Informatics

# Inverse Method for Baked Lighting

BACHELORARBEIT

zur Erlangung des akademischen Grades

**Bachelor of Science**

im Rahmen des Studiums

**Medieninformatik und Visual Computing**

eingereicht von

**Mathias Schwengerer**
Matrikelnummer 01527875

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 23. August 2023

Mathias Schwengerer Michael Wimmer

# Inverse Method for Baked Lighting

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Bachelor of Science**

in

**Media Informatics and Visual Computing**

by

**Mathias Schwengerer**
Registration Number 01527875

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, BSc

Vienna, 23rd August, 2023

Mathias Schwengerer Michael Wimmer

# Erklärung zur Verfassung der Arbeit

Mathias Schwengerer

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, 23. August 2023

Mathias Schwengerer

# Danksagung

An dieser Stelle möchte ich mich bei meinen beiden Betreuern David Hahn und Lukas Lipp für die hervorragende Unterstützung und Zusammenarbeit bedanken. Ich schätze vor allem die gute Kommunikation mit euch, eure Hilfsbereitschaft und auch, dass ihr mir die Möglichkeit gegeben habt, an einem spannenden und aktuellen Forschungsthema zu arbeiten.

Weiters möchte ich meiner Familie, meinen Freund:innen und auch meinen Kolleg:innen danken. Ihr habt mich während des Studiums beziehungsweise bereits mein ganzes Leben begleitet und immer wieder für schöne und unvergessliche Momente gesorgt. Ohne euren Rückhalt wäre ich nicht hier, wo ich heute stehe.

# Acknowledgements

I would like to take this opportunity to thank my supervisors David Hahn and Lukas Lipp for their excellent support and cooperation. I especially appreciate the good communication with you, your helpfulness and that you gave me the opportunity to work on an exciting and current research topic.

I would also like to thank my family, friends, and colleagues. You have been with me during my studies and throughout my life and have provided me with many wonderful and unforgettable moments. Without your support I would not be where I am today.

# Kurzfassung

In dieser Bachelorarbeit stellen wir eine neue Anwendung des inversen Renderings vor, bei der Lichtquellen aus einer gespeicherten Beleuchtungsinformation berechnet werden. Als Eingabe verwenden wir eine Szenendatei, welche die Geometrie und die Beleuchtungsinformationen einer Szene enthält. Die extrahierten Szeneninformationen werden dann verwendet, um ein Beleuchtungssetup zu schätzen, welches eine Beleuchtung der Szene erzeugt, die möglichst nah an der gespeicherten Beleuchtungsinformation ist.

Aufgrund von Hardwarebeschränkungen haben Echtzeit-Rendering-Anwendungen wie Videospiele in der Vergangenheit nur eine begrenzte Realitätsnähe in Bezug auf die Beleuchtung geboten. Gespeicherte Beleuchtung war eine gängige Methode, um die Szene unter Echtzeitbedingungen zu illuminieren. Heutzutage ermöglicht hardwareunterstütztes Raytracing dynamische Beleuchtung und globale Beleuchtung in Echtzeit. Unsere Methode zielt darauf ab, ein physikalisch basiertes Beleuchtungssetup zu erstellen, das den gespeicherten Lightmaps so nahe wie möglich kommt. Dieses rekonstruierte Beleuchtungssetup ermöglicht die erneute Beleuchtung der Szene durch erweiterte Rendering-Effekte wie etwa indirekte Beleuchtung, Reflexionen, Lichtbrechungen und weiche Schatten. Dies könnte den Prozess der Anpassung klassischer Spiele an moderne Standards erleichtern – insbesondere, wenn die Originaldaten nicht verfügbar oder verloren sind.

Dieses Projekt stützt sich auf ein differenzierbares Rendering-Framework, das in der Rendering and Modeling Group (Prof. Wimmer, TU Wien) entwickelt wird. Das Ziel dieser Bachelorarbeit ist es, die Fähigkeiten einer auf diesem System aufbauenden inversen Rendering-Methode zu evaluieren und zu demonstrieren. Konkret geht es um die Schätzung von Lichtquellen aus bestehenden, vorberechneten Lightmaps, die im Spiel Quake III Arena verwendet werden. Daher besteht unser erster Schritt darin, die Geometrie und Lichtkarten aus einer Szenendatei zu extrahieren, die Daten in das Rendering-Framework zu importieren und dann ein geeignetes Optimierungsschema zu implementieren, um neue Lichtquellen zu konstruieren. Mit diesen neuen Lichtquellen können wir die Szenen mit Raytracing und globaler Beleuchtung rendern, um realistische Beleuchtungseffekte zu erzielen – einschließlich indirekter Beleuchtung, genauer Reflexionen und weicher Schatten.

# Abstract

In this thesis, we present a novel application of inverse rendering through the use case of estimating light source parameters from baked lighting information bundled with a 3D scene. As input, we use a scene file that contains the geometry and the baked lighting information of a scene. The extracted scene information is then used to estimate a lighting configuration. With the resulting lighting configuration, it is possible to reproduce a closely matched shading of the scene.

Because of hardware limitations, real-time rendering applications such as video games have historically provided limited realism in terms of lighting. Baked lighting was a common method used to illuminate the scene under real-time constraints. Nowadays, hardware-supported ray tracing enables dynamic lighting and global illumination in real time. Our method aims to build a physically based lighting setup that comes as close to the baked lightmaps as possible. This reconstructed lighting setup allows the relighting of the scene through advanced rendering effects such as dynamic lighting, indirect lighting, reflections, refractions, and soft shadows. This could facilitate the process of bringing classic games up to modern standards, especially when the original data is unavailable or lost.

This project relies on a differentiable rendering framework under development in the Rendering and Modeling Group (Prof. Wimmer, TU Wien). The goal of this bachelor's thesis is to evaluate and demonstrate the capabilities of an inverse rendering method built on this system. Specifically, we aim to estimate light sources from existing precalculated lightmaps used in the game Quake III Arena. For this purpose, our first step is to extract the geometry and lightmaps from a scene file, import the data into the rendering framework, and then implement a suitable optimization scheme to construct new light sources. With these new light sources, we can render the scenes using ray tracing and global illumination to achieve realistic lighting effects, including indirect lighting, accurate reflections, and soft shadows.

# Contents

# CHAPTER 1

# Introduction

Real-time rendering with realistic lighting – including effects such as dynamic lighting, global illumination, indirect lighting, reflections, and refractions – is a highly demanded topic in video games and virtual reality (VR) applications. Because of hardware limitations, real-time rendering applications remain challenging. There is always a trade-off between quality and rendering time. Recent advances in hardware, particularly the introduction of hardware-assisted ray tracing, have made it possible to use ray-based rendering techniques within the constraints of real-time rendering. As a result, the quality of real-time rendered scenes has improved significantly, allowing effects such as dynamic lighting and global illumination to be applied in real time.

In the past, a common approach to meeting real-time constraints was to precompute the lighting and store and distribute it with the game. This precomputed lighting information is often referred to as *baked lighting*. To improve rendering speed and quality while staying within memory constraints, the baked lighting information is highly optimized in terms of access speed and memory usage and stored as lightmaps to fit the rendering process ideally. These lightmaps may be created using physically based lighting models, simplified inaccurate lighting models, or even hand drawing by artists. However, modern ray-based techniques, such as photon mapping [Jen96, PDC$^{+}$05], have emerged, allowing for advanced effects in real time [SA19]. Ray-based methods, such as ray tracing or path tracing, estimate the scattering of light in a scene by tracing the path of light backwards to estimate the color of each pixel. By extending these techniques to include photon mapping, even more realistic results can be achieved, including the capture of caustic effects. Unlike baked lighting, ray-based methods allow advanced effects such as global illumination with dynamic lighting and dynamic geometry because they can respond to changes in lighting and geometry. For our goal of recomputing a physically based lighting setup that closely matches the precalculated lighting environment, photon mapping is the best choice.

The goal of our method is to construct a physically based lighting setup that produces lighting as close to the baked lightmaps as possible, based on the geometry and the baked lighting information. This reconstructed lighting setup allows the scene to be re-lit using advanced rendering effects such as indirect lighting, reflections, refractions, and soft shadows. This could facilitate the process of bringing classic games up to modern standards, especially when the original data is unavailable or lost.

Our project builds upon a differentiable rendering framework being developed in the Rendering and Modeling Group led by Prof. Wimmer at the TU Wien and the inverse rendering method [Pri23] built on top of it. The inverse rendering method uses a radiance target that represents the desired light distribution and is defined directly in the geometry of the scene. Each light source is characterized by parameters such as position, color, and intensity, which are then optimized to achieve a light distribution that closely matches the radiance target. The radiance target itself is defined using the lighting information stored in the lightmaps. To this end, the radiance target is used to optimize a physically based lighting setup, resulting in rendered outputs that resemble the target lighting.

We use Quake III Arena – a game developed by id Software in the late nineties – to demonstrate our method. Specifically, we aim to estimate light sources from precomputed lightmaps stored together with the geometry in highly optimized binary scene files. To achieve this, we first extract the geometry and lightmaps from the highly optimized binary scene file and import the data into the rendering framework. Next, we construct the radiance target and implement an appropriate optimization scheme to create a lighting configuration that matches the baked lighting information. The applied optimization scheme aims to minimize the difference between the target and the rendered result. With this new lighting configuration, we can then render the scenes using modern rendering techniques such as ray tracing and global illumination to achieve realistic lighting effects, including indirect lighting, accurate reflections, and soft shadows.

Our contribution includes the following points:

- *Description of the used method:* The method used in this thesis to reconstruct a physically based lighting setup from baked lighting information is introduced. Furthermore, we explain the decisions and changes we make in order to adapt the underlying approach to our specific problem. Chapter 3.
- *Extraction of the scene data:* We extract the geometry and lightmaps stored in the binary scene file. Furthermore, the albedo textures associated with the models are loaded. Chapter 4.
- *Preparation of the data:* Before the data can be used within our differentiable renderer, it must be converted into a suitable format. To this end, a scene graph is constructed, and all the needed assets – such as albedo textures and lightmaps – are associated with the appropriate node. Chapter 4.
- *Preparation of the radiance target:* The main contribution is to transfer the lightmaps and the albedo texture to the radiance target, which is then used by the inverse rendering method. Chapter 4.

- *Implementation of optimization schemes:* We implement different optimization schemes to construct light configurations that produce illumination as close as possible to the reference illumination stored in the scene file. The optimization schemes vary in the initial light setup as well as in the optimization algorithm and its configuration. Chapter 5.
- *Evaluation of the estimated light configuration against the baked lighting:* In the last step, we evaluate the estimated light sources against the baked lighting stored in the scene files. Then we demonstrate the benefit of this generated lighting setup by relighting the scene with ray tracing. Chapter 5.

In Chapter 2, we describe the basics of photorealistic rendering, the limitations of real-time rendering, and some approaches to overcome them as well as provide a short introduction to inverse rendering. The description of the method used in this thesis is provided in Chapter 3. We also explain the changes and choices we make to adapt the underlying approach to our specific needs. Chapter 4 is dedicated to the implementation. Specifically, we describe some of the key aspects of the file format used in Quake III Arena, how we extract the necessary data, and how we transform the data into a usable format. In Chapter 5, we evaluate the inverse rendering method by comparing different optimization approaches. We demonstrate the effect of the initial configuration and the optimization parameters on the final result. In addition, we demonstrate the resulting scenes. In Chapter 6, we highlight the main findings of our thesis and propose an optimization scheme. Furthermore, we mention some possible future work.

CHAPTER 2

# Background

In the field of computer graphics, photorealistic rendering and real-time rendering have been areas of intense research for years. However, the core concept of rendering and generating images based on a scene description has not changed. Although hardware developments have made it possible to apply more advanced rendering algorithms and reproduce real-world effects with increasing detail, one of the key aspects of photorealistic rendering is the scattering of light in a scene. As a result, many rendering algorithms attempt to model the distribution of light in a scene as realistically as possible. Another popular topic in computer graphics in recent years has been inverse rendering. The goal of inverse rendering is to extract scene information from images, i.e., the inverse of the rendering process.

