code for area lights to allow for IES texture usage. This also means that we can now simulate normal IES lights as area lights, which was not possible before. We adjust the shader code for both path tracing and light tracing, because we want the IES lights to work in both renderers. For light tracing, we have to modify the function that generates light rays, as the rays get traced into the scene from the light. This means, that when a light texture is present for an area light, the IES texture needs to be sampled with the randomized ray direction in order to calculate the correct ray intensity for the emission angle. We also need to scale the ray intensity with the dimensions of the window, since when the window is bigger more light shines through and illuminates the scene.

For path tracing on the other hand, we had to change the way area lights are evaluated when hit with a ray in a similar manner to the light tracing modifications. To realize this we change the function that evaluates lights when hit by a ray to check whether a light texture is present for the area light, and if it is, we sample the texture with the incident ray direction to get the correct intensity value.

For the general usability of the window lights, we also need to implement the descriptions and controls in the Graphical User Interface (GUI) that appear when selecting a window light, to show relevant information about the light and also change values on the fly. Apart from standard values that are present on all light sources, we add the emission angles and the direction of the light so they can be changed easily.

Another consideration we have to make is to expand the importing and exporting of glTF files to also support our new window lights. When saving a scene with a window light, the most important properties like the height, width and the model the light is connected to get saved, and also the HDR file the light was created from. Because our window lights are custom, other programs like Blender will ignore them when importing the glTF file, but when loading the scene in Tamashii all properties of the light are set correctly.

4.2 Conversion Algorithm for HDR files

As mentioned in section 3.1, to use HDR files in the rendering framework we choose to convert the files to IES profiles. The conversion of HDR image files solely happens inside Tamashii, either when loading a scene or when creating a new window light, where it is possible to choose a file through a file dialog. Converting the file and then saving it into an IES file is not the goal of this conversion, so when saving a scene, there will still be only the original HDR file present, which gets converted again when the scene is loaded.

```
1  void generateAnglesSampling (float aHorizStartAngle, float aHorizEndAngle, float
       aVertStartAngle, float aVertEndAngle, int aHorizSize, int aVertSize, std::vector<
       float>& aHorizAngles, std::vector<float>& aVertAngles) {
2      float horizMiddle = (aHorizEndAngle + aHorizStartAngle) / 2.0f;
3      float vertMiddle = (aVertEndAngle + aVertStartAngle) / 2.0f;
4
5      // Create 256 horizontal angles
6      for (int i = 0; i < aHorizSize/4; ++i) {
7          float t = static_cast<float>(i) / (static_cast<float>(aHorizSize) / 4 - 1);
8
9          aHorizAngles[0 * aHorizSize/4 + i] = lerp(horizMiddle, aHorizEndAngle, t);
10         aHorizAngles[1 * aHorizSize/4 + i] = lerp(aHorizEndAngle, horizMiddle, t);
11         aHorizAngles[2 * aHorizSize/4 + i] = lerp(horizMiddle, aHorizStartAngle, t);
12         aHorizAngles[3 * aHorizSize/4 + i] = lerp(aHorizStartAngle, horizMiddle, t);
13     }
14
15     // Create 1024 vertical angles in total
16     for (int i = 0; i < aVertSize; ++i) {
17         float t = static_cast<float>(i) / (static_cast<float>(aVertSize) - 1);
18
19         aVertAngles[0 * aVertSize + i] = lerp(vertMiddle, aVertEndAngle, t);
20         aVertAngles[1 * aVertSize + i] = lerp(vertMiddle, aVertStartAngle, t);
21         aVertAngles[2 * aVertSize + i] = lerp(vertMiddle, aVertStartAngle, t);
22         aVertAngles[3 * aVertSize + i] = lerp(vertMiddle, aVertEndAngle, t);
23     }
24 }
```

Listing 4.4: Implementation of the function for creating the sampling angles.

The algorithm we use to convert the HDR files to IES profiles, can be seen in pseudo-code in listing 4.1. At first, we need to load the data of the HDR file, and define the texture size, which is 256×256 pixels in this case. Furthermore, we need to define the sampling ranges, which gets set to 0° to 360° for the horizontal range and 90° to 180° for vertical range, as we only want to sample the upper half of the HDR image. We sample over the whole width of the environment map, even though a normal window would only receive light from a range of approximately 180°, to capture the brightness variance of the environment map better. After defining the sampling ranges, we calculate 256 angles for the horizontal, and 1024 angles for the vertical sampling angles.

As already mentioned in section 3.1, we choose to sample in a circular pattern around the center of the defined sampling angles to more accurately reflect the environment map on the IES texture, which is why we need one set of 256 vertical sampling angles for each of the four quadrants. The function that calculates the vertical and horizontal sampling angles can be seen in listing 4.4. The `lerp()` function is for linear interpolation between the first and second function argument, with the third argument being the interpolation factor.

Algorithm 4.1: Pseudo-code of the algorithm for converting the HDR file to a window light object.

Input: HDR file
Output: WindowLight Object

1. Load HDR data from HDR file
2. Define constants for texture size and sampling angle ranges
3. Generate sampling angles
4. **foreach** *horizontal sample angle h* **do**
5. Calculate the quadrant we are in **foreach** *vertical sample angle for the quadrant v* **do**
6. $x, y \leftarrow$ Calculate image coordinates from angles (h, v)
7. Compute integer and fractional parts of (x, y)
8. Get surrounding pixel coordinates and clamp within bounds
9. **foreach** *color channel* **do**
10. Bilinearly interpolate color values of the 4 pixels
11. Calculate luminance from interpolated pixels and store in array
12. Find maximum luminance from luminance array
13. Calculate average color from HDR data
14. Normalize the average color if necessary
15. Create sampler and image with luminance array
16. Create texture with image and sampler
17. Create WindowLight Object with color, texture and emission angles

After calculating the sampling angles, we can start with the sampling loop. For each

step in the loop, the sampling angles of the current iteration get converted into image coordinates on the environment map by the function in listing 4.5.

```
std::pair<float, float> anglesToImageCoords(float aVertAngle, float aHorizAngle, int
    aWidth, int aHeight) {
    // Convert angles to latitude and longitude
    float latitude = 180.0f - aVertAngle;
    float longitude = aHorizAngle;

    // Convert latitude and longitude to image coordinates
    float x = (longitude / 360.0f) * static_cast<float>(aWidth);
    float y = (latitude / 180.0f) * static_cast<float>(aHeight);

    return {x, y};
}
```

Listing 4.5: Implementation of the function for the conversion of sampling angles to pixel values.

Afterwards, we split the image coordinates into the fractional and non-fractional part, as the conversion from angles to image coordinates returns floating point values. The non-fractional image coordinates are then used to pick 4 pixels next to each other in a rectangular pattern, which we interpolate bilinearly on each color channel, with the fractional part of the image coordinates as the interpolation factor. We also clamp the pixels values to prevent an accidental out of bounds error. After that the algorithm converts the interpolated color values into a luminance value, which then gets added to an array that is needed for the creation of the light texture later. As previously stated in section 2.3, IES profiles do not specify the color of the light, so we decide to use the arithmetic mean over all pixels of the upper half of the environment map to calculate an average value that represents the color of the sky.

Before creating the window light object we still need the emission angles and an intensity value. As mentioned in section 3.1, we want to let the user control the emission angles themselves, so we set the emission angles to range from 0° to 360° horizontally and 0° to 35° vertically. On creation of the window light object, a function that is similar to the function in listing 4.4 calculates the 256 vertical and 256 horizontal emission angles we need for our texture, as one value in the texture corresponds to one combination of vertical and horizontal angles.

Calculating an intensity value for the window light in watts from an HDR image is not really possible, since the pixel values are just an encoding for the brightness in the scene, and there is no standard way of converting them to a physically accurate value. This is why we set the intensity of the window light to the maximum value of the luminance values that were calculated in the sampling loop.

4.3 Attaching windows to models

As mentioned in subsection 3.2.1, we want to make our window lights attachable to models in the scene, to provide better usability and a realistic representation of windows in the rendering framework.

The most reasonable way to do this is by adding a unidirectional association from the window light to a reference of a model, so it is easy to access the attributes of the model for various purposes. A bidirectional association would only be needed if the model would also need to access the windows properties, for example when rotating both the model and the attached light together, but since this was not a priority we choose to use the unidirectional association.

