TU WIEN Informatics

Gradient-based Light Optimization with Variable Light Count

Dynamic Generation and Merging of Light Sources

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

David Köppl
Registration Number 12022493

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, 22nd December, 2023

David Köppl | Michael Wimmer

Erklärung zur Verfassung der Arbeit

David Köppl

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.

Wien, 22. Dezember 2023

David Köppl

Danksagung

Ich möchte meinen Betreuern, Lukas Lipp und David Hahn, für ihre Anleitung, Unterstützung und kreativen Ideen im Verlauf dieser Forschung danken. Ihre Einsichten und Expertise waren entscheidend für die Entstehung dieser Arbeit.

Acknowledgements

I would like to express my gratitude to my advisors, Lukas Lipp and David Hahn, for their guidance, support, and creative ideas throughout the course of this research. Their insights and expertise have been vital in shaping this work.

Kurzfassung

Diese Arbeit konzentriert sich auf Verbesserungen eines interaktiven Beleuchtungsdesign-Ansatzes, der GPU-beschleunigtes Ray-Tracing und einen blickwinkel-unabhängigen Global-Illumination-Solver verwendet. Unser Ziel ist es, ein automatisiertes Lichtdesign für eine Reihe von benutzerdefinierten Beleuchtungszielen in 3D-Szenen zu ermöglichen. Derzeitige Solver sind sehr effektiv, haben jedoch einige Einschränkungen. Sie basieren auf einer anfänglichen Anzahl von Lichtquellen und deren Platzierungen in einer 3D-Szene, was zu unzureichenden Lösungen führen kann, wenn es mehr Zielstellen als Lichtquellen gibt. Andererseits kann die resultierende Lösung suboptimal sein, wenn mehr Lichtquellen als nötig verwendet werden, was zu überlagerten Lichtern führen und die Leistung sowie den Rechenaufwand negativ beeinflussen kann.

Als Reaktion auf diese Einschränkungen untersuchen wir mehrere Strategien, um die Wirksamkeit und Effizienz des Optimierungsalgorithmus zu erhöhen. Wir entwickeln einen dynamischen Ansatz zur Lichtquellengenerierung, der Lichtquellen in der 3D-Szene programmatisch einfügt und entfernt, um eine verfeinerte Lichtplatzierung zu erreichen. Unsere Ergebnisse zeigen, dass dieser spezialisierte Optimierungsansatz zu verbesserten Beleuchtungslösungen im Vergleich zu etablierten Algorithmen führt.

Darüber hinaus implementieren wir eine Technik zum Zusammenführen von Lichtquellen, um das Problem von Lichtquellen mit überlappenden Einflussbereichen zu adressieren. Durch Anwendung linearer Interpolation und Festlegung von Bedingungen für Intensität und Nähe können wir überlappende Lichtquellen so kombinieren, dass der Einfluss auf die Leistung und der Rechenaufwand minimiert werden. Wir ergreifen auch Maßnahmen, um Lichter mit geringem Beitrag zur Szenenbeleuchtung während des Optimierungsprozesses zu entfernen. Die Beweise aus unseren Experimenten legen nahe, dass unser Ansatz, den Lösungsraum zu erweitern und die Lichtquellenplatzierung zu verbessern, zu überlegenen Beleuchtungslösungen für jede gegebene Szene führt.

Abstract

This thesis focuses on improvements for an interactive lighting design approach that utilizes GPU-accelerated ray tracing and a view-independent global illumination solver. Our goal is to enable automated lighting design for a set of user-specified illumination targets in 3D scenes. Current solvers are highly effective but still have some limitations. For instance, they rely on an initial number of light sources and their respective placements in a given 3D scene and this can result in insufficient solutions when there are more target spots than provided light sources. On the other hand, if there are more light sources than needed, the resulting solution can be sub-optimal, leading to superimposed lights that can negatively impact performance and increase computational cost.

In response to the limitations, we investigate several strategies for increasing the effectiveness and efficiency of the optimization algorithm by developing a dynamic light source generation approach that programmatically inserts and removes lights in the 3D scene to achieve a more refined light placement. In our results, we show that our specialized optimization approach, yields improved lighting solutions compared to established algorithms.

Moreover, we also implement a light source merging technique to address the issue of light sources with overlapping areas of influence. By formulating conditions on intensity and proximity and then applying linear interpolation, we can combine overlapping light sources in a way that minimizes performance impact and computational cost. We also take measures to remove lights with a small illumination contribution to the scene during the optimization process. Evidence from our study suggests that our approach of expanding the solution space and improving the light source placement achieves superior lighting solutions for any given scene.

Contents

CHAPTER 1

Introduction

Light plays a central role in computer graphics, influencing not only the aesthetics of a scene but also the mood, depth, and realism. The correct interplay of light and shadow can transform a regular scene into a visual masterpiece or vice versa. However, achieving the desired lighting effect is no simple task. It requires thoughtful placement and adjustment of light sources to illuminate the scene in a way that aligns with the artist's vision or the scene's requirements.

Creating the optimal lighting solution is a task found in various creative and practical areas. In game design and computer graphics, level designers need to place light sources in a realistic way, while setting a specific mood that influences the player's experience and immersion. Similarly, in architectural design, the architects use light for functional and aesthetic illumination, often to accentuate architectural features. In the context of film or theater productions, lights become part of the story. Set designers not only need to make the performers visible but also set the tone of the scene and support the narrative with light. Especially in theaters the difference between a calm and a dramatic set is often only the lighting on stage.

Placing lights at the right spot with the perfect intensity and ideal color, to achieve a desired light effect, is a difficult task. It often involves manual experimentation and fine-tuning. Recent research around this topic led to the creation of Tamashii [LHEN+23], a specialized framework designed for finding ideal lighting solutions for 3D environments, providing the basis for our work. Tamashii stands out for its ability to create lighting constellations according to predefined lighting goals, allowing designers to concentrate on the desired visual outcomes while the framework handles the technical aspects of realizing these effects. However, Tamashii's approach depends on a fixed number of light sources, which, in certain scenarios, can result in sub-optimal lighting solutions and potentially slower performance.

Our research builds on Tamashii and extends its capabilities, with the goal to address these limitations. We focus on an approach that dynamically adjusts the number of light sources in 3D scenes. This flexibility allows for an arbitrary number of lights, thereby generating solutions that are closer to the ideal one. For that, we use the optimization tools provided by the Tamashii framework. These tools can become unstable or ineffective if scene conditions, such as the number of lights, change abruptly. Therefore, we have to be careful in the way we insert or remove lights.

In overcoming this challenge, we developed a light source generation approach that compares the scene's current lighting with the targeted lighting and introduces new lights at the position of largest difference. Moreover, to reduce redundant lights, we present an approach that can remove or merge lights based on their light contribution. While Tamashii supports various types of light sources, such as directional lights, and spotlights, this work solely focuses on point lights, as these are commonly used and can model a wide range of light constellations.

The subsequent chapter 2, delves deeper into the theoretical background of our work. It begins with an overview of optimization and relevant algorithms, followed by a discussion on rendering and inverse rendering. We then further examine Tamashii, and explain concepts like the Inverse-Square Law and the Lambertian Reflectance model, setting the stage for our optimization method.

In chapter 3 we present our method for optimizing lights in detail. It covers the process of finding positions for new lights, how we represent potential new lights, and how we select and visualize them. It also includes the ADAM Dynamic optimization algorithm, which combines traditional optimization with our proposed heuristic methods. We also discuss our approach for cleaning up light sources and applying constraints to the optimization.

Chapter 4 describes the practical aspects of implementing our methods. This includes the use of a Min-Heap data structure for new lights, details about our optimization algorithm, and how we implement our visualization.

In chapter 5 the results of our research are presented. We perform experiments on two different scenes. With each scene, we conduct a performance evaluation against other optimization algorithms and critically analyze the convergence of our approach.

In the last chapter 6, we conclude the thesis by summarizing our key findings and contributions. We also discuss potential areas for future research that can build on the work presented in this thesis.

CHAPTER 2

Background

Before we explain our approach we will quickly touch on the fundamental concepts needed to understand the subsequent chapters in this thesis. We will introduce the process of optimization and how it relates to light design applications in the field of computer graphics. Subsequently, we will explore important optimization algorithms and then go over the framework used to implement our optimization approach.

2.1 Optimization

At its core, optimization is the process of fine-tuning parameters to find the minimum or maximum of a function. This can be observed by businesses that aim to maximize their profits, bees creating the ideal form for their honeycombs, or light rays that minimize their travel distance. In lighting design, the goal of the designer is to form the arrangement of lights so that it aligns with the specific needs of the room or scene. Optimization provides the necessary theory to reach such light constellations systematically.

For optimization to be applicable there needs to be a well-formulated model of the system it should be applied on. There needs to be a single value or a set of values that express the combination of all parameters influencing the system. This value is the objective function. The variables influencing the objective function can also be constrained in some way [Noc06]. For example, the intensity of a light source must not become negative in a light simulation.

2.1.1 Objective Function

Optimization can be expressed with an objective function f which evaluates the objective value we wish to minimize or maximize and the vector x containing the variables that influence our model. In our context, this function is scalar-valued. The optimization problem can then be written as:

$$\min_{x \in R^n} f(x). \tag{2.1}$$

In the context of light optimization, f could measure how close the current lighting is to the target lighting. Luckily optimizing the objective function can be done using tried algorithms.

Our focus will be on gradient-based, first-order optimization algorithms, as they are well suited for light optimization, provided the objective function is smooth and continuous. For cases where the objective function is noisy, meaning the output of the objective function is less predictable, gradient-free methods should provide a more robust approach. Such methods include the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [Han16], which stochastically samples the solution space. Before delving into specific algorithms, it is crucial to understand the concept of a gradient, as it is essential for first-order optimization methods.

2.1.2 Gradient

In the context of optimization and calculus, a gradient represents the slope or the rate of change of a function. Mathematically, the gradient of a function f at a point x is a vector that points in the direction of the steepest ascent of f. This is the direction in which the function increases most rapidly [CZ13].

For functions of a single variable, the gradient is simply the derivative. However, for functions of multiple variables, the gradient is a vector containing all the partial derivatives. For instance, for a function $f(x, y)$, the gradient is given by:

$$\nabla f(x, y) = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]^T. \tag{2.2}$$

The gradient plays a central role in optimization algorithms as it often provides the direction to move to find the function's minimum or maximum value.

2.2 Optimization Algorithms

While there are various algorithms for optimization, they generally start with an initial guess and then refine the parameter vector x to minimize the objective function. The interesting part is how each of these algorithms decides to refine the parameter vector [Noc06].

Let's consider some common approaches: Gradient Descent refines the parameter vector by computing the gradient of the objective function at the current point. The algorithm then takes a step in the direction of the steepest descent and repeats the process at the new point. Newton's method goes further, not only considering the gradient but also the curvature of the objective function, which requires the Hessian matrix. These

methods can converge faster than gradient methods but tend to be more computationally expensive. Quasi-Newton methods work similarly to Newton's method but are cheaper to compute. There are various other approaches such as the conjugate direction method, which is ideal for objective functions that have the form of a quadratic because their search direction is optimal for these problems [CZ13].

In the following section, we will briefly discuss relevant optimization algorithms and the key concepts needed to understand them. We will start with Gradient Descent, which optimizes straightforwardly, ideal to get a basic understanding to build upon. Then we will focus on ADAM [KB14] which is an extension of Gradient Descent and the core of the approach presented in this thesis. Lastly, we will go over L-BFGS [Noc80], which in contrast to the Gradient Descent and ADAM algorithm, represents a Quasi-Newton method. L-BFGS is also used in our approach.

2.2.1 Gradient Descent

Gradient descent is a typical optimization algorithm that works by iteratively adjusting the input parameters according to the direction of the gradient. Each iteration the algorithm computes the gradient of the objective function and updates the parameters by moving in the opposite direction of the gradient, scaled by a constant factor [CZ13]. This factor is known as the step size or learning rate and is often denoted as α. It needs to be chosen carefully because a too-small step size leads to many short iterations and takes longer to converge. On the other hand, a too-large step size can overshoot the minimum and might not converge at all.

A common way to gain the intuition behind this method is to imagine standing on a foggy mountain and wanting to reach the valley at the bottom as fast as possible. The Gradient Descent would look at the immediate area and walk towards the steepest descent. After one step, it would again look where the ground is the steepest and continue. At some point, it should reach the valley, although there is a risk of ending up in a lake or a cave.

2.2.2 ADAM

ADAM [KB14], which stands for Adaptive Moment Estimation, is a newer optimization technique that can be seen as a combination of the AdaGrad and RMSProp algorithms. AdaGrad [JS11] adapts learning rates based on past gradient information, giving more attention to rarely occurring features, but it can suffer from diminishing learning rates over time. RMSProp [DCB15] addresses this by introducing an exponential decay to the accumulated squared gradient, ensuring only recent iterations are considered.

With these foundations, ADAM further improves the approach. Similar to AdaGrad and RMSProp, ADAM adapts the learning rates for individual parameters, making it effective for problems with large data or parameters. The ADAM algorithm updates parameters with a momentum that is calculated as a moving average of past gradients. This has the effect of "smoothing out" the updates and navigating plateaus more effectively.

Additionally, ADAM uses an adaptive learning rate, adjusting the step size for each parameter based on a moving average of the squared gradients. If the variance is high, meaning the gradients change drastically between iterations, the learning rate is decreased, ensuring more stable updates. Conversely, if the variance is low, the learning rate is increased, allowing for faster convergence.

The combination of momentum and adaptive learning rates allows ADAM to optimize faster than regular Gradient Descent. It also can regularly navigate out of shallow local minima which the Gradient Descend would not manage to do.

To go back to the analogy, if Gradient Descent is a person trying to find their way down a foggy mountain by always moving in the steepest direction, ADAM is a person with a GPS without a map and a memory of the most recent steps. This person not only knows the steepest direction but also has an idea of the general direction they should be heading based on their past steps.

2.2.3 L-BFGS

L-BFGS [Noc80], which stands for Limited-memory Broyden-Fletcher-Goldfarb-Shanno, is part of the Quasi-Newton methods. L-BFGS aims to approximate the inverse Hessian matrix, which drives the direction of the optimization. The primary advantage of L-BFGS is that it doesn't store the full Hessian matrix. Instead, it approximates the effect of the inverse Hessian on a vector, simplifying the computations and memory requirements.

Unlike Gradient Descent, which solely relies on the gradient to determine the search direction, L-BFGS takes the curvature of the objective function into account. This can significantly improve the convergence rate, especially for convex functions. It can be proven that Newton's method can solve a quadratic problem of n variables in a single step [CZ13]. The way it approximates the inverse Hessian gives the algorithm its "limited-memory" property. It only stores the most recent gradients and, using a two-loop recursion, computes the matrix-vector product of the approximated inverse Hessian. An important aspect of L-BFGS is its line search. This process involves iteratively testing different step sizes along the direction suggested by the approximated matrix, to find the step that optimally reduces the objective function.

