Universitat de Girona

PhD THESIS
OPTIMIZATION OF INVERSE REFLECTOR DESIGN

Albert Mas Baixeras

2010

PROGRAMA DE DOCTORAT EN TECNOLOGIA

Advisors: Ignacio Martín Campos
Gustavo Patow

Memòria presentada per optar al títol de doctor per la Universitat de Girona

To Judith and Conchi.

Acknowledgments

First of all, I would like to thank Ignacio Martín and Gustavo Patow for advising me on this thesis. To Ignacio for giving me the opportunity of learning from him many scientific knowledge and technical skills. To Gustavo for the same reason, and in addition for his special humanity and advice, pushing me to the next step on the most difficult moments. I would also like to thank Xavier Pueyo for having motivated me to do the PhD program and giving me a lot of support for do it.

I am also very grateful at people of the Geometry and Graphics Group (GGG) for the treatment that I have received during these years and the good work environment. Especially I would like to thank Carles Bosch for the countless hours spend together and his technical advice on my first thesis steps. To Marité, Gonzalo and Toni, for the support received all this time. To the younger PhD students, Isma, Fran, Tere, Oriol and Mei, for the nice and fun moments, the countless coffees and good conversations, suffering my special sense of humor. To the system managers, Robert and Marc. To the secretariat office for helping me on my classical administrative slip-ups.

Finally, I would like to thank all people that have believed in me during these years and had worked hard to help me in my way. To everybody, thanks for convincing me that I could do it.

The work that has led to this thesis has been funded by the following grants and projects:

- Beca de Recerca de la Universitat de Girona BR04/04.
- TIN2004-07672-C03 from the Spanish Government.
- TIN2007-67120 from the Spanish Government
- 7th Framework Programme-Capacities (grant No 222550, project EcoStreet-Light) from the UE

Abstract

This thesis presents new methods for the inverse reflector design problem. We have focused on three main topics that are deeply related with the problem: the use of real and complex light sources, the definition of a fast lighting simulation algorithm to compute the reflector lighting in a fast way, and the definition of an optimization algorithm to more efficiently find the desired reflector.

To achieve accuracy and realism in the lighting simulations, we have used near-field light sources. We present a method for compressing measured datasets of the near-field emission of physical light sources. We create a mesh on the bounding surface of the light source that stores illumination information. The mesh is augmented with information about directional distribution and energy density. We have developed a new approach to smoothly generate random samples on the illumination distribution represented by the mesh, and to efficiently handle importance sampling of points and directions. We show that our representation can compress a 10 million particle rayset into a mesh of a few hundred triangles. We also show that the error of this representation is low, even for very close objects.

Then, we have proposed a fast method to obtain the outgoing light distribution of a parameterized reflector, and then compare it with the desired illumination. The new method works completely in the GPU. We trace millions of rays using a hierarchical height-field representation of the reflector, and multiple reflections are taken into account. We show that our method can calculate a reflector lighting at least one order of magnitude faster than previous methods, even with millions of rays, complex geometries and light sources.

Finally, a new global optimization algorithm has been specifically tailored to minimize the function that calculates the difference between the reflector lighting and the desired one. The optimization is an iterative process where each step evaluates the difference between a reflector illumination and the desired one. We have proposed a tree-based stochastic method that drives the optimization process, using heuristic rules to reach a minimum below a threshold that satisfies the user-provided requirements. We show that our method reaches a solution in less steps than most other classic optimization methods, also avoiding many local minima.

Resum

En aquesta tesi es presenten nous mètodes per al problema del disseny invers de reflectors. Ens hem centrat en tres temes principals que estan profundament relacionats amb el problema: l'ús de fonts de llum reals i complexes, la definició d'un algorisme ràpid de simulació de la il·luminació, i la definició d'un algorisme d'optimització per cercar més eficientment el reflector desitjat.

Per aconseguir resultats precisos i realistes en la simulació de la il·luminació, s'han utilitzat fonts de llum *near-field*. Presentem un mètode per comprimir els conjunts de dades mesurats que defineixen un *near-field*. Per fer-ho, el mètode crea una malla al voltant la font de llum i emmagatzema la informació de la il·luminació. Alhora, a la malla se li afegeix la informació sobre la distribució direccional i la densitat de l'energia. Donat el model comprimit, hem desenvolupat un mètode per generar raigs aleatòriament i per importància sobre la distribució de la il·luminació representada per la malla. Els resultats mostren com es poden comprimir models de 10 milions de partícules en una malla de tan sols uns centenars de triangles. També observem que l'error generat per aquesta representació és molt petit, inclòs per distàncies molt properes a la font de llum.

Llavors, hem proposat un mètode per calcular la il·luminació d'un reflector amb rapidesa, i comparar aquesta distribució amb la il·luminació desitjada. Aquest nou mètode s'executa completament en la GPU i permet traçar milions de raigs amb múltiples reflexions, utilitzant una representació de la geometria en forma de *height-field*. Els resultats mostren com el mètode permet calcular la il·luminació d'un reflector amb, com a mínim, un ordre de magnitud més ràpid que mètodes anteriors, fins i tot amb milions de raigs i geometries i fonts de llum complexes.

Finalment, s'ha dissenyat un mètode d'optimització global adaptat al problema i que minimitza la funció que calcula la diferència entre la il·luminació generada per un reflector i la desitjada. Aquest algorisme d' optimització és un procés iteratiu on en cada pas s'avalua la diferència entre ambdues il·luminacions. Hem proposat un mètode estocàstic basat en la construcció d'un arbre i dirigit mitjançant regles heurístiques, per tal de poder assolir el mínim sota un llindar que satisfà els requeriments especificats per l'usuari. Els resultats mostren que el nostre mètode obté una solució en menys passos que altres mètodes clàssics d'optimització, alhora que evita el processat de molts mínims locals.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Nowadays, lighting design is a very important aspect in our society. It is present in many places where human activities are involved, and it influences life quality and work performance. A good lighting design, driven by energy efficiency and lighting quality, is the key to get correctly lighted environments.

Lighting design is based on the definition of a bulb light source and a reflector, together producing the desired lighting. The bulb and its lighting distribution are usually known beforehand, following rules about power consumption and energy efficiency. The reflector shape should provide the control to manage this distribution and to generate the desired light distribution.

When manufacturers need to create a reflector that produces a desired illumination, an iterative process starts, and a set of reflectors are built and tested. Because the reflector shape is unknown, this process is usually carried out in a very empirical way, since there is not any established engineering process to get the final shape. Therefore, experienced users follow a trial and error procedure, which involves a high manufacturing cost, both in materials and in time. This inverse engineering design is usually carried out with software tools that help designers to model the reflector shape and to perform the lighting simulation. However, the trial and error process is similar, and the time costs are high, too.

Another aspect to be considered is the accuracy in lighting simulations. The reflector shape geometry has to be detailed enough and feasible to be constructed. Moreover, the light source distributions generally used in the design process, called *far-fields*, are based on isotropic point light models, considering that the bulb is placed at a large distance from the measurement point. This is not correct, as in practice the bulb volume is located very close to the reflector shape.

The objective of this thesis is to present new solutions for the whole problem: Inverse reflector design using accurate lighting simulation techniques. First, complex light sources (*near-field* light distributions) have to be considered instead of far-field distributions to get accurate results in lighting simulations. One of the main problems of using complex light sources is the management cost, because this kind of light representations are too huge to be used with classic lighting simulation tools. To solve it, a new compression method will be presented that compresses the light distribution with a minimum error.

Next, we will focus on reflector lighting simulation. During an optimization process, we need to calculate lighting distributions from a huge set of possible reflectors. Therefore, we need to use a very fast method to compute each one of these lighting simulations. We will use GPU techniques to compute and compare the light distributions in a fast way.

Finally, we will define an optimization method to perform the search for the reflector which produces the minimum illumination difference in the family of feasible reflectors. A new global optimization algorithm, that spans the reflector domain in a stochastical way, will be presented, searching for a suitable reflector which produces a minimum difference between the current light distribution and the desired one.

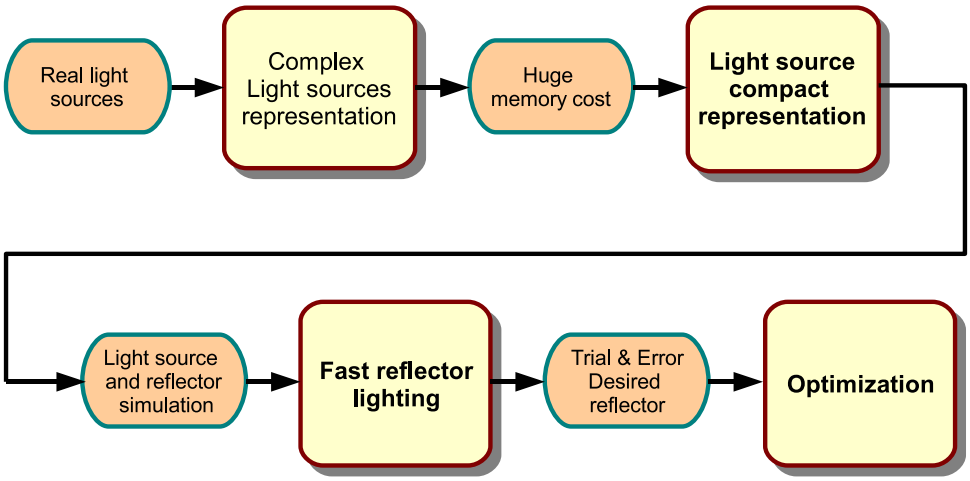


Figure 1.1: Overall scheme of this thesis. The orange boxes are the problems to solve, and the yellow boxes are the solutions. The boxes with text in bold are the developed solutions in this thesis.

1.1 Overview

Reflector design is based on the definition of three components: the light source, the reflector and the desired lighting. Since designers know how the desired lighting must be, we should focus on the light source and the reflector.

The light source is usually defined as a far-field light source. As is said in the previous section, this is not correct for bulbs located very close to the reflector shape. To achieve accurate lighting simulations, we have to use anisotropic light models that are near-field light distributions. The main problem of near-field light representations is the huge amount of data required. They are usually represented as a set of millions of particles captured on a virtual bounding enclosure of the bulb light source. To manage them we need high computational resources, which turns into a problem when we want to use them with classic lighting simulation algorithms. For that reason, we need to define a method to loseless compact the near-field. A clustering algorithm is proposed that creates clusters over the bounding surface where the particle data is similar. The resulting clustering data requires small memory size. Then, a reconstruction algorithm generates samples on demand following the original near-field light distribution with a small error.

The reflector is defined as a surface that represents a constructible design. The reflector must be able to be manufactured through a press-forming process, where the reflector shape is deformed only in one direction. This surface shape, together with the light source, should produce the desired lighting. But usually it is not possible to get this shape directly from the desired lighting, only for very simple cases. In general, we need to resort to some sort of optimization algorithm where we test a huge set of reflectors, from a family of feasible reflectors, and choose the best one. For that reason, a fast lighting simulation algorithm is needed to get the light distribution for each reflector and compare it with the desired one. To achieve it, a GPU ray tracing algorithm is proposed to calculate each reflector light distribution in a fast way. In addition, a GPU algorithm is also defined to calculate the difference between the obtained and the desired light distributions.

Because we want to get a reflector that produces a distribution as close as possible to the desired one, we need an iterative process where a set of reflectors is tested, choosing the reflector with the minimum light distribution difference. That means we need an optimization algorithm to minimize an unknown function, becoming a strongly non linear problem, due can be found many local minima. In general, optimization algorithms can be classified in local or global optimization methods. The local methods do not guarantee that the solution found is the minimum solution, as they tend to fall in local minima, and depend on the starting minimization point. The

classic global methods search for the best minimum solution, but do not fit well to our problem because of its strongly non-linear nature, and the computational time requirements are too high. We propose a new global optimization algorithm, specifically tailored for this problem, as a tree-based stochastic method that drives the optimization process, using some heuristic rules, to reach the desired minimum.

1.2 Contributions

The main contributions of this thesis are :

- Improving current inverse reflector design tools and methods to achieve faster and more accurate results.
- Utilization of complex real light sources in inverse reflector design.
- A fast method to compute the reflector lighting distribution using GPU algorithms
- A new optimization algorithm to get the desired reflector in a fast way.

1.3 Organization of the Thesis

The thesis is organized as follows:

Chapter 1: Introduction. Introduces the motivation of the thesis and describes the interest of the Inverse Reflector Design problem in the context of illumination engineering. Also summarises the contributions of the thesis and its organization.

Chapter 2: Previous work. General Inverse Rendering problems are presented and summarized. Next is described the state of the art for real and complex light source. Then is included a summary of the most relevant methods to render the reflector lighting in a fast way. Finally, is shown a summary of significative local and global optimization methods.

Chapter 3: Compact representation of near-field light sources. Presents the differences on using near-field and far-field light sources for reflector lighting calculation. Due the high computational costs required to manage near-field models, a new compression method is presented that allows to use efficiently real complex light sources for reflector lighting rendering.

Chapter 4: A fast algorithm for reflector lighting. Proposes solutions to calculate the reflector lighting distribution in a fast way. A new adapted method for ray tracing on the GPU is presented. Also most recent GPU capabilities have been tested in the GPU ray tracing field.

Chapter 5: Optimization. Define and fit an optimization algorithm into the inverse reflector design problem. A new global optimization method is presented that allow to reach the solution in a few minutes.

Chapter 6: Conclusions and Future work. Concludes the thesis summarizing its main contributions and pointing out unsolved problems of our approach. Also, possible future directions of research in the context of the thesis are commented.

Chapter 2

Previous work

The work presented in this thesis focuses into obtaining the reflector shape that generates a desired light distribution.

In general, we can consider the light transport theory and the general rendering equation [Kaj86] to solve it:

$$L(p,\omega) = L_e(p,\omega) + \int_\Omega f_r(p,\omega,\omega_i) L_i(p,\omega_i) \cos\theta_i d\omega_i$$

$L_e(p,\omega)$ is the emitted *radiance* at point p and direction ω. The *radiance* is a measure of the flux density per unit solid viewing angle [Rye02], and is defined as the power radiated at point p in a direction ω per unit of projected area perpendicular to that direction, per unit solid angle for a given frequency, that is $[Watt \cdot m^{-2} \cdot sr^{-1}]$. Radiance is commonly used for light sources, that are one of the most important parts of the luminaire, so it defines how the light source rays arrive at the reflector surface. In section 2.1 we will discuss how this light sources are represented to get accurate results.

The scattered radiance from point p in direction ω is the sum of the emitted radiance and the incoming radiance $L_i(p,\omega_i)$ from all directions ω_i controlled by the scattering function $f_r(p,\omega,\omega_i)$ and the light attenuation θ_i from the incident angles. This part of the equation depends on the scene geometry: the reflector shape in our case. Therefore, we consider p as a point on the reflector surface. In section 2.2 we will show a summary of methods to simulate the lighting in a general scene. Moreover, we will focus on those algorithms, with acceleration structures and methods, that fit well to our purpose.

$L(p,\omega)$ is is the radiance exiting from a point p at direction ω. The goal is to get a radiance $L(p,\omega)$ as closest as possible to a desired one. Thus, we need an optimization algorithm to search for a good solution progressively

from a set of suitable reflectors. The most common optimization algorithms will be reviewed in section 2.3.

On the whole, the problem can be classified as an inverse design problem. In section 2.4 there is the state of the art of inverse design problems, focusing on those methods that are in the scope of inverse lighting methods.

2.1 Light source distribution representations

Light sources are objects that emit light defined mainly by the shape, the light distribution and the emittance energy. The light source shape defines how the rays are emitted from the light source surface, or a virtual representation of it. Some classic examples are point light sources (rays are emitted from the same point), spherical light sources (rays are emitted from the surface of a sphere) and area light sources (rays are emitted from a generic surface). The light source distribution specifies how the rays are distributed from the light source into the geometrical space. These distributions are defined often by an analytical function that makes easy the light generation process. This light can be represented as a set of rays, photons or emittance radiance (L_e in equation (1)). Some classic examples are the directional distribution (the most basic, where all rays are emitted in a unique direction), radial distribution (rays are emitted in all directions with the same probability), cosine and Phong distributions (rays are emitted following a cosine or Phong lobe, where ray density is greater as the ray directions are closer to the light surface normal vector). Since the mentioned light sources try to simulate real light sources, more detailed representations are needed to get accurate results. Real world light sources are represented as complex light sources, using more detailed shapes and light distributions. There are two kind of complex light source representations: far-field and near-field.

A far-field representation models a luminaire as an anisotropic point light source, and assumes that objects to be illuminated are located far away. As stated in [Ash93], the distance to consider a light source as a point light is about seven times the longest dimension of the light source bounding volume. There are some established standards to represent far-field light sources, the most important being IESNA and EULUMDAT [ANS02][bCL99]. However, far-field representations do not produce accurate results when objects are close to the light source.

The near-field models represent real light sources, modelled as extended light sources, without any assumption on the exact light source placement. The standard procedure to construct a near-field is to take several raw images from the light source. In [Ash93] it was proposed a simple acquisition

system controlled by two degrees of freedom (see Fig. 2.1). A photosensor device turns around the light source taking images at each step. At the same time, the light source can be rotated over itself. The result is a goniophotometer capable to get, over a light source bounding sphere, a set of particles composed by points and directions.

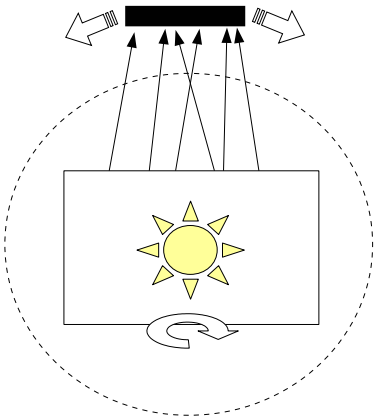


Figure 2.1: Goniophotometer system description, as is shown in [Ash93]

In recent years there has been an important effort to improve real light source capture and representation methods using near-field models. However, the raw data produced by these capture methods can produce several gigabytes of images. Captured data can be compressed to a light field [LH96][GGSC96] or interpolated to generate a rayset [AR98].

A light field is usually represented with a Lumigraph [GGSC96]. A Lumigraph is a 4D structure that is created from the relationship between two levels of 2D structures or slices (see Figure 2.2). Each cell at the first level is linked with one slice at the second level. The second level slices store the radiance for each ray, represented by a 4D parameterization. The quality of the represented light field depends on the slices resolution and the distance between them. The first approach to generate a light field was proposed in the canned light sources method [HKSS98]. They used a representation similar than the one proposed in [Ash95], computing a light field to be used later in a ray tracing algorithm. The main drawback of this method is that they do not show any way to use importance sampling for the light field to be used in a light ray tracing method, and it is not suitable to be used at short distances due the regular pattern of the light field. Another method is presented in [GGHS03a] that solves these drawbacks. However it needs around 100Mb per light using an efficient memory representation.

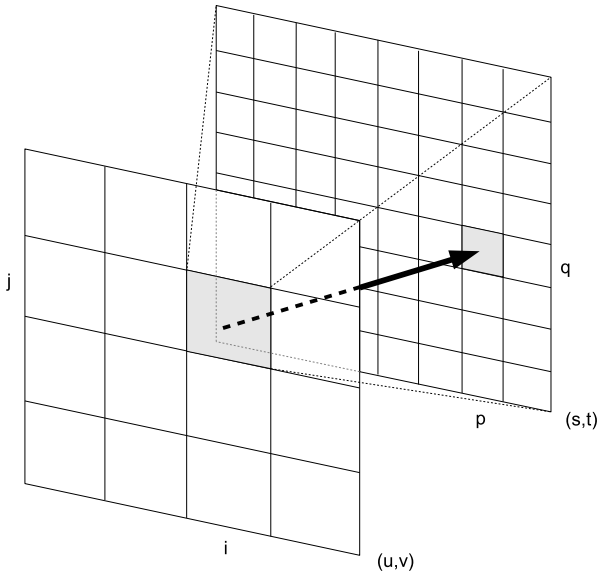


Figure 2.2: Lumigraph description. Each ray is represented by a 4D parameterization of a pair of cells in two parallel planes. In the example, the ray is represented by (u_i, v_j, s_p, t_q), where u_i,v_j and s_p,t_q are the cell coordinates for first and second planes respectively.

A rayset is a set of particles (point, direction) with equal energy and without spectral distribution, emitted at the light source and captured in a virtual bounding surface. The capture process uses a gonio-photometer that could be mounted on two rotating arms that allow the device to capture the light coming from a source from all possible directions, but other setups are also possible [AR98][GGHS03a][GGHS03b]. This representation is the industry standard for optical illumination analysis software [Rad][Lam][Bre, OPT]. Nevertheless, a rayset dataset needs around 220Mb of memory allocation for a light source with 10M particles.

2.2 Lighting simulation

For a given reflector and light source we have to compare the generated light distribution with the desired one. Thus, we need to perform a lighting simulation (rendering) to get the reflector light distribution. Although a direct lighting simulation method would be enough, we need realistic renderings to get accurate results. Therefore we need direct and indirect lighting methods (global lighting algorithms). On the next section we present a summary of

the most relevant global illumination methods. Next, we focus on techniques to accelerate the rendering times using current GPU capabilities.

2.2.1 Global illumination

Lighting simulation for realistic rendering is performed using *global illumination* techniques. There are three main aspects to consider for global illumination: the light sources, the light transport through the scene and the light interaction with scene objects. All of them can be represented using the rendering equation (1) as it is explained at the start of the current chapter. The scattering function $f_r(p, \omega, \omega_i)$ is usually defined as a BSDF (Bidirectional Scattering Distribution Function), and defines how the incoming rays ω_i are related with the outgoing direction ω at point p. This function is commonly the composition of two other functions: BRDF and BTDF. The BRDF (Bidirectional Reflectance Distribution Function) defines how the rays reflect over the surface [Pho73][Bli77][Whi80][CT81][Sch94][AS00]. The BTDF (Bidirectional Transmittance Distribution Function) define how the rays are transmitted through the surface [He93]. In addition there is the BSSRDF (Bidirectional Surface Scattering Distribution Function), that covers those materials where the ray is transmitted through the surface, and scattered inside, entering in a point and outcoming in a different one, like translucent materials or layered surfaces [HK93][JMLH01]. The BSDFs can be considered a subset of the BSSRDFs. For our case only will be considered the BRDF of the reflector surface.

According on how the rendering equation is solved, there are different global illumination techniques. Some examples are *ray tracing-based* techniques [Gla89], such as *path tracing*, *bidirectional path tracing*, *metropolis light transport*, or *photon mapping*; and *radiosity* techniques. In the following subsections we are going to explain in more detail these techniques.

2.2.2 Ray Tracing

In [Whi80] it was introduced the ray tracing technique. From the eye, so for each pixel of the image on the viewport, a ray is traced through the scene. The ray intersection point is calculated with the *ray casting* algorithm. When an intersection point is found, three new rays could be considered. First, the reflected ray, that is traced again through the reflected direction, taking into account the BRDF. Second, the refracted ray, tracing the incoming ray through a translucent or transparent material, taking into account the BTDF. Third, the shadow ray, where the current point is checked in order to know if it is in shadow or not, tracing the ray until reach the light sources. If any ray

intersects first with an opaque object, the current point is in shadow. The three rays and the current point material properties are used to calculate the shading for this point, and the final color for the pixel image. This iterative process stops when no more ray intersects with geometry, or when a maximum number of bounces is reached.

The ray tracing is a view dependent algorithm, so if the eye position or view direction changes, the scene must be rendered again. This is necessary to get realistic details such as reflections or specular effects. Also, the quality of the results depends on the image resolution, but more rays are traced into the scene if not enough rays are used. This can produce aliasing effects. To solve it, multiple rays are sampled for each pixel. In [CPC84] it was presented some improvements to get more realistic effects, such as motion blur, depth of field, shadow penumbra and fuzzy reflections.

A variation of ray tracing, taken from another point of view, is *light ray tracing*. This method, first presented in [Arv86], computes the light rays from the light sources through the scene, until no more geometry is intersected (see Figure 2.3). The light rays are sampled from the light source distribution description using Monte Carlo methods (see next section). Furthermore, shadow rays are not used, and the technique is view independent. This method is usefull when it is required to obtain the outgoing light distribution on a scene, like for example reflector lighting. The number of light rays traced will depend on desired light distribution quality.

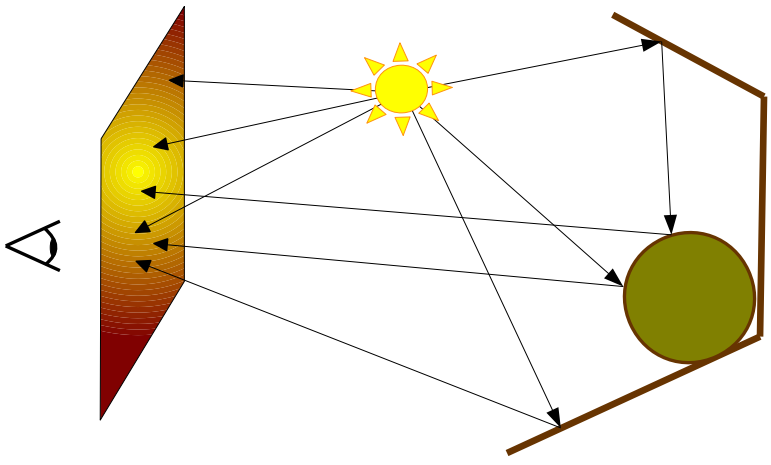


Figure 2.3: Light ray tracing description.

Monte Carlo methods

In [Whi80] and [CPC84] it was introduced the idea of stochastic methods to sample view rays or light sources. This led to the rendering equation [Kaj86] and the first unbiased Monte Carlo transport algorithm, called *Path Tracing*. The main difference with ray tracing is the reflected ray generation, that is sampled stochastically on the hemisphere over the intersected point. Also, the shadow ray is not traced. The idea is to bounce recursively the ray through the scene until it reaches any light source. This path is commonly known as *random walk* The main drawback of this method is the need to trace a large amount of rays per pixel to get enough sampled paths, and consequently accurate results.

To improve it, in [LW93] and [VG994] it was presented *Bidirectional Path Tracing*. Two random walks are traced, one from eye view, like in path tracing, and the another one from each light source. The light ray is also sampled by Monte Carlo methods. Then, all hit points for the respective paths are connected together using shadow rays, and the appropriate contributions are added to final result.

In [VG97] it was presented the *Metropolis Light Transport* algorithm, based on a variation of Monte Carlo Metropolis. This method improves bidirectional path tracing for effects such as caustics, or cases when there is concentrated indirect lighting. From the traced light rays, each path is mutated inserting new points in the path and new propagation directions. The mutations are done by sampling a probabilistic function based on how the light arrives to the eye.

Photon Mapping was presented in [Jen96] and [Jen01]. This method improves the previous bidirectional path tracing and metropolis light transport algorithms in the way of obtaining realistic results with complex effects, such as realistic caustics, diffuse interreflections, or subsurface scattering. In a first step, the algorithm traces the photons trough the scene. The photon hits are stored in a KD-Tree (see Section 2.2.2), also called *Photon Map*. Also, the absorbed energy by the surface, that is the indirect illumination, is specified by a stochastic choice, where the photon energy is absorbed, reflected or refracted based on the surface material properties. Another map, the *Caustic Map*, stores the photons whose previous intersection is on a specular surface. The caustic map is a subset of the photon map, but it has to store more samples to get accurate caustics results. In the next step, view rays are traced through the scene. When a hit succeeds, a density estimator gets the radiance for the intersected point from the interpolation of the nearest photons on the photon map (see Figure 2.4). This radiance is accounted for together with the direct illumination, obtained by tracing a ray to the light sources;

the specular lighting, obtained using the classic ray tracing; and caustics, calculated from the caustics map and using also a density estimator. Unlike previous methods, this is a biased method, which means that results could be incorrect. But this is usually solved by increasing the number of sampled photons.

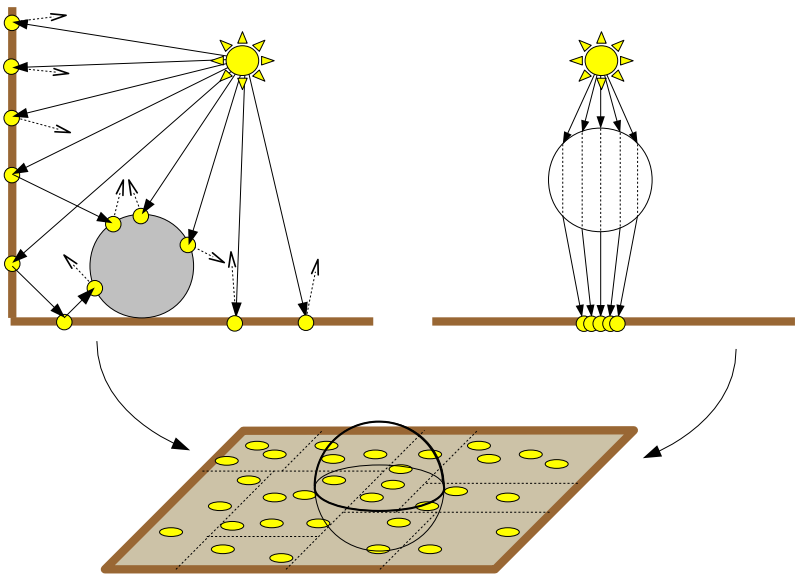


Figure 2.4: Photon Map (left) and Caustic Map (right) creation. The sphere (bottom) is used as density estimator over the KD-Tree that contains the maps

Acceleration structures

One of the most costly lighting simulation operation is the ray traversing trough the scene geometry, so a ray-geometry intersection check algorithm must be executed for each object. The organization of the scene geometry in triangle or quad meshes simplifies the intersection test, but it remains a slow rendering stage for large scenes. To improve it, some acceleration structures were proposed. The most basic is a regular grid, where a list with all geometry that is contained or intersected is stored . But this only reduces the problem linearly. To improve the efficiency we have to use hierarchical structures.

The *Octree* [Gla84] [San85] is a structure of square boxes (voxels), where each box is recursively subdivided into eight non overlapping equal boxes,

defining a tree. The octree nodes that are empty or that contain less than a specified amount of geometry, are not subdivided any more. Since the ray-box intersection is easy to compute, the ray traversal is easy too. First, the ray is tested on top level voxels. If it intersects, the next level is considered, and the intersection test is performed for the eight subboxes. This process is repeated until the box intersected is empty, or until it is a tree leaf, where the ray intersection test is calculated against the list of the stored geometry. The octree size is specified by this list size, a smaller list means a larger octree. There are many works and octree improvements for ray traversal [ES94], such as [Sam89] that introduces the neighbor finding, [Grö93] for dynamic scenes, [WSC*95] that constructs the octree in basis of an estimation of ray-geometry intersection tests, or [RUL00], that shows an interesting traversal algorithm that works in tangent space and uses a voxel encoding method that allows the traversing in a fast way.

The *Binary Space Partition* (BSP) tree [FKN80] is another structure that subdivides the space into planes (in 3D space), where each one of these planes is a node in a tree, called BSP tree. Each node subdivides the space in two parts, placing the plane at the middle of the current geometry bounding box, repeating it until the subdivided geometry is small enough. The traversal is done checking in each tree level at what side of the plane the ray hits. The most difficult part of this algorithm is the BSP tree generation, so for each node we have to choose the subdivision plane, and this affects directly the ray traversal performance. Many works and improvements can be found in the literature [TN87] [Nay93]

The *KD-Tree* [Ben75][Ben80] could be considered as a particular case of BSP. In this case, the planes are axis aligned, so there are only three possible planes, and can be placed anywhere in the geometry bounding box. The most usefull advantage of KD-Tree is the use of the *k-nearest neighbor* algorithm. From a selected point, the algorithm searches the closest k nearest neighbors climbing into the hierarchy, and checking for each adjacent node if it intersects with a virtual sphere with the same radius as the maximum neighbor distance. This is the basis of the *Photon Mapping* method (see Section 2.2.2) to calculate the average energy around a hit point on the *Photon Map*.

The *Bounding Volume Hierarchy* (BVH) [RW80][KK86][GS87] is a tree structure where each node is a minimum axis aligned bounding box (AABB) that encloses the geometry (see Figure 2.5). Each node is subdivided into two children nodes, that represent the bounding boxes of the subdivided geometry. The traversal is done like in an octree, checking the ray-box intersection until the leaf is reached, or discarding the tree branch if it does not intersect.

As stated in [Hav00], KD-Trees are the best option for static scenes.

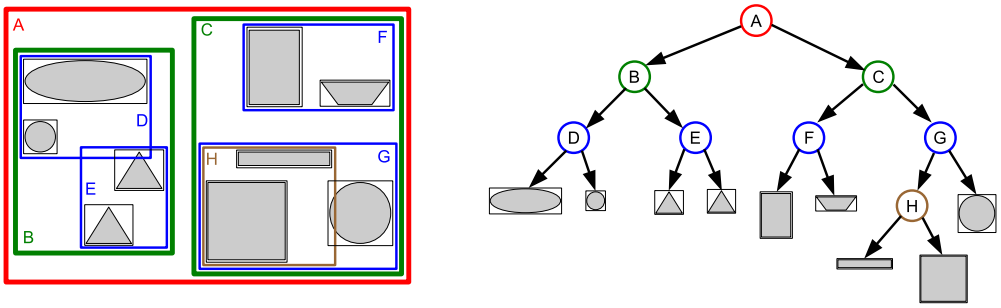


Figure 2.5: Bounding Volume Hierarchy example.

Furthermore, efficient BVH software implementations remained inferior to performance of KD-Trees [WBS06]. In general, both methods are not suitable for dynamic scenes. The best results are obtained when a *Surface Area Heuristic* (SAH) [Hav00][Wal07] is used, but this implies an exhaustive analysis of the scene geometry. There is an hybrid approach between the BVH and the KD-Tree, the *Bounding Interval Hierarchy* (BIH) [WK06], that takes advantage on the best of both methods, and that is useful for dynamic scenes.

2.2.3 Radiosity

Radiosity [GTGB84][CG85][SP94] is a global illumination algorithm that tries to solve the rendering equation in a context of finite elements and diffuse surfaces. Firstly the scene is subdivided into patches. The rendering equation is solved in an equation system where all patches are considered. The radiance transmitted between them is controlled by the *form factor*, that defines their visibility. The overall process is an iterative method. When the form factors have been calculated, the rendering equation computes the radiance for the scene patches considering only the light source emitting patches (usually area light sources) Then, the system is solved again with the radiance obtained in the last step. The process stops when there are no more radiance changes in the patches, or when the results are good enough. The main disadvantage is that the rendering requires high computational times. The form factor calculations consume a lot of time because all visibility conditions between all patches have to be calculated [WEH89][BRW89]. Moreover, in each iteration step, we have to solve the rendering equation for all scene patches. On the other hand, the result is view independent, so the rendering has to be calculated only one time if geometry, material surfaces or light sources do not change. There are some improvements to add non-diffuse effects, like specular lighting or merging ray tracing with radiosity techniques (two pass

solutions) [WCG87][SP89]. In [CCWG88][Che90][DS97] there were presented progressive algorithms to accelerate the method, avoiding unnecessary patch updates. In [BMSW91][HSA91] there were presented some improvements on mesh generation for automatic geometry subdivision in function of lighting variation.

The radiosity methods fall out of our scope, because they are useful for diffuse geometries, but not for specular surfaces like our reflector ones.

2.2.4 GPU acceleration rendering methods

Image synthesis is based on the computation of radiance on those points that are visible through the pixels of a virtual camera. This means that each point can be shaded independently. Hardware graphical cards (GPU) are capable to process many execution threads in parallel. On the other hand, ray tracing based algorithms can be considered for parallel processing. Therefore, we can define new ray tracing algorithms based on GPU usage, exponentally accelerating rendering times. The survey presented in [UPSK07] summarizes the most important methods.

GPU ray tracing algorithms

There are several GPU ray tracing algorithms to consider. In [CHH02] it is presented the *Ray Engine*. This method computes all ray primitive intersections on the GPU. The ray data is stored into a texture, and each scene triangle is rendered using the *full screen quad technique*: This method renders a textured quad on the full viewport, where each texture texel encodes the data to be processed in each fragment (see Figure 2.6).

Then, the pixel shader calculates for each ray the intersection with the current rendered triangle. This is not a full ray tracing engine, and usually it is used in combination with a CPU method, with the consequent increase of the overload between CPU and GPU communication. A full ray tracing engine is defined in [PBMH02]. This method defines the shaders and structures to calculate the ray tracing in different steps, such as the primary ray generation, traversal, intersection, shading and secondary ray generation. The main drawback is that only one pixel shader can be loaded at the same time on the GPU, so for each step, many shaders have to be stored and loaded.

The main GPU problem for ray tracing is the creation and traversal of the acceleration structures, as happens in classic CPU ray tracing algorithms, they need a stack for each ray. Stacks are poorly supported by GPUs, and considering that we process one ray per thread, the parallel processing allows just few memory resources for each thread.

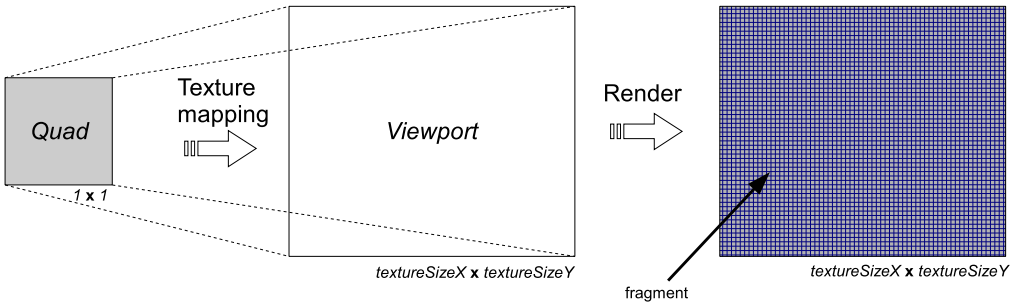


Figure 2.6: Full screen quad technique. A quad is rendered on the full viewport. The viewport has the same size than the desired output texture. Then, each pixel becomes a fragment that is processed by a GPU fragment shader.

As is stated in [Hav00], the best acceleration structure for static scenes is the KD-Tree. To solve the stack problem, some stackless proposals were presented in [FS05], [HSHH07] and [PGSS07] for KD-Tree acceleration structures in the GPU. In [FS05] there were presented two stackless KD-Tree traversal algorithms, called KD-restart and KD-backtrack. Both algorithms are based in restarting the traversal of a ray when it does not intersect with a leaf. Then, the visited branch is discarded using an axis aligned bounding range that decreases as more branches are discarded. The main drawback of this method is the need of a restart, or backtrack of the ray until it reaches the intersection with any leaf. Also, the method performance is uncompetitive with optimized CPU algorithms. The method was improved in [HSHH07], taking the new GPU possibilities in branching and dynamic looping. In [PGSS07] it is used a KD-Tree traversal based on links between nodes [HBZ98]. The rays are packed to be processed together as a GPU block of threads, and to maintain ray coherence. For each node, the whole ray pack is tested for the intersection for both children nodes, traversing the tree by the node with more intersection rays. If a node is a leaf, then the final geometry intersection is checked for the leaf node. The non-intersecting rays on the leaf nodes traverse the KD-Tree by the previously defined links between leafs, avoiding to backtrack, and in consequence, without the need of a stack.

In [GPSS07] there was proposed an algorithm to use the *Bounding Volume Hierarchy (BVH)* acceleration structure in the GPU. The BVH is constructed as a binary tree aligned to the three axis, following the classic SAH node density estimation function [MB90], that is calculated using a method based on

the classification of the geometry in bins. The rays are processed like in [PGSS07]. For each node, all the rays are tested for suitable intersections on both children nodes. Then the child node with more suitable intersections is chosen, storing the index of the other children nodes into a stack, that is stored into the chunk shared memory. The children nodes without any suitable ray intersection are discarded. If none of both children nodes has suitable intersections with the current chunk of rays, the process continues using the first node in the stack. If the current node is a leaf, final intersections are performed. This method improves the previous one described in [TS05] in the way that the current method is based on a view dependent traversal algorithm, hence it uses less rays. The main drawback of this method is the required BVH construction time. This is improved in later proposals, such as in [Wal07] and [LGS*09]. Furthermore, this algorithm achieves a comparable performance to the KD-Tree GPU implementation presented in [PGSS07]. As also is seen in [Wal07], the BVH is suitable both for large scenes and dynamic ones.