Algorithm 4.2: Pseudo-code algorithm for attaching the window light to a model in the scene.

```
   Input: Reference to window light, Reference to new model to attach to
 1 Get target direction from model to attach the light to
 2 Get current direction the window light is facing
 3 Calculate dot product of current window light direction and target direction
 4 Calculate angle with arc cosine of dot product
 5 if Dot product equals 1 then
 6 |  Set new position of window light and exit function
 7 if Dot product equals -1 then
 8 |  Set rotation axis to tangent of window light
 9 else
10 |  Calculate rotation axis with cross product between initial direction and target
   |   direction
11 Apply the rotation to light's model matrix with angle and rotation axis
12 Set new position of window light
```

For the workflow of attaching windows to models that are present in the scene, we first implement the attachment to models when the window light is created. Since Tamashii has a menu for adding lights that appears when right-clicking on the mouse, we extend it to additionally show an option to add window lights. We limit this option to only show up when a model is selected with a left click, so the light and the model can be connect directly on creation of the light. Furthermore, we restrict the attachment to only work with models that are flat, which we check by testing whether any of the model's dimensions are zero. During the creation the association between the light and the model is made and the light gets positioned in the center of the model, with the direction of the light aligning with the front facing vertices of the model.

Since only being able to attach the window light to a model when creating it is rather unintuitive and not perfectly user-friendly, we decide to also implement a way to change

the attached model on the fly through the edit menu of the UI. This means that we need to change the edit menu of the user interface, and implement a dropdown menu where it is possible to interactively choose which model the light should be attached to. When changing the model the window light is attached to, we need to align the window light with the new model as explained in subsection 3.2.1. The algorithm that handles the alignment can be seen in listing 4.2.

We also made sure it is possible to create a window light without attaching it for testing and debugging purposes. By combining these methods for creating new window lights and changing their attached model we aim to make the workflow user-friendly and as intuitive as possible.

4.4 Movement and scaling constraints for windows

After making our window lights attachable, we need to utilize the unidirectional association between the window light and the model to also constraint the movement and scaling. As explained in section 3.2 we choose the approach to do the calculations in the model's object space. How the implementation works in detail is discussed below, and the algorithm itself is shown in listing 4.3 in pseudo-code.

Algorithm 4.3: Pseudo-code algorithm for the movement constraint of window lights while translating manually.

Input: Reference to WindowLight, Position of WindowLight, WindowLight's new translation for this frame

Output: WindowLight's manipulated translation for this frame

```
1  if Reference is of type Light then
2      if Lightreference is of type Window then
3          if Lightreference is connected to a model then
4              Get model matrices, translations, and scale
5              Get model's AABB and light's dimensions
6              Calculate bounding box of the model with the AABB and the scale
7              Transform WindowLight's position and this frame's translation into
                 model's object space
8              Calculate bounding box of the WindowLight with the position and
                 scale
9              if light's bounds exceed model's bounds on any axis then
10                 Correct new Translation based on how far the light exceeds the
                     model's bounds
11             Transform new translation back to world space
```

The algorithm that constraints the movement, takes the reference to the window light, the current position and the translation for the current frame as input parameters. At first, we calculate the bounding boxes of the model the light is connected to by using an already implemented function, which returns an axis-aligned bounding box in object space, that gets multiplied by the scale of the model to get the correct dimensions. After that, the translation for this frame and the window light's position need to be transformed into the model's object space, like mentioned in subsection 3.2. To calculate the bounding box of the light, we combine the dimensions with the transformed position and the scaling. It is important to scale both the bounding boxes to the scale the objects have in world space in order to keep their proportions, as otherwise our calculations would not be correct. With both bounding boxes calculated, the algorithm checks whether the light's bounds would exceed the model's bounds if the translation of this frame would be applied individually for each axis.

If this is the case, we reduce the translation for this frame by the exact amount the window light would exceed the model, so it is positioned right at the edge. After that, we transform the manipulated translation values back into world space by multiplying them with the model matrix of the connected model.

This algorithm ensures that the light can never leave the model via manual gizmo movement. The algorithm is called every frame while manually moving the window light by the colored arrows indicating the light's coordinate axes, or the two-dimensional plane defined by the axes, as seen in 4.1.

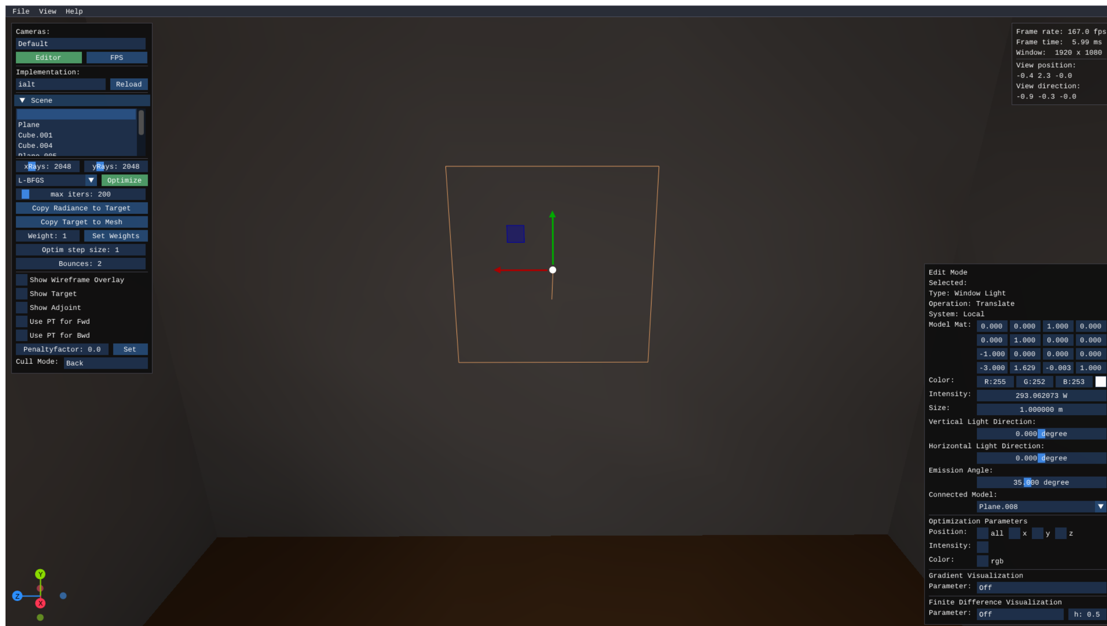


Figure 4.1: Manual gizmo movement can be done by the arrows or the plane in the UI.

The constraint for manual scaling of the window light works mostly the same, except

that the scaling factors get set to 1 when the light would exceed the connected model's bounds.

4.5 Movement constraint for windows in the optimization process

As already mentioned in section 3.3, for implementing a new constraint that can be evaluated by the algorithm, we need to create a new class that derives from the superclass `LightConstraint`, which can be seen in listing 4.6.

```
1  class LightConstraint{
2  public:
3      LightConstraint() : mPenaltyFactor(1.0), mIsActive(true) {}
4      virtual ~LightConstraint() = default;
5      void setPenaltyFactor(const double aFactor = 1.0) { mPenaltyFactor = aFactor; }
6      void setActive(const bool aActive = true) { mIsActive = aActive; }
7      void addLight(tamashii::RefLight* aRefLight) { mLights.insert(aRefLight); }
8      void removeLight(tamashii::RefLight* aRefLight) { mLights.erase(aRefLight); }
9
10     virtual double evalAndAddToGradient(Eigen::VectorXd& aGradient) = 0;
11 protected:
12     std::set<tamashii::RefLight*> mLights;
13     double mPenaltyFactor;
14     bool mIsActive;
15 };
```

Listing 4.6: Implementation of the `LightConstraint` superclass

The superclass has several important functions and parameters that are needed by all constraints, like adding and removing lights from constraints, setting the penalty factor and activating or deactivating the constraint. The penalty factor is an important variable, as it lets the user adjust how big the influence of the constraint is on the optimization algorithm. Our implementation of the `WindowInModelConstraint` does not need any additional parameters, since the evaluation function `evalAndAddToGradient` only needs the gradient vector and the lights that are affected by the constraint.

Our implementation of the `evalAndAddToGradient` function for the gradient and objective function value manipulation, which can be seen in pseudo-code in listing 4.4, has a lot of similarities in the underlying logic with the algorithm for the manual movement constraints 4.3.

We first need to calculate the bounding boxes of both the model and the window light in the model's object space, because we want to compare the bounding boxes in the model's object space. One big difference is that we also need to transform the gradients into the model's local space. We need to do this to ensure that the penalties we calculate are added to the correct axis, and also to set the y component of the gradient to zero, because in the model's object space the y-axis points into the direction of the normal vector, which is where the light should never be moved because it would break the attachment

between the window light and the model. As mentioned in subsection 3.3, we also scale the model's bounding box by a threshold value, to discourage the optimization algorithm from finding a solution that is very close to the edges of the connected model. In this case the threshold value is set to 0.75, which means the bounding box gets shrunk by 25% and penalties get added when the window light's bounding box is int the outer 25% of the actual model.