In our analogy, L-BFGS is like having a topographical map of the mountain. This map doesn't show every single detail (due to the limited memory), but it provides enough information about the surface to make informed decisions about the best path downwards. It is worth noting that the L-BFGS algorithm may not perform well with noisy objective functions, which is where algorithms like ADAM, which are more robust to noise, have an advantage."

Now we have introduced the concept of optimization and a few relevant algorithms, two of which are part of our proposed approach. In the next section, introduce the concept of rendering and inverse rendering and take a look at the inverse rendering framework we used for the implementation of our approach.

2.3 Rendering and Inverse Rendering

To understand the foundation of our work, we need to discuss the theory behind forward rendering and inverse rendering.

The process of rendering aims to generate an image from a given scene description, including lights, geometry, and textures. This can be done on most computers with a graphics processing unit (GPU). In the field of computer graphics there exist many approaches for rendering images, especially photo-realistic ones. However, reverting this process, namely to extract scene parameters from a rendered image, is a much harder problem, which is generally referred to as inverse rendering. If we think of the render process as a function $g(x)$ that takes an input vector x, consisting of lights, geometry, materials, and so forth, and transforms it into an image y, we can express the rendering process as follows:

$$y = g(x). \tag{2.3}$$

An inverse function can then be defined by:

$$x = g^{-1}(y). \tag{2.4}$$

However, this inverse function often cannot be calculated directly because the rendering process is very complex. One approach to solving inverse rendering problems is through gradient-based optimization. The optimization objective can be formulated as:

$$x^* = \arg\min \frac{1}{2} \parallel g(\tilde{x}) - y \parallel^2, \tag{2.5}$$

where x^* represents the optimal solution, $\tilde{x}$ is the variable being optimized, and y is the target image. The idea here is to minimize the squared difference between the rendered image $g(\tilde{x})$ and the target image y. Ideally, if $g(\tilde{x}) - y$ becomes 0, we have found a perfect solution where $g(\tilde{x}) = y$. However, in practice, achieving a difference of exactly zero is often unattainable due to the complexities and potential non-linearities in the rendering function g. Therefore, the aim is to find a solution that closely approximates the target image [ZJL20].

The rendering process can consist of rasterizing triangles or tracing light rays. But because differentiating the rasterization step is not directly possible as evidenced by PyTorch 3D [RRN+20] and SoftRas [LCLL19], ray-traced rendering is often preferred for solving inverse rendering problems. With the increase of GPU-accelerated ray-tracing, new inverse rendering frameworks such as Redner, Mitsuba 2, and Mitsuba 3 have emerged [LADL18, NDVZJ19, JSRV22]. However, none of them specifically focus on the subject of light optimization like Tamashii [LHEN+23].

2.4 Tamashii - Differentiable Light Tracer

Our optimization strategy was developed using Tamashii [LHEN+23], a differentiable light tracer. It integrates both rasterization and ray-tracing pipelines and provides implementations for optimization algorithms such as ADAM and L-BFGS, that enable us to optimize different light sources.

2.4.1 Implementation

Tamashii [LHEN+23] provides a solid foundation for researchers to experiment with diverse optimization strategies. It is capable of importing and exporting a variety of 3D modeling formats and supports common light sources, such as point lights and spotlights.

Like many other render frameworks, Tamashii is written in C++ and is compatible across both Windows and Linux platforms. The rendering module accesses the Vulkan [Vk2] application programming interface (API) to leverage the GPU's hardware acceleration for rendering the scene. Tamashii also provides a user interface (UI) that is built using the immediate graphics user interface (ImGUI) library [Img].

2.4.2 Light Target

Because Tamashii aims to be view-independent it uses spatial data structures instead of rendered images for optimizing lights. These data structures include directional and non-directional camera-free radiance fields. Each field holds both a current light solution and a target light solution, and the objective of the optimization algorithm is to reduce the differences between these respective solutions to a minimum. For simplicity we focus on diffuse surfaces, therefore we only need the non-directional radiance field. Extensions that take the directional one into account, are left for future work.

Therefore, to use Tamashii, a light target needs to be defined. For each vertex in the scene, the user can add a target value, together with a masking weight. Light targets can be intuitively drawn on the scene's geometry using an editor, or externally via the vertex color attribute. The light target's quantitative value represents radiance.

Radiance measures the light either reflected from or emitted from a specific area and falling in the direction of a solid angle. It is often used in radiometry, which is the science of measuring electromagnetic radiation, such as visible light [NRH+77].

Additionally, each vertex contains a calculated radiance value, which is derived from solving the rendering equation and depends on the current lighting configuration in the scene. This radiance value represents the amount of light the vertex receives from all the scene's lights. The optimization algorithms can then compare the target and calculated values via the objective function and adjust the parameters accordingly.

If a vertex has a light target of 1 unit and a masking weight of 1, the optimizer will try its best to increase or decrease the radiance at this vertex until its actual value is 1 unit. If the masking weight is 0, the vertex would be ignored by the algorithm.

Before we can start discussing our method for light optimization, we need to introduce two fundamental concepts in computer graphics, the inverse-square law and the Lambertian reflectance model. In a later section, these concepts will enable us to calculate the starting positions of light sources for our approach.

2.5 Inverse-Square Law

The inverse-square law is a fundamental principle in physics that describes how the intensity of a physical quantity, such as light, diminishes with the distance from a point source. Specifically, the intensity decreases in proportion to the square of the distance from the source. This principle is important in scenarios where a point source emits energy uniformly in all directions, such as a point light in computer graphics.

The law can be expressed as:

$$I = \frac{P}{4\pi r^2}. \tag{2.6}$$

Where I is the intensity of the light, P is the power of the light source, and r is the distance from the light source. The term 4π accounts for the total solid angle over which the light is emitted, assuming a uniform distribution over the entire unit sphere surrounding the point source.

2.6 Lambertian Reflectance Model

The Lambertian reflectance model, often referred to as Lambert's cosine law, describes how light interacts with diffuse surfaces. The model assumes that light is reflected from surfaces uniformly in all directions, making the surface appear equally bright from any viewing angle. The amount of light reflected is proportional to the cosine of the angle between the incident light and the surface normal [AvD13]. Mathematically, the Lambertian reflectance is given by:

$$L = \frac{\rho}{\pi} I \cos\theta. \tag{2.7}$$

Where L is the reflected radiance or color, ρ is the albedo or surface color, I is the irradiance or power per unit area, and θ is the angle between incident light and surface normal.

In the context of our work, understanding the Lambertian reflectance model is essential as it provides a basis for how light interacts with surfaces, especially when considering diffuse reflection. Notably, the Lambertian reflectance model describes ideal diffuse reflection and does not account for other types of reflections, such as specular reflection seen on glossy surfaces. Therefore, it only approximates real-world light interactions, as real surfaces never feature purely diffuse reflections.

For our optimization approach, we need to calculate a reasonable starting point for the light sources so that their position and intensity can be fine-tuned by the optimizers afterward. If the starting point is too far from the ideal position, the optimization process may slow down significantly or even become inefficient. By accepting a small error from ignoring specular light and subsurface scattering contributions, we obtain a fast approximation that is sufficiently close for most materials.

With that, we explained the necessary concepts to understand the methodologies presented in this thesis. In the next chapter, we will go over the theory of our optimization approach.

CHAPTER 3

Method

With the necessary concepts established, we can look into our approach for light optimization. To dynamically adjust the number of light sources during the optimization process, we essentially had to solve a non-continuous optimization problem. This is achieved by combining traditional optimization algorithms with heuristic approaches, designed to insert and delete light sources without creating noticeable jumps in the objective function, which would otherwise compromise the optimization algorithms we use.

The strategy used for light optimization with a variable number of light sources is composed of several essential methods. This chapter looks into these methods, explaining the motivation and processes behind them. We begin by discussing the insertion of lights, detailing how we choose positions for new lights, understanding the notion of light candidates, and outlining the procedure employed to select one of these candidates. We then explain the visualization of light candidates and describe how we utilize available optimization algorithms in our optimization strategy. Finally, we address the issue of eliminating unnecessary lights and describe the methodology used to merge light sources.

3.1 Determining Light Position

Before we start inserting new lights into the scene, we first have to decide if the scene needs additional lights and if that's the case, where in the scene those lights need to be positioned. In this section, we present our approach to these challenges, utilizing scene geometry and radiance data to find suitable positions for new lights.

As an introductory example, consider a room with a table and a single source of light. Our task is to effectively light the table using this light source. By employing a differentiable renderer alongside an optimization algorithm, we can mark the table as a target and let the optimizer minimize the objective function until the light is at a satisfying position.

In scenarios where the initial light source is nonexistent, or when we have two rooms each with a table but only one light source, we must insert at least one or two additional lights to enable an acceptable solution. Both situations require a method for choosing an appropriate position to insert a light.

3.1.1 Radiance Difference

Our scene includes data on the vertices of the scene geometry, each containing information about its position and normal vector, which we will use later to derive positions for new lights. Radiance data is also available for every vertex, encompassing the target radiance value and the currently rendered radiance. These radiance values exist for red, green, and blue (RGB) channels. Given a defined lighting target, we can analyze each vertex's radiance values. This involves calculating the difference between the target and current radiance for every color channel, and then converting the resulting differences per channel ΔR, ΔG, and ΔB into a greyscale value representing the radiance difference $\Delta radiance$.

The radiance difference $\Delta radiance$ serves as a grade for the light position selection. This is a signed value, meaning it can be negative, indicating the solution is brighter than the target. If we were to place a light source near a vertex with a radiance difference $\Delta radiance$ close to zero, the overall solution would likely not improve. Conversely, inserting a light source near a vertex with a large positive radiance difference would likely improve the solution.

The greyscale conversion allows us to simplify our first optimization phase by letting us focus light placement and intensity without the complexity of color. Our strategy is to start with a neutral white colored light, optimize the position and intensity, and then adjust the color in a later optimization phase.

The color conversion can be expressed by:

$$\Delta\text{radiance} = \frac{\Delta R + \Delta G + \Delta B}{3}. \tag{3.1}$$

The red color difference for a given vertex, and similar for the other channels, can be obtained by $\Delta R = R_{target} - R_{actual}$. Where R_{target} is the red radiance channel of the target and R_{actual} is the currently rendered, actual red radiance channel. Instead of a greyscale conversion, the color could be converted to a luminance value, which would better represent the way humans see color [Ir15, Fai13]. Our optimizer works without any color bias therefore we choose to only use a greyscale value.

Let's recall that each vertex contains a light target and masking weight. The masking weight dictates the importance of the vertex during the optimization. Therefore we also need to consider the masking weight in our selection process. We achieve this by multiplying the radiance difference with the vertex's masking weight. A vertex is considered for further selection only if this calculated value exceeds a user-defined threshold. This threshold is controllable via a GUI slider. Without this step, the number

of positions that require consideration could become overwhelmingly large, because we would need to look at vertices that are less significant for the light solution. Especially in scenes with many vertices, this can become a problem. This threshold also enables the user to re-execute the optimization with a lower threshold, if the solution from the previous execution was unsatisfactory.

To further avoid problems in larger scenes we limited the number of selected positions per iteration to 2048. For that, a min-heap data structure maintains the top 2048 vertices, based on their radiance difference as described in section 4.1.

3.1.2 Distance along Normal Vector

We now have a set of vertices in 3D space, where inserting a light source nearby, should improve the lighting solution. Placing a light directly at a vertex would likely position it inside a 3D model. From there it cannot contribute to our light solution, and our optimization algorithms won't be able to adjust the light's intensity or position.

Fortunately, each vertex has a normal vector, which in the context of 3D models, points in the outside direction, away from the model's surface. So by positioning the light in the direction of the normal vector, we assume it is placed outside of the corresponding 3D model. Again our goal is to place the light close to the vertex, to improve our lighting solution. However, placing it too close could result in an overlit vertex, leading to steep gradients and a potentially unstable optimization.

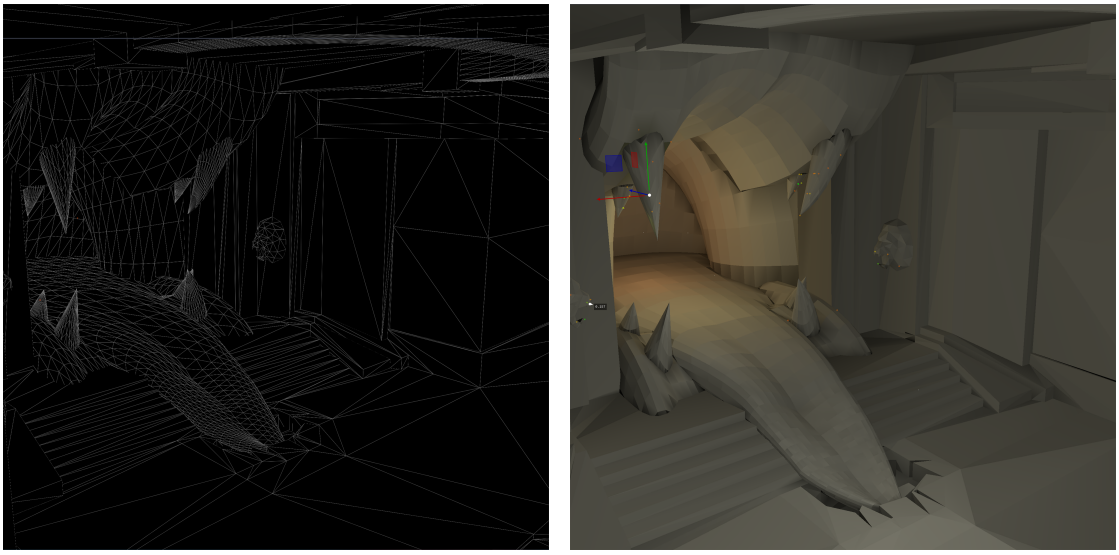
Positioning the light too far from the vertex might diminish its intended influence on that point, and inadvertently affect other vertices or even result in the light being placed inside another 3D model. Therefore, for our approach to be functional, we need to carefully choose our distance along the normal vector of the vertex. This section explains how we determine this distance.