In [RAH07] it was presented a GPU algorithm for the classic Whitted ray tracing algorithm, focusing on the tracing of secondary rays. They create a hierarchical structure with spatial coherence of rays over the scene, where the triangles are intersected. Once all triangles are checked for the current rays, new rays are generated from the current bounces, and a new structure is created. The number of maximum ray bounces on the scene has to be limited to get fast renderings. This method has a similar performance than the previous ones, and it is also suitable for dynamic scenes.

In [SSK07] it was proposed a KD-Tree based method for dynamic scenes. They focused on the fast creation with maximum quality of the KD-Tree to get interactive rendering times. To do it, the SAH function is redefined, creating a new simplest binning method to classify the geometry and to construct each tree node. This permits to assign one thread for each tree node construction with the related geometry, doing the whole tree construction in parallel. The results show that, for high resolution scenes, the KD-Tree has less quality than for low resolution scenes, and that to get high KD-Tree quality, the rendering time increases.

GPU relief texturing methods

Since we are interested on rendering scenes where there is only one reflector, we do not need a ray tracing engine to render a full scene with complex geometry. There are specific acceleration methods based on relief texturing that are more interesting for our case. As the reflector to be built must be able to be manufactured through a press-forming process, where the reflector

shape is deformed only in one direction, this allows us to process the geometry in a discretized space like a height field. In [SKU08] it is presented a survey of techniques to calculate displacement mapping on the GPU. The relief geometry is stored into textures as height fields, that are traversed in tangent space by scene rays to obtain the texture coordinates of the intersection point. These coordinates are used to fetch color and normal vector data previously stored in textures. There are many algorithms that offer different solutions in basis of the method to get the intersection point. In [POJ05] it is proposed a *binary search method*. An interval over the ray, with minimum and maximum heights as bounds, is halved in an iterative process. Then the middle interval height and the real height field value are compared at the middle point, checking what section of the interval subdivision contains the intersection point (see Figure 2.7). The method is fast, but the final intersection will not be necessarily the first intersection. Also, the method can be affected by false intersections and artifacts for high frequency height fields [BD06]. This is solved by performing some initial fixed-step iterations, but this could miss important information.

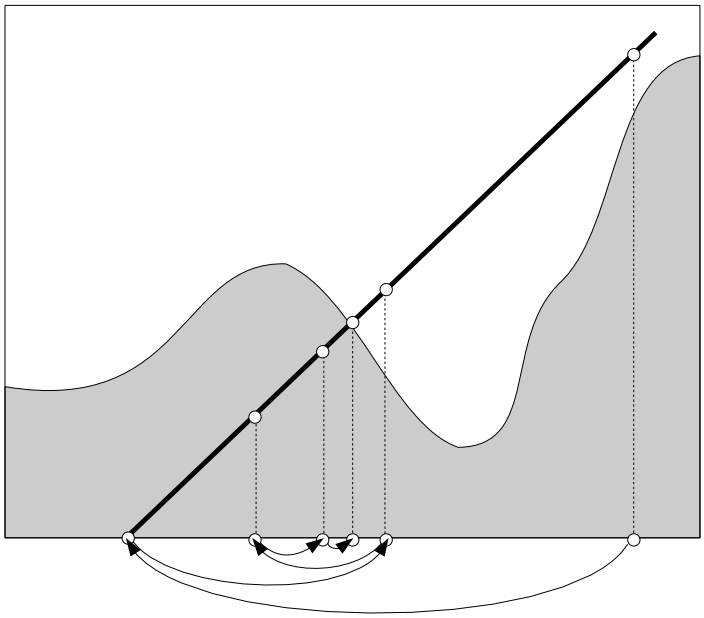


Figure 2.7: Binary Search algorithm

An improvement over binary search is presented in the *precomputed robust binary search* method [BD06]. The algorithm moves a point along the ray with a precomputed *safety radius*, that is a distance where can be at most

one intersection. If the next point is closer to the intersection point, a classic binary search is used to reach it with more precission.

In [YJ04] a *secant method* it is proposed. It assumes that the surface, between the current ray point height and the next one, is planar calculating the intersection point between both height bounds. But this method and the previous one do not guarantee that the resulting intersection is the first one. The *sphere tracing* method [Don05] improves the fixed step size calculating a 3D distance map, where each 3D texture texel stores the radius of a virtual sphere that guarantees no intersect with the height field (see Figure 2.8). Then, the step size will be the sphere radius. The problem of this method is the 3D distance map calculation and storage cost as it has to be precomputed.

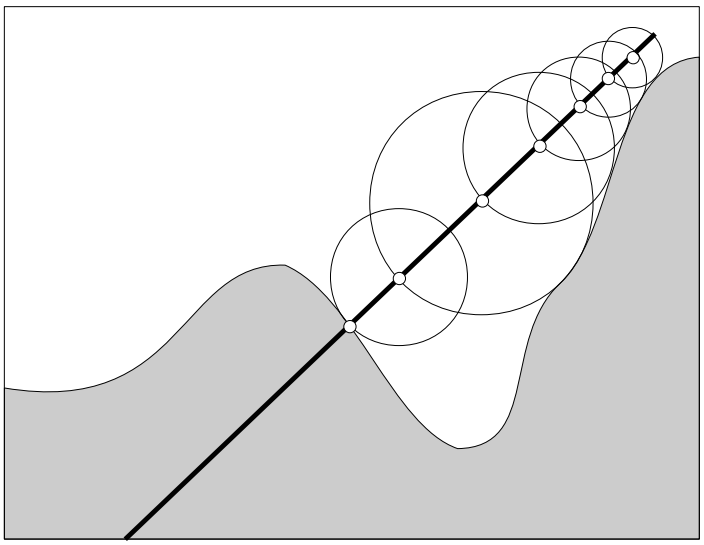


Figure 2.8: Sphere Tracing algorithm

One of the most well-known and used traversal methods for height fields is the *Relief Mapping* algorithm [POJ05]. The method has two stages. First it is performed a linear search algorithm, based on *ray-marching* [Lev90], where the point advances over the ray with a fixed step size. Second, when the current ray height is under the height field value, a binary search starts to reach the intersection point, avoiding the stair-stepping artifacts of linear search methods [SKU08] (see Figure 2.9). The main drawback of the relief mapping method is the decision on the linear search step size. Larger steps mean faster results, but possibly missing intersection points. Also, the method does not guarantees to get the first intersection point, like the binary search based methods.

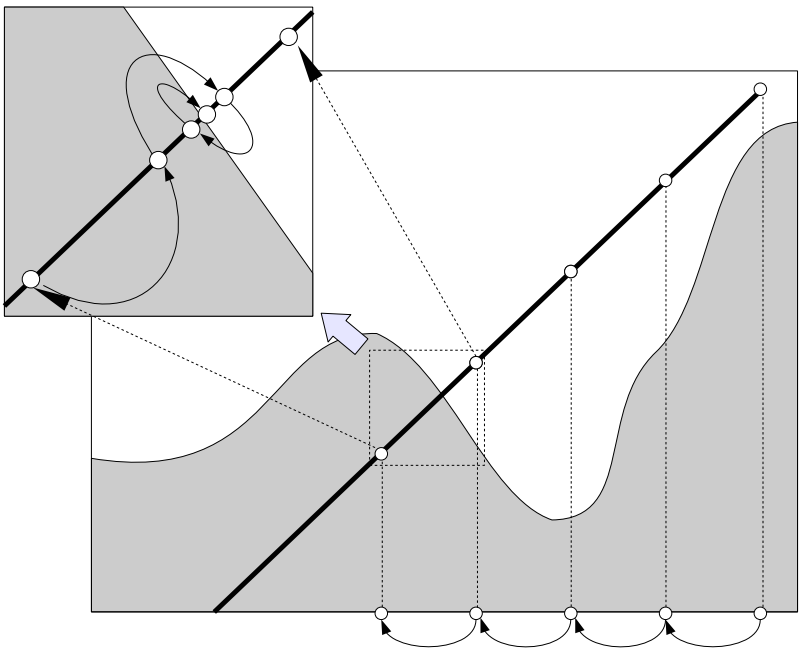


Figure 2.9: Relief Mapping algorithm.

To improve the fixed step size problem, hierarchical height fields are considered, implementing them as mip-map textures (or height maps). At each level there is a texture with a different height field resolution. This resolution at each level defines the step size over the ray. The *Quadtree Relief mapping* method [SG06] uses this idea. The ray is projected onto the XY plane of the height map texture (considering the height values aligned with the Z axis) at the lower level. This gives a starting ray height. Then, and starting at the quadtree top level, the ray advances according to the texture resolution at the current quadtree level. Two intersection distances are calculated. First, the intersection distance to the current quadtree height. Second, the intersection distance to the current quadtree texel boundary. If the height intersection distance is negative, the ray does not advance, and the quadtree level is decreased. Otherwise, the minimum distance is chosen as advancement step size. If the lowest quadtree level is reached, and the current ray height is under the current quadtree texel height, the intersection is found (see Figure 2.10).

A very similar method is found in the *Pyramidal Displacement mapping* method [OKL06], differing only on the method to move the ray along the height map.

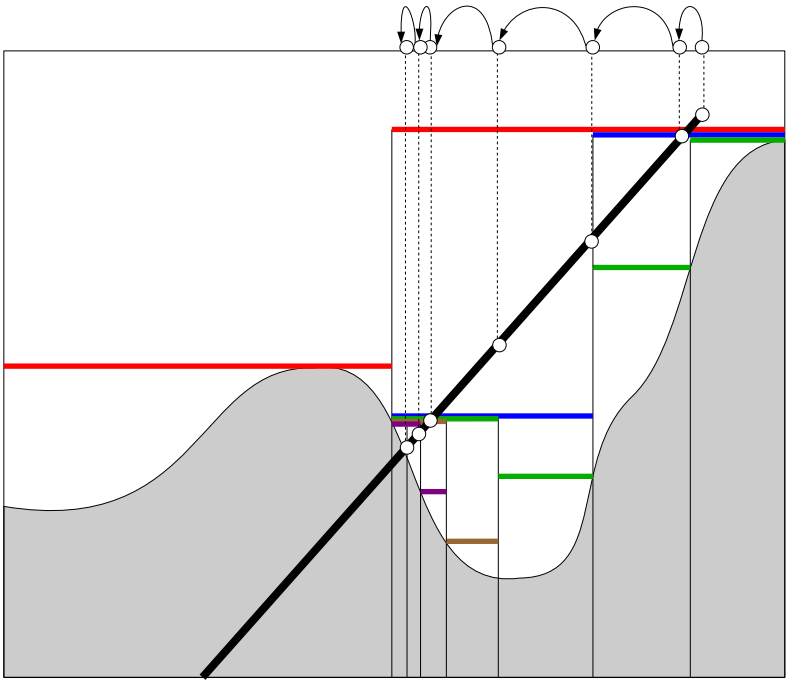


Figure 2.10: Quadtree Relief Mapping algorithm.

2.3 Optimization algorithms

We are interested in finding the minimum value for an unknown function that returns the difference between the calculated reflector light distribution and the desired one. To get this minimum we use an optimization algorithm. Optimization algorithms are usually based on an iterative processes that approach step by step to the solution traversing the function domain. There are two main types of optimization algorithms for minimizing a function: local algorithms, that try to get a solution, and global algorithms, that try to get the best solution for the whole function domain.

2.3.1 Local optimization methods

Local optimization algorithms try to get the minimum locally on the unknown function domain. This kind of algorithms is suitable when the solution is near to the starting point. There is a large amount of local optimization methods that minimize a function. Here we focus on the most relevant local methods capable to manage multidimensional functions. For other methods, we recommend to the reader to references the classic litera-

ture [Fle87][NW99][Pre07].

One possible classification of local algorithms is between *direct search* and *pattern search* methods. Direct search methods calculate or approximate the derivatives of the function. One of the most commonly used method is the *conjugate gradients* algorithm [Pre07]. This method calculates the function derivatives to know in which direction (in parameter space) the function decreases faster. To calculate the derivatives, the function must be known. Because the function is unknown, many evaluations of the function at near points have to be calculated to interpolate the derivatives.

Pattern search methods compute patterns of function points trying to get the best optimization direction, and avoiding the derivative dependence. The most classic algorithm is *Hill Climbing* [Wei07]. This simple method starts from a point on the domain and searches between its neighbors for a point with the best value. The optimization ends when there is not any best near point. The method is very sensitive to local minima. A more efficient algorithm is the *Downhill Simplex* method, also known as *Nelder-Mead* method [Pre07]. This is an optimization algorithm that for each step, creates and manipulates systematically a pattern of points trying to get the best direction to drive the optimization. This pattern defines the shape of a simplex with as many vertices as function parameters. Then, the worst one is chosen and it is reflected over the simplex centroid if the new point is better than the current better one. Otherwise, the simplex is contracted. The process continues until the simplex vertices are close enough, so the solution is found. This method is suitable for smooth functions, and the main drawback is the number of function evaluations to calculate.

Another pattern search method is the *Hooke & Jeeves* algorithm [HJ61]. This method is based on a downhill simplex search method, but only considers two points along each parameter axis to construct the pattern. These points are shifted along their axis with an initial offset from the current parameter point. The combination that produces the best function evaluation is chosen. Then, it is performed a jump in the parameter space from the last best point and through the new one, obtaining a new point, that is checked again shifting the parameters along their axis. When no new point produces better results than the current one, the jump step is reduced, and the process starts again at last point. Therefore, the pattern becomes more accurate as the method approaches the solution (see Figure 2.11).

In [GL93] it is presented the *Tabu Search* algorithm, that tries to increase the optimization speed avoiding already visited points on the domain. For each optimization step, a new near point is selected. The new point is searched into a *taboo* list that stores the already visited points. If it is not visited, the point is evaluated and stored in the taboo list, and the current

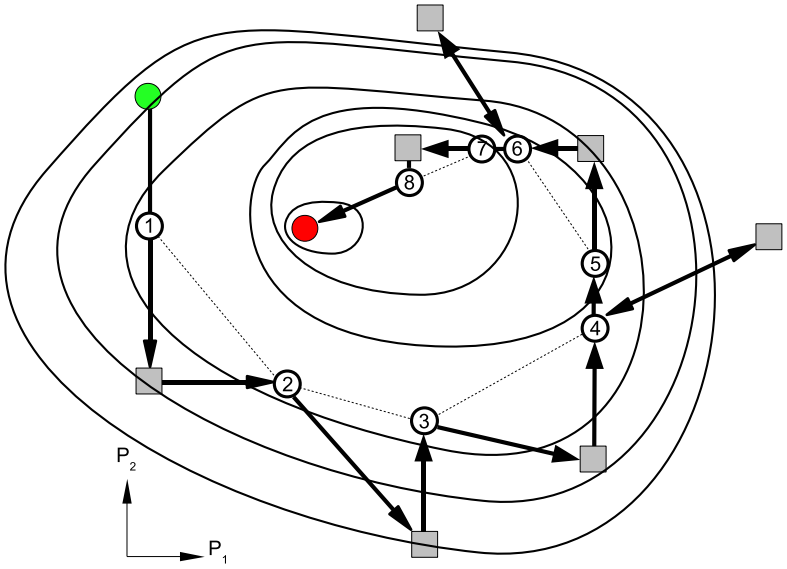


Figure 2.11: Example of Hooke & Jeeves optimization method for a function of two parameters (P_1, P_2). The node numbers show the optimization progression. The gray squares are the point shiftings along the axis. Note that in nodes 4 and 6 no new point produces better results, thus the shift jump size is reduced and the process starts again at the same point.

minimum is updated.

The main drawback of local optimization methods is that it is not guaranteed to reach the best minimum of a function, so a point X_0 is a local minimum if

$$\exists \varepsilon \;\; f(X_0) \leq f(X') \qquad \forall X' \in D' \subseteq D \subset \mathbb{R}^N$$

with

$$D' = \{\forall Y \;\; | \;\; \|X_0 - Y\| \leq \varepsilon\}$$

where $X_0 \in D$, $Y \in D$, $\varepsilon \in D$, and ε is small enough. By the other hand, they are usually fast and appropriate when we are near the desired minimum.

2.3.2 Global optimization methods

Global optimization algorithms, in contrast of local optimization methods, are capable of getting the global minimum of a function, such that

$$\exists X_0 \;\; f(X_0) \leq f(X) \qquad \forall X \in D \qquad f : D \to \mathbb{R}$$

converging to the optimal solution of a non-convex problem. Generally, global optimization methods can be classified in deterministic and stochastic algorithms.

Deterministic algorithms are used when there exists a clear relation between the features of a possible solutions and the problem to solve, or when the parameter space is not too large. In that case, techniques like divide and conquer space subdivisions are used. One example is the *Branch & Bound* method [LD60]. This method is based on the creation of a tree, where each node represents a bound on a function parameter domain. The algorithm has two parts for each optimization step: branch and bound. In the branching part, the selected node range is splitted into new ones for creating the children nodes. In the bounding part, a function estimator returns the children nodes upper and lower bounds for the respective domains. Then, the node with the minimum upper bound is selected, and all branches with lower bound smaller than the selected upper bound are discarded. This pruning process allows to discard those branches that never will drive the algorithm to the solution (see Figure 2.12). The main drawback of this method is the calculation of the children bounds if the function is unknown.

Deterministic algorithms are not suitable for problems like ours because we can handle large parameter spaces and the solutions are not directly related with the problem to solve, because our parameter space defines the reflector configuration domain and the solutions are a difference between a reflector lighting distribution and a desired one (see Section 5.1). In this case, stochastic optimization algorithms are a better option. Stochastic methods check the function domain in a probabilistic way using Monte Carlo algorithms. This guarantees the fast optimization termination with a result that can be correct or wrong with a given probability. If the result is wrong it means that the desired result is not exact, but it is not too far from the desired result. Monte Carlo methods guarantee exact results in an indeterministic termination time. *Simulated Annealing* [KGV83] is one of the most used stochastic optimization algorithms. It is a variant of the classic Metropolis algorithm, and it is based on a physics analogy where the metal tends to cool and anneal. Following this analogy, the method starts at an initial point and tries to reach a state with minimum temperature, that is our function minimum. For each optimization step, the algorithm evaluates the function at a stochastically chosen point near the current one. Then, an heuristic decides, stochastically too, if the process has to jump to the new point, or remains at the current one. This heuristic is defined to drive the process to a function minimum, giving more probability to paths where the function evaluation differences are greater following the metal cooling analogy. One of the main advantages of this method is that it does not fall easily

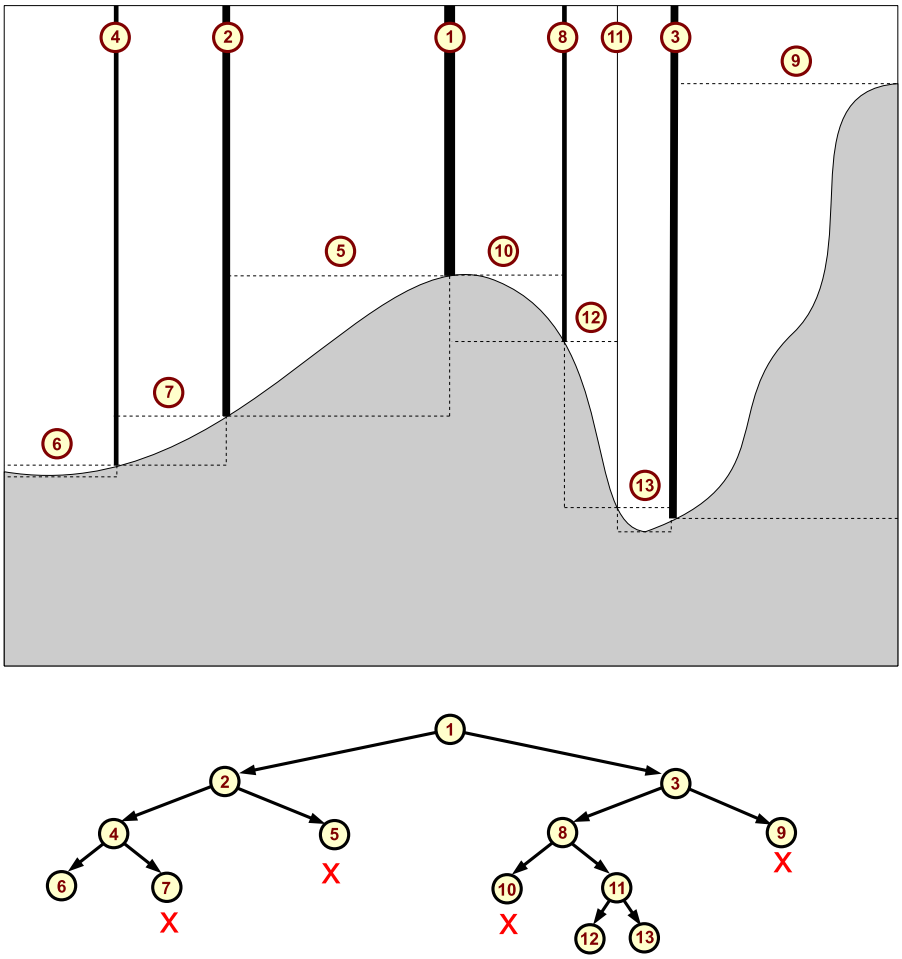


Figure 2.12: Example of Branch & Bound optimization method. The numbers show the optimization process. The first local minimum is found at 6, and it allows to prune the branches 7, 5, 10 and 9 since their bounding minima are greater or equal than the bounding maximum of the branch 6. The next local minimum is found at branch 12, and finally, the global minimum is found at branch 13.

to local minima, avoiding them when the cooling temperature is near to 0, that is the optimization process is near to the solution. Simulated annealing algorithms with appropriate cooling heuristics can asymptotically converge to a global solution. But this does not guarantee that the solution will be reached in a fast way. Also the method does not guarantees completely avoiding local minima. If a local minimum is found, the process must restart at

another starting point. It could be faster a brute force algorithm, checking all possible states, than finding the global minimum with absolute certainty with a simulated annealing algorithm.

2.3.3 Other optimization methods

There are other optimization techniques that solve specific cases. The algorithms based on *perturbation theory* [Hol95] are used when the function is known, but it is difficult to solve. These kind of algorithms apply small offsets to each function parameter, transforming the problem into a polynomial form. The higher the polynomial order, the higher the precission.

linear programming optimization methods [Dan63] try to reach the function minimum or maximum using restrictions on this function. The restrictions are represented as linear equations composed by a set of independent variables. A combination of linear equalities and inequalities define a polytope from the intersection of all equations. The algorithm searches for a point inside this polytope that represents the minimum or maximum of the objective function. In other words, the goal is to solve the linear equation system composed by all restrictions. One of the most popular methods on linear programming is the Simplex method. However, the functions used in our problem does not have enough restrictions to consider feasible this kind of optimization method. There are also nonlinear programming optimization methods. This kind of algorithms applies restrictions on different parts, replacing independent variables by new restrictions. Although these solutions can handle global optimization problems, they do not guarantee the convergence in a finite time if the problem is non-convex, so many local minima can be found.

Other methods are the *clustering* algorithms, such as the *Global k-means clustering* algorithm [LVV03]. The optimization algorithms based on clustering start by evaluating a sparse set of points over the function domain. Then, the points are clustered on points that generate values near to the local minima. When the data is clustered enough, the cluster with minimum representative value is chosen as solution. The clustering methods try to solve the problem of starting points on local search methods. To avoid local minima, usually a set of possible starting points are evaluated over the full function domain (see Figure 2.13). But this does not guarantees that some of these points drive to the same local minimum. This is usually solved by sparsely sampling the function domain, thus their main drawback is the evaluation of a huge number of points.

Finally, there are combinations of different methods, and variations of previously commented algorithms. The literature on such problem is vast

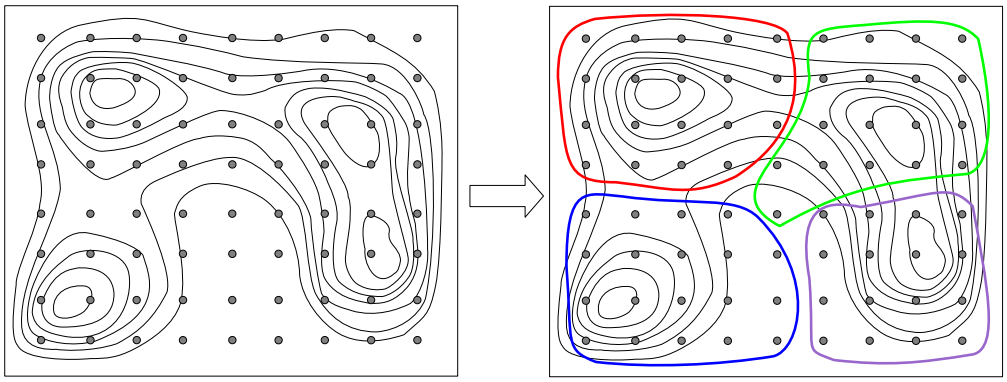


Figure 2.13: Clustering optimization. The clusters identify the local minima and the best one is chosen.

and outside our scope. To know more about optimization algorithms, we recommend the reading of classic optimization surveys, such as [Pre07] and [Wei07]

2.3.4 Interval arithmetic

Interval analysis [Neu03] is the study of theory and computation with interval data.

$$[\underline{a}, \overline{a}] = \{x \in \mathbb{R} \mid \underline{a} \leq x \leq \overline{a}\}$$

This is important for optimization methods if function parameter ranges are considered as intervals, for example by improving the bounds calculation. Another avantage of interval analysis is that classic analysis is extended in its ability to provide semilocal existence and optimality conditions for a pre-specified local region around some point, while classic analysis generally only asserts the existence of such neighborhoods without providing a simple way to find them. The basic arithmetic operations are provided for intervals, such as additions or substractions.

$$[\underline{a}, \overline{a}] + [\underline{b}, \overline{b}] = [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \qquad [\underline{a}, \overline{a}] - [\underline{b}, \overline{b}] = [\underline{a} - \overline{b}, \overline{a} - \underline{b}]$$

More complex operations can be defined from the basic ones ever considering that the functions are piecewise monotonic. For a general function with more than one parameter, an overestimated but tighten interval range can be defined as

$$g : \mathbb{R}^N \to \mathbb{R}$$

$$Range\{g, \mathbf{x}\} = \{g(x) | x \in \mathbf{x}\}$$

$$g(z) - g'(\mathbf{x})(\mathbf{x} - z) \supseteq Range\{g, \mathbf{x}\} + O(\varepsilon^2)$$
$$O(\varepsilon) = \overline{x} - \underline{x} \qquad x, z \in \mathbf{x} \in \mathbb{R}^N$$

One of the most interesting applications of interval arithmetic is the convexity check, that tests if a function interval is convex. The main condition to do this check is to create a matrix of intervals calculated as an enclosure of $g''(\mathbf{x})$. A more detailed description of the checking algorithm is found in [Neu03].

The main drawback of interval analysis, for our case, is the need of function knowledge to calculate the derivatives, and the need to guarantees that the interval is the enclosure of a monotonic function.

2.4 Inverse design

Considering a general model composed by a reflector and a light source, the inverse design problems can be divided in function of the nature of the unknown data [JL06] [PP03]:

- Inverse Lighting Problems (ILP), where the reflector shape geometry and material properties are known, but the emittance and light sources placement are unknown.
- Inverse Reflectometry Problems (IRP), where the reflector material properties are unknown, and some features about the emittance and reflector shape are known. These kind of problems are usually called Inverse BRDF Design or Inverse Texture Design [Sch94].
- Inverse Geometry Problems (IGP), where the reflector shape geometry is unknown.

2.4.1 ILP problems

There are different examples of solutions of ILP problems. In [SDS*93] an optimization algorithm adjusts the light source intensities, where the placements and light distributions are known, until reaching the user specified lighting. The work presented in [PRJ97] tries to get the light source placements from the user specified shadows in the scene. Another local illumination algorithm is found in [Gui00] for multiple light sources, where a grid of simple light sources is defined and clustered in each optimization step, trying to reach the approximate light source positions and emittance. In [Mar98] it is presented an algorithm that, starting from an already lighted scene, constructs a system of equations for all light source unknown data,

controlled by variable weights. An optimization algorithm searches for the best weights to get the closest solution to desired one. In a similar way, the work presented in [LFD*99] starts from the real lighted scene images, and extracts the lighted geometry using an optimization algorithm. Whereas previous mentioned solutions consider local illumination, this work considers the scene radiosity, although some simplifications are assumed, such as diffuse BRDFs and a hierarchical radiosity approach [SP94]. The result is a matrix system to solve. Other solution for global illumination is found in [OH95], where an inverse heat transfer method applied to light transfer is defined. This method tries to get the lighting of a zone from other scene areas where a desired lighting has been specified. An iterative process adjust the lighting of the whole scene until the solution is found. Another work in inverse lighting problems is found in [RH01], where a solution for ILP problems is presented. It simplifies the radiosity equation into an equation system using spherical harmonic components as an approximation. However, only specular reflections can be considered to avoid large equation systems. There are global illumination methods based on ray tracing, such as [CSF99], where an inverse ray tracing algorithm is used based on the user specification of the intensities over the scene surfaces, obtaining the approximate scene light source placements. Then a simple optimization algorithm is used to better fit the final light source positions. The work presented in [SL01] is similar, but it uses human perception factors. Finally, other particular solutions have been developed, such as in [PRJ97], where the light placements are approximated by the user specified shadows on the scene, or the work presented in [Gui00], where a starting regular grid of lights is clustered in an optimization process until the clusters produce lighting similar to the desired one.

As is stated in [PP03], ILP methods have the drawback of computing large equation systems, and there are no global illumination solutions for light source placements.

2.4.2 IRP problems

The goal of IRP problems is to define the material properties of the scene geometry. In [BG01] the geometry and lighting are known, and in basis of a real image, different BRDFs are tested until the result is close enough to the image. In [LKG*01] a set of images are taken around an interesting point. Then, diffuse BRDFs are used to approximate the results to the images taken. A similar work is presented in [YDMH99], where the BRDF of human skin is approximated. As in ILP problems, [RH01] propose a large equation system to solve in an optimization method. In the global illumination solutions field, the work presented in [LD00] tries to identify the BRDF from the known

scene radiosity, and compare them to real images. Also, it is demonstrated that the camera calibration is very important.

The main drawbacks of these solutions are the dependence on camera acquisition parameters, and the limited BRDF parameters that can be found.

There are also solutions to ILP and IRP combined problems, where the geometry is known, but not the emittance and the reflectance. Among global illumination solutions, the work presented in [KPC93], starts from an initial radiosity distribution and light sources positions, searching the light source emittance and the diffuse reflectance. To do it, an optimization process finds the approximation error in three stages: the light source intensities, the directional distribution, and the scene BRDF.

2.4.3 IGP problems

IGP problems can be grouped in two sets: analytical and numerical solutions. For analytical solutions, the work in [Oli89] presented a solution for rotationally symmetric reflectors, demonstrating the uniqueness of the solution. For more general cases, in [KO97] they demonstrated a general weak solution, showing existence, uniqueness and smooth of the function. The numerical solutions can be classified by the kind of lighting method used to simulate the light propagation, that is either local or global illumination methods. Local illumination methods [KO98][CKO99] do not take into account the reflector inter-reflections, considerably affecting the accuracy of the results. In [KO98] it was presented a method that constructs the reflector boundary from the intersection of confocal ellipsoids, where the other foci of the ellipsoid lies in a point that already has the desired intensity. A very similar approach was described in [CKO99] and [KO03], but this time using paraboloids, and where the paraboloid direction lies in a direction that already has the desired intensity. For both methods (see Figure 2.14), the main difference is that the former is suitable for near-field desired lighting, and the second for far-field desired lighting. In [EN91] and [Neu97] methods were proposed for the reflector representation as a cubic tensor product splines. Then, a local optimization algorithm was used to search the spline coefficients under some function domain constraints.

The numerical solutions that use global illumination methods consider the reflector inter-reflections. In [DCC01] it was presented a global illumination IGP method for specular reflector surfaces constructed from a 2D Bezier curve. The four control points of a Bezier curve are specified by a genetic optimization algorithm. The illumination algorithm is a recursive ray tracing method, that uses the number of bounces as the optimization stopping criteria. The main drawback is the problem on convergence of the optimization

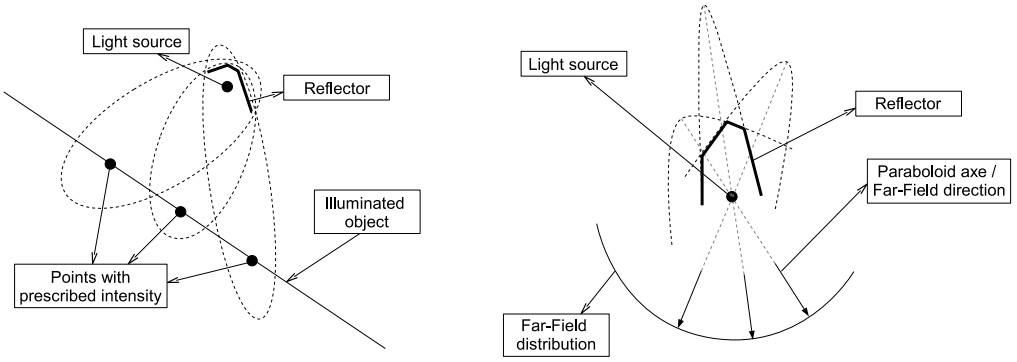


Figure 2.14: Reflector construction from the intersection of confocal ellipsoids (left) [KO98] and reflector construction from the intersection of paraboloids (right) [CKO99][KO03].

algorithm.

In [PPV04] and [PPV07] methods were proposed starting from an initial reflector mesh representing a height field. Then new reflectors were obtained changing the height of each mesh vertex. A brute force optimization algorithm was used trying to minimize a distance function that measures the difference between each reflector light distribution and the user specified one (see Fig.2.15). The reflector lighting is calculated by a global illumination method (see Fig. 2.16), with an anisotropic point light source (see Section 2.1). The global illumination algorithm samples by Monte Carlo, and it is able to compute diffuse reflector surfaces, in contrast with previous methods. The main drawback of this method is the high number of reflectors needed to be calculated and the high computational cost to compute each reflector lighting distribution. The computational order is $O(h^n)$, where n is the number of control points, and h is the number of discrete positions where the control points are moved. To reach a good enough solution, the algorithm spends a considerable amount of time (hours or even days).

Another consideration to take into account for IGP problems is the kind of desired reflector light distribution. For far-field representations, we can enumerate the works of [EN91], [Neu97], [CKO99] and [PPV04]. For near-field representations, there are the works of [EN91], [Neu97], [KO98] and [DCC01]. On the other hand, common reflector industry requirements are far-field representations.

More recently, a new approach presented in [ASG08] proposed the generation of reflectors based on parametric equations. An optimization process searches for the parameters that generate the desired reflector lighting using

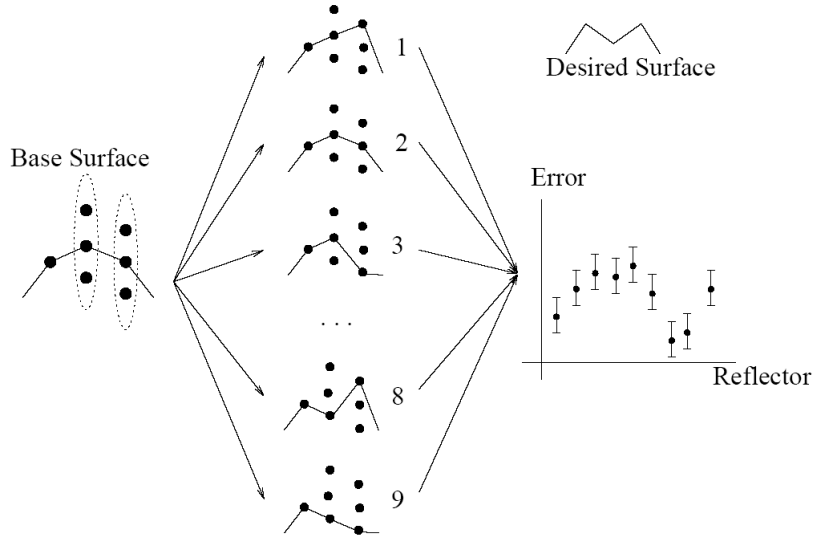


Figure 2.15: The overall optimization system presented in [PPV04]

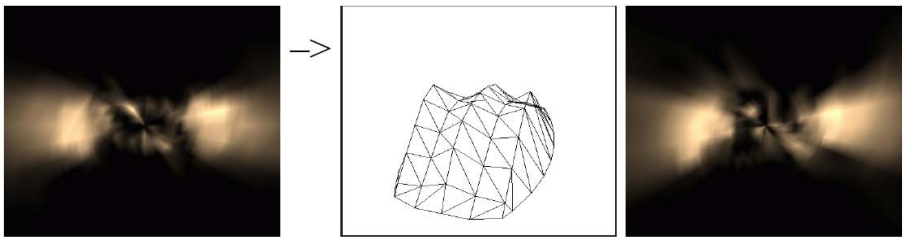


Figure 2.16: From the desired lighting (left), the reflector mesh is transformed until the lighting distribution is close to desired one (right)

photon mapping as global lighting algorithm. Although the method is fast, its main drawback is the optimization method, the Hooke & Jeeves algorithm, that is a local algorithm, and, as seen on section 2.3.1, this does not guarantees the convergence of the method.

Finally, the recent work presented in [FDL10] shows a method to reconstruct geometry with specular reflectance from caustic images. The method uses a B-spline surface to define the geometry, where the control points are moved in vertical direction, like in [PPV04] and [PPV07]. For each geometry shape, caustics are calculated by a GPU Monte Carlo light transport simulation ignoring multiple interactions. They use a global optimization algorithm based on perturbation theory (see 2.3.3). Although the method is very fast, its main drawback is the single reflection limitation, that makes

it not suitable for classic reflector shapes, such as paraboloid ones. Another drawback is the optimization method, since it only guarantees a probabilistic convergence to a global solution.

In [PP05] and [PP03] there are presented more in deep summaries of IGP methods.

2.5 Conclusions

In this chapter, we have described different works and approaches that directly or indirectly focus on the same goal: providing tools for the inverse reflector design. Most of these works concentrate on the reflector and lighting design techniques, real-world light source representations, simulation of global illumination, and optimization methods. Although we have described the best methods for each field, there is still much room for improvement on the efficiency, both in computational costs and reliable results.

From realistic lighting simulation methods, that is, global illumination methods, there are many techniques that depend on the required simulation quality, rendering speed or computer requirements. For our purposes, we need a very fast rendering algorithm to get the lighting simulation of a reflector in a minimum time. Ray tracing methods are more suitable for this objective, in contrast with radiosity methods, because we need to compute purely specular reflections, in accordance with industry standards, and using only a reflectance attenuation factor. From that, we could consider the GPU ray tracing algorithms to get the required performance. However, we do not need ray tracing engines for general scene geometries, since our case is very particular and constrained. We have a mesh that defines the reflector and a complex light source. Moreover, the mesh material is considered fully specular, so we can avoid the BRDF considerations. Therefore, and considering that the reflector can be represented by a height map, we can use relief mapping-based techniques, such as quadtree relief mapping. In addition, we must note that we do not need a ray tracing algorithm that generates an image from the viewpoint. We need a light tracer only, as we are only interested in the reflector outgoing light distribution.

From the light source point of view, it is clear that we need to work with near-field light sources. Current representations, such as light fields, have some drawbacks, as they are too large models. The raysets have the same problems. However, since raysets represent raw data, we could consider them to define a new compressed structure to make them suitable to be used in a light ray tracing algorithm.

Concerning the optimization methods, two main kinds of algorithms have

been reviewed: local and global optimization. The use of local optimization methods could be discarded, since we are interested in finding the best solution in the function domain, avoiding local minima and assuring convergence. By the other hand, local methods are faster than global ones. We have seen two global methods: Branch & Bound and Simulated Annelaing. The former has the problem of the bounding calculation, so we do not require any prior knowledge about the function to optimize. The second uses an efficient heuristic to search for the minimum, but the convergence is not guaranteed, and it would be necessary to restart the process from another initial point. We have seen other optimization methods such as the ones based on perturbation theory, that need some knowledge of the function; linear programming, that needs an enough number of function constraints; or clustering methods, that need a large number of function evaluations. Interval theory shows us the possibility to fit the function minimum by an analytical way, but with the drawback of the lack of function knowledge to compute derivatives we cannot use these methods. It is clear that a specific optimization algorithm is needed, considering the most interesting parts of current optimization algorithms, and creating new ones to adapt them to our problem.