## 2.1 Photorealistic Rendering

Regarding photorealistic rendering, lighting has a major impact on the overall quality of a rendered scene. In the real world, light emitted from a light source is scattered throughout the scene – first illuminating those surfaces that are directly visible from the light source. These directly illuminated surfaces then reflect a fraction of the incoming light and scatter the reflected light further into the scene. In other words, light bounces off surfaces and illuminates areas that are not directly lit. Therefore, illumination can be categorized as direct or indirect illumination as illustrated in the following figures. Figure 2.1 illustrates the effects of direct and indirect illumination. Figure 2.1a shows only direct illumination, i.e., zero indirect bounces; Figure 2.1b shows one indirect bounce; and Figure 2.1c shows five indirect bounces. The effect of indirect lighting is large – especially the effect of the first bounce – and must be considered for an appropriate lighting model. Global illumination techniques aim to produce realistic lighting by approximating a global lighting environment that considers not only direct illumination but also indirect

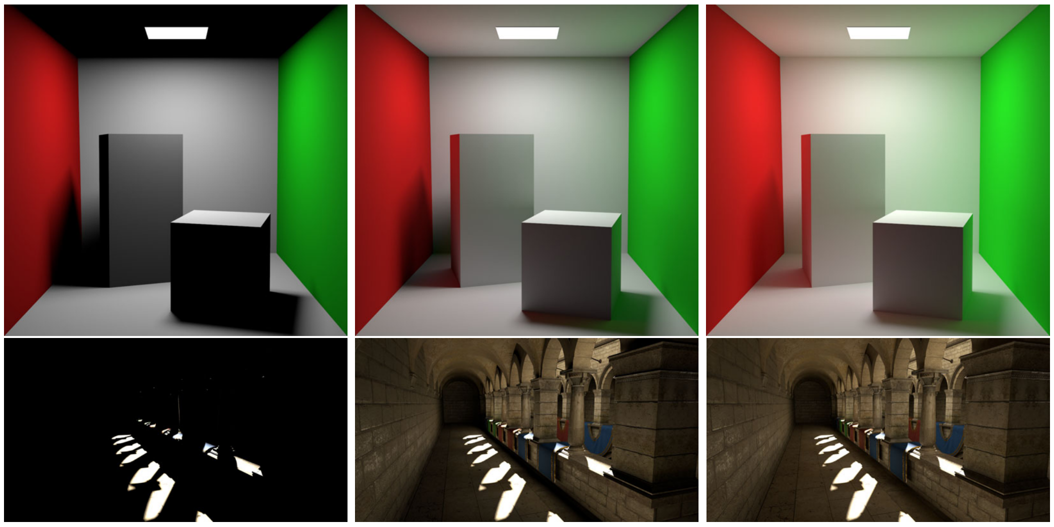


Figure 2.1: Comparison of direct light only (a) and with additional indirect bounces. Direct light only (a), 1 indirect bounce (b), 5 indirect bounces (c). The first row shows a modified version of the Cornell box originally introduced by Goral et al. [GTGB84]; the second row shows a version of Sponza by Frank Meinl. Both scenes were rendered using the rendering framework developed by the Rendering and Modeling Group (Prof. Wimmer, TU Wien).

illumination. Our goal is to reconstruct a lighting configuration so that the resulting global illumination matches the original baked lighting.

The rendering equation [Kaj86] describes the physical scattering of light in a scene, and many global illumination techniques aim to approximate solutions of the rendering equation. It was proposed as a generalization of many rendering algorithms and was based on phenomena in the field of radiative heat transfer [Kaj86]. The rendering equation describes the total amount of light passing from a point toward a certain angle. Specifically, it describes the radiance passing toward an outgoing direction $\omega_o$ for a given point $x$ on the surface of the scene.

$$L_o\left(x, \omega_o\right) = L_e\left(x, \omega_o\right) + \int_\Omega L_i\left(x, \omega_i\right) f_r\left(x, \omega_i, \omega_o\right) \left(\omega_i \cdot n\right) d\omega_i \tag{2.1}$$

- $L_o\left(x, \omega_o\right)$: The total radiance passing from point $x$ toward direction $\omega_o$.
- $L_e\left(x, \omega_o\right)$: The emitted radiance from point $x$ itself toward direction $\omega_o$.
- $\Omega$: The unit hemisphere at point $x$.
- $L_i\left(x, \omega_i\right)$: The incoming radiance at point $x$ from direction $\omega_i$.

- $f_r(x, \omega_i, \omega_o)$: The bidirectional reflectance distribution function (BRDF), which describes the amount of reflected light at point $x$ from direction $\omega_i$ toward direction $\omega_o$.
- $(\omega_i \cdot n)$: The geometric term where $n$ is the normal at point $x$. Describes the effect of incident light on a given point depending on the angle of incidence.

The integral is the sum of the radiance reflected from all directions $\omega_i$ in the unit hemisphere $\Omega$. The amount of radiance arriving at point $x$ from a single direction $\omega_i$ is described by the radiance passing from the point $y$ visible in that direction. This radiance passing from point $y$ to point $x$ can also be described by the rendering equation. Therefore, the rendering equation has a recursive structure, and $L_i(x, \omega_i) = L_o(y, -\omega_i)$.

Many photorealistic rendering methods used in the industry rely on ray-based rendering techniques to physically simulate or approximate the lighting in a scene [KFF+15]. Pixar's RenderMan renderer uses path tracing to render photorealistic images and visual effects for movies [CFS+18]. Disney's Hyperion renderer also uses path tracing to render movies [BAC+18]. Essentially, these ray-based techniques approximate the recursive integral in the rendering equation through Monte Carlo integration. Path tracing and ray tracing approximate the distribution of light by tracing the light backward and passing rays through each pixel to determine its color. Another approach is to trace the light forward from the light source into the scene. This is what photon mapping [Jen96] does. Photons are scattered through light sources into the scene to physically approximate the light distribution.

## 2.2 Real-Time Rendering

Depending on the complexity of the scene, rendering a single frame in a photorealistic scene, such as a movie, can take anywhere from a few seconds to several days. Real-time rendering applications, such as video games, must render at least more than 60 frames per second (fps) to look smooth. To accomplish this, some compromises and limitations must be made to achieve the minimum 60 fps. Light calculation is an expensive operation, so many of the real-time global illumination techniques take advantage of radiance caches to move the costly light calculation from the runtime computation to a precomputation step. This precomputed lighting information is often referred to as *baked lighting*. The shift from runtime to precomputed lighting can improve the quality and performance of real-time rendered scenes, but it reduces the ability to respond to light changes at runtime.

### 2.2.1 Light Mapping

One simple approach to approximate the rendering equation is to calculate the lighting information in a preprocessing step and store it in a texture. These textures are called *lightmaps*. To evaluate the lighting information for a single point on the surface of a scene at runtime, it must be looked up in the precomputed lightmap – this is what light mapping essentially does: it precomputes the lighting information and stores it in a lightmap. Furthermore, because of the nature of indirect lighting, it often does not change much between neighboring points on a surface. Most of the neighboring points have similar lighting. Compared to albedo textures, lightmaps allow storage of indirect lighting information at a low resolution, which saves space without losing much information.

One disadvantage of using precomputed lighting information stored in simple lightmaps is that the radiance is computed for the base surface normal of the mesh and stores a single value for each channel (RGB). Therefore, lightmaps can only store diffuse illumination and do not allow normal mapping. Normal mapping, which increases the visual appearance of surfaces by adding more details by changing the base surface normal, is not possible in combination with lightmaps [Pet16]. The lighting information is restricted to a single value for each color channel, and there is no information stored about the incoming direction. More precisely, the directional light distribution is not stored. Spherical harmonics [SKS02] and spherical Gaussians [CDAS20] store the directional light distribution by projecting the incoming light onto basis functions [Gre03, Pet16], which can then be evaluated for any direction.

### 2.2.2 Spherical Harmonics

To approximate the rendering equation with spherical harmonics (SH), SH probes containing information about the directional light distribution are placed within the scene. To compute the SH, the directional light distribution at each probe location is computed in a preprocessing step, e.g., with Monte Carlo integration techniques. The resulting light distribution is then projected onto basis functions, which are represented by a set of coefficients. A more precise approximation with more basis functions needs more coefficients to be stored. To get the approximated light distribution, the base functions are scaled by the coefficients and then summed [Gre03]. This approach with SH allows evaluation of the lighting information for any direction on the probe location instead of having a single lighting value for a point. For instance, this allows the use of normal mapping in combination with precomputed lighting information.

## 2.3 Inverse Rendering

In contrast to the rendering algorithms mentioned above, inverse rendering, as the name suggests, is the inverse process of rendering. Rendering can be seen as a function that takes a 3D scene description (geometry, materials, lights, camera) and produces a 2D image of that scene. Inverse rendering is the inverse process, starting from a reference image and then producing the scene description. The reference image can be a real-world photograph, an artist's work, or even a rendered image. The goal of inverse rendering [LHJ19, NDVZJ19, ZWZ$^{+}$19, Rob21] is to find parameters, i.e., a scene description that produces a result similar to the target when rendered.

Inverse rendering can be treated as an optimization problem where the scene parameters that produce the best-matching result compared to the reference target must be found. In other words, we want to find the scene parameters that minimize the error between the target and the rendered result. This error between the current result and the reference is described by an objective function, which gets optimized.

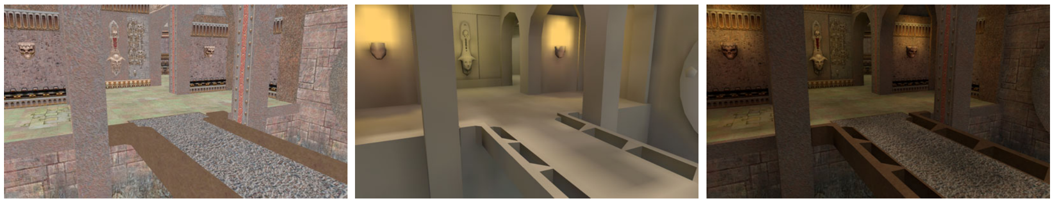
Differentiable rendering allows estimation of the derivatives (i.e., gradient of the optimization objective) with respect to scene parameters [Rob21, SZ21]. These estimated derivatives can be interpreted as the effect of a particular scene parameter on the final image. Differentiable rendering describes a direct connection between a scene parameter and the objective function. With that differentiable rendering function, gradient-based optimization methods, such as gradient descent, can be applied to effectively solve the inverse rendering problem.

The introduction of the first differentiable path tracer in 2018 by Li et al. [LADL18] enabled the handling of secondary effects such as global illumination and shadows with respect to arbitrary scene parameters. Previous work in differentiable rendering, based on differentiable rasterizers, was unable to handle these kinds of effects with respect to arbitrary scene parameters [LADL18]. The differentiable renderer proposed by the Rendering and Modeling Group of TU Wien [Pri23] and demonstrated in this paper relies on a ray-based approach.

Specifically, the application demonstrated in this thesis uses a novel method for view-independent differentiable rendering and considers secondary effects such as global illumination. To this end, a reference target that stores the desired radiance is directly specified in the scene geometry. In a forward light-tracing pass, the current scene configuration is evaluated, and the objective function is applied to the entire radiance data. The application takes advantage of hardware-supported ray tracing and allows the optimization of all lighting parameters with interactive rates, even for complex scenes [Pri23].

### 2.3.1 Optimization Algorithms

As mentioned above, differentiable rendering allows to obtain the gradients of the objective function. These gradients can be used to effectively solve the inverse rendering problem by applying gradient based optimization algorithms. In particular, we use the L-BFGS optimization algorithm [Noc80] and the Adam optimization algorithm [KB14]. The L-BFGS optimization algorithm was developed in 1980 by Jorge Nocedal and is an approximation of the BFGS optimization algorithm with limited memory consumption [Noc80]. It belongs to the family of quasi-Newton methods. The Adam optimization algorithm [KB14] was introduced in 2014 by Diederik P. Kingma and Jimmy Lei Ba. It is an extended version of stochastic gradient descent.



(a) Albedo Texture (b) Lightmap (c) Albedo × Lightmap

Figure 2.2: Quake III Arena scene with albedo texture only (a), lightmap only (b), and albedo texture with lightmap (c).

## 2.4 Quake III Arena

Quake III Arena was developed by id Software and released on December 2, 1999. Id Software released the source code [iS99] for Quake III Arena under the GNU General Public License, which allows many developers to experiment with the game and develop modifications. Many of those projects are built on Quake III Arena and aim to improve the visual quality of the game or even implement real-time ray tracing [Lip20].

The game takes advantage of precomputed lighting information stored in lightmaps. This precomputed lighting is then sampled at run-time to meet the run-time requirements [San12]. The geometry is stored in a data structure called the *binary space partitioning* (BSP) tree, which recursively subdivides the scene into two parts. Together with the lightmaps, the BSP tree is encoded in a binary scene file [San12]. Such highly optimized scene files store precalculated information such as lightmaps or visibility information to reduce the number of calculations needed at run-time. These precalculations allow an increase in the visual quality of rendered scenes while still meeting real-time constraints.

Figure 2.2 shows the extracted scene information from a Quake III Arena scene file. Figure 2.2a illustrates the geometry together with the albedo texture, and Figure 2.2b illustrates the geometry with the lightmap. Figure 2.2c shows the geometry with the albedo texture and lightmap blended together.

CHAPTER 3

# Method

This chapter introduces the method used in this thesis to reconstruct a physically based lighting setup from baked lighting information. Our approach adapts the novel view-independent differentiable global illumination method [Pri23] currently developed in the Rendering and Modeling Group at TU Wien. We explain the decisions and changes we make in order to adapt the underlying approach to our specific problem.

Our method can be separated into the following four parts:

- Loading the scene
- Building the target
- Defining an initial lighting configuration
- Optimization process

The information about the scene is stored in a binary scene file. Therefore, it is necessary to extract geometry and lighting information by parsing the scene file. Subsequently, the extracted information is used to build a target lighting distribution as a reference for the optimization process. Next, it is necessary to establish an initial lighting configuration. This configuration is defined by a set of light sources. Finally, the optimization is applied to update the light source parameters. Figure 3.1 illustrates the optimization pipeline.