For each vertex, we calculate the distance along the normal vector by adapting the inverse-square law from Eq. 2.6 and the Lambertian reflectance model from Eq. 2.7 for our use case. Note that we place lights only along the normal vector, which implies their light will hit perpendicular to the surface. Therefore, in the Lambertian reflectance model the angle θ will be 0 and as $\cos(0) = 1$ the term $\cos(\theta)$ is omitted. If we now also substitute the irradiance I with the inverse-square law from Eq. 2.6, we obtain:

$$L = \frac{\rho P}{4\pi^2 r^2}. \tag{3.2}$$

This equation shows the dependence of the observed brightness L on the power of the light source P, the distance from the light source r, and the surface color ρ. Now we can solve for the distance r:

$$r = \sqrt{\frac{\rho P}{4\pi^2 L}}. \tag{3.3}$$



(a) Light placement with Eq. 3.3 (b) Light placement with Eq. 3.4

Figure 3.1: Comparison of running the Quake experiment from section 5.1.2 with (a) the derived formula and (b) our adjusted formula, with $C = 4\pi$. Both screenshots were taken after the fifth optimization iteration. Notice that in the left image, the optimizer could not adjust a single light source, resulting in a dark scene. A wireframe overlay was added for visibility.

The surface color ρ can either be sampled from the vertex color or a texture using the vertex UV coordinates. In our initial experiments, especially in complex scenes like the one described in section 5.1.2, we observed that the original distance formula placed lights too close to the geometry as depicted in Fig. 3.1. This reduced the effectiveness of our optimization approach. To correct this, we modified the distance formula by adding a scaling factor C, resulting in the equation:

$$r = \sqrt{\frac{\rho P}{4\pi^2 L}\, C}. \tag{3.4}$$

We determined an optimal value for C by increasing it with multiples of π until the lights were positioned at a safe distance. For our purposes, we found that setting $C = 4\pi$ placed the light sufficiently far away. In a later step, the light's position as well as its intensity will be refined further, correcting the errors of our approximation. Now we can define the light position p_{light} based on the vertex position p_{vertex} by:

$$p_{\text{light}} = p_{\text{vertex}} + r \cdot v_{\text{normal}}, \tag{3.5}$$

where r is the distance from equation 3.4 and v_{normal} is the vertex normal vector.

Additionally, to ensure we place the light outside of 3D models, we cast a ray from the vertex in the direction of the normal vector, recording any intersections with scene geometry along the way. If this ray hits something, we compare the distance from the vertex to the closest hit-point v_{hit} with the distance r calculated by Eq. 3.4. In the case that the length of v_{hit} is shorter than the distance r, the light would likely be placed inside a 3D model. To prevent this, we need to position the light somewhere between the vertex and the hit-point. Based on our observation, where such ray intersections typically happened with ceilings, positioning the light at half the length of v_{hit} led to a consistent optimization process. Our light position p_{light} in this case is calculated by:

$$p_{\text{light}} = p_{\text{vertex}} + \frac{1}{2} v_{\text{hit}}. \tag{3.6}$$

This approach uses available vertex data to find light positions that will contribute most significantly to the illumination solution. By taking the inverse-square law and the Lambertian reflectance into account, we ensure that lights are placed at a distance where they can be easily optimized further as described in section 3.3.

3.1.3 Initial Light Intensity

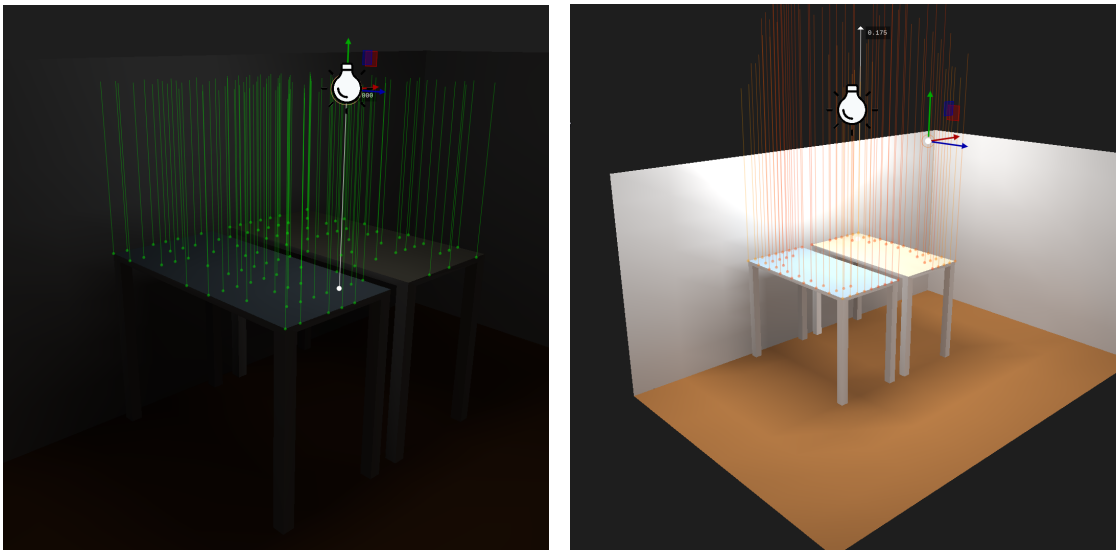
When we introduce a new light source into the scene, it needs to have some intensity. One might consider setting the intensity of the new light to zero, meaning it adds no illumination to the scene. An issue arises from how gradients are computed in the optimization algorithms. A light that fails to contribute significantly to the lighting solution could have a vanishing gradient, consisting of zeros. For such lights, the optimization algorithm receives no guidance on how to adjust the light's parameters to improve the lighting, as changes seem to have no effect.

To avoid this issue, new lights are introduced with a non-zero intensity. However, introducing a light with a too high intensity can lead to abrupt changes in the scene's illumination. These sudden changes can disrupt the gradient landscape that the optimization algorithm relies on, potentially causing erratic behavior in the optimization trajectory. This behavior can make it difficult for the algorithm to quickly navigate to the optimal solution. So we need to find a balanced initial intensity that is high enough to avoid vanishing gradients, and low enough to prevent sharp jumps in radiance.

In our approach, we choose to set the initial intensity to 3.14 watts. By inserting lights with such a low wattage, the radiance in the scene is hardly increased, keeping the optimization stable. Of course, our initial guess is too low to provide enough light for most applications. Therefore, after inserting a new light, we refine its intensity using the L-BFGS algorithm as described in section 3.3.

3.1.4 Light Candidate Selection

Until now, we identify underlit vertices in a scene and determine potential light positions, where placing a light would improve the lighting solution. One might consider the



(a) Inserted light source (b) Optimized light source

Figure 3.2: Comparison of light candidates at the start of an optimization, with one pre-existing light. In the left Figure, the radiance difference is virtually identical across all light candidates. In the right Figure, the light is notably brighter, resulting in different calculated light candidates.

straightforward approach of simultaneously placing a light at every potential light position. However, the number of potential light positions can vary, depending on the 3D scene and light target. In our case, this number can grow up to 2048. Adding such a large amount of lights into the optimization process is computationally challenging. Furthermore, for most real-world use cases the number of lights should be as low as possible, to stay cost-effective. With that in mind, we need to ensure that each added light has a valuable contribution to the scene lighting. This is achieved by only adding one light at a time. As a side effect the optimization process is more predictable and diagnosing issues is easier.

The process of selecting the next light from the set of potential light positions is the subject of this section. For simplicity, we introduce the notion of a light candidate, which represents a potential light source. A light candidate includes data for comparing it against others, which is vital in the selection process. It also includes information used to visualize it, which becomes relevant in section 3.2

A simple approach to selecting a light candidate might start by looking at the candidate's radiance difference. This value represents how much light is missing at a given vertex. So the light candidate with the largest radiance difference should be the best choice for the next light to be added, as it is supposed to improve our lighting solution the most. However, selecting the candidates in this way leads to some complications in the optimization.

For instance, consider a dark room where a table serves as our light target. Calculating the radiance difference for each light candidate, without any initial light in the scene, will result in identical values. This result can be seen in Figure 3.2a. The identical radiance differences prove to be challenging when we want to compare them. As there is no practical order, the "best" candidate often defaults to the first one we calculate. In our example, this is typically a light candidate from a vertex at a table corner, which can be problematic for several reasons. Firstly, if the table is near a wall, the light source could end up near the wall, either becoming stuck in the wall's geometry or expelled from the scene's boundary – in both cases, rendering it ineffective. Additionally, should it survive and stay in the optimization phase, the algorithm would still need to position the light source above the table's center to reach an optimal solution. Therefore, it would be more efficient to initially position the light nearer to the center of the table.

After one optimization iteration with the first light source, the process would continually place lights at the target's edges and move them toward the center. This pattern as shown in Figure 3.2b is observed because the radiance differences at the edges of the table are usually greater than at the center, where sufficient light is already present, assuming a uniform light target on the table. This further explains why it might be less effective to always pick the light candidate with the highest radiance difference.

To avoid these issues, our selection process considers the radiance difference of a given light candidate and selects with an element of randomness. We adopt a weighted random selection approach for choosing the next light candidate. In genetic algorithms, this is also known as fitness proportionate selection or roulette wheel selection [Mel99]. This selection process assigns each light candidate a probability of being selected, based on the candidate's radiance difference.

Mathematically this can be expressed as follows. Given a set of n light candidates with an associated fitness value f_i for $i = 1, 2, ..., n$, the probability Pr_i of selecting the i-th candidate is:

$$\mathrm{Pr}_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \tag{3.7}$$

For our purposes, f_i is the radiance difference value of the candidate and the term $\sum_{j=1}^{n} f_j$ is the sum of all radiance differences, ensuring that all probabilities sum up to 1. It is important to note that the radiance difference is generally signed, therefore we only consider values above a certain positive threshold as candidates. With each light candidate assigned a probability value, we can sample a single candidate and flag it as selected. In the next iterations, the selected candidate then gets inserted into the scene, based on the position calculated in section 3.1.2. This strategy prevents premature convergence on local minima in the solution space by not always opting for the highest-ranking candidate. For comparison purposes, our GUI offers an option to switch between probability-based selection and the "best" candidate selection.

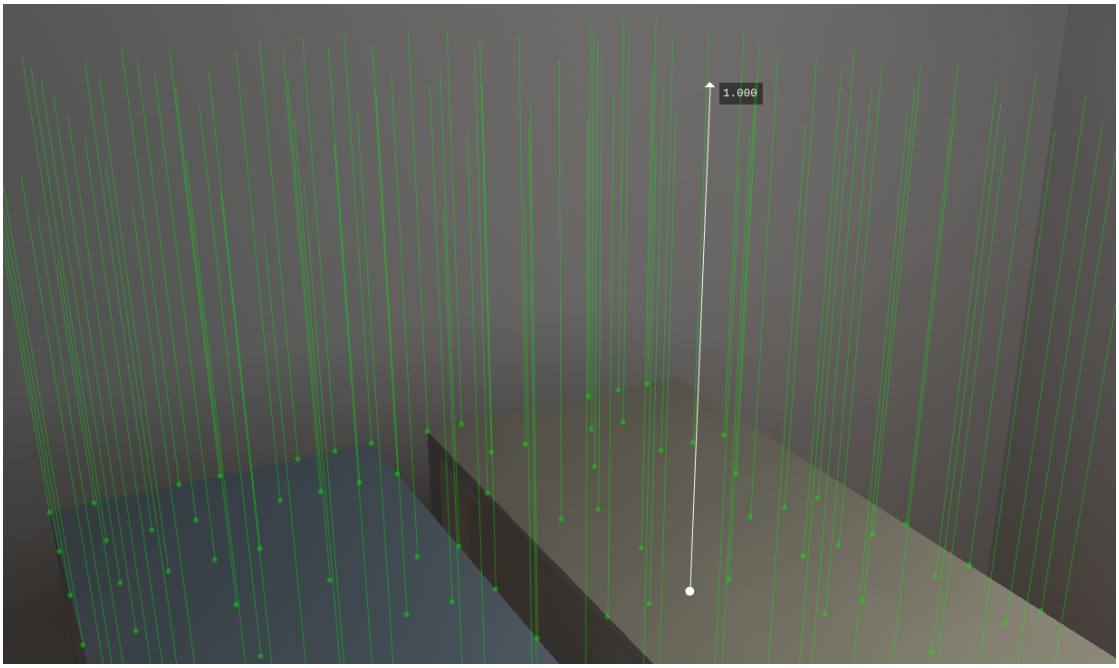


Figure 3.3: Light candidates before inserting a light source. A light was added afterward to illuminate the scene. The selected candidate is highlighted in white.

3.2 Light Candidates Visualization

In each iteration of the optimization process, several light candidates can be found. As previously discussed, the number of candidates could reach up to 2048. Each candidate includes vectors, a radiance difference value, and a probability. Identifying inefficiencies in the code or understanding and adjusting the weighted random sampling strategy, can become difficult when only dealing with the candidate's data. Therefore we simplify the abstract information by visualizing it during the optimization.

In this approach, we represent a light candidate as an on-screen arrow as displayed in Fig. 3.3 and Fig 3.4. The arrow's origin corresponds to the coordinates of the vertex position, while the tip signifies the calculated light position. Additionally, the radiance difference is encoded in the arrow's color.

3.2.1 Uniform Sampling for Visualization

The final component in our description of a light candidate relates to the sampling process we use for visualization, as described in section 3.2. The purpose behind sampling the candidates is to prevent the visualization from becoming excessively cluttered. Without this step, we could potentially be dealing with as many as 2048 visible light candidates. This can become overwhelming in larger scenes, particularly when meshes with a high polygon count are involved. To avoid this, we sample a subset of up to 100 candidates,

flagging them as sampled, and then only visualize the flagged candidates.

The method used here is essentially a form of uniform sampling from a finite set. By denoting the total number of light candidates N, and the number of samples to be drawn k, we can describe this process:

1. Each light candidate i has an index $i \in \{0, 1, 2, ..., N-1\}$
2. We generate a random permutation of the indices, denoted as $\pi(i)$, where π is the permutation function.
3. For each $j \in \{0, 1, ..., k-1\}$, we mark the light candidate with index $\pi(j)$ as sampled.

Additionally, we also marked the light candidate which was selected by the weighted random sampling as sampled. This ensures the light candidate that gets inserted next, will always be part of the visualization.

With this method, we reduce the amount of visible clutter, while still presenting enough information to understand the distribution of light candidates. The difference can be seen in Figure 3.4

Up until now we can find suitable light positions and build light candidates, then randomly select a candidate based on a fitness value and provide data for visualizing these candidates. Before we delve into the optimization strategy, we will briefly touch on how the light candidates are visualized.