Finally, we have shown that this is a case of IGP problem. There are some interesting works done in this field, but they require high computational costs, in time and memory size, to make them suitable for obtain accurate results in a fast way. We think a new IGP method is needed to get the desired reflector mesh, from a known light source that produces the desired lighting as a far-field description. This method will be described in the next chapters of this thesis.

Chapter 3

Compact representation of near-field light sources

One of the most important factors for accuracy and realism in global illumination is lighting complexity. This is achieved using real light source representations. However, most of the times non measurement-based or analytical light sources are used. This is reasonable in applications where physical accuracy is not important, but it is critical in situations where we want the lighting simulations to be as close as possible to the real illumination. Traditionally, a far-field approximation has been used in industry to model real light sources. A far-field representation models a luminaire as an anisotropic point light source, and assumes that objects to be illuminated are located far away. There are some established standards to represent far-field light sources, the most important being IESNA and EULUMDAT [ANS02, bCL99]. However, far-field representations do not produce accurate results when objects are close to the light source. As an example, a far-field representation of a bulb can not be used to compute the light distribution produced by a reflector, since the distance between the bulb and the reflector surface is usually very small. The alternative is to use near-field representations. A near-field representation models a light source as a complex light source, where the light source geometry is considered in addition with the light source distribution. In this case, the luminaires are modeled as extended light sources, and there is no assumption on the distance of the objects to be illuminated [Ash93, SS96, GGHS03a].

In this chapter we present a novel approach for using, in an efficient way, near-fields of real light sources for global illumination algorithms that will help to solve the overall goal of inverse reflector design.

The rest of the chapter is organized as follows. The near-field acquisition and data set models are presented in Section 3.1. In Section 3.2 it is presented

the overview of proposed method for efficiently managing the near-fields. The following sections 3.3 and 3.4 show the method details and how to use the near-field representation into a global lighting algorithm. The main results are presented in Section 3.5 Finally, some discussion on the results and the method is presented in 3.6.

3.1 Near-field light sources acquisition and representation

In recent years there has been an important effort to improve real light source capture and representation methods using near-field models. This is obtained from the capturing process around the virtual bounding surface of the light source. A gonio-photometer performs this capture. It is usually based on two rotating arms that allow the device to capture the light coming from a source from all possible directions (Figure 3.1), but other setups are also possible. However, the raw data produced by these capture methods can produce huge near-field data sets.

To make these models useful we can compress the model to a Light Field representation [HKSS98, Ash95] or to a rayset representation. Although the former generates a compressed structure where the near-field is represented, the required amount of memory for data sets is high (around 100MB [GGHS03a]). On the other hand, the data can be processed to generate a rayset [AR98]. A rayset consists of a list of pairs of a point and a direction (see Figure 3.2). Thus, each particle has a location and an outgoing direction. Each pair in the list can be considered as an exitant particle that comes from the measured light source, all of them carrying the same energy.

Raysets are a convenient representation for global illumination algorithms such as light ray tracing or photon mapping, since they provide a set of particles that can be used directly for shooting rays from the light source. However, raysets are usually too big (10M particles) to be efficiently sampled. To solve it, a new compressing method is proposed for dense raysets of near-field light source measurements. This compressing method transform the rayset into a reduced set of anisotropic point light sources, with no significant loss of information. Then, a Monte Carlo method based on importance sampling is used to efficiently sample the emittance of the light source from the compressed data.

Figure 3.1: Gonio-photometer RiGO [Rad] used to capture the rayset from the light source

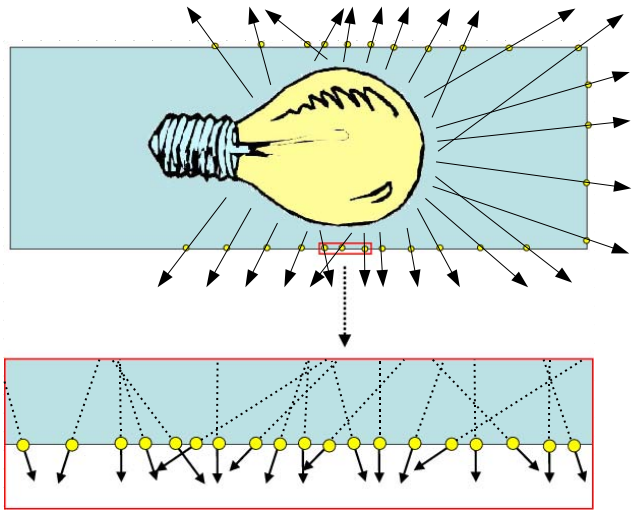


Figure 3.2: A rayset is a set of particles (point + direction) stored on a bounding surface. These particles represent the light emission from a light source.

3.2 Overview

The method presented here deals with rayset models. The goal is to highly compress the data set with a small error even for closely illuminated objects.

The particles of a rayset are located on a virtual bounding surface that wraps the light source. This surface usually corresponds to a convex surface like sphere or a cylinder, but different providers use different supporting shapes. To take into account this variety, the algorithm is able to handle any sort of surface as long as it is convex.

In a first step, a partition of the initial rayset into clusters is computed using the particle locations and their directions. For each cluster a representative point is computed together with an average particle density, obtained from the particles included in this cluster. In order to accurately capture the particle density changes, the clustering produces more density in areas with a rapid variation of particle density. Areas with constant particle density will have less clusters. Once the clustering is finished, it is created an anisotropic point light source for each cluster, where the position is the representative point, and the directional distribution is computed using the directions of the particles of the corresponding cluster. The point light source energy is the sum of all original related particles to this cluster. The directional distribution is stored using a simple constant basis function over a subdivision of the sphere of directions into spherical triangles (see Figure 3.3) in a hierarchical way.

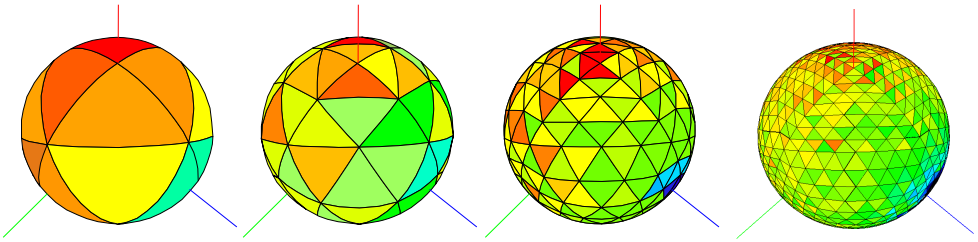


Figure 3.3: A regular subdivision of the sphere using spherical triangles. From left to right, the images correspond to levels 1,2,3 and 4 of the subdivision.

At the end of this process we have the compressed rayset by defining a set of direction-dependent point light sources (see Figure 3.4) located on the virtual bounding surface of the light source. Then, to uncompress and use it for a ray shooting algorithm, the stored structure is used to sample each particle, in both position and direction. First, a triangle mesh is created from

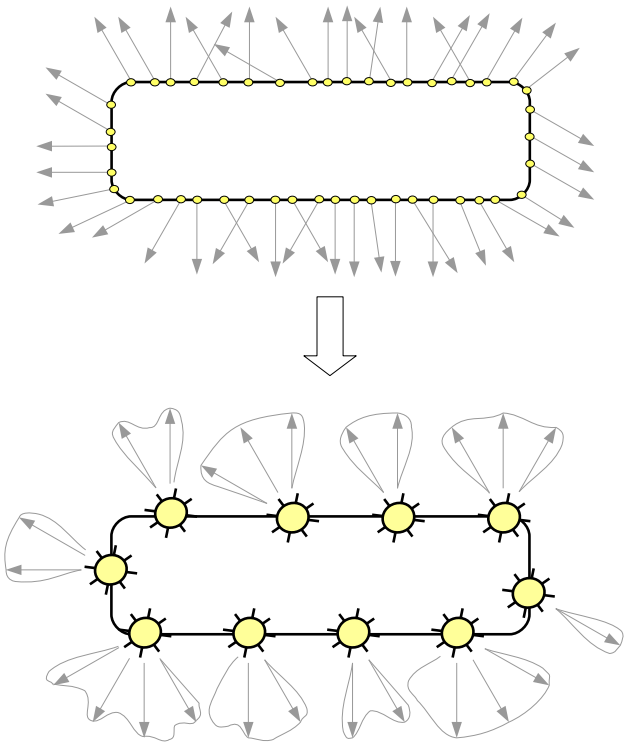


Figure 3.4: The process of transforming a rayset into a set of anisotropic point light sources.

the stored clusters, where the triangle vertices are the point light source positions, obtaining a mesh representation of the bounding surface (see Figure 3.5). Then, the position is chosen by sampling a triangle on the mesh and sampling a position inside of it. Next, the direction is chosen by sampling the directional distributions stored in the nearest triangle vertex, that is the nearest cluster. This way, it is ensured that we sample all the domain of possible outgoing directions.

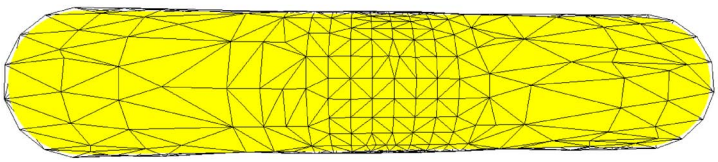


Figure 3.5: Mesh produced from a 10M rayset corresponding to a OSRAM PowerBall bulb.

3.3 Compressing the Near-Field

Rayset compression has two steps. The first one groups the particles using a clustering technique. The second one creates directional distributions over each cluster from the original particle directional data. The result is a set of anisotropic point light sources that represents, in a compact way, the original light source.

3.3.1 Clustering creation

The first step of the method is to group the original set of particles into a set of clusters. At the end, each cluster will have associated a subset of particles and a representative point. The goal is to have a higher cluster density where the emittance is more variable, so we can correctly capture the high frequencies on the emission distribution, both positional and directional.

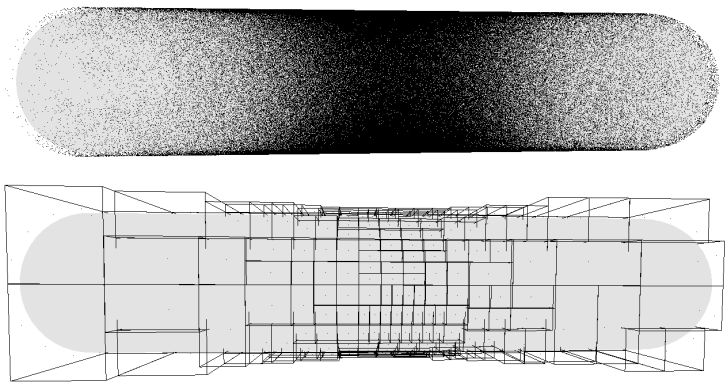


Figure 3.6: Set of clusters (below) that represents the original particle spatial distribution (above).

The algorithm starts with a very dense initial clustering that is computed by creating an octree, see Figure 3.6. The criterion for voxel subdivision in this first step is to have a maximum number of particles in each voxel. This helps avoiding a too fine initial discretization, which would lead to an unacceptable computational cost. The experiments show that octrees with a number of leafs between 20000 and 30000 are enough for a 10M dataset. Once the octree has been created, each leaf voxel of the octree corresponds to a cluster. Then, an iterative process removes unnecessary clusters until no cluster needs to be removed. The particles of the removed clusters are

redistributed to the remaining clusters. Each of these iterations has the following steps:

1. 3D triangulation of cluster representatives. This produces a mesh approximation of the luminaire virtual bounding surface.
2. All clusters are traversed and marked for removal in case that they are unnecessary. A cluster can be removed if it does not help to capture detail. The specific criteria used will be shown below.
3. Cluster removal and particle redistribution. Particles of the removed clusters are redistributed to the nearest neighboring cluster that has not been removed.

After each iteration, the new cluster set is re-triangulated before a new iteration starts. Following, it is explained in more detail the first two steps.

Triangulation

In the first iteration step we need to triangulate the virtual bounding surface of the light source using the cluster representatives as vertices. A cluster representative is the particle location belonging to the given cluster $\mathcal{C}_i$ that is closer to the average location of the subset of particles associated to the cluster. This average location of a cluster is computed as:

$$\mathbf{C}_i = \frac{1}{N_i} \sum_{j}^{N_i} \mathbf{P}_j$$

where $\mathbf{P}_j$ is the position of particle j, and N_i is the number of particles associated to cluster i. Then we choose the representative $\mathbf{R}_i$ as:

$$\mathbf{R}_i = \{\mathbf{P}_n : \forall m \in 1..N_i, m \neq n, \mathbf{P}_m \in \mathcal{C}_i \quad \|\mathbf{P}_m - \mathbf{C}_i\| > \|\mathbf{P}_n - \mathbf{C}_i\|\}$$

There are many methods that calculate 3D triangulated surfaces from point clouds, such as [SR01] or [AB99]. Unfortunately, most of these methods do not guarantee that all input points are used as mesh vertices, considering some of them irrelevant for mesh construction. As we are dealing with clusters of energy-carrying photons, we can not neglect any point. This can occur if the point is not relevant for the mesh definition, or if the point produces a small concavity. In the later case, it would not be possible to extract the faces of tetrahedra that contain this point, and it would be lost (see Figure 3.7).

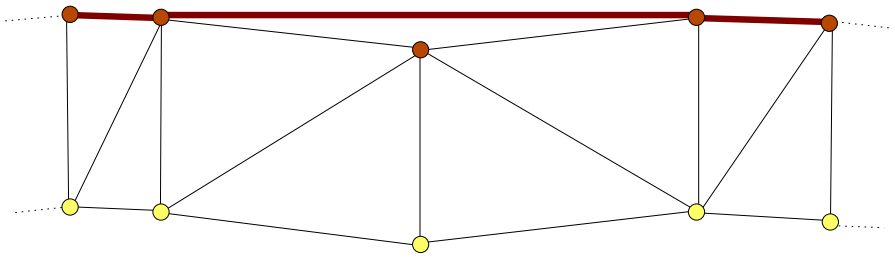


Figure 3.7: If a point particle produces a small concavity, the tetrahedra faces that contain it will be discarded.

So, we need a method that considers all input points, because each point represents a cluster. The classic methods ([AB99][SR01]) have a high computational cost for a simple surface like the ones used by most rayset providers. Also, none of these methods guarantees a mesh with mostly equilateral triangles. As is explained in Section 3.4, the method needs to sample over triangle areas. Thus, to get well distributed samples, the mesh has to be composed by approximately equilateral triangles.

A new method is proposed, that starts from the tetrahedralization (analogous to a 3D triangulation) of the point cluster representation and the later elimination of tetrahedra sides that do not belong to the virtual bounding surface. The tetrahedralization is done by the Delaunay algorithm [She97]. To discard the tetrahedra sides we follow the next algorithm:

1. All input points are duplicated into a new set. The original ones are projected over a sphere that wraps the virtual bounding surface. The new ones are projected over the same sphere but with a larger radius.

2. The overall set of points, the original and the new ones, are used to compute the tetrahedralization.

3. Select those triangles that have their three vertices projected together onto the first bounding sphere.

4. Project again the selected vertices and triangles over the original virtual bounding surface.

In Figure 3.8 a simplified 2D sectional illustration of this method can be observed.

This algorithm is only valid for star-shaped surfaces, but the nature of the input models used in industry already guarantees that. Note that the center of the star must be also the center of the bounding spheres. This algorithm works for simple convex shapes, such as spheres or cylinders. For more

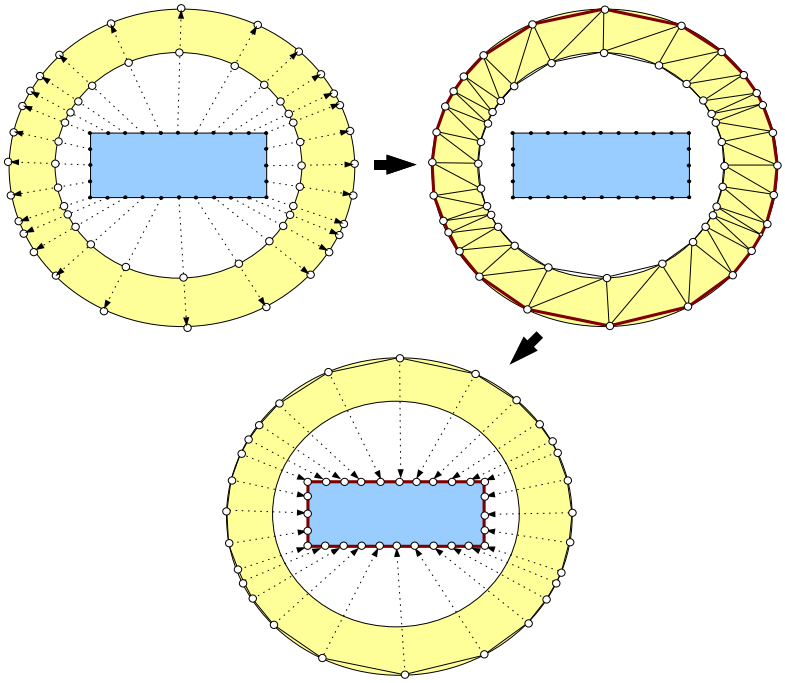


Figure 3.8: Simplified 2D sectional illustration of light source bounding surface triangulation.

complex convex shapes we could use the ball pivoting algorithm [BMR*99] with a large sphere. Note also that the bounding sphere does not assure that all mesh triangles are near to be equilateral triangles, but it can be guaranteed for most of them. One special case is the cylinder light source bounding volume, because the projection on the spheres generates distorted triangles near the caps (see Figure 3.5). Since the point density for this kind of near-fields decreases near bounding cylinder caps (see Section 3.5), this does not affect too much to the overall mesh quality. Other bounding volume shapes have been tested, such as ellipsoids, and different projection methods based on other references, such as axis aligned projections or from light source bounding volume medial axis. In basis on the tested cases, we have found that the sphere bounding is the best choice.

Cluster removal

On the second step of the clustering algorithm, and once the triangulation is finished, all clusters are traversed and tested for removal. The idea is that, if the density of a cluster can be approximated by linear interpolation of the neighboring vertices, then the cluster can be removed since the particle

density is still correctly represented without it. Then, the particles are redistributed among its neighbors, and the set is re-triangulated. Note that it is much more efficient the re-triangulation in this case than keeping information for an incremental update.

The algorithm consists of the following steps. For each cluster $\mathcal{C}_i$, a normal vector $\mathbf{N}_i$ and a plane $\mathcal{S}_i$ are approximated. This approximation is computed by performing a nearest neighbor search centered on the cluster representative $\mathbf{R}_i$. This search of nearest clusters is accelerated by using a KD-tree built with the original particle data. Next, all adjacent cluster representatives $\mathbf{R}_j$ in the triangulated mesh are projected onto $\mathcal{S}_i$ (see top of Figure 3.9). For each projected cluster representative $\mathbf{Q}_j$ and for $\mathbf{R}_i$, and considering only the 2D projected coordinates, the respective density values d_j and d_i are added as a third coordinate. Finally, a new regression plane $\mathcal{S}_r$ is calculated for the new points (see bottom of Figure 3.9). If the projection distance t between d_i and $\mathcal{S}_r$ is larger than a user-defined threshold (called density threshold t_d), the cluster cannot be removed from the mesh. The threshold is a percentage of the maximum projection distance of neighboring clusters. Otherwise, if this threshold test is passed, the distance from the cluster to its neighbors is verified, in a way such that the cluster will not be removed if the distance is larger than a given threshold, called edge filtering threshold t_e. The edge filtering threshold is a percentage of the length of the longest edge of the bounding box, which was observed to be a good distance measure. Finally, the cluster's mean emittance direction is compared with the ones from its neighbors, and the cluster is removed if the angle they form is smaller than a third threshold, the angle threshold t_a. Observe that this last verification avoids collapsing clusters on edges with sharp angles. Note also that this is only a first approximation that works sufficiently well on the experiments.

In Table 3.1 there are some results of clustering creation method.

3.3.2 Creation of Point Light Sources

Once the clustering is finished, we create a point light source for each cluster $\mathcal{C}_i$ at its representative point. The approximation used is to accumulate all the particles to $\mathbf{R}_i$. This will result in a set of directions centered at the point. As the number of directions can be quite high, we have to choose a more compact representation that can be efficiently sampled with importance sampling.

We have used a piece-wise constant set of basis functions defined over the sphere of directions in a hierarchical way. The support of each basis function is a spherical triangle. The sphere of directions is initially subdivided into 8

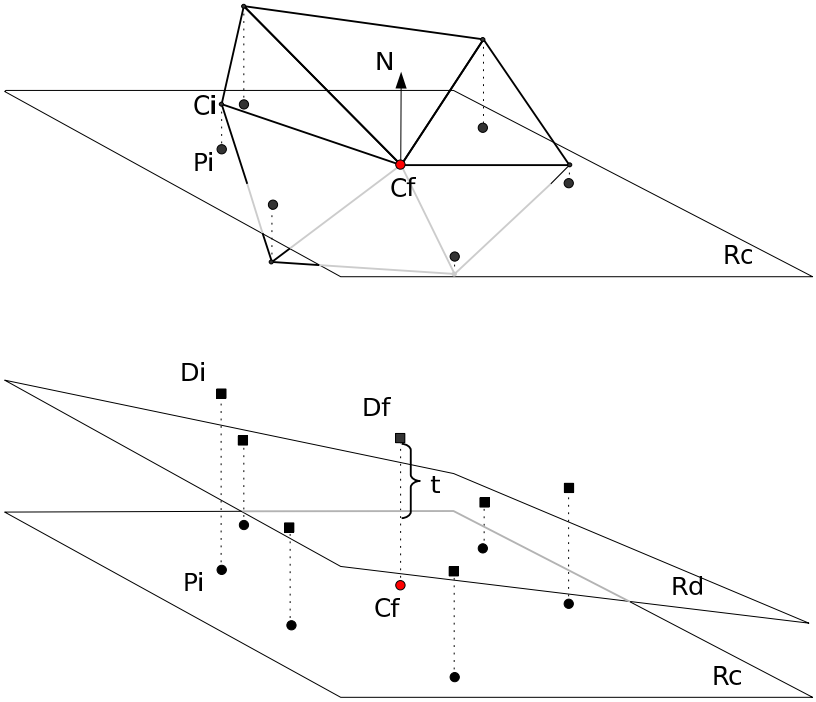


Figure 3.9: Top: neighboring cluster representatives $\mathbf{R}_j$ are projected onto plane $\mathcal{S}_i$. Bottom: each projected cluster representative $\mathbf{Q}_j$ is augmented with its density values. A regression plane is computed with all d_j values.

t_a (deg)	t_d (%)	t_e (%)	Clusters	Loops	Time (sec)	Size MB
90	50	20	347	29	476	1.2
70	45	20	841	38	428	2.6
60	40	20	1680	48	496	5.8
30	35	15	4696	57	531	16.6
25	25	15	8776	51	642	31.6

Table 3.1: Number of clusters, number of loops in the iterative clustering process, precomputation time and resulting memory usage for different thresholds (angle difference, density and edge filtering threshold), for the Osram PowerBall.

spherical triangles, and then each triangle is recursively subdivided using a quaternary tree [Arv95]. We force this subdivision for a fixed initial number of levels so we get a uniform subdivision of the sphere of directions (see

Figure 3.3). Then, a new subdivision is done adaptively to perform a tigh directional representation. For the raysets we have used to test the technique, we have found that between 3 and 5 levels of subdivision are enough to get an accurate representation (see Figure 3.10).

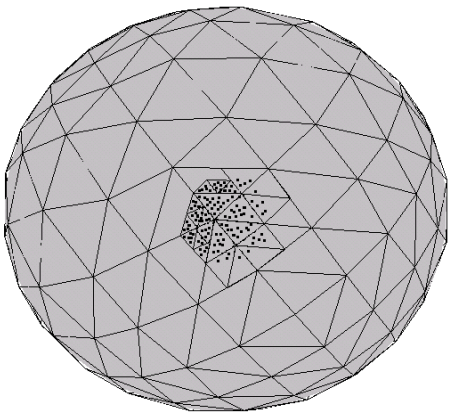


Figure 3.10: Adaptive spherical triangle subdivision for directional data of a cluster. Each point over triangles is a ray direction in directional space.

Once the subdivision is created, we store at each spherical triangle the number of original particles in the cluster with directions that belong to the corresponding solid angle. This number also represents the exitant energy through the corresponding solid angle. To improve compression, we only store the triangles that have non-zero energy. Typically, less than half of the triangles need to be stored as only half of the sphere is pointing towards the source of the light.

3.4 Importance Sampling

Once the model is compressed, it has to be decompressed to be used in a ray shooting algorithm. This is performed by sampling rays on the compressed model by importance sampling. The process has to guarantee that with enough samples, we maintain as much as possible the accuracy of the original rayset.

In order to be able to sample the complete domain, we create a triangulation of the bounding surface using the point light sources as the vertices for such triangulation (see Section 3.2). Once we have created the triangulation, every time we want to generate the 3D position of a particle, we first have

to choose a triangle. We construct a probability density function (*pdf*) for this, and each triangle is assigned a given value proportional to its energy. We set this energy to the amount of original particles that exit the light surface through the given triangle, without taking into account the densities computed for the respective vertices. This value has to be stored with the compressed model.

Therefore, with this *pdf* we can choose a triangle by importance sampling. Then we have to select a random point on the triangle. The straightforward approach would be to choose a point following a uniform distribution. But this poses another problem: the *pdf* that results is not continuous over the edges of the mesh, resulting in illumination artifacts. These artifacts are caused because the spatial distribution generated by this approach changes strongly from triangle to triangle (see Figure 3.11, left).

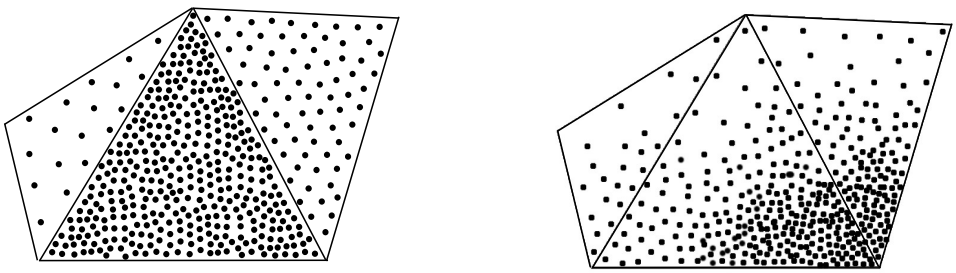


Figure 3.11: Uniform sampling over triangles produces artifacts (left). A continuous sampling across the edges of the mesh avoids them (right).

To solve this problem we propose a non-uniform pdf that is C^0 continuous over the edges of the mesh (see Figure 3.11, right). The idea is to compute the density of particles at each vertex of the mesh and perform a linear interpolation across each polygon. For a triangle n we define the pdf at a point x in parametric space as (see Section 3.4.1):

$$p_n = \frac{d_0 + u(d_1 - d_0) + v(d_2 - d_0)}{2A_n \int_0^1 \int_0^{1-v} (d_0 + u(d_1 - d_0) + v(d_2 - d_0))\, du\, dv}$$

where u, v are the coordinates of point p within the triangle, d_0, d_1 and d_2 are the densities of vertices V_0, V_1 and V_2 (see Figure 3.12), and A_n is the area of the triangle.

Unfortunately, sampling values from p_n is a complex task due to the integration needed over the triangular domain, which would require a slow numerical integration. In order to simplify computations, we created an

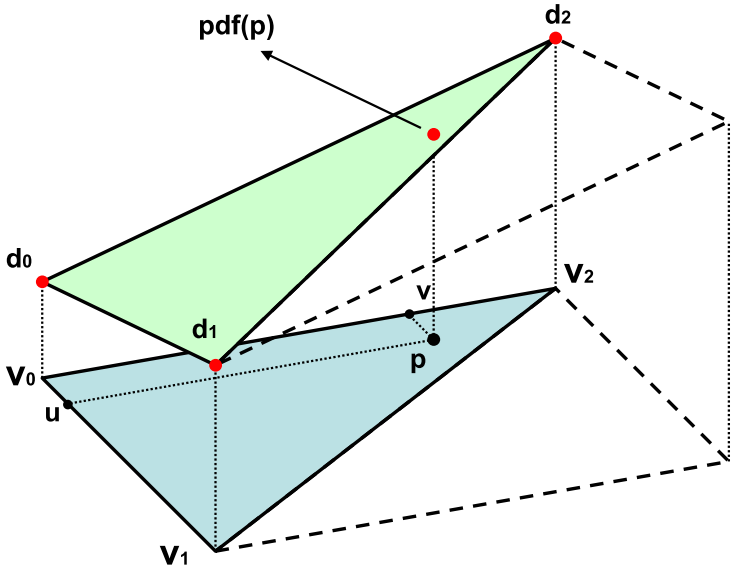


Figure 3.12: Plot of the pdf used for sampling a point inside a triangle. We consider the function over the whole quadrilateral, but samples outside the triangle V_0,V_1,V_2 are rejected.

instrumental distribution p'_n, which is the extension of the previous pdf to the whole unit square, and get the samples by the rejection sampling method [Bek99], see below. To guarantee the positiveness of the new pdf, we choose the origin of the u, v parameterization (vertex V_0) as the vertex with a lower density, allowing us to be sure that the pdf is positive all over the domain. The pdf also holds the condition that the integral over the domain is 1:

$$p'_n(u,v) = \frac{d_0 + u(d_1 - d_0) + v(d_2 - d_0)}{2A_n \int_0^1 \int_0^1 (d_0 + u(d_1 - d_0) + v(d_2 - d_0))\, du\, dv} \tag{3.1}$$

If we want to generate random points proportionally distributed to this pdf we have to apply the principles described in [Scr66, Shi90]. Given two uniformly sampled random numbers $r_u \in [0,1]$ and $r_v \in [0,1]$, the sample point is $u_i = F_0^{-1}(r_u)$, $v_i = F_1^{-1}(r_v)$, where functions F_0 and F_1 are defined using a function F:

$$F(u,v) = \int_0^v \int_0^u p'_n(u',v')\, du'\, dv'$$

and, from here:

$$F_0(u) = F(u,1)$$
$$F_1(v) = \frac{F(u_i,v)}{F(u_i,1)}$$

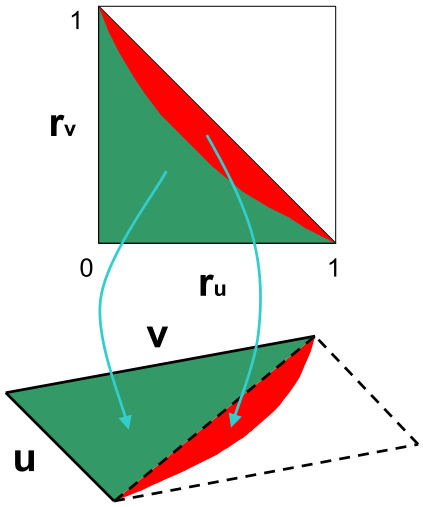


Figure 3.13: Not all of the random sample pairs r_u, r_v such that $r_u + r_v \leq 1$ produce u, v pairs such that $u + v \leq 1$. However, all the random pairs in the green area produce u, v pairs that are not rejected.

Solving these integrals gives a quadratic polynomial that can be easily evaluated. Unfortunately, this mechanism produces samples all over the domain $u_i \in [0, 1]$, $v_i \in [0, 1]$. That means that we have to apply rejection sampling and reject samples that verify $u_i + v_i > 1$. This can be very inefficient since at least 50% of the samples are rejected. Even worse, in case that d_0 is much lower than d_1 and d_2, the number of rejected samples can be much higher.

However, as vertex V_0 corresponds to the lowest density, it is clear that random numbers such that $r_u + r_v > 1$ will always produce invalid samples (see Figure 3.13). Taking this into account, we always generate uniform random points on the triangle defined by equation $r_u + r_v \leq 1$. Results show that, by using this sampling strategy, the number of rejected samples is very small: in the experiments, less than 10% of the samples were rejected.

Figure 3.14 shows the original rayset point distribution (no directions shown) and the point set produced by the sampling technique.

3.4.1 PDF for Position Sampling

In order to have a continuous Probability Density Function defined all over the light surface, it is necessary to propose a well defined, continuous global function, and normalize it. We decided to use a piece-wise linear function defined over each triangle of the triangulation: at each vertex $\mathbf{P}_i$, we require the evaluation of the global function to be $f(\mathbf{P}_i) = d_i$, where d_i is the density

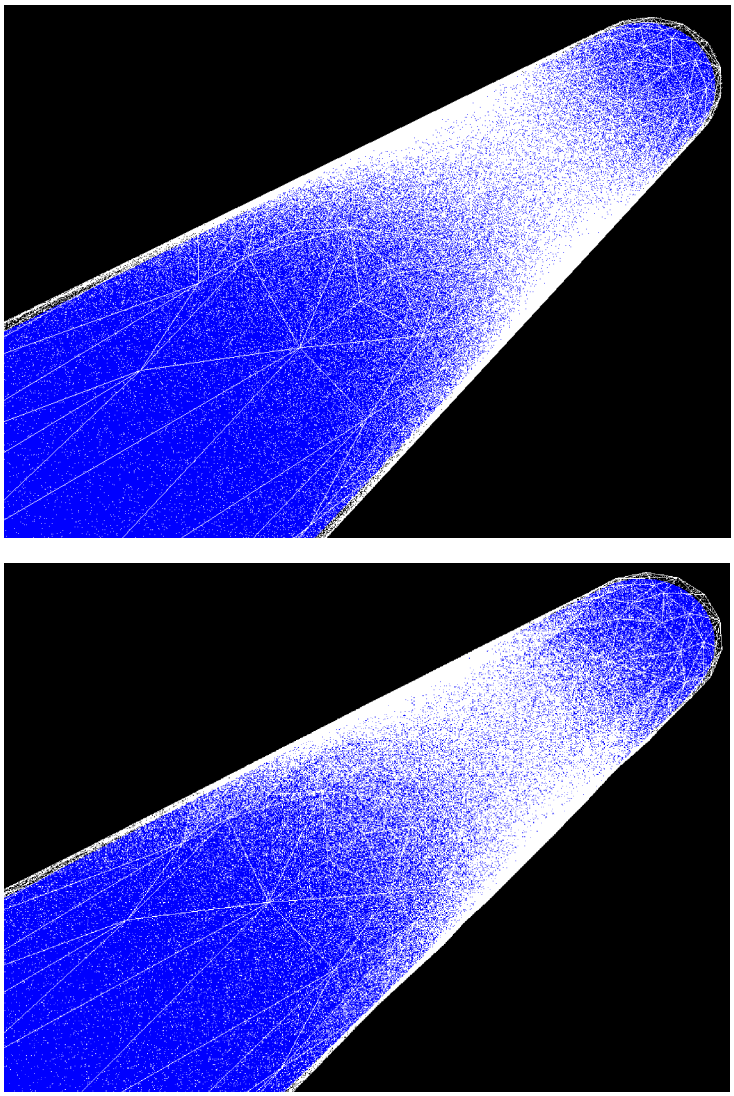


Figure 3.14: Comparison of original rayset with importance sampled set. Top image shows the point distribution of the original rayset, and bottom image shows the point distribution generated by the sampling technique.

associated with the corresponding vertex. Continuity is granted over different triangles as triangles with common vertices will use the same d_i values. This way, continuity is C^∞ within the interior of the triangles and C^0 at the edges.

In the u, v parameter space we can formulate an expression for the n-th triangle

$$f_n(u,v) = (d_1 - d_0)u + (d_2 - d_0)v + d_0$$

Now that we have the function defined over all the triangles i, we can compute the global normalization, defined as the integral $I = \int_\Omega f(\mathbf{r})d\mathbf{r}$ where the domain Ω is the triangulated surface we got. This expression is nothing else than the sum over all N triangles, each with domain Ω_n:

$$I = \int_\Omega f(\mathbf{r})d\mathbf{r} = \sum_{n=0}^{N-1} \int_{\Omega_n} f_n(\mathbf{r})d\mathbf{r}$$

by changing variables to the unit square, we get that this is equal to

$$I = \sum_{n=0}^{N} \int_{Tri_n} f_n(\mathbf{r}(u,v)) \left| \frac{\partial(x,y)}{\partial(u,v)} \right| dudv$$

where the integration domain Tri_n is the lower half triangle in the unit square, and $\frac{\partial(x,y)}{\partial(u,v)}$ is the Jacobian of the transformation [Wu99].

It is important to notice that computing the absolute value of the determinant of the Jacobian gives the same expression as $2A_n$, i.e. twice the area of the triangle $A_n = 1/2|(\mathbf{r_j} - \mathbf{r_i}) \times (\mathbf{r_k} - \mathbf{r_i})|$, where $\mathbf{r_i}$, $\mathbf{r_j}$ and $\mathbf{r_k}$ are the three vertices of the triangle.

Now, we can proceed with the integration

$$I = 2\sum_{n=0}^{N} A_n \int_0^1 \int_0^{1-v} f_n(u,v)dudv = 2\sum_{n=0}^{N} A_n I_n$$

Then, we can write the final pdf as

$$pdf = \frac{\sum_{n=0}^{N} f_n(r)\delta(\Omega_n, r)}{I}$$

where $\delta(\Omega_n, r)$ is 1 over the triangular domain Ω_n and zero everywhere else. In order to have a clearer sampling strategy, we can multiply and divide each term by $A_n I_n$, what will give

$$pdf = \sum_{n=0}^{N} \frac{A_n I_n}{I} \frac{f_n(r)\delta(\Omega_n, r)}{A_n I_n} = \sum_{n=0}^{N} \frac{A_n I_n}{2\sum A_n I_n} \frac{f_n(r)\delta(\Omega_n, r)}{A_n I_n}$$

which clearly is a linear combination of pdfs [Bek99]. Actually, we can say that it is a linear combination of N primary pdf's p_n:

$$p_n = \frac{f_n(r)\delta(\Omega_n, r)}{2A_nI_n} = \frac{f_n^{'}(r(u,v))}{2A_nI_n}$$

To be able to sample the pdf, a primary p_n is selected first with probability

$$p(n) = \frac{A_nI_n}{\sum A_nI_n}$$

and next a sample is drawn using p_n. So, the final expression for the *pdf* becomes

$$pdf = \sum_{n=0}^{N} p(n)p_n$$

Although only a single primary pdf is sampled, the result of that sampling is obtained following the combined pdf. Obviously, when sampling p_n the $\delta(\Omega_n, x, y)$ factor can be omitted, as it was built to be 1 all over the domain.

3.4.2 Sampling Directions

Sampling a direction involves using importance sampling on the three directional distributions stored at the vertices of the triangle.

Once we have selected a point x, we compute its barycentric coordinates u, v and $1-u-v$. As their sum is 1, we can use these values to construct a small CDF for the three vertices. This allows us to select one vertex and its corresponding point light source. Then, we can perform importance sampling on the point light source by creating a CDF from all the spherical triangles. With the CDF we can select a spherical triangle and then sample it uniformly with respect to the solid angle.

In Figure 3.15 is shown the overall scheme of the full particle generation algorithm.

3.4.3 Direct illumination

As explained above, the light source is represented by a bounding geometry, which can be sampled to find the illumination at a given point. But, for lights with a high directional variation, this approach for a scene point can be quite inefficient, for example in those cases where we need to know the received energy from the light source. If we think of this light source model as a *many light sources model*, we could use a light tree technique [War94][SWZ96][PPD98][WFA*05] to adaptively sample the light source depending on the relative position of the illuminated point.