## 3.1 Optimization Process

First, we want to give a brief overview of the novel inverse rendering approach [Pri23] we use as our foundation. The proposed method [Pri23] aims to support lighting design problems through differentiable rendering and optimization of a given lighting

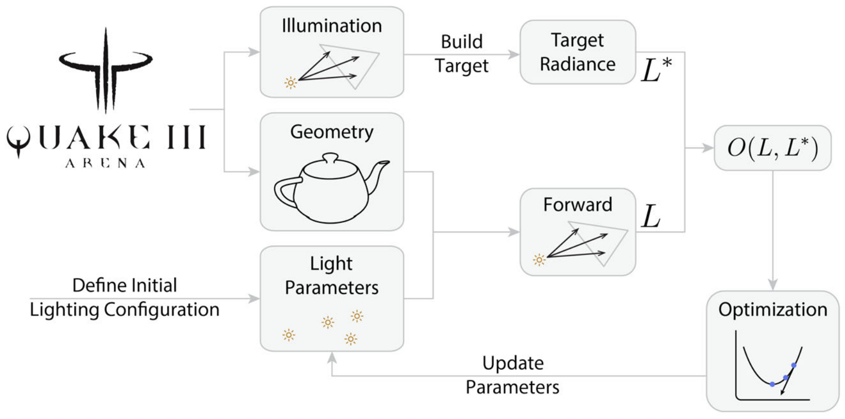


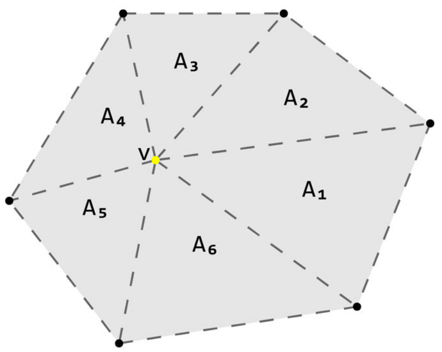
Figure 3.1: Illustration of the optimization pipeline. $L$ and $L^*$ represent the current resulting radiance and the desired target radiance. $O$ represents the objective function that describes the error between two given radiance distributions. Image taken from [Pri23] and modified.

configuration. The goal of the optimization process is to minimize the error between the actual resulting illumination and the target illumination. Before the actual optimization takes place, it is necessary to define a starting light configuration and a target radiance distribution. In our case, we try different initial configurations and use the lighting information baked into the Quake III Arena scene files to construct the target radiance. The actual optimization process remains the same as in the proposed method [Pri23]. First, a forward light tracing pass is performed. Starting from the light sources, a radiance field is generated and stored directly on the scene geometry. Specular highlights are represented by hemispherical harmonics. Second, an objective function is evaluated on the entire radiance data and derivatives are propagated back to the lighting parameters.

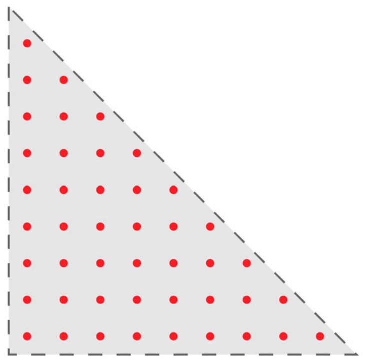
The underlying approach [Pri23] uses adjoint light tracing to obtain the gradient information which can be used to apply gradient-based optimization algorithms. In particular, the Adam and L-BFGS optimization algorithms described in section 2.3.1 are used to reconstruct the desired lighting configuration. We aim to find an optimization scheme to reconstruct the baked lighting of a scene. An optimization scheme is defined by an initial lighting configuration and one or more optimization phases. Each phase consists of an optimization algorithm and its parameters. In Chapter 5 we describe and evaluate different optimization schemes.

## 3.2 Radiance Target

The radiance target stores the desired lighting distribution within the scene and is used during the optimization process to measure the quality of the current lighting configuration. To accomplish this, all radiance information in the scene is stored directly



(a) Illustration of the vertex area $A_\Delta$ of a single point in the scene. The yellow point $v$ indicates the point for which the lighting information is to be determined.



(b) Illustration of the uniform lightmap and texture sampling of a single triangle. Each red dot represents a single lightmap and texture sample.

Figure 3.2: Illustration of the vertex area formed by neighboring triangles as well as the uniform sampling of a single triangle.

at the vertices. For our purpose – the reconstruction of a baked lighting distribution – we use the lighting information stored and shipped with the Quake III Arena game files as a radiance target. Specifically, Quake III Arena uses scene files to store the lighting information of scenes. The lighting information is stored in lightmaps and, to this end, needs to be transformed to the vertices of the scene. We propose a sampling strategy – Chapter 3.2.1 – to project the lighting and color information onto the vertices of the scene. The basic idea is to sample the entire neighborhood of each vertex in a preprocessing step, weight the samples, and store them at each vertex. Figure 3.2a illustrates the vertex area of a single point in the scene and Figure 3.2b illustrates the sampling of a single triangle of that vertex area. We provide a precise description of the sampling strategy in the following Chapter 3.2.1. In Chapter 4, we describe the implementation of the sampling algorithm as well as the entire data transformation process in detail.

### 3.2.1 Sampling Strategy

To build the radiance target, we want to project all the lighting and texture information onto the vertices of the scene. To do this, we compute the area-weighted average of the lightmap and texture for each vertex in the scene.

Our sampling strategy is based on concepts presented by Mats G. Larson and Fredrik Bengzon in their book The Finite Element Method [LB13]. In particular, we take advantage of the *linear shape function* [LB13, Chapter 8.1.2], the *$\mathcal{L}_2$-projection* [LB13, Chapter 1.3] and the *mass lumping* [LB13, Chapter 5.5.1].

First, we define a linear shape function $\varphi_i(\mathbf{x})$ associated with the $i$-th vertex of the scene. Within a triangle, each point can be described by a linear combination of the vertices of that triangle. We define a linear shape function that describes the weight of a vertex for a given point on the surface of the scene. More precisely, for each point $\mathbf{x}$ on the surface of the scene, the linear shape function $\varphi_i(\mathbf{x})$ describes the weight that the $i$-th vertex has at point $\mathbf{x}$. That leads to partition of unity:

$$\sum_j^n \varphi_j(\mathbf{x}) = 1\,, \tag{3.1}$$

where $n$ is the total number of vertices in the scene and $\mathbf{x}$ any point on the surface of the scene.

Second, we define the radiance target $r(\mathbf{x})$ for each point $\mathbf{x}$ on the surface of the scene. The radiance target $r(\mathbf{x})$ at a point $\mathbf{x}$ is defined by the lightmap color multiplied by the texture color at point $\mathbf{x}$.

The goal is to find the radiance target $r_i$ that approximates the given lightmap and texture for the $i$-th vertex of the scene. The approximation $r_i$ should include not only the radiance $r(\mathbf{v}_i)$ at vertex $\mathbf{v}_i$, but also the entire neighborhood of the $i$-th vertex. We define the neighborhood of a vertex $\mathbf{v}_i$ as the affected area of that vertex, or mathematically, all $\mathbf{x}$ on the surface of the scene where $\varphi_i(\mathbf{x}) > 0$.

Subsequently, we can take advantage of the linear shape function and interpolate the per-vertex radiance $r_i$ to receive the approximate target $\tilde{r}(\mathbf{x})$ for any point $\mathbf{x}$ on the surface of the scene. The approximate target $\tilde{r}(\mathbf{x})$ can be described by the sum of all per-vertex radiances $r_i$ multiplied by the corresponding shape function $\varphi_i$ as defined below:

$$\tilde{r}(\mathbf{x}) = \sum_j^n r_j\, \varphi_j(\mathbf{x})\,. \tag{3.2}$$

We are now going to determine $r_i$ via mass-lumped $\mathcal{L}_2$-projection. The standard $\mathcal{L}_2$ projection, minimizing the approximation error $0.5\,\|\tilde{r} - r\|^2_{\mathcal{L}_2(\Omega)}$, is given by

$$\int_\Omega (\tilde{r}(\mathbf{x}) - r(\mathbf{x}))\, \varphi_i(\mathbf{x})\, d\mathbf{x} = 0\,, \tag{3.3}$$

where $\Omega$ refers to the entire mesh.

Expanding the approximate target function and rearranging the terms yields the following equation:

$$\textstyle\sum_j r_j \displaystyle\int_\Omega \varphi_j(\mathbf{x})\, \varphi_i(\mathbf{x})\, d\mathbf{x} = \int_\Omega r(\mathbf{x})\, \varphi_i(\mathbf{x})\, d\mathbf{x}\,. \tag{3.4}$$

Instead of solving this system directly, we apply mass-lumping, which means we replace $r_j$ by $r_i$ on the left-hand side, and simplify the summation due to partition of unity, resulting in

$$r_i = \frac{\int_\Omega r(\mathbf{x})\, \varphi_i(\mathbf{x})\, d\mathbf{x}}{\int_\Omega \varphi_i(\mathbf{x})\, d\mathbf{x}}\,, \tag{3.5}$$

where the integral is now independent of $r_i$ and moved to the right-hand side. Note that this denominator is the vertex-associated area, and we can therefore write

$$r_i = \frac{1}{A_i} \int_\Omega r(\mathbf{x})\, \varphi_i(\mathbf{x})\, d\mathbf{x}\,, \qquad A_i = \int_\Omega \varphi_i(\mathbf{x})\, d\mathbf{x} = \frac{1}{3} \sum\nolimits_{k \in N(\mathbf{v}_i)} A_k\,, \tag{3.6}$$

where $N(\mathbf{x}_i)$ denotes the set of triangles adjacent to vertex $\mathbf{v}_i$. Integrating the shape function yields the 1/3 term of the vertex-associated area definition in Equation 3.6. Finally, we approximate the remaining integral by a basic uniform quadrature, summing over $n$ samples $\mathbf{x}_l$ per triangle $k$ with constant weight $A_k/n$ each, such that $\int_{\Delta_k} f(\mathbf{x})\; d\mathbf{x} \approx A_k \sum_{l=1}^{n} f(\mathbf{x}_l)/n$.

Consequently, we find the radiance target:

$$r_i = \sum\nolimits_k \frac{1}{A_i} \frac{A_k}{n} \sum\nolimits_{l=1}^{n} r(\mathbf{x}_l)\, \varphi_i(\mathbf{x}_l)\,. \tag{3.7}$$

Now that we have mathematically derived the sampling strategy, we can implement it. To do this, we take advantage of the GPU and define a compute shader that runs on it. In Chapter 4, we describe the implementation of the sampling algorithm – illustrated in Algorithm 4.1 – as well as the entire data transformation process in detail.

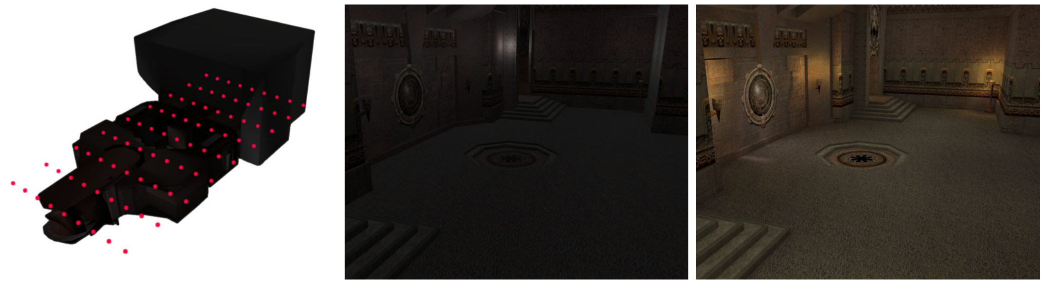
## 3.3 Initial Lighting Configuration

In addition to the target radiance and geometry, the optimizer requires an initial lighting configuration to optimize. Moreover, the optimizer only optimizes existing light sources; no light sources are added or removed during the optimization process. Therefore, the initial lighting configuration has a significant impact on the final result. Apart from the total number of light sources, the initial parameters of each light source also greatly affect the results. A light is defined by its position, color, and intensity, all of which can be optimized.

Our initial lighting configuration consists of a grid of light sources evenly distributed within the bounding box of the scene. The light sources are initially set to a low-intensity white color. Figure 3.3a displays an initial light grid, while Figure 3.3b compares the initial intensity and color with the target illumination in Figure 3.3c. In our evaluation, Chapter 5, we adjust the grid size and aim to find an optimal size for each map.

Because of the simple initial light layout – a uniformly distributed grid of light sources within the scene's bounding box – and the layout of our scene, or during the optimization process, light sources may be placed outside rooms and behind walls. This may affect the rendering time and increase it unnecessarily, but these light sources will not lead to any further problems. Therefore, we are going to ignore light sources that are outside of the scene and do not have a visual effect on the scene.

