Bachelor Thesis: Automated Lighting Design For Photorealistic Rendering

Silvana Podaras* Vienna UT

Supervisor: Károly Zsolnai † Vienna UT

Abstract

We present a novel technique to minimize the number of light sources in a virtual 3D scene without introducing any perceptible changes to it. The theoretical part of the thesis gives an overview on previous research in the field of automated lighting design, followed by an introduction to the theory of rendering and genetic algorithms. The implementation is done as extension called "Light Source Cleaner" to *LuxRender*, a physically based, open-source renderer. The algorithm adjusts the intensities of the light sources in a way that certain light sources can be canceled out, thus enabling to render a similar image with significantly less number of light sources, introducing a remarkable reduction to the execution time of scenes where many light sources are used.

Keywords: radiosity, global illumination, constant time

Figure 1: *An example of a physically correct rendered scene.* Source: http://www.luxrender.net

1 Introduction

With the advance of technology, the use of computer graphics has become an everyday routine in motion picture production. Since the making of "Toy Story", the first feature-length film that was entirely computer-animated, there have been a lot of improvements in technology. Nowadays, it is possible to create images by means of computer graphics that are almost indistinguishable from reality. One of the key elements in order to make a film look "real" is to simulate physically correct light transport when computing an image.

One of first attempts for such a simulation was made by Appel [1968], who introduced the ray tracing algorithm. Although the images rendered with that method were far from photorealistic, the fundamental concept has become the basis of state-of-the-art algorithms. A major problem of photorealistic rendering is the time needed to gain high-quality images, because rigorous mathematical and statistical methods are used to simulate realistic effects. We provide a brief overview of the theory behind simulation of light propagation and the rendering process is given in more detail in chapter 3.

Until today, many improvements have been made in order to enhance image quality on one side and to increase rendering speed on the other side. Industry giants like PIXAR use physically based ray-tracing systems in their daily work [Hery and Villemin 2013]. In order to achieve not only a physically realistic looking image, but also a certain look and feel desired by the artist, many light sources are placed in a scene - sometimes up to hundreds of them [Christensen et al. 2003]. It is not only hard to keep track of such a vast amount of lights, but also cumbersome to do the fine-tuning of each light source. Also, the task of positioning light sources and setting their properties is not necessarily an intuitive one. The idea to support the user in this process with computational means has been topic to a great deal of previous research work. An extended overview on related work in the research field of automated lighting design is provided in chapter 2.

Another issue is that render time is also affected by the amount of lights used in a scene. The more light sources exist, the more of them have to be sampled in order to get a smooth and correct output. But what if the overall amount of lights could be reduced? When having so many light sources in a scene, would it be possible to render an (almost) identical image with a lesser number of light sources? For a large number of sources, the answer to this question can hardly be given in finite time when letting a user try out all different settings. Instead, the idea came up to let an algorithm perform the adjustment of the light intensities in order to find a similar result image which uses less lights than the initial setup. The overall concept how such an algorithm could be designed is explained in more detail in chapter 4. In addition, in chapter 5, details on our implementation are presented.

Speeding up the rendering process has been a topic in science and industry for years, and was usually involving the creation of more efficient sampling strategies or better algorithms. This motivation comes from the prohibitively long render times (up to hours or days) in physically based rendering [Pharr and Humphreys 2010]. This thesis is founded on the assumption that using less lights will result in faster execution times. In order to prove this assumption, an empirical evaluation on several scenes was done. Those measurements were made by means of testing on five carefully chosen scenes with a varying number of light sources and a comparison to the ground truth image. The results are presented in chapter 6.

*e-mail: spodaras@cg.tuwien.ac.at

†e-mail: zsolnai@cg.tuwien.ac.at

2 Previous and related work

Although little prior work has been done to automatically reduce the number of light sources in a scene, the more general field of automated lighting design has been a popular topic. This field focuses on computationally adjusting the parameters of light sources in a virtual scene instead of letting the user handle the cumbersome fine-tuning of those parameters. Examples for such parameters are light intensities and emission colors, but also the position of light sources and the direction in which they are pointing. The process of finding appropriate values is often formalized as an optimization problem. All possible light settings for a virtual scene form a multi-dimensional solution space in which the optimal solution has to be found according to some constraints. Those constraints depend on the target of the application.

Even when performing the parameter tweaking by the assistance of a computer, some user input is still necessary. There are different approaches on how to involve the user in the process. The user has to somehow define what kind of results he desires, or at least to accept or discard a proposed solution.
The majority of methods treats the light design problem as an inverse problem: the user has to provide a "desired" image as input, and the computer tries to calculate the according parameters to achieve this effect. Schoeneman et al. [1993] for example assume that the designers of a virtual scene know where to place light sources, but fail in choosing the right intensities or colors to achieve the desired lighting effect. After having the designer "paint" these effects such as shadows and spots of light on a target image, an optimization process is started in order to determine the settings that match the painted image best. In the optimization process, only light intensities and colors are changed, but not the light positions. The matching is done by means of constrained least squares fitting. The light-painting process is interactive, and is able to provide the user direct feedback, but only with approximate preview-images which are not physically correct. For the preview images, only direct illumination is taken into account, and surfaces are assumed to be ideal diffuse reflectors. After the best settings have been evaluated, an optional ray tracing step can be done to compute the final solution, which is expected to be drastically different from the preview image.

Anrys and Dutré [2004] use prerecorded photographs to virtually design lighting for real-life objects. The goal is to find light settings that can be reproduced in reality for those objects. This approach requires a set of light sources with fixed positions, which will not be changed, only the intensities are adjusted by this method. For each light source, a photograph of the real-life object getting illuminated only by this source has to be made. The user then paints the chosen effects (like shadows, light regions) on a neutrally lit photograph of the object with the help of any common image processing software. An optimization routine adjusts the intensities of each light source by using the previously made photographs and combining them in order to achieve the desired result. The downside of this technique is that the objects have to exist in reality, which makes this technique inapplicable for virtual environments. Additionally, the designer has to choose the light and camera positions before making the photographs. This means that the designer should already have at least some experience with lighting design, because a re-arrangement of the positions is a time-consuming process.

Costa et al. [1999] facilitate lighting design for real-life environments by an inverse approach. They assume a virtual 3D scene with geometry and material properties as given. The designer has to place "fictitious luminaires" (virtual light sources) in the scene and define constraints for the lighting effects he wants to achieve. An optimization routine then finds suitable parameter settings with respect to the given constraints. This method makes use of a lighting simulation toolkit called *Radiance* and the optimization software *ASA*, which uses simulated annealing for constrained optimization problems. If a solution is found, an image rendered with global illumination is presented to the user. To define lighting goals or constraints, Costa's method introduces different types of fictitious luminaries and a custom script-language, which allows the user to define a cost function to the optimizer. This also means that an additional learning effort has to be made before the system can be used efficiently.
Costa also mentions that calculation times are rather long and even then it is not guaranteed to find a optimal solution because of the size and high-dimensional property of the search space. It is worth pointing out that genetic algorithms are mentioned as a possible option to conquer this problem.

Another effort to facilitate lighting design by solving an inverse problem has been made by Kawai et al. [1993]. Here, parameters for light emission, direction and material reflectance are being manipulated so that user-set constraints like "minimizing overall energy" or "subjective pleasantness of lighting" are fulfilled.

The premise that the user already has to have an idea about what he actually wants may be seen as downside of the inverse design methodology. A contrary approach is "interactive evolution", which lets the user explore and experiment with a set of possible solutions that the computer creates. Sims [1991] used evolutionary algorithms in combination with user input to generate different sets of plant structures, procedural textures and animations. The quality for each solution is determined by the subjective judgment of the user, before the next evolution step is applied. This approach differs from the common practice in genetic algorithms, where a pre-defined fitness function automatically calculates the fitness. By interactively participating in the process, the user can "guide" the results in a specific direction without having to know about the underlying mechanisms for calculating the parameters.

The design galleries approach from Marks et al. [1997] pushes the concept of interactive evolution even further. The idea is to have the computer set up as many different light settings as possible and present them to the user, so that he can choose a setup that seems most appealing to him. Contrary to the above mentioned methods, this one is focused on varying the placement, type and direction of the lights instead of their intensities. In a 3D scene, the user has to define surfaces on which the lights can be positioned ("light hooks"), and surfaces toward which directional lights should point ("light targets"). Then a dispersion phase tries to compute as many different results as possible using the Manhattan distance between the luminance of two images. The resulting images are then ordered by similarity and presented to the user in the form of a gallery which he can browse. An advantage of this approach is that the user does not have to know a priori what lighting effects he is looking for, and that he may discover settings that he would have never tried out when manually tuning the lighting parameters.

More recent work has been done by Shacked and Lischinski [2001] who use an automatic approach to calculate light settings that result in "comprehensible" images. This means that light parameters are chosen in a way that they enhance those features of a 3D model which convey important information for human viewers. The features deemed as important are, for example edges, shapes and surface properties. An image quality function consisting of several weighted terms for each of those features then calculates a suggestion for an optimal lighting setup with respect to this objective. Bousseau et al. [2011] propose a technique to maximize the amount of information shown in an image, while the focus is set on enhancing physically correct appearance of materials in a scene. The observation that specific materials show their characteristic features

best under certain light settings is the basis for the optimization of lighting in a scene. The most realistic appearance of an object is thus achieved by calculating an environment map that emphasizes the object's material properties.

What distinguishes the previous done research from the method presented in this paper is the goal of the optimization process. While all of the above presented work focuses on tweaking the parameter vector so that an aesthetically pleasing result is produced, or certain image features are enhanced, the goal of this work is to reduce the number of light sources used in a scene by modifying the parameters that control light intensities. Unlike in real-life applications (for example lighting design for a real room), the use of up to hundreds of light sources is common in modern motion picture production. With the exception of the "Design Galleries" approach from Marks et al., the above mentioned work is not dealing with scenes were such a high amount of lights are in use. Another difference is that the majority of approaches described above use linear optimization for simplicity, where the full extent of many of these optimization problems may require non-linear techniques.

3 Theoretical Background

The following section gives an overview on the theoretical background of this thesis. First, an introduction to the creation of images from a virtual 3D scene is given. Apart from the papers quoted, this introduction is mostly based on the book PBRT [Pharr and Humphreys 2010], which will not be explicitly quoted in the text for better readability. Furthermore, an overview of the open source renderer *LuxRender* is presented. The theoretical background of genetic algorithms is then also briefly explained, because it is a crucial element in understanding the design of the algorithm described in the next chapter.