Algorithm 4.4: Pseudo-code algorithm for manipulating the gradients and the objective function value.

```
   Input: Gradient vector
   Output: Manipulated gradient vector and overall penalty value
 1 if Constraint is active then
 2     foreach Light reference added to the constraint do
 3         Get WindowLight Object and connected Model from Light reference
 4         Create penalty vector
 5         Compute model's inverse model matrix
 6         Extract gradients for light's position from gradient vector
 7         Transform gradients using the models' inverse model matrix
 8         Get model's AABB and light's dimensions
 9         Calculate bounding box of the model with the AABB and the scale
10         Transform WindowLight's position into model's object space
11         Calculate bounding box of the WindowLight with the position, scale and
            threshold
12         if light's bounds exceed model bounds on any axis then
13             Update penalty vector based on how far light exceeds model's
                threshold
14         Add the penalty vector multiplied by the penalty factor to the gradients
15         Transform gradients back to world space
16         Update the gradient vector with manipulated gradients
17         Calculate the overall penalty value with the squared norm of the penalty
            vector multiplied by the penalty factor
```

The check whether the window light exceeds the model's bounds is also mostly the same as in listing 4.3, except that the penalties get saved to a vector first, because we multiply the penalties by the penalty factor before adding them to the gradients and objective function value. The penalties are calculated per axis and represent the distance of how far the window light's bounds exceed the model's bounds. After manipulating the gradient, we need to transform the gradients back to world space and save them to the gradient vector. Furthermore, we add the squared norm of the calculated penalties to the return value, which then is added to the objective function value to evaluate the current solution

outside of our function.

To control the constraint in the framework, we also added an additional slider in the UI of the light tracing pipeline, where the penalty factor can be changed. If the penalty factor is zero or lower, the constraint is deactivated, and for everything above zero it is activated with the chosen penalty factor.

4.6 Summary

In the previous subsections, we explained the practical implementations of the challenges we discussed earlier in chapter 3.

At first we defined the general implementation of our new `WindowLight` class, and explained what attributes we need to realize our plans for the window lights.

After that, we demonstrated our implementation of the conversion algorithm for HDR files in pseudo-code.

Furthermore, we showed how we realized the attachment of window lights to models and the constraint for the manual movement and scaling, to improve the usability of our window lights.

At last, we described our new `WindowInModelConstraint` class and explained how our implementation of the `evalAndAddToGradient` function manipulates both the gradient and the objective function value to enable the optimization algorithm to find valid positions for window lights.

CHAPTER 5

Evaluation

In this chapter we will evaluate our implementation of windows in the rendering framework. First we will compare our implementation with the Cycles rendering engine in Blender, specifically on how similar the results of our implementation are compared to a typical implementation of environment maps. Since Blender can work with glTF files just like Tamashii, we can use exactly the same scenes for our test in both programs. After that we will also show 2 scenes with multiple window lights attached to the walls to show how different configurations could work.

5.1 Overall lighting and histogram comparison

At first, we do an overall lighting comparison by creating Blender scenes with the classical implementation of environment maps, and then try to recreate it with our window light in Tamashii. To not only compare the two scenes visually, we also create histograms of both scenes with short Matlab script, to see whether the overall distribution of pixel intensity values matches. For this comparison to be useful it is necessary to configure both programs correctly, to make the histograms as comparable as possible.

For accurate comparisons between both programs we use the path-tracing renderer Cycles in Blender and also the native camera path-tracing implementation in Tamashii, with special considerations being needed for parameters like the sample count, denoising and output resolution. The output resolution of both programs is set to 1920×1080 and the samples for our tests are dependent on the scene in order to have an image that has as minimal noise as possible, which is especially important when the scene is relatively dark with only one window. The clamping in both programs is set to 0 for direct lighting and 10 for indirect lighting. Because Tamashii does not have an implemented denoising solution, we used a standalone OptiX denoiser for both images for a more accurate comparison. We cannot use the Blender denoising, because these builtin denoisers have access to more information than just the raw image data, like albedo or normal data, which would

make them perform a lot better than a standalone denoiser. In our comparisons we will always use only one window, because the more windows we add the harder it would be to compare our implementation to the Blender scene.

5.1.1 First comparison

For the first comparison we use the small office scene and the Victoria Sunset HDR image [Zaa24], which can be seen in figure 5.1.

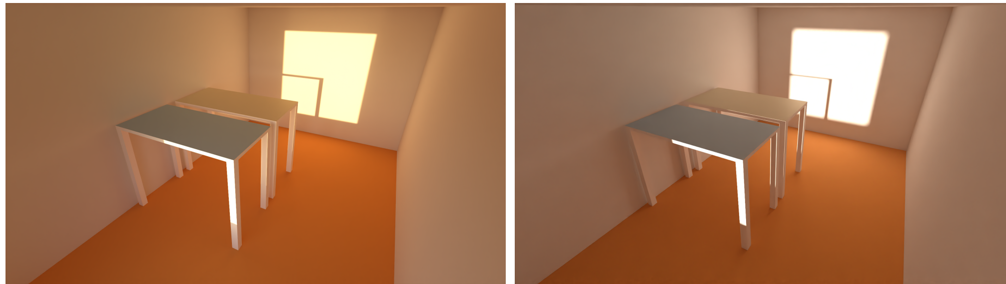
Figure 5.1: The Victoria Sunset environment map.

In Blender, we import the gltf file and place it at the world origin, and then create holes in the walls by using the Boolean modifier with a cube, to cut exactly a $2m \times 2m$ sized square out of the wall where our window should be. The environment map in Blender is used as the world background with strength set to 1, which means the environment map is infinitely far away from the scene and illuminates everything. To create a lighting condition where the sun shines straight through the window, we also had to rotate the environment map by 216° around the z-axis and 5.5° around the y-axis.

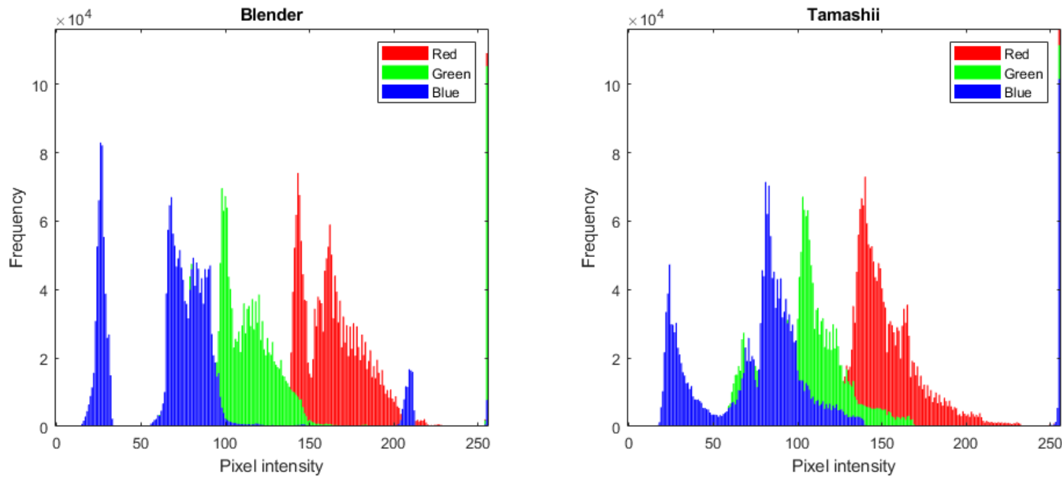
In Tamashii, we simply load the scene and create a new window light at the center of a wall, with $2m \times 2m$ as the dimensions and without changing the intensity value. To have similar lighting conditions, we also need to set the emission angles of our window light to 1°, to get the same hard lighting effect as the Blender scene, and the exposure is also set to 6 under the tone-mapping settings. We set sample count for this comparison to 1024 for Blender and 30000 for Tamashii, because with the low emission angles in Tamashii we need a lot of samples to have a relatively noise-free image. This is because the lower the emission angles, the more difficult it is for rays to hit the window light at an angle it is still emitting light. In Blender on the other hand, having only 1024 samples is not

a problem, since the rays can go through the hole in the wall at any angle and hit the environment map.

As we can see in figure 5.2, the hard lighting that is present in the Blender scene from the sun can be recreated pretty accurately by changing the emission angles of the window light in Tamashii. The overall brightness of the scene also matches, however the color of the window light is less yellow which is likely because in our implementation, the color is averaged over the whole sky, while in Blender most of the light that shines through the window is the sun, which has a yellow tint.



(a) Blender small office test scene (b) Tamashii small office test scene



(c) Histograms of Blender and Tamashii scene

Figure 5.2: First overall lighting and histogram comparison in the small office test scene.

This can also be seen in the histogram comparison of the scene in figure 5.2, where at 255 pixel intensity on the x-axis the histogram of the Tamashii scene has a lot higher frequency of the blue channel than the Blender scene, which makes the scene more white instead of yellow. The rest of the histograms are pretty similar in comparison, as the most frequent intensity values of all color channels are distributed similarly.