The following Chapter 4 describes the implementation of our method in detail. In Chapter 4.1, we describe the data layout of the binary scene files, and in Chapter 4.2, we



(a) Initial light grid (b) Initial light color/intensity (c) Target illumination

Figure 3.3: Illustration of the initial light configuration. The initial light layout is illustrated in Figure 3.3a. Figure 3.3b illustrates the scene illuminated by an initial light configuration. Each light source within the initial configuration has a low intensity white color, resulting in a rather dark scene compared to the target scene shown in Figure 3.3c.

explain the extraction of all the data we need. The construction of the radiance target is documented in Chapter 4.3. Chapter 5 compares various optimization schemes and initial starting configurations to determine the best strategy for reconstructing the target illumination.

CHAPTER 4

# Implementation

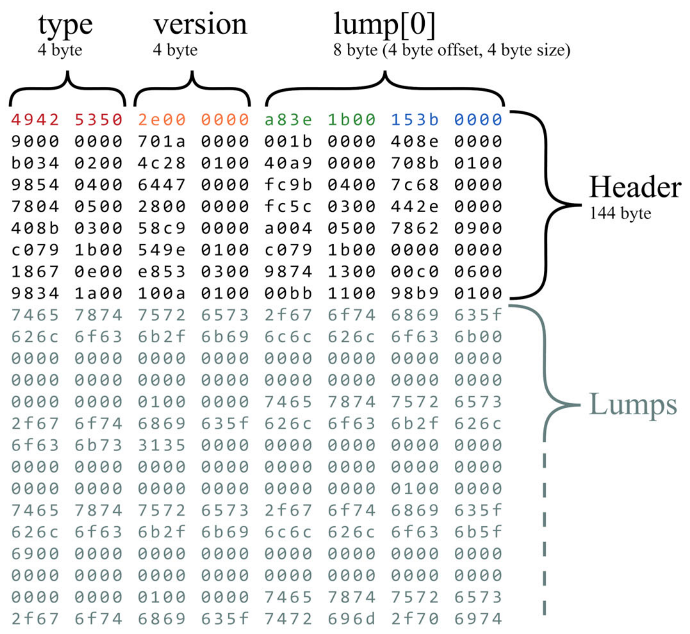
As described in Chapter 2.4, Quake III Arena stores the geometry and the lightmaps in a binary scene file. To use the geometry and lightmaps, we need to extract them from the binary files and convert them into a suitable format. After parsing the BSP file, the geometry, the lightmaps, and the albedo textures are used to describe a radiance target, which is needed to perform the inverse rendering.

The implementation can be divided into the following major steps:

- *BSP file format:* A brief overview of the relevant parts of the BSP file format. Section 4.1.
- *Parsing BSP file:* Loading the binary data from the BSP file and converting them into a readable representation. Section 4.2.1.
- *Extracting the necessary data:* Data such as geometry, lightmaps, or associations between assets must be extracted and bundled into a consistent representation. Section 4.2.2.
- *Constructing the scene:* The scene graph, which is used by the framework to model the scene, must be built. Section 4.2.3.
- *Defining the radiance target:* The extracted albedo textures and lightmaps are used to define a radiance target on the geometry. Section 4.3.

## 4.1 BSP File Format

The BSP file format is a binary file format and was developed by id Software. In the following section, we provide a compact overview of the file format used in Quake III Arena based on the specifications provided by Proudfoot [Pro].



```
4942 5350 2e00 0000 a83e 1b00 153b 0000
9000 0000 701a 0000 001b 0000 408e 0000
b034 0200 4c28 0100 40a9 0000 708b 0100
9854 0400 6447 0000 fc9b 0400 7c68 0000
7804 0500 2800 0000 fc5c 0300 442e 0000
408b 0300 58c9 0000 a004 0500 7862 0900
c079 1b00 549e 0100 c079 1b00 0000 0000
1867 0e00 e853 0300 9874 1300 00c0 0600
9834 1a00 100a 0100 00bb 1100 98b9 0100
7465 7874 7572 6573 2f67 6f74 6869 635f
626c 6f63 6b2f 6b69 6c6c 626c 6f63 6b00
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0100 0000 7465 7874 7572 6573
2f67 6f74 6869 635f 626c 6f63 6b2f 626c
6f63 6b73 3135 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0100 0000
7465 7874 7572 6573 2f67 6f74 6869 635f
626c 6f63 6b2f 6b69 6c6c 626c 6f63 6b5f
6900 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0100 0000 7465 7874 7572 6573
2f67 6f74 6869 635f 7472 696d 2f70 6974
```

Figure 4.1: Snippet from a BSP file illustrating the binary structure. The first 4 bytes represent the file type, the second 4 the version number. The 8 bytes that follow, 4 bytes each for the offset and the size, describe the first lump.

Because of the lack of official documentation, we have to rely on third-party documentation and resources. Specifically, we use the source code review by Fabien Sanglard [San12], the map specification by Kekoa Proudfoot [Pro], and the BSPViewer by Lee Zher Huei [Hue16].

The BSP file format consists of four basic data types: `unsigned byte`, `integer`, `float`, and `string`. The detailed specification can be found in the appendix.

Each BSP file starts with a header followed by a list of lumps containing the actual map data. The header specifies the file type, the file version, and for each lump contained in the file the offset relative to the beginning of the file and the lump size. Figure 4.1 illustrates the structure of the binary file.

The scene data is stored in 17 lumps. Each lump is dedicated to a specific kind of scene information. The most important lumps for our purpose are the lumps describing the surface of the scene, i.e., the geometry and the lightmaps. The following list provides an overview of all the lumps needed to extract the static geometry and the lightmaps:

- *Texture lump:* Textures are stored in separate image files. The texture lump stores the location and name of these texture files in a list.

- *Lightmap lump:* The lightmap lump stores the actual lightmap data in a list. Each lightmap consists of $128 \times 128$ texture elements (texels). Figure 4.2 shows a lightmap stored in a BSP file.

- *Vertex lump:* The vertex lump stores a list of vertices describing the scene geometry. Each vertex consists of a position, albedo texture coordinates, lightmap texture coordinates, and a vertex color.

- *MeshVertex lump:* The mesh vertex lump stores a list of offsets. These offsets are used to describe a triangulation of a surface defined by a set of vertices.

- *Face lump:* The face lump stores information about coherent entities, i.e., about the surfaces of the scene. These entities are called *faces* and each face represents a polygon, a triangle mesh, or a Bezier surface. Each face specifies the associated albedo texture and lightmap as well as a triangulation of the face. Depending on the face type, the geometry-related members of the face have different meanings.

  Faces of the type polygon describe a list of vertices within the vertex lump and a list of offsets within the mesh vertex lump. Each offset in the offset list identifies a single vertex in the vertex list by its offset from the first vertex. Every sequence of three consecutive offsets specifies three vertices that form a triangle. In this way, the vertex list and the offset list are used to describe a valid triangulation of the entire polygon. Figure 4.3 illustrates the composition of a valid triangulation.

  In the same way as polygons, faces of the type mesh describe a valid triangulation by a list of vertices in combination with a list of offsets.

  Entries of the type Bezier surface describe a grid of control points that represent a smooth surface. Each $3 \times 3$ subsection of the grid forms a biquadratic Bezier patch. These patches can be used to describe a triangulation of the face by approximating the smooth surface.

## 4.2 BSP Loader

The BSP loader takes the path to the scene file to be loaded as an input parameter. We parse the file, extract all the necessary information, and convert it into a suitable format. Then we create a scene graph that contains all the static geometry as well as the albedo textures and lightmaps associated with it.

The BSP loader consists of several parts:

- File parsing. Section 4.2.1.

- Mesh construction. Section 4.2.2.

- Scene creation. Section 4.2.3.

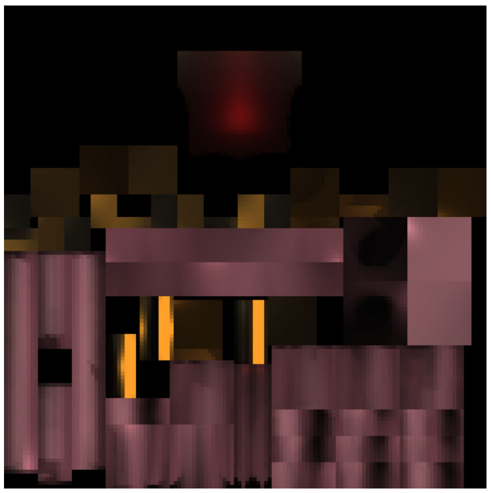
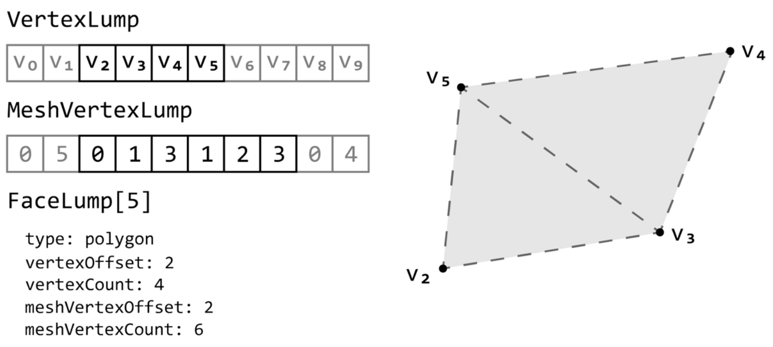


Figure 4.2: Lightmap stored in a BSP file.



```
type: polygon
vertexOffset: 2
vertexCount: 4
meshVertexOffset: 2
meshVertexCount: 6
```

Figure 4.3: Composition of a polygon's triangulation. The polygon is defined by the vertices $v_2$ to $v_5$ within the VertexLump. The triangulation is defined by the offset $o_2$ to $o_7$ within the MeshVertexLump. In this example, the first three offsets (0, 1, 3) describe the points (v2, v3, v5) of the first triangle. The second three offsets (1, 2, 3) describe the points (v3, v4, v5) of the second triangle.

### 4.2.1 File Parsing

The first step is to load the binary file into appropriate data structures. The file header consists of the file type, the version number, and the location of the lumps stored in the file. Therefore, we first load the header from the binary scene file and then check for compatibility. For each lump, the header specifies both the offset from the beginning of the file and the size in bytes. With this pair of offset and size, we load all the needed lumps to construct the static geometry into the corresponding data structures. As a result we get a `Face` list, a `Vertex` list, a `MeshVertex` list, a `Texture` list, and a `Lightmap` list containing the raw data from the file.

We use the `Face` list, the `MeshVertex` list, and the `Vertex` list to create the `BSPMesh` list as described in Section 4.2.2. Each entry in the `Texture` list stores a string, that describes the location and name of the actual texture file. These texture files are loaded via the framework during the scene creation step, as described in Section 4.2.3. Lightmaps are directly stored in the BSP file, and each lightmap consists of $128 \times 128$ texels. Each texel consists of three values (RGB) between 0 and 255. Figure 4.2 shows a $128 \times 128$ lightmap that was stored in a BSP file.

### 4.2.2 Mesh Construction

Faces in the BSP file format represent different entities, such as polygons, biquadratic Bezier surfaces, or meshes – based on the type property – as described in Section 4.1. Depending on the specific type, members of the face have different meanings and represent surfaces in different ways. The optimization framework uses triangle meshes as the surface representation, so we need to convert them into a consistent structure. We convert the different surface representations into an indexed triangle list and bundle this surface representation together with the associated textures in a newly introduced structure called `BSPMesh`. A `BSPMesh` is a simple container that holds the indexed triangle list and the associated albedo texture and lightmap.

Faces of the type polygon define a sublist of vertices in the `Vertex` list by a vertex offset and a vertex count. This sublist of vertices can be used directly in the `BSPMesh`. Furthermore, polygons describe a triangulation of the surface through a sublist within the `MeshVertex` list with a mesh-vertex offset and a mesh-vertex count. This sublist of offsets can also be used directly in the `BSPMesh` as indices.

The faces of the type mesh are similar to those of the type of polygon. Each mesh describes a sublist of the `Vertex` list using a vertex offset and a vertex count. A valid triangulation is described by a sublist of `MeshVertex` defined by a mesh-vertex offset and a mesh-vertex count. Figure 4.4 illustrates the conversion of a `Face` into a `BSPMesh`.

Biquadratic Bezier surfaces do not store a valid triangulation of the surface directly. The sublist of vertices within the `Vertex` list, described by a vertex offset and a vertex count, forms a rectangular grid of control vertices with the dimensions defined in the face. Within the grid of control vertices, each $3 \times 3$ grid forms a patch, with neighboring patches

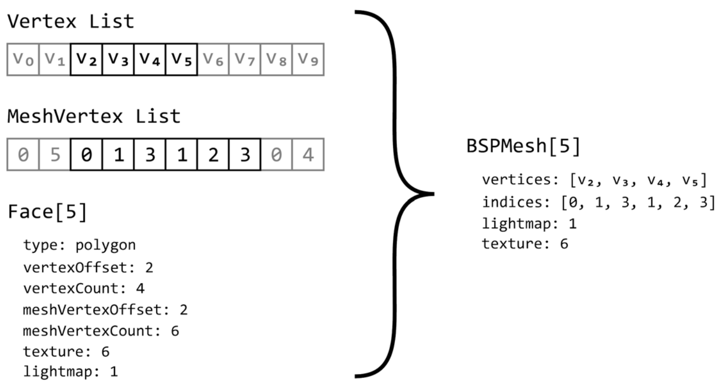


Figure 4.4: Illustration of the transformation and the bundling of the geometric information together with the textures into a unified surface representation.

sharing adjacent vertices. We used these patches to create a triangulation through a tessellation step. To tessellate a patch, each horizontal and vertical line is approximated by a sequence of points. We use the Quake III default of 4 vertices for each line, resulting in 16 points per patch. Figure 4.5 illustrates the tessellation of two adjacent lines.