3.1 Rendering

In general, the term rendering refers to the process of calculating a 2D image from the description of a virtual 3D scene. The scene usually consists of objects with certain material properties, one or more light sources and a point and direction from which the scene is being viewed. The software that calculates the images from the description of the scene is called "renderer". The variety of applications is broad: rendering is used in interactive applications like video games, but also in visualization of medical data or in movies, to name just a few examples.
Roughly, one can distinguish between applications where rendering has to happen in real time and applications where physical accuracy is the primary goal instead of high performance. In both cases, to gain a 2D image, the parts of the scene which are visible from the given viewpoint and how they get illuminated by the light sources in the scene have to be determined. While in real time-rendering, simplifying assumptions are made about light transport, the goal of physically based rendering is to simulate light transport as it happens in nature, so that the resulting image is not distinguishable from reality. Achieving a physically correct simulation of light transport is a complex task and state-of-the-art algorithms are often based on the ray-tracing algorithm, which will be explained further in the next paragraph.
It should also be mentioned that in recent years, it was tried to combine physically based rendering and real-time rendering with the goal to provide ray tracing as an alternative to the current techniques used in real time rendering. Although significant improvements have been made to speed up the ray tracing process, the results are still too slow to be used in applications where a high frame rate is required.

Ray tracing The ray tracing algorithm was first formalized by Appel [1968]. The basic idea is to imitate how a photographic camera works. What a real camera does is to 'collect' incoming rays of light that pass trough its lens on a film. An even simpler model is the pinhole camera, which has only a small hole instead of a lens through which the light has to pass. To simulate this mechanism with a renderer, the following components are needed: a camera, a film (or image), rays (represented as vectors), and a scene (as described before).
The virtual camera in a renderer is positioned at the viewing point described in the scene. One can also imagine the camera position to represent the eye of a virtual viewer. The film can be modeled as a virtual image plane, for example as an array of image pixels. The task of the camera is to construct rays that have their origin at the camera's position and point in direction of each pixel on the image plane. One can imagine that those rays now get followed or "traced" when they make their way into the scene. The color of each pixel the viewer is going to see depends on how much light arrives from the scene along the corresponding ray. For each ray, an intersection test with all the objects in the scene has to be done. If a ray intersects an object, this point should be visible. If a ray intersects more than one object, the intersection point that is the nearest one to the viewer should be visible on the image plane. To form an image, a color has to be assigned to each pixel. The color of a pixel is thus dependent from the object with which the nearest intersection was made.
A naive approach would be to simply set the color of that pixel to the color that was defined for the object when creating the scene. This would result in a flat looking image, where every pixel an object is covering has the same color and brightness. (See figure 2 for an example.) In order to get a three dimensional impression,

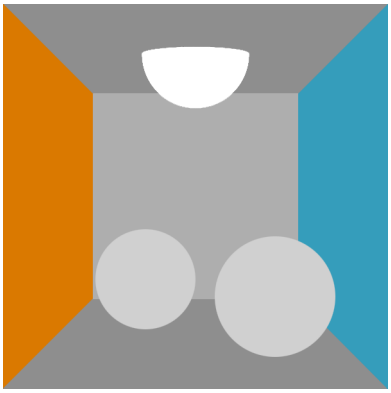


Figure 2: *Result when assigning the object's color to a pixel instead of performing correct shading.*
Source: http://www.wikipedia.com

the correct shading and coloring of each point has to be evaluated. To do so, the lights distribution in the scene has to be simulated and additional information about the geometry (for example surface normals) is also necessary. The actual calculation is done by local or global illumination models.
To summarize the ray tracing algorithm: Rays are cast into a scene, intersection tests are made with the objects and pixel colors are determined depending on the nearest intersection point between a ray and the objects. It is noticeable that although this technique is relatively simple, it is already possible to calculate (hard) shadows for an image. This is done by constructing a shadow ray from the intersection point towards the light source. If it hits another object

before reaching the light, this means that this object is occluding the light source. So the corresponding intersection point lies in shadow and the pixel on the image plane gets rendered in black. Figure 3 shows a visualization of the ray tracing algorithm, which should make the procedure easily understandable.

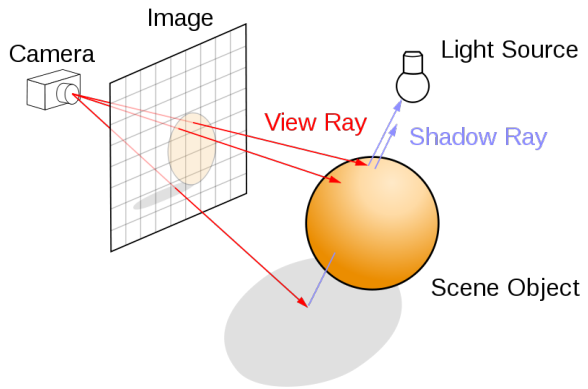


Figure 3: *Illustration of the ray tracing algorithm*
Source: http://www.wikipedia.com

Local illumination In real life, if we see an object, the appearance of a each single point of that object is depends on two factors: the amount of light that arrives at that point, and the that light gets scattered in that point - more concretely from the amount of light that gets scattered into our eye.
Transferred to rendering, this process sums up to the question: "How much light arriving at an intersection point is reflected back to the camera along the corresponding ray?" The direction and distance from which the light is arriving can be determined easily from the position of the intersection point and the position of the light source (which is already provided in the scene description). How light is scattered is more complex: it depends on the incident angle of the light and the object's material.

Different materials scatter light in different ways. For example, diffuse materials scatter light almost equally in all directions while glossy materials reflect light in a certain direction, depending on the incident angle. That is the reason why a material like unlacquered wood appears to be matte from all viewing directions while materials like plastic have strong highlights which let them appear shiny. Firgure 4 illustrates reflection behavior of different materials.

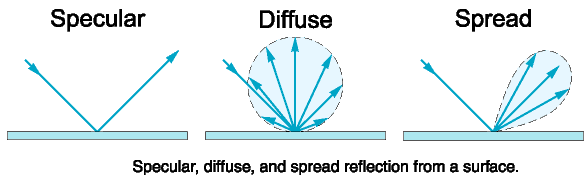


Figure 4: *Light reflection depending on material properties. Perfect mirrors reflect the incoming ray into perfect reflection direction only. Diffuse materials scatter light into all directions equally. Specular surfaces scatter the light in one direction.*
Source: http://library.thinkquest.org/26162/howbehav.htm

To model this interaction between light and materials, so called local illumination models have been established. The simplest one (because it uses the minimum information necessary) is the Lambertian model for rendering perfectly diffuse surfaces. The idea is that the more orthogonal light arrives on a surface, the more efficiently those rays will illuminate the surface. The relation between the incident angle of a single light ray and the normal in the point on the surface to be shaded is expressed by $(\cos(\theta))$, which can be calculated by taking the dot product of the normal and the direction to the light source. By multiplying with the objects color and the lights intensity, the final color can be determined. If more than one light is present in a scene, the contribution of each source is evaluated and summed up.

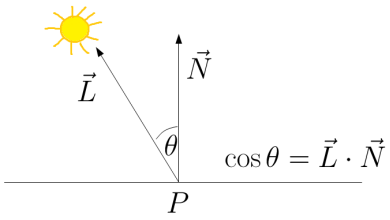


Figure 5: *Lambertian reflection model*

There are a lot of different illumination models which are able to simulate a greater variety of materials, like the Blinn-Phong model, which is able to simulate shiny surfaces [Phong 1975]. This model takes the Lambertian model as basis but adds a term for specular highlights by considering the light reflected back to the viewer as if a perfect mirror would do. A shininess factor then allows to make materials appear more or less shiny. More sophisticated models are able to simulate rough surfaces or surfaces with anisotropic highlights, but the description above should give a sufficient overview of how illumination models generally work [Torrance and Sparrow 1992] [Ward 1992].

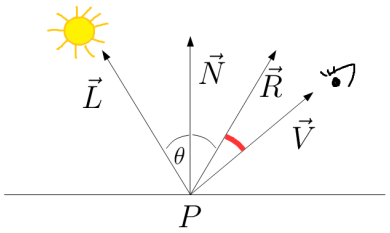


Figure 6: *Phong model. Incoming light is reflected along the normal as it would happen with a perfect mirror. The incident angle and the exitance angle are of the same size. The appearance of a highlight depends on the angle between the direction of the reflected light and the viewing direction.*

Recursive ray tracing Computing the illumination at an intersection point by considering local illumination only does not yet lead to a physical correct simulation of light. Local models are only a simplification of what happens in real life. The benefit of simplifying is that computation time can be saved, thus local illumination models are widely used where graphics should run in real time, e.g. video games. To establish a more accurate model, one crucial observation is that in nature, light rays do not only illuminate an object if they hit it. Instead, they interact with that object: they can get absorbed, reflected or refracted. The kind and degree of interaction depends on different factors, like the material properties, and also on properties of the light ray (for example its wavelength).
After the first hit, the light ray continues traveling according to the physical laws of reflection and refraction. The light propagates until it hits the next object, where it gets refracted, reflected or absorbed again, and so on. Whitted [1980] improved the simple Ray Tracing algorithm by taking this recursive nature of light propagation into account. This modification added more realism to the rendered 3D scenes: for the first time, perfect mirror and glass surfaces could be

easily computed. At an intersection point with a specular or transparent object, new rays in the direction of perfect reflection and refraction are created and traced recursively. The recursion itself can be stopped after a predefined amount of bounces.