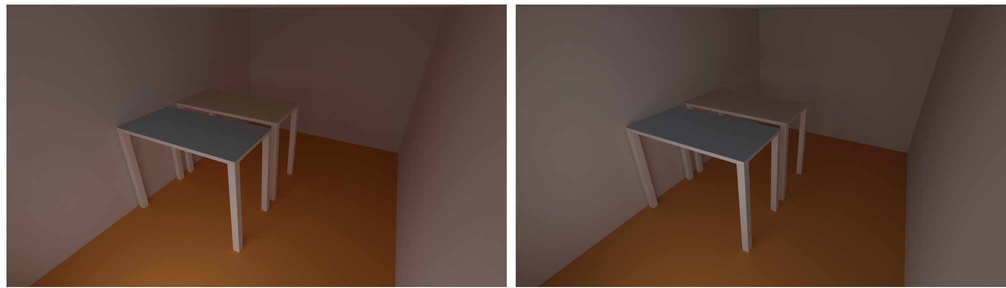
5.1.2 Second comparison

For the second comparison we also use the small office scene with the same window size, but with a different HDR file called Overcast Soil [SM24], which can be seen in figure 5.3. For this test both programs use a sample count of 1024, and in Tamashii the exposure is set to 0.8 under the tone mapping settings. To match the scene from Blender, we set the emission angles of the window light to 90°. The intensities of the environment map in Blender and the window light in Tamashii are not changed.

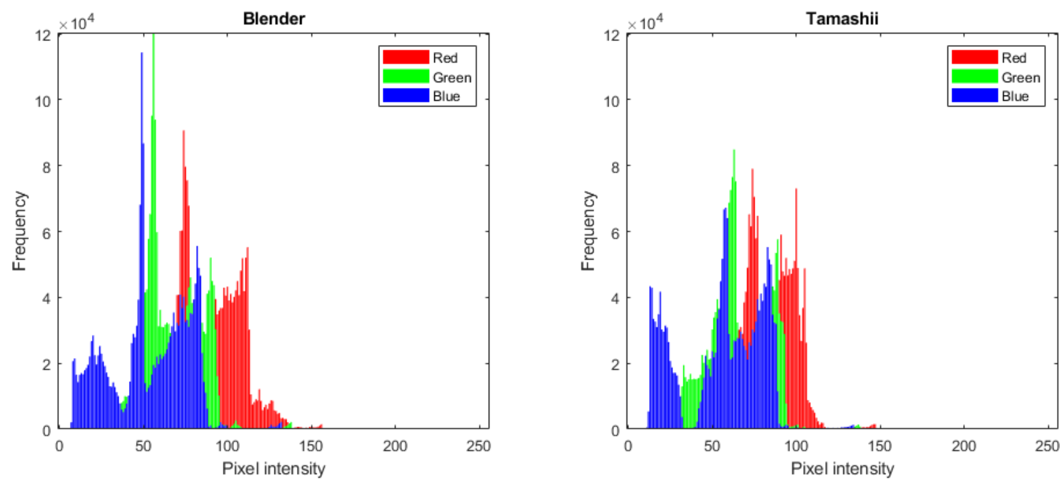
Figure 5.3: The Overcast Soil environment map.

The environment map used in this comparison has very diffuse lighting mainly because the sky is very cloudy, which scatters the light from the sun a lot. As we can see in figure 5.4 the overall brightness of the scene in Tamashii matches very well compared to the Blender scene. This is because when adjusting the emission angles to a high value like 90°, it is possible to emulate the diffuse lighting through the clouds. In the Blender scene the light gets reflected by the ground a little more than in the Tamashii scene, which can be seen on the walls as they have a slight orange tint. This can be explained by the angle the light falls into the scene, as in Blender most of the light falls into the scene from the sky to the ground, while in the Tamashii scene the light emits from the window light evenly in all directions.

When comparing the histograms of both scenes in figure 5.4, we can see that the pixel values with the highest frequencies are at roughly the same intensity values across all channels, which aligns with the assessment that our implementation in Tamashii can achieve a similar look to Blender with this environment map.



(a) Blender small office test scene (b) Tamashii small office test scene



(c) Histograms of Blender and Tamashii scene

Figure 5.4: Second overall lighting and histogram comparison scene in the small office test scene.

5.2 Evaluation of lighting optimization modifications

In this section we will evaluate our modifications to the lighting optimization algorithm, specifically how well the algorithm finds valid positions inside the connected model. We can do this by plotting the objective function value and see whether it converges to a local minimum.

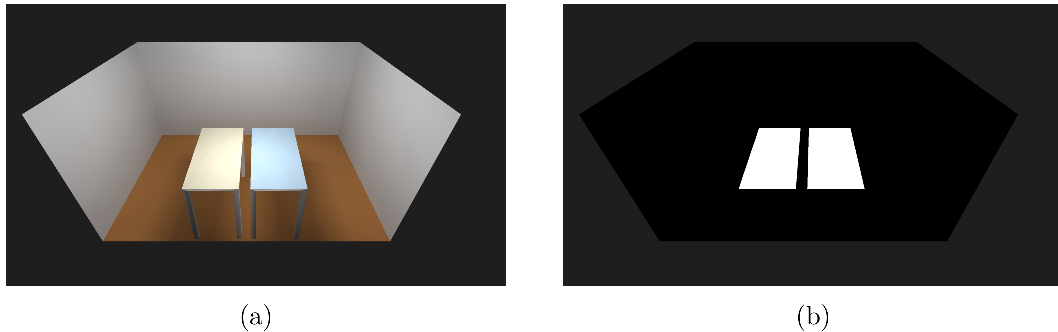
5.2.1 First optimization test

For the first test scene we will use the small office test scene with the predefined lighting target being the two tabletops, and the HDR file Symmetrical Garden [DS24b], which can be seen in figure 5.5. We use the L-BFGS algorithm as the optimization algorithm and cap the maximum iterations at 200. The optimizer also uses a step size of 0.3 and the penalty factor for the movement constraint we implemented is set to 5. For the window lights we use one light on the left wall of the scene with the intensity set to 2000W and

dimension set to $1m \times 1m$.

Figure 5.5: The Symmetrical Garden environment map.

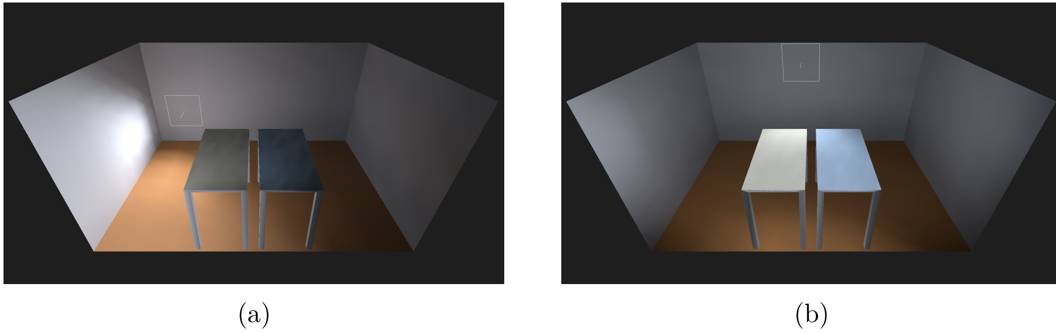
The scene consists of a small room with two tables being placed in the middle against one of the longer walls. In figure 5.6 the scene is displayed together with the lighting target which represents the two tabletops.



(a) (b)

Figure 5.6: On the left the small office test scene can be seen lit by a point light (a). On the right we see the predefined lighting target on the tabletops. For our test the optimization algorithms will determine the best position for the window lights to optimally light the target.

For this test the window light is positioned in the lower left corner of the wall before the optimization starts, as seen in figure 5.7. After 46 evaluations and around 2.6 seconds the algorithm is finished and moved the window light to the top of the wall almost right in the middle. This is a valid position, as the upper edge of the window light is still below the upper edge of the wall, and the light is able to evenly illuminate both of the tabletops.



(a) (b)

Figure 5.7: In this figure the left image (a) represents the starting position of the window light before the optimization, where the the tabletops are not well illuminated. Image (b) is the optimized lighting position for the target tabletops.

In figure 5.8, we can see how the both the objective function value and the calculated constraint penalties changed over the 46 evaluations the optimization algorithm performed in total. As we can see, after the first two evaluations the algorithm already found a reasonably good position, but then at evaluation 4, the constraint penalty gets really high because the algorithm tried to move the window light out of the bounds of the model. After that the constraint penalties reduce until evaluation 14, where the optimization algorithm tried to move the window away from the wall again. From evaluation 17 until the end, both the objective function value and the constraint penalties stay relatively stable.