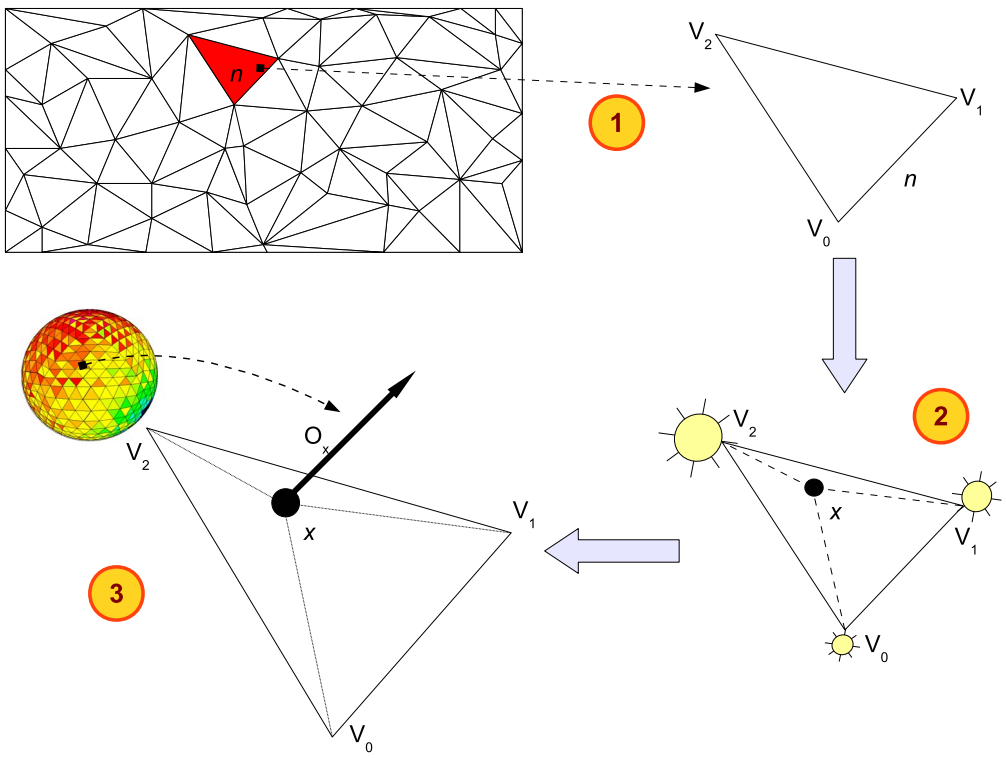


Figure 3.15: Overall sampling method. At step 1, one triangle is sampled from the mesh. At step 2, a position x inside the triangle is calculated by sampling the weighted barycentric coordinates. At step 3, a direction O_x is sampled from the stored directional distribution on one of the triangle vertices V_i

Lightcuts

We will use the *Lightcuts* technique [WFA*05] that has been proven to be very effective for complex illumination environments.

This technique builds a binary tree for all light sources. At each node there is a point light source that represents the illumination of all the lights in its sub-tree. For each point x that we want to illuminate we have to create a *cut*. A *cut* is defined as the set of nodes of the tree such that every path from the root node to a leaf contains just one node from the *cut* (see Figure 3.16). If S is the set of nodes of a given *cut* then the direct illumination at point x will be:

$$L_S(x,\omega) = \sum_{i \in S} M_i(x,\omega) G_i(x) V_i(x) I_i$$

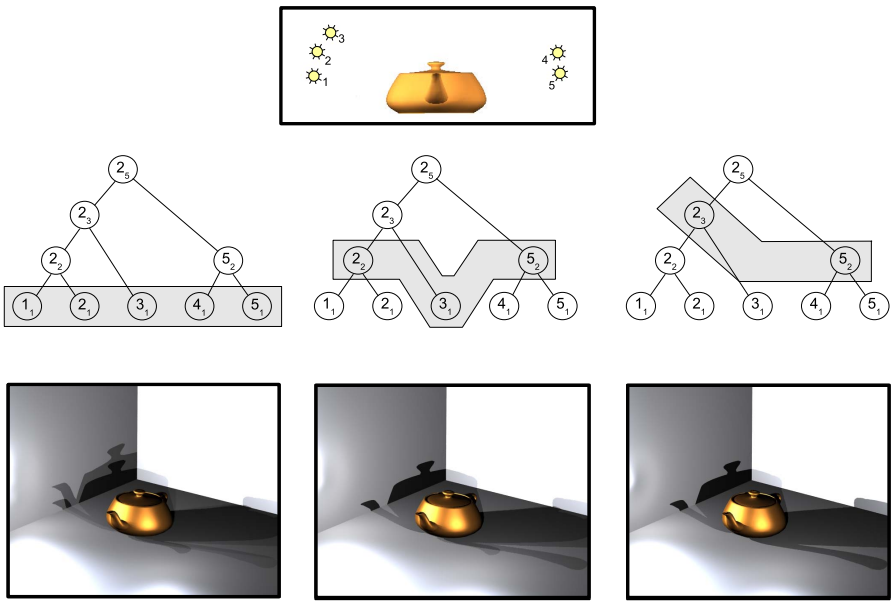


Figure 3.16: Lightcuts example. The scene have 5 point light sources. A light tree is constructed, where each node shows the main light source number at this tree level and branch, and the number contained light sources as subscript. From left to right, three examples show how the illumination changes with different *cuts*.

where $M_i(x, \omega)$ is the material term (BRDF), $G_i(x)$ is the geometric term, $V_i(x)$ is the visibility term, and I_i is the intensity of light i. Each node j of the *cut* represents a set of point light sources S_j that correspond to the leaves of its sub-tree. The direct illumination from the cluster j can be approximated as:

$$L_j(x, \omega) \approx M_j(x, \omega) G_j(x) V_j(x) \sum_{i \in S_j} I_i$$

This equation means that the illumination is approximated using a single material, geometric factor, and visibility term, computed using the representative point of node j. The intensity of the node is the sum of the intensities of the lights it represents. The lightcuts approach needs to compute bounds on terms M, G and V in order to ensure the accuracy and smoothness of the approximation. However, the lightcuts technique defines bounds for a small set of light sources: omnidirectional, directional and oriented. None of these types can be used directly for our non-uniform directional distributions. Moreover, the fact that our method represents the original light source with

a relatively small set of point light sources (a few hundred) makes it impossible to use these points as the leaves of the tree since it would be a sampling too coarse for nearby objects. Finding a tight bound for point light sources with arbitrary directional distributions is very complex. But even worse is the fact that it would be too computationally expensive since it would mean to evaluate the directional distributions over projections of the bounding box on the sphere. We have chosen a much simpler approach even if it is not a very tight bound. We simply fit a cosine directional distribution that represents an upper bound of the arbitrary directional distribution. This way we simply apply the strategy for oriented lights.

Light ray tracing

The nature of the rayset makes it a perfect choice to be used in a light ray tracing algorithm. The same fact can be considered for the compressed model, as it is based on the original rayset. To use a rayset into a light tracing algorithm, the model has to be preloaded before the rendering starts, or a stored data set has to be queried each time a new light ray is needed. None of these solutions is efficient, because of the huge memory requirements to preload the rayset, or because of the computing time delays on querying the stored data set. On the other hand, the compressed rayset can be preloaded without many computer resources, and each time a new light ray is needed in rendering process, a new ray is sampled from the compressed data in a fast way, as we have already explained.

We have tested the model with the *Photon Mapping* algorithm (see Section 2.2.2). Like in [GGHS03a], the global map is separated in the *direct map*, that stores the direct light source impacts, and a new global map, that stores indirect lighting. Once the direct map is created, it can be used to reconstruct the illumination directly from the map. This is very efficient for highly directional sources, which can be made even better by storing a large quantity of photons in the direct map, while keeping a low quantity for the global map.

3.5 Results

We have tested the method with two raysets corresponding to real measurements. Both of them have 10 million particles. The first one corresponds to an OSRAM PowerBall bulb (courtesy of Lambda Research), and the second one is a Tungsten Halogen bulb (Radiant Imaging demo) (see Figure 3.17).

Also, we have tested four synthetic raysets (see Figure 3.18), sampling 10

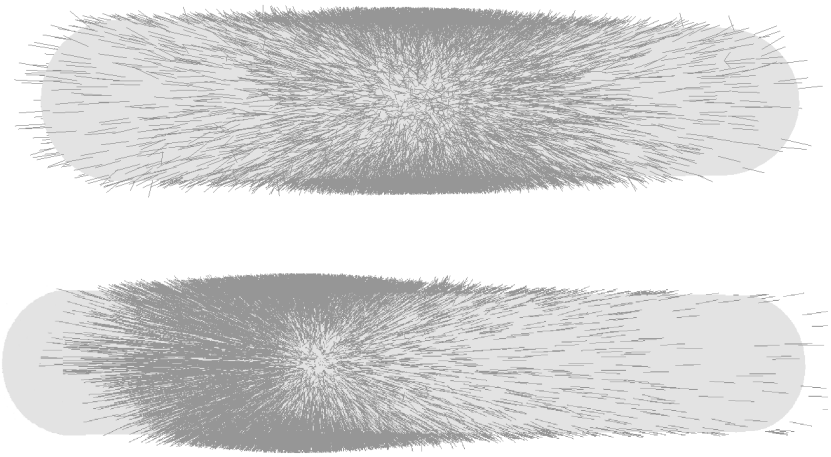


Figure 3.17: Real-measured raysets. At top, the OSRAM PowerBall model. At bottom, the Tungsten Halogen model.

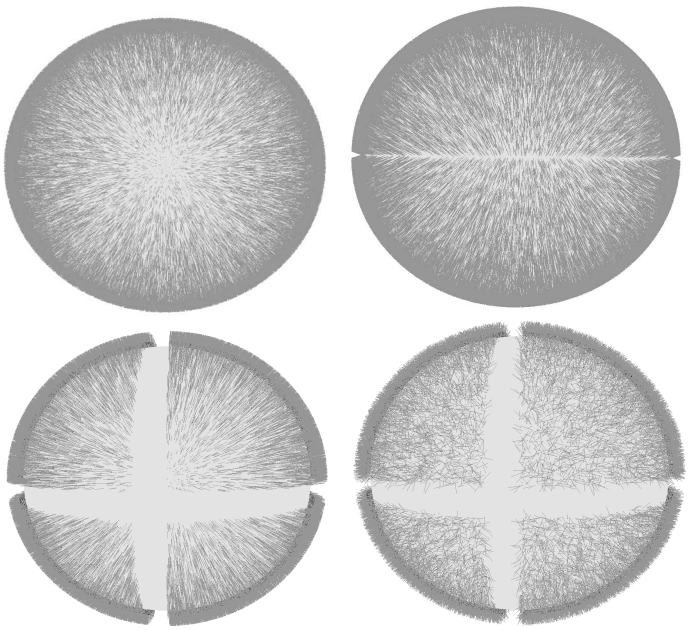


Figure 3.18: Synthetic tested raysets: Phong (top left corner, Phong exponent = 500), Phong directional pattern (top right corner, Phong exponent = 25), radial with pattern (bottom left corner) and cosine with pattern (bottom right corner, exponent = 1) distributions.

million particles for each one in a uniform way. The first one has a Phong distribution over the sphere. We use the same sampling method than [LW94] to construct the Phong distribution. The second one has Phong distribution, but with a directional pattern distribution. The other two are radial and a cosine ray direction distribution over the sphere, but with a positional pattern onto the sphere of origins. These synthetic raysets are used to check the method performance in different conditions, such as high frequencies in ray positions or directions. It is specially interesting the case of the Phong directional pattern distribution, which leads to a triangularization which is shown in Figure 3.19, showing that the angle threshold criteria for the triangularization effectively preserves the discontinuity in the distribution.

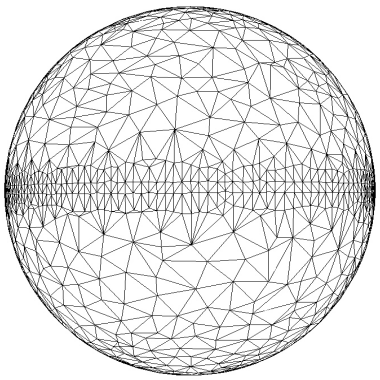


Figure 3.19: The resulting triangulation of the Phong directional distribution with a directional patterns shown on the right top part of Figure 3.18.

Figures 3.20, 3.21 and 3.22 show false color images for particle emission experiments. The images represent the energy arriving at a plane that is located 1 mm from the bounding surface of the rayset. There is a side by side comparison between the $10e6$ original particles and a $10e6$ particle emission using importance sampling from the compressed datasets. Also, difference images are displayed for each one. Observe that, with the method explained above, no photons are generated *inside* the bounding surface or *pointing inwards* from it, so any surface intersecting its interior will not receive any hit. Actually, this cannot be a problem since this is the space physically occupied by the light bulb itself.

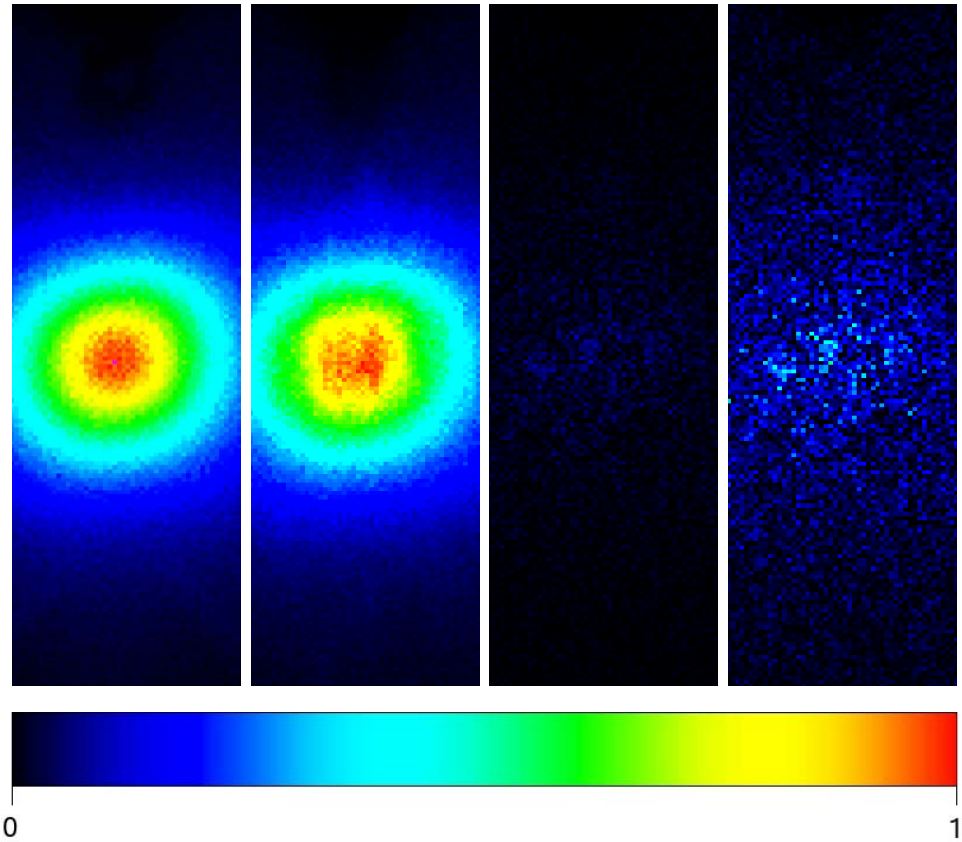


Figure 3.20: Images of 10 million particles gathered on a plane situated at 1mm of the bounding surface. The images correspond to the OSRAM PowerBall bulb with a compressed data of 1680 clusters (see Table 3.2). In columns, from left to right, the images correspond to original rayset, sampled compressed data, difference image (error), and scaled difference image respectively (x5). Under the false color images you can find the scale used, normalized over the entire set of positions/directions.

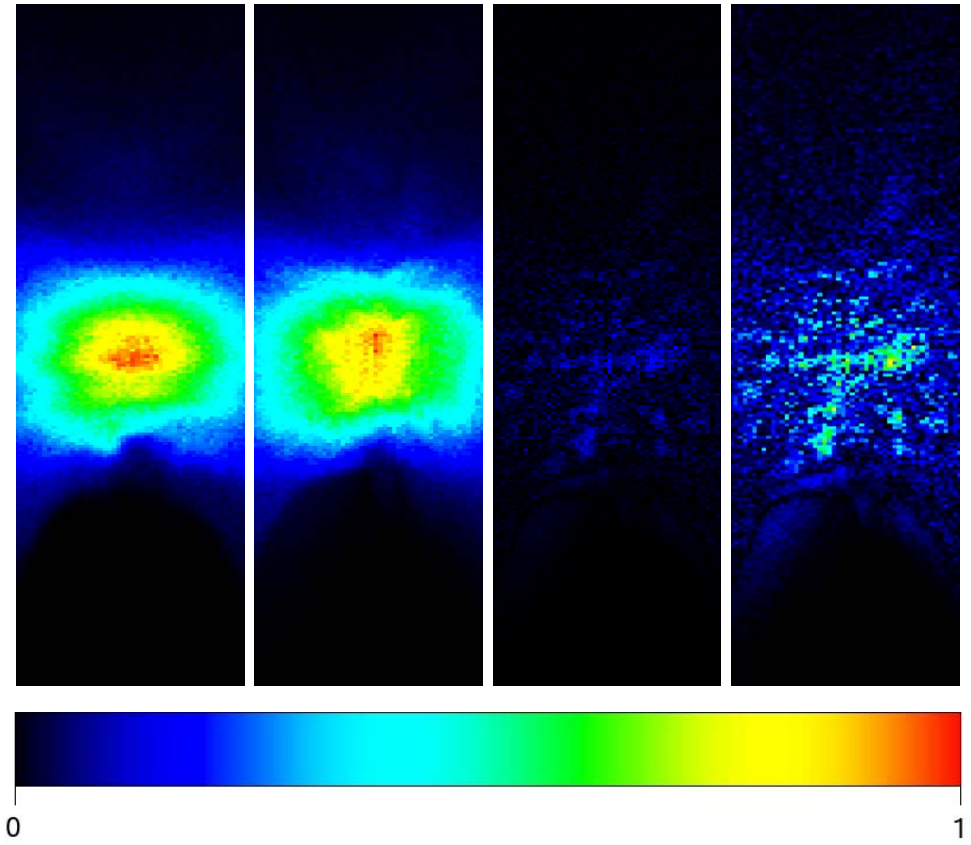


Figure 3.21: Images of 10 million particles gathered on a plane situated at 1mm of the bounding surface. The images correspond to the Tungsten Halogen bulb with a compressed data of 452 clusters 3.2). In columns, from left to right, the images correspond to the original rayset, the sampled compressed data, the difference image (error), and the scaled difference image respectively (x5). Under the false color images you can find the scale used, normalized over the entire set of positions/directions.

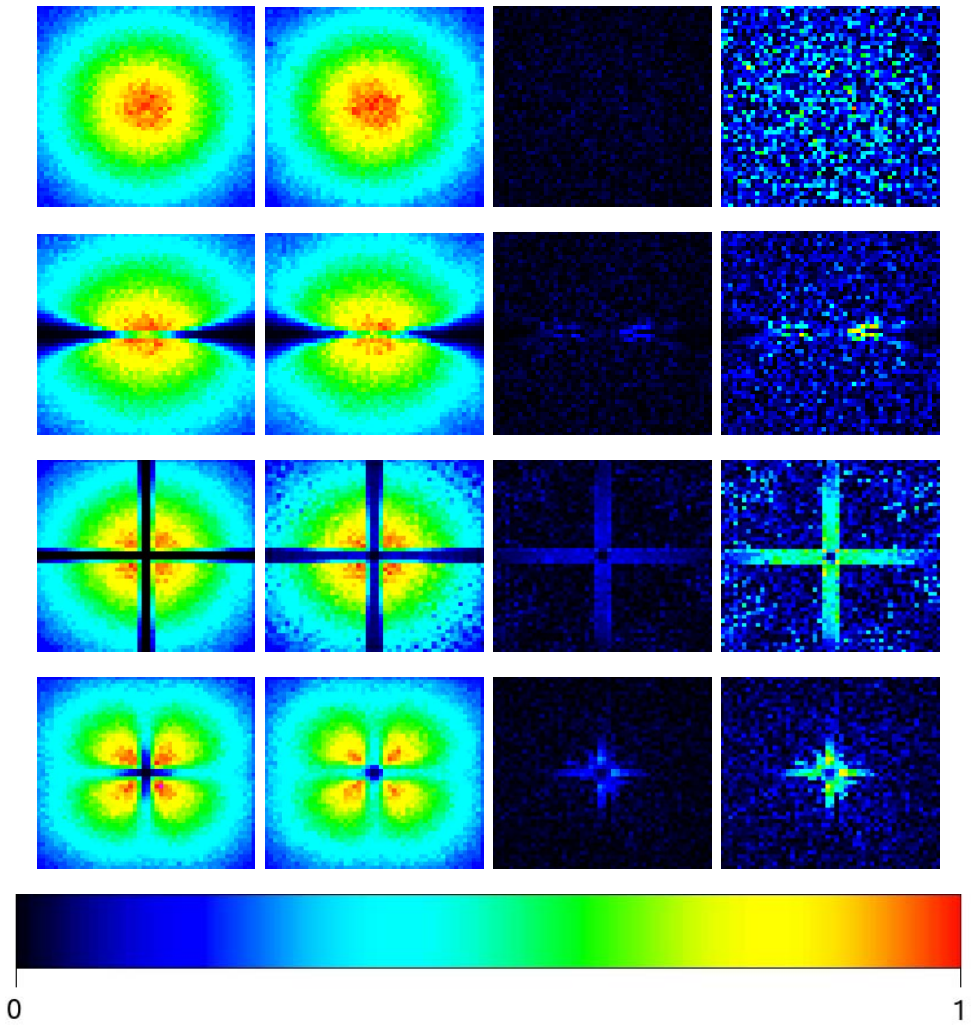


Figure 3.22: Images of 10 million particles gathered in a planes situated at 1mm of the bounding surface. First row corresponds to the Phong synthetic rayset, using a compressed data of 1597 clusters (see Table 3.2). Second row corresponds to the Phong Pattern synthetic rayset, using a compressed data with 1146 clusters. Third row corresponds to the radial pattern synthetic rayset, using a compressed data of 4454 clusters. And the fourth row corresponds to the Cosine Pattern synthetic rayset, using a compressed data of 2244 clusters. In columns, from left to right, the images correspond to the original rayset, the sampled compressed data, the difference image (error), and the scaled difference image respectively (Phong model at x8, and the others at x3). Under the false color images you can find the scale used, normalized over the entire set of positions/directions.

We have tested different compression levels for each one of the avaliable raysets. The memory sizes have been reduced drastically, as can be seen in Figure 3.23, since the rayset representation of all of these models has a memory consumption of about 270MB. In Figure 3.24 and Figure 3.25 there are some results for the OSRAM PowerBall rayset using different compression levels and measuring the error at different distances. Two error metrics have been used: l^2

$$D_{l^2}(a,b) = \sqrt{\sum_i^N (a_i - b_i)^2}$$

and Hellinger [RFS03],

$$D_{Hellinger}(a,b) = \sqrt{\frac{\sum_i^N (\sqrt{\frac{a_i}{N}} - \sqrt{\frac{b_i}{N}})^2}{2}}$$

with similar behavior on the results.

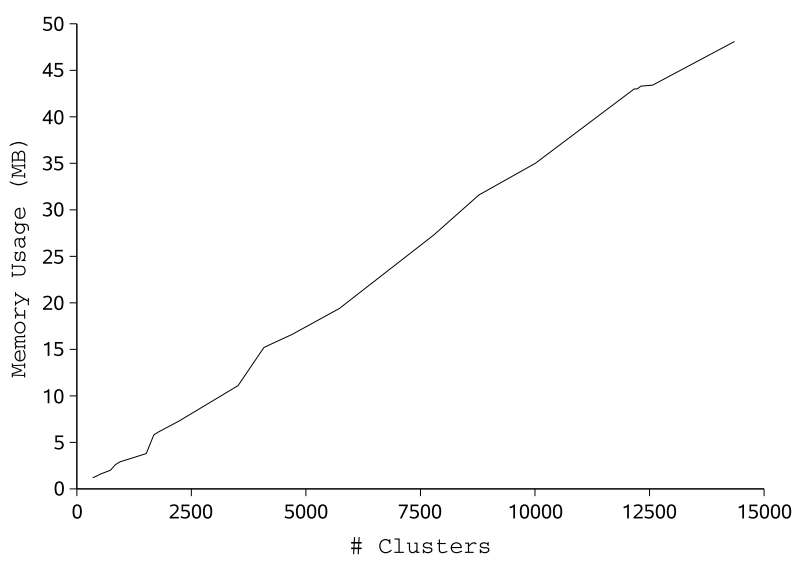


Figure 3.23: Relationship between number of clusters and memory usage for the OSRAM PowerBall.

In Figure 3.24 it can be observed how the error decreases as the number of clusters increases, in the same way for each tested distance.

In Figure 3.25 three zones of interest are shown. The first one is the error obtained at near distances, as 1 mm. In this case the importance sampling

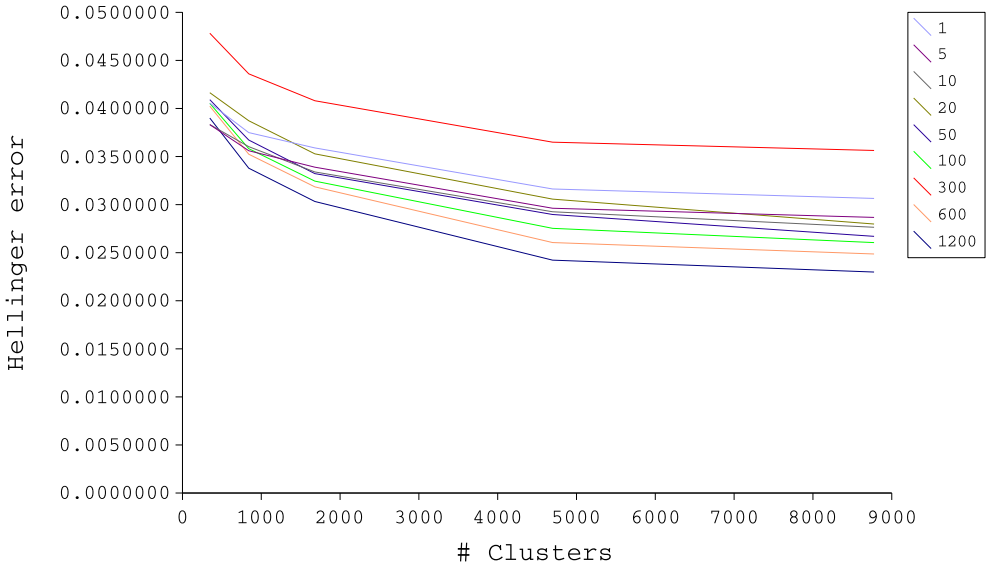


Figure 3.24: OSRAM PowerBall Hellinger errors for different measurement distances in function of the number of clusters .

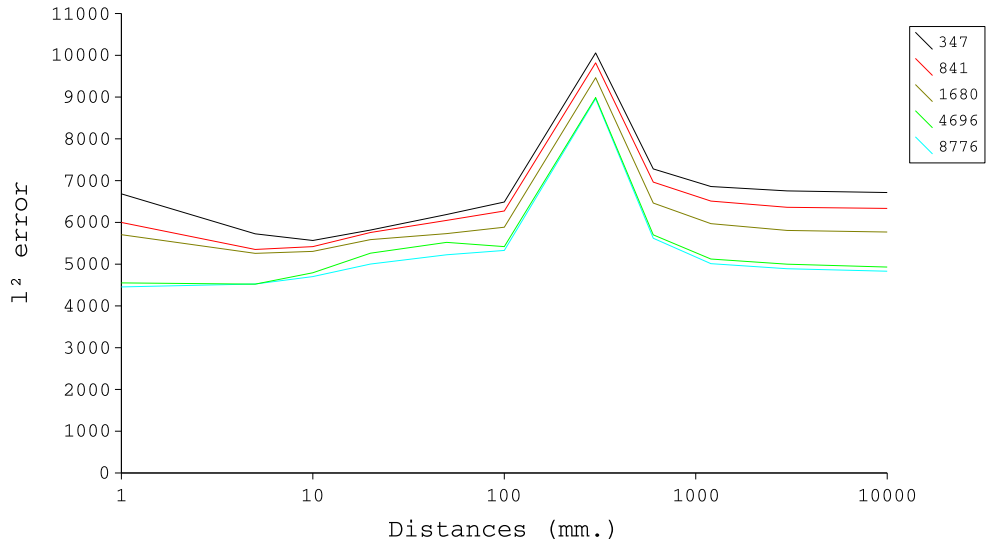


Figure 3.25: OSRAM PowerBall l^2 errors for different number of clusters in function of measurement distances .

positional error is the main contributor to the overall error. The second one is the error obtained at large distances. Here, the directional sampling error

was found to be the main source of error. The third case is the peak observed at distance 300 mm. To explain it, we have traced the particles of original rayset on a set of bounding spheres of different radii. The results (Figures 3.26 and 3.27) show that, at distance 300 mm, it can be observed a pattern over the sphere. This is because of the pattern of the acquisition mechanism that has been used to obtain the rayset can be found in this region. If the gonio-photometer used in the acquisition system uses photosensors placed over a virtual bounding sphere, then the gathering distance (bounding sphere radii) is 300 mm. So, each accumulation point in the pattern corresponds to each photosensor position in the acquisition process.

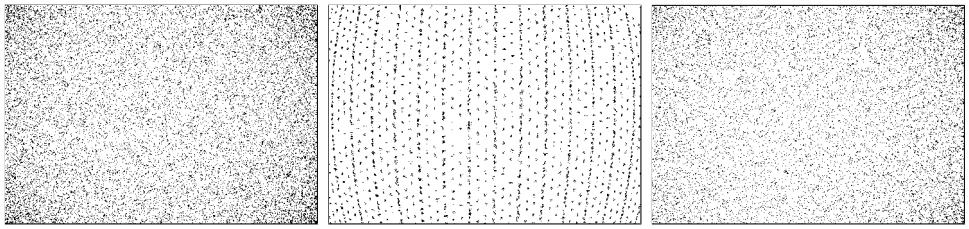


Figure 3.26: Ray gathering over bounding spheres at different distances (from left to right, 50, 300 and 1200 mm). At distance of 300mm appears a pattern due the acquisition method.

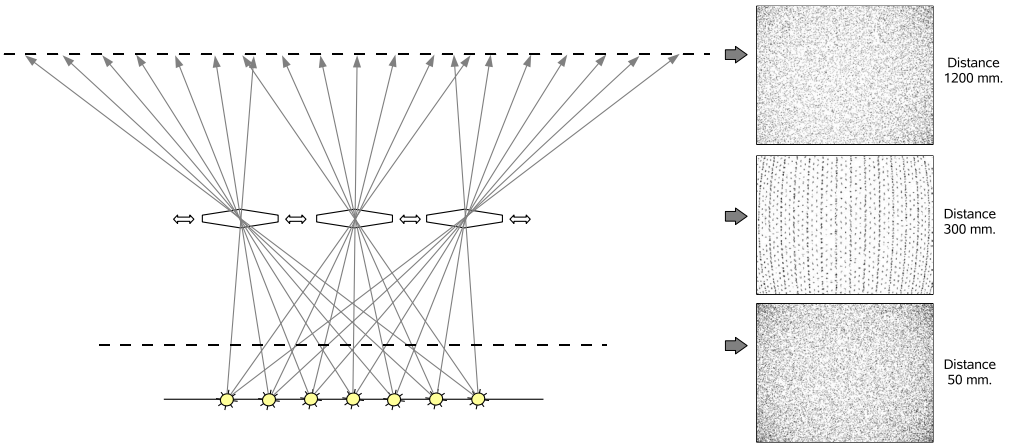


Figure 3.27: Left: acquisition system scheme. Right: ray gathering over bounding spheres at different distances. The observed pattern at distance of 300 mm corresponds to photosensor distance, and each shot accumulation is each photosensor placement.

All the other models have also been tested. In Table 3.2 there is a sum-

mary of l^2 error values obtained for each rayset, with different numbers of clusters, and at different distances from the light sources. All models show similar behaviors to the previously explained one. As a reference, we have included for each rayset the errors for a far-field distribution created with the original raysets, but using 2048 spherical triangles, as a distribution with less spherical triangles fails to keep the different pattern details. As we can see, the new compression method outperforms the far-field representation at short distances, also demonstrating that a far-field representation is unsuited for real light bulbs at short distances, as they cannot be approximated by an anisotropic point light. At large distances compared with the size of the light bulb, both converge to the same values, showing that for those distances it is better to use a far field representation, because of the easy evaluation. However, many applications (e.g. reflector design) require evaluations at short distances, where a far-field is clearly not good enough. One further point should be noted: all measurements in Table 3.2 have been evaluated by the procedure described in Section 3.4, so they have a variance associated. The variance depends on the the emitting distribution, the more diffuse, the more variance, as shown in [PPV04]. So, the l^2 error values have a variance, which we have measured to range from ± 31 and ± 64 for the Tungsten Halogen and the OSRAM Powerball respectively, to values of ± 125 for the cosine pattern (which is like a Phong lobe with exponent $k = 1$), of ± 64 for the radial pattern, of ± 41 for the Phong and Phong pattern distributions (with $k = 500$). This variance is enough to explain some strange behaviors at large distances for some distributions, as the values plus their respective variances overlap, as happens for the cosine pattern distribution at 100 and 1200 mm. Also, in Table 3.2 we have included the resulting sizes of each compressed set, clearly showing the much lower memory usage required by the compression method.

To prove that the representation is accurate enough for cases such as reflector design, we show in Figure 3.28 a set of renderings of the OSRAM PowerBall bulb model mounted in a reflector, illuminating a plane. We have used three representations of this bulb: the original rayset, the compressed rayset and the farfield. The compressed model has 1680 clusters (see Table 3.2).

Some examples have been rendered using the Mental Ray Renderer on Maya®. To do it, we have developed a plugin that works as interface between the compressed rayset and the Maya®rendering system. In Figure 3.29 you can see a comparison between two Photon Mapping results (without gathering), one using the original rayset, and the other using the compressed rayset, both placed in a near (1 mm.) bounding box around the light source bounding volume. The figures are rendered using only the Direct Map mentioned in Section 3.4.3. The figure uses the OSRAM Powerball example, which has

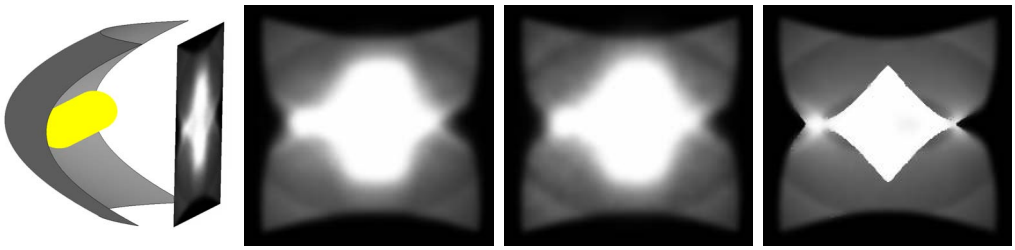


Figure 3.28: Lighting from a reflector with the OSRAM Powerball mounted in. At left, the reflector and bulb setup, and the plane used to gather the lighting. Next, from left to right, the lighting using the original rayset, using the compressed rayset (1680 clusters, see Table 3.2) and using only the bulb farfield.

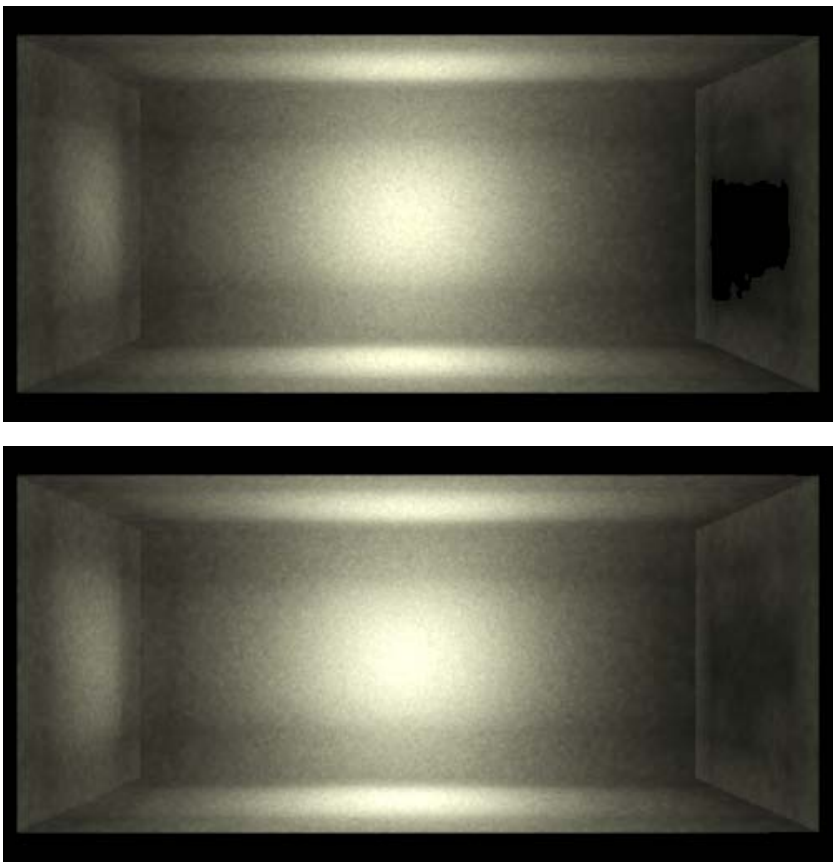


Figure 3.29: Photon Map results (without gathering). At top there are the original rayset result. At bottom there is the compressed rayset result.

a bounding cylindrical shape (70 mm. length and 20 mm. diameter dimen-

sions). There are some gaps on the illumination of the sides of the box for original rayset, because very few photons are emitted at the cylinder caps. These gaps disappear on compressed rayset results, due to an insufficient sampling, creating a smooth filtering effect.

The same comparison has been done in 3.30. In this case, a bigger scene has been compared. Here, the difference between using a near-field or a far-field is minimum. Therefore, the differences are related just for directional light distribution component. Again, the main differences between both images are due an insufficient sampling.

Finally, a similar example than 3.29 has been rendered, this time using Lightcuts (see Figure 3.31). There are two noticeable problems with that result. First, the lighting is quite different from previous examples. Second, there is a triangle pattern on lighted box sides. Both problems are related to the same one: the Lightcuts technique assumes a small error because of the choosing of only the more relevant light sources. This error is not relevant if simple light sources are used, such as point or oriented light sources. But for this case, this error is enough to change the tight near-field light distribution. In addition, the undersampling method introduces some pattern effects because the new points are sampled uniformly using a constant energy value for each triangle. We can conclude that we cannot render highly directional distributions with LightCuts.

3.6 Discussion

The most time and storage consuming part of the method is the directional distribution management. Obtaining the spherical triangles from a given direction means descending through the levels of the subdivision, and this is a costly operation. It must be taken into account that this kind of directional non-analytical representations always have a higher cost than analytic distributions [LRR04, MPBM03].

On the other hand, directional distributions are also very storage consuming. In the current technique, we have the advantage that, with a low number of point light sources, we can characterize the illumination distribution of the bulb and, as we only store information of the spherical triangles with non-zero energy, the total memory used is not too high in comparison.

Finally, it has been demonstrated that this model is suitable for ray shooting rendering algorithms.

Figure 3.30: Scene rendered with the same light source than 3.29 using Photon Mapping. At top, results for the original rayset. At bottom, results for the compressed rayset.