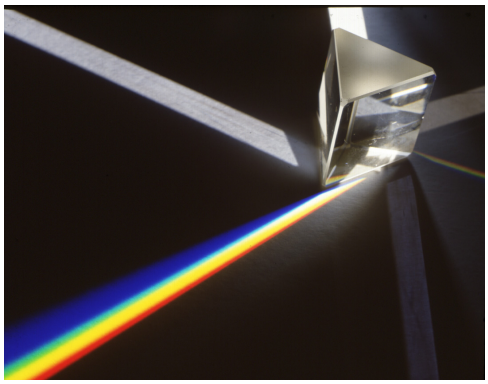
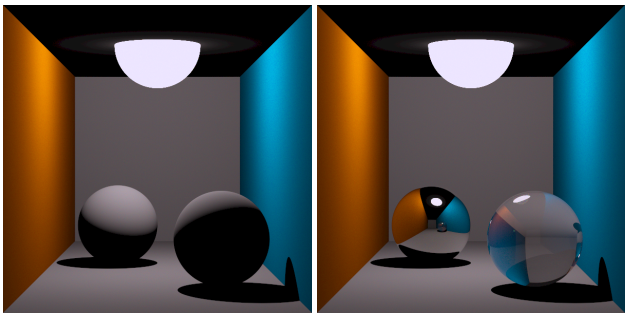


Figure 7: *Effect of dispersion demonstrated with a prism.*
Source: http://feigel.files.wordpress.com

The direction of reflection is easy to determine when handling specular surfaces: The ray gets reflected along the normal, the exitance angle is as big as the incident angle. The calculation of refraction is bit more complicated and requires the application of Snell's law. When light enters a medium (for example, when it travels from vacuum into glass), it changes the direction it travels depending on the density of the new medium. This happens because the light gets "slowed down" when entering a denser medium. The index of refraction (IOR) is a numerical value that describes how much slower light travels in that medium in relation to its speed in vacuum. When the materials in a scene description provide the material's IOR, physically correct refraction can be simulated. As mentioned above, the refraction direction also depends on frequency of light. Different frequencies are refracted more or less strongly, causing an effect called dispersion - white light gets "split up" and the different wavelengths are visible as colored spectrum (as pictured in figure 7). For simulating this effect, the wavelengths have also to be taken into account when programming a renderer.
When hitting a transparent object, both reflection and refraction of the light ray occurs. The amount of each effect is dependent from the incident angle and can be be calculated by the Fresnel equations.



(a) *Ray Tracing* (b) *Recursive Ray Tracing*

Figure 8: *Comparison between different algorithms. Rendering the mirror and glass balls as seen in figure (b) is only possible with recursive ray tracing.*
Source: http://www.wikipedia.com

Global illumination Whitted's model still uses simplifying assumptions to save computation time. For example, only rays in the direction of perfect reflection or refraction are traced. Also, although the light rays get traced recursively, still only direct illumination from a light source is taken into account when computing a color for an intersection point. In real life, objects get not only directly illuminated by a light source, but also indirectly illuminated by other objects. When for example a blue object is positioned near a white wall, light rays that get reflected cause a subtle blue hue on the wall. This effect is called color bleeding and may be very subtle, but adds a lot of realism to a rendered scene. Various other effects like caustics or shadows with soft edges can only be achieved when considering indirect illumination. To do that, Whitted's concept has to be generalized: For every intersection point, light from all incoming directions (not only from direction of the light sources) would have to be taken into account when calculating its color. This would lead to the following equation that would have to be solved for every intersection point:

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_{S^2} \mathrm{f}(\mathrm{p},\omega_\mathrm{o},\omega_\mathrm{i})\,\mathrm{L_i}(\mathrm{p},\omega_\mathrm{i})\,|\cos\theta_\mathrm{i}|\,\mathrm{d}\omega_i \quad (1)$$

Rendering equation Equation (1) is also known as the rendering equation and was first formalized by Kajiya [Kajiya 1986]. The equation basically means that for every intersection point, the light a viewer observes is the sum of light that gets emitted in the cameras direction from that point plus the light that arrives at that point from every direction and gets reflected in the cameras direction.

The first term, $L_e(p,\omega_o)$ is the light emitted in the viewers direction from the point p. This term is nonzero if the point is a light source.

The next term, $f(p,\omega_o,\omega_i)$, is the Bidirectional reflectance distribution function (BRDF). It evaluates the amount of light that gets reflected in point p to the viewers direction depending on the direction of an incoming light ray. The BRDF can be seen as a more general model of local illumination models. Or, vice versa, illumination models like the Phong model are special-cases of a BRDF.
The BRDF takes three input parameters: the incoming direction of the light, ω_i, the point p on a surface, and the outgoing direction to the camera, ω_o. Physically based BRDF's are reciprocal (that means, incoming and outgoing direction can be permuted, but the result of the BRDF stays the same) and energy conserving (no more energy can leave a surface than has entered). BRDF's can be derived in different ways, for example by using measured data, phenomenological models, simulation, physical optics or geometrical optics.
For the transmittance of light, a Bidirectional transmittance distribution function (BTDF), can be modeled analogically. With the help of BRDF's and BTDF's (sometimes also referred as BSDF when both of them are meant, the "S" stands for "scattering") it is possible to simulate a great variety of different materials.

$L_i(p,\omega_i)$ is the amount of incoming light from a certain direction in point p.

$|cos\theta_i|$ accounts for the incident angle of light in relation to the object's normal in point p. Light arriving at a high angle, for example 90 degrees to the surface normal, illuminates this surface more than light arriving at a steep angle. The cosine term takes this into account by providing a weighting dependent on the angle: for 90 degrees, the cosine function returns 1. The steeper the angle, the smaller the value of the cosine function becomes.

The integral means that all incoming light directions over the hemisphere S^2 are taken into account to evaluate the net illumination for the point p.

Monte Carlo integration When looking at the rendering equation, the term we want to evaluate - $L_o(p,\omega_i)$ - itself is part of the in-

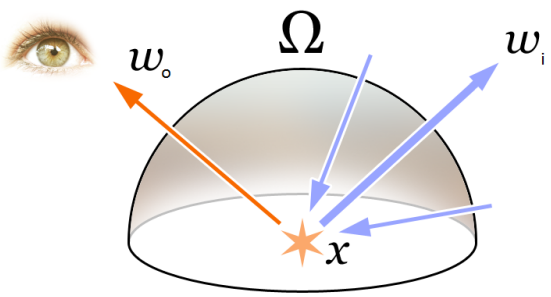


Figure 9: *Rendering equation visualized for one point. The incoming light ω_i arrives at point x. The BRDF for the material of the surface on which x lies, determines which amount of light is scattered into direction of the viewer, ω_o. This process has to be evaluated for all incoming directions over the hemisphere Ω.*
Source:http://www.wikipedia.com

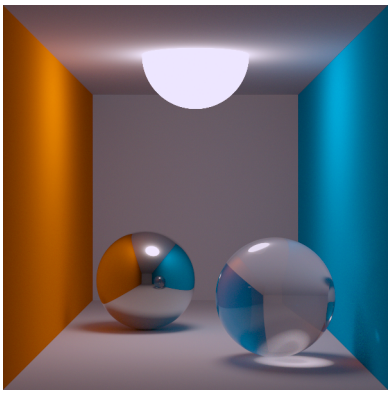


Figure 10: *Image rendered with global illumination. Soft shadows and soft orange and blue hues on the ceiling resulting from indirect illumination are noticeable.*
Source: http://www.wikipedia.com

tegrand. This means that the calculation for every point p requires information from all the other points in the scene which in turn also still have to be evaluated. Thus, solving this integral analytically is not an option.
In practice, the solution has to be calculated by using numerical integration techniques like Monte Carlo Integration. In general, Monte Carlo algorithms are a class of algorithms that make use of random numbers. A characteristic of Monte Carlo Algorithms is that if they are run several times with the same input data, they give different results depending on the random numbers used in a run - however, on average, they give the correct result. When using Monte Carlo techniques for numerical integration, the integral gets estimated by sampling the integrand at random positions and averaging the sum of samples taken. The benefit of Monte Carlo integration in comparison to other numerical methods is that the convergence rate with a carefully designed sampling technique is independent of the dimensionality of the integral, thus making it applicable for the high-dimensional integrals occurring in rendering.
Applied to rendering, the integral that has to be evaluated is the amount of light from all incoming directions in point p. Instead of considering all directions, random samples over the hemisphere around p are made, the light contribution from those directions is evaluated and then averaged and multiplied with the BRDF/BTDF and a cosine term to get the final result. A downside of Monte Carlo integration are artifacts that appear in form of noise, because some pixels are estimated either too dark or too bright. Taking more samples would increase image quality, but the overall convergence would be slower - to reduce the error by a half, four times as many samples have to be made.

Increasing performance One strategy to achieve a better image quality without increasing the number of samples taken is importance sampling. Instead of sampling the integrand at completely arbitrary positions, it is more efficient to use a distribution for the samples that is similar to the integrand. In this way, it is more likely to sample the integrand at positions where the function has high values. When considering the half sphere around point p, it would be more efficient to take samples in directions that are not almost parallel to the equator. The angle between the surface's normal and those directions would be almost 90 degrees and cause the cosine term to be almost 0, thus making the contribution of those samples very low. Other factors that could be considered when choosing a distribution for sampling could be the BRDF and BTDF. On the other side, when choosing a poor probability density function for the sampling, the noise in the result picture could get even worse.
Another strategy to increase convergence speed is Russian roulette path termination. The longer a path is traced, the smaller the contribution to the final color after each bounce gets. Russian roulette tries to randomly terminate paths when their contribution gets too small. A simple termination would of course introduce bias because the integrand gets underestimated every time, so Russian roulette works with a termination probability. If a path gets not terminated, it is weighted by a term that accounts for all the samples skipped before, guaranteeing that a correct result image is achieved.

Rendering algorithms By the means of Monte Carlo Integration, a scene can be rendered with global illumination which is capable of simulating indirect lighting and thus allows the rendering of effects like the previously mentioned color bleeding or caustics. When the materials used for objects in a scene are also modeled with physical accuracy (i.e by considering their Index of Refraction), wavelengths of light rays and other properties that occur in nature are also considered, a physically based renderer can be designed. Rendered images will look indistinguishable from reality, but computational cost is high and results in render times that can last from hours to days.
Several approaches to enhance image quality and to minimize render time were made in the recent years, leading to the development of different algorithms for rendering. They will be presented in the next paragraphs, and an outlook on the newest developments is also be given.

Before going into further detail, it should be mentioned that global illumination algorithms can roughly be distinguished by the property of being biased or not. An unbiased algorithm gives a correct result after an infinite amount of samples. No estimation errors are made while rendering (for example, a systematical over- or underestimation of the amount of incoming light). This means, after an (infinite) amount of samples, the procedure surely converges to the correct image. Because of that, a scene can be rendered on different computers at the same time and later the resulting images can be merged into a better, final picture. This is not the case when using a biased algorithm.
Another property of an algorithm is consistency. When using a consistent algorithm for rendering, it is guaranteed that a longer render time leads to a better result. This is not the case with inconsistent algorithms, where a longer render time can lead to a worse result. The choice for the rendered images provided in this paper fell on two unbiased algorithms, (Bidirectional) path tracing and Metropolis light transport.