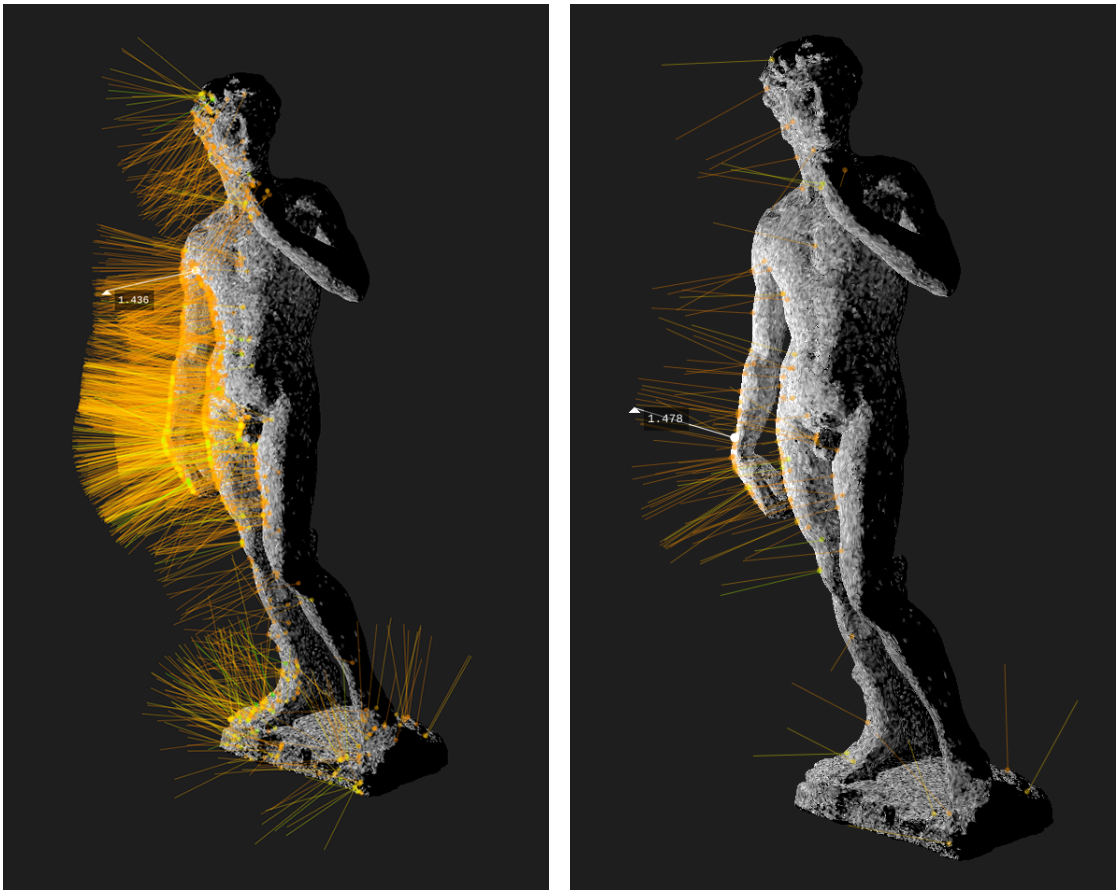
3.2.2 Color Mapping

In our visual representation, we correlate the smallest found radiance difference *minRadiance* to the color red, and the largest found difference *maxRadiance* to green. In the hue, saturation, value (HSV) color scheme, red is assigned a hue of 0° while green corresponds to a hue of 120° [AvD13]. Thus, the radiance difference $\Delta radiance$ can be mapped to the hue H as follows:

$$H = 120° \cdot \frac{\Delta\text{radiance} - \text{minRadiance}}{\text{maxRadiance} - \text{minRadiance}}. \tag{3.8}$$

To highlight the light candidate selected through the weighted random sampling, we render its arrow with full opacity, and the remaining arrows are shown slightly transparent. Additionally, the selected candidate's radiance difference is displayed above its arrow as shown in Figure 3.3.

By visualizing the light candidates as arrows from vertex position to light position, the spatial relation between these two becomes apparent. The color mapping then helps identify the areas of high radiance difference (green) and low radiance difference (red).



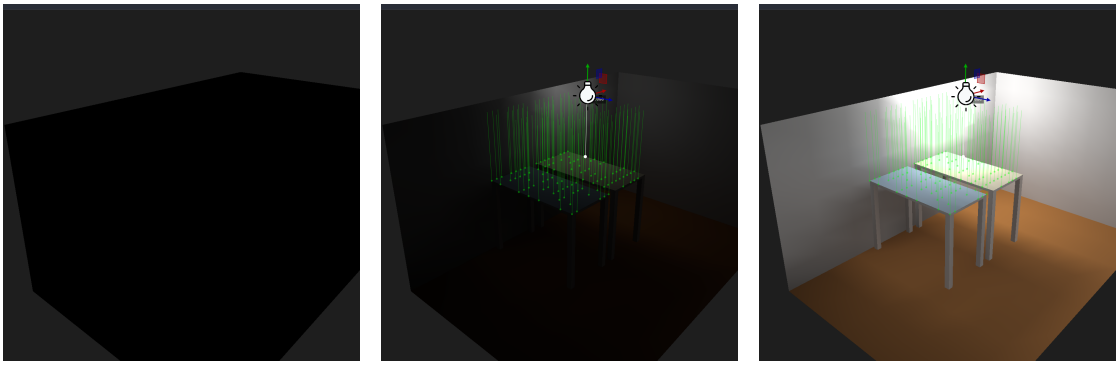
(a) Without random sampling, k=2048 (b) With random sampling, k=100

Figure 3.4: Light candidate visualization with a high poly model of the David statue. The left image visualizes all light candidates that were found, while the right image only shows a subset of them. The shading on the model indicates the light targets.

Displaying the selected light candidate's radiance difference provides a quantitative understanding of the current state of the optimization and it allows the user to track its progress by observing how the radiance difference decreases with each iteration.

3.3 ADAM Dynamic - Optimization Algorithm

Let's recall that current light optimization approaches, including Tamashii [LHEN+23], only allow a fixed number of lights for optimization. Our goal is to overcome this limitation by dynamically adjusting the number of lights in a scene. Although we have presented methods that can determine light candidates, we still need to carefully integrate these candidates into the optimization process to achieve this goal. In this section, we combine our heuristic methods with existing optimization algorithms, presenting the core



(a) initial dark room (b) inserted light (c) optimized intensity

Figure 3.5: Phase 1 of the optimization - A light source is placed in a dark room (a) and then its intensity is optimized from the initial 3.14 W (b) to a sufficient level (c). Note that the light candidate visualization only updates at the start of an optimization step, leaving the arrows here green.

of our thesis - a light optimization algorithm, which we call ADAM Dynamic.

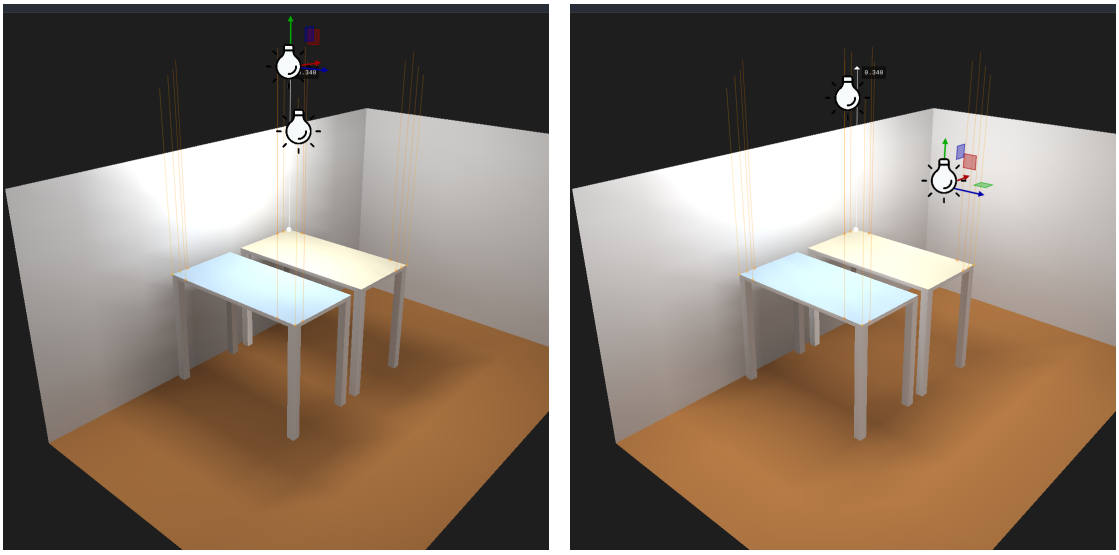
As the name suggests, the regular ADAM algorithm [KB14] provides the basis for our approach. The ADAM optimizer refines the lighting solution through a series of iterations, although with a fixed light count. To increase the light count, we could insert a new light into the scene and execute the ADAM optimizer for a few iterations. Then afterward check if another light is needed and repeat the process. Ideally, the optimizer would always adjust the parameters of all lights, including the newly added one, to match the target lighting. However, our approach uses an additional optimization step, after inserting a new light, splitting the optimization into two phases.

3.3.1 Phase 1 - Initial Light Intensity Optimization

As mentioned in section 3.1.3, new lights initially have a low intensity and contribute little to the lighting solution. This initial low intensity can lead the optimization algorithm to prioritize adjustments to existing, more impactful lights, potentially neglecting the newly added ones. Consequently, without an initial boost in intensity, these new lights might remain ineffective.

To address this, we implement an intensity optimization step immediately after introducing a new light. This step focuses on increasing the intensity of the new light source as depicted in Fig. 3.5. During this process, we maintain the light's initial position and ensure that other existing lights in the scene remain unaffected. This targeted adjustment should help integrate the new light into the subsequent optimization process.

In this step, we employ the L-BFGS optimizer, instead of the ADAM optimizer, which is used in the second phase of the process. The reason for this choice stems from the characteristics of the L-BFGS. This phase only has one optimization parameter, the new



(a) adding a second light (b) all lights optimized

Figure 3.6: Phase 2 of the optimization - The process of optimizing a second light, after its intensity is optimized. Both lights change their position and intensities.

light's intensity. The intensity can either be too dim or too bright, with an optimal value somewhere in between. Therefore, we expect the objective function to be convex. For such functions the L-BFGS algorithm should converge to the optimal intensity level within few iterations. With the use of its line search the algorithm should ensure that it doesn't overshoot the minimum, leading to precise adjustments.

By implementing an intensity optimization phase, we increase the light contribution of these new lights, enabling seamless integration into the scene. With the proper light intensity, our optimization approach can proceed with the next phase.

3.3.2 Phase 2 - Full Scene Light Optimization

Now that we added a light and increased its intensity, we can start with the main optimization phase. In this phase, our goal is to achieve a lighting solution that aligns closely with the given light target, ensuring that the newly added light contributes to an improved lighting solution.

To better understand the motivation behind this phase, let's consider an example: an office scene illuminated by a single light above a table. This light might cast an uneven illumination across the table's surface. By introducing a second light near the table, we aim to balance the light distribution. The goal here is not just to add more light, but to adjust the light sources for improved lighting. A similar process is pictured in Fig. 3.6

In this phase, the optimizer is allowed to update both the intensities and the positions of all light sources in the scene. In some cases, the colors of the lights need to be

adjusted as well, which may also happen in this phase. Because we expect this to be a multi-dimensional optimization problem, with a less predictable objective function, we utilize the ADAM optimizer. Ideally, the result of this phase is the best solution that the ADAM optimizer could find, given the number of lights. Our optimization approach can take multiple iterations, each adding a light and executing the two-phase optimization. We touch on these iterations in the next section.

3.3.3 Concept of Effective Iteration

Our approach executes the two-phase optimization at least once. In the first phase, the intensity optimization with the L-BFGS algorithm consists of up to 5 iterations. Each of these iterations may require multiple objective function evaluations, depending on how the line search progresses. Note that, the number of search steps in the L-BFGS implementation is capped to 15, meaning there is a theoretical upper bound on the number of objective function evaluations. The limit of 5 iterations suffices since it only needs to adjust the intensity of a newly introduced light to a usable range. Despite this limit, the L-BGFS optimizer often surpasses these evaluations if it can further improve the objective function.

The second phase includes up to 80 iterations with the ADAM optimizer, which might be low for certain applications. However, given that the ADAM optimizer is likely to be executed multiple times, it's important to consider the cumulative impact of these repeated iterations. In practice, increasing the iteration limit could potentially lead to diminishing returns in terms of solution improvement versus computational cost, as demonstrated by the data in Table 3.1 Additionally, our results suggest that the ADAM iteration limit could likely be lowered with each new light.

Iteration Limit	**Normalized Objective Value**
20	3.17%
40	3.07%
80	2.72%
160	2.69%
320	2.65%

Table 3.1: Final objective values for varying ADAM iteration limits in the small office scene. Objective values are normalized based on the objective value at the start of the optimization

To avoid confusion between iterations of the first phase, iterations of the second phase, and iterations of our algorithm, we introduce the term "effective iteration". One effective iteration corresponds to the ADAM Dynamic approach going through both phases once. Compared to the individual optimization phases, one effective iteration would be equal to 85 iterations.

In each effective iteration, we search for a light candidate and introduce it as a light source into the scene. Then in phase one, optimize the light's intensity and in phase

two optimize all lights in the scene. This happens multiple times until the process can terminate. Ideally, it terminates when it can't find more light candidates, meaning there are no vertices with a radiance difference exceeding the user-defined threshold. However, in some cases the solution might not improve between effective iterations, opening up the possibility of an endless loop. To prevent this, the number of effective iterations is capped at a user-specified amount.

With these steps, we conclude our main optimization algorithm. This process is designed to dynamically improve light positions and intensities within a scene, making it possible to programmatically add lights when needed. It should be noted that adding a light might not always result in an improved lighting solution. So our approach still requires a way to reduce the number of lights. In the next section, we discuss how our algorithm deals with unnecessary lights.

3.4 Cleaning Up Light Sources

With each added light source the cost of rendering a frame increases. So does the complexity of the optimization. While we take great care that we select and insert lights in a way that improves the objective function, there are instances when an added light may not significantly contribute to the scene's illumination. To keep the optimization efficient, we need to consider these cases and either remove the light source or merge it with another.

3.4.1 Merging Light Sources

To simplify lighting, we can merge less intense lights into brighter ones. Because we only use point lights in our approach, merging can be done solely using the light's intensities and positions. For other types of light sources such as spotlights, there are more properties to consider, such as the light's orientation, and inner and outer cones. One could go further and try to merge different types of light sources, in which case the process for merging quickly becomes very complex. Because of that and the fact that our optimization was designed with point lights in mind, we will not address the merging process for other types of light sources in this thesis.

The first step of this process involves categorizing all the light sources in the scene based on their intensity. Lights are considered less intense or "dim" if their power is below a certain threshold, and "bright" if it exceeds this threshold. The user can adjust this value via the GUI, with the default setting being 10 watts.

Once we've categorized the lights based on their intensity, we examine each dim light to find the nearest bright light by comparing their respective positions. When merging two lights, we utilize linear interpolation, in our case, interpolating the position based on the intensities of the lights. Thus, the position of the merged light is a weighted average of the original lights' positions, the weights corresponding to the ratio of each light's intensity to the total intensity.