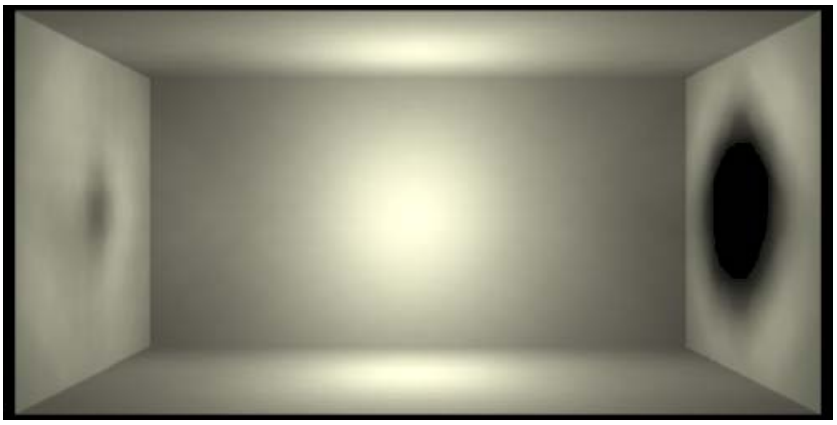


Figure 3.31: Image obtained with a LightCuts implementation for Mental Ray on Maya. The images represent direct illumination (no bounces) for a box located at 50mm from the bounding surface of the light source (the same light source than 3.29. There is no possible comparison because the rayset model cannot be used for direct illumination since it is a set of particles.

rayset	♯Clust.	Size (MB)	Distances (mm.)								
			1	**5**	**10**	**50**	**100**	**300**	**1200**	**3000**	**10000**
Osram	347	1.2	6682.60	5726.10	5566.62	6188.14	6488.13	10057.50	6858.35	6754.26	6714.48
Power-	1680	5.8	5705.55	5258.51	5306.23	5731.51	5884.42	9463.65	5969.38	5807.02	5768.99
Ball	8776	31.6	4455.03	4525.02	4702.06	5224.36	5328.28	8960.63	5011.70	4890.81	4831.09
	FF	0.017	12567.00	9957.12	8288.00	5356.63	4964.11	8850.31	4741.15	4586.17	4530.26
Tungsten	452	0.66	6222.04	6559.96	6866.29	7911.24	8383.64	8888.70	9167.13	9227.08	9251.94
Halogen	1160	2	4664.81	4923.44	5159.80	5672.74	5875.72	6125.81	6293.39	6281.38	6274.02
	2702	2.5	4460.94	4713.48	4948.81	5068.34	5159.71	5265.77	5369.85	5329.95	5345.32
	FF	0.017	6215.19	5764.81	5595.39	5456.35	5576.94	5575.54	5640.79	5638.35	5600.29
Phong	432	0.94	10672.50	12590.80	11269.40	7594.12	7424.74	7288.99	7232.32	7268.94	7277.87
Pattern	658	1.4	9085.03	10265.20	8963.22	6897.97	7224.52	7191.77	7023.90	7056.02	7054.05
$k = 25$	1146	3.2	5844.87	6301.60	5876.75	6179.90	6744.38	6669.15	6430.00	6442.95	6478.13
	FF	0.017	50999.89	48431.90	45218.50	27912.10	18765.60	9441.57	6165.64	5894.46	5863.09
Phong	430	0.76	4609.50	4647.55	4706.65	5426.99	5977.92	6555.81	6811.54	6874.29	6905.22
$k = 500$	1597	2.2	4419.81	4469.87	4550.48	5207.57	5553.68	5872.38	6033.69	6070.29	6091.30
	5349	6	4281.86	4367.94	4474.97	5121.33	5207.98	5032.52	4975.92	5007.21	5028.44
	FF	0.017	4439.08	4413.14	4408.29	4362.60	4328.09	4316.66	4291.46	4288.56	4294.95
Radial	350	0.35	27099.80	27008.30	26889.60	26619.90	26634.10	26762.00	26858.60	26888.90	26904.40
Pattern	1431	0.7	20322.60	20253.20	20166.90	19812.40	19765.60	19892.40	20041.90	20083.60	20105.70
	4454	1.3	10765.80	10794.50	10828.90	10675.10	10432.10	10148.10	10080.50	10078.10	10069.90
	FF	0.017	18449.10	18436.50	18420.40	18376.50	18364.00	18359.10	18354.90	18354.10	18352.00
Cosine	2244	16.9	6687.94	4544.10	4433.10	4430.48	4461.37	4711.09	4749.27	4471.47	4429.14
Pattern	3782	29.9	5300.57	4381.88	4391.92	4384.90	4423.53	4703.22	4883.37	4550.63	4449.72
	8177	71.5	4760.15	4248.68	4287.43	4375.61	4483.85	4625.79	4998.64	4542.61	4391.55
	FF	0.017	22778.30	15483.10	11775.30	5671.10	4850.43	4584.34	4435.97	4340.02	4308.85

Table 3.2: Summary table of memory storage needs and l^2 errors for the tested raysets. Three representative cluster solutions and a far-field (FF) representation have been tested for each rayset at different distances from the light source. The far-field spherical triangle subdivision is similar for each case, so the memory usages differ only in a few bytes

Chapter 4

A fast algorithm for reflector lighting

Global illumination algorithms perform realistic renderings to simulate the light interaction through a scene. For our case this means that we need to calculate how the light rays are traced between the bulb and the reflector. The outgoing rays of this light ray tracing define the reflector light distribution. The objective is to obtain the reflector light distribution in a fast way and compare it with the desired light distribution. This comparison will be used to drive an optimization algorithm (explained more in detail on the next chapter) that will search for the best solution.

In this chapter we will present a new GPU-based method to compute the reflector light distribution in a fast way. This new method works completely in the GPU, including the light ray tracing and the reflector light distribution comparison with the desired one. We show that our method can calculate the reflector lighting at least one order of magnitude faster than previous methods, even with millions of rays, complex geometries and light sources.

The rest of chapter is organized as follows. First it is explained in Section 4.1 the overview of the method. Then, the explanation of how the input data is processed is presented in Section 4.2. In Section 4.3 it is proposed a GPU light ray tracing algorithm to compute the reflector light distribution. The light distribution comparison algorithm is explained in Section 4.4. The main results are presented in Section 4.5. In addition, in Section 4.6 is briefly explained a new released full GPU ray tracing engine. Finally, some discussion on results and the method is presented in 4.7.

4.1 Overview

The goal is to obtain a reflector shape that produces a minimum error between the desired and the resulting light distributions. This is accomplished using an optimization algorithm that minimizes this error (see Chapter 5). The most expensive part of an optimization method is the evaluation of the function to minimize. In our case, the function evaluation is the reflector lighting simulation and the light distributions comparison to get the error. Therefore, the main objective here is to define a method to evaluate this function in a fast way.

The problem has three inputs: the light source, the desired luminaire light distribution and the reflector shape. The light source is represented by an implicit rayset or any other light source data representation able to sample a set of rays, such as the compressed rayset model seen in Chapter 3. Because the light source is very close to the reflector surface, we need a near-field to get precise results. The desired outgoing light distribution is a far-field representation to match with industry requirements, based on industry standard formats (IESNA [ANS02], EULUMDAT [bCL99]). However, the presented algorithm can deal with more complex representations, like near-fields, as well. Finally, the reflector shape is defined as a mesh that is able to be manufactured through a press-forming process. Therefore, the reflector shape is deformed only in one direction. The reflector material is considered as purely specular, and only a reflectance attenuation factor is taken into account.

The method has three main steps:

- The input data is processed.
- A light ray tracing algorithm computes the reflector lighting distribution in a fast way.
- The reflector light distribution and the desired light distribution are compared, and the difference value is returned.

The overall algorithm, called *FIRD (Fast Inverse Reflector Design)* is implemented using GPU shaders (see Figure 4.1), where each GPU fragment processes a light ray. This results in a very fast algorithm that is able, even for millions of rays and complex reflector geometry shapes, to calculate the reflector lighting is a few seconds, as shown in Section 4.5.

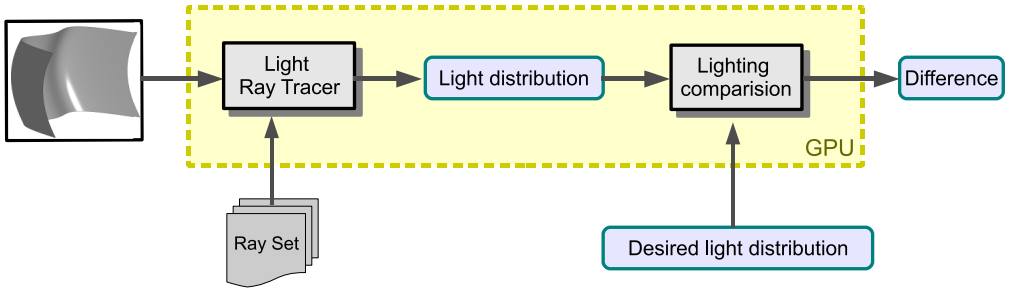


Figure 4.1: Overall scheme of the method.

4.2 Preprocessing of the input data

The user-provided data is composed of the desired far-field illumination specification, the light source characteristics and the reflector shape. The bulb light source and the reflector are embedded in a holder box (see Figure 4.2). All data is preprocessed and stored into GPU memory optimized structures.

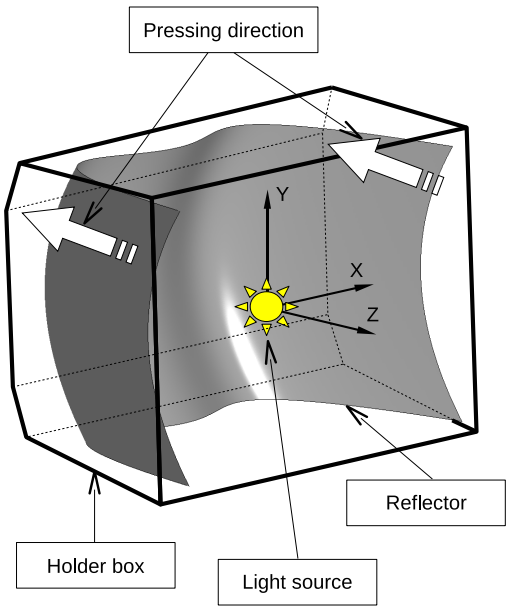


Figure 4.2: General configuration of a reflector and a light source inside the holder box. The reflector is manufactured through a press-forming process where the shape is deformed in height direction, in our representation the Z axis.

The *light source* specification provides the light source position and di-

mensions, and the near-field emittance description. From the near-field we obtain a rayset, either by loading an original ray data set or by sampling a light source to generate it, as we did in Chapter 3. Next, the rays that do not intersect with the holder bounding box are discarded. The non-discarded rays are stored into GPU memory using two RGB textures, one for ray directions and another one for ray origin positions. The energy units for the light sources are lumens. The energy for each ray is calculated by dividing the light source lumens by the rayset size.

The *desired* light distribution far-field is given by an IES specification (following IESNA or EULUMDAT standard specifications), and assumes large distances from the sources to the lighting environment, so spatial information in the emission of the light can be neglected, considering it as a point light source with a non-uniform directional distribution emittance model. The provided far-field only takes into account the vectors reflected from the desired reflector, discarding the direct rays from the light source. The set of ray directions are stored into GPU memory using one single component texture that represents a histogram over the directional space. Each texel value of this texture is the energy emitted on the direction that the texel represents, in lumens. Please, see Section 4.4 for more details.

The reflector shape generation depends strongly on the light ray tracing method, that is explained in detail in next section.

4.3 GPU light ray tracing

Ray tracing on the reflector is based on the Quadtree Relief Mapping method (QRM) [SG06], which it is a variation of relief rapping [POJ05]. QRM takes adaptive steps along the view rays in tangent space without overshooting the surface, due to the use of a quadtree on the height map. The goal is to advance a cursor position over the ray until we reach the lowest quadtree level, thereby obtaining the intersection point with the height field. For a more detailed description and comparison with other methods, see Section 2.2.4.

Reflector light calculation occurs in three steps (see Figure 4.3):

- Transformation of the reflector geometry into a hierarchical height field that defines a quadtree, in order to efficiently trace rays in the GPU.
- The set of rays is traced through the height field, searching for intersections with the reflector.
- Gather all reflected rays and create a far-field distribution that is compared with the desired far-field using a histogram comparison method,

and an error value is generated. Note that once the light rays leave the light source, further collisions with it are ignored.

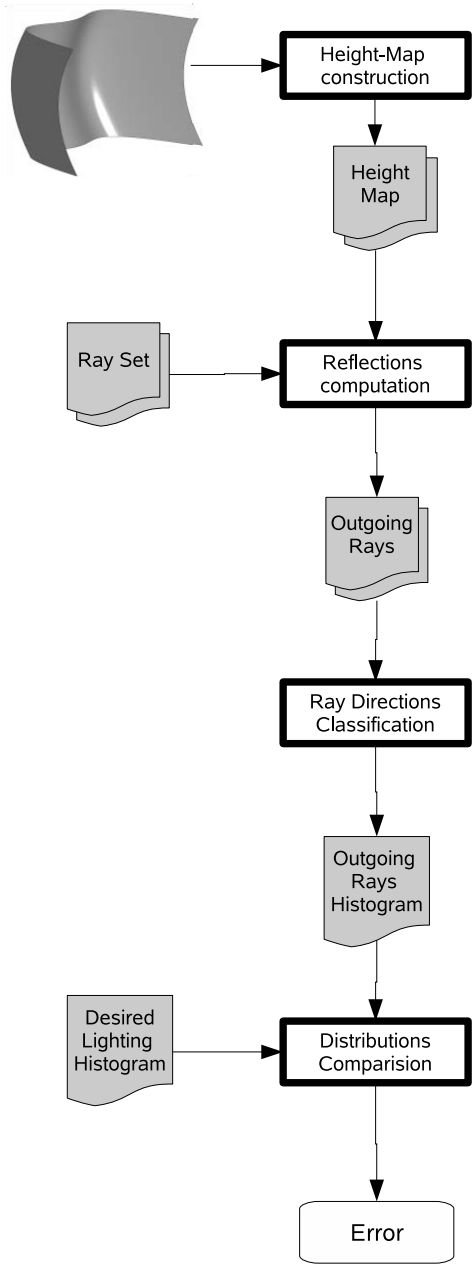


Figure 4.3: Overall scheme of reflector lighting pipeline and the used shaders and textures.

4.3.1 Quadtree construction

To store the reflector geometry to be used by QRM, a hierarchical height-field structure is constructed. This is a quadtree represented by a mip-map texture. Each quadtree node contains the maximum height of its child nodes (see Figure 4.4).

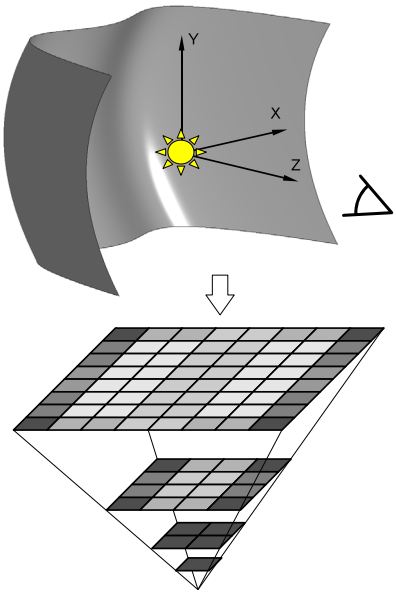


Figure 4.4: The reflector mip-map height texture is constructed from the z-buffer, using a view point where all the reflector geometry is visible. Darker texel colours mean greater heights.

The overall method does not depend on reflector geometry complexity, the only restriction being that the reflector must be able to be manufactured through a press-forming process, where the reflector shape is deformed only in the vertical direction (see Figure 4.2). More precisely, the shape must satisfy certain constructive constraints that require the shape of the reflector to be the graph of a function defined on a subset of the plane delimited by the reflector's border. That is, in our formulation, for the shape to be "build-able", it must be a function of type $z = f(x, y)$.

We calculate one orthogonal projection view from which all reflector geometry is visible. The view direction can be used as the pressing direction, so in our case, the Z axis is chosen as the press-forming vertical direction. For our experiments, just fitting the viewport to the reflector front is good enough. To avoid an excessive number of texels representing the background,

we fit the mip-map texture into a tight bounding rectangle around the reflector. Therefore the mip-map texture is non-power-of-two size, which means the number of mip-map levels will depend on the largest texture dimension. When the viewport is specified, the reflector is rendered, and then the hardware z-buffer is read, considering the Z component as heights. Then, a GPU shader creates the mip-map single component texture, where the highest map level is a texture with one texel that contains the maximum reflector height. This process is based on a reduction scheme (see Figure 4.5) and it is explained in Algorithms 4.1 and 4.2. The Algorithm 4.1 is the process for the CPU, where a mip-map texture is built from the depth buffer (from the z-buffer) contents at the initial step (see line 1 of Algorithm 4.1). Then it is started an iterative process where for each loop the GPU shader is launched, reducing the texture size by two, and increasing the mip-map level. The functions on lines 6-9 of Algorithm 4.1 set the GPU shader to run, the mip-map texture, the control parameters, and executes the GPU shader respectively. The process stops when it achieves the maximum mip-map level, that is when texture size is 1. As is shown in the Algorithm 4.2, each time the GPU shader is executed, it calculates the maximum of the four height values stored at previous mip-map levels (*SampleMipMapTexel* function returns the texel value for given mip-map texture, at given position and at given level), and stores it at mip-map texel for the current level (*SetTexel* function stores at given texture, texel coordinates and mip-map level the given value). Note that this reduction algorithm only needs one texture because the mip-map texture allows reading and writing at different levels.

Algorithm 4.1 $MipMapCPUCreation(MipMapTex, texSize)$

```
1: StoreDepthMapIntoTexture(MipMapTex)
2: level ← 1
3: while texSize > 1 do
4:    Δ ← 1/(texSize * 2)
5:    texSize ← texSize/2
6:    GPUSetShader(MipMapGPUCreation)
7:    GPUSetTexture(MipMapTex, texSize)
8:    GPUSetParameters(Δ, level)
9:    GPURun()
10:   level ← level + 1
11: end while
```

Finally, another GPU shader extracts the reflector normal vectors, and stores them in a second RGB texture. These normals will be later used to calculate the reflection vectors.

Algorithm 4.2 $MipMapGPUCreation(tex, frag, \Delta, level)$

1: $h1 \leftarrow SampleMipMapTexel(tex, frag.x - \Delta, frag.y - \Delta, level - 1)$
2: $h2 \leftarrow SampleMipMapTexel(tex, frag.x - \Delta, frag.y + \Delta, level - 1)$
3: $h3 \leftarrow SampleMipMapTexel(tex, frag.x + \Delta, frag.y - \Delta, level - 1)$
4: $h4 \leftarrow SampleMipMapTexel(tex, frag.x + \Delta, frag.y + \Delta, level - 1)$
5: $hmax \leftarrow max(h1, max(h2, max(h3, h4)))$
6: $SetTexel(tex, frag, level, hmax)$

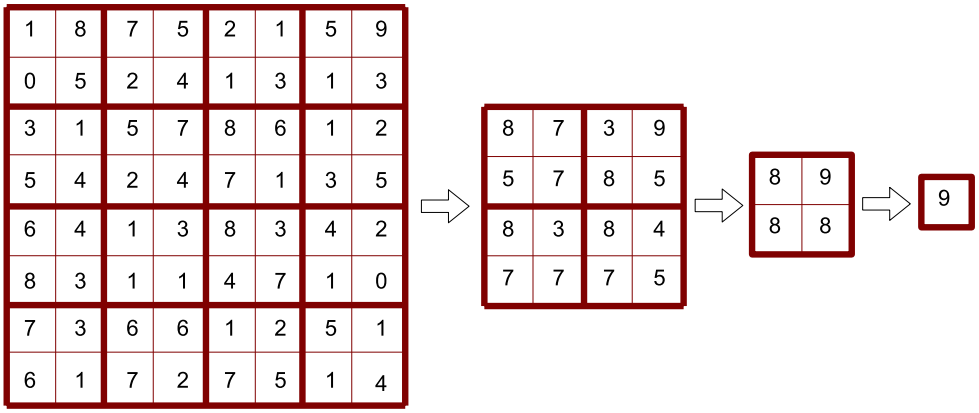


Figure 4.5: GPU reduction algorithm example. At each step, the texture size is reduced by 2, and each new texel value is calculated from the desired arithmetical operation between the four previous texels related to the current one. In this example, the algorithm calculates the maximum texel value.

4.3.2 Traversing the quadtree

Once the geometry has been stored into the hierarchical height-field structure, the set of rays is traced through it using an method based on the QRM algorithm. The method starts at the highest quadtree level, where the root node has the maximum height. The ray cursor displacement at this point is $t_{cursor_0} = 0$. To advance the cursor, the ray is intersected with the quadtree node bounds (see Fig. 4.6, left), and with the stored quadtree node height (see Fig. 4.6 right).

There are two possible node bound intersections in tangent space: t_x and t_y. From them, we use the smaller one, called t_{bound}. Also, an intersection called t_{height} is obtained by intersecting the ray with the height value stored in the node. If t_{bound} is greater than t_{height}, it means that the ray intersects with the current quadtree cell. So, the quadtree level is decreased, and the process starts again with one of the four child nodes. In this case, the cursor

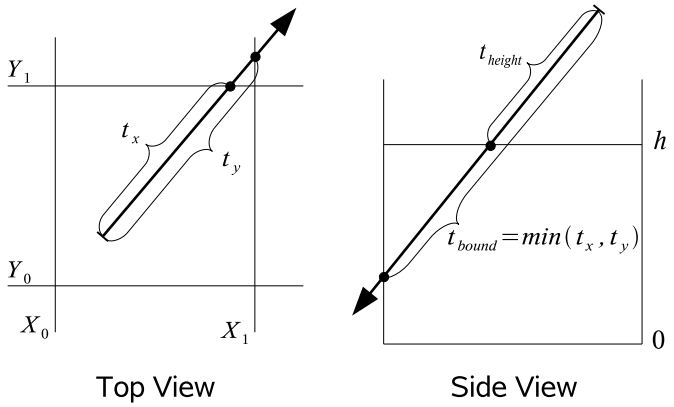


Figure 4.6: Two ray steps are calculated for a quadtree node. At the left, t_{bound} is the minimum displacement to quadtree node bounds t_x and t_y. At the right, t_{height} is the displacement to the stored node height h. The final selected step is the minimum between both.

does not advance, so $t_{cursor_{i+1}} = t_{cursor_i}$. Otherwise, the cursor advances to the cell bound, $t_{cursor_{i+1}} = t_{bound}$, and the process starts again with the neighboring cell. This process stops when the minimum quadtree level is reached, or when the cursor position is out of the texture bounds. In Figure 4.7 there is an example of the execution of this algorithm.

In the QRM algorithm, the first cursor position is found by intersecting the view ray with the geometry bounding box. In our case, the first cursor position is the light ray origin (see Figure 4.7). This means that one more step is processed in comparison with QRM, because we need to intersect the root quadtree node in an initial step. However, we avoid the ray-bounding box intersection calculation that QRM performs.

On the other hand, QRM only processes rays going down the quadtree hierarchy, being unable to process the rays going up. This is the case when the light source is inside the reflector, or when more than one ray bounce inside the reflector are considered. To solve it, We propose an intersection search algorithm going up the quadtree hierarchy. The pseudo-code for the new algorithm, called RQRM, is presented in Algorithm 4.3.

The original algorithm assumes that the cursor always advances in the opposite direction to the height map direction. Otherwise, QRM discards the ray because it does not intersect with the surface. If the reflected ray separates from the surface, going up, we start the algorithm from the highest quadtree level using the new ray, composed of the current intersection point and reflection direction. A small offset is applied as initial cursor displace-

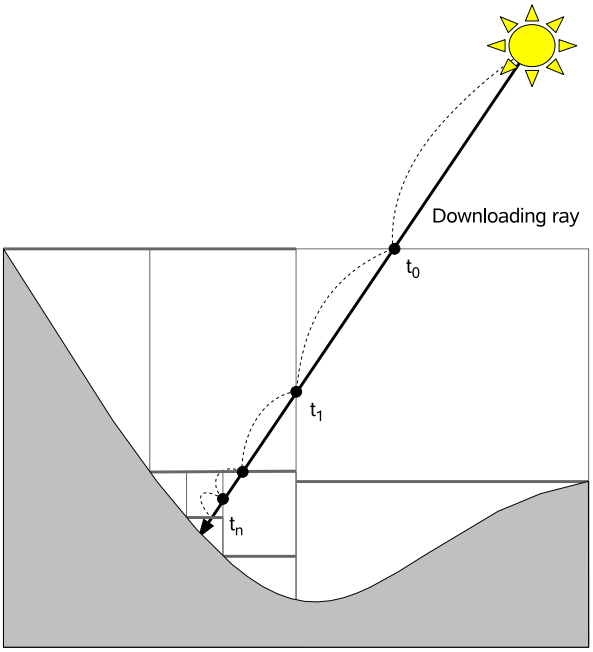


Figure 4.7: Intersection search going down the quadtree hierarchy.

Algorithm 4.3 $RQRM(texCoord)$

```
1:  RQRMInitialization(texCoord)
2:  while level ≤ log(max(reliefMapSize_xy)) do
3:    RQRMCalculateTangentSpaceBounds
4:    RQRMStep
5:    if OutOfLimits(cursor) then
6:      if FirstBounce then
7:        return DISCARDED
8:      else
9:        return FINISHED
10:     end if
11:   end if
12: end while
13: finalPos ← reflectorTex[cursor]
14: finalNormal ← reflectorNormTex[cursor]
15: reflectRay ← reflect(rayDir, finalNormal)
16: return(finalPos, reflectRay)
```

ment to avoid self-intersections, thus $t_{cursor_0} = \delta$ (see lines 4-7 of Algorithm 4.4). Then, we go down through the quadtree until $t_{cursor_i} > t_{height}$ (see Al-

gorithm 4.5 for tangent space bound calculations), which means the height of the current cursor position is above the current node height. Now, we are sure there are not any nodes under the current one that have a height that might intersect with the ray. Hence, we jump to the neighbor node, so $t_{cursor_{i+1}} = t_{bound}$, and we update the quadtree level increasing it by one. If $t_{cursor_i} < t_{height}$ then there is not any possible intersection under current level. Thus, we decrease the current quadtree level, and do not update t_{cursor_i} (see lines 8-14 of Algorithm 4.6). The process stops when the intersection is reached, or when the cursor position falls out of the texture bounds (see lines 6-10 of Algorithm 4.3). In the second case, it is a reflected ray with no more bounces, and it is stored as an outgoing ray. In Figure 4.8 there is an example of this algorithm.

Algorithm 4.4 $RQRMInitialization(texCoord)$

1: $rayPos \leftarrow rayPosTex[texCoord]$
2: $rayDir \leftarrow rayDirTex[texCoord]$
3: $cursor \leftarrow ReflectorMapProjection(rayPos)$
4: **if** $FirstBounce$ **then**
5: $\quad tcursor \leftarrow 0$
6: **else**
7: $\quad tcursor \leftarrow \delta$
8: **end if**
9: $cursor \leftarrow cursor + rayDir \cdot tcursor$
10: $startPoint \leftarrow cursor$
11: $quadrant \leftarrow (sign(rayDir) + 1)\ div\ 2$
12: $level \leftarrow 0$

Algorithm 4.5 $RQRMCalculateTangentSpaceBounds$

1: $bound \leftarrow \lfloor (cursor \cdot 2^{level}) + quadrant \rfloor$
2: $tbound \leftarrow \frac{\frac{bound}{2^{level}} - startPoint}{rayDir_{xy}}$
3: $tmin \leftarrow min(tbound_x, tbound_y)$
4: $height \leftarrow reliefMap[cursor, level]$
5: $heightNorm \leftarrow (height - rayPos_z) \cdot \alpha$
6: $t \leftarrow \frac{heightNorm}{rayDir_z}$

The algorithm is implemented in a GPU fragment shader. The rayset data is provided by the previously stored rayset textures. The textures are mapped into a quad, so each ray corresponds to a fragment (see Section 2.2.4). Each fragment program runs an iterative process that ends with an

Algorithm 4.6 *RQRMStep*

1: **if** $rayDir_z \leq 0$ **then**
2: $tcursor \leftarrow max(tcursor, min(t, tmin + \delta))$
3: $cursor \leftarrow startPoint + (rayDir_{xy} \cdot tcursor)$
4: **if** $t < (tmin + \delta)$ **then**
5: $level \leftarrow level + 1$
6: **end if**
7: **else**
8: **if** $t > tcursor$ **then**
9: $level \leftarrow level + 1$
10: **else**
11: $tcursor \leftarrow tmin + \delta$
12: $cursor \leftarrow startPoint + (rayDir_{xy} \cdot tcursor)$
13: $level \leftarrow level - 1$
14: **end if**
15: **end if**
16: **return** *level*

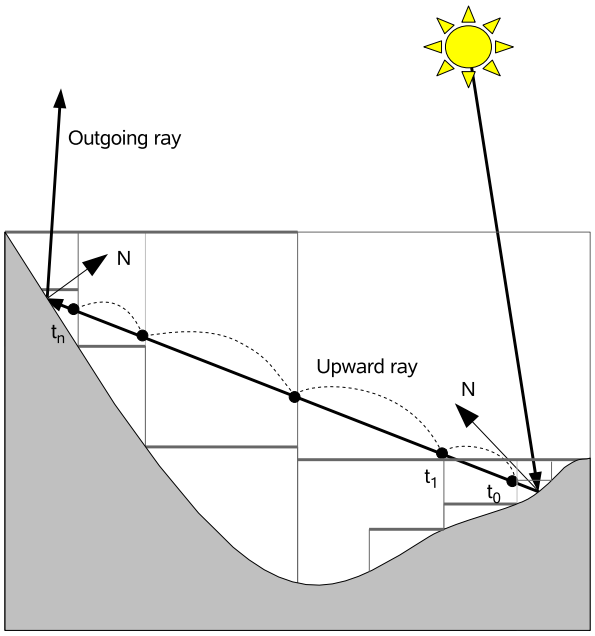


Figure 4.8: Intersection search going up the quadtree hierarchy.

intersection point and a reflection vector. These values are stored in two output RGB textures, one for the intersection positions, and the another one for

the reflected directions and bounce counters (see next section). This shader is executed as many times as the maximum number of allowed bounces. The resulting textures are used as input textures for the next execution, thus a GPU ping-pong approach is used (see Figure 4.9). In addition, the shaders contain the algorithms to check the ray intersections with the light source: The light source volume is defined as a bounding sphere or cylinder, so we can analytically check the intersection.

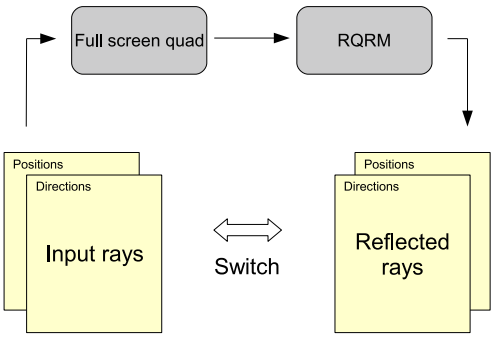


Figure 4.9: GPU ping-pong method for the RQRM algorithm. Two pair of textures (positions and directions) are used as input data and output data respectively. When the reflected rays have been calculated, both pairs are switched, allowing to use the previous result as new input data.

4.4 Comparison with the desired distribution

When light tracing is done and all rays have left the reflector, they are gathered generating a new rayset. At this step we compare the obtained light distribution with the desired one. The luminaire light distribution is converted to a far-field, just discarding the positional component of each rayset particle. Both distributions are converted to a histogram structure to be compared in the same domain (see Figure 4.10).

The histograms, one for each distribution, are a regular grid used to classify ray directions. Each grid cell represents a pair of azimuth and altitude directions in a horizontal coordinate system, and contains the gathered energy in this direction. The total azimuth and altitude ranges are $[-\pi...\pi]$ and $[\pi/2...-\pi/2]$ respectively. The grid size depends on the specified far-field directional space discretization. We use two textures to store both grids, with the same resolution as the sizes of the grids, so that each texel represents a grid cell.

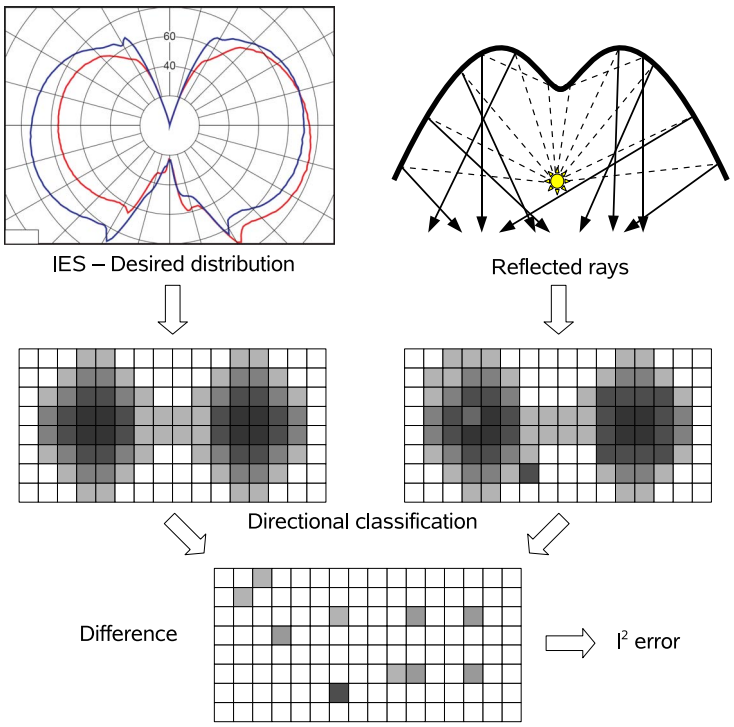


Figure 4.10: Both the desired distribution and the reflected rays are classified into histograms. Next, the histograms are compared using the l^2 metric.

We classify the reflected directions by calculating a histogram, where each grid cell represents an interval. The algorithm, based on [SH07], has two steps (see Figure 4.11). First, after the last reflection step, the results are stored into a vertex buffer object. Next, this vertex buffer is rendered, and a vertex shader classifies the directions by calculating the fragment coordinates for each reflected direction. Then, the fragment shader gathers the directions and the emitted energy using counters and the hardware additive blending. To calculate the energy for each ray we define

$$rayEnergy_i = e_i \cdot RF^{bounces_i} \quad [lm] \qquad RF \in [0..1]$$

where e_i is the initial energy for ray i (see Section 4.2), $bounces_i$ is the number of ray bounces inside the reflector, and RF is the reflector reflectance factor, as an attenuation factor.

We use the same algorithm to classify the desired distribution. In this case, we do not have to use a counter because each far-field directional component has its respective emitted energy.

The comparison between both textures is done with a shader that calcu-

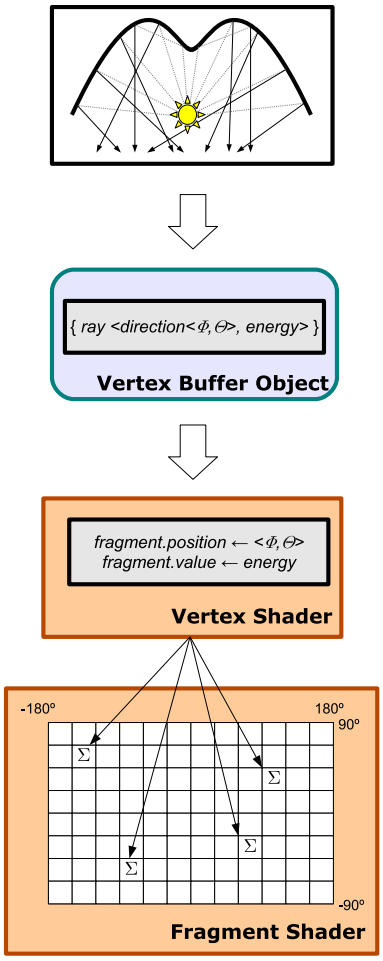


Figure 4.11: Classification algorithm for the reflected rays.

lates, for each fragment, the l^2 error metric:

$$D_{l^2}(a,b) = \sqrt{\sum_{i}^{N} (a_i - b_i)^2}$$

where a_i and b_i are the values of the texels indexed by i on textures a and b. In addition, a reduction shader is used to calculate the summation part of the formula (see Figure 4.5; in this case the arithmetic operation is a sum).

4.5 Results

To test the algorithm we use a standard and simple optimization method to obtain a reflector shape that produces a light distribution close to the desired one. The algorithm is an iterative process where each parameter is increased inside a given range by its step value [PPV04]. For each optimization step, a new reflector shape is obtained, and the outgoing light distribution is compared with the desired one. If the difference value is below a user-specified threshold, the process stops. If no reflector produces a light distribution close enough to the objective, the best one is chosen. Figure 4.12 shows the overall scheme of the optimization algorithm.

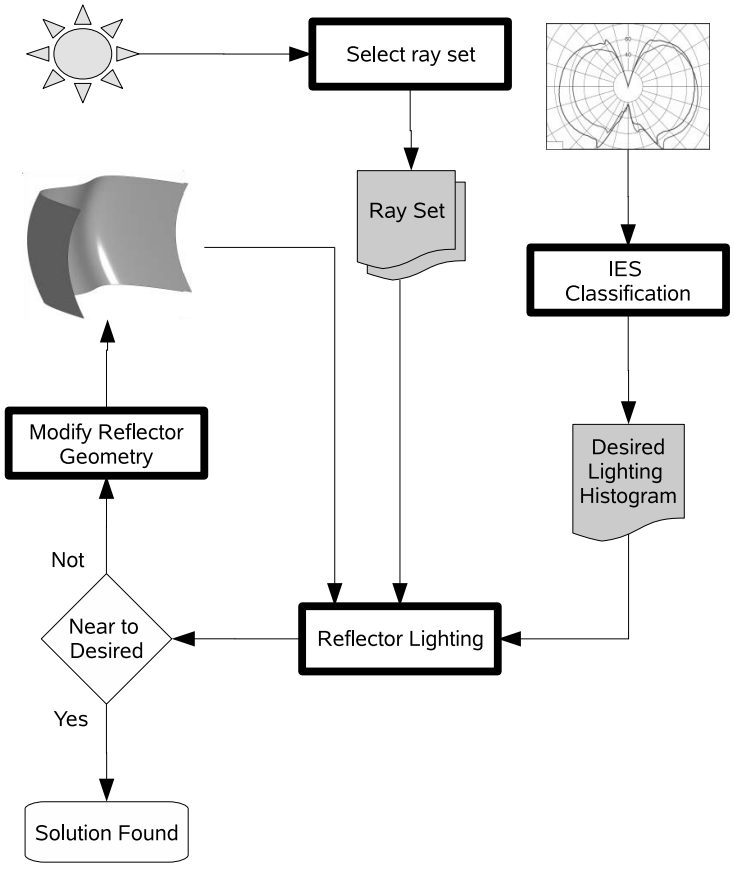


Figure 4.12: Overall scheme of the optimization algorithm.

The mip-map height texture must be regenerated at each iteration, due to reflector geometry changes. Hence, for each iteration we have to recalculate the outgoing light distribution. However we do not have to recalculate the rayset for each reflector, so the initial ray intersection step on the reflector

bounding box guarantees that the rayset is valid for any reflector inside this box (see Section 4.2).

We have tested our method with three cases. The first one, called *Model A*, uses a cylindrical light source with a cosine emittance along its surface, except for the caps that do not emit light. The cylinder dimensions are 4.1mm in length and a 0.65 mm radius. It is placed at (0,0,0), inside a holder bounding box located between (-30, -20, -20) and (30, 20, 0), also in mm. The second case, called *Model B*, uses a spherical light source with a cosine emittance. It has a radius of 0.5mm, and it is placed at (5, -5, -5) inside a holder bounding box located between (0, -10, -6) and (10, 0, 0). The third one, called *Model C*, uses a spherical light source with a cosine emittance. It has a radius of 1mm, and it is placed at (5, 5, 0) inside a holder bounding box located between (0, 0, -6) and (10, 10, 0). The cross sections of the three cases and light source relative positions are shown in Figure 4.13. For *Models A* and *C*, the light sources emit 10 million rays, and 5 million rays for *Model B*. All of them have an overall energy of 1100 lumens. Also, for all cases, the mip-map height texture resolution is 1200×800, and a quadtree is created with 9 subdivision levels.

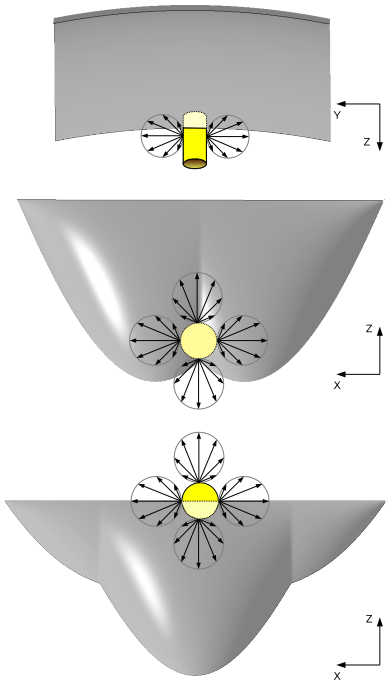


Figure 4.13: Cross section views of reflectors and their associated light sources used to test our method.