Light Path notations To specify what interactions a light ray made before finally reaching the eye or camera, Heckbert introduced the

light path notation in 1990 [Heckbert 1990]. It uses four symbols: L (Light source), S (Specular), D (Diffuse) and E (Eye). The notation for a path that comes from the light source and is reflected by a specular surface directly into the eye would be: LSE. Multiple bounces of the same type can be marked by a "*", either one type or the other can be marked with a "|". Not every algorithm is capable of rendering every light path that could occur in a scene. For example, a non-recursive ray tracer is only capable of rendering LDE paths. Examples for possible paths and their notations are depicted in figure 11.

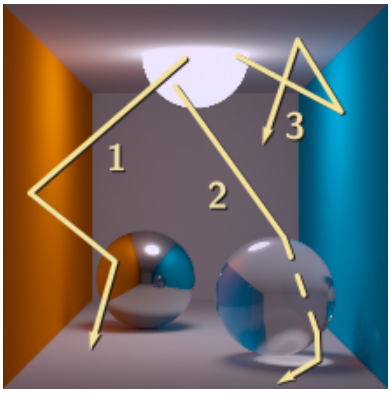


Figure 11: *Examples for the Heckbert path notation. 1)LDSE, 2) LSSDE, 3) LDDE.*
Source: http://www.wikipedia.com

Path Tracing Path tracing was first introduced by Kajiya in the same paper he presented the rendering equation. Basically, it is an extension of recursive ray tracing that tries solving the rendering equation by making use of Monte Carlo integration as described above. Instead of sampling only perfect reflection and refraction directions when a ray intersects an object, random directions are sampled for further tracing and their contributions are then weighted by a BSDF. Each time a light source is hit, its contribution is added to gain the estimated color for corresponding pixel on the image plane.
The major drawback of a path tracer is that a path has to hit at least one light source in order to collect light contribution to the scene - otherwise no contribution is made at all. This leads to slow convergence time in scenes with small light sources, because the probability to randomly hit them is small. Many paths constructed will return no energy of light, resulting in a black pixel on the image. Scenes which make use of point lights only remain totally black, because the probability to hit an infinitely small, zero-dimensional point by randomly selecting directions is zero. The same is the case for highly specular BSDF's - randomly sampling the perfect reflection direction is very unlikely.
"Next event estimation" tries to overcome this problem by evaluating the local illumination at every diffuse intersection point along a path and then adding this contribution to the radiance estimated with Monte Carlo sampling. If a light source is hit directly by a random diffuse bounce, it is not considered - because the next event estimation has already included that light's contribution. Using next event estimation makes it possible to treat direct an indirect illumination separately.
Like importance sampling and Russian roulette termination, this is another way to increase performance. It can be used additionally to those two methods.

Bidirectional path tracing Although many improvements can be made, there are still some difficulties with conventional path tracing can not overcome easily. For example, if a room is illuminated only indirectly by a light source which is partially occluded, it is very hard for any randomly cast ray to reach the light source. Even with next event estimation it is unlikely to cast a ray for considering direct illumination from that light source. To handle such scenes, Bidirectional path tracing has been developed. Instead of sending out rays from the camera only, rays are also cast into the scene from the light sources positions. The endpoints of those paths are then connected if they are mutually visible. (Mutual visibility can easily be tested by casting a shadow ray between the endpoints and checking if no object lies between them.) For difficult light settings, bidirectional path tracing can give better results than conventional path tracing in the same computation time.
Both Path tracing and its bidirectional variant are unbiased and consistent algorithms which converge to the "correct" image after an infinite amount of samples. Thus, these two techniques are often used to render ground truth images to which the output and performance of other algorithms is compared to.

Metropolis light transport The name of Metropolis light transport comes from the utilization of a certain type of Monte Carlo sampling, the Metropolis-Hastings algorithm. As with "conventional" Monte Carlo sampling, an integral is evaluated by drawing random samples from the integrand. Instead of drawing them at arbitrary positions, the distribution of the random samples is made so that the probability density of a sample is proportional to its magnitude. This is remarkable, because this means when computing an integral, optimal importance sampling can be made without knowing the concrete form of the integral. The algorithm is based on a Markov Chain, and therefore chooses the sample of the next iteration depending solely on last sample. A sample near to the last one is made and accepted with a certain probability - the bigger the value of the sample is in relation to the last sample made, the higher is the acceptance rate. In this way, regions of the integral with large magnitudes get "explored". To avoid that the algorithm gets stuck at local maxima, mutations are made so that all non-zero regions of the integrand have an positive chance to be sampled.
This algorithm can be applied to a path tracer: Once a path to a light source is found in a scene, it would be a waste of information to discard it. Instead, it could be used as basis to construct other rays that surely add light contribution to the final image. By making small modifications to a once found path, it is possible to explore the space of nearby paths more thoroughly with less samples then with (bidirectional)path tracing.
Metropolis light transport can be thought of as a improvement of path tracing by equipping it with an effective sampling method. It is especially useful for difficult light transport situations, for example when light enters into a room only through a small opening. It should be mentioned that depending on the scene, metropolis light transport does not always perform better than path tracing. For simple scenes, path tracing is able to process more samples per pixel which results in a better image. The original Metropolis light transport algorithm was introduced by Veach and Guibas [1997] and was later simplified by Kelemen et al. [2002].

Photon Mapping An example for a biased algorithm is photon mapping which was first introduced by Henrik Wann Jensen [1996]. Other then the previous methods, the key idea is not to follow light paths but light "particles", also known as photons. The rendering process is split up into two steps. First, photons are shot out of the light sources into the scene. Those photons get reflected or refracted like the light rays in the previous methods, but when they finally hit a diffuse surface, they get recorded into a photon map. In the second step, the final image gets rendered by interpolating the recorded values. Photon mapping is an efficient technique when rendering caustics, but storing photon maps is very memory intensive (the storage needed grows with the number of photons

used) and the result images are not ”correct” because the final values are achieved by averaging blurred, interpolated photon maps. The algorithm would only be consistent if an infinite amount of photons would be used - which is physically not possible because of the limitations in memory.
(Stochastic) Progressive Photon Mapping [Hachisuka and Jensen 2009] tries to overcome this problem by regularly calculating a new photon map during the rendering process. This makes details in result images look less blurred than with conventional photon mapping.

Recent developments The above described techniques may be able to render a variety of effects, but some problems remain unsolved. One is the rendering of SDS-type light paths (see figure 12 for an example), where a ray of light gets transmitted or reflected by a transparent or specular surface, bounces off from a diffuse surface, arrives at a specular surface again and hits the eye. Randomly sampling such a path is very unlikely with the previous methods because of the diffuse surface. It is very unlikely to sample an outgoing direction which creates a ray hitting the transmissive surface exactly in that point where it gets refracted into the light source (this probability can be even exactly 0 depending on the material models used). The same applies for sampling a random direction that hits the specular surface in exactly that spot that reflects the ray into the camera. A normal path tracer has virtually no chance to sample such a light path, and bidirectional path tracing also fails to capture this phenomenon, because merging paths which hardly can be found does not increase the change of sampling such a light path. New ideas to develop algorithms that work regardless of the properties of the scene to be rendered were made, such as Vertex Connection and merging [Georgiev et al. 2012] and path space regularization[Kaplanyan and Dachsbacher 2013]. The first method combines bidirectional path tracing and photon mapping, and tries to exploit the benefits of both algorithms depending on the type of light path to be sampled. The second method is based on MLT, but treats the difficult paths with relaxing criteria. For example, a specular BRDF can be treated as diffuse to be able to sample a SDS path. This introduces bias, but only for those specific paths - the algorithm is also equipped with a contraction technique which ensures that the bias vanishes in the limit.

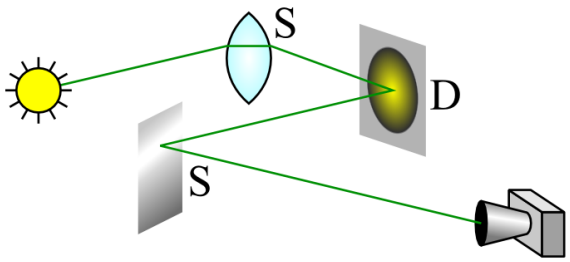


Figure 12: *An example for an SDS path.*
Source: http://www.reedbeta.com/

3.2 LuxRender

LuxRender is an open-source software renderer based on Pharr’s and Humphrey’s physically based renderer PBRT [2010], which was developed for educational and academic use. Figure 13 shows the basic architecture of PBRT. The Sampler provides the SamplerRendererTask with random samples for BRDF sampling. With that sample, the camera then constructs a ray towards the image plane for the next pixel position and passes it to the Integrator. The integrator calculates the radiance carried along that ray. The collected radiance then gets saved on the film.

LuxRender is a stand-alone renderer and not a modeling software. Thus the creation of scenes and models has to be done in other software, and then they have to be exported for rendering.

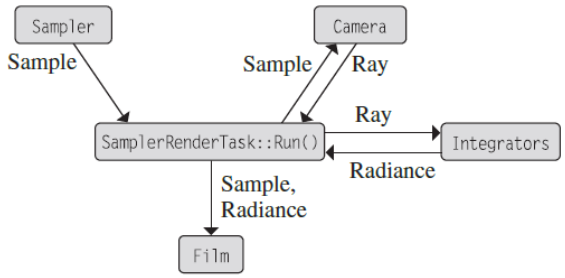


Figure 13: *Basic architecture of PBRT.*
Source: Pharr and Humphreys, 2010

In 2007, programmers adapted the original source code to make it available for artistic use. [LuxRenderProject] *LuxRender* implements a lot of different rendering algorithms and provides a long list of features such as different material types for objects, post-processing effects, HDR rendering, etc. The implementation presented in this paper works on level of the ”Film” stage depicted in figure 13 and makes use of a feature called *Light Groups*.

When modeling a scene with several lights, those lights can be associated with a light group. An arbitrary number of lights - also only a single one - can belong to such a light group. During the rendering process, the light contribution of each group is saved in a separate buffer. Every group also has a intensity and a color temperature, which can be controlled by two parameters. This enables to change the initial light settings in a scene while the rendering is still in process or after it is finished. The user can modify those parameters in the GUI.