Let's say we want to merge two point lights: $light_1$ and $light_2$, then the intensity ratio r is calculated as:

$$r = \frac{I_1}{I_1 + I_2}, \tag{3.9}$$

where I_1 is the intensity of $light_1$ and I_2 is the intensity of $light_2$. The merged position P_m is then calculated as:

$$P_m = r \cdot P_1 + (1 - r) \cdot P_2, \tag{3.10}$$

where P_1 is the position of $light_1$ and P_2 is the position of $light_2$. Finally, the merged intensity I_m is just the sum of the original intensities:

$$I_m = I_1 + I_2. \tag{3.11}$$

After these calculations, we can update the position and intensity of $light_1$ and delete $light_2$. For future work, it might be beneficial to consider the distance when merging the intensities, as the overall lighting changes only slightly if the intensities of two close lights are merged.

To gain greater control over the merging process, the user can specify a distance range within which light sources can be merged. In larger scenes with multiple distinct targets, it might be preferable to only merge lights within the immediate vicinity of each target to avoid eliminating light sources positioned near other targets. This range allows the user to account for such situations.

The threshold for distinguishing dim and bright lights provides a sort of minimal wattage setting. Ideally, all lights smaller than the threshold will be merged and removed from the scene. Given that light bulbs generally come in discrete powers, this feature can be helpful in real-world lighting design applications where a minimum wattage may be necessary.

3.4.2 Removing Low-Intensity Lights

Even after light merging, there might still be some low-intensity light sources remaining. This could occur if there are no bright lights available to merge with. Moreover, some lights might not even be eligible for the optimization process, for instance, if they are positioned out of bounds. To address these issues, we remove light sources after each iteration if they meet certain criteria.

As previously stated, lights are initially added with an intensity value of 3.14 watts. Given that a solution generally requires more than this, we can eliminate all lights that maintain an intensity value equal to or lower than 3.14 watts. Lights of such low intensity

have a minimal impact on the overall lighting solution. With the render engine still needing to trace the same amount of light paths as it would with more intense lights, the removal of such lights should improve the render performance and the efficiency of the optimization.

Another scenario in which a light source is removed is if it no longer contributes to improving the objective function. This can happen if a light is placed within a closed mesh. If such a situation arises, the light can't be further optimized, as gradient-based optimization requires a meaningful gradient. Adjusting the intensity of a light stuck in a mesh will not impact the scene's lighting and, therefore, won't influence the optimization. We remove such lights from the solution if their position and intensity derivatives are smaller than 10^{-7}. In our experiments even virtually optimal light constellations, still had derivatives around 10^{-3} and 10^{-4}, so by setting the threshold even lower, we can ensure that only problematic lights are removed.

With these steps, less intense lights can be merged with bright lights. This not only improves performance but also often results in a lighting solution with fewer light sources and a comparable objective value, as compared to optimization without the merging step. Furthermore, if a light turns out to be a poor candidate or becomes stuck in geometry and irrelevant for the optimization, it will be removed.

3.5 Optimization Constraints

Certain environments such as office settings or parking structures usually have their light fixtures arranged at a consistent height and intensity. By incorporating soft constraints into our optimization process, we can guide it towards similar, practical lighting solutions. Soft constraints extend the objective function by adding a penalty value to the objective value while keeping the optimization formally unconstrained. This means the constraints won't get strictly enforced, keeping the optimizations flexible. In the case of low penalties, this approach resembles regularization techniques. For the optimizer to be aware of constraints, it's crucial to incorporate the gradients of these constraints into the optimization process.

Specifically, we add height and intensity constraints to our optimization, to keep light sources at the same height and their intensities similar to one other. The height difference penalty is calculated by looping through each pair of lights and looking at their height differences. Assuming the y-vector is the up-vector then let y_1 and y_2 be the heights of two lights, and α be the penalty factor. The penalty function ψ and the gradients $\frac{\partial \psi}{\partial y_1}$ and $\frac{\partial \psi}{\partial y_2}$ (partial derivatives of ψ with respect to y_1 and y_2) for the height difference are:

$$\psi = \frac{\alpha}{2} \cdot (y_1 - y_2)^2, \tag{3.12}$$

$$\frac{\partial \psi}{\partial y_1} = \alpha \cdot (y_1 - y_2), \tag{3.13}$$

and

$$\frac{\partial \psi}{\partial y_2} = \alpha \cdot (y_1 - y_2) \cdot (-1). \tag{3.14}$$

The intensity difference penalty can be calculated in the same way, by replacing the height values with intensities.

When optimizing, we consider every pair of lights in the scene. For each pair, we compute the penalty for the height difference and intensity difference as shown in equations 3.12 through 3.14. The total penalty for the system is the sum of these individual penalties over all pairs of lights.

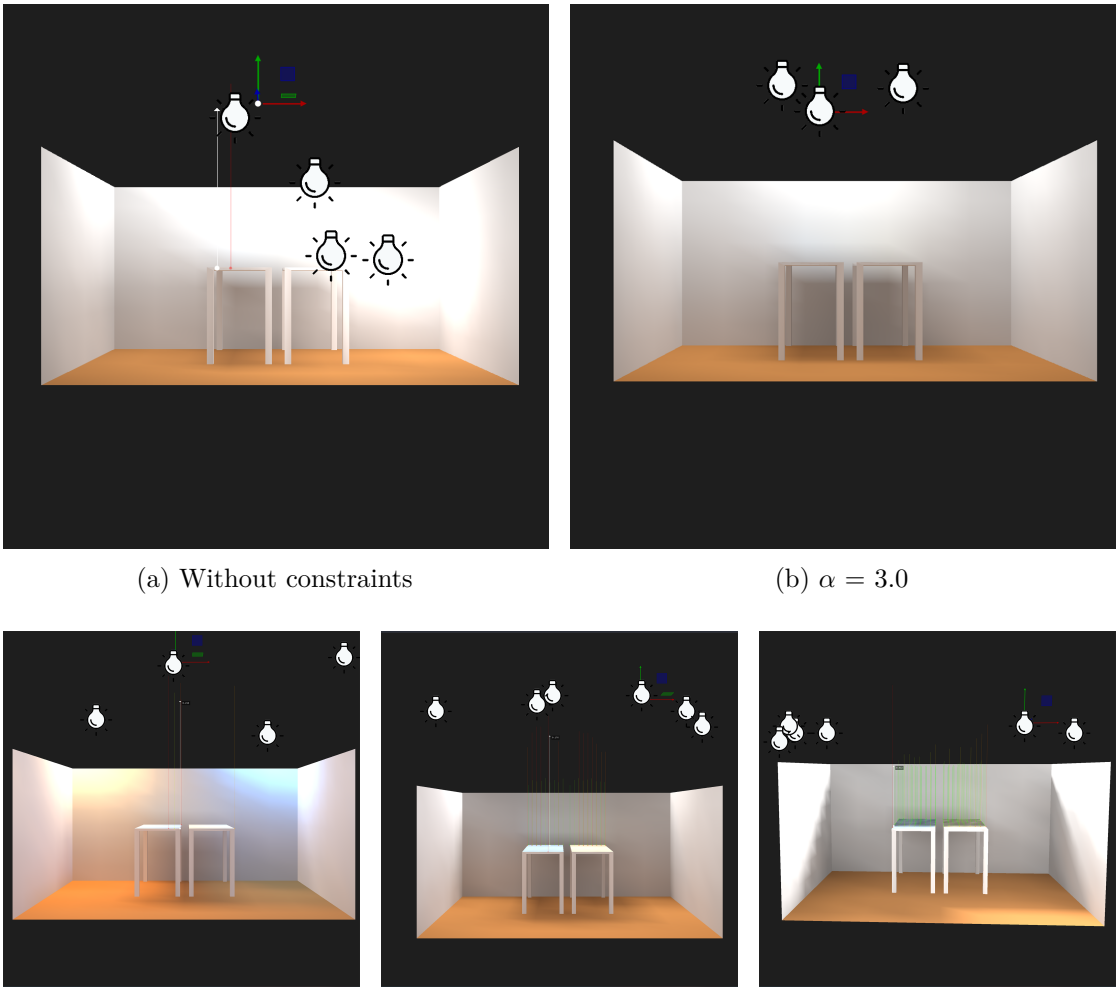
This total penalty is added to our main objective function, yielding the function that the optimizer tries to minimize. As for the gradient, which informs the optimizer on how to adjust each parameter, it is modified by the gradients of these penalties. The influence of varying penalty values can be observed in Fig. 3.7. Each light in our system is characterized by multiple parameters, including its position and intensity. The partial derivatives we compute for each light pair, as per the penalty function, are inserted into the gradient at the appropriate positions. This ensures that the optimizer adjusts each light's position and intensity, considering both the main objective and the constraints imposed by our penalty terms.

These constraints enable us to reduce the search space to a region that offers more viable solutions. In practical scenarios, it's often more economical to purchase lights with identical wattage and install them at the same distance from the ceiling. A potential drawback of this approach is that it might pose challenges in multi-floor simulations, as the penalty for height difference would attempt to align all lights at the same level. This effect can be reduced by considering the horizontal distance between the lights in the penalty function.

It's worth mentioning that the current implementation doesn't scale efficiently. As we loop through all pairs of lights, the computational complexity grows quadratically with the number of lights. One way to mitigate this would be by clustering the lights. That way, the constraints are applied only to lights within the same cluster, which not only offers greater control but also improves scalability. Another potential improvement would be to evaluate each light against the average of its cluster, rather than a pairwise evaluation.

An illustrative example of these constraints in action can be seen in a small office environment in Figure 3.7, where the lighting solutions are applied in a way that provides an even distribution of light with all light sources placed at similar heights and intensities. Note that the influence of the height constraint is noticeably smaller for $\alpha = 0.00001$.

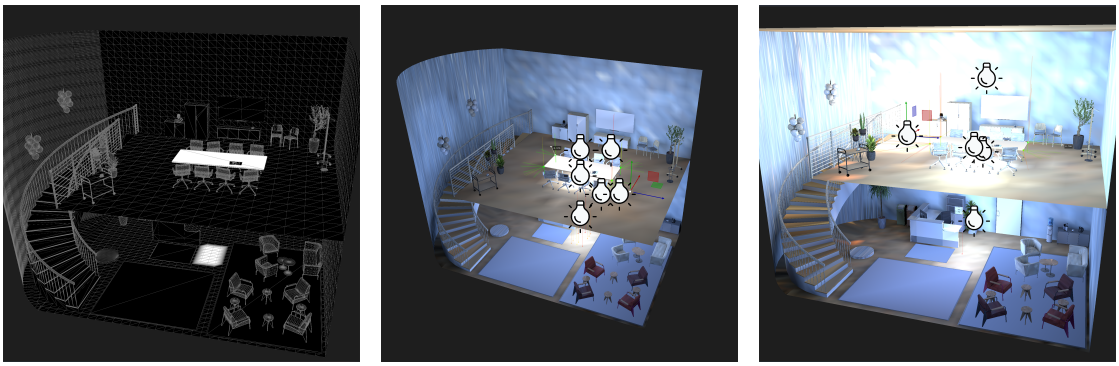
However, when these constraints are applied in a larger office setting with multiple floors (Figure 3.8), the aforementioned issue of height alignment across multiple levels can be observed. Here the influence of the height constraint with $\alpha = 0.00001$ is low enough to keep the lights on separate floors. With a penalty set to 10, the optimizer may not find a stable light constellation at all as evident in Fig. 3.8f.



(a) Without constraints (b) $\alpha = 3.0$

(c) $\alpha = 0.00001$ (d) $\alpha = 0.001$ (e) $\alpha = 1.0$

Figure 3.7: Same-Height Constraint - Comparison of different penalty values in the small office scene. Note that the heights only start to differ in height for $\alpha = 0.00001$ (c).



(a) Light target (b) Without constraint (c) $\alpha = 0.00001$

(d) $\alpha = 0.1$ (e) $\alpha = 1.0$ (f) $\alpha = 10.0$

Figure 3.8: Same-Height Constraint - Comparison of different penalty values in the large office scene. The light solutions are generated by our ADAM dynamic approach using 8 effective iterations. This comparison includes (a) the light target,(b) an optimization without constraints, and (c) an optimization where the penalty is low enough to allow lights on different floors. A higher penalty (f) may prevent the optimizer from finding a stable solution.

CHAPTER 4

Implementation

As mentioned in section 2.4 the Tamashii framework provides working implementations of various optimization algorithms, such as ADAM and L-BFGS. Using these two algorithms we build our light optimization approach. During development, we encountered some engineering challenges. In this section, we look at some of them and discuss our solutions to tackle these challenges.

4.1 Min-Heap

A min-heap is a specific variant of the heap data structure, which is inherently a specialized tree implementation [CLRS]. It allows the insertion of elements based on a given key value. The elements are organized in an order determined by their keys. Specifically, in a min-heap, the key of a parent node is always smaller than the keys of its child nodes. Consequently, the element with the smallest key is guaranteed to be on the "top" of the heap.

As mentioned in section 3.1.1, we choose a min-heap data structure for the light candidates selection. Throughout this process, we iterate over lots of different vertices and require a way to maintain the last 2048 vertices with the highest radiance difference. This means, that as soon as we add the 2049th element, its radiance difference needs to be compared against the lowest one in the collection. If the new element has a higher radiance difference, we can replace it with the lowest element. Ideally, the lookup for the smallest element happens in $O(1)$. A min-heap perfectly fits this use case as it keeps its elements sorted by key and the element with the lowest key is instantly accessible.

Although the C++ Standard Library does not include a min-heap directly, it provides a priority queue that is implemented using a heap. A priority queue orders the elements that are added to it, by an associated priority. By passing a custom comparator to this priority queue, we obtain a min-heap.

For our implementation, we created this comparator in Listing 4.1 that evaluates pairs of tuples. These tuples consist of a float value for the radiance difference, a vertex object, and an index for the vertex's buffer location.

```
struct CandidateComparator {
    bool operator()(const std::tuple<float, vertex_s, uint32_t>& a,
                    const std::tuple<float, vertex_s, uint32_t>& b)
                       const {
        return std::get<0>(a) > std::get<0>(b);
    }
};
```

This comparator ensures that the tuple with the lowest radiance difference is always at the top of the heap.

4.2 Optimization Algorithm

As discussed in section 3.3, our algorithm aims to improve the lighting solution of a given scene using two optimization phases. This algorithm first checks if the scene needs an additional light, by calculating light candidates. If a candidate is found, then it is inserted into the scene, where the first optimization phase increases its intensity. Then in the second phase, all lights in the scene are optimized with respect to both their position and intensity. Afterward, unnecessary lights are either removed from the scene or merged with brighter ones. Then the algorithm can check again if the scene needs additional lights, and repeat the process until there are enough lights or a certain number of optimization cycles have passed. The detailed algorithm is provided by Listing 4.1.