The optimization results for each case are shown in Figures 4.14, 4.15 and 4.16. The desired and obtained reflectors are shown, with the respective far-field distributions and difference images. In the figures, both far-field and difference images are represented by false-colour histograms. These histograms are defined as far-field textures, thus the columns of the texture grid correspond to horizontal angles, and the rows correspond to vertical angles. The directional space resolution is 1800×900 for horizontal and vertical angles respectively. Therefore, each histogram cell represents an angle of 0.2×0.2 degrees. The colour scale represents the amount of energy for each histogram cell.

Table 4.1 provides a summary of the data for the overall inverse reflector search process for each model. The number of effective rays is the number of non-discarded rays from the initial rayset. For *Model B* there are not any discarded rays because the light source is inside the reflector bounding box, and all the rays intersect the height map. The time needed to compute the reflector lighting depends on the number of effective rays and the number of maximum allowed bounces. All models have a similar number of effective rays, but *Model A* has the lower computation time because only one bounce is specified. The optimization time depends on the reflector lighting time and the number of tested reflectors, and the number of tested reflectors depends in turn on the number of optimizable parameters and on the range and offsets applied in the optimization procedure.

Model	A	B	C
Effective rays	$7.38x10^6$	$5x10^6$	$6.05x10^6$
Max. bounces	1	5	6
Reflector lighting mean time (sec.)	1.3	3.2	2.7
Best l^2	0.599456	0.975587	0.245821
Tested reflectors	1728	2401	6561
Optimized parameters	3	4	4
Optimization time (hours)	0.63	2.2	4.9

Table 4.1: Results for our three configurations: From left to right, we find the number of traced rays, maximum number of bounces inside the reflector, mean time of reflector lighting computation, total time of optimization, number of tested reflectors, number of optimized parameters and resulting error.

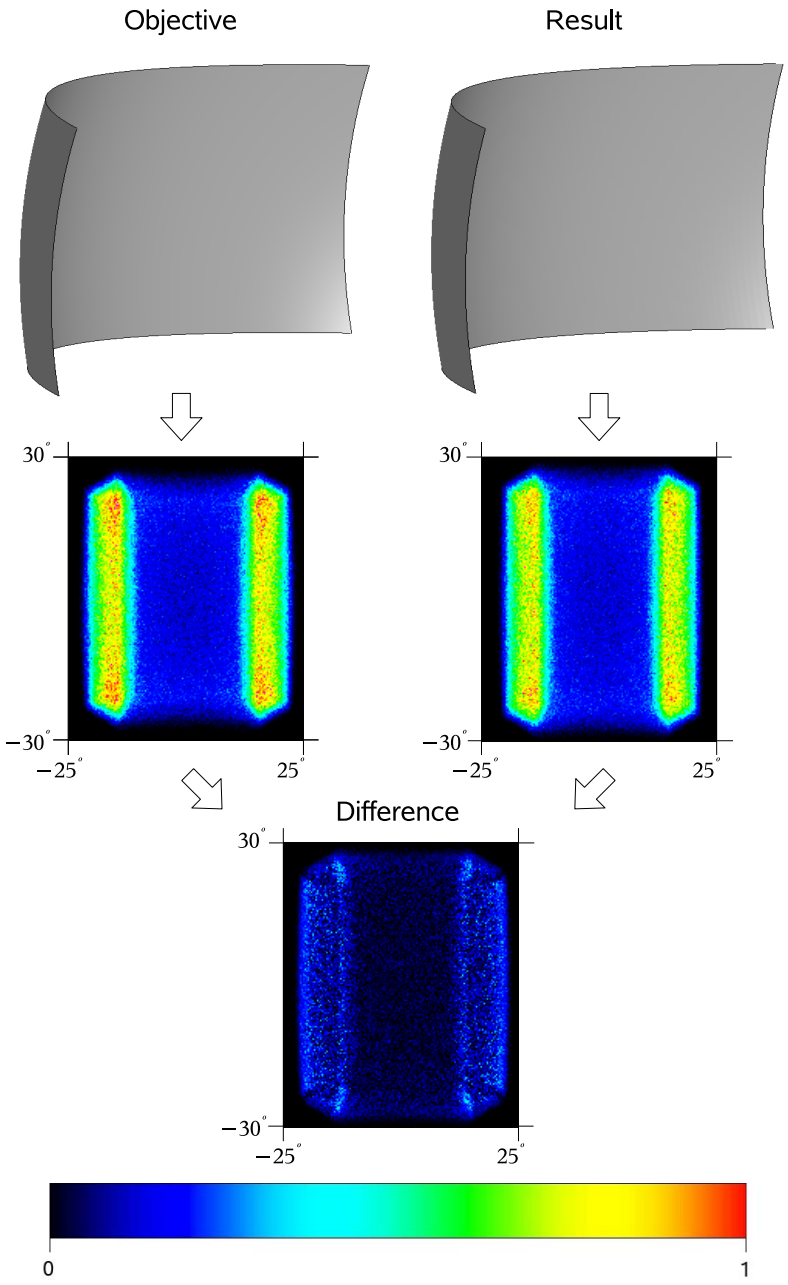


Figure 4.14: Results for our *Model A*. At the top, the desired and obtained reflectors. In the middle, the desired and obtained far-field histograms in false-colour, indicating the respective angle domains. At the bottom, the histogram difference between both

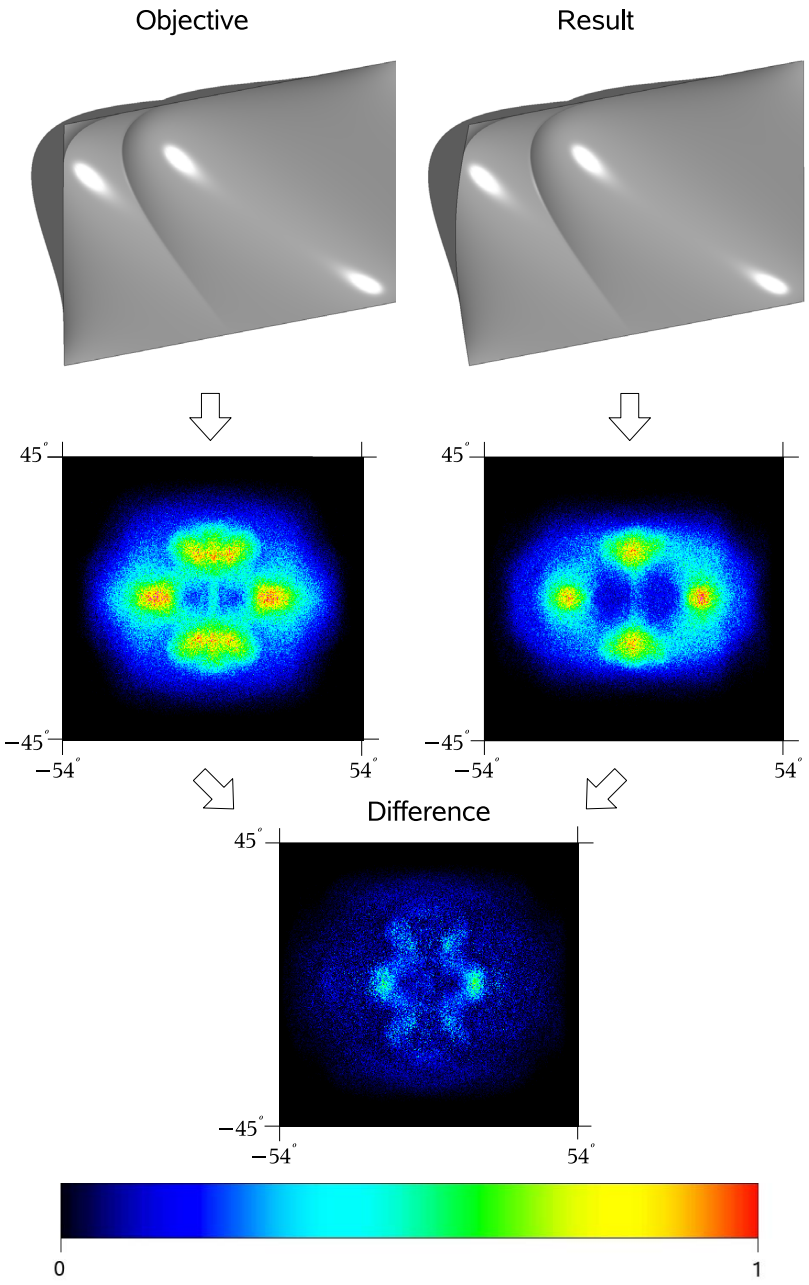


Figure 4.15: Results for our *Model B*. At the top, the desired and obtained reflectors. In the middle, the desired and obtained far-field histograms in false-colour, indicating the respective angle domains. At the bottom, the histogram difference between both.

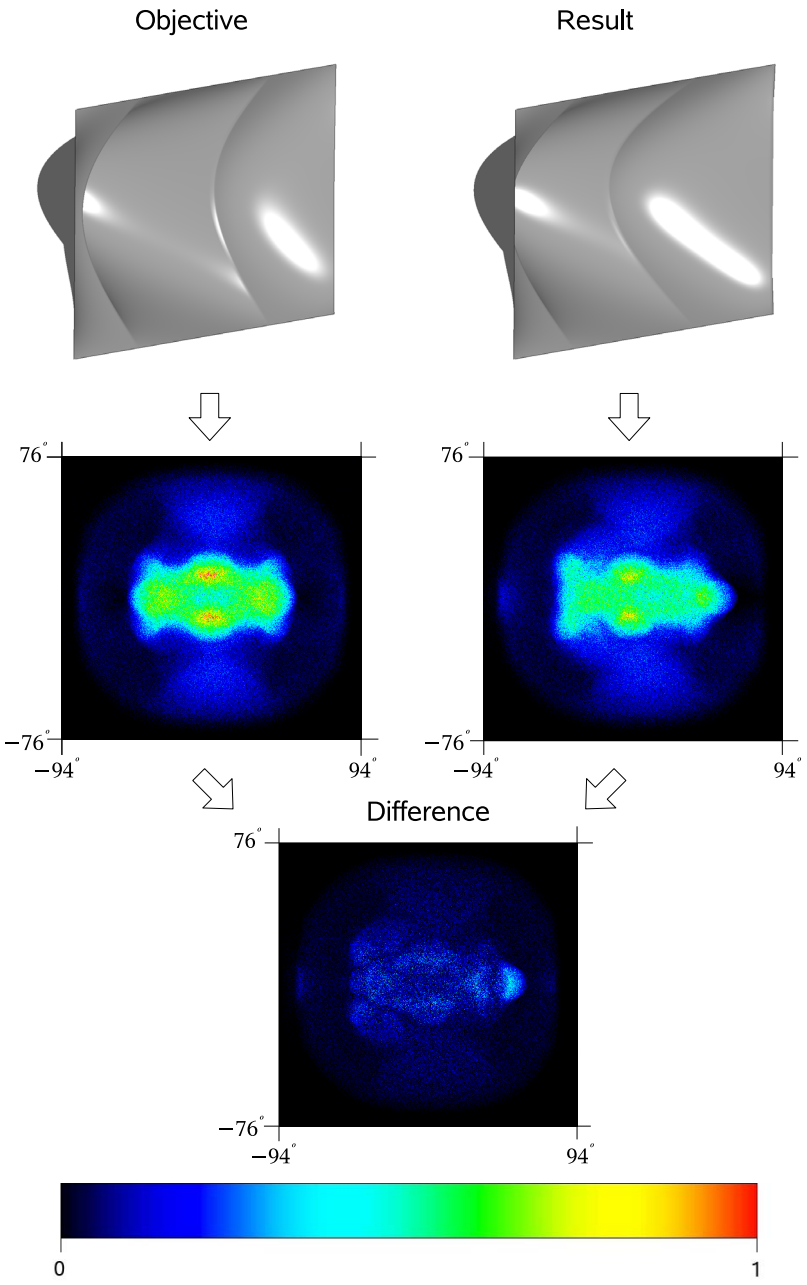


Figure 4.16: Results for our *Model C*. At the top, the desired and obtained reflectors. In the middle, the desired and obtained far-field histograms in false-colour, indicating the respective angle domains. At the bottom, the histogram difference between both.

Table 4.2 summarizes the isolated times for each reflector lighting step. The height map creation times are similar because all the models use the same mip-map height texture resolution. The intersection search time depends on the number of traced rays, on the maximum number of allowed bounces and on the height map levels. This results have been obtained executing the algorithm on a NVIDIA® GeForce™ 8800 GTX graphic card.

Model	Heigh map construction	Intersection search	Error calculation
A	56	976	277
B	34	2963	278
C	86	2406	263

Table 4.2: Mean times (in milliseconds) broken down into the three main algorithm sections.

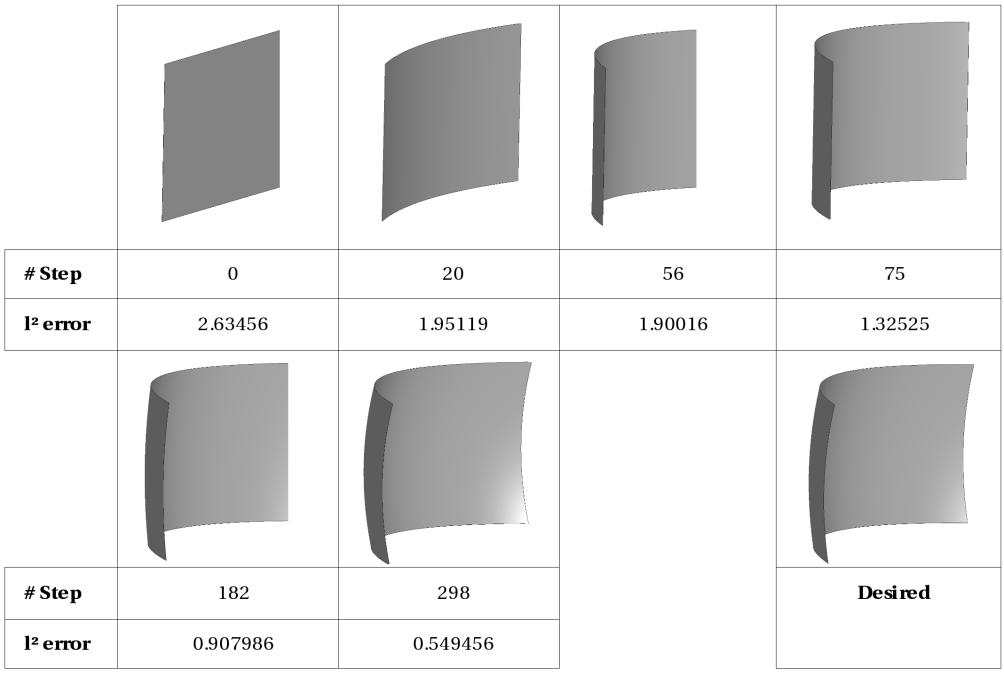
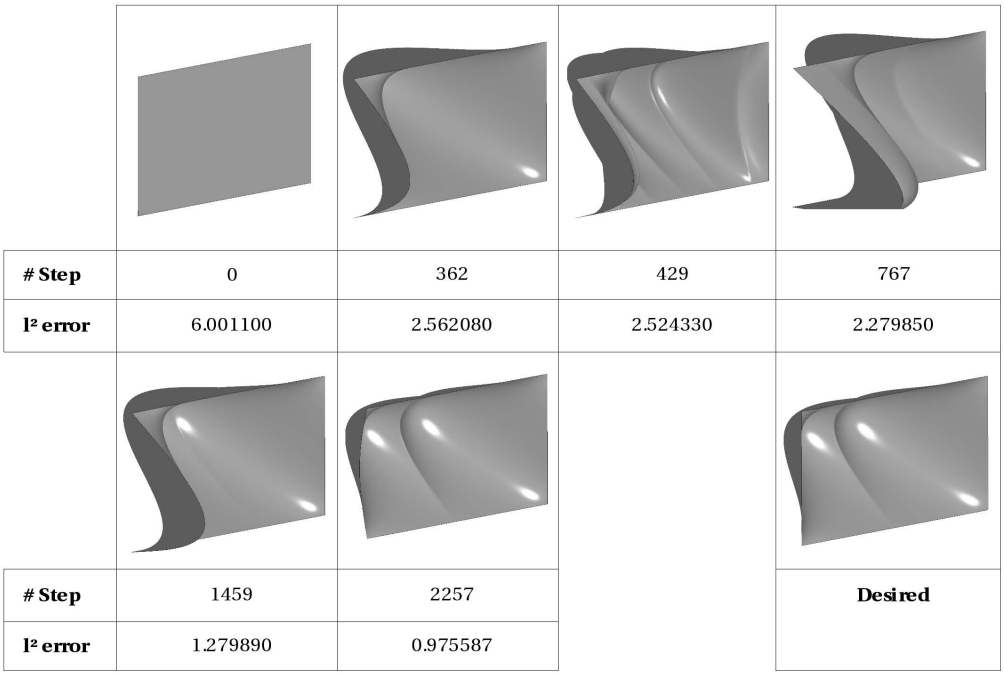


Figure 4.17: Reflector searching progress for model A, from an initial shape (left), to the desired one (right). Below each reflector, there are the current number of steps in the optimization process and the l^2 error



# Step	0	362	429	767
l^2 error	6.001100	2.562080	2.524330	2.279850

# Step	1459	2257		Desired
l^2 error	1.279890	0.975587		

Figure 4.18: Reflector searching progress for model B, from an initial shape (left), to the desired one (right). Below each reflector, there are the current number of steps in the optimization process and the l^2 error

Figures 4.17, 4.18 and 4.19 show the progress in the optimization process for the three tested models. The results are very similar between the three different models, because they have the same height map texture sizes (thus, the same number of quadtree levels), and the number of traced rays is also similar. The GPU parallel processing involves a linear computational cost on the rayset size. Therefore, the most important factor in the intersection search procedure is the maximum number of allowed bounces. Finally, the error calculation has similar times for all cases, since the outgoing textures have the same size.

4.5.1 Method calibration

To test the accuracy of method, we performed a preprocessing step, where the lighting simulation was calculated several times using the desired reflector and combining different rayset sizes. For each test the light source was resampled. The Table 4.3 presents the results for the *Model A* example (see Section 4.5), showing the mean and variance of the l^2 errors from the differ-

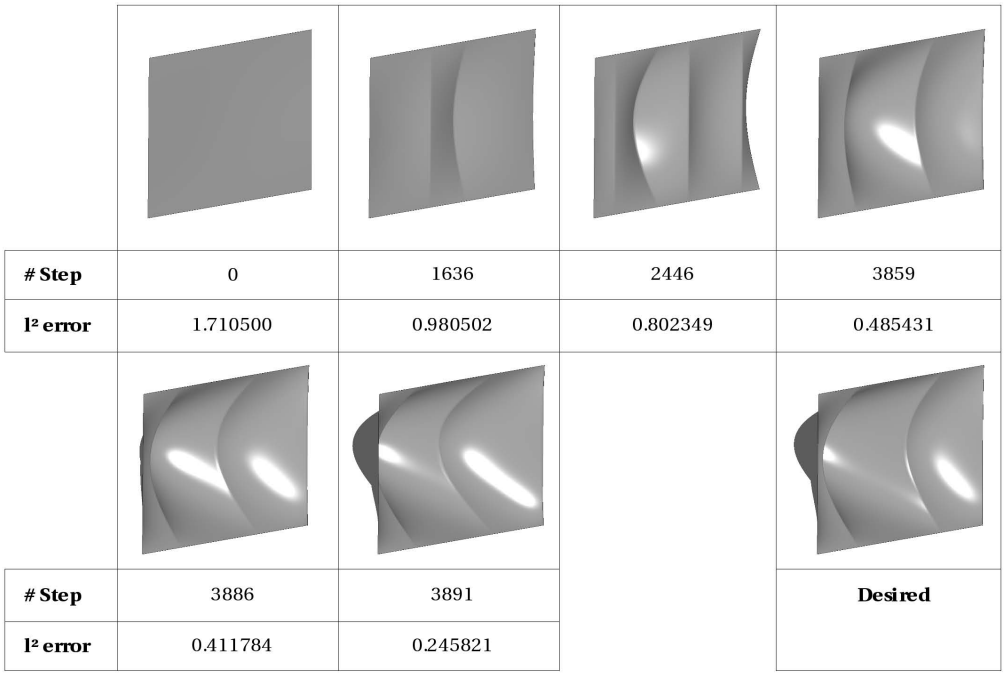


Figure 4.19: Reflector searching progress for model C, from an initial shape (left), to the desired one (right). Below each reflector, there are the current number of steps in the optimization process and the l^2 error

ence between each result and the desired light distribution. For each rayset size, 100 lighting simulations have been calculated.

Rays	**Mean l^2 Error**	**Variance l^2 Error**
1000	271.68	15078.60
10000	27.97	20.97
100000	2.91	0.04
1000000	0.38	6×10^{-4}
10000000	0.13	3×10^{-7}

Table 4.3: Results of several lighting simulations on the *Model A* using different rayset sizes.

We observe that the variance error decreases when the rayset increases. On the rayset of 1 million rays, the mean error is quite good, so we can use this rayset to perform the optimizations. The last row shows the calibration

values to consider the quality of our method.

Moreover, we are interested in knowing the minimum optimization parameter step required to consider that two consecutive measures are different. For this, we use the semivariogram [Ole99], a statistical measure that assesses the average decrease in similarity between two random variables as the distance between the variables increases. The measure defines a lag called range, at which the semivariogram reaches a constant value. We can consider that two measures separated by a distance larger than the range are stochastically independent, so the range is equivalent to the notion of influence of an observation. That is, if we want to get significant measurements without being influenced by statistical noise problems, we can not take measurements that are closer than the range. From this, we can find a lower bound for the step size in the optimization process.

The semivariogram is defined as follows: Given two locations x and $x+h$ inside the domain of a random function Z, the semivariogram is:

$$\gamma(h) = \frac{1}{2n(h)} \sum_{i=1}^{n(h)} [Z(x_i + h) - Z(x_i)]^2$$

where $n(h)$ is the number of pairs of measurements at a distance h apart. Figure 4.20 presents the semivariogram for one of the parameters of *Model A*. From this graph we can see that the range of the semivariogram is about 0.1 units in the l^2 metric. So, we have to use values larger than 0.1 units as the step size for this parameter in the optimization process. We computed this lower bound for every degree of freedom included in the optimization process.

4.6 Full GPU ray tracing engine: OptiX™

Recently, a new full GPU ray tracing engine has been presented, the NVIDIA® OptiX™ system [NVI09]. This is a programmable pipeline to achieve fast ray tracing results on the GPU. OptiX implements some optimized geometry structures for ray tracing on the GPU, such as KD-trees or BVHs, and incorporates algorithms for traversing these structures in a fast way.

4.6.1 Implementation

We have tested OptiX on our method by replacing the heightmap construction and the RQRM ray tracing algorithm.

The OptiX implementation has been focused on two parts. First, an appropriate acceleration structure has been chosen. The most important

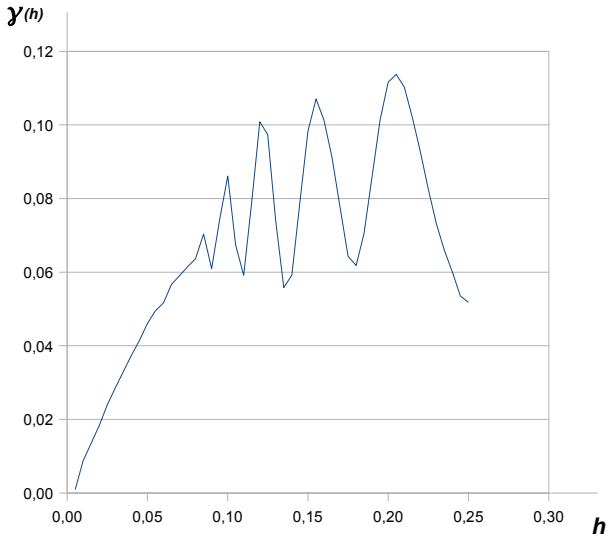


Figure 4.20: Semivariogram when changing one parameter of *Model A* using 10^6 rays.

aspect is the structure construction time, as it has to be reconstructed for each new reflector. The reflector geometry is composed by a triangle mesh with indexed vertices. Our experiments showed that the BVH structure is slightly more efficient than the KD-tree. Second, it has been implemented a simple ray tracer that traces light particles through the BVH. The light rays are initially stored into a texture, one for each texel. Then, each texel is used as an entry point mapped to a thread. For each thread, OptiX traces the ray through the BVH until a final node is found. Then, the intersected triangle is returned with the vertex indices. The triangle data is used to perform a ray-triangle intersection test that has been implemented inside the OptiX framework. Once the intersection is found, the reflected ray is calculated using the interpolated vertex normals and a reflectance factor. The later only affects at the ray energy as an attenuation factor, the same way as described in the FIRD system (see Section 4.4). Note that we don't use any BRDF or transparency factors, so the reflector material is purely specular. Then, the reflected ray is traced again searching for new intersections. The thread stops when the ray has not intersected any more geometry, and the current ray is stored as an outgoing ray. In contrast of CUDA applications, OptiX manages automaticly the threads management. Thus, we are not able to specify the thread block size. OptiX chooses automaticly the best configuration from the current entry data.

Finally, we use the same method as used in FIRD (see Section 4.4) to compare the final light distribution with the desired one.

4.6.2 Results

To test the OptiX system we need a high performance graphics hardware device. We have tried it using a NVIDIA Quadro™ FX 4800 with 1.5GB of internal memory.

We have repeated the same tests used for FIRD, but in this case, only the mean reflector computation time has been considered. Because the FIRD results were previously computed with a different GPU, we have repeated them using the mentioned Quadro GPU to compare them with the OptiX results. This comparison is shown in Table 4.4.

Method	**Model**	**Method stages**			**Total**
		PG	**RT**	**CL**	
FIRD	A	39	95	13	147
	B	36	200	6	242
	C	56	260	20	336
OptiX	A	32	69	55	156
	B	33	477	69	579
	C	31	190	58	279

Table 4.4: Comparison between FIRD and OptiX reflector computation times (in milliseconds) for each model (see Figure 4.13) and for each method stage. The stages are: Preprocessing the reflector geometry (PG), ray tracing (RT) and comparing final light distribution with desired one (CL). For all models we have traced 10^6 rays.

The preprocessing of the reflector geometry (PG), that is the height map construction for FIRD and the BVH construction for OptiX, have similar consumption times.

There are noticeable differences between both in the ray tracing stage (RT). OptiX is more efficient than FIRD when the number of ray bounces inside the reflector is low. This is the case of models A and C (see Figure 4.13). For the model B, the light source is placed very close to the reflector, so there are many light rays that bounce many times inside the reflector. In that case, FIRD is more efficient. On the other hand, as can be seen in the results for FIRD (see Section 4.5), the model C computes more bounces than model B. The reason for that is due to the OptiX thread management,

that only allows to process a fixed number of threads, as happens in CUDA. Considering that each thread processes a light ray, the number of light rays able to be processed in parallel is smaller than the whole rayset to trace. Therefore, the rest of the threads have to wait until the initial entries are returned. If there are a lot of threads computing many bounces, the rest of the threads wait longer to be processed. Our experiments show that the model C computes more ray bounces inside the reflector, but for less rays. In contrast, the model B computes less bounces for more rays. In comparison, the FIRD method computes the whole rayset, stored into a texture, for each bounce, discarding those rays that have already finished, and finishes when there are no more bounces to calculate. In this way, FIRD only has to wait for the last ray bounce, resulting in a faster evaluation for these cases.

Finally, the comparison of final light distributions (CL) is slightly better using FIRD. In the FIRD method, the data structure in the whole GPU algorithm is always the same. But in OptiX, the comparison is computed using a GPU shader, outside of the OptiX system. OptiX is a GPU ray tracing engine, but it is not a general purpose GPU shader framework.

4.7 Discussion

As shown in the results section (Section 4.5), we cannot obtain the desired reflector with zero error. This is because the optimization algorithm tests different parameterized reflectors by changing the parameter values in a constant step size and in a floating point space. On the other hand, we can improve the results by optimizing in very small steps, thereby guaranteeing convergence to a better solution, but this would strongly affect the processing times. Also, the semivariogram gives us a lower bound to the step size for each parameter in the optimization.

The most time consuming part of FIRD algorithm is the intersection search algorithm (RQRM). If we use a very refined height map, we will need more time to traverse the ray through the quadtree. If we wanted to manage very complex reflector shapes, we would need height maps with high resolutions. Therefore, we should find a compromise between time costs and quality of results.

The presented method is fast and works well for many reflectors, specially under the constraint of be constructable by a pressing reflector procedure. This gives us the chance of representing the reflector geometry on a texture (height map), storing only the inner reflector geometry. This way, FIRD is an image-space based technique, creating a dependence between the texture resolution and the geometry, since each texel represents a small amount of

geometry. Image-space based GPU ray tracing methods have the common problem of loosing geometric information due this texture-geometry dependence. This is a problem if we have a reflector with some parts parallel or near parallel to the pressing direction, such is a hemispherical reflector shape. The problem can be improved by interpolating the geometry between texels, or supersampling at problematic reflector zones, but neither solves the problem completely (see Figure 4.21). By the other hand, these kind of reflectors are not efficient, since they can produce too many light ray bounces inside the reflector, loosing energy in each bounce and increasing the reflector temperature.

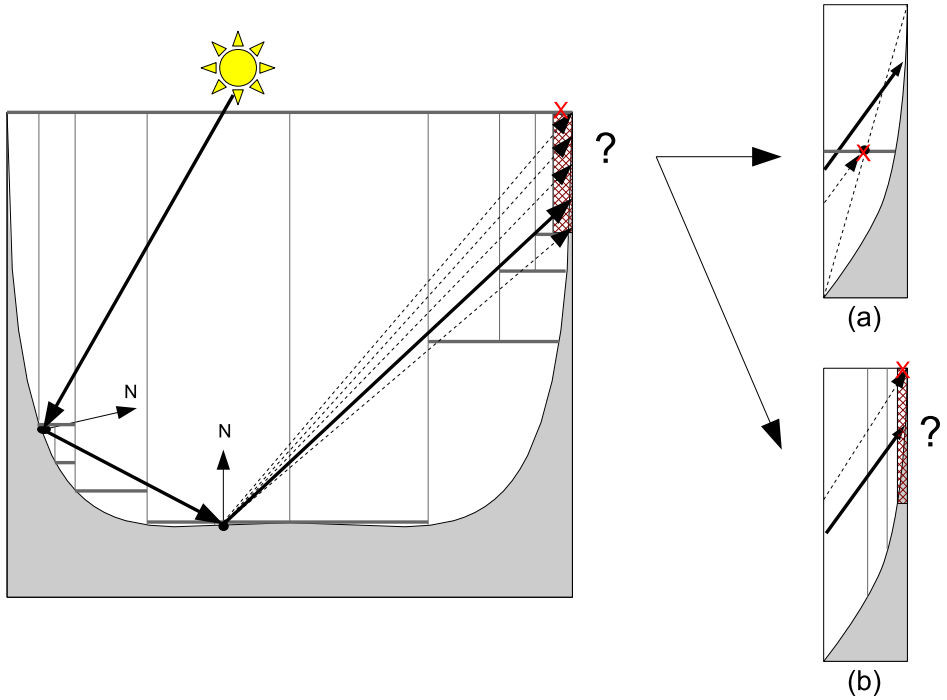


Figure 4.21: FIRD intersection problem when there are geometry almost parallel to the vertical direction. Height interpolation (a) or supersampling (b) can improve the intersection error, but cannot solve it completely

Finally, we have studied the OptiX system as a full GPU ray tracing engine. This system improves some of the mentioned FIRD problems, as it is not image-based. Moreover, our experiments showed that for non-problematic reflectors, the quality differences between the ray tracing methods are small. In Table 4.5 is shown a comparison between both methods. We have used the three desired reflector shapes, with a reference value computed

with raysets of 5 million rays. Then, the l^2 errors from the light distribution comparisons have been compared using the same reflector shapes and raysets of 3 million rays.

Model	**RQRM**	**OptiX**	**Difference (abs.)**
A	0.412	0.233	0.179
B	0.23	0.181	0.049
C	0.092	0.139	0.047

Table 4.5: Comparison between RQRM and OptiX ray tracers. The values, in l^2, are the difference between the light distributions of a reflector shape using a rayset of 5 million rays, and the light distribution of same reflector shapes using a rayset of 3 millon rays.

Note that we use raysets of 5 million rays because OptiX cannot handle bigger raysets. This limitation depends on the memory integrated in the graphical card and the memory used to construct the BVH.

Finally, our experiments show that the OptiX solution is faster than FIRD if there are a low number of light ray bounces inside the reflector, but slower on the opposite case. Furthermore, we need a high performance graphic card to use it.

Chapter 5

Optimization

As is exposed in previous chapters, the inverse reflector design procedure is based in a trial and error process where a set of reflectors are checked, until the reflector that produces the light distribution closest to desired one is found. This is a classic optimization approach. This particular case is a systematic search to reach the right reflector. The optimization tries to minimize (or maximize) a function, that is in our case the difference between a reflector light distribution and the desired one. However, this solution implies high computational costs due to the size of the search space.

To improve the search, other optimization methods can be considered. Local optimization methods are fast, but converge only if the solution is near, making them suitable only for local solutions, not for global ones. Global optimization methods are used to search the solution in the whole domain, but usually these methods are slower than the local ones, or they do not fit well on our problem. In Section 2.3 there is a description of these kinds of methods.

Here we propose a new specific optimization algorithm that allows to reach the global minimum in a fast way. A tree is constructed on the fly, where each node is the evaluation of a reflector shape. We use one of the GPU-based raytracing algorithms described in Chapter 4 to calculate the illumination of each reflector and to evaluate the difference function in a fast way. Tree branches are constructed from the stochastically selected nodes following heuristic rules. The goal is to successively refine the reflector shape in regions where it produces the best results and where it better approaches the global minimum, avoiding to fall in local minima. When we are close to the solution, a classic local search optimization algorithm is started to faster reach the minimum.

5.1 Problem formulation

In this section we will show the problem formulation, defining the function to be optimized and explaining the function evaluation system.

5.1.1 Function to optimize

The goal of the optimization is to obtain the reflector that produces a light distribution as close as possible to the desired light distribution. The reflector shape is created using a function $r(\overrightarrow{P})$ from a predefined family of functions, such that

$$r(\overrightarrow{P}) :: \mathbb{R}^N \rightarrow \mathbb{S} \qquad \overrightarrow{P} \in \mathbb{R}^N$$

where $\overrightarrow{P}$ is a vector of N parameters in the function domain, and $\mathbb{S}$ is the space of reflector shapes. We need the function $l(r(\overrightarrow{P}))$ that returns the light distribution of the reflector defined by $\overrightarrow{P}$, such as

$$l(r) :: \mathbb{S} \rightarrow \mathbb{L}$$

We also define

$$L(\overrightarrow{P}) \equiv l(r(\overrightarrow{P})) \qquad L :: \mathbb{R}^N \rightarrow \mathbb{L}$$

where $\mathbb{L}$ is the domain space of the light distributions. We define a function f that returns the difference between a given reflector light distribution and a desired light distribution $L^{Desired}$, as

$$f(\overrightarrow{P}) = diff(L(\overrightarrow{P}), L^{Desired})$$

$$f :: \mathbb{R}^N \rightarrow \mathbb{R}$$

where $diff(A, B) :: \mathbb{L} \times \mathbb{L} \rightarrow \mathbb{R}$ is the difference between the light distributions A and B. Therefore, we have to calculate a value $q \in \mathbb{R}^N$ for the domain of f that satisfies:

$$q = \overset{min}{\overrightarrow{P}} \{f(\overrightarrow{P})\}$$

5.1.2 Reflector Evaluation

To evaluate f we use an algorithm with three steps (see Figure 5.1):

- The reflector is constructed by evaluating r from the parameters $\overrightarrow{P}$.
- The illumination L is calculated from the reflector and the given light source. This is the most time consuming step since we are using light sources with millions of rays, and it is calculated on the GPU.

- The desired and resulting light distribution are compared using $diff$. First, L and $L^{Desired}$ are classified in histograms. Next, both histograms are compared using a difference function that uses the l^2 metric. Both steps are also calculated on the GPU.

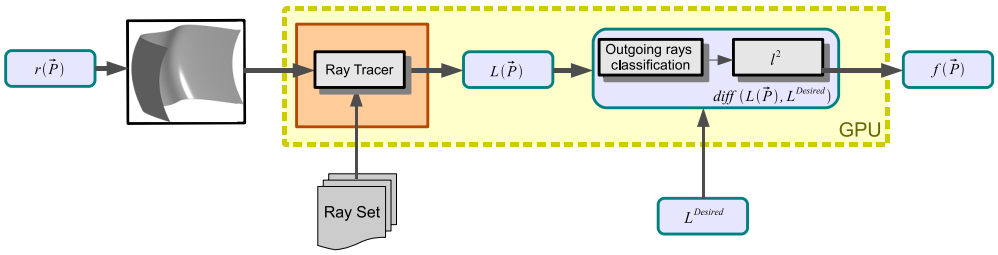


Figure 5.1: Detailed scheme of the node evaluation system.

Following this pipeline, we use both the FIRD and OptiX (see Section 4.6) methods to compute each reflector as is seen in the last chapter.

5.2 Overview

The objective of this algorithm is to obtain a reflector that produces a light distribution as close as possible to the desired light distribution in a rapid way.

The algorithm searches parameters in $\mathbb{R}^N$ space for the best combination such that it produces a reflector that minimizes the function f. To do it, the method is based on the construction of a binary tree, where each tree node represents a reflector where the function f is evaluated. In each tree construction step, an already created node is chosen in a stochastic way, using heuristics based on the node evaluation and current statistical information of surrounding tree nodes. The chosen node is replaced by two new children nodes. These nodes contain all the same parameter ranges as their parent except for one, which is splitted in two subranges. For each new node, we use an heuristic, based on its ancestors, to choose what parameter range to split. Since we evaluate the nodes after they are created, and because we are using a greedy breadth-first search algorithm, we do not need to construct the full tree structure. The process stops when a node evaluation result is below a user termination threshold. However, when the chosen node is close enough to a minimum, or when the parameter space size is smaller than a user defined threshold, a local optimization method is used to converge in a rapid way.

In this case, the chosen node becomes a tree leaf. If the minimum found by the local optimization process is under the user termination threshold, the process stops. Otherwise, the process continues choosing a new tree node.

5.3 Tree construction

As mentioned in the previous section, the tree is lazily created. The tree is constructed according to a greedy breadth-first search algorithm. At each iteration of the optimization algorithm, a tree branch is chosen depending on two stochastic selections: the node selection and the selection of the parameter space to split. To better guide the process, in order to obtain good rates in the optimization convergence, we use a set of heuristics on both stochastic selections.

Now we are going to explain the node selection process, and after that the parameter splitting mechanism.

5.3.1 Node selection

Each time a node is created it is stored into a list called *"tree section"*, which is a sorted list where all current tree leaves are stored and sorted by their evaluation values. When a node is selected, it is removed from the list, and two new children nodes are created and stored into the list (see Figure 5.2).

The tree section objective is to help in the decision of what node to choose to continue the optimization. However, simple node evaluation is not enough to decide if we are approaching to the global minimum or not. We should also consider the differences between the current evaluation and the last evaluated values to get information about how we are approaching a minimum. But we cannot know if this is a local or global minimum. However, we can use the information of already created nodes to know how the function f behaves on different parameter ranges, and decide what domain zone is more suitable to be a global minimum.

For each tree section node, we calculate a corresponding weight value. Then, the list is sorted by the node evaluation values multiplied by their respective weights. This used weight is the linear combination of three component weights, that are estimations of how close the current node is to a minimum.