3.3 Genetic algorithms

Genetic algorithms belong to the greater class of evolutionary algorithms, which in general work as meta heuristics for optimization problems. The basic idea is inspired by the process of natural evolution and tries to imitate its processes in order to find solutions for search and optimization problems. The theoretical groundwork for genetic algorithms was laid by Holland [1992].

A minimal genetic algorithm consists of a set of possible solutions (called ”population”) and a fitness function which evaluates the quality of a single solution. Those initial solutions are altered by applying genetic operators, and their fitness is evaluated again. The goal is to find better and better solutions by continuously applying the genetic operators. A single solution in a genetic algorithm is also called ”chromosome”. Each chromosome of the population can be modified by the genetic operators ”crossover” or ”mutation”. In this way, new ”offspring” solutions are generated, which may be better solutions for the problem the algorithm is trying to solve or optimize. In order to apply modifications by genetic operators, the solutions have to be encoded in a way that they can be processed computationally. This is often done by encoding values as a string or storing them in a vector. The type of encoding depends on the application - for binary problems integer encoding in form of zeros and ones may be sufficient, while more sophisticated problems may use float values or other types.

Crossover The crossover operator mimics the exchange of genetic information between two chromosomes. It works by splitting up two chromosomes at an arbitrary index and recombining their sub-parts to form a new solution (also called offspring). For example, making a crossover operation on $\vec{c_1} = [0,0,0,0,0]$ and $\vec{c_2} = [1,1,1,1,1]$ at the third index would result in $\vec{c_3} = [0,0,0,1,1]$. Choosing only one index to split up the chromosomes is not mandatory, it is also possible to choose more indices. Another variant would be "parametrized uniform crossover", by which a crossover happens at every index with a certain probability. Crossover operations help to explore a big part of the search space instead of getting stuck to local optima, because in comparison to mutation they allow a bigger change in a chromosome's values. The implementation of the crossover operator can turn out to be tricky if certain constraints have to be fulfilled to generate a valid solution. For example, when trying to solve a permutation problem (a famous example would be the traveling salesman problem), a random crossover operation could lead to an invalid solution because one of the numbers is missing. Consider a crossover between $\vec{c_1} = [1,2,3,4,5]$ and $\vec{c_2} = [3,5,4,1,2]$ at the third index. The result would be $\vec{c_3} = [1,2,3,1,2]$, where the numbers 1 and 2 appear twice, but 4 and 5 are missing. One solution would be to forbid such crossovers or to design the process so that the first chromosome gets filled up by the missing values in the order they occur in the second chromosome.

Mutation The mutation operator randomly chooses a single value in a chromosome and changes it. This change could be for example a bit flip, or the addition of a random value. For example, the chromosome $\vec{c_1} = [1,0,1,0,0]$ could be mutated by a bit flip at index 2, resulting in $\vec{c_2} = [1,1,1,0,0]$. Mutation also avoids that the population gets stuck to a local optimum, but does this in a less "radical" way than crossover does, because only one index is affected in each iteration step.

Fitness After the genetic operators have been applied, a fitness function has to evaluate the "quality" of the newly created solutions. Finding a suitable fitness function is an important step when programming a genetic algorithm, and its design is very problem-specific. Thus, a general "recipe" for a fitness function does not exist. An important property is that its result should be a representation of the quality of a solution which allows the comparison of the chromosomes against each other. This makes it possible to clearly determine which solutions are (currently) the best.
A simple problem with an example for a fitness function would be the following: When the overall goal is to minimize the number of 1's in a binary vector, a suitable fitness function could be the sum of all of its values. The best solution would be the one with the smallest fitness value among all vectors.

Selection When all the chromosomes of the population have been processed, or the maximum number of offspring has been created, a selection process chooses the best solutions, from which new "offspring" is created. Different selection strategies have been developed. For example, an approach would be to rank the solutions by their fitness, and only the fittest chromosomes are selected. This could lead to "premature convergence", which means that the search space gets not explored thoroughly because of a fixation to chromosomes with a good fitness value from the begin of the search. More sophisticated methods try to overcome this problem by statistically choosing the chromosomes by mapping their fitness values to probabilities.

After selecting the "parents", another step of evolution by the genetic operators is applied to form the next generation. This cycle is repeated until a satisfying solution is found or a certain number of processing steps (also called "generations") has been reached.

The above description depicts how a very basic genetic algorithm works. Several refinements and variations can be made. One of them is the principle of elitism, which keeps the best solutions of a generation unaltered when applying the next mutation and crossover steps. This prevents that the best solution(s) of a generation do not accidentally get lost by further applying genetic operators. Elitism was first introduced by De Jong [1975], who found out that guaranteeing the "survival" of temporary best matches improves the performance of a genetic algorithm.

The strength of genetic algorithms is that they are able to explore huge search spaces. Another property of genetic algorithms is that they can be applied to a broad field of problems apart from optimization, for example automatic programming, machine learning, modeling of economic processes or procedural production of images and textures. [Mitchell 1998] [Goldberg 1989]

With those strengths come also the downsides of genetic algorithms. Because they are applicable to a variety of problems, there are cases in which they perform good, but also cases in which they perform worse then other algorithm classes. A general answer to the question if a genetic algorithm should be applied to a problem can not be given. Also, the concrete implementation is highly dependent from the problem the algorithm is intended for. This affects not only the type of encoding of chromosomes, but also the fitness-, mutation-and crossover strategies. Finding reasonable parameters for the population size, generation count and the probability of crossover and mutation is also a problem-depended task. In research, various settings have been used, but no "general tendency" of what settings work well under which conditions has been reported. A common practice is to use settings that performed well in other work or to simply find values that work well by trial and error.
Another problem caused by the heuristic nature of genetic algorithms is that even after a lot of iterations it is not guaranteed that the real the global optimum for a given problem is going to be found. It is possible that it never appears in a run of the algorithm. Also, two runs with the same initial settings are not likely to produce the same results.

4 Concept

As already explained, rendering images with high physical accuracy is a time consuming process and many efforts have been made to accelerate it. A new approach to achieve a speed up would be to reduce the overall amount of light sources used in a scene. Especially when having many light sources in a scene, eventually the same lighting could be achieved by using less of them when turning some of them off or changing their intensities. Trying out all variations manually is not an option, as it is almost impossible to do in finite time. On the other hand, automatizing this process is relatively simple. An algorithm for this task would have to try out different light settings for a scene and compare the resulting images to an initial image the user wants to achieve. The more similar a new image is to the desired one, and the less light sources are used, the better the solution is.

To determine what a good or similar solution is, some kind of objective quality metric has to be found. The simplest way to evaluate the difference between two digital images would be to subtract them from each other and add up the absolute differences. When adding the amount of lights used for rendering, the resulting number represents the "quality" of a given light setting. A smaller number means a better solution. The algorithm would have to find the settings which result in the smallest number representing the quality of a solution.

Despite the idea itself is very simple, it is crucial to understand that on the other side, the solution space is rather complex. The whole problem can be seen as an optimization problem, where a global minimum has to be found among the quality values of all possible solutions. When trying to reduce the number of lights by simply turning them on and off, a binary problem has to be solved. Although the number of solutions increases with the number of lights used, the total amount of solutions is finite (up to $2^{amount\ of\ lights}$). This method would be sufficient for simple scenarios, where for example a light source definitely has no contribution to the scene. (This could be the case if an artist has created a light source with almost no intensity in the scene, or when a light source gets occluded by some object and suddenly has no contribution to the image at all, etc.) To decide if a proposed solution is close enough to the original image, a user-defined threshold could be set. The algorithm could then calculate the fitness.

In practical applications, such easy scenarios where some lights have no contribution at all will be the minority of cases. To reduce the overall amount of lights, changing the intensity of some of them before turning others of will be necessary. When considering also the light's intensities, each light's parameter could be changed a little to find a new setting which results in similar illumination as the initial one. The solution space in this case varies drastically from the first one: An infinite amount of solutions exists, and each one is a valid image gained with certain light settings. For an arbitrary scene, it is also not clear which light's intensities have to be changed in what way. Exploring this search space without making more constrained assumptions (because the algorithm should work for any scene, not just for those matching some special criteria) is a tricky task. Conquering this problem could be done by using a genetic algorithm, because of its strength to explore such huge search spaces.

When designing a genetic algorithm for this problem, a chromosome would have to store two information: the values for the light sources and the fitness-value of the solution it represents. Storing the values can be done by using a vector. To determine the fitness of a solution, the amount of lights used would have to be calculated first. This is done by summing up all non-zero components of a vector $\vec{X_n}$ (also known as calculating the weight of a vector).

The weight of a vector $\vec{X_n} = [a_1, a_2, a_3, ..., a_n]$ can be calculated in two ways:

$$w(\vec{X_n}) = \sum_{i=1}^{n} a_i \ \ \forall a_i \neq 0, a_i \in \{0,1\} \tag{2}$$

$$w(\vec{X_n}) = \sum_{i=1}^{n} a_i \ \ \forall a > 0, a_i \in \mathbb{R} \tag{3}$$

The first case would be the binary optimization where lights only get turned off or on, the second one is the way to calculate the weight when adapting the lights intensities.

Using only the weight of a vector to determine the fitness of a solution would be not sufficient, because the image quality is not being evaluated at all. A method to determine if a solution is good enough would be to define a threshold, which determines the maximum allowed difference between the initial image and a possible solution image. The solution is only considered as valid if the difference lies below this threshold, otherwise it is discarded. For valid solutions, the fitness gets calculated as the weight of a vector.
This approach basically works, but leaves two problems unresolved. The first problem is that when creating an invalid solution, the algorithm never gets any kind of feedback on how close it was to a valid one. Not having this knowledge of how "close" a solution is makes optimization very hard if not impossible. The algorithm needs a more sophisticated feedback in order to know if the optimization is heading in a good direction. The second problem can be depicted by the following scenario: two valid solutions are available, both of them use three lights out of many. The three lights used in the first solution are different from the ones in the second - which of the two solutions is the better one, when both of them have the same fitness?