In this section, we will provide the algorithm used for our approach and share the challenges faced when implementing it.

4.2.1 Logging and Data Processing

Measuring and comparing modifications to the algorithm is essential for understanding its performance and finding potential areas for improvement. Ignoring this would lead to a lack of clarity on how different optimizers interact and affect the overall result.

To measure our algorithm, we require data for each iteration. Both the ADAM and L-BFGS optimizers already log their objective and constraint values to the application's console window. The output can easily be changed to a text file, but the data was still disconnected from the ADAM dynamic's data.

We address this, by writing a separate log into a CSV file and then combining the data from this CSV with the data from the text file. The resulting logging data ensures that we have a comprehensive view of the algorithm's performance across different stages. This allows us to better analyze its behavior and make more informed decisions.

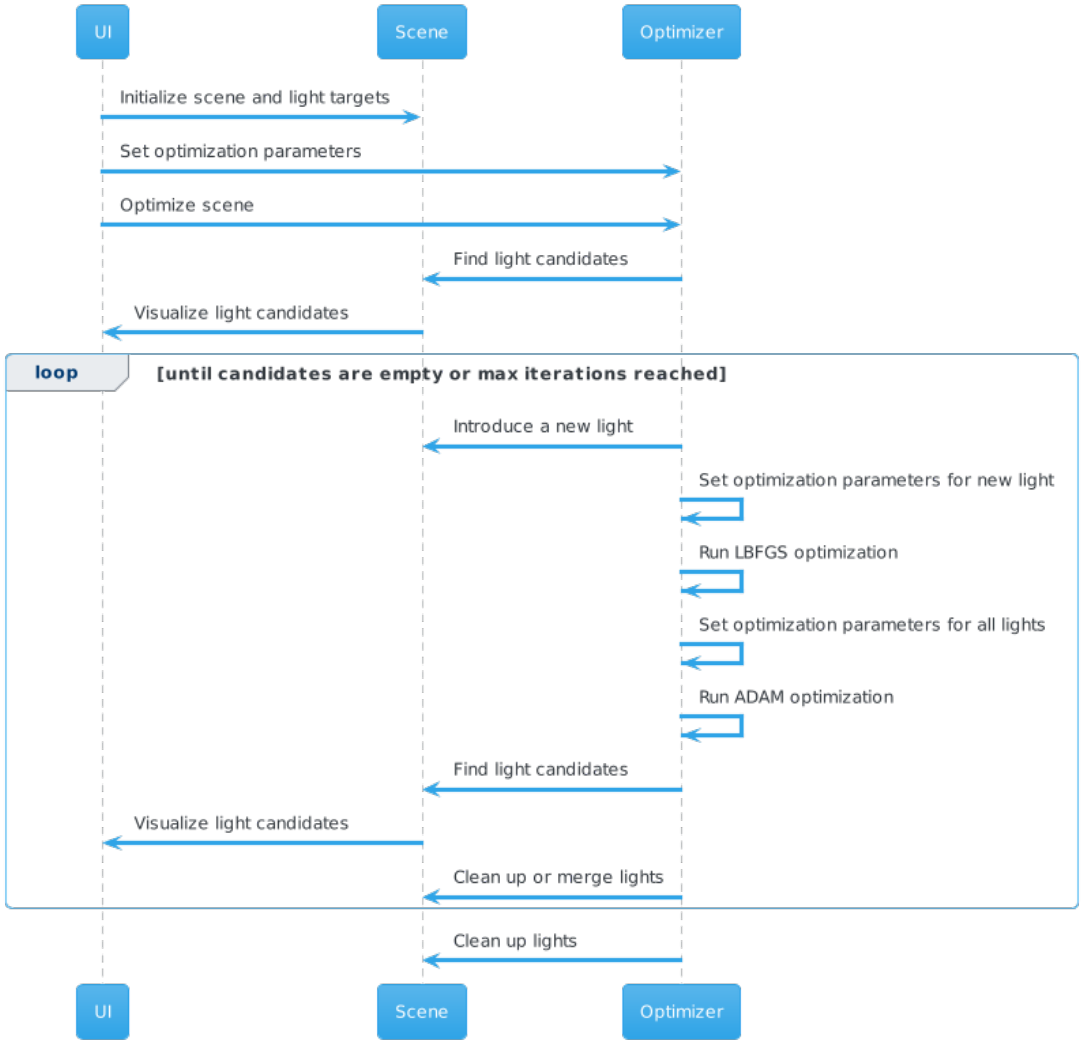


Figure 4.1: Sequence diagram of a typical execution of an ADAM dynamic light optimization. The UI represents the elements the user can interact with, the scene describes the rendered and optimized scene, and the optimizer is the ADAM dynamic algorithm.

Algorithm 4.1: Dynamic Optimization of Light Positions and Intensities

```
Data: Radiance buffer, step size, iteration counts
Result: Optimized light placements and intensities
Initialize rendering scene and light targets
Create L-BFGS and ADAM optimizers
L-BFGS Max Iterations ← 5  ADAM Max Iterations ← 80
L-BFGS initial step size ← L-BFGSStepSize  ADAM step size ← AdamStepSize
Find light candidates
if scene has light candidates then
    candidates ← get candidates from scene
    max iterations ← MaxIters
    while candidates are not empty and max iterations are not reached do
        Place a new light source into the scene
        Set optimization parameters for all lights to false except the intensity of
         new light
        Run L-BFGS optimization
        Set optimization parameters for all lights to true
        Run ADAM optimization
        Find light candidates
        Clean up or merge light sources
    end
end
else
    Set optimization parameters for all lights to true
    Run ADAM optimization
end
Clean up light sources
Stop optimization
```

4.2.2 ADAM Gradient History

As mentioned in section 3.3, we encountered difficulties when only relying on the ADAM optimizer for both optimization phases. Our primary concern was that the ADAM optimizer's adaptive learning rate would lead to aggressive updates when dealing with newly added lights in the first optimization phase. Because the light's intensity is initially low, this behavior could potentially increase this intensity dramatically. Therefore, we perform the first optimization phase with the L-BFGS optimizer.

However, the alternating use of two optimization algorithms poses challenges. While the ADAM algorithm executes, it aggregates gradient data to build its momentum [KB14]. This momentum represents the current state of the scene. When we introduce a new light and adjust its intensity with the L-BFGS optimizer, we change the scene without updating the momentum. This can lead to the ADAM optimizer continuing the optimization with

a false understanding of the current scene.

To avoid this, we reset the internal variables of the ADAM optimizer before each use. This leads to the algorithm always starting without any momentum. Because of this, the algorithm should initially perform similarly to the regular Gradient Descent algorithm. Over multiple effective iterations, this could noticeably slow down the convergence of our approach.

Addressing this challenge might involve adopting strategies from other contexts, such as a warm restart approach similar to those used in deep learning, as suggested by Loshchilov and Hutter (2016) [LH16]. With a strategy like this, the momentum could be maintained even after resets. Additionally, from a user perspective, a system of checkpoints that records intermediate optimization states could allow for a more interactive optimization process. This would enable the user to revert to a previous state if the provided solution was sub-optimal.

This section discussed the challenges of using alternating optimization algorithms and highlights potential areas for improvement in our approach, for future work. The implementation of a warm restart strategy and a checkpoint system could improve the convergence rate and user-friendliness of the optimization algorithm. However, it would require further testing to assess the feasibility and effectiveness of a warm restart approach.

4.2.3 Light Candidate Ownership

Visualizing the light candidates brings its challenges with it. The biggest being that the candidates needed to be accessible to the optimization algorithm and the ImGUI library, responsible for the visualization. Our approach is to save the light candidates as the member variable *mLightCandidates* in the scene. The function for finding light candidates acts as a setter and the optimization function and the visualization module can access the light candidates through a getter function. This architecture is reflected in algorithm 4.1. Here we can see that in line 5 the candidate search is performed and that the *mLightCandidates* variable is updated with the result. In a multi-threaded implementation in contrast to ours, this variable may cause problems with thread-safety. In line 6 we check if any light candidates were found. If we did, the getter for the light candidates is called in line 7, and the optimization proceeds. This architecture makes it possible to run the light candidate search independent from the optimization. We added a button to the UI that performs this search and visualizes the candidates, which simplifies debugging our approach.

4.3 Arrow Projection in ImGUI

As part of the visualization, the light candidates are represented as arrows. Instead of modifying the render process and building our visualization pipeline using Vulkan directly to draw these arrows, we opted to use the visualization tools provided by the

ImGUI Library, which operates on top of Vulkan. This makes it easier to prototype the visual representations and simplifies the debugging of the light candidates.

The light candidate arrows contain two 3D points, the origin and the tip. These points must be projected onto a 2D screen. This fundamental transformation in 3D graphics can be performed with matrix multiplications. For an arrow's origin v_{origin} this transformation is done with these steps:

$$v = \begin{pmatrix} x_{\text{origin}} \\ y_{\text{origin}} \\ z_{\text{origin}} \\ 1 \end{pmatrix}, \tag{4.1}$$

$$v' = P \cdot V \cdot v, \tag{4.2}$$

$$v_{\text{ndc}} = \begin{pmatrix} x'/w' \\ y'/w' \\ z'/w' \end{pmatrix}. \tag{4.3}$$

In Eq. 4.1 the 3D position of the light candidate's origin is converted into a 4D homogeneous coordinate. The equation in Eq. 4.2 illustrates the multiplication of the view matrix V, projection matrix P, and the homogeneous vector v'. This operation effectively positions the point within a canonical viewing volume. The division in Eq. 4.3 is the homogenization step, which essentially brings our projected point to normalized device coordinates (NDC). In NDC, the visible volume of space is typically represented as a cube with dimensions from -1 to 1 in all three axes. The same operations are applied to the tip of each arrow [AvD13].

Now we need to map the transformed positions from NDC to actual screen positions:

$$v_{normalized} = \left(\begin{pmatrix} x_{ndc} \\ y_{ndc} \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right) \frac{1}{2}, \tag{4.4}$$

$$v_{\text{pixel}} = \begin{pmatrix} x_{normalized} \cdot x_{resolution} \\ y_{normalized} \cdot y_{resolution} \end{pmatrix}. \tag{4.5}$$

First, the operations in Eq. 4.4 shift the range of NDC from [-1,1] to [0,2] and normalize it to a [0,1] range. Afterward, the normalized coordinates are multiplied by the screen resolution, mapping them to the final pixel coordinates.

The arrows are presented using ImGUI shapes and their corresponding pixel coordinates. This approach has one main disadvantage over the Vulkan graphics pipeline: It's missing a depth buffer. Without a depth buffer, overlapping arrows are drawn in a seemingly

random order, instead of based on their distance from the camera. The result can be confusing because there is no way to know which arrow is closer to the viewer. Additionally, because the scene geometry is rendered before the visualization, arrows that are actually behind the scene, are drawn in front of it. Every arrow will always be visible, regardless of any 3D objects that should occlude them. Only by repositioning or rotating the camera does the viewer get a sense of spatial understanding, as arrows move differently at different depths.

One potential solution for these issues could be to visualize arrows using the Vulkan pipeline. The depth buffer is already in use, therefore the arrows would be placed at the correct depth. However, this involves generating arrow geometries and managing additional Vulkan code.

A more straightforward strategy might involve sorting arrows by their average depth and then drawing them back-to-front using Painter's Algorithm [AvD13]. While this could help with overlapping arrows, it won't draw arrows occluded by the scene correctly. Requiring the arrows to be sorted every frame, could also be inefficient for a large number of arrows.

In conclusion, while our current visualization approach using ImGUI has limitations, it nonetheless provides a sufficiently accurate representation of the optimization process. It helps users understand how the light candidates are selected and positioned within the scene and how the optimization progresses.

4.4 Summary

In this chapter, we looked at the many parts that make up our light optimization strategy and the different challenges we faced implementing it. With the use of the C++ priority queue, paired with a custom comparator, we created a min-heap data structure for the light candidate selection.

Our optimization algorithm was explained with pseudo-code and a sequence diagram, to provide a comprehensive understanding of the process.

For the visualization, we projected the positions of light candidates onto the screen. However, our approach lacks a depth buffer, which could be an area of improvement for future projects.

In the next chapter, we turn our focus toward the evaluation of our implementation, assessing the convergence rate in different scenes and comparing the performance against other optimization algorithms.

CHAPTER 5

Results and Evaluation

In this chapter, we analyze the performance and convergence of our approach in two different scenes. Moreover, we compare the solutions from our approach against solutions provided by the ADAM and L-BFGS optimizers.

5.1 Performance Evaluation

The comparisons below demonstrate that the ADAM dynamic approach offers improvements over previous methods. Here we compare the ADAM dynamic against the regular ADAM solver as well as the L-BFGS solver, which are both used in combination in our approach. As our goal was to improve the light solution optimization, our metric for comparison is the best objective value, obtained from a completed optimization run.

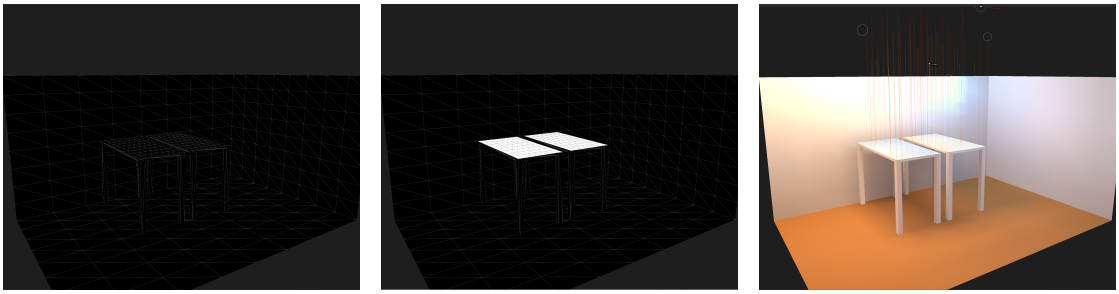
5.1.1 Small Office Benchmark

This benchmark aims to verify the effectiveness of our ADAM dynamic approach in smaller scenes. We compare the best objective values of our method against the regular ADAM and the L-BFGS. Furthermore, the benefit of our merging technique will be demonstrated by optimizing a scene filled with an excess of lights. The test scene is the "Small Office" scene, comprised of a single room with two tables as depicted in Fig. 5.1. Each table has a light target on it.

Test Design

For the small office scene, all three solvers will perform two sets of benchmarks: the first set to evaluate the performance of our ADAM dynamic approach without light merging, and the second set to evaluate its performance with light merging.