As a result, we get a consistent surface representation for all the static geometry in the scene as a triangle list. Together with the associated lightmap and albedo texture, the geometric information is bundled into a `BSPMesh` list.

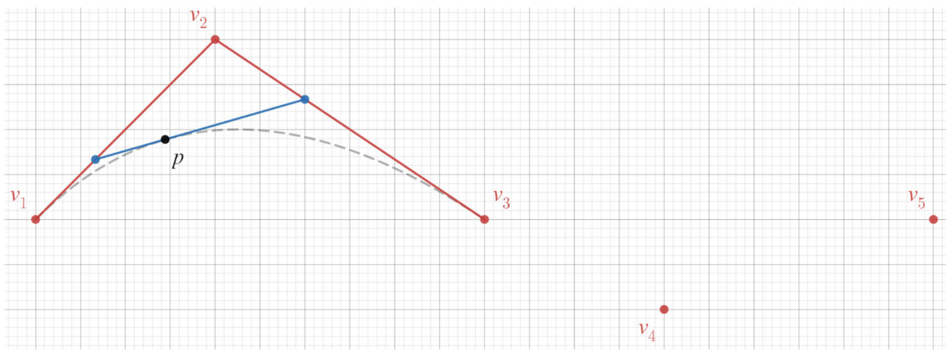
### 4.2.3 Scene Creation

In the last step, we use all the extracted surface descriptions and combine them to construct the scene. The framework internally uses a scene graph to describe scenes, so we insert all the extracted data into an empty scene graph. The vertices in the BSP file are stored directly in world-space coordinates without any instancing. Therefore, we cannot take full advantage of the scene graph. Each extracted surface is inserted as a separate node, bundling the geometry and textures associated with it directly at the root.

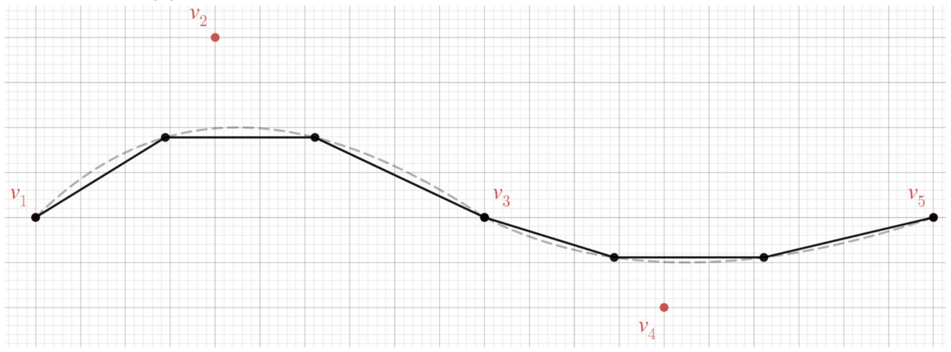
First, we create a scene graph and insert the root node. The root node can be used to translate, rotate, and scale the complete scene. For each entry in the previously created `BSPMesh` list, we create a new node and associate the geometry, albedo texture, and lightmap with it. We attach the node directly to the root node.

## 4.3 Preparing the Radiance Target

At this point, Quake III Arena maps can be loaded and viewed in the framework. The last step that needs to be done is to build the radiance target. It stores the desired radiance for each vertex in the scene. The lighting information extracted from the binary scene file and stored in lightmaps contains the lighting information for the entire surface.



(a) Evaluation of a single point on a quadratic Bezier curve



(b) Resulting approximation of two adjacent quadratic Bezier curves

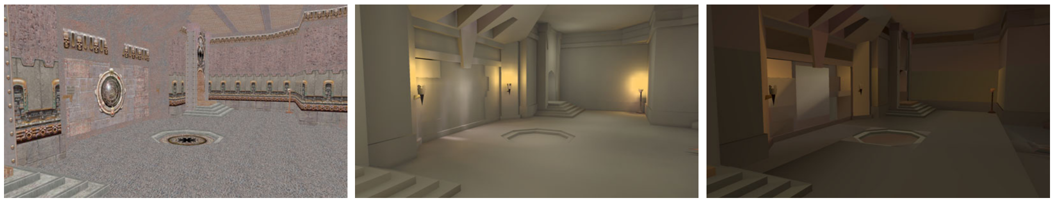
Figure 4.5: Tessellation of two adjacent quadratic Bezier curves; $v_1$, $v_2$, and $v_3$ form the first curve, and $v_3$, $v_4$, and $v_5$ form the second curve.

For this purpose, to build the target, this lighting information must be sampled and bundled at the vertices of the scene. As briefly discussed in Chapter 3.2, for each vertex in the scene, we sample its neighborhood to build the target for that vertex. To perform the sampling, we define a compute shader that is executed for each triangle in the scene. Each triangle in the scene contributes to the radiance target of three vertices – the three vertices that make up the triangle. We take uniform samples of the triangle, as illustrated in Figure 3.2b, and calculate the weight for each vertex within each sample. We then apply the weight and add the weighted sample point to the radiance target. Algorithm 4.1 illustrates the sampling process performed by the shader. Figure 4.6 illustrates the albedo textures and the lightmaps as well as the resulting radiance target.

We use the lightmaps in combination with the textures as the radiance target because the desired lighting setup should illuminate the textured scene similarly to the baked lightmap. The baked lighting can include effects such as indirect illumination, which means that colored light can be reflected from objects onto other objects in the scene. This effect is called *color bleeding* and can be seen in Figure 2.1c, where green light is reflected from the green wall onto the right cube in the scene. To evaluate a current light setup

**Algorithm 4.1:** Target Building

```
1  for each triangle $t$ do
2      $v_0, v_1, v_2 \leftarrow$ vertices of $t$
3      samples $\leftarrow$ barycentric sample positions
4      for $w$ of samples do
5          $w_0, w_1, w_2 \leftarrow w$
6          $x \leftarrow v_0\, w_0 + v_1\, w_1 + v_2\, w_2$
7          $r \leftarrow \texttt{texture}(x) \times \texttt{lightmap}(x)$
8          add to radiance target of $v_0 \leftarrow r \times \frac{w_0 \times \texttt{triangleArea}(t)}{\texttt{vertexArea}(v_0) \times \texttt{length}(\text{samples})}$
9          add to radiance target of $v_1 \leftarrow r \times \frac{w_1 \times \texttt{triangleArea}(t)}{\texttt{vertexArea}(v_1) \times \texttt{length}(\text{samples})}$
10         add to radiance target of $v_2 \leftarrow r \times \frac{w_2 \times \texttt{triangleArea}(t)}{\texttt{vertexArea}(v_2) \times \texttt{length}(\text{samples})}$
11     end
12 end
```



(a) Albedo Texture (b) Lightmap (c) Target

Figure 4.6: Scene with albedo texture only (a). Scene with lightmap only (b). Scene with albedo texture and lightmap projected onto the vertices (c).

during the optimization process, the scene is illuminated, and the resulting illumination – i.e. the light reflected from each surface – is compared to the target illumination. As mentioned above, we want to consider the effects of colored light reflecting from colored surfaces, so we need to use the textured scene to evaluate a current light setup. This leads to the need to include texture information in the radiance target as well.

Now we have parsed the original BSP file and loaded the data into the rendering system, including the geometry as triangle meshes, textures, and the lighting target. We can now build a lighting confirmation that matches the target lighting. At this point, we have not defined any light sources in the scene. Therefore, in Chapter 5, we describe various optimization schemes, including different initial lighting configurations and optimization strategies to build a matching lighting environment.

CHAPTER 5

# Evaluation

With the scene loaded into the framework and the radiance target constructed, we can now perform the inverse rendering to reconstruct a lighting configuration. We describe several optimization schemes to reconstruct a lighting configuration for a given scene and then evaluate the resulting configuration by comparing the objective function and the illuminated scene visually against the target illumination baked into the lightmaps. Each optimization scheme is defined by an initial light configuration and by one or more optimization phases. To set up the initial lighting, we calculate the scene's bounding box and distribute a grid of point lights uniformly within that box. Each optimization phase consists of an optimization algorithm and its parameters.

In a first step, we compare various basic optimization schemes, which are defined by the initial lighting setup and a single optimization phase. Our aim is to demonstrate the impacts of different initial light configurations, optimization algorithms, and optimization parameters. Subsequently, we will assess more complex optimization schemes that involve a sequence of optimization phases. Our objective is to identify an optimization scheme that can reconstruct the most visually appealing lighting setup for a scene.

We use the following three maps to evaluate the lighting reconstruction:

- Map 1: Arena Gate (q3dm1.bsp)
- Map 2: House of Pain (q3dm2.bsp)
- Map 3: Arena of Death (q3dm3.bsp)

Each optimization algorithm has a maximum iteration limit per run. We have set a maximum iteration limit of 400 for Adam and a theoretical limit of 10 000 iterations for L-BFGS, although the latter is never reached because the algorithm terminates before that. When we display the data, we consider only improvements; temporary declines

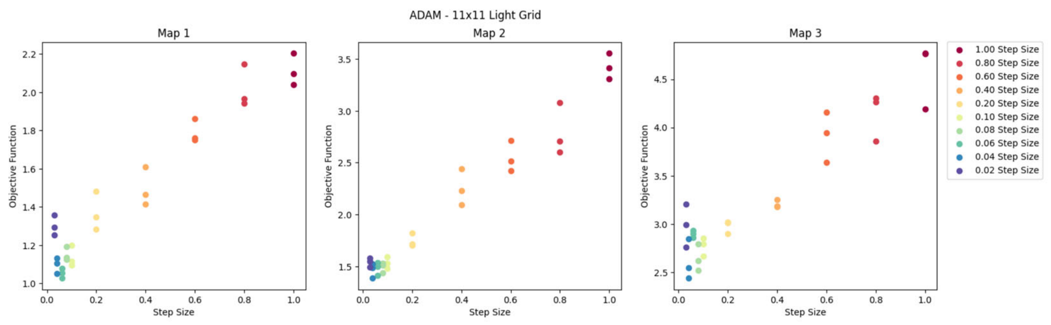


Figure 5.1: Comparison of the impact of different step sizes on all three maps using the Adam optimization algorithm. The results of all three runs of each scheme are shown.

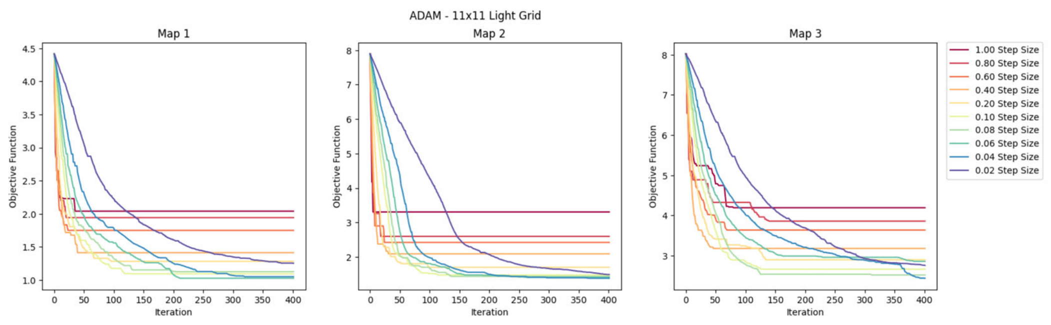


Figure 5.2: Comparison of the objective function during the optimization process of different step sizes using the Adam optimization algorithm.

in performance are disregarded. To evaluate a single lighting configuration, we use approximately 100 000 rays per light source throughout the evaluation process.

## 5.1 Basic Optimization Schemes

First, we compare the effects of different step sizes for both algorithms. To do this, we use an $11 \times 11$ light grid as our initial configuration and 100 000 rays per light source.

We start with the Adam optimization algorithm. To avoid outliers, we have three runs for each configuration and each map. All results are shown in Figure 5.1. Although the sample size of three runs per configuration is not large, it seems that a smaller step size leads to better results for all maps – if the iteration limit is not reached too early.

Figure 5.2 shows the objective function during each optimization run. We choose the best of the three runs to compare. As a result, we can see that a smaller step size may lead to a better approximation but requires more iterations. Furthermore, it seems that the smallest step size (0.02) did not reach the best result because of the limit of 400 iterations.