To solve both problems at once, the difference between the two images also gets considered when calculating the fitness. First the difference between the initial image and the current solution gets evaluated. This is done by subtracting the images pixel-wise from each other and summing up the absolute differences. This difference gives direct feedback on how close a solution is, and can simply be added to the weight of a solution vector of a chromosome. The overall fitness $f()$ of a solution vector $\vec{X_n}$ gets calculated as depicted in equation 4, where T stands for "target image" (the initial image) and C for "current solution image". The weight $w()$ gets calculated as explained above in equations 2 and 3.

$$f(\vec{X_n}) = p_L * w(\vec{X_n}) + p_D * |T - C| \tag{4}$$

Adding the second term to the overall fitness is also done in integer optimization mode. Two different solutions, that are both below the threshold, could have the same amount of lights turned on, but one of them could still be more similar to the initial image than the other one. By adding the absolute difference, it is guaranteed that the "better" solution gets a better fitness value.

In the equation 4, there are also two additional parameters for weighting the lights (p_L) and the difference (p_D). They act as a kind of quality switch and allow more control on the optimization routine. The contribution to the weight of the vector of each light is multiplied by the factor p_L, so better fitness values are achieved if the algorithm tries to turn off lights instead of finding a solution which has little difference to the initial image. If on the other hand, a result image that's very close to the original is desired, the weight for the difference would have to be set to an appropriate value.

After having defined the fitness function, the general procedure of the algorithm follows the schema of a typical genetic algorithm. In each generation, new solutions are processed and ranked according to their fitness values. The best solutions are kept by the principle of elitism, while the others are modified in order to gain better solutions from generation to generation. Mathematically speaking, the algorithm is searching for a global minimum among the fitness values of possible solutions.

5 Implementation

The next section will bring a description of how the algorithm for the Light Source Cleaner (from now on referred to as LSC) was implemented in *LuxRender*.

The LSC can be accessed by the user via an extra tab that was added to the *LuxRender* GUI. Before starting the algorithm, the user can choose the number of chromosomes and iterations by entering the desired values in the fields "population size" and "generation count". A big number of chromosomes is of course possible, but when using big numbers for both, computation time can take very long. For this problem, good values seemed to be a relatively small

number of chromosomes and a big generation count, resulting in acceptable computation time and results.

The pseudo-code for the algorithm is summarized in algorithm 1.

Algorithm 1 Pseudo-code for the Light Source Cleaner

```
T ← {currentFramebuffer}

// Assign initial light values to each chromosome,
// calculate initial fitness
for i = 1 → populationSize do
    population[i] ← initial LightGroup values;
    chromosome.fitness = WEIGHT(population[i]);
end for

for i = 1 → generationCount do
    for j = 1 → populationSize do
        if RANDOM() == 0 then
            MUTATE(j);
        else
            CROSSOVER(j);
        end if
        // Calculate fitness
        FITNESS(j);
    end for
    // Sort chromosomes by fitness, ascending
    SORT(population);
end for
// First chromosome has best fitness
DISPLAYBESTSOLUTION(population[0]);

function FITNESS(chromosome)
    set light settings to chromosome's values;
    updateFramebuffer;
    C ← {currentFramebuffer}
    chromosome.fitness =
    pL* WEIGHT(chromosome) +pD * |T −C|;
end function

function WEIGHT(chromosome)
    weight = 0;
    for i = 1 → chromosomeSize do
        if chromosome[i] > 0 then
            weight ← weight + 1;
        end if
    end for
return weight;
end function
```

The algorithm starts by copying the image gained with the initial light setting to the array T. Then the population is created by using the initial light scales as values for each chromosome. A chromosome is thus represented in form of a vector of the size of the total amount of light groups used in a scene. Additionally, each chromosome also has a fitness value that indicates how "good" its values are. The initial fitness values are calculated only by taking a chromosome's weight into account (because there is no difference to the target image yet). After randomly applying either mutation or crossover operations on a chromosome, its fitness has to be recalculated.
For evaluating the fitness, the weight of the chromosome and the difference to the target image are weighted by user-set parameters and then added. To gain the difference between the two images, the parameters of the light groups are set to the values of the current chromosome. Then the framebuffer gets updated in order to gain the current solution image, C. This image gets compared to the initial image. In both modes, the luminance of the pixels in both the initial and the target image is calculated by weighting their R,G,B values according to the human visual system. In order to avoid unnecessary calculations, the original image already gets converted to an image of luminance values at the beginning of the optimization. So when comparing the pixel values, only the luminance values of the new image have to be calculated again. There are two comparison modes: "Simple Difference" and "Mean Squared Error". When using "Simple Difference" mode, the absolute difference of the two images is calculated pixel-wise. Those absolute values get summed up and averaged by the total pixel count. "Mean squared error" squares the differences after doing the subtraction instead of taking absolute difference values. This should avoid loosing small features of an image like specular highlights.
After reaching the maximum number of generations, the best solution is displayed to the user (the others are also accessible, ranked by their fitness).

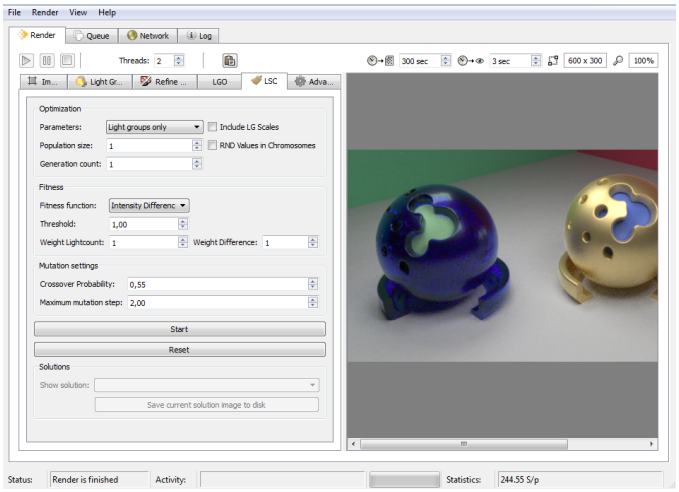


Figure 14: *Screenshot of the LuxRender Graphical User Interface with the LSC tab open.*

The LSC can be used in two modes, which can be controlled by the selection of the check box labeled "Include LG scales". When unchecked, the algorithm is doing binary integer optimization and tries to turn light groups off and on. The chromosomes in this mode are sequences of binary values. The mutation strategy works by choosing a random index of the current chromosome and inverting its value, which means 1 becomes 0 and vice versa. The goal is to completely turn off light sources which have (almost) no contribution to the scene. The fitness function evaluates the fitness of a solution considering a threshold given by the user. If the average error per pixel is smaller then the threshold, the fitness of the solution is determined by summing up the nonzero values in the chromosome and adding the difference between initial and current image. Else, the solution's fitness is set to the maximum integer possible.

The other mode takes the parameter values for each *Light Group* into account by changing their values slightly in every mutation step. This is done by addition or subtraction of a randomly generated small value. How big the mutation steps maximally are can be controlled by the value of the spin box "Maximum mutation step". If for example this value is set to 5, a random value between 0 and 5 will be added or subtracted. Additionally, if a light group is already set to a small value (< 0.8), only a value between 0 and 1 is added and subtracted, to allow only "small" mutations when a value approaches 0.
If a light group gets assigned a value that's almost 0 after a mutation

step, it is set to 0 and thus turned off. The assumption here is that light groups with a low intensity value contribute only marginally to the scene lighting. Thus the probability is high that they can be turned off completely. Without this assumption, it could happen that many light sources get turned down to a very low level but never get turned off completely, which would never reduce the overall amount of lights used in the scene.

In both modes, the crossover strategy chooses chromosomes of the last generation and generates offspring by randomly combining parts of them. The algorithm also works with the elitism method, described in chapter 3.3. The best solutions are kept untouched in each generation and up to 80 % of the "useless" chromosomes get altered by mutation and crossover.
Depending on the mode used, the chromosomes of the initial population are either filled only with 1's when doing integer optimization (assuming that the artist intended to have all light sources turned on in the beginning) or with the parameter settings the artist chose for the light groups when doing float optimization. This guarantees that the best solution is at least a valid one even if no optimization could be found.
When programming a genetic algorithm, it is common to use random values for the chromosomes of the initial population. This was also tried for the LSC. When checking the box "use RND values", every 5th chromosome gets filled up with some random values. In practice, using random values often generates invalid solutions in binary mode, and ends up with solutions that were almost never converging. When using the mode that adapts the light group's scales, replacing some values of a chromosome with random ones instead of the initial ones sometimes led to a good result after a run for only few iterations.

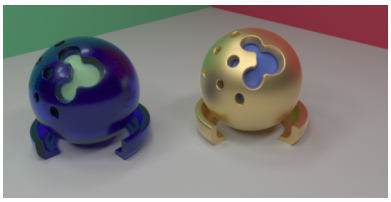
The GUI also provides the two weighting factors ("weightLightcount" and "weightDifference") which allow additional control of the two components of the fitness function. A high value for weightLightcount would prefer to turn off lights rather than considering image quality. The higher the value used for weightDifference is, the better the image quality will be in the end, but lights are less likely to get reduced.

6 Results

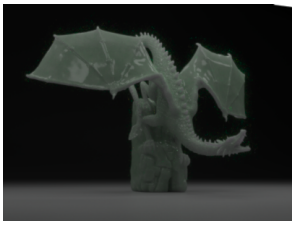
The following section consists of two parts, the first one are the results for the empirical test study to verify the assumption that rendering with less light sources increases overall rendering speed. The second part presents the results achieved when using the light source cleaner (further referred to as LSC) on two specially designed test scenes. The scenes were all rendered on a computer with an Intel Core i7-2600K CPU @ 3.40GHz (8 (logical) CPUs), 16 GB DDR3 RAM, and an NVIDIA GeForce GTX 560 Ti, 1 GB.