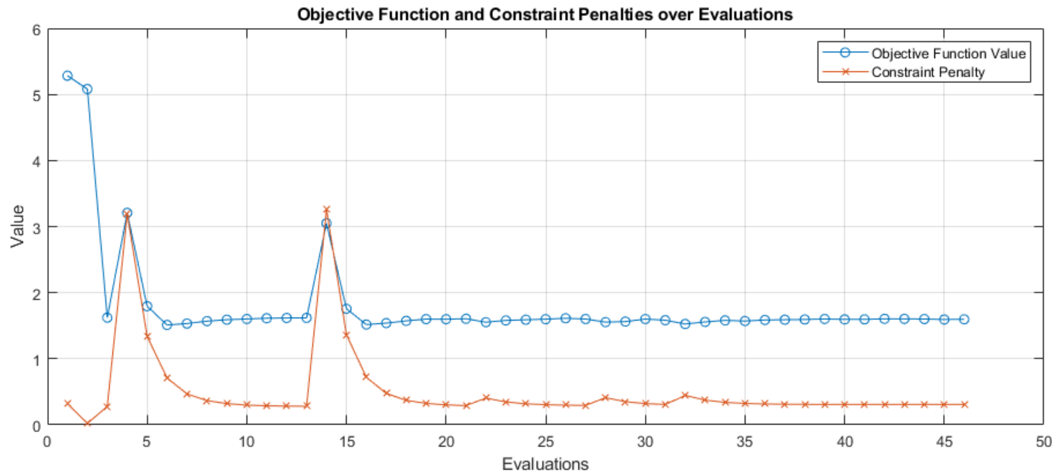


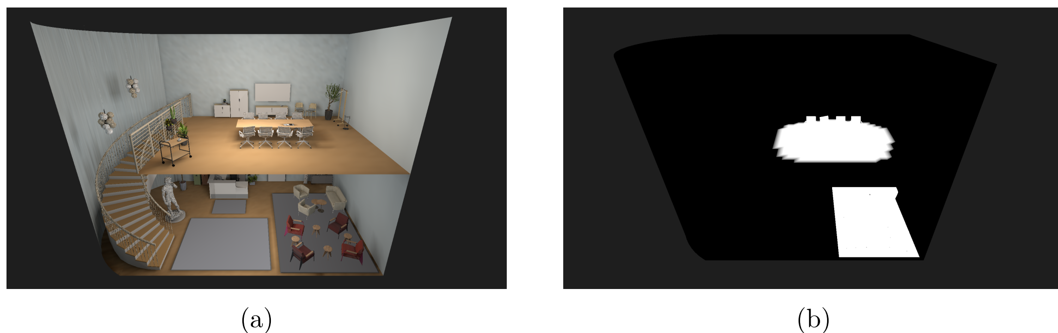
Figure 5.8: Diagram of the objective function and constraint penalty values over the evaluations.

It is important to note that the constraint penalty never becomes zero, because as we mentioned in section 4.5, we scale the bounding box of the model down by a threshold value to discourage the algorithm from finding a solution that is very close to the edge.

This means, that the penalty is above zero even before the window light reaches the edge of the model. In this case, the algorithm chooses to still position the window light almost at the upper edge of the wall, because it seems the objective function value gets better the higher the window light is positioned. This means that even though the window light is positioned right at the edge, shrinking the model's bounding box still helped in preventing the algorithm from choosing an invalid beyond the upper edge. Increasing the penalty factor would also help to keep the window light further away from the upper edge in this case.

5.2.2 Second optimization test

For the second optimization test we will use the large office scene, with two different lighting targets, one being the area around the big table on the upper floor, and the other being the chill out area on the lower floor. We also use the L-BFGS algorithm as the optimization algorithm and cap the maximum iterations at 200. The optimizer also uses a step size of 0.3 and the penalty factor for the movement constraint we implemented is set to 5 again. As we have two lighting targets for this test on two different floors, we will also use two window lights that get optimized at the same time. Both lights use the HDR file Symmetrical Garden [DS24b] again, which can be seen in figure 5.5.

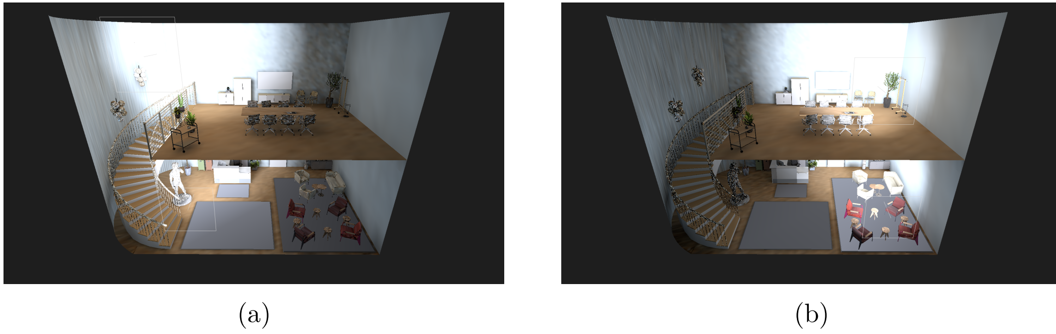


(a) (b)

Figure 5.9: On the left the large office test scene can be seen lit by one point light on each floor (a). On the right we see the predefined lighting targets on both floors. For our test the optimization algorithms will determine the best position for the window lights to optimally light both targets.

The scene for this test consists of two floors, with the lower floor being a reception and waiting room area, and the upper floor being a room with a big table in the middle. As we can see in figure 5.9 the lighting targets are placed around the seats on the bottom floor and the table area on the upper floor.

In figure 5.10, the start position of both of the window lights can be seen on the left, and the position after the optimization process can be seen on the right. We place one window light on each of the floors, so the optimization algorithm can find a a solution for both lighting targets simultaneously. The algorithm manages to find a valid position



(a) (b)

Figure 5.10: In this figure on the left side the position of both window lights before the optimization process is displayed (a). On the right side we can see the position the optimization algorithm found.

for both lights in around 8.3 seconds with 49 evaluations, where both lights are able to illuminate the respective lighting targets sufficiently.

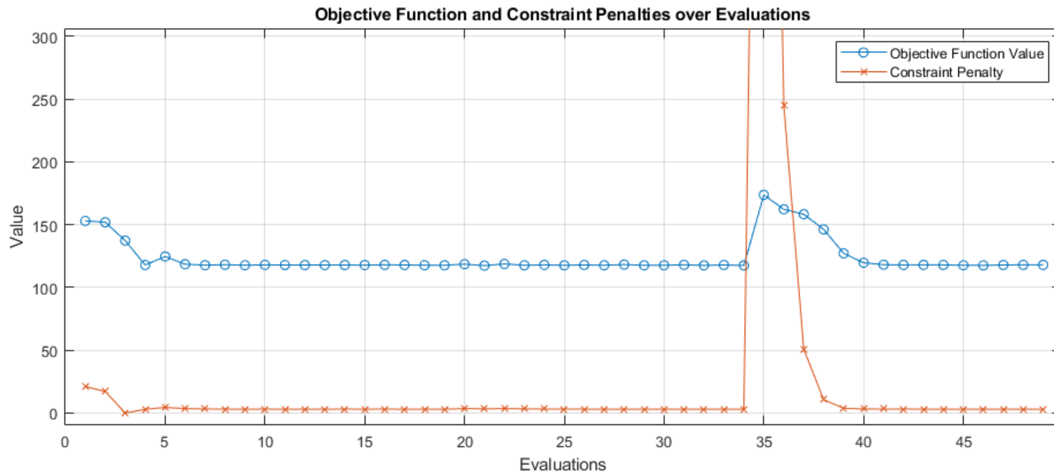


Figure 5.11: Diagram of the objective function and constraint penalty values over the evaluations.

In the diagram in figure 5.11, we can see that around evaluation 6 both the objective function value and the constraint penalty stay relatively stable until evaluation 35, where the algorithm tried to move both windows out of the bounds of the wall at the same time, which is why the constraint penalty goes up to 1102. After that the objective function value and constraint penalties start to lower again until evaluation 46 where the algorithm stops. The reason why the the constraint penalty never reaches zero is similar as in subsection subsec:first-optimization-test, only that this time the edges of both window lights are right at the threshold where the penalties start to get added, which means our constraint prevented the algorithm from placing the lights too close to the edges successfully.

5.3 Lighting configurations

In this section we will show a lighting configuration with multiple different HDR files loaded in window lights, to show the versatility of our implementation in the adjoint light tracing pipeline. Since we implemented our window lights to be able to load a HDR file on creation of the light, it is possible to load different HDR files for different windows.

Figure 5.12: Large office scene lit by two window lights on each floor.

In figure 5.12, we created 2 window lights on each floor of the large office scene, with different HDR files loaded for each floor. The window lights on the lower floor use a HDR file called The Sky Is On Fire [GZnd], and the window lights on the upper floor use the Symmetrical Garden Environment Map [DS24b], which can be seen in figure 5.13 and figure 5.5 respectively. As we can see, with our implementation it is possible to experiment with a lot of different configurations for lighting design optimization, especially with the parameter settings the window lights offer.