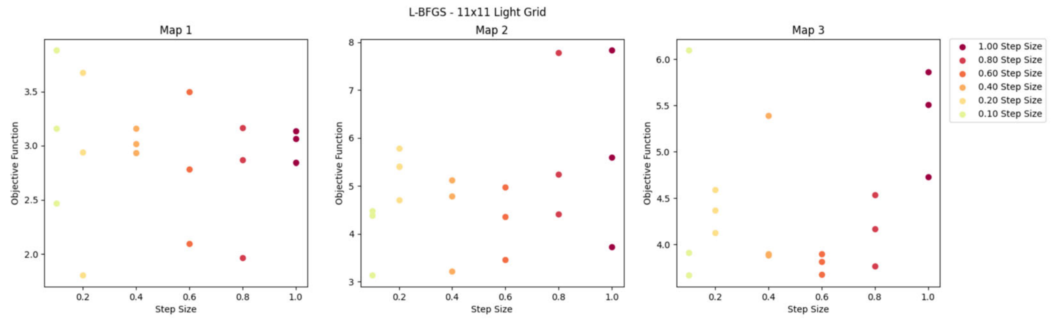


Figure 5.3: Comparison of the impact of different step sizes on all three maps using the L-BFGS optimization algorithm. The results of all three runs of each scheme are shown.

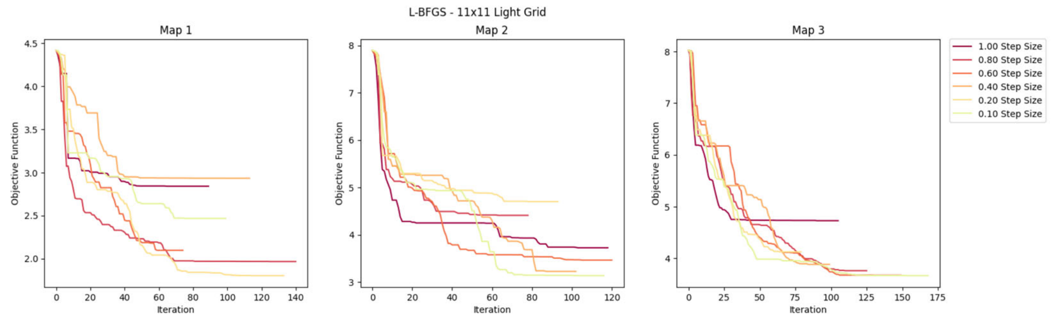


Figure 5.4: Comparison of the objective function during the optimization process of different step sizes using the L-BFGS optimization algorithm.

Now we repeat the same experiment using the L-BFGS algorithm and compare the two algorithms. All the results using the L-BFGS algorithm are shown in Figure 5.3. It is clearly noticeable that the variance between runs with the same configuration is much higher than with the Adam optimization algorithm. Furthermore, this variation does not lead to a clear indication of an optimal step size. It seems that the initial step size does not have much influence on the final results. However, a major advantage of the L-BFGS algorithm is clearly noticeable: It terminates much faster, or terminates at all, before reaching the hard-coded iteration limit. We define an iteration as one call to the renderer. The Adam algorithm requires a single evaluation of the objective during an iteration. However, the L-BFGS algorithm may internally require multiple evaluations of the objective during an single iteration.

Second, we want to compare different grid sizes. We start with the Adam optimization algorithm and a step size of 0.1. Based on the results of the previous example, a step size of 0.1 leads to a good balance between the quality of the result and the number of iterations needed so that the iteration limit is not reached too early. For the comparison, we use a $5 \times 5$, a $7 \times 7$, a $9 \times 9$, an $11 \times 11$, and a $13 \times 13$ light grid.

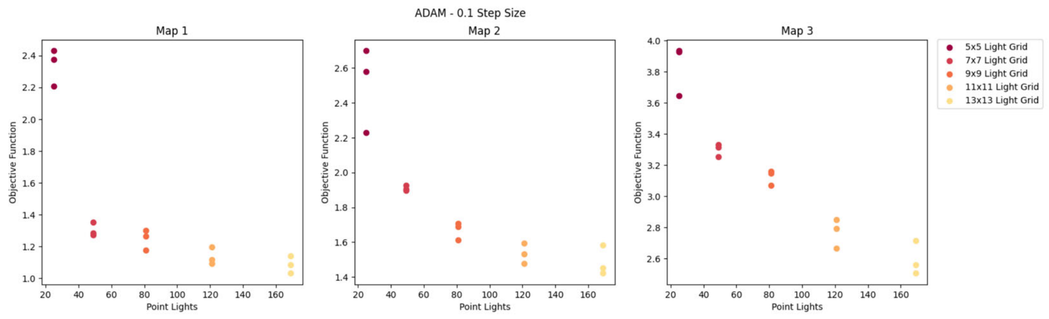


Figure 5.5: Comparison of the impact of different grid sizes on all three maps using the Adam optimization algorithm. The results of all three runs of each scheme are shown.

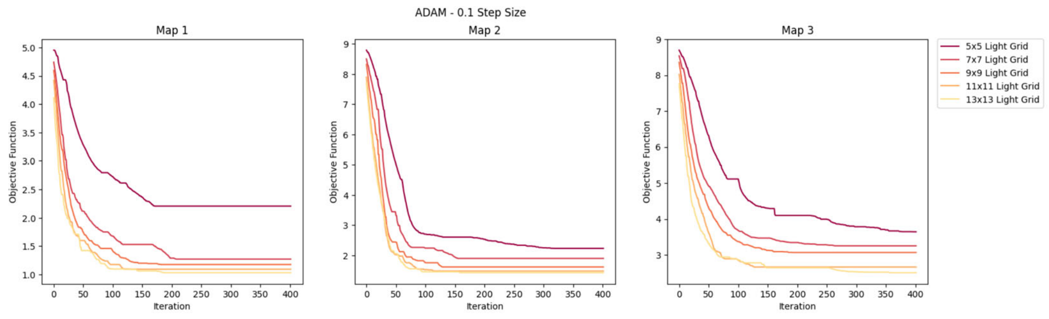


Figure 5.6: Comparison of the objective function during the optimization process of different grid sizes using the Adam optimization algorithm.

As expected and shown in Figure 5.5, a higher number of lights can better approximate the target lighting environment. However, the benefit of more lights is not linear. For example, it appears that a $5 \times 5$ light grid is not enough to approximate the light configuration in Map 1, and there is a large performance gain by using a $7 \times 7$ grid. However, the benefit of a grid larger than $7 \times 7$ is not as great as between a $5 \times 5$ and a $7 \times 7$ grid.

Figure 5.6 shows the objective function during the best run of each setup. It is also noticeable that with more lights, fewer iterations are needed to achieve better results than with fewer lights. However, because of the 100 000 rays per light source, each iteration takes longer to evaluate with more lights. In the end, there is a trade-off between accuracy and runtime.

If we now focus on the results of the L-BFGS optimization algorithm, shown in Figure 5.7, we again see a higher variance within runs. It seems that for map 1, the $13 \times 13$ grid is the best because all three results are better than all other results. However, the best result for map 2 is obtained with the $9 \times 9$ light grid, but the other two results with the $9 \times 9$ grid are within the four worst results.

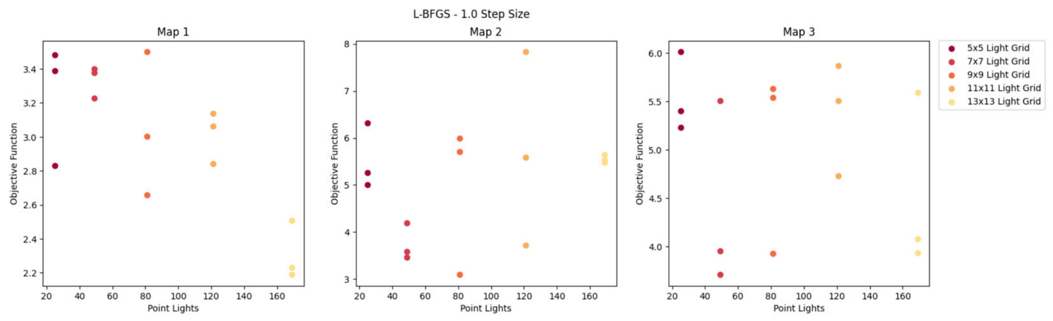


Figure 5.7: Comparison of the impact of different grid sizes on all three maps using the L-BFGS optimization algorithm. The results of all three runs of each scheme are shown.

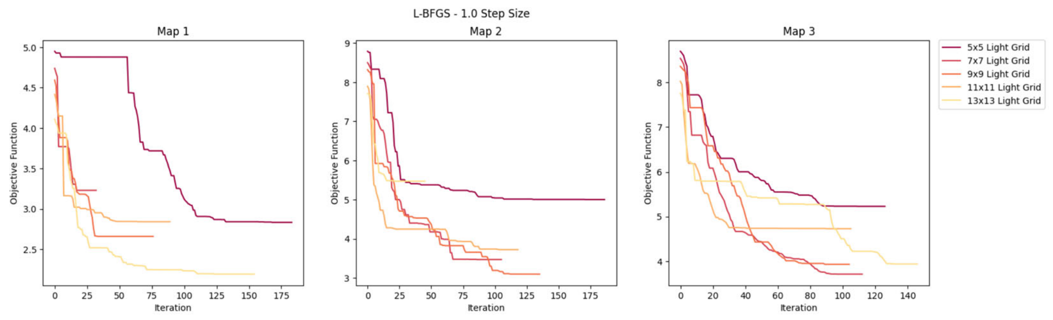


Figure 5.8: Comparison of the objective function during the optimization process of different grid sizes using the L-BFGS optimization algorithm.

Figure 5.8 shows that the L-BFGS algorithm needs fewer iterations compared to the 400 iterations of the Adam algorithm. In our evaluation, the L-BFGS algorithm always stops before 200 iterations. Overall, the L-BFGS is much faster in terms of runtime compared to the Adam algorithm.

Adding more light sources always reduces performance, but adding more lights as needed can also lead to other artifacts. Too many lights can lead to overfitting. A single light source in the original lighting configuration may be approximated by multiple light sources in the reconstructed lighting configuration. This can result in multiple shadows or noticeable bright spots on the surface of the scene.

## 5.2 Complex Optimization Schemes

Now we want to combine different phases to build even better optimization schemes. In addition to the step size and the grid size, we also vary the light parameters that are optimized during a single optimization phase. Specifically, the optimization can be restricted to one or more of the following light parameters: position, color, and intensity. The default configuration of each light created within a light grid is white with an

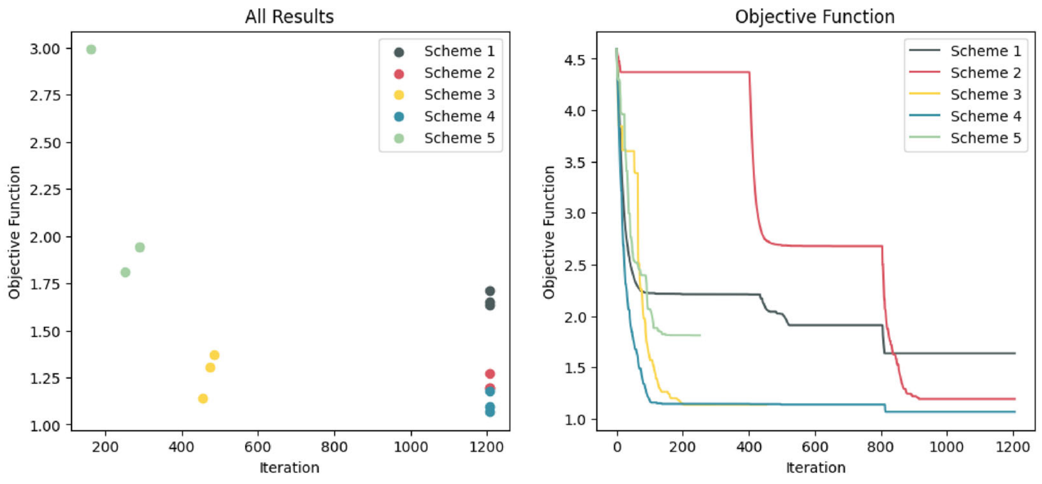


Figure 5.9: Comparison of the results of all optimization schemes.

intensity of 1.0 as illustrated in Figure 3.3b. We demonstrate all optimization schemes on map 1, and we repeat each experiment three times to reduce outliers.

The first optimization scheme consists of three phases. In the first phase, only light intensity and color are optimized. In the second phase, the position is optimized; in the final phase, all three scene parameters are optimized. For all three phases, the Adam algorithm is used with a step size of 0.1 and an iteration limit of 400. Based on the results in Section 5.1, we chose a $9 \times 9$ grid for map 1.

The second optimization scheme is very similar to the first, but we change the order of the phases. First, we optimize only the position of the light sources, then the color and intensity, and finally all three parameters.

In the third optimization scheme, we use the L-BFGS algorithm with a step size of 1.0 in the first phase and the Adam algorithm with a step size of 0.1 in the second phase. In both phases, we optimize all parameters.

The fourth and fifth optimization schemes serve as references. Each scheme consists of three phases in which all scene parameters are optimized. The fourth optimization scheme uses the Adam algorithm with a step size of 0.1, and the fifth scheme uses the L-BFGS optimization scheme with a step size of 1.0. For a compact overview of all optimization schemes, see Table 5.1.