The first weight, named w_{diff}, represents the difference between the current node and the parent node evaluation values, so

$$w_{diff_i} = \frac{max(f(P_i^{parent}) - f(P_i), 0)}{maxVal} \cdot weightDiff$$

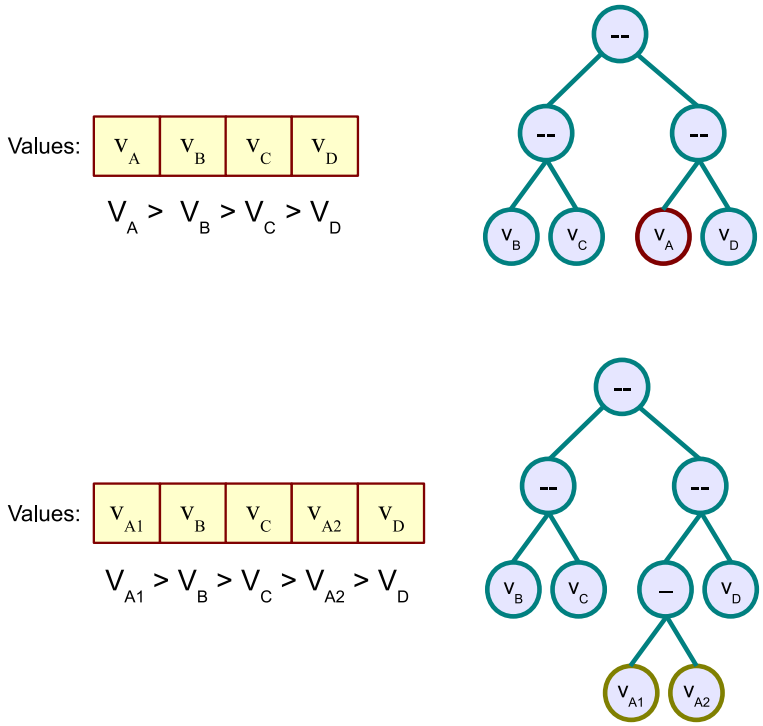


Figure 5.2: Node selection example. When a node is selected from the *tree section* it is replaced by its two children. The *tree section* is updated sorting the nodes by their weighted evaluation values.

where $f(P_i^{parent})$ is the evaluation of the parent node, $maxVal$ is the maximum accepted node evaluation value, and $weightDiff$ is a user-defined factor for controlling the importance of this difference. The nodes with greater evaluation values than $maxVal$ are stored at the end of the tree section with a weight value of 0. We cannot simply discard these nodes because the parameter values used only are representative values of their ranges. Therefore, these nodes are considered after all the other suitable branches are processed. For the other nodes, if w_{diff} is high it means that $f(P_i)$ is on a steep slope of the function, thus we probably are approaching faster to a minimum (see Figure 5.3)

The second weight, named $w_{density}$, represents the density of nodes around the current one, so

$$w_{density_i} = \left(1 - \frac{near_i}{NTreeNodes}\right) \cdot weightDensity$$

where $near_i$ is the number of nodes in a neighborhood of node i (we get the nodes using an Euclidean distance and a user specified maximum distance),

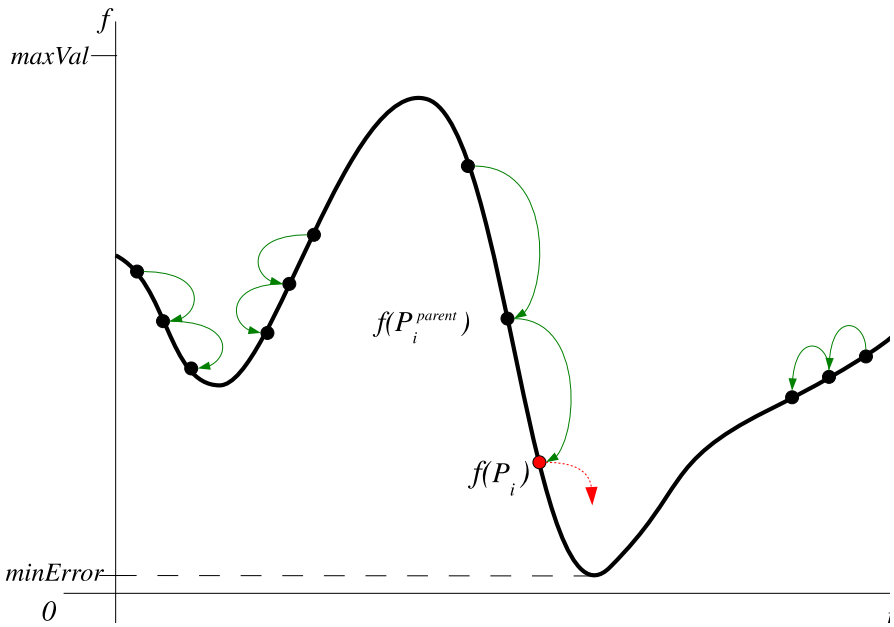


Figure 5.3: The w_{diff} weight allows the selection of nodes that are approaching faster to a minimum

$NTreeNodes$ is the number of current tree nodes, and $weightDensity$ is a user-defined factor for controlling the importance of this density. If the density is high, it means that the current parameter range contains a lot of evaluated nodes, so $w_{density}$ should be low because we consider that the region has been evaluated enough (see Figure 5.4).

The third weight, named w_{stdev}, represents the variance on evaluation values of nodes around the current one, so

$$w_{stdev_i} = \left(\frac{stdev_i}{near_i \cdot (maxVal \cdot mean_i)^2} \right) \cdot weightStDev$$

where $stdev_i$ is the standard deviation of the evaluation values of the near nodes, $mean_i$ is the mean of evaluation values of the near nodes, and $weightStDev$ is antoher user-defined factor to control the importance of this variance. If the current parameter domain region has low variance, it probably means that the function is nearly flat in this region, so it might imply there is not any minimum. In this case, w_{stdev} should to be low (see Figure 5.5).

The final composed value is normalized to avoid biased data, so the

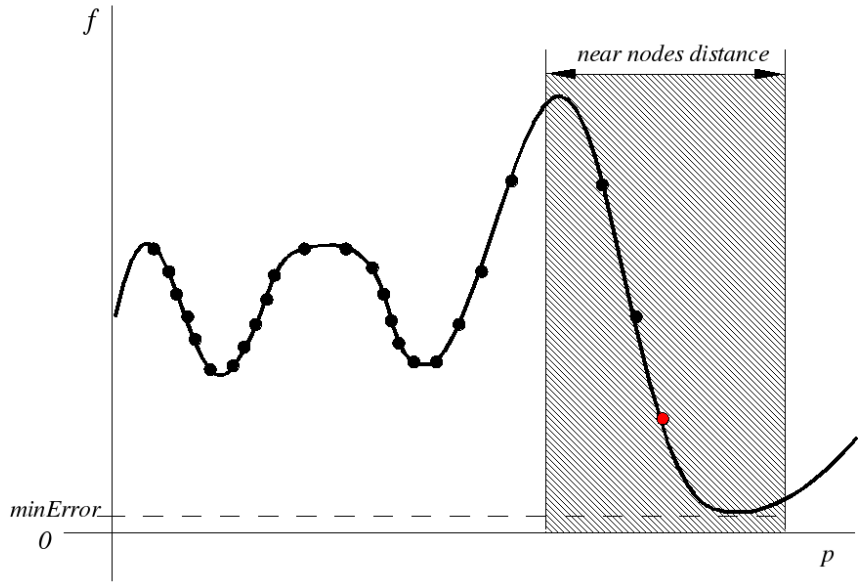


Figure 5.4: The nodes with smaller number of near nodes have higher $w_{density}$ values.

weighted evaluated value v_i for node i is

$$v_i = \left(1 - \frac{f(\overrightarrow{P}_i)}{maxVal}\right) \cdot w_i \tag{5.1}$$

$$w_i = \frac{w_{diff_i} + w_{density_i} + w_{stdev_i}}{weightDiff + weightDensity + weightStDev}$$

The node selection is performed stochastically by importance sampling (see Figure 5.6), giving more priority to the nodes with greater v values. Note that this random selection avoids focusing the optimization progress only on one tree branch, so the optimization always considers a set of possible minima at the same time.

5.3.2 Node parameter space splitting

As we already mentioned, for each parameter, a range is defined and stored in each node. Initially, these ranges are the domain of each parameter space, that are taken from the initial constraints for the $\mathbb{R}^N$ space. Since we need a

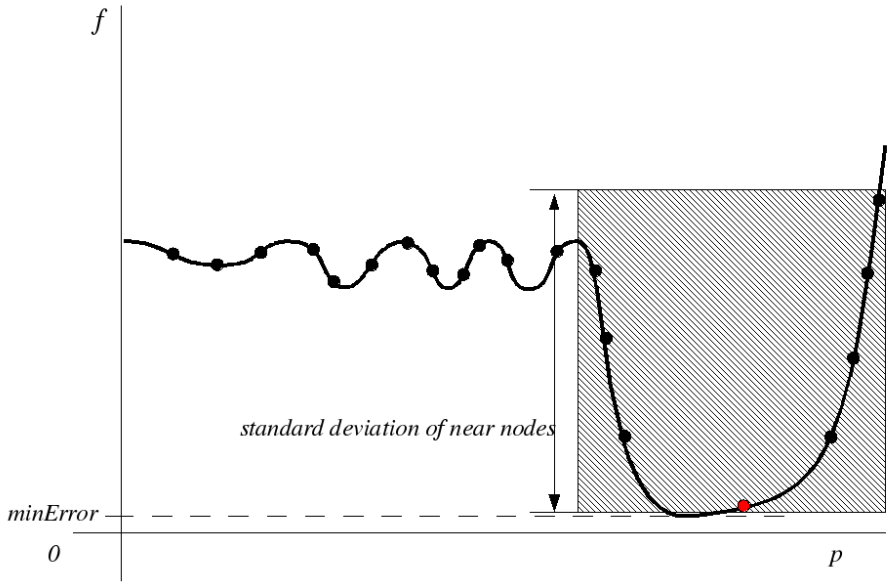


Figure 5.5: The nodes with greater standard deviation with respect to near nodes have higher w_{stdev} values.

parameter vector $\overrightarrow{P} \in \mathbb{R}^N$ to evaluate f, the central values of each parameter range are used.

$$\overrightarrow{P} = \{p_0, ..., p_i, ..., p_N\}$$

$$p_i = \frac{d_{i_{min}} + d_{i_{max}}}{2} \qquad p_i \in [d_{i_{min}}, d_{i_{max}}]$$

where $d_{i_{min}}$ and $d_{i_{max}}$ define the range of parameter p_i.

When a node is selected from the *"tree section"*, two new children nodes are created. Then, a parameter p_j is chosen stochastically from $\overrightarrow{P}$, and its range is splitted in two equal parts, creating two new vectors named $\overrightarrow{P_{left}}$ and $\overrightarrow{P_{right}}$, that are assigned to the created children nodes.

$$\overrightarrow{P_{left}} = \{p_0, ..., p_{j_{left}}, ..., p_N\}$$

$$\overrightarrow{P_{right}} = \{p_0, ..., p_{j_{right}}, ..., p_N\}$$

$$p_{j_{left}} = \frac{d_{j_{min}} + p_j}{2} \qquad p_{j_{left}} \in [d_{j_{min}}, p_j]$$

$$p_{j_{right}} = \frac{p_j + d_{j_{max}}}{2} \qquad p_{j_{right}} \in [p_j, d_{j_{max}}]$$

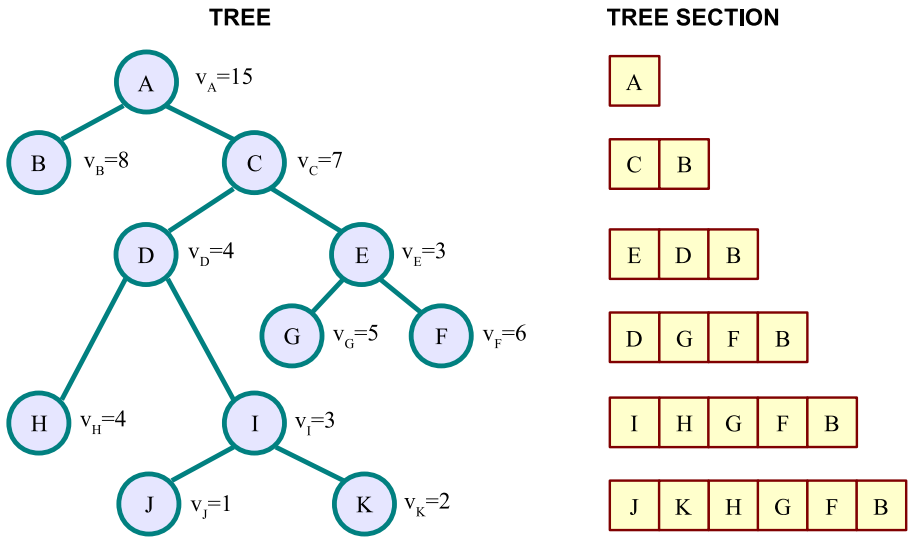


Figure 5.6: Node selection example. The tree is created on the fly by selecting for each optimization step the best suitable node by importance sampling. The tree section is updated with the new children that are inserted. The list is kept sorted by the values v_i.

We use an heuristic based on node ancestors to choose p_j. For each node we store a vector $\overrightarrow{H}$, named *"history vector"* (of size N), where each entry contains information about one parameter. The purpose of this vector is to provide the information of the minimization progress depending on the previously chosen parameters in the same tree branch. On the first optimization step, a random vector index is chosen from the root node history vector to split the parameter range of the first children nodes. Next, when the new children nodes are created and evaluated, the history vectors of each one are cloned from their parent node. Then, the vector entry with the same index as the previous chosen one on the parent node is updated with the difference between the current and parent node weighted evaluation value v_i (see Equation 1) :

$$\overrightarrow{H_{parent}} = \{h_0, ..., h_k, ..., h_N\}$$
$$\overrightarrow{H_{left}} = \{h_0, ..., h'_{k_l}, ..., h_N\}$$
$$\overrightarrow{H_{right}} = \{h_0, ..., h'_{k_r}, ..., h_N\}$$

with

$$h'_{k_l} = v(\overrightarrow{P_{parent}}) - v(\overrightarrow{P_{left}})$$

$$h'_{k_r} = v(\overrightarrow{P_{parent}}) - v(\overrightarrow{P_{right}})$$

where $\overrightarrow{H_{parent}}$ and $\overrightarrow{P_{parent}}$ are the history and parameter vectors of the parent node, $\overrightarrow{H_{left}}$, $\overrightarrow{P_{left}}$, $\overrightarrow{H_{right}}$ and $\overrightarrow{P_{right}}$ are the history and parameter vectors of the left and right child nodes respectively, and k is the previous selected parameter index of the parent node.

To choose the parameter range to split, we use the history vector to perform a stochastic selection by importance sampling, giving more priority to the parameters with greater difference values. Therefore, we give more importance to parameters that previously have had greater differences with their parents, which means that the optimization progress with these parameters probably approaches faster to the minimum.

In Figure 5.7 there is an example of the parameter selection. The example shows a tree parameter selection with $N = 3$. At the root node the first parameter is chosen and splitted in two new children nodes named left and right nodes (each one contains $\overrightarrow{P_{left}}$ and $\overrightarrow{P_{right}}$ respectively). Then, the two new nodes are evaluated, and the differences between the weighted values of each one and its parent are stored in the first history vector entry of each new node. In this example the right node is chosen from the tree section list, the first parameter is selected again and splitted to create its two new children. Later on, the new left node is chosen in turn, it turns out the first history vector entry value is smaller. This means that the previously chosen parameter does not carry significant changes with respect its parent, so we are not approaching to a minimum in a fast way. In this case, the algorithm chooses other parameters that previously produced similar or better approaches to a minimum, selecting stochastically the second history vector entry, and splitting the second parameter.

5.4 Local search

As we said before, we use a local search method to get as close as possible to the final solution. There are many methods to perform a local search, like the conjugate gradients algorithms (see Section 2.3.1). We have chosen the Hooke and Jeeves method [HJ61] because it is fast, like other conjugate gradient algorithms, and it is not necessary to calculate the derivatives of the function, replacing them by a few function evaluations

This method has two steps, the Exploratory Step and the Pattern Step. The Exploratory Step searches for a better parameter value by evaluating new parameter vectors around the original one. A small offset Δ is added and subtracted for each parameter, evaluating each combination. We search

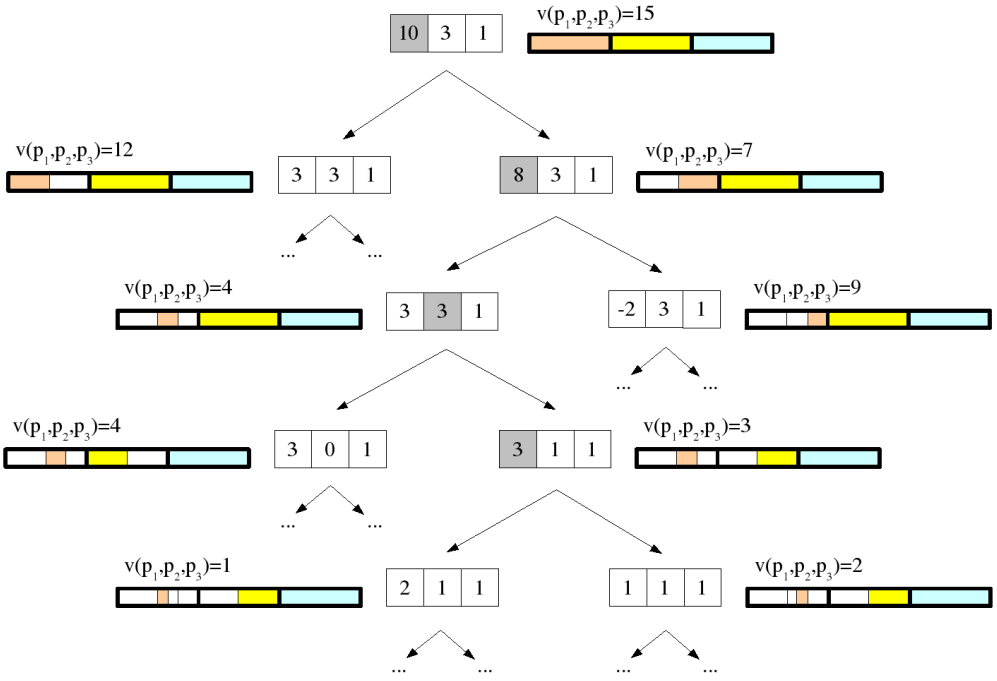


Figure 5.7: Example of parameter selection with 3 parameters. The color cells are a representation of the parameter ranges. Observe that these ranges are reduced for each level of the tree with respect to its parent node. The function $v(p_1, p_2, p_3)$ is the node weighted evaluation on the three parameters. The gray vector cells are the chosen parameters for each node.

a new parameter vector $P_k{}'$ from the original one P_k such that

$$f(P_k{}') = min(f(P_k^+), f(P_k^-), f(P_k))$$
$$P_k^+ = (p_1, ..., p_i + \Delta, ..., p_N)$$
$$P_k^- = (p_1, ..., p_i - \Delta, ..., p_N)$$

where k is the local search step. If there is not any combination that produces better results than the original one, the Exploratory Step fails. Otherwise, the best combination is chosen.

The main step is the Pattern Step. When it starts, an Exploratory Step is performed with the original tree node parameter vector. Next, the parameters change with a jump in the direction of the chosen Exploratory Step parameter vector P_k', creating a new parameter vector P_{k+1}, such that

$$P_{k+1} = P_k{}' + ((P_k{}' - P_k) \cdot stepSize$$

where *stepSize* is the jump value. If the Exploratory Step fails, it means that this is not a good way to reach the minimum, and the previous jump is undone. Then, the offset is reduced, and a Exploratory Step is performed again.

The process stops when it reaches the global minimum, or when the offset is small enough to be able to assess that the current local minimum is not the desired minimum. Back in our method, if the desired minimum is not found, the whole tree node is discarded.

In the Figure 5.8 there is an example of Hooke and Jeeves process. Note the Exploratory Step fail in steps 8 and 13. In these cases, the jump is undone and the *Delta* offset value is reduced, that is the axis lengths in the Figure 5.8.

5.5 Results

We have tested our method with three test-cases. The first one, called *Model A* (see Figure 5.9), uses the basis function

$$r(\overrightarrow{P})(x,y) = p_0x^2 + p_1y^2 + p_2 \qquad \overrightarrow{P} = (p_0, p_1, p_2) \in \mathbb{R}^3 \tag{5.2}$$

to construct the reflector, where $x, y \in [-0.5, 0.5]$ to match the holder boundary. The light source is a sphere of radius 0.05 mm placed at the origin. The emittance is a cosine distribution with 1100 lumens. The second test-case, called *Model B* (see Figure 5.10), uses the basis function

$$r(\overrightarrow{P})(x,y) = p_0 * (-|\sin(y/\pi) * (p_1 * \sin(x/\pi) -$$

$$-p_3 * (1 - |\cos(p_4 * x/\pi)|))|) \qquad \overrightarrow{P} = (p_0, p_1, p_2, p_3) \in \mathbb{R}^4 \tag{5.3}$$

to construct the reflector, where $x, y \in [0, 10]$ to match the respective holder boundary. The light source is a sphere of radius 0.5 mm placed at (5, 5, -0.5). The emittance is a cosine distribution with 2500 lumens. The third test-case, called *Model C* (see Figure 5.11), uses 7 control values to define the geometry shape of a real-case reflector. The light source has a cosine emittance of 28000 lumens on a cylinder of length 65 mm and radius 4 mm, placed along the main reflector axis.

The desired lighting and optimization results for each case are also shown in figures 5.9, 5.10 and 5.11 respectively. For each one it is shown the desired and resulting lighting in IES format. Also, a render of the lighting projection is shown to help a better understanding of how the lighting is. Finally, we can observe the reflector shape with the initial parameter values and the

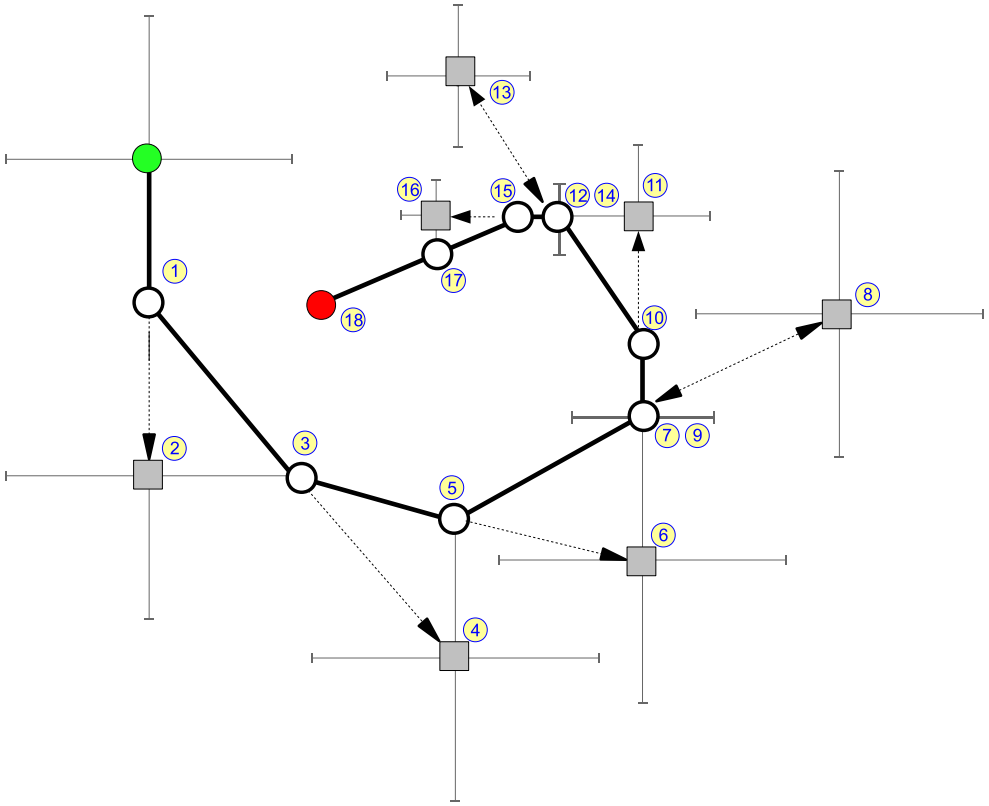


Figure 5.8: Example of Hooke & Jeeves optimization method for a function of two parameters. The numbers show the process order. The gray squares symbolize each Pattern Step. The axis around the squares are the parameters of the function. *Delta* is represented as the axis lengths. The circles represent the successful Exploratory Steps.

obtained parameters after the optimization. The differences between the desired lighting and the obtained one are quite small, in all cases below 4% (see Table 5.1)

The overall optimization times, number of processed reflectors and relative errors are shown in Table 5.1. It is also shown the number of local searches processed. We have used a 2×10^6 rayset size for *Model A*, and 1×10^6 for the others. The lighting calculation for each reflector lasts, in average, around 200 ms for all cases. The evaluation of the l^2 error for each final lighting distribution lasts around 70 ms in average. Considering some extra time to compute the reflector shape, less than 300 ms are needed in average to compute the final $f(\overrightarrow{P})$ value for each reflector.

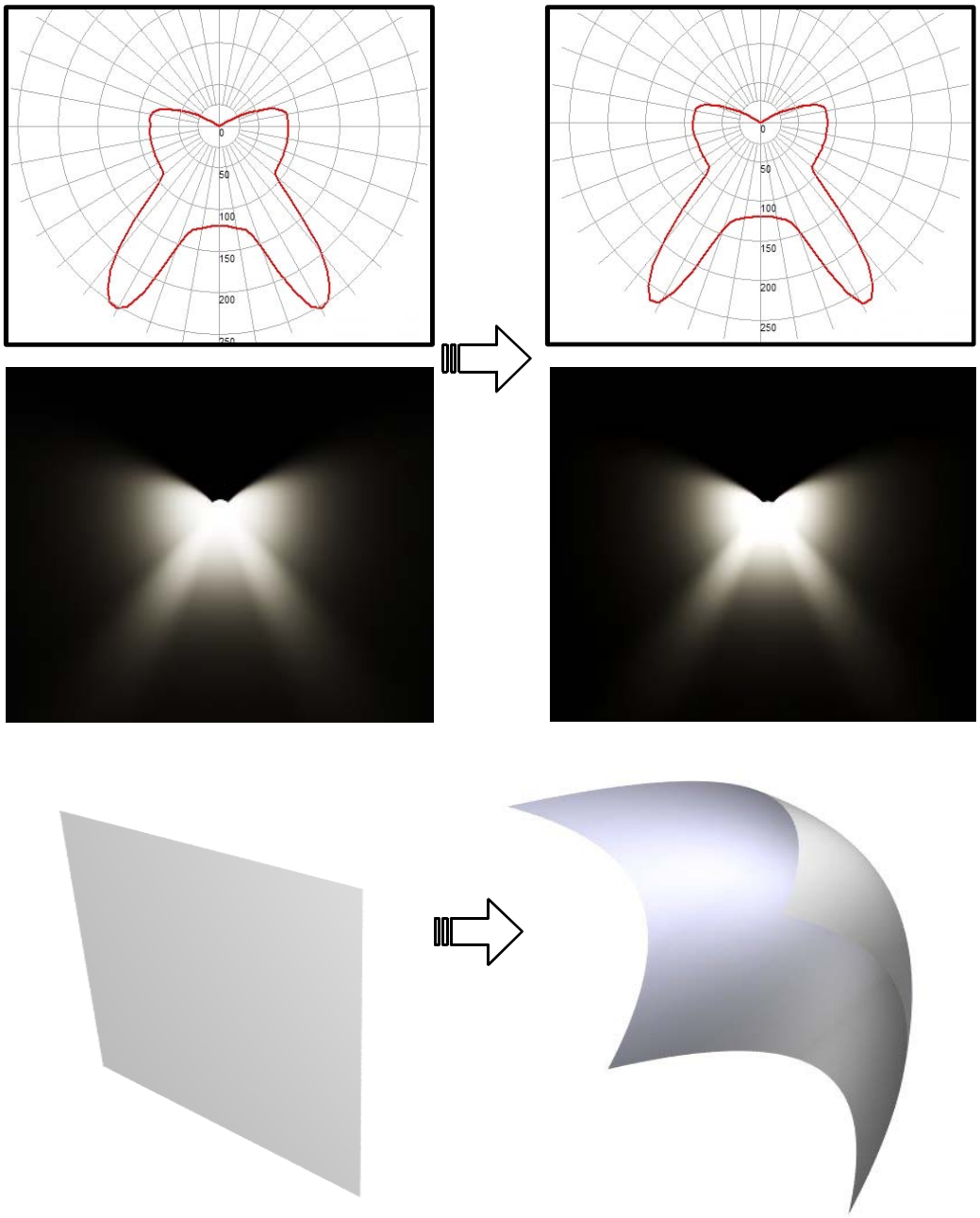


Figure 5.9: Results for our *Model A*. At the left, the desired lighting. At the right, the optimization result. From the top to the bottom, the IES representations, a render projection of lighting distribution, and the initial and final reflector shapes. These shapes are constructed using the parametric function (2).

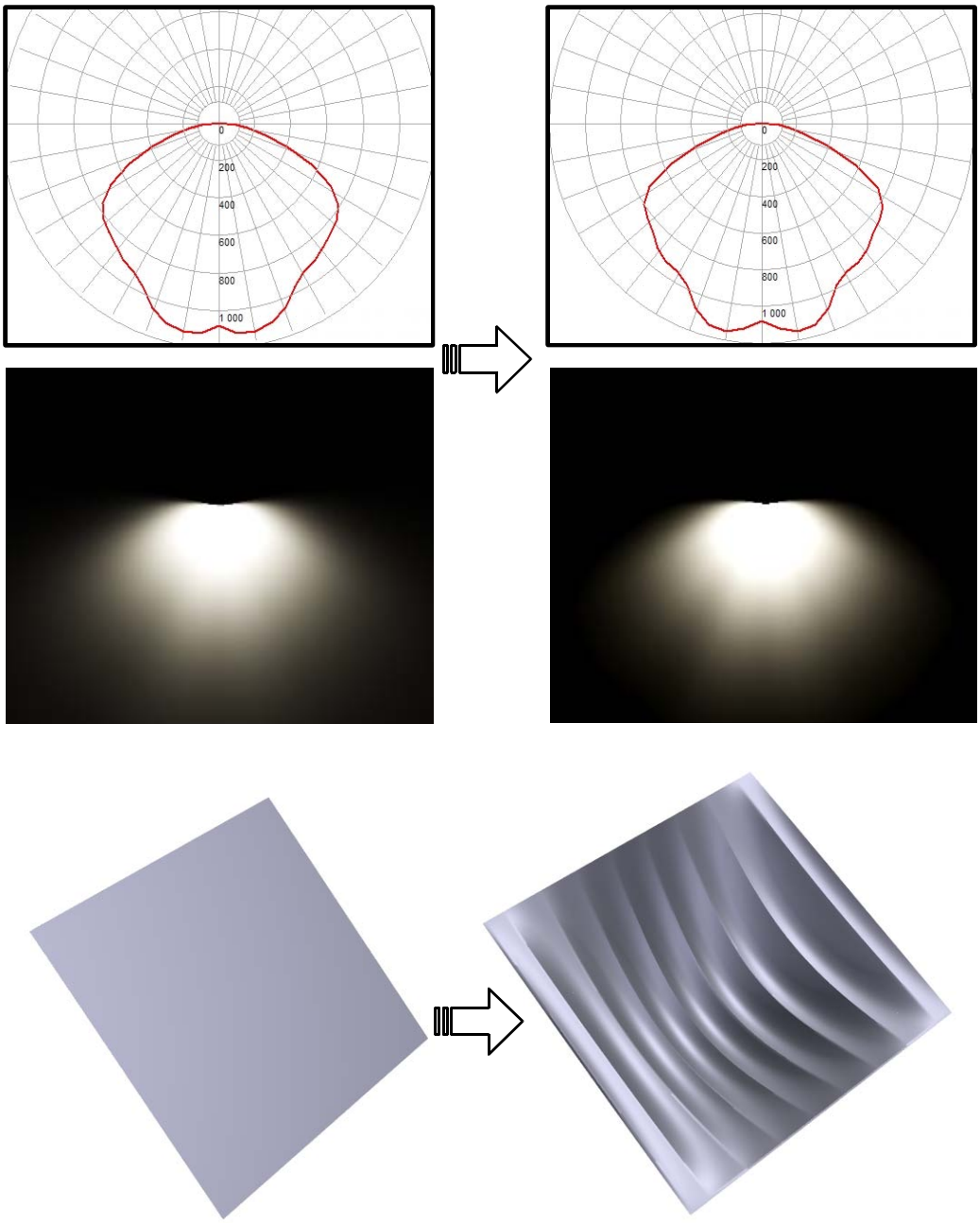


Figure 5.10: Results for our *Model B*. The image structure is the same as in Figure 5.9. The reflector shapes are constructed using the parametric function (3).

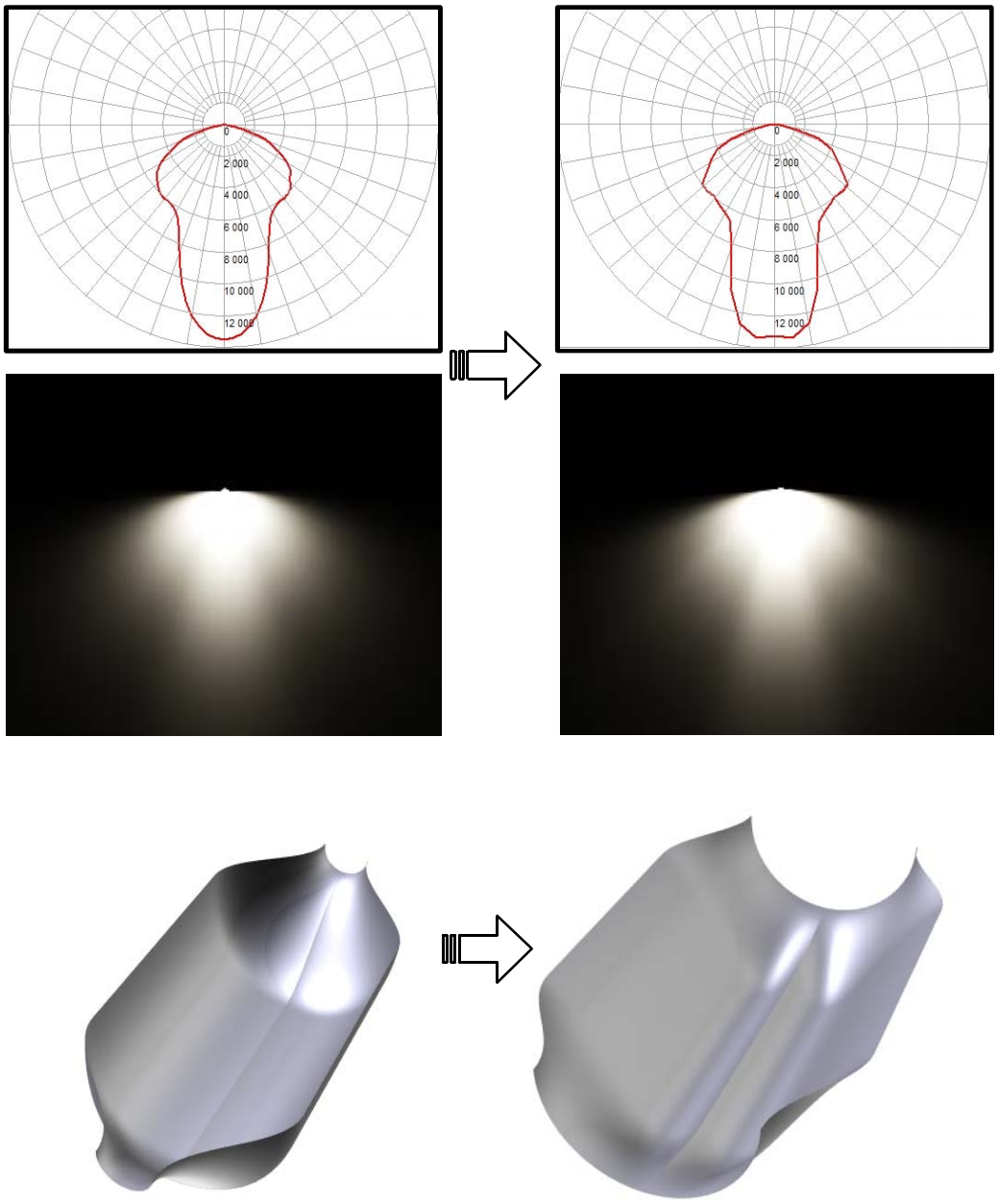


Figure 5.11: Results for our *Model C*. The image structure is the same as in Figure 5.9. The reflector shapes are constructed by modifying 7 control parameters. This kind of reflector is based on a real reflector.

Model	A	B	C
Mesh triangles	19796	19404	16200
Parameters	3	4	7
Evaluated reflectors	637	3172	1431
Local searches	1	2	2
Optimization time	3m 2s	12m 50s	6m 30s
Relative error (%)	1.08	0.32	3.7

Table 5.1: Optimization results for all tested cases.

The same cases have been also tested using a brute force optimization and the Simulated Annealing algorithms (see Section 2.3.2) for comparison purposes. The results for the comparison of the brute force optimization are shown in Table 5.2. The number of tested reflectors (see Figure 5.12) and, in consequence, the time to reach a solution (see Figure 5.13) are exponentially related with the number of parameters, as the brute force algorithm tests all parameter combinations. Also the relative errors tend to be larger because the local search method used in our algorithm approaches to the minimum with more precision than what is possible with the brute force algorithm (see Figure 5.14).

Model	A	B	C
Evaluated reflectors	39965	75486	772677
Optimization time	2h 52m	4h 43m	53h 21m
Relative error (%)	1.09	0.54	11.62

Table 5.2: Optimization comparison with Table 5.1 using a brute force optimization algorithm.

Finally, we present the comparison done using the Simulated Annealing optimization algorithm. The results are shown in Table 5.3. Although Simulated Annealing is a global optimization algorithm, the process could quickly fall into a local minimum if the initial temperature is low and the cooling factor is high. Hence the temperature should increase with the number of parameters to optimize. Also, the number of evaluated reflectors is higher than our method (see Figure 5.12). This implies higher computational times to get the solution. The visited reflectors are the number of reflectors that have been evaluated one or more times. To avoid the recalculation of the already evaluated reflectors, we use a sort of cache data structure. Therefore, although for the model B a similar number of reflectors than our method is

evaluated, the final computational time is higher. Note that, for the model A, the optimization time for the Simulated Annealing algorithm is higher than for the other methods. This is because the global minimum is closer to the global maximum (see Figure 5.16), making it difficult for Simulated Annealing in that domain region. On the whole, the Simulated Annealing algorithm is faster than the brute force algorithm, but slower than our method (see Figure 5.13). Moreover, the relative errors are similar since we can approach to the minimum with more precission than with the brute force method (see Figure 5.14).

Model	A	B	C
Initial temperature	3×10^6	7×10^6	15×10^6
Cooling factor (%)	0.02	0.2	0.2
Evaluated reflectors	25397	4654	5024
Visited reflectors	51550	6282	14035
Optimization time	1h 31m	23m 36s	17m 46s
Relative error (%)	1.1	0.35	4.36

Table 5.3: Optimization comparison with Table 5.1 using the Simulated Annealing optimization algorithm.

Industrial application case

The method has been tested on a real-world industry case, where the goal was to get a reflector to optimize road lighting. The input data and the reflector basis correspond to the third example case seen in the previous section. The resulting reflector is shown in Figure 5.11. In Figure 5.15 there is the lighting comparison, in false color, between the desired lighting over the road, and the resulting one. Although the road lighting is obtained from the composition of many reflectors, we actually only need to calculate one. Then, the lighting distribution is translated to each pole position to compute the final road lighting. As is shown in the image, the differences between the desired lighting and the obtained one are quite small.