Figure 5.13: The environment map called The Sky Is On Fire.

CHAPTER 6

Conclusion and Future Work

For lighting design optimization in office or living spaces, natural light shining through windows plays a big role, and to make the optimization process more realistic and true to life, our implementation aims to provide a simulation of real windows through area lights. By converting HDR files to IES profiles we can accurately simulate the surrounding environment of a scene. Furthermore, the implemented configuration settings for our window lights can help configuring the light, and the attachment of windows to other models combined with the implemented movement constraints can help finding valid positions for the windows in the scene.

For future work it would be possible to refine the optimization algorithm, so that the algorithm can automatically choose which wall the window light should be attached to, in order to find the best lighting configuration. Another topic for further exploration would be to calculate the emission angles that fit the scene automatically, by locating areas in the environment map that emit light stronger than other parts and then using that information to calculate fitting emission angles. An example of this would be that the emission angles get set to a higher value when the intensity values of the environment map are evenly distributed in the sky, and set to a lower value when there is a small area in the environment map where the intensity values are a lot higher than the rest. Furthermore it would be possible to implement more shapes for the window lights, so that round, oval or triangular windows can be created, and also enabling windows to be placed on curved walls. Another idea to consider is to implement the automatic optimization for other parameters of window lights like the emission angles or the light direction.

In conclusion, our implementation of window lights is able to simulate windows in a more performant way than the traditional usage of environment maps in a light tracing pipeline, and can therefore be used in light parameter optimization.

Appendix

The algorithms in this chapter are the implementations of the pseudocode algorithms we explained in chapter 4. First, in listing A.1 we show the algorithm for the conversion of HDR files to IES profiles. Furthermore, listing A.2 contains the algorithm that aligns a window light with objects in the scene, such as walls or ceilings, when the object to which the window light is attached to is changed. After that, listing A.3 presents the function that prevents a window light from leaving the object it is connected to when manually moving it. Finally, listing A.4 shows the algorithm that calculates the penalties for the automatic optimization process when a window light would leave the object it is connected to.

```
1  std::unique_ptr<WindowLight> io::Import::loadHdriToIes(const std::filesystem::path &
       aFile) {
2
3      // Load HDR file
4      int width, height, desiredChannels = 3, nrComponents;
5      float *data = stbi_loadf(aFile.string().c_str(), &width, &height, &nrComponents,
           desiredChannels);
6      if (!data) {
7          spdlog::error("Failed to load HDR image.");
8          return nullptr;
9      }
10
11     // Define the size of the candela texture
12     constexpr int vertSize = 256, horizSize = 256;
13     // Angles for sampling
14     std::vector<float> verticalAnglesSampling(4 * vertSize), horizontalAnglesSampling(
           horizSize);
15     // Angles for emission
16     std::vector<float> verticalAnglesEmitting, horizontalAnglesEmitting;
17     // Sampling angle range
18     float sampleVertAngleStart = 90, sampleVertAngleEnd = 180, sampleHorizAngleStart =
           0, sampleHorizAngleEnd = 360;
19     // Emission angle range
20     float emitVertAngleStart = 0, emitVertAngleEnd = 35, emitHorizAngleStart = 0,
           emitHorizAngleEnd = 360;
21     // Initialize candela values
22     std::vector candelaValues(vertSize * horizSize, 0.0f);
23     // Generate the sampling angles
24     generateAnglesSampling(sampleHorizAngleStart, sampleHorizAngleEnd,
           sampleVertAngleStart ,sampleVertAngleEnd, horizSize, vertSize,
           horizontalAnglesSampling, verticalAnglesSampling);
25
```

```
    // Extra index to pick correct vertical Angles for sampling
    int vertAngleIndex = 0;
    // Sample the HDRI and calculate luminance
    for (int h = 0; h < horizSize; ++h) {
        float horizAngle = horizontalAnglesSampling[h];
        if(h == horizSize / 4) {
            vertAngleIndex = 1;
        } else if (h == horizSize / 2) {
            vertAngleIndex = 2;
        } else if (h == 3 * horizSize / 4) {
            vertAngleIndex = 3;
        }
        for (int v = 0; v < vertSize; ++v) {
            float vertAngle = verticalAnglesSampling[vertAngleIndex * (vertSize - 1) +
                v];

            // Get pixel values from sampling angles
            auto [x, y] = anglesToImageCoords(vertAngle, horizAngle, width, height);

            // Calculate the non-fractional and fractional part of the coordinates
            float yFloor = std::floor(y);
            float xFloor = std::floor(x);
            float fx = x - xFloor;
            float fy = y - yFloor;

            // Determine the integer coordinates of the surrounding pixels
            int x0 = std::clamp(static_cast<int>(xFloor),0, width - 1);
            int x1 = std::clamp(x0 + 1, 0, width - 1);
            int y0 = std::clamp(static_cast<int>(yFloor), 0, height - 1);
            int y1 = std::clamp(y0 + 1, 0, height - 1);

            // Interpolate the color values
            std::vector<float> rgb(desiredChannels, 0.0f);
            for (int c = 0; c < desiredChannels; ++c) {

                float p00 = data[(y0 * width + x0) * desiredChannels + c];
                float p01 = data[(y1 * width + x0) * desiredChannels + c];
                float p10 = data[(y0 * width + x1) * desiredChannels + c];
                float p11 = data[(y1 * width + x1) * desiredChannels + c];

                rgb[c] = lerp(lerp(p00, p10, fx), lerp(p01, p11, fx), fy);
            }
            // Calculate the index in the one-dimensional list
            int index = h * vertSize + v;
            // Calculate luminance and store it in the array
            candelaValues[index] = calculateLuminance(rgb);
        }
    }
    float intensity = *std::max_element(candelaValues.begin(), candelaValues.end());

    // Calculate the average color value
    glm::vec3 totalColor(0, 0, 0);
    for (size_t i = 0; i < width * (height/2) * desiredChannels; i += 3) {
        float r = data[i], g = data[i+1], b = data[i+2];
        totalColor[0] += r;     // Red channel
        totalColor[1] += g;     // Green channel
        totalColor[2] += b;     // Blue channel
    }
    glm::vec3 averageColor = totalColor / (static_cast<float>(width) * (static_cast<
        float>(height)/2));

    // Make sure the highest value is 1.0 and scale the others accordingly
```

```
86      if (float maxColorValue = glm::compMax(averageColor); maxColorValue > 1.0f) {
87          averageColor /= maxColorValue;
88      }
89
90      // Create sampler
91      constexpr Sampler sampler = {
92          Sampler::Filter::LINEAR, Sampler::Filter::LINEAR, Sampler::Filter::LINEAR,
93          Sampler::Wrap::CLAMP_TO_BORDER, Sampler::Wrap::MIRRORED_REPEAT,
94          Sampler::Wrap::CLAMP_TO_EDGE, 0, 0
95      };
96
97      // Create image
98      Image* img = Image::alloc(aFile.filename().string());
99      img->init(horizSize, vertSize, Image::Format::R32_FLOAT, candelaValues.data());
100
101     // Create texture with image and sampler
102     Texture* texture = Texture::alloc();
103     texture->image = img;
104     texture->sampler = sampler;
105
106     // Create the WindowLight object
107     auto light = std::make_unique<WindowLight>();
108     light->setFilepath(aFile.string());
109     light->setIntensity(intensity);
110     light->setColor(averageColor);
111     light->setCandelaTexture(texture);
112     light->generateHorizontalEmissionAngles(emitHorizAngleStart, emitHorizAngleEnd,
            horizSize);
113     light->generateVerticalEmissionAngles(emitVertAngleStart, emitVertAngleEnd,
            vertSize);
114     light->setConnectedModel(nullptr);
115
116     stbi_image_free(data); // Free the original data
117     return std::move(light);
118 }
```

Listing A.1: Algorithm for converting the HDR file to a window light object.

```
1       void MainGUI::alignWindowWithModelRefLight(RefLight &aRefLight, const RefModel &
            aRefModel) {
2
3       // Get the direction of the light and the target direction
4       glm::vec3 targetNormal = glm::normalize(glm::mat3(aRefModel.model_matrix) * glm::
            vec3(aRefModel.refMeshes.front()->mesh->getVerticesArray()->normal));
5       glm::vec3 initialDirection = glm::normalize(glm::mat3(aRefLight.model_matrix) * glm
            ::vec3(0, 0, -1));
6
7       // Calculate the angle between the directions
8       float dotProduct = glm::dot(initialDirection, targetNormal);
9       float angle = acos(std::clamp(dotProduct, -1.0f, 1.0f));
10
11      glm::vec3 rotationAxis;
12
13      if (glm::epsilonEqual(dotProduct, 1.0f, 1e-6f)) {
14         // Vectors are parallel, only new position needed
15         aRefLight.model_matrix[3] = glm::vec4(glm::vec3(aRefModel.model_matrix[3]), 1.0f
               );
16         return;
17      }
18      if (glm::epsilonEqual(dotProduct, -1.0f, 1e-6f)) {
```

```
19         // Vectors are antiparallel, choose light's tangent as rotation axis
20         rotationAxis = aRefLight.light->getDefaultTangent();
21      } else {
22         glm::mat3 toObjectSpaceMatrix = glm::mat3(glm::inverse(glm::scale(aRefLight.
               model_matrix, glm::vec3(1.0f) / aRefLight.transforms.front()->scale)));
23         rotationAxis = toObjectSpaceMatrix * glm::normalize(glm::cross(initialDirection,
                targetNormal));
24      }
25      // Apply rotation and new position
26      aRefLight.model_matrix = glm::rotate(aRefLight.model_matrix, angle, rotationAxis);
27      aRefLight.model_matrix[3] = glm::vec4(glm::vec3(aRefModel.model_matrix[3]), 1.0f);
28   }
```