Figure 5.9 illustrates the best result of all three runs for each scheme. The fourth optimization scheme performs best, followed by the third optimization scheme. It seems that the best strategy is to use the Adam algorithm repeatedly. A significant disadvantage is that the fourth scheme terminates slowly. That is where the third scheme excels. In terms of speed/accuracy, the third scheme seems to be the best.

**Scheme 1**

| Phase | Algorithm | Parameter | Step Size |
|---|---|---|---|
| 1 | Adam | Intensity, Color | 0.1 |
| 2 | Adam | Position | 0.1 |
| 3 | Adam | Position, Intensity, Color | 0.1 |

**Scheme 2**

| Phase | Algorithm | Parameter | Step Size |
|---|---|---|---|
| 1 | Adam | Position | 0.1 |
| 2 | Adam | Intensity, Color | 0.1 |
| 3 | Adam | Position, Intensity, Color | 0.1 |

**Scheme 3**

| Phase | Algorithm | Parameter | Step Size |
|---|---|---|---|
| 1 | L-BFGS | Position, Intensity, Color | 1.0 |
| 2 | Adam | Position, Intensity, Color | 0.1 |

**Scheme 4**

| Phase | Algorithm | Parameter | Step Size |
|---|---|---|---|
| 1 | Adam | Position, Intensity, Color | 0.1 |
| 2 | Adam | Position, Intensity, Color | 0.1 |
| 3 | Adam | Position, Intensity, Color | 0.1 |

**Scheme 5**

| Phase | Algorithm | Parameter | Step Size |
|---|---|---|---|
| 1 | L-BFGS | Position, Intensity, Color | 1.0 |
| 2 | L-BFGS | Position, Intensity, Color | 1.0 |
| 3 | L-BFGS | Position, Intensity, Color | 1.0 |

Table 5.1: Compact overview of the optimization schemes.

Looking at all the results as shown in Figure 5.10, we can see that the greatest disadvantage of the Adam optimization algorithm is that it never terminates; therefore, the iteration limit has a significant impact on the accuracy and runtime of the algorithm. Setting a low limit can lead to inaccurate results, while setting a high limit can lead to an excessively long runtime.

Now we want to visually compare the results of each scheme. Figure 5.11 shows the results of the best run for each scheme. The first row shows the resulting radiance, the second row shows the rasterized scene, and the third row shows the path traced scene with the reconstructed lighting configuration.

We can see that the overall lighting ambience of our results is very close to the original lighting stored in the lightmaps. Even potential point lights used to bake the lightmaps

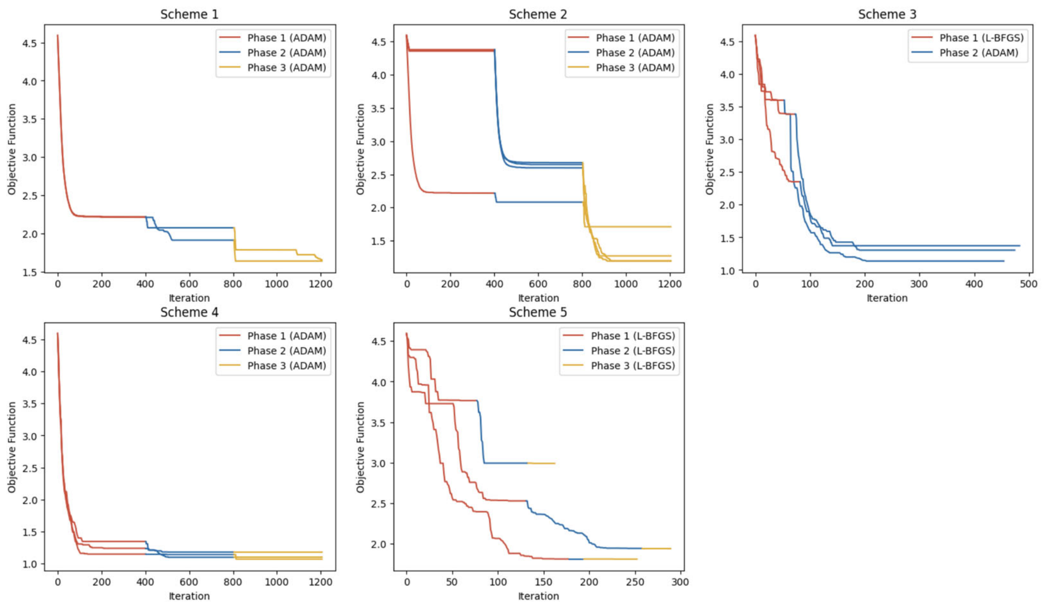
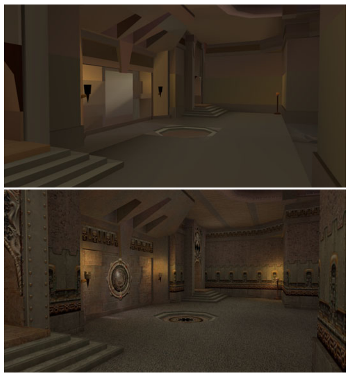


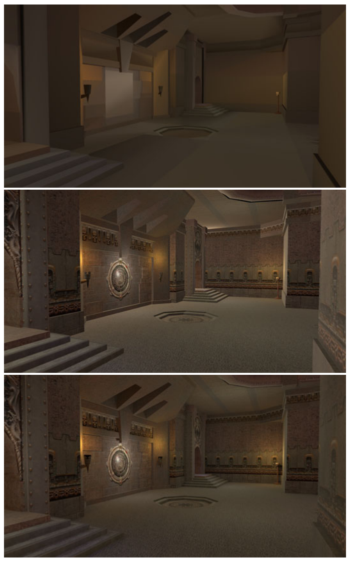
Figure 5.10: Comparison of the objective function of all runs.

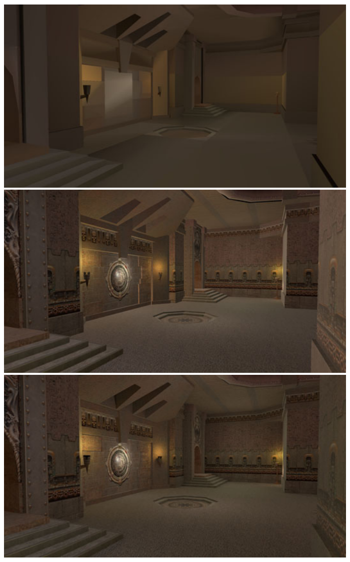
could be successfully approximated, as shown in Figure 5.11e or Figure 5.12.

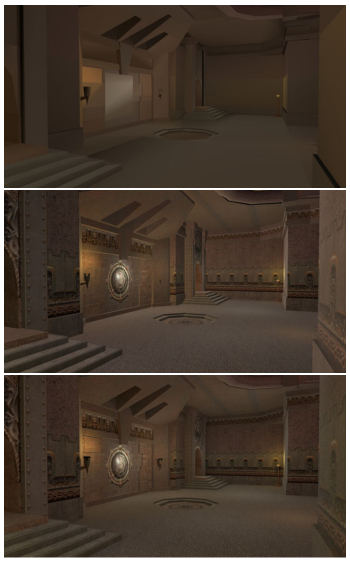
The general lighting mood can be reproduced without any manual adjustments, as shown in Figure 5.11e. However, if we look closely, we can notice some artifacts, such as light sources that are too close to the surface or light sources that are outside the scene. Figure 5.13 demonstrates such artifacts. Fortunately, these artifacts can be easily fixed manually by dragging/removing the problematic light sources. Another approach would be to formulate a penalty term within the objective function to penalize such situations.

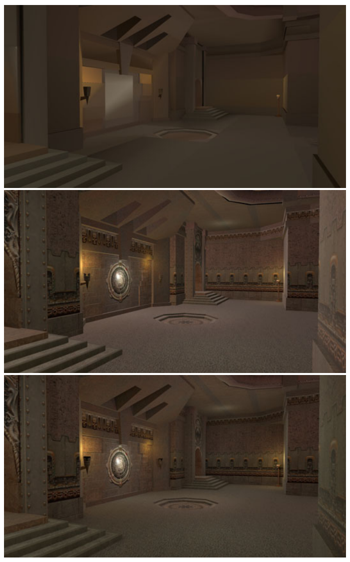
Finally, to demonstrate the potential of the inverse rendering application presented in this thesis, we use the approximated lighting environments to render the scenes with a physically based path tracer. Figure 5.14 shows some of the highlights of each map.


(a) Target


(b) Scheme 1


(c) Scheme 2


(d) Scheme 3


(e) Scheme 4

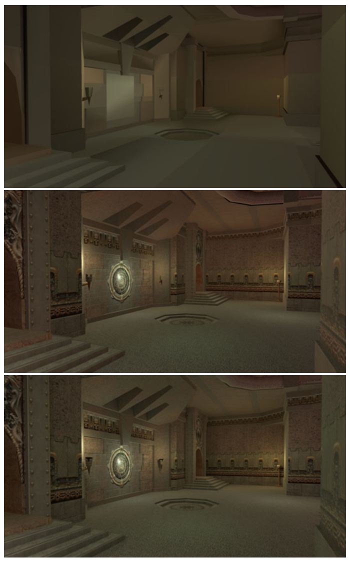

(f) Scheme 5

Figure 5.11: Visual comparison of the best results with respect to the objective function of each optimization scheme. The first row shows the resulting radiance, the second row shows the rasterized scene, and the third row shows the path traced scene with the reconstructed lighting configuration.

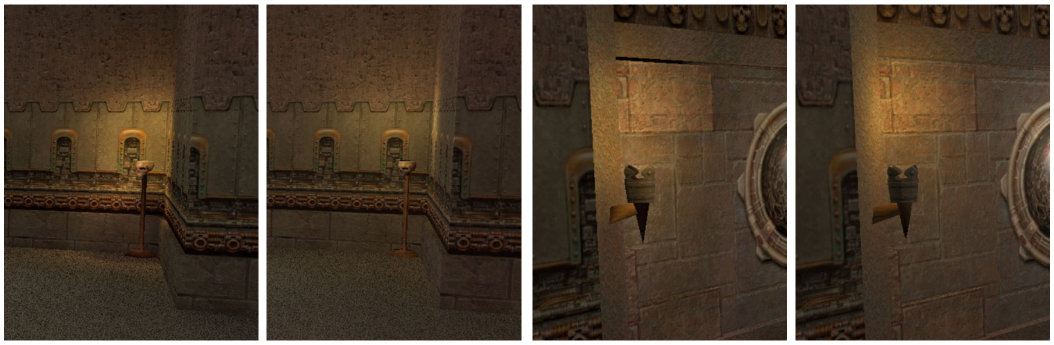

(a) Original (b) Reconstruction (c) Original (d) Reconstruction

Figure 5.12: Comparison of approximated light sources with light sources baked into the lightmaps.

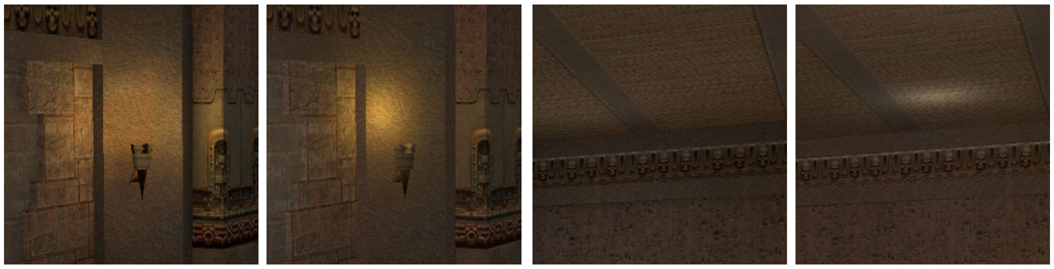
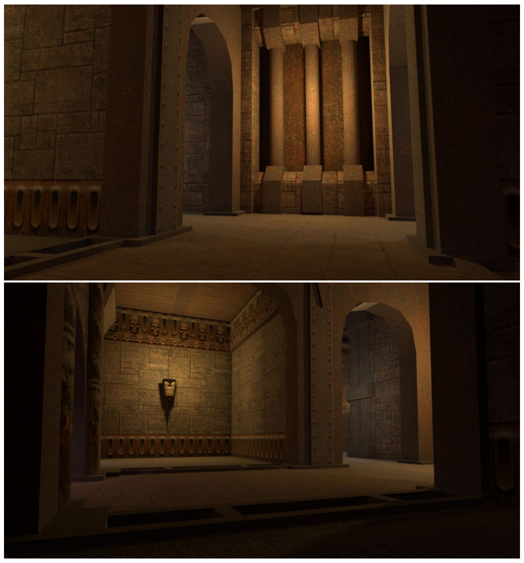
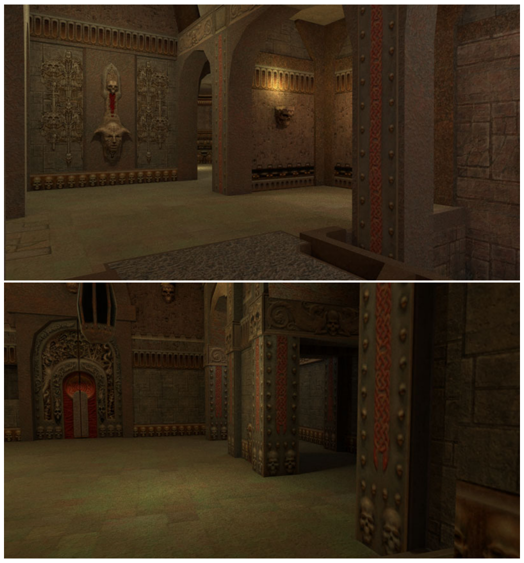

(a) Original (b) Reconstruction (c) Original (d) Reconstruction

Figure 5.13: Artifacts caused by point lights that are placed too near to a surface.