The road lighting optimization often needs to calculate the most suitable distance between poles, the boom angle, or other pole placement parameters. Since the nature of our algorithm, these parameters can be easily adapted and optimized together with the rest of the reflector parameters. Road lighting also requires considering quality criteria, which is a composition of different lighting evaluations [SB01]. The most important values of this quality

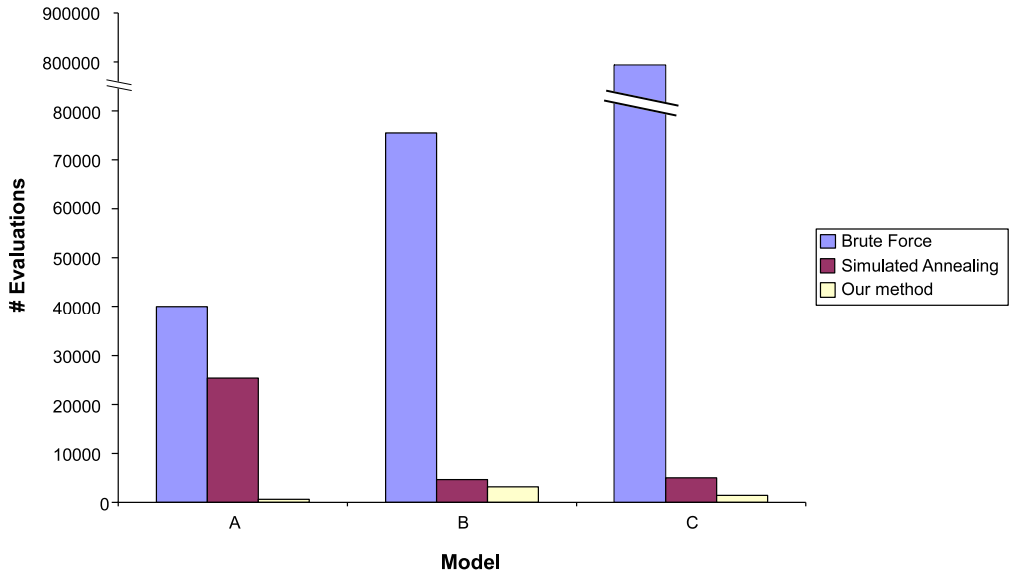


Figure 5.12: Comparison graph for the **number of reflectors evaluated** for each model using the three tested optimization methods.

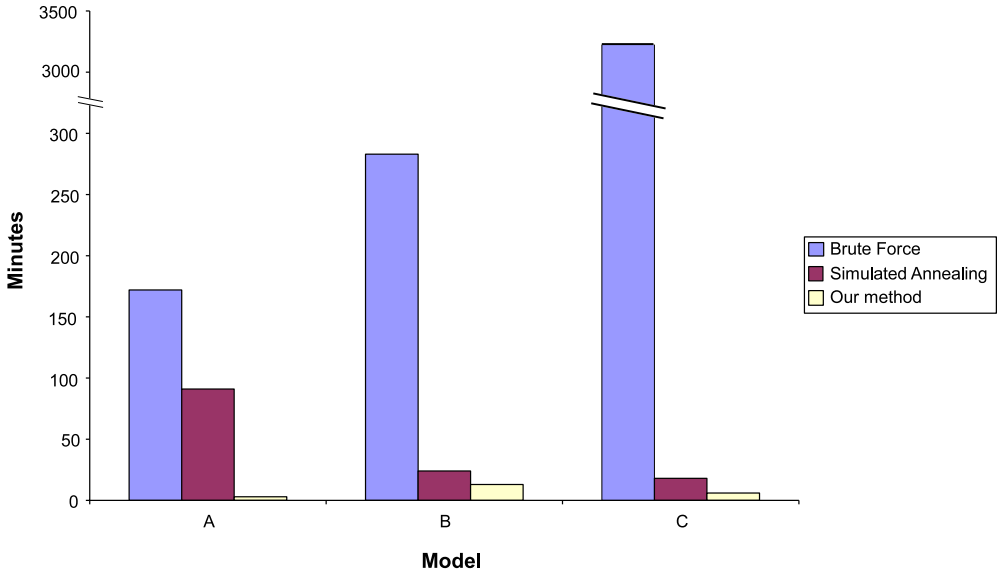


Figure 5.13: Comparison graph for the **optimization time** for each reflector model using the three tested optimization methods.

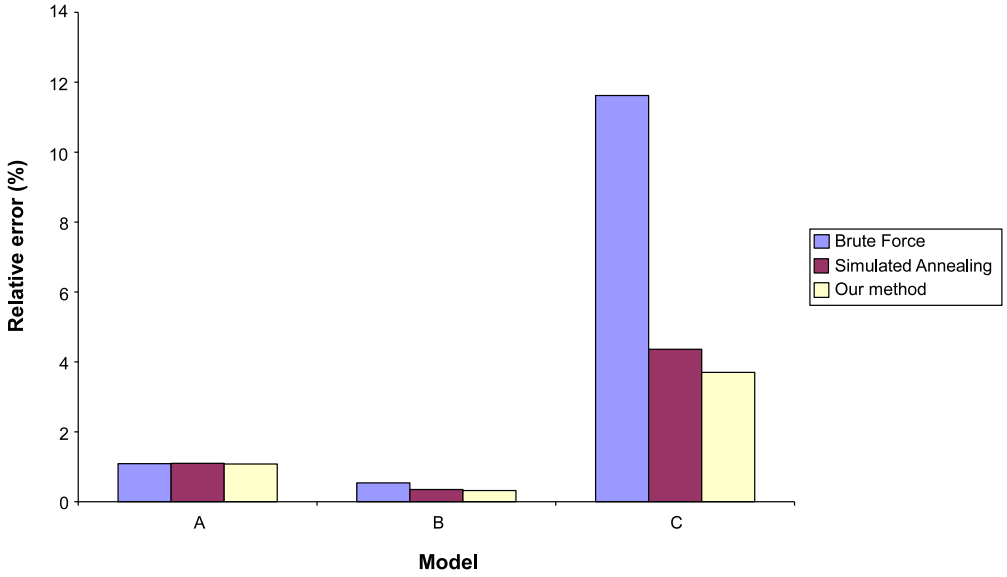


Figure 5.14: Comparison graph for the **relative error** for each reflector model using the three tested optimization methods.

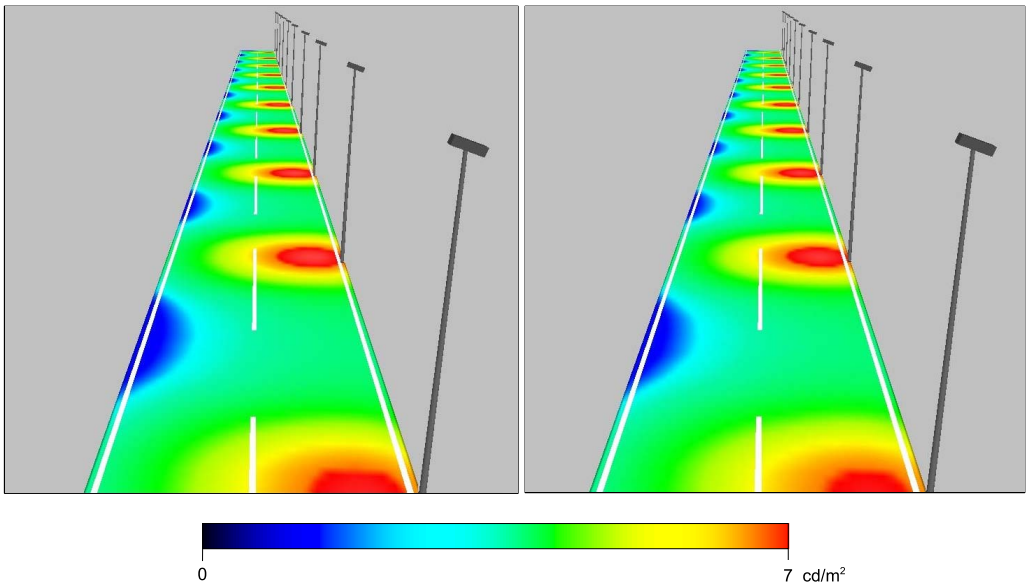


Figure 5.15: Road lighting example in false color (cd/m^2). At the left, the desired lighting. At the right, the optimization result. The differences are quite small. The used reflector is shown in Figure 5.11.

criteria are: the average road luminance, the overall uniformity and the longitudinal uniformity. Other values define the surrounding lighting ratio or the threshold increment of disability glare. Our algorithm permits to include this quality criteria parameters inside the error function using new weights to drive the optimization to a suitable result.

5.6 Discussion

As shown in the previous section, the optimization is very fast, and returns reflectors that generate lighting distributions very close to the desired ones. The main bottleneck of the method is the ray tracing calculation for each reflector. On the other hand, the ray tracing system has a very good ray performance in comparison with other methods.

The key of the optimization method is the node selection and splitting. The weighted evaluations gives us an heuristic way to do it. It is important to cover all the parameter space domain, but always focusing on suitable minima. This way, this is not too far from the kd-tree axis subdivision space idea, so we can be sure that all of the parameter domain space is covered. The selection of the user thresholds and weights is also relevant. In Figure 5.16 it is shown the error graph for a part of the domain of the reflector model A. This example shows how the heuristics that the method uses help the optimization to reach the global minimum and avoid local minima. The flat region is easily discarded using the weight w_{stdev}. Then, the optimization can fall in two minima in a fast way using w_{diff}, but it is discarded with $w_{density}$ when it is sampled enough. It can be seen that the lineal combination between the three weights improve the results for each one of these decisions.

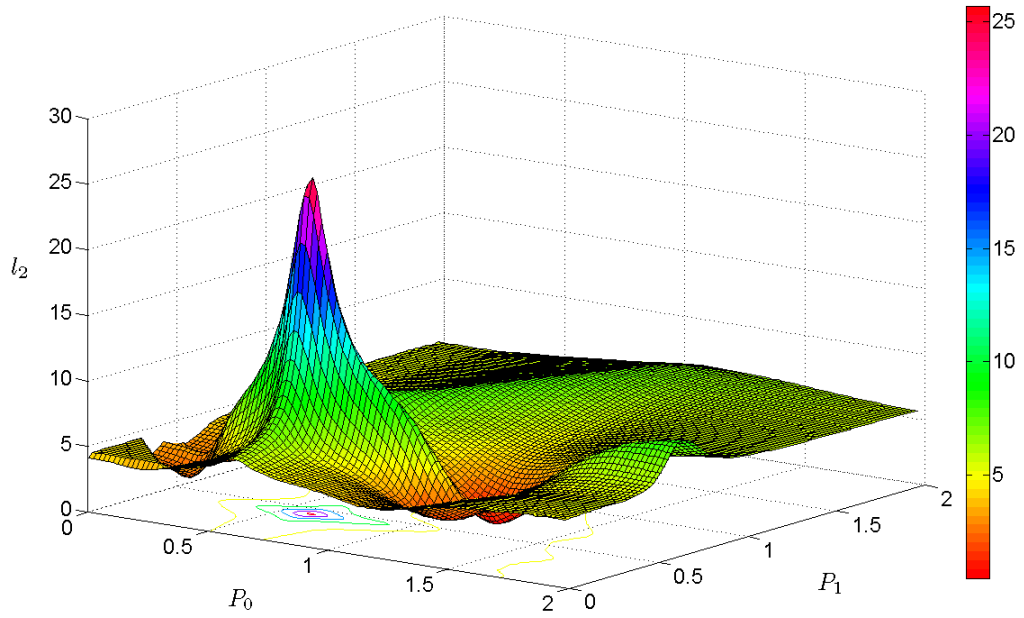


Figure 5.16: Function error graph for model A example and parameters P_0 and P_1. The parameter P_2 has been fixed at -0.5 to help the visualization.

Chapter 6

Conclusions and future work

6.1 Conclusions

We have presented new methods for the inverse reflector design problem that improve previous approaches. We have focused into three main topics: using real and complex light sources, defining a fast ray tracing algorithm to compute the reflector lighting and defining a new optimization algorithm to faster achieve the desired reflector.

To achieve accuracy and realism in the lighting simulations, we have used near-field light source measurements, represented as raysets. Since raysets require a huge amount of data, a novel approach has been presented for compressing them. The compressed model is created by calculating a bounding mesh of triangles for the light source, where each triangle contains the illumination information from the related particles. We have defined a method to perform importance sampling on this representation that allows for smooth particle distributions over the mesh with almost no artifacts, even for very close objects. The mesh representation is also very efficient in storage terms. High precision raysets can contain up to 10 million particles, and the storage needed for this representation is about 270Mb, while our equivalent mesh representation uses only up to a few megabytes.

To perform realistic lighting simulations with complex light sources and a reflector, we have defined a new GPU-based ray tracing method to compute the reflector light distribution in a fast way. The method is based on a very fast GPU algorithm that calculates the reflected rays on the reflector (with an arbitrary number of bounces) in a few seconds, using millions of rays and highly complex reflector shapes. The ray tracing acceleration structure is a height map that is constructed from an image of a reflector, thus we may say that the method is image-based. Then the algorithm focuses on the fast

traversal of the scene through the height map hierarchy. In addition, the final light distribution is compared with the desired one, also using GPU algorithms. The presented method works well in most cases. Since this is an image-based method, it is restrictive on those cases where the height map cannot handle enough geometrical information from the reflector. In addition, we have also used a recent full GPU ray tracing engine: NVIDIA® OptiX™. OptiX can handle any kind of geometry, and we have concluded that it can be faster if there is a low number of light ray bounces inside the reflector, but slower in the opposite case. Moreover, OptiX needs a high performance graphic device, and it can only handle smaller raysets than the method we have proposed.

Finally, we have presented a new global optimization method for the inverse reflector design problem. The goal is to minimize the function that calculates the difference between a reflector light distribution and the desired one. The method is based on a stochastic tree, and it is driven by heuristic rules. The tree is constructed on the fly, where each node stores the function evaluation for a reflector shape from a wide set of parameterized reflectors. The heuristic rules are based on this evaluation and the already calculated tree node evaluations, trying to choose the most suitable branch to reach the desired minimum, and avoiding local minima. The minimum that satisfies the user provided requirements is reached in minutes and with less evaluations than other classic optimization algorithms.

6.2 Contributions

The main contributions of this thesis are:

- A method for compressing dense point (or particles) distributions and restoring them with almost no artifacts. It has been used to compress raysets, but it also could be used to compress photon maps or point-based surface representations.

- A raytracing method based on a very fast GPU algorithm that calculates the reflected rays on a geometry stored into a heightmap acceleration structure. The method can handle millions of rays and it is independent of the complexity of the geometry.

- A global optimization method based on a stochastic tree driven by heuristic rules. The minimum or maximum of the function is reached with fewer evaluations than other classical optimization algorithms,

and it avoids many local minima. The method can be used with any function of the kind $f :: \mathbb{R}^N \rightarrow \mathbb{R}$.

- A global solution for the inverse reflector design problem. It uses complex light sources and reflector geometries into a very fast GPU ray-tracing algorithm. Each step of an optimization algorithm computes a reflector lighting in less than one second, comparing it with the desired lighting. The reflector that produces the lighting closer to the desired one is reached in minutes.

6.3 Publications

The following papers were published with the results of the research in this thesis:

- *Compression and Importance Sampling of Near-Field Light Sources*, Albert Mas, Ignacio Martín and Gustavo Patow Computer Graphics Forum, Volume 27, Number 8, pages 2013-2027, 2008.

- *Fast Inverse Reflector Design (FIRD)*, Albert Mas, Ignacio Martín and Gustavo Patow. Computer Graphics Forum, Volume 28, Number 8, pages 2046-2056, 2009.

- *Stochastic Tree-based Inverse Reflector Design*, Albert Mas, Gustavo Patow and Ignacio Martín. Congreso Español de Informática Gráfica, pp 165-174, 2010

6.4 Future work

The near-field compression method could be considered in a future work to be used to compress other dense set of points or particles, as mentionend in the previous section. For example, the method could be used to compress photon maps in static scenes, since a photon map is a set of points with a related radiance or energy. Another example would be the compression of 3D models stored as a set of points representing the surface, where well-defined models require large data sets. The compression method can reduce these data sets considering only the density of points, and restoring them again with almost no loss of geometric information. Because the restoring method is based on importance sampling, we can specify the number of samples to obtain models with different levels of detail without losing the most significative geometry.

With respect to the GPU-based raytracing method, we consider as future work to include more parameters in the lighting system. We would consider using reflectors with complex BRDF surfaces to simulate real materials. In addition, we would like to consider how the manufacturing procedure stretches and deforms the reflector material, changing its properties. Also, we would like to consider to include new elements in the luminaire system such as specific lenses, usually used to refract and set-up the final lighting.

Finally, the optimization method has demonstrated a good performance for the tested cases, and we think that this could be improved using more specific heuristics. Although this optimization method has been defined specifically for the inverse reflector design problem, we think that it is interesting to explore whether this algorithm can be used in other optimization problems. In addition, we consider as a future work to increase the algorithm parallelization using GPU tools, such as CUDA, to allow the processing of multiple tree branches at a time.

We have presented an efficient method to get a reflector from a desired lighting in a few minutes, but we believe that it could be improved by focusing in more detail on the idea of obtaining the desired reflector shape, or a closer one, directly from the desired lighting, avoiding the optimization process.

Bibliography

[AB99] Amenta N., Bern M.: Surface reconstruction by voronoi filtering. *Discrete Computational Geometry 22*, 4 (1999), 481–504.

[ANS02] ANSI/IESNA: Lm-63-02. ansi approved standard file format for electronic transfer of photometric data and related information, 2002.

[AR98] Ashdown I., Rykowski R.: Making near-field photometry practical. In *Journal of the Illuminating Engineering Society* (1998), vol. 27, pp. 67–79.

[Arv86] Arvo J.: Backward ray tracing. In *In ACM SIGGRAPH 86 Course Notes - Developments in Ray Tracing* (1986), pp. 259–263.

[Arv95] Arvo J.: Stratified sampling of spherical triangles. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1995), ACM Press, pp. 437–438.

[AS00] Ashikhmin M., Shirley P.: An anisotropic phong brdf model. *Journal of Graphics Tools 5* (2000), 25–32.

[ASG08] Anson O., Seron F. J., Gutierrerz D.: Nurbs-based inverse reflector design. In *CEIG '08: Proceedings of the Congreso Español de Informática Gráfica 2008* (2008), Matey L., Torres J., (Eds.), pp. 65 – 74.

[Ash93] Ashdown I.: Near-Field Photometry: A New Approach. *Journal of the Illuminating Engineering Society 22*, 1 (Winter 1993), 163–180.

[Ash95] ASHDOWN I.: Near-Field Photometry: Measuring and Modeling Complex 3-D Light Sources. In *ACM SIGGRAPH '95 Course Notes - Realistic Input for Realistic Images* (1995), pp. 1–15.

[bCL99] BYHEART CONSULTANTS LIMITED: Eulumdat file format specification, 1999. http://www.helios32.com/Eulumdat.htm.

[BD06] BABOUD L., DÉCORET X.: Rendering geometry with relief textures. In *GI '06: Proceedings of Graphics Interface 2006* (Toronto, Ont., Canada, Canada, 2006), Canadian Information Processing Society, pp. 195–201.

[Bek99] BEKAERT P.: *Hierarchical and Stochastic Algorithms for Radiosity.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, 1999.

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[Ben80] BENTLEY J. L.: Multidimensional divide-and-conquer. *Commun. ACM 23*, 4 (1980), 214–229.

[BG01] BOIVIN S., GAGALOWICZ A.: Image-based rendering of diffuse, specular and glossy surfaces from a single image. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001), ACM, pp. 107–116.

[Bli77] BLINN J. F.: Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph. 11*, 2 (1977), 192–198.

[BMR*99] BERNARDINI F., MITTLEMAN J., RUSHMEIER H., SILVA C., TAUBIN G.: The ball-pivoting algorithm for surface reconstruction. *IEEE Transactions on Visualization and Computer Graphics 5*, 4 (October/December 1999), 349–359.

[BMSW91] BAUM D. R., MANN S., SMITH K. P., WINGET J. M.: Making radiosity usable: automatic preprocessing and meshing techniques for the generation of accurate radiosity solutions. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (1991), ACM, pp. 51–60.

[Bre] Breault Research Organization: *http://www.breault.com/*.

[BRW89] Baum D. R., Rushmeier H. E., Winget J. M.: Improving radiosity solutions through the use of analytically determined form-factors. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1989), ACM, pp. 325–334.

[CCWG88] Cohen M. F., Chen S. E., Wallace J. R., Greenberg D. P.: A progressive refinement approach to fast radiosity image generation. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (1988), ACM, pp. 75–84.

[CG85] Cohen M. F., Greenberg D. P.: The hemi-cube: a radiosity solution for complex environments. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques* (1985), ACM, pp. 31–40.

[Che90] Chen S. E.: Incremental radiosity: an extension of progressive radiosity to an interactive image synthesis system. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques* (1990), ACM, pp. 135–144.

[CHH02] Carr N. A., Hall J. D., Hart J. C.: The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, 2002), Eurographics Association, pp. 37–46.

[CKO99] Caffarelli L. A., Kochengin S. A., Oliker V. I.: On the numerical solution of the problem of reflector design with given far-field scattering data. *Contemporary Mathematics 226* (1999), 13–32.

[CPC84] Cook R. L., Porter T., Carpenter L.: Distributed ray tracing. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 137–145.

[CSF99] Costa A. C., Sousa A. A., Ferreira F. N.: Lighting design: A goal based approach using optimization. In *Rendering Techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)* (June 1999), Springer-Verlag, pp. 317–328.

[CT81] Cook R. L., Torrance K. E.: A reflectance model for computer graphics. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques* (1981), ACM, pp. 307–316.

[Dan63] Dantzig G.: *Linear Programming and Extensions*. Princenton, NJ: Princenton University Press, 1963.

[DCC01] Doyle S., Corcoran D., Connell J.: A merit function for automated mirror design. *Journal of the Illuminating Engineering Society 30*, 2 (2001), 3–11.

[Don05] Donnelly W.: Per-pixel displacement mapping with distance functions. In *In GPU Gems 2* (2005), Addison-Wesley, pp. 123–136.

[DS97] Drettakis G., Sillion F. X.: Interactive update of global illumination using a line-space hierarchy. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques* (1997), ACM Press/Addison-Wesley Publishing Co., pp. 57–64.

[EN91] Engl H. W., Neubauer A.: Reflector design as an inverse problem. In *Proceedings of the Fifth European Conference on Mathematics in Industry* (1991), M. H., (Ed.), pp. 13–24.

[ES94] Endl R., Sommer M.: Classification of ray-generators in uniform subdivisions and octrees for ray tracing. *Comput. Graph. Forum 13*, 1 (1994), 3–19.

[FDL10] Finckh M., Dammertz H., Lensch H.: Geometry construction from caustic images. In *Proceedings of the 11th European Conference on Computer Vision (ECCV)* (2010), Springer.

[FKN80] Fuchs H., Kedem Z. M., Naylor B. F.: On visible surface generation by a priori tree structures. In *Computer Graphics* (1980), pp. 124–133.

[Fle87] Fletcher R.: *Practical methods of optimization*. Wiley-Interscience, 1987.

[FS05] Foley T., Sugerman J.: Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2005), ACM, pp. 15–22.

[GGHS03a] Goesele M., Granier X., Heidrich W., Seidel H.-P.: Accurate light source acquisition and rendering. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers* (2003), ACM, pp. 621–630.

[GGHS03b] Granier X., Goesele M., Heidrich W., Seidel H.-P.: Interactive visualization of complex real-world light sources. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applications* (2003), IEEE Computer Society, p. 59.

[GGSC96] Gortler S. J., Grzeszczuk R., Szeliski R., Cohen M. F.: The Lumigraph. In *Computer Graphics Proceedings, Annual Conference Series, 1996 (ACM SIGGRAPH '96 Proceedings)* (1996), pp. 43–54.

[GL93] Glover F., Laguna M.: Tabu search. In *Modern Heuristic Techniques for Combinatorial Problems* (Oxford, England, 1993), Reeves C., (Ed.), Blackwell Scientific Publishing.

[Gla84] Glassner A. S.: Space subdivision for fast ray tracing. *IEEE Computer Graphics & Applications 4*, 10 (1984), 15–22.

[Gla89] Glassner A. S. (Ed.): *An introduction to ray tracing.* Academic Press Ltd., London, UK, UK, 1989.

[GPSS07] Gunther J., Popov S., Seidel H.-P., Slusallek P.: Realtime ray tracing, on gpu with bvh-based packet traversal. In *Symposium on Interactive Ray Tracing* (2007), pp. 113 – 118.

[Grö93] Gröller M. E.: Oct-tracing animation sequences. In *Proceedings der International Conference on Computer Graphics* (jun 1993), pp. 96–101.

[GS87] Goldsmith J., Salmon J.: Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7*, 5 (1987), 14–20.

[GTGB84] Goral C. M., Torrance K. E., Greenberg D. P., Battaile B.: Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph. 18*, 3 (1984), 213–222.

[Gui00] Guillou E.: *Simulation d'environnements complexes non lambertiens à partir d'images: Application à la réalité augmenté.* PhD thesis, Informatique, Signal et électronique et

Télécomunications. IFSIC-IRISA, France, 2000. Available from http://www.cs.ubc.ca/nest/imager/th.html.

[Hav00] Havran V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[HBZ98] Havran V., Bittner J., Zára J.: Ray tracing with rope trees. In *in Proceedings of 13th Spring Conference On Computer Graphics, Budmerice in Slovakia* (1998), pp. 130–139.

[He93] He X. D.: *Physically-Based Models for the Reflection, Transmission and Subsurface Scattering of Light by Smooth and Rough Surfaces, with Applications to Realistic Image Synthesis*. PhD thesis, Cornell University, 1993.

[HJ61] Hooke R., Jeeves T. A.: " direct search" solution of numerical and statistical problems. *J. ACM 8*, 2 (1961), 212–229.

[HK93] Hanrahan P., Krueger W.: Reflection from layered surfaces due to subsurface scattering. pp. 165–174.

[HKSS98] Heidrich W., Kautz J., Slusallek P., Seidel H.-P.: Canned light sources. In *Proceedings of Eurographics Rendering Workshop '98* (1998), Drettakis G., Max N., (Eds.), Springer Wien, pp. 293–300.

[Hol95] Holmes M. H.: *Introduction to Perturbation Methods*. No. 20 in Texts in Applied Mathematics. Springer-Verlag, New York, NY, 1995.

[HSA91] Hanrahan P., Salzman D., Aupperle L.: A rapid hierarchical radiosity algorithm. In *SIGGRAPH '91: Proceedings of the 18th annual conference on Computer graphics and interactive techniques* (1991), ACM, pp. 197–206.

[HSHH07] Horn D. R., Sugerman J., Houston M., Hanrahan P.: Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 167–174.

[Jen96] Jensen H. W.: Global Illumination Using Photon Maps. In *Rendering Techniques '96 (Proceedings of the Seventh Eurographics Workshop on Rendering)* (1996), Springer-Verlag/Wien, pp. 21–30.

[Jen01] Jensen H. W.: *Realistic Image Synthesis Using Photon Mapping.* AK Peters, 2001.

[JL06] Jacobs K., Loscos C.: Classification of illumination methods for mixed reality. *Computer Graphics Forum 25*, 1 (2006), 29–51.

[JMLH01] Jensen H. W., Marschner S. R., Levoy M., Hanrahan P.: A practical model for subsurface light transport. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 2001), ACM, pp. 511–518.

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

[KGV83] Kirkpatrick S., Gelatt C. D., Vecchi M. P.: Optimization by simulated annealing. *Science, New Series 220*, 4598 (1983), 671–680.

[KK86] Kay T. L., Kajiya J. T.: Ray tracing complex scenes. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (1986), ACM, pp. 269–278.

[KO97] Kochengin S. A., Oliker V. I.: Determination of reflector surfaces from near-field scattering data. *Inverse Problems 13*, 2 (1997), 363–373.

[KO98] Kochengin S. A., Oliker V. I.: Determination of reflector surfaces from near-field scattering data ii. numerical solution. *Numer. Math 79*, 4 (1998), 553–558.

[KO03] Kochengin S. A., Oliker V. I.: Computational algorithms for constructing reflectors. *Computing and Visualization in Science 6* (2003), 15–21.

[KPC93] Kawai J. K., Painter J. S., Cohen M. F.: Radiooptimization - Goal Based Rendering. In *Computer Graphics Proceedings, Annual Conference Series, 1993 (ACM SIGGRAPH '93 Proceedings)* (1993), pp. 147–154.

[Lam] Lambda Research, Inc.: *http://www.lambdares.com*.

[LD60] Land A., Doig A.: An automatic method for solving discrete programming problems. *Econometrica*, 28 (1960), 497–520.

[LD00] Loscos C., Drettakis G.: Low-cost photometric calibration for interactive relighting. In *First French-British International Workshop on Virtual Reality* (Brest, France, 2000).

[Lev90] Levoy M.: Efficient ray tracing of volume data. *ACM Trans. Graph. 9*, 3 (1990), 245–261.

[LFD*99] Loscos C., Frasson M. C., Drettakis G., Walter B., Grainer X., Poulin P.: *Interactive Virtual Relighting and Remodeling of Real Scenes*. Available from www.imagis.imag.fr/Publications RT-0230, Institut National de Recherche en Informatique en Automatique (INRIA), Grenoble, France, April 1999.

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D.: Fast bvh construction on gpus. *Computer Graphics Forum 28*, 2 (2009), 375–384.

[LH96] Levoy M., Hanrahan P.: Light Field Rendering. In *Computer Graphics Proceedings, Annual Conference Series, 1996 (ACM SIGGRAPH '96 Proceedings)* (1996), pp. 31–42.

[LKG*01] Lensch H. P. A., Kautz J., Goesele M., Heidrich W., Seidel H.-P.: *Image-Based Reconstruction of Spatially Varying Materials*. Research Report MPI-I-2001-4-001, Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany, March 2001.

[LRR04] Lawrence J., Rusinkiewicz S., Ramamoorthi R.: Efficient BRDF importance sampling using a factored representation. In *ACM SIGGRAPH 2004* (August 2004), pp. 496–505.

[LVV03] Likas A., Vlassis N., Verbeek J. J.: The global k-means clustering algorithm. *Pattern Recognition 36*, 2 (2003), 451 – 461.

[LW93] Lafortune E. P., Willems Y. D.: Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (COMPUGRAPHICS 93)* (1993), pp. 145–153.

[LW94] Lafortune E., Willems Y.: *Using the modied phong reflectance model for physically based rendering.* Technical report cw197, Dept. of Computer Science, Katholieke Universiteit Leuven, Leuven, Belgium, 1994.

[Mar98] Marschner S. R.: *Inverse Rendering in Computer Graphics.* PhD thesis, Program of Computer Graphics, Cornell University, Ithaca, NY, 1998.

[MB90] MacDonald D. J., Booth K. S.: Heuristics for ray tracing using space subdivision. *Vis. Comput. 6*, 3 (1990), 153–166.

[MPBM03] Matusik W., Pfister H., Brand M., McMillan L.: A data-driven reflectance model. In *ACM SIGGRAPH 2003* (August 2003), pp. 759–769.

[Nay93] Naylor B.: Constructing good partitioning trees. In *Proc. Graphics Interface '93* (1993), pp. 181–191.

[Neu97] Neubauer A.: *Design of 3d-reflectors for near field and far field problems.* Springer, 1997.

[Neu03] Neumaier A.: Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica 13* (2003), 271–369.

[NVI09] NVIDIA: Nvidia optix ray tracing engine - programming guide, 2009. http://developer.nvidia.com/object/optix-home.html.

[NW99] Nocedal J., Wright S.: *Numerical optimization.* Springer verlag, 1999.

[OH95] Oguma M., Howell J. R.: Solution of the two-dimensional blackbody inverse radiation problem by inverse monte carlo method. In *ASME/JSME Thermal Engineering Conference* (1995), vol. 3, ASME, pp. 243–250.

[OKL06] Oh K., Ki H., Lee C.-H.: Pyramidal displacement mapping: a gpu based artifacts-free ray tracing through an image pyramid.

In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology* (2006), pp. 75–82.

[Ole99] Olea R.: *Geostatistics for Engineering and Earth Scientists.* Kluwer Academic Publishers, 1999.

[Oli89] Oliker V. I.: On reconstructing a reflecting surface from the scattering data in the geometric optics approximation. *Inverse Problems 5*, 1 (1989), 51–65.

[OPT] OPTIS, Inc.: *http://www.optis-world.com/.*

[PBMH02] Purcell T. J., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[PGSS07] Popov S., Günther J., Seidel H.-P., Slusallek P.: Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (Sept. 2007), 415–424. (Proceedings of Eurographics).

[Pho73] Phong B. T.: *Illumination for computer-generated images.* PhD thesis, 1973.

[POJ05] Policarpo F., Oliveira M. M., Jo a. L. D. C.: Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph. 24*, 3 (2005), 935–935.

[PP03] Patow G., Pueyo X.: A survey of inverse rendering problems. *Computer Graphics Forum 22*, 4 (2003), 663–687.

[PP05] Patow G., Pueyo X.: A survey of inverse surface design from light transport behaviour specification. *Computer Graphics Forum 24*, 4 (2005), 773–789.

[PPD98] Paquette E., Poulin P., Drettakis G.: A light hierarchy for fast rendering of scenes with many lights. *Computer Graphics Forum (Proc. Eurographics '98) 17*, 3 (September 1998), C63–C74.

[PPV04] Patow G., Pueyo X., Vinacua A.: Reflector design from radiance distributions. *International Journal of Shape Modelling 10*, 2 (2004), 211–235.

[PPV07] Patow G., Pueyo X., Vinacua A.: User-guided inverse reflector design. *Comput. Graph. 31*, 3 (2007), 501–515.

[Pre07] Press W. H.: *Numerical recipes : the art of scientific computing*, 3 ed. Cambridge University Press, September 2007.

[PRJ97] Poulin P., Ratib K., Jacques M.: Sketching shadows and highlights to position lights. In *Proceedings of Computer Graphics International 97* (June 1997), IEEE Computer Society, pp. 56–63.

[Rad] Radiant Imaging, Inc.: *http://www.radiantimaging.com*.

[RAH07] Roger D., Assarsson U., Holzschuch N.: Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the gpu. In *Proceedings of the Eurographics Symposium on Rendering* (2007), Eurographics and ACM/SIGGRAPH, the Eurographics Association, pp. 99–110.

[RFS03] Rigau J., Feixas M., Sbert M.: Refinement criteria based on f-divergences. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 260–269.

[RH01] Ramamoorthi R., Hanrahan P.: A signal-processing framework for inverse rendering. In *Computer Graphics Proceedings, Annual Conference Series (SIGGRAPH 2001)* (August 2001), pp. 117–128.

[RUL00] Revelles J., Urena C., Lastra M.: An efficient parametric algorithm for octree traversal. In *Journal of WSCG* (2000), pp. 212–219.

[RW80] Rubin S. M., Whitted T.: A 3-dimensional representation for fast rendering of complex scenes. In *Computer Graphics* (1980), pp. 110–116.

[Rye02] Ryer A.: Light measurement handbook, 2002.

[Sam89] Samet H.: Implementing ray tracing with octrees and neighbor finding. *Computers And Graphics 13* (1989), 445–460.

[San85] Sandor J.: Octree data structures and perspective imagery. *Computers & Graphics 9*, 4 (1985), 393–405.

[SB01] Simons R. H., Bean A. R.: *Lighting Engineering: Applied Calculations*. Architectural Press, 2001.

[Sch94] Schlick C.: A survey of shading and reflectance models. *Computer Graphics Forum 13*, 2 (1994), 121–132.

[Scr66] Screider Y.: *The Monte Carlo Method*. Pergamon Press, New York, N.Y, 1966.

[SDS*93] Schoeneman C., Dorsey J., Smits B., Arvo J., Greenberg D.: Painting With Light. In *Computer Graphics Proceedings, Annual Conference Series, 1993 (ACM SIGGRAPH '93 Proceedings)* (1993), pp. 143–146.

[SG06] Schroders M. F. A., Gulik R. v.: Quadtree relief mapping. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (2006), ACM, pp. 61–66.

[SH07] Scheuermann T., Hensley J.: Efficient histogram generation using scattering on gpus. In *I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2007), ACM, pp. 33–37.

[She97] Shewchuck J.-R.: *Delaunay Refinement Mesh Generation*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[Shi90] Shirley P.: *Physically Based Lighting Calculations for Computer Graphics*. Ph.D. thesis, November 1990.

[SKU08] Szirmay-Kalos L., Umenhoffer T.: Displacement mapping on the GPU - State of the Art. *Computer Graphics Forum 27*, 1 (2008).

[SL01] Shacked R., Lischinski D.: Automatic lighting design using a perceptual quality metric. *Computer Graphics Forum (Eurographics 2001) 20*, 3 (2001), 215 – 227.

[SP89] Sillion F., Puech C.: A general two-pass method integrating specular and diffuse reflection. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques* (1989), ACM, pp. 335–344.

[SP94] Sillion F., Puech C.: *Radiosity and Global Illumination.* Morgan Kaufmann Publishers Inc., San Francisco, CA., 1994.

[SR01] S.Floater M., Reimers M.: Meshless parameterization and surface reconstruction. *Computed Aided Geometric Design 18* (2001), 77–92.

[SS96] Siegel M. W., Stock R.: A general near-zone light source model and its application to computer automated reflector design. *SPIE Optical Engineering 35*, 9 (September 1996), 2661–2679.

[SSK07] Shevtsov M., Soupikov A., Kapustin A.: Highly paralell fast kd-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum (Proc. Eurographics '07)* (2007).

[SWZ96] Shirley P., Wang C., Zimmerman K.: Monte Carlo Techniques for Direct Lighting Calculations. *ACM Transactions on Graphics 15*, 1 (January 1996), 1–36.

[TN87] Thibault W. C., Naylor B. F.: Set operations on polyhedra using binary space partitioning trees. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 153–162.

[TS05] Thrane N., Simonsen L. O.: *A comparison of acceleration structures for GPU assisted ray tracing.* Tech. rep., 2005.

[UPSK07] Umenhoffer T., Patow G., Szirmay-Kalos L.: *GPU Gems 3.* Addison-Wesley, 2007, ch. Robust Multiple Specular Reflections and Refractions, pp. 387–407.

[VG97] Veach E., Guibas L. J.: Metropolis light transport. In *Computer Graphics (SIGGRAPH '97 Proceedings* (1997), Addison Wesley, pp. 65–76.

[VG994] Bidirectional estimators for light transport. In *EGRW '94: Proceedings of the 5th Eurographics workshop on Rendering* (1994), Eurographics Association, pp. 147–162.

[Wal07] Wald I.: On fast construction of sah-based bounding volume hierarchies. In *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 33–40.

[War94] WARD G.: Adaptive shadow testing for ray tracing. In *Proceedings of the Second Eurographics Workshop on Rendering* (1994), Springer-Verlag, pp. 11–20.

[WBS06] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26* (2006), 2007.

[WCG87] WALLACE J. R., COHEN M. F., GREENBERG D. P.: A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. *SIGGRAPH Comput. Graph. 21*, 4 (1987), 311–320.

[WEH89] WALLACE J. R., ELMQUIST K. A., HAINES E. A.: A ray tracing algorithm for progressive radiosity. *SIGGRAPH Comput. Graph. 23*, 3 (1989), 315–324.

[Wei07] WEISE T.: *Global Optimization Algorithms. Theory and Application*, july 16, 2007 ed. Thomas Weise, 2007.

[WFA*05] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P.: Lightcuts: a scalable approach to illumination. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers* (New York, NY, USA, 2005), ACM, pp. 1098–1107.

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Commun. ACM 23*, 6 (1980), 343–349.

[WK06] WÄCHTER C., KELLER A.: Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 139–149.

[WSC*95] WHANG K.-Y., SONG J.-W., CHANG J.-W., KIM J.-Y., CHO W.-S., PARK C.-M., SONG I.-Y.: Octree-r: An adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics 1*, 4 (1995), 343–349.

[Wu99] WU K.: *The Transformation Between Positional Space and Texture Space*. Hpl-1999-103 technical report, HP Labs, 1999.

[YDMH99] YU Y., DEBEVEC P., MALIK J., HAWKINS T.: Inverse global illumination: recovering reflectance models of real scenes from photographs. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive*

techniques (1999), ACM Press/Addison-Wesley Publishing Co., pp. 215–224.

[YJ04] YEREX K., JAGERSAND M.: Displacement mapping with ray-casting in hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Sketches* (New York, NY, USA, 2004), ACM, p. 149.