Listing A.2: Algorithm for aligning a window light with the connected model.

```
1  void MainGUI::translationValidityCheck(const std::shared_ptr<Ref> &aSelection, glm::
       vec3 &aOldTranslation, glm::vec3 &aNewTranslation) {
2      if (aSelection->type == Ref::Type::Light) {
3         if (const auto &lightRef = dynamic_cast<RefLight &>(*aSelection);
4            lightRef.light->getType() == Light::Type::WINDOW) {
5            if (lightRef.connectedRefModel) {
6               const auto &windowLight = dynamic_cast<WindowLight &>(*lightRef.light);
7               const auto &refModel = dynamic_cast<RefModel &>(*lightRef.
                    connectedRefModel);
8
9               // Get Model matrices
10              const auto modelMatrix = refModel.model_matrix;
11              const auto inverseModelMatrix = glm::inverse(modelMatrix);
12
13              glm::vec3 modelTranslation = refModel.transforms.front()->translation;
14              glm::vec3 modelScale = refModel.transforms.front()->scale;
15
16              // Get AABB of model
17              aabb_s AABB = refModel.model->getAABB();
18              // Get dimensions of light
19              glm::vec3 lightDimensions = windowLight.getDimensions();
20
21              // Divide dimension by 2 for the calculation of the "bounding box" later
22              for (int i = 0; i < 3; i++) {
23                 lightDimensions[i] == 0 ? lightDimensions[i] = 0 : lightDimensions[i]
                       /= 2;
24              }
25
26              // Change axes so it aligns with the models axes
27              lightDimensions[2] = lightDimensions[1];
28              lightDimensions[1] = 0;
29
30              // calculate the "bounding box" of the model
31              const glm::vec3 minModelBounds = {
32                 modelScale.x * AABB.mMin.x, modelScale.y * AABB.mMin.y, modelScale.z *
                       AABB.mMin.z
33              };
34              const glm::vec3 maxModelBounds = {
35                 modelScale.x * AABB.mMax.x, modelScale.y * AABB.mMax.y, modelScale.z *
                       AABB.mMax.z
36              };
37
38              // Transform the translation in world Space into the models Object space
39              glm::vec3 oldTranslationObjectSpace = inverseModelMatrix * glm::vec4{
                    aOldTranslation, 1.0f};
```

```
40              glm::vec3 newTranslationObjectSpace = inverseModelMatrix * glm::vec4{
                    aOldTranslation + aNewTranslation, 1.0f};
41              newTranslationObjectSpace -= oldTranslationObjectSpace;
42
43              // Multiply by the scale because it was reversed by the inverse matrix
44              oldTranslationObjectSpace *= modelScale;
45              newTranslationObjectSpace *= modelScale;
46
47              // calculate the "bounding box" of the light
48              const glm::vec3 minLightBounds = {
49                 oldTranslationObjectSpace.x - lightDimensions.x,
                       oldTranslationObjectSpace.y - lightDimensions.y,
50                 oldTranslationObjectSpace.z - lightDimensions.z
51              };
52              const glm::vec3 maxLightBounds = {
53                 oldTranslationObjectSpace.x + lightDimensions.x,
                       oldTranslationObjectSpace.y + lightDimensions.y,
54                 oldTranslationObjectSpace.z + lightDimensions.z
55              };
56
57              // Determine wether the light is still inside or outside the model
58              if (minLightBounds.x + newTranslationObjectSpace.x < minModelBounds.x) {
59                 newTranslationObjectSpace.x = minModelBounds.x - minLightBounds.x;
60              } else if (maxLightBounds.x + newTranslationObjectSpace.x > maxModelBounds
                    .x) {
61                 newTranslationObjectSpace.x = maxModelBounds.x - maxLightBounds.x;
62              }
63              if (minLightBounds.y + newTranslationObjectSpace.y < minModelBounds.y) {
64                 newTranslationObjectSpace.y = minModelBounds.y - minLightBounds.y;
65              } else if (maxLightBounds.y + newTranslationObjectSpace.y > maxModelBounds
                    .y) {
66                 newTranslationObjectSpace.y = maxModelBounds.y - maxLightBounds.y;
67              }
68              if (minLightBounds.z + newTranslationObjectSpace.z < minModelBounds.z) {
69                 newTranslationObjectSpace.z = minModelBounds.z - minLightBounds.z;
70              } else if (maxLightBounds.z + newTranslationObjectSpace.z > maxModelBounds
                    .z) {
71                 newTranslationObjectSpace.z = maxModelBounds.z - maxLightBounds.z;
72              }
73
74              // Transform the new translation back into the world coordinates
75              aNewTranslation = modelMatrix * glm::vec4{((newTranslationObjectSpace /
                    modelScale)), 1.0f} - glm::vec4{
76                                 modelTranslation, 1.0f};
77           }
78        }
79     }
80  }
```

Listing A.3: Algorithm for the movement constraint of window lights while translating.

```
1  double evalAndAddToGradient(Eigen::VectorXd& aGradient) override {
2      double f = 0.0;
3      if( mIsActive ){
4         for(const tamashii::RefLight* refLight : mLights) {
5            const auto &windowLight = dynamic_cast<tamashii::WindowLight &>(*refLight->
                 light);
6            const auto &refModel = dynamic_cast<tamashii::RefModel&>(*refLight->
                 connectedRefModel);
7
```

```
8           const auto idx = static_cast<Eigen::Index>(refLight->ref_light_index);
9           const glm::vec3& p = refLight->position;
10          Eigen::Vector3d d; d.setZero();
11
12          // Get Model matrices
13          const auto modelMatrix = refModel.model_matrix;
14          glm::vec3 modelScale, modelTranslation;
15          glm::quat modelRotation;
16          tamashii::math::decomposeTransform(modelMatrix, modelTranslation,
                modelRotation, modelScale);
17          const auto inverseModelMatrix = glm::inverse(glm::scale(modelMatrix, glm::
                vec3(1.0f) / modelScale));
18
19          glm::dvec3 gradients = {aGradient(idx * LightOptParams::MAX_PARAMS +
                LightOptParams::POS_X)
20                                  ,aGradient(idx * LightOptParams::MAX_PARAMS +
                                        LightOptParams::POS_Y)
21                                  ,aGradient(idx * LightOptParams::MAX_PARAMS +
                                        LightOptParams::POS_Z)};
22
23          gradients = glm::dmat3(inverseModelMatrix) * gradients;
24
25          // Get bounding box of model
26          const tamashii::aabb_s AABB = refModel.model->getAABB();
27          // Get dimensions of light
28          glm::vec3 lightDimensions = windowLight.getDimensions();
29
30          // Divide dimension by 2 for the calculation of the "bounding box" later
31          for (int i = 0; i < 3; i++) {
32             lightDimensions[i] == 0 ? lightDimensions[i] = 0 : lightDimensions[i] /=
                   2;
33          }
34
35          // Change axes so it aligns with the models axes
36          lightDimensions[2] = lightDimensions[1];
37          lightDimensions[1] = 0;
38
39          // calculate the "bounding box" of the model
40          const float factor = 0.75;
41          const glm::vec3 minModelBounds = {modelScale.x * AABB.mMin.x * factor,
                modelScale.y * AABB.mMin.y * factor, modelScale.z * AABB.mMin.z * factor
                };
42          const glm::vec3 maxModelBounds = {modelScale.x * AABB.mMax.x * factor,
                modelScale.y * AABB.mMax.y * factor, modelScale.z * AABB.mMax.z * factor
                };
43
44          // Transform the translation in world Space into the models Object space
45          glm::vec3 positionObjectSpace = inverseModelMatrix * glm::vec4{p, 1.0f};
46
47          // calculate the "bounding box" of the light
48          const glm::vec3 minLightBounds = {positionObjectSpace.x - lightDimensions.x,
                positionObjectSpace.y - lightDimensions.y, positionObjectSpace.z -
                lightDimensions.z};
49          const glm::vec3 maxLightBounds = {positionObjectSpace.x + lightDimensions.x,
                positionObjectSpace.y + lightDimensions.y, positionObjectSpace.z +
                lightDimensions.z};
50
51          // Determine whether the light is still inside or outside the model
52          if (minLightBounds.x <= minModelBounds.x) {
53             d[0] = static_cast<double>(minLightBounds.x) - static_cast<double>(
                   minModelBounds.x);
54          } else if (maxLightBounds.x >= maxModelBounds.x) {
```

```
55             d[0] = static_cast<double>(maxLightBounds.x) - static_cast<double>(
                   maxModelBounds.x);
56          }
57          if (minLightBounds.y <= minModelBounds.y) {
58             d[1] = static_cast<double>(minLightBounds.y) - static_cast<double>(
                   minModelBounds.y);
59          } else if (maxLightBounds.y >= maxModelBounds.y) {
60             d[1] = static_cast<double>(maxLightBounds.y) - static_cast<double>(
                   maxModelBounds.y);
61          }
62          if (minLightBounds.z <= minModelBounds.z) {
63             d[2] = static_cast<double>(minLightBounds.z) - static_cast<double>(
                   minModelBounds.z);
64          } else if (maxLightBounds.z >= maxModelBounds.z) {
65             d[2] = static_cast<double>(maxLightBounds.z) - static_cast<double>(
                   maxModelBounds.z);
66          }
67
68          gradients[0] += mPenaltyFactor * d[0];
69          gradients[1] = 0;
70          gradients[2] += mPenaltyFactor * d[2];
71
72          gradients = glm::dmat3(glm::scale(modelMatrix, glm::vec3(1.0f) / glm::vec3(
                modelScale))) * gradients;
73
74          f += 0.5 * mPenaltyFactor * d.squaredNorm();
75
76          aGradient(idx * LightOptParams::MAX_PARAMS + LightOptParams::POS_X) =
                gradients[0];
77          aGradient(idx * LightOptParams::MAX_PARAMS + LightOptParams::POS_Y) =
                gradients[1];
78          aGradient(idx * LightOptParams::MAX_PARAMS + LightOptParams::POS_Z) =
                gradients[2];
79       }
80    }
81    return f;
82 }
```