(a) Map 1


(b) Map 2

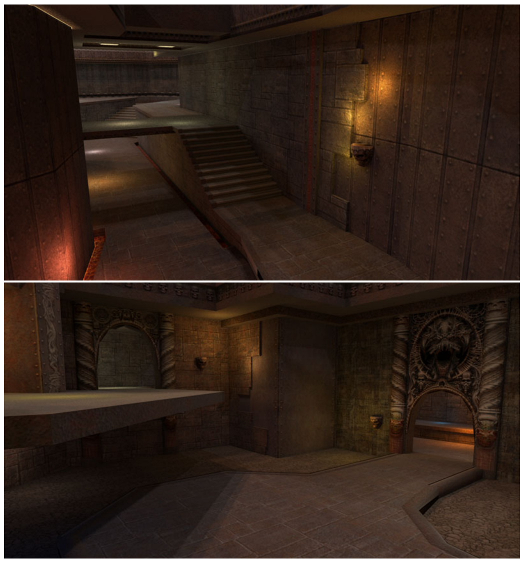

(c) Map 3

Figure 5.14: Path-traced images using the approximated lighting environments.

CHAPTER 6

# Conclusion

In this thesis, we demonstrate the capability of a novel inverse rendering application [Pri23] to estimate light source parameters from baked lighting information. Our methodology involves the extraction of the geometry and baked lighting information from scene files of the game Quake III Arena, the construction of a radiance target from the baked lighting information, and the subsequent application of various optimization schemes to produce a physically based lighting setup that accurately matches the baked lighting information. We achieve a successful reconstruction of the overall lighting ambience for different scenes that preserves the original look of the game. We successfully reconstruct the overall lighting atmosphere for different scenes, preserving the original look of the game and improving the visual quality of these scenes by enabling modern physics-based rendering techniques. Moreover, we demonstrate the use of these physically based reconstructed lighting setups by rendering the scenes using modern techniques, such as global illumination and soft shadows. Our research also illustrates that the initial lighting configuration plays a significant role in determining the quality of the final result. Therefore, the quality of a reconstructed lighting setup is not determined solely by the optimization scheme applied but also by the initial configuration.

To achieve the best result, we can recommend an optimization scheme consisting of several phases in which all light parameters are optimized at once using the Adam optimization algorithm. A smaller step size produces a better result but may increase the number of phases needed to reach the best result. In addition, a single phase using the L-BFGS optimization algorithm at the beginning of the scheme reduces the total number of phases but may lead to a slightly worse result. The light grid chosen for the initial configuration should be as small as possible and as large as necessary. Continually increasing the grid size and comparing the results will reveal a certain grid size where the benefit of adding more lights is negligible compared to the increased complexity.

In conclusion, we highlight some potential areas for further work. As mentioned above, the initial configuration has a major impact on the result, especially because the framework does not dynamically add or remove light sources during the optimization process. Because of the structure of the scenes and the rather simple construction of the initial configuration by placing light sources uniformly within the bounding box of the scene, light sources end up behind walls or outside rooms. To this end, a better initial lighting configuration generation can significantly reduce the total number of lights without compromising the quality of the resulting lighting configuration.

Furthermore, automating the dynamic addition or removal of light sources during the optimization process could improve usability and reduce the impact of the initial configuration.

Another problem we encounter is when light sources are placed too close to the surfaces of the scene, either during the initial configuration generation or during the optimization process. A possible solution would be to formulate a penalty term within the objective function to penalize such situations.

The last possible improvement we want to mention is the subdivision of the triangles in the scene. The radiance target is defined at the vertices in the scene. However, the baked lighting information stored in the scene files describes the lighting information for the entire surface of the scene, not just the vertices. We project the lighting information onto each vertex, and much information is lost. A finer mesh may produce better results, but it may also increase the complexity of the reconstruction process. At some point, a finer subdivision than the lightmap result may not yield much improvement.

We have reconstructed a physically-based lighting environment from lighting information stored in highly optimized binary scene files. We have also analyzed and compared a variety of ways to reconstruct this lighting environment.

# List of Figures

# List of Tables

# List of Algorithms

# Bibliography

[BAC$^+$18] Brent Burley, David Adler, Matt Jen-Yuan Chiang, Hank Driskill, Ralf Habel, Patrick Kelly, Peter Kutz, Yining Karl Li, and Daniel Teece. The design and evolution of disney's hyperion renderer. *ACM Transactions on Graphics*, 37(3):1–22, jun 2018.

[CDAS20] R. R. Currius, D. Dolonius, U. Assarsson, and E. Sintorn. Spherical gaussian light-field textures for fast precomputed global illumination. *Computer Graphics Forum*, 39(2):133–146, may 2020.

[CFS$^+$18] Per Christensen, Julian Fong, Jonathan Shade, Wayne Wooten, Brenden Schubert, Andrew Kensler, Stephen Friedman, Charlie Kilpatrick, Cliff Ramshaw, Marc Bannister, Brenton Rayner, Jonathan Brouillat, and Max Liani. RenderMan. *ACM Transactions on Graphics*, 37(3):1–21, jun 2018.

[Gre03] Robin Green. Spherical harmonic lighting: The gritty details. 2003.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. *ACM SIGGRAPH Computer Graphics*, 18(3):213–222, jul 1984.

[Hue16] Lee Zher Huei. Bspviewer. `https://github.com/leezh/bspviewer`, 2016. Accessed: 23.08.2023.

[iS99] id Software. Quake iii arena gpl source release. `https://github.com/id-Software/Quake-III-Arena`, December 1999. Accessed: 23.08.2023.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Eurographics*, pages 21–30. Springer Vienna, 1996.

[Kaj86] James T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques - SIGGRAPH '86*. ACM Press, 1986.

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

[KFF$^+$15] A. Keller, L. Fascione, M. Fajardo, I. Georgiev, P. Christensen, J. Hanika, C. Eisenacher, and G. Nichols. The path tracing revolution in the movie industry. In *ACM SIGGRAPH 2015 Courses*. ACM, jul 2015.

[LADL18] Tzu-Mao Li, Miika Aittala, Frédo Durand, and Jaakko Lehtinen. Differentiable monte carlo ray tracing through edge sampling. *ACM Transactions on Graphics*, 37(6):1–11, dec 2018.

[LB13] Mats G. Larson and Fredrik Bengzon. *The Finite Element Method: Theory, Implementation, and Applications*. Springer Berlin Heidelberg, 2013.

[LHJ19] Guillaume Loubet, Nicolas Holzschuch, and Wenzel Jakob. Reparameterizing discontinuous integrands for differentiable rendering. *ACM Transactions on Graphics*, 38(6):1–14, dec 2019.

[Lip20] Lukas Lipp. Real-time ray tracing in quake iii. Master's thesis, Research Unit of Computer Graphics, Institute of Visual Computing and Human-Centered Technology, Faculty of Informatics, TU Wien, Favoritenstrasse 9-11/E193-02, A-1040 Vienna, Austria, October 2020.

[NDVZJ19] Merlin Nimier-David, Delio Vicini, Tizian Zeltner, and Wenzel Jakob. Mitsuba 2: A retargetable forward and inverse renderer. *ACM Transactions on Graphics*, 38(6):1–17, dec 2019.

[Noc80] Jorge Nocedal. Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782, 1980.

[PDC$^+$05] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses on - SIGGRAPH '05*. ACM Press, 2005.

[Pet16] Matt Pettineo. Sg series. `https://therealmjp.github.io/posts/sg-series-part-1-a-brief-and-incomplete-history-of-baked-lighting-representations/`, October 2016. Accessed: 23.08.2023.

[Pri23] View-independent adjoint light tracing for lighting design optimization. *submitted to SigGraph Asia 2023*, 2023. private communication.

[Pro] Kekoa Proudfoot. Unofficial quake 3 map specs. `https://www.mralligator.com/q3/`. Accessed: 23.08.2023.

[Rob21] Ariane Robineau. An overview of differentiable rendering. `https://blog.qarnot.com/an-overview-of-differentiable-rendering/`, October 2021. Accessed: 23.08.2023.

[SA19] Niklas Smal and Maksim Aizenshtein. Real-time global illumination with photon mapping. In *Ray Tracing Gems*, pages 409–436. Apress, 2019.

[San12] Fabien Sanglard. Quake 3 source code review: Renderer. https://fabiensanglard.net/quake3/renderer.php, June 2012. Accessed: 23.08.2023.

[SKS02] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques - SIGGRAPH '02*. ACM Press, 2002.

[SZ21] Sai Bangaru Shuang Zhao, Ioannis Gkioulekas. Physics-based differentiable rendering. https://www.diff-render.org/, June 2021. Accessed: 23.08.2023.

[ZWZ+19] Cheng Zhang, Lifan Wu, Changxi Zheng, Ioannis Gkioulekas, Ravi Ramamoorthi, and Shuang Zhao. A differential theory of radiative transfer. *ACM Transactions on Graphics*, 38(6):1–16, dec 2019.

# Appendix

## Detailed File Specification

Since there are no official specifications, we have to rely on third-party documentation. To this end, the following specification is heavily based on the work of Proudfoot [Pro]. This appendix serves as a detailed description of all the data we need to extract from BSP files. Game specific data or other data that are not necessary to construct the static scenes are omitted.

### Data Types

All the structures stored in a BSP file consist of four basic data types and records of these basic types. The basic types are:

| Basic Types | |
|---|---|
| **Type** | **Definition** |
| `ubyte` | unsigned byte |
| `int` | 4-byte integer, little-endian |
| `float` | 4-byte IEEE float, little-endian |
| `string[n]` | string of n ASCII bytes, not necessarily null-terminated |

### File Structure

Each BSP file starts with a header followed by a list of lumps. The header contains the layout of the file and is, therefore, used to determine the location of the lumps stored in it. In the following, we will describe all the lumps we need for the reconstruction of the static geometry and textures associated with it.

#### Header

The header has the following structure.

| **Header** | |
|---|---|
| **Member** | **Definition** |
| `string[4]` type | File type. BSP files distributed with Quake III Arena: "IBSP" |
| `int` version | Version number. BSP files distributed with Quake III Arena: 46 |
| `lump[17]` lumps | Lump's location described by an offset and size. |

| **lump** | |
|---|---|
| **Member** | **Definition** |
| `int` offset | Offset from the beginning of the file to the beginning of the lump. |
| `int` size | Size of the lump. |

### Vertex Lump

The vertex lump stores a list of vertices and each vertex has the following structure.

| Vertex | |
|---|---|
| **Member** | **Definition** |
| `float[3]` position | Vertex position. |
| `float[2][2]` texcoord | Vertex texture coordinates. 0=surface, 1=lightmap. |
| `float[3]` normal | Vertex normal. |
| `ubyte[4]` color | Vertex color. RGBA. |

### MeshVertex Lump

The mesh vertex lump stores a list of offsets.

| MeshVertex | |
|---|---|
| **Member** | **Definition** |
| `int` offset | Vertex index offset, relative to first vertex of corresponding face. |

### Face Lump

The face lump stores information about surfaces in the scene. Each face has the following structure.

| Face | |
|---|---|
| **Member** | **Definition** |
| `int` texture | Texture index. |
| `int` effect | (not used) |
| `int` type | Face type. 1=polygon, 2=patch, 3=mesh |
| `int` vertex | Index of first vertex. |
| `int` vertexCount | Number of vertices. |
| `int` meshVertex | Index of first meshVertex. |
| `int` meshVertexCount | Number of meshVertex. |
| `int` lightmap | Lightmap index. |
| `int[2]` lmStart | (not used) |
| `int[2]` lmSize | (not used) |
| `int[2]` lmOrigin | (not used) |
| `float[2][3]` lmVecs | (not used) |
| `float[3]` normal | (not used) |
| `int[2]` size | Patch dimensions. |

**Texture Lump**

The texture lump contains a list of the path and the name of each texture.

| Texture | |
|---|---|
| **Member** | **Definition** |
| `string[64]` name | Texture name. |
| `int` surface | (not used) |
| `int` content | (not used) |

**Lightmap Lump**

The lightmap lump contains a list of lightmaps. The data of each lightmap are directly stored in the BSP file.

| Lightmap | |
|---|---|
| **Member** | **Definition** |
| `ubyte[128][128][3]` data | Lightmap color data. RGB. |