In contrast to our ADAM dynamic, the regular ADAM and L-BFGS optimizers do not introduce new lights during the optimization. They typically terminate either after



(a) Wireframe model (b) Light target (c) Optimization result

Figure 5.1: Small office scene - The benchmark starts with the scene being completely dark (a) and two light targets that are drawn onto the tables (b). The optimization process then terminates with 4 generated light sources (c). Note that the resulting intensity on the tables seemingly matches the light targets.

a specified number of iterations or upon reaching a local minimum. To ensure a fair comparison, it's essential to set an initial light count and a maximum iteration limit, as well as define light positions and intensities. The appropriate values for these parameters are determined based on multiple test runs using the ADAM dynamic approach.

In the non-merging tests, each solver executed five optimization runs. The ADAM dynamic's final light count ranged between 1 and 4 lights, leading us to initiate the other tests with a 2-by-2 grid of lights. This setup was chosen with the assumption that the optimizers can easily reposition redundant lights or simply reduce their intensity. Although the ADAM dynamic had a limit of 6 effective iterations, we initially believed that other methods would require roughly 480 iterations. However, in reality, they often concluded earlier, typically between 40 to 160 iterations. Therefore, both ADAM and L-BFGS runs had their iteration limits set at 160. All solvers operated with a step size of 0.15 and the placement threshold for the ADAM dynamic was also fixed at 0.15.

The subsequent benchmarks incorporated light merging and were performed similarly. Here, all three methods started with a 5-by-5 grid of lights. The termination conditions were left unchanged.

Results

Our primary focus is on the objective values attained by the three solvers during the benchmarks. For clearer visualization, we generated box plots from the recorded data. To simplify comparisons, we normalized the objective values relative to the objective values at the start of the optimization.

The results of the non-merging benchmarks are presented in Figure 5.2. In our results we normalized all objective values based on the initial objective value of the optimization, making it easier to compare different results. The graphic mirrors the insights from

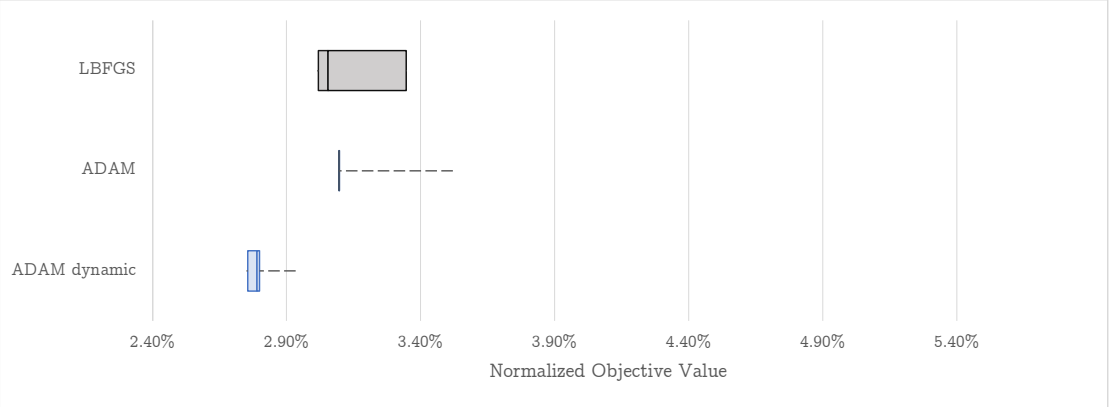


Figure 5.2: Small office scene - Box plot of the best normalized objective values of the optimization runs without light merging. The ADAM dynamic algorithm started with 0 lights, the others with 4. Lower values indicate a better solution.

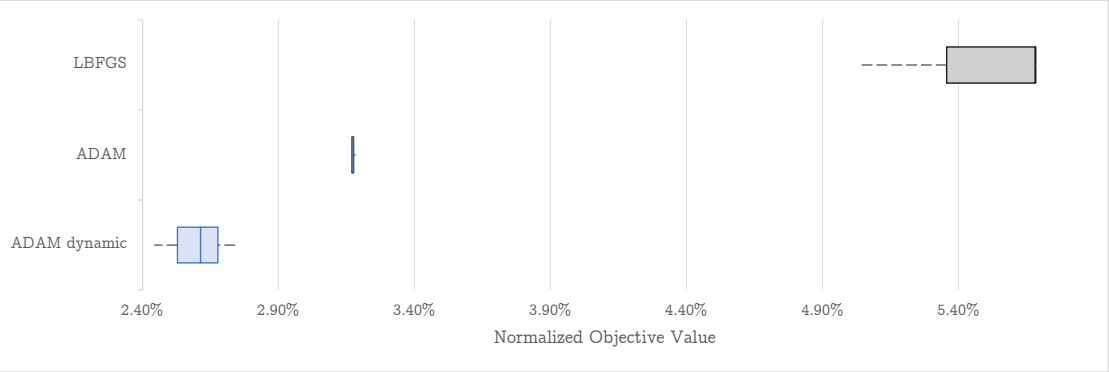


Figure 5.3: Small office scene - Box plot of normalized objective values of the optimization runs with ADAM dynamic applying light merging. All three algorithms started with 25 light sources. Lower values indicate a better solution.

Figure 5.8, indicating the ADAM dynamic approach typically outperforms both the regular ADAM and L-BFGS, improving the probability of finding the global minimum.

The advantage of dynamically adjusting the number of lights during optimization becomes apparent in Figure 5.3. Both, the ADAM and the L-BFGS solvers were able to minimize the objective function effectively with 25 lights in the scene. Notably, ADAM consistently converged on the same minimum in every run, as shown by the narrow box plot centered around 3.1%. Our findings revealed that the ADAM dynamic not only reached the lowest objective value but also reduced the final light count dramatically, from the starting 25 down to just 6 or at times 11. This outcome suggests our method is superior in optimizing scenes overloaded with lights, compared to the conventional ADAM or L-BFGS methods.

Lastly, the distribution of the ADAM dynamic with light merging is compared to the

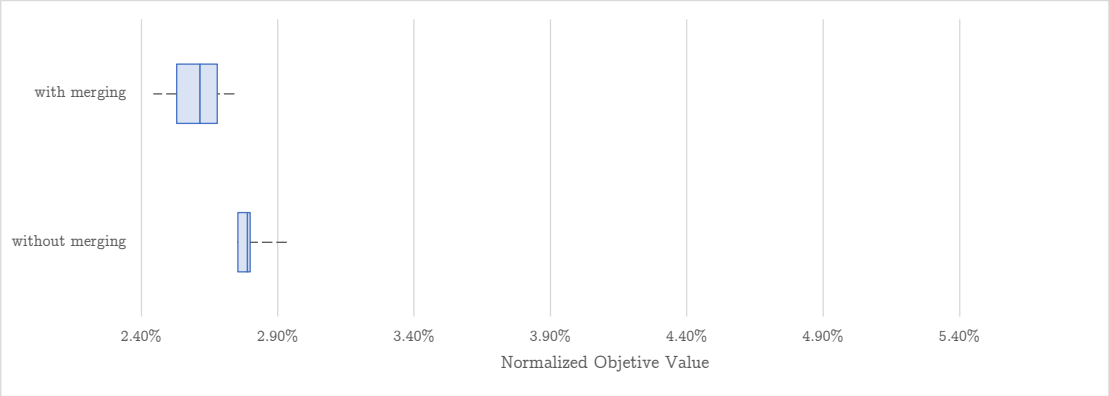
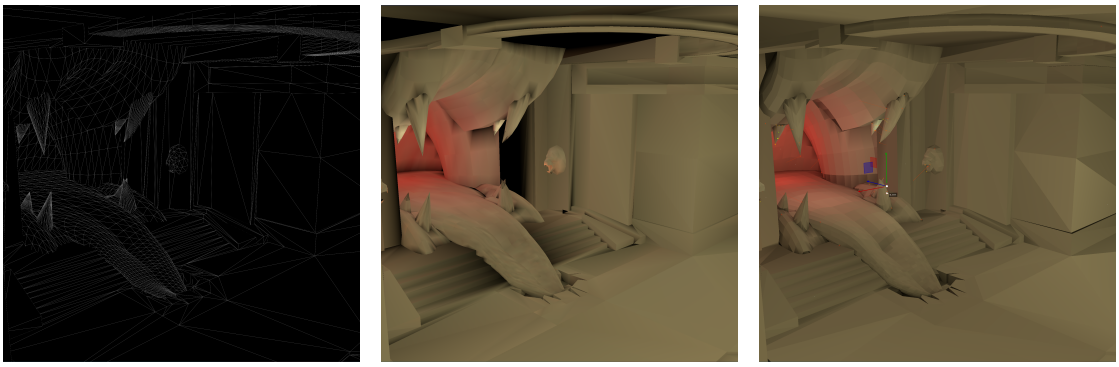


Figure 5.4: Small office scene - Box plot of ADAM dynamic with light merging starting from 25 lights (Fig. 5.3) and without light merging starting with 0 lights (Fig. 5.2). Ideally, with the same scene and light targets we would expect the result of these experiments to be identical.

one without merging. It is important to note that these two experiments differ in their setups and were not designed for direct comparison. Despite this, the insights gained are valuable, as the scene and target lighting between them, are identical. This means the expected optimal lighting for both of them should be the same, yet the results differ. As illustrated in Figure 5.4, the inclusion of light merging tends to yield improved lighting solutions in the small office scene. This indicates potential improvements in our light placement approach. In the non-merging scenario, a total of 4 lights were added, whereas in the merging scenario, the number of lights was reduced from 25 to 6.

5.1.2 Quake Benchmark



(a) Wireframe model (b) Light target (c) Optimization result

Figure 5.5: Quake scene - Comparison of the light solution generated by our ADAM dynamic approach (c) and the light target derived from the scene's baked lighting (b). The optimization started with the scene being completely dark (a) and terminated with 20 generated light sources.

In this benchmark, we want to compare the three distinct approaches in a large 3D scene. We use a map from the game "Quake 3" (Figure 5.5) for this purpose. This map features baked-in lighting, serving as the target for our optimizers.

Test Design

Just as in section 5.1.1, we first performed multiple runs with the ADAM dynamic optimizer and no initial lights, then decided on an initial light count and maximum iteration count for the other approaches.

For the ADAM dynamic benchmarks, we used step sizes of 1. Light candidates were selected probabilistically, light merging was disabled, the light insertion threshold was fixed at 0.05 and the limit for effective iterations was 20. With such a low light insertion threshold, we can assume that the optimizer will terminate after 20 effective iterations, rather than pushing the solution beneath the threshold. We chose 20 iterations because, during our test runs, the objective value appeared to plateau after this number.

To illustrate this, consider Figure 5.6. The blue line represents the normalized objective values, where 100% represents a dark scene. Here we stopped the optimization after 26 iterations because at this point, the computations became very slow on our machines and it was unclear if the solution was improving at all. In Figure 5.6 a total of 26 grey dots mark the duration of each effective iteration at the end of it. A vertical line highlights the 20th iteration. Here we can observe, that past the 20th iteration, there is minimal improvement in the objective value, with only a 1.3% drop from the 20th to the 26th iteration. In Contrast, the duration for each effective iteration is rising nearly linearly, as depicted by the trend line aligning with the grey dots. The entire run took 1 hour and 4

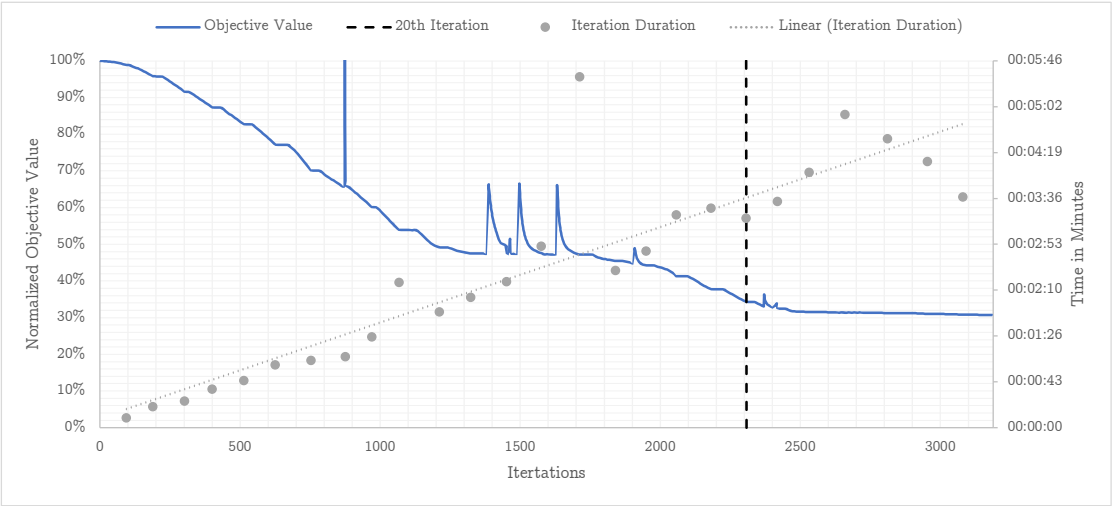


Figure 5.6: Quake scene - Normalized objective value over 26 effective iterations of ADAM dynamic optimization vs. time spent per effective iteration. The grey dots indicate effective iterations and the dashed line our chosen iteration cut-off for the benchmark.

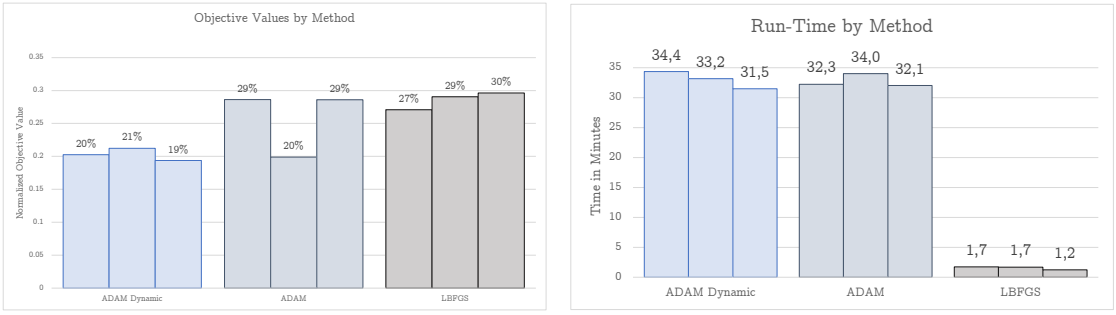
minutes. Notably, the last 6 effective iterations took 25 minutes (or 40% of the run time) resulting in only a 1.3% improvement. Therefore we decided to limit this experiment to 20 effective iterations.

To account for colored light as illustrated in Figure 5.5 the whole scene optimization step also optimizes for color in these runs. The scene notably has some shadows that are impossible to model using a physically correct light model.

Because each of the ADAM dynamic runs took 20 effective iterations and ended with an average of 20 lights, we standardized the initial light count to 20 for the other approaches. In this scene, an average effective iteration took 80 iterations, so we settled on a maximum iteration count of 1600. Both the ADAM and L-BFGS benchmarks had their step size set to 1, and 20 lights were uniformly distributed across the scene in a 4-by-5 grid, each with an intensity of 0.5 watts.