Listing A.4: Algorithm for manipulating gradients and objective function value.

Overview of Generative AI Tools Used

When writing this thesis we used ChatGPT 4o as a helping tool for questions about the documentation and syntax of C++, GLSL and LaTeX. This included questions about whether built-in functions for a specific use case exist and general information about datatypes, keywords and access modifiers for example.

In LaTeX, we also let the AI generate the basic structure of a pseudocode algorithm, which we then modified and extended based on our implementation of the algorithms.

Furthermore, we asked ChatGPT for improvements on individual passages in the thesis we wrote, but never directly copied generated text and only used the suggestions the AI gave to further improve the structure and wording of our writing ourselves.

List of Figures

2.1 Example of an environment map called Rosendal Plains [DS24a]. 8
2.2 Visualization of the angles in an IES file [Ill19]. 10
2.3 Light distribution of the unshortened version of Listing 2.1. 11

3.1 Two area lights with an emission angle of 1° and 45° respectively. 15
3.2 Image (a) shows the small office test scene in Blender with the The Sky Is On Fire environment map [GZnd]. Image (b) shows the same scene, with the same environment map rotated by 100° around the Z axis. 16
3.3 This is a sketch on how we convert HDR files to IES profiles. 16
3.4 A sketch of a wall, where the red lines indicate the threshold from which penalties get added to the gradient. 20

4.1 Manual gizmo movement can be done by the arrows or the plane in the UI. 31

5.1 The Victoria Sunset environment map. 36
5.2 First overall lighting and histogram comparison in the small office test scene. 37
5.3 The Overcast Soil environment map. 38
5.4 Second overall lighting and histogram comparison scene in the small office test scene. 39
5.5 The Symmetrical Garden environment map. 40
5.6 On the left the small office test scene can be seen lit by a point light (a). On the right we see the predefined lighting target on the tabletops. For our test the optimization algorithms will determine the best position for the window lights to optimally light the target. 40
5.7 In this figure the left image (a) represents the starting position of the window light before the optimization, where the the tabletops are not well illuminated. Image (b) is the optimized lighting position for the target tabletops. . . . 41
5.8 Diagram of the objective function and constraint penalty values over the evaluations. 41
5.9 On the left the large office test scene can be seen lit by one point light on each floor (a). On the right we see the predefined lighting targets on both floors. For our test the optimization algorithms will determine the best position for the window lights to optimally light both targets. 42

5.10 In this figure on the left side the position of both window lights before the optimization process is displayed (a). On the right side we can see the position the optimization algorithm found. 43
5.11 Diagram of the objective function and constraint penalty values over the evaluations. 43
5.12 Large office scene lit by two window lights on each floor. 44
5.13 The environment map called The Sky Is On Fire. 45

List of Tables

Glossary

candela Candela is the SI unit of luminous intensity, which measures the amount of light emitted in a particular direction.. 10

glTF A file format that stores scene or model information in JSON format.. 1, 5, 26, 35

Acronyms

API Application Programming Interface. 5

CPU Central Processing Unit. 23

GLSL OpenGL Shading Language. 5

GPU Graphics Processing Unit. 5, 24, 25

GUI Graphical User Interface. 26

HDR High Dynamic Range. xiii, 2, 8, 9, 14–16, 23, 26–28, 34, 36, 38, 39, 42, 44, 47, 49, 51, 59

HLSL High-Level Shader Language. 5

IES Illuminating Engineering Society. xiii, 2, 3, 5, 9, 10, 14–17, 23, 25–28, 47, 49, 59

LDR Low Dynamic Range. 8, 9

PBR physically based rendering. 8

UI User Interface. 7, 15, 17, 23, 30, 31, 34, 59

Bibliography

[BN76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547, oct 1976.

[CCng] Omar Cornut and Contributors. Dear imgui: Bloat-free graphical user interface for c++, 2014–ongoing. Still in development, accessed: 10.11.2024.

[DS24a] Jarod Guest Dimitrios Savva. Rosendal plains 1, 2024. `https://polyhaven.com/a/rosendal_plains_1`. Last Accessed: 29.11.2024.

[DS24b] Jarod Guest Dimitrios Savva. Symmetrical garden 02, 2024. `https://polyhaven.com/a/symmetrical_garden_02`. Last Accessed: 29.11.2024.

[Gre86] Ned Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, 1986.

[GZnd] Rico Cilliers Greg Zaal. The sky is on fire, n.d. `https://hdri-haven.com/hdri/beautiful-sunrise-at-coast`. Last Accessed: 29.11.2024.

[ies] Bega luminaire 50975.6k3. `https://ieslibrary.com/browse#ies-007cfb11e343e2f42e3b476be4ab684e`. Last Accessed: 29.11.2024.

[Ill19] Illuminating Engineering Society. *IES LM-63-19: IES Standard File Format for the Electronic Transfer of Photometric Data*. Illuminating Engineering Society, New York, NY, 5th revision edition, 2019.

[Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[KB17] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[Lab] Light Laboratory. Photometric & optical testing. Accessed: 01.12.2024.

[LADL18] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Trans. Graph.*, 37(6), December 2018.

[LHEN+24] Lukas Lipp, David Hahn, Pierre Ecormier-Nocca, Florian Rist, and Michael Wimmer. View-independent adjoint light tracing for lighting design optimization. *ACM Transactions on Graphics*, 43(3):1–16, May 2024.

[LRS97] Charity Lu, Alex Roetter, and Amy Schultz. Types of ray tracing, 1997. Accessed: 10.11.2024.

[MSH84] S. Miller, Magi Synthavision, and C. R. Hoffman. Illumination and reflection maps : Simulated objects in simulated and real environments gene. In *Course Notes for Advanced Computer Graphics Animation, SIGGRAPH 84*, 1984.

[Noc80] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35:773–782, 1980.

[SM24] Jarod Guest Sergej Majboroda. Overcast soil (pure sky), 2024. `https://polyhaven.com/a/overcast_soil_puresky`. Last Accessed: 29.11.2024.

[Uni15] International Telecommunication Union. Recommendation itu-r bt.709-6: Parameter values for the hdtv standards for production and international programme exchange, June 2015. Accessed: 25.11.2024.

[Vea97] Eric Veach. *Robust Monte Carlo Methods for Light Transport Simulation.* PhD thesis, Stanford University, 1997.

[War91] Greg Ward. Real pixels. In *Graphics Gems II*, chapter 2.5, pages 80–83. Academic Press, 1991.

[Zaa24] Greg Zaal. Victoria sunset, 2024. `https://polyhaven.com/a/victoria_sunset` Last Accessed: 29.11.2024.

[Zim99] Paul Zimmons. Spherical, cubic, and parabolic environment mappings, December 1999. Accessed: 29.11.2024.