6.1 Test scenes with many/less lights

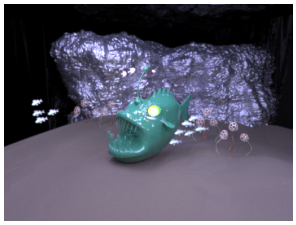
In order to verify the assumption that render time can be saved when using less lights in a scene, test renderings of five scenes of varying complexity were done. The test scenes where designed to make use of up to 47 light sources. In the scene "LuxBalls", three spheres with the LuxRender logo embedded were placed on a gray surface. Each sphere was assigned either a glass, metal or glossy material to have various effects visible in the rendered scene. The scene "Dragon" is featuring a green dragon made out of a diffuse, matte material which stands on a matte, gray surface. This scene is relatively easy to render, because no complex light interactions (like caustics, etc.) occur. The "Corridor" scene is the most complex one where light from outside enters through a glass window into a school corridor with a glossy floor. The "Cherry" Scene pictures a cherry that gets thrown into water and produces splashes. The "Watch" scene shows a wristwatch on a wooden table. Each scene was rendered with different rendering algorithms and the "all" light strategy, and twice for every rendering algorithm: Once only with a set of light sources that have a contribution to the lighting of the scene, and once with additional light sources that have almost no contribution to the scene. A ground truth image of each scene was also rendered for one hour.
The resulting images were then compared with the open-source software *imagemagik* using the RMSE-metric. RMSE stands for 'root mean squared error'. It calculates the average error per pixel by pixel-wise subtracting the images from each other, squaring the values, and summing them up before dividing by the total amount of pixels. Table 1 shows that when comparing the RMSE-values for each of the scenes, the scenes where less lights are used always have a smaller difference to the ground truth image than the ones with many lights. With exception of the School Corridor scene, which was rendered for half an hour due to its complexity, the test scenes were rendered for 10 minutes each. The differences may not seem big, but rendering for 10 minutes is not a long time either. When rendering the test scenes longer, the difference between using less or many lights gets clearly visible for the human observer, without using a comparison software.



(a) *LuxBalls Scene*



(b) *Dragon Scene*



(c) *Fish Scene*

Figure 15: *Scenes used for testing the algorithm.*

6.2 Results of the LSC

For testing if the algorithm works, three scenes were designed. In every scene, each light used was assigned to a separate light group - otherwise the light sources can not be manipulated individually.

First, the integer optimization mode was tested with the scenes that were already used in the empirical study. In those scenes, there were lights which had surely no contribution to the lighting of the particular scene. The assumption was that the "unnecessary" lights should be easy to determine, so when running for enough generations, the algorithm should be able to detect all of them and turn them off. This worked fine for the tested scenes and should also work for any arbitrary other scene where lights are used which have (almost) no contribution.

For the testing the float optimization mode, three scenes were modeled and tested. Two rather simple cases were constructed to show

Scene	Balls		Dragon		Corridor		Cherry		Watch	
Lights	30	30+17UL	20	20 + 17UL	10	10 + 17UL	31	31 + 17UL	11	11+17UL
Pathtracing	1040.44	1148.54	224.41	250.42	4918.52	5250.92	2012.82	2238.13	1246.18	1489.03
Bidirectional	1218.10	1333.16	257.46	275.04	2957.23	3827.03	2567.93	3115.10	1220.64	1887.13
Metropolis	1209.58	1347.72	283.97	304.10	3103.88	4082.38	2541.94	3031.82	1408.88	2342.89

Table 1: *RMSE Difference of test scenes compared to ground truth image. "UL" stands for "Unnecessary Lights", and indicates how many lights with almost no contribution were placed in the scene. Each scene was rendered for 10 minutes each except the Corridor scene, which was rendered for half an hour due to its complexity. The ground truth images were rendered for one hour. It is clearly visible that the scenes using more light sources have a bigger difference to the GT image.*

that the algorithm generally works. The first one consists of two LuxBalls in front of a wall. Three area lights - two small lights and one big light - were then arranged in the following way: Each one of the small lights is half the size and half the power of the big light, and the lights were positioned so that the two small lights together are covering the big light exactly. Also, the lights are placed exactly at the same height. Figure 16 shows a screen shot of the 3D-view of the scene for better understanding. The assumption was that if the algorithm worked correctly, it should be possible to achieve the same lighting for a scene when turning off the small light sources completely and increasing the intensity of the big lights a bit.
To make the whole scenario not too simple, this basic setup of three lights was copied and pasted into the scene several times, so that 12 of those arrangements (which makes 33 lights in total) are present in the scene.

The second scene features the dragon model again, but this time it gets illuminated by 50 area lights. The area lights are positioned pair-wise on the same position and height with the same light intensity. So in this scene, it should be possible to turn off at least half of the lights when increasing the intensity of the other half. A 6x3 array of those lights illuminates the dragon from top, and 7 lights from the front. The light setup is also rather simple here, but the amount of lights is already high. Figure 17 shows a screenshot again for better understanding.

The third scene which shows an angler fish is the most complex one because it features 100 light sources. The majority are blueish area lights, and there are also point lights hidden behind the big stone wall which have almost no contribution at all. The algorithm should be able to turn off many of the point lights and reduce the amount of area lights also significantly. Contrary to the previous scenes, the area lights are positioned arbitrary instead of being arranged in a special way. This was done to simulate a scene with completely arbitrary lighting as it could occur in film production.

The initial images were rendered with bidirectional path tracing, and the algorithm was run on each of them several times for 100, 500 and 1.000 generations with 15 chromosomes each and different weighting parameters. At the beginning, some runs which last only 100 generations were made several times to ensure that the algorithm works "right" and gives somewhat similar results. The problem is that because of the nature of genetic algorithms, each run can give a different result even when using the same input and the same parameter settings. So a direct comparison of two runs with the same settings is difficult, because the results are likely to be different, but they can't be classified as being "wrong" or "right". Figures 18, 19 and 20 show some of the result images.

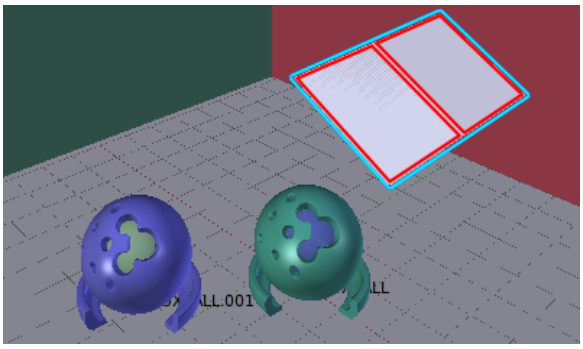


Figure 16: *Example for the light setup in the Luxballs scene. The cyan colored rectangle marks the big area light, while the two red ones mark where the two small area lights are located. The small lights together cover the same area as the big one and are placed exactly at the same height.*

7 Conclusion and discussion

We presented a light source minimization technique to provide a solution for reducing the overall amount of light sources used in a scene by applying a genetic algorithm to a multivariate optimization problem. A definitive strength is the simplicity of the concept and it general applicability. Although for this paper the implementation was done in *LuxRender*, this technique can be implemented in any photorealistic rendering engine as long as there is a mechanism that stores the contributions of light sources at different locations. We demonstrated that our technique works well our test scenes. We note that as we are using unbiased and consistent techniques, it is possible to reduce the number of light sources while the rendering is still in progress, and not after rendering of an image is finished.

The binary optimization mode could be sped up by evaluating beforehand which light groups have an average contribution per pixel that is above the threshold. Those light groups for sure can not be turned off, because of the additive nature of light transport, and turning them off would for sure result in the generation of an invalid solution. Those light groups could be flagged in the initial population, so that the algorithm does not make an invalid mutation or crossover operation on them. At the moment, there is no method in the *LuxRender* API which allows accessing the single light group buffers. This extension to *LuxRender* itself would have to be made, too.

At the moment, the metric used for deciding if an image is below the threshold is a global. Thus, images with local extrema may impose problems due to the omittance of small local features. Using mean squared error metric instead of the simple difference between the pictures could help, but one improvement would be to let the user interactively pre-define which local effects are important and should be kept after optimization.

Our technique is simple to implement, and is capable of yielding a significant speedup in the execution time of the rendering step in difficult lighting scenarios with a vast amount of light sources.

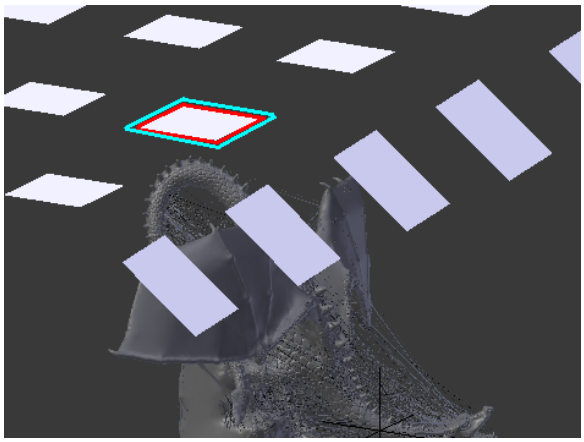


Figure 17: *Example for the light setup in the Dragon scene. The cyan and red rectangles exemplary mark two area lights, which are placed exactly at the same position and have the same intensity.*

References

ANRYS, F., AND DUTRÉ, P. 2004. Image-based lighting design. CW Reports CW382, K.U.Leuven, Department of Computer Science, June.

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ACM, New York, NY, USA, AFIPS '68 (Spring), 37–45.

BOUSSEAU, A., CHAPOULIE, E., RAMAMOORTHI, R., AND AGRAWALA, M. 2011. Optimizing environment maps for material depiction. In *Proceedings of the Twenty-second Eurographics Conference on Rendering*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR'11, 1171–1180.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *In Eurographics 2003*, Blackwell Publishers, 543–552.

COSTA, A. C., DE SOUSA, A. A., AND FERREIRA, F. N. 1999. Lighting design: A goal based approach using optimisation. In *Rendering Techniques*, 317–328.

DE JONG, K. A. 1975. *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD thesis, Ann Arbor, MI, USA. AAI7609381.

GEORGIEV, I., KŘIVÁNEK, J., DAVIDOVIČ, T., AND SLUSALLEK, P. 2012. Light transport simulation with vertex connection and merging. *ACM Trans. Graph. 31*, 6 (Nov.), 192:1–192:10.

GOLDBERG, D. E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GUMHOLD, S. 2002. Maximum entropy light source placement. In *VIS '02: Proceedings of the conference on Visualization '02*, IEEE Computer Society, Washington, DC, USA, 275–282.

HACHISUKA, T., AND JENSEN, H. W. 2009. Stochastic progressive photon mapping. In *ACM SIGGRAPH Asia 2009 Papers*, ACM, New York, NY, USA, SIGGRAPH Asia '09, 141:1–141:8.

HECKBERT, P. S. 1990. Adaptive radiosity textures for bidirectional ray tracing. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '90, 145–154.

HERY, C., AND VILLEMIN, R., 2013. Physically based lighting at pixar. Accessed: 2013-12-04.

HOLLAND, J. H. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA.