Results

Each optimization method was executed thrice to minimize anomalies and account for variability in our results. Our evaluation will primarily focus on the best objective value while also commenting on the time taken to achieve it.



(a) Comparison of objective values

(b) Comparison of run-time

Figure 5.7: Quake scene - Results after 20 effective iterations. ADAM dynamic seems to achieve the best solutions (a), while L-BFGS performs the fastest (b).

In Figure 5.7a the objective values of our nine tests grouped by the method used, are compared with each other. The ADAM dynamic arrived at comparably low objective values and even got the lowest out of all three. The regular ADAM occasionally matched this performance, achieving a comparably low value in one test. L-BFGS, on the other hand, delivered consistent results that were slightly higher than those of the ADAM dynamic. In Figure 5.8 we plotted histograms based on our performance results. The data suggest that the approach with the lowest expected objective value tends to be the ADAM dynamic.

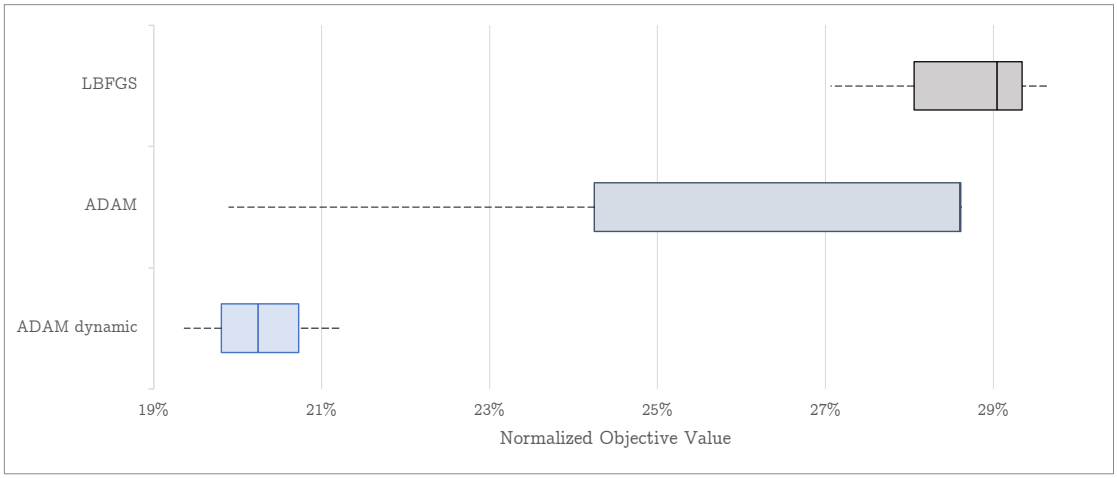


Figure 5.8: Quake scene - Box plot of the normalized objective values from Fig. 5.7a. Lower values indicate a better solution.

The run-time of each approach as displayed in Figure 5.7b clearly shows that the L-BFGS optimizer converges considerably faster than both ADAM variations, while still providing useful results. The latter two typically required over 30 minutes to complete their runs while the L-BFGS usually finished in under two minutes. While we employ both ADAM

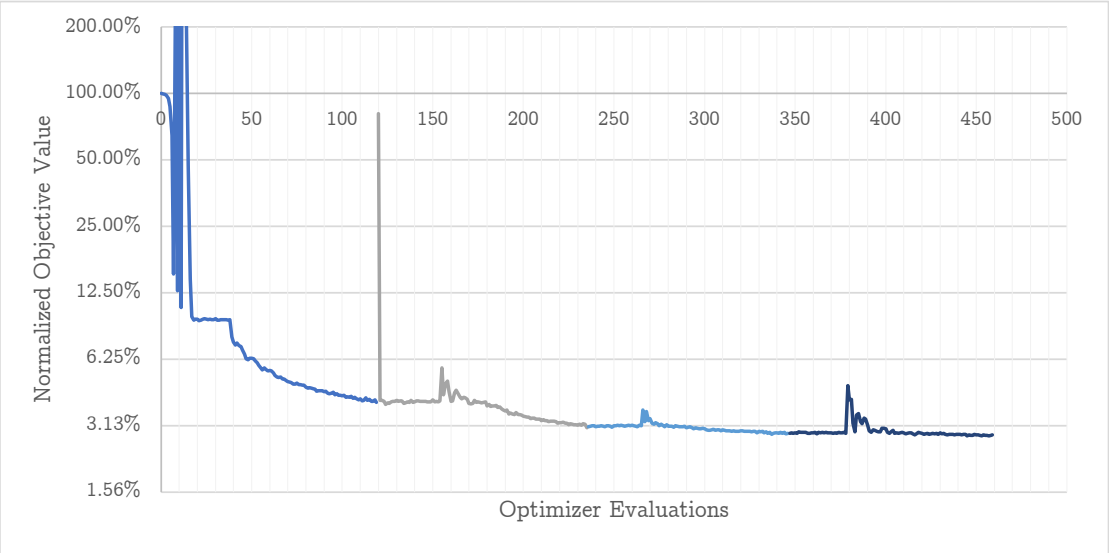


Figure 5.9: Small office scene - Normalized objective value for each optimizer evaluation for 4 effective iterations of the ADAM dynamic. Each color represents a different effective iteration. The final solution has an objective value of around 3% from the initial one.

and L-BFGS, our optimization approach predominantly utilizes ADAM iterations which explains the similar runtime.

5.2 Convergence Analysis

To verify that our approach converges to local minima, we tracked the objective value during the optimization process. Specifically, we recorded data at the beginning of each iteration, after optimizing the light intensity with L-BFGS and following the scene optimization using ADAM. This section looks into how our strategy converges.

5.2.1 Small Office Scene

To identify potential problems with the optimization strategy, we analyze the convergence trend in more detail by plotting each objective value. In Figure 5.9 we depict the convergence pattern of the small office scene, with the step size and placement threshold both fixed at 0.15. The optimization stopped after four effective iterations with an objective value of around 3% of the initial one.

At the start, the first light is inserted and its intensity is optimized, during this step, the objective value fluctuates sharply until it plateaus and terminates at around 35 evaluations. Although the L-BFGS optimizer used in this phase, has a limit of 5 iterations, the way it is implemented can sometimes extend its computation for longer than that. Following

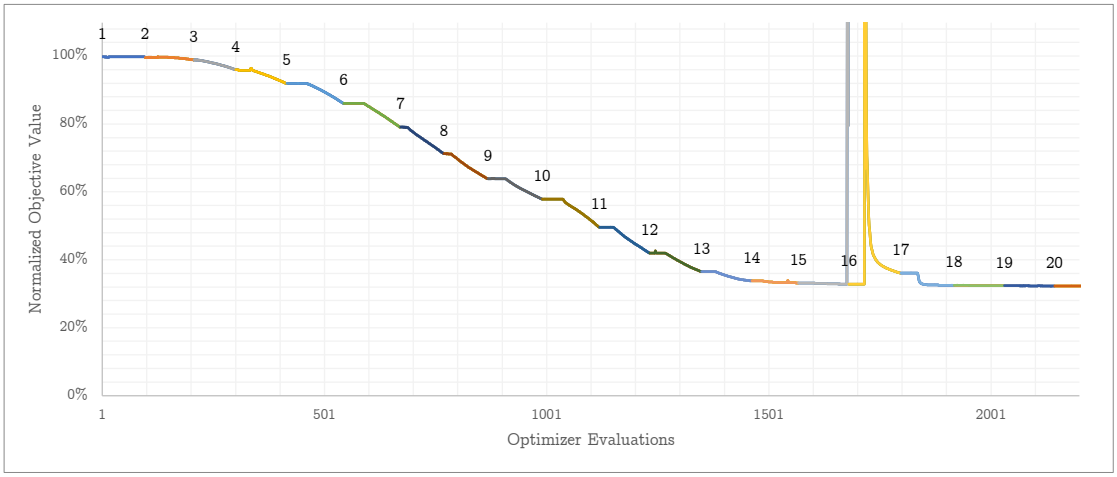


Figure 5.10: Quake scene - Normalized objective values for each optimizer evaluation for 20 effective iterations of the ADAM dynamic. Each color represents a different effective iteration. The final solution has an objective value of around 30% from the initial one.

this, the ADAM optimizer begins positioning the light source and the objective value quickly drops, implying the light was initially positioned away from its ideal location.

In the second effective iteration a new light is added, causing a brief spike in the objective value, which then rapidly stabilizes. This could be attributed to suboptimal light positioning or the L-BFGS over-shooting of the intensity. Afterward, the ADAM optimizer starts oscillating from 155 to 170 evaluations, then converges to an objective value of around 0.4% of the starting value. Beyond this point, two additional effective iterations are executed, each adding new lights, but they only marginally improve the solution, decreasing the objective value by about 0.04%.

5.2.2 Quake Scene

The trend illustrated in Figure 5.10 demonstrates a typical convergence pattern. Within the initial 6 effective iterations, the objective value drops from 100% to 77%. Given that the scene starts void of any lights, the decline in the objective value during the initial iterations is logical, as more regions of the scene are progressively lit.

During the intermediate phase, the convergence rate slows down, and the objective value appears to decline almost linearly with the iteration count. This might be because the light solution already illuminates the scene close to the target, with only finer details left for optimization.

After 13 effective iterations, although the optimizer continues to reduce the objective value, the rate of reduction is significantly slower. Until the end of the optimization, the objective value is only improved by about 2%. During the 17th iteration, the objective

value goes up, as soon as the ADAM optimization starts, indicating strong over-shooting of the target lighting.

This observed pattern confirms that our optimization algorithm systematically diminishes the objective value. The progression, marked by a swift initial decline followed by gradual refinements aligns with the expected outcomes. Future work may investigate the nature of the plateaus at the beginning of each effective iteration and the reason for the rapid jump near the end.

This detailed analysis shows that our approach works, with potential for improvement, especially for the light placement. The L-BFGS also seems to perform more iterations than needed before the ADAM starts. Meanwhile, the maximum iteration count for ADAM could potentially be lowered with each successive effective iteration.

CHAPTER 6

Conclusion and Future Work

Global light optimization in 3D scenes is a complex process, with lots of challenges to overcome. This thesis presented the ADAM dynamic for light optimization, a specialized approach that can generate and merge point lights while utilizing established optimization algorithms. By making use of GPU-accelerated ray tracing we offer a forward-looking approach for solving ideal light placements in 3D scenes.

One of the significant contributions of this thesis is the method of generating light sources. By programmatically finding suitable light candidates and inserting them into a scene without destabilizing the underlying optimization algorithms and carefully removing unnecessary light, we achieved improved light solutions.

However, as with any new approach, there's room for refinement. Potential possibilities for improvement include the light placement. The number of steps to reach an ideal solution could be reduced if the added lights are optimally placed. This could be achieved by better estimating the position of new lights. The existing heuristics for merging lights, which currently require manual input, could also benefit from automation and a more elaborate selection process. A promising optimization strategy might also involve starting the process with a grid of lights and a normal optimizer, then as the convergence rate slows, deploy our method to reduce lights and refine the solution further. Experimenting with various optimizer combinations and initial light counts could potentially accelerate convergence and yield even better lighting solutions.

On a broader level, the approach could be extended to other types of lights sources such as spot lights or IES lights, which model real lights more closely. This would require the incorporation of additional properties, such as light orientation, in the light placement method. The properties must also be taken into account in the optimization phases. Merging different types of lights would also require a more sophisticated approach. Additionally, our approach only uses the non-directional radiance data structure, which

only models diffuse surfaces. Directional data could be incorporated into the approach, providing a more precise light optimization.

In conclusion, the ADAM Dynamic for Light Optimization has successfully addressed the challenge of optimizing with a variable light count without compromising the final lighting solution. It also offers potential for future projects in the area of interactive light design.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AvD13] Morgan McGuire John F. Hughes Andries van Dam. *Computer Graphics Principles and Practice*, volume 3. 2013.

[CLRS] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms, third edition.

[CZ13] Edwin K. P. Chong and Stanislaw H .Zak. *AN INTRODUCTION TO OPTIMIZATION*, volume Fourth Edition. 2013.

[DCB15] Yann N Dauphin, Junyoung Chung, and Yoshua Bengio BENGIOY. Rmsprop and equilibrated adaptive learning rates for non-convex optimization rmsprop and equilibrated adaptive learning rates for non-convex optimization harm de vries, 2015.

[Fai13] Mark D Fairchild. *COLOR APPEARANCE MODELS Third Edition.* 2013.

[Han16] Nikolaus Hansen. The cma evolution strategy: A tutorial. 4 2016.

[Img] Omar Cornut imgui github. `https://github.com/ocornut/imgui`. Accessed: 2023-10-07.

[Ir15] Itu-r. Recommendation itu-r bt.709-6 parameter values for the hdtv standards for production and international programme exchange bt series broadcasting service (television), 2015.

[JS11] John Duchi JDUCHI and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization * elad hazan, 2011.

[JSRV22] Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. Dr.jit: A just-in-time compiler for differentiable rendering. *ACM Transactions on Graphics*, 41, 7 2022.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. 12 2014.

[LADL18] Tzu Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. Association for Computing Machinery, Inc, 12 2018.

[LCLL19] Shichen Liu, Weikai Chen, Tianye Li, and Hao Li. Soft rasterizer: Differentiable rendering for unsupervised single-view mesh reconstruction. 1 2019.

[LH16] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. 8 2016.

[LHEN+23] Lukas Lipp, David Hahn, Pierre Ecormier-Nocca, Florian Rist, and Michael Wimmer. View-independent adjoint light tracing for lighting design optimization. 10 2023.

[Mel99] Mitchell Melanie. 0262631857 (pb) 1. genetics-computer simulation.2. genetics-mathematical models, 1999.

[NDVZJ19] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Transactions on Graphics*, 38, 11 2019.

[Noc80] Jorge Nocedal. Updating quasi-newton matrices with limited storage, 1980.

[Noc06] Stephen Wright Jorge Nocedal. *Numerical Optimization (Jorge Nocedal, Stephen Wright)*. 2006.

[NRH+77] F E Nicodemus, J C Richmond, J J Hsia, I W Ginsberg, and T Limperis. Geometrical considerations and nomenclature for reflectance, 1977.

[RRN+20] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. 7 2020.

[Vk2] Khronos Group vulkan api. `https://www.vulkan.org`. Accessed: 2023-10-07.

[ZJL20] Shuang Zhao, Wenzel Jakob, and Tzu-Mao Li. Physics-based differentiable rendering: From theory to implementation. In *ACM SIGGRAPH 2020 Courses*, SIGGRAPH '20, New York, NY, USA, 2020. Association for Computing Machinery.