JENSEN, H. W. 1996. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, Springer-Verlag, London, UK, UK, 21–30.

KAJIYA, J. T. 1986. The rendering equation. *SIGGRAPH Comput. Graph. 20*, 4 (Aug.), 143–150.

KAPLANYAN, A. S., AND DACHSBACHER, C. 2013. Path space regularization for holistic and robust light transport. *Computer Graphics Forum (Proc. of Eurographics 2013) 32*, 2.

KAWAI, J. K., PAINTER, J. S., AND COHEN, M. F. 1993. Radioptimization: Goal based rendering. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '93, 147–154.

KELEMEN, C., SZIRMAY-KALOS, L., ANTAL, G., AND CSONKA, F. 2002. A simple and robust mutation strategy for the metropolis light transport algorithm. In *Computer Graphics Forum*, vol. 21, Wiley Online Library, 531–540.

LUXRENDERPROJECT. http://www.luxrender.net. Accessed: 2013-12-30.

MARKS, J., ANDALMAN, B., BEARDSLEY, P., FREEMAN, W., GIBSON, S., HODGINS, J., KANG, T., MIRTICH, B., PFISTER, H., RUML, W., RYALL, K., SEIMS, J., AND SHIEBER, S. 1997. Design galleries: A general approach to setting parameters for computer graphics and animation. 389–400.

MITCHELL, M. 1998. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA.

PHARR, M., AND HUMPHREYS, G. 2010. *Physically Based Rendering, Second Edition: From Theory To Implementation*, 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

PHONG, B. T. 1975. Illumination for computer generated pictures. *Commun. ACM 18*, 6 (June), 311–317.

SCHOENEMAN, C., DORSEY, J., SMITS, B., ARVO, J., AND GREENBERG, D. 1993. Painting with light. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '93, 143–146.

SHACKED, R., AND LISCHINSKI, D. 2001. Automatic lighting design using a perceptual quality metric. *Computer Graphics Forum 20*, 3, 215–227.

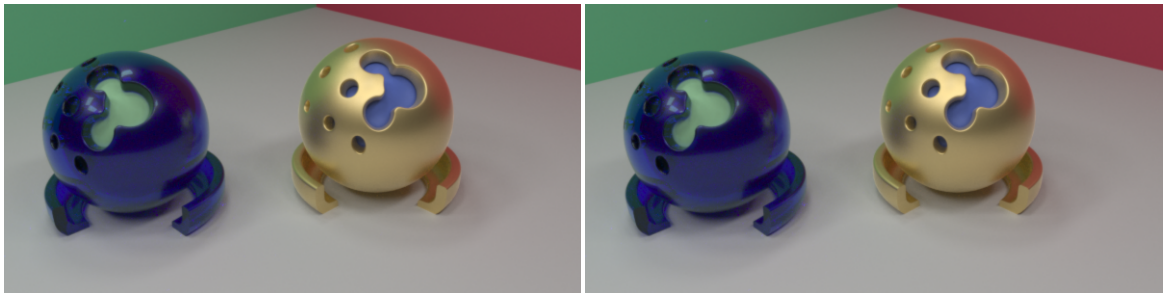
SIMS, K. 1991. Artificial evolution for computer graphics. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, New York, NY, USA, SIGGRAPH '91, 319–328.

TORRANCE, K. E., AND SPARROW, E. M. 1992. Radiometry. Jones and Bartlett Publishers, Inc., USA, ch. Theory for Off-specular Reflection from Roughened Surfaces, 32–41.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press, Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 65–76.

WARD, G. J. 1992. Measuring and modeling anisotropic reflection. *SIGGRAPH Comput. Graph. 26*, 2 (July), 265–272.

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



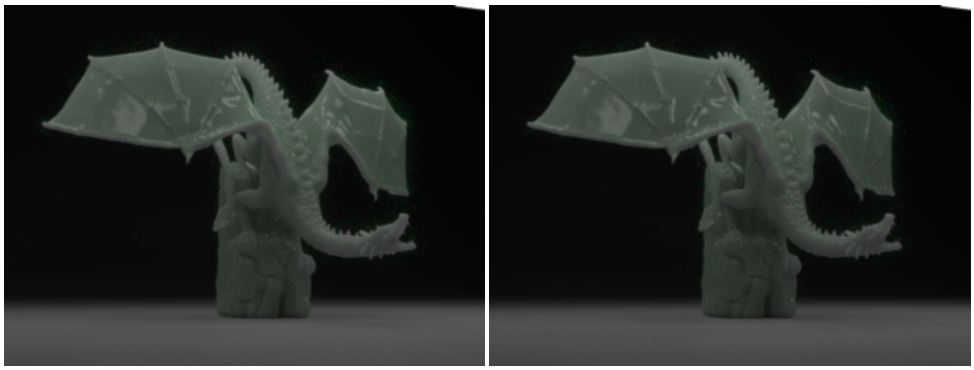
(a) *Initial Image. Amount of lights used: 33*

(b) *Solution after 100 generations. Amount of lights used: 29*

(c) *Solution after 500 generations. Amount of lights used: 11*

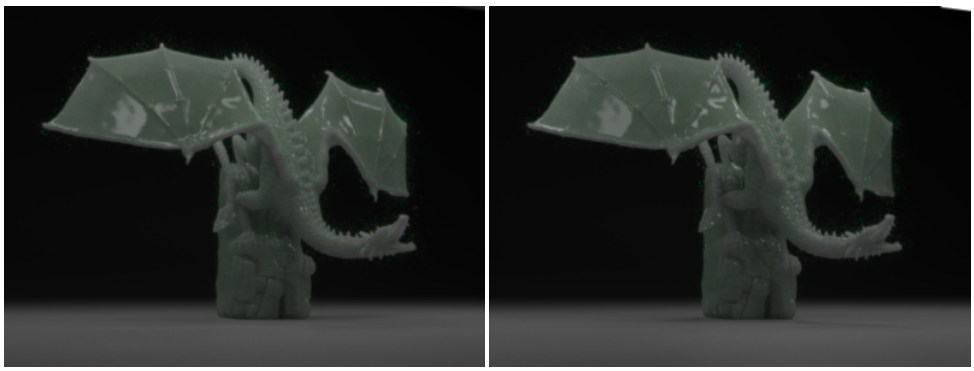
(d) *Solution after 1000 generations. Amount of lights used: 12*

Figure 18: *Luxballs Scene. 33 Lights in total.*



(a) *Initial Image. Amount of lights used: 50*

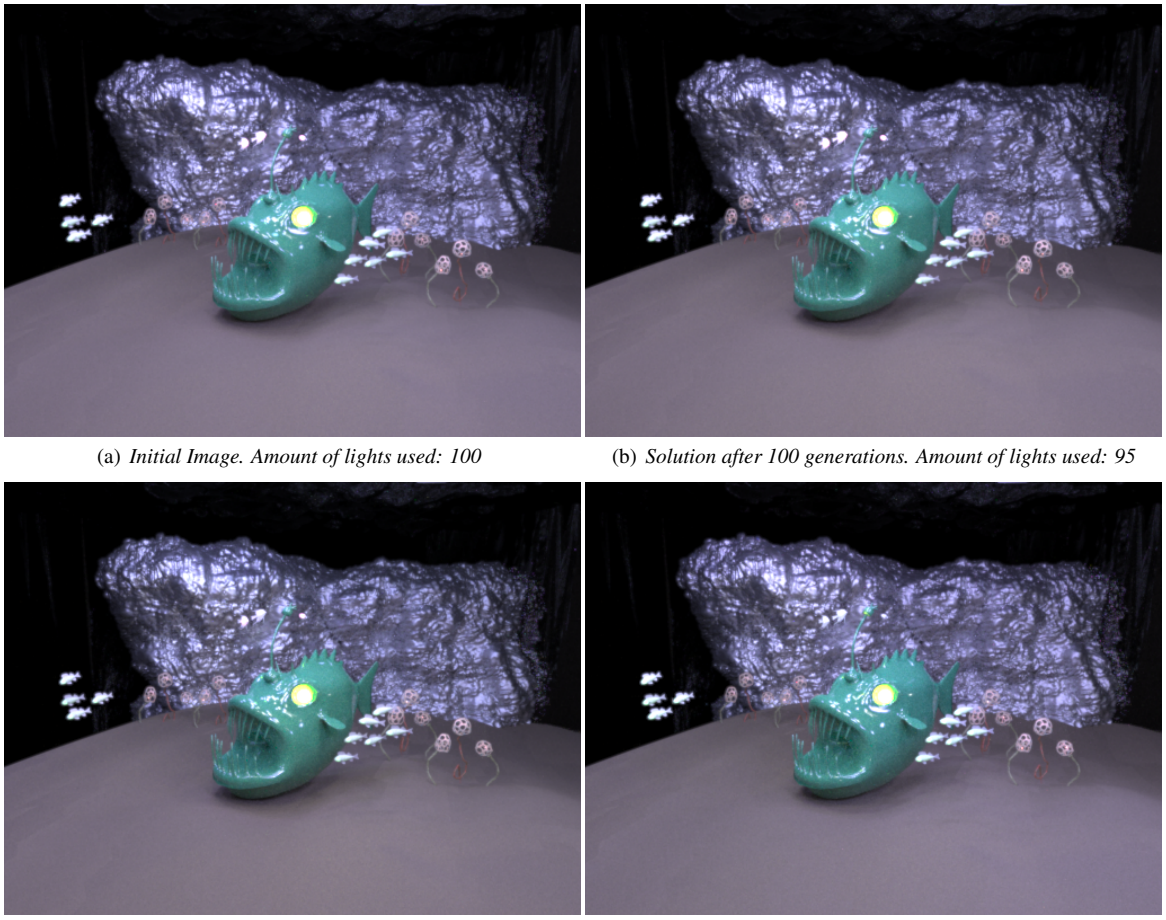
(b) *Solution after 100 generations. Amount of lights used: 41*



(c) *Solution after 500 generations. Amount of lights used:26*

(d) *Solution after 1000 generations. Amount of lights used: 24*

Figure 19: *Dragon Scene. 50 lights in total.*



(a) *Initial Image. Amount of lights used: 100*

(b) *Solution after 100 generations. Amount of lights used: 95*

(c) *Solution after 500 generations. Amount of lights used: 70*

(d) *Solution after 1000 generations. Amount of lights used: 58*

Figure 20: *Fish scene*