

Spectral Mipmapping

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Christian Niederreiter

Matrikelnummer 0726258

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung

Betreuender Assistent: Dipl.-Ing. Dr.techn. Andrea Weidlich

Betreuer: Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer

Wien, 29.09.2011

(Unterschrift Verfasser)

(Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Christian Niederreiter
Eschenweg 4
4800 Attnang-Puchheim

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen, Internet-Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, am 29. September 2011

Christian Niederreiter

Abstract

Full-spectral color rendering capability of global illumination renderers is still in need of improvement, particularly regarding performance, which is a reason why many modelling programs for global illumination are focused on three-component based color representation (e.g. RGB). Full-spectral color representations are the basis for realistic color appearance in realistic image synthesis. Currently, if compared to common three-component rendering, the major disadvantage of full-spectral rendering comes from higher computation costs because of the greater amount of processed information. This thesis investigates promising approaches of spectral color representation in the context of a multi-level color rendering model called *Spectral Mipmapping*, focused on the trade-off between time-efficiency and resulting image quality in terms of perceived color difference. A test implementation in a basic ray tracing renderer is used to evaluate results and make suggestions concerning appropriate code design optimizations and spectral sampling strategies. The influence of compiler optimization and vector instruction use (based on SSE instructions) is discussed.

Kurzfassung

Die Fähigkeit zur Berechnung von Bildern unter Verwendung spektraler Farbrepräsentationen bei Programmen zur Abbildung global beleuchteter virtueller Szenen ist noch immer verbesserungsbedürftig, hauptsächlich betreffend die Berechnungsgeschwindigkeit. Dies ist ein Grund weshalb viele Modellierungsprogramme auf dreikomponentenbasierte Farbrepräsentation wie etwa RGB fokussiert sind. Spektrale Farbrepräsentationen sind die Basis für ein farblich realistisches Erscheinungsbild. Der Hauptnachteil im Vergleich zu dreikomponentenbasierter Farbrechnung (z.B. RGB) besteht im größeren Rechenaufwand aufgrund der größeren Menge zu verarbeitender Information. Die vorliegende Arbeit untersucht vielversprechende Ansätze zur spektralen Farbrepräsentation im Rahmen eines mehrstufigen Modells der spektralen Farbverarbeitung, genannt *Spectral Mipmapping*, mit dem Schwerpunkt auf der Abstimmung von Zeiteffizienz und Qualität des erzeugten Bildes im Hinblick auf den wahrgenommenen Farbunterschied. Eine Testimplementierung in einem elementaren, auf Strahlverfolgung basierenden Bildgenerator wird verwendet, um Resultate zu bewerten und Ratschläge für eine passende Optimierung des Quelltexts und eine geeignete spektrale Abtastung zu geben. Der Einfluss von Compileroptimierung und der Verwendung von Vektorbefehlen (anhand von SSE-Befehlen) wird diskutiert.

Acknowledgements

I would like to thank my supervising assistant Dr. Andrea Weidlich for several ample introductory discussions and especially for voluntarily supporting my work after her change from the university to a private corporation, and I thank my supervising professor Dr. Werner Purgathofer for his organizational effort that arose from my master's thesis during its becoming.

Many thanks to my friends and family for their patience regarding my long-standing inavailability during critical periods of my work.

Contents

1	Introduction	11
1.1	Domain Definition	11
1.1.1	Realistic Image Synthesis	11
1.1.2	Global Illumination	11
1.1.3	Full-Spectral Rendering	14
1.2	Objectives	14
1.2.1	Essential Questions	15
1.3	Notation Conventions	15
2	State of the Art	17
2.1	Hardware and Software Optimization	18
2.2	Perception and Color Reproduction	19
3	Fundamentals	21
3.1	Perception, Color Spaces and Color Difference	21
3.1.1	Standard Colorimetric Observers	23
3.2	Color Representations for Rendering	25
3.2.1	Three Components compared to Full Spectrum	25
3.2.2	Full Spectrum Representations	25
3.3	Code Optimization	27
4	Preliminary Investigation	29
4.1	Mathematics of Light Distribution and Sampling	29
4.1.1	Observations in Time Domain concerning Color Difference	30
4.1.2	Observations in Frequency Domain concerning Color Difference	31
4.1.3	Color Multiplication and Color Difference	34
4.2	Evaluation of Alternatives to Plain Equidistant Sampling	38
4.2.1	Basis functions	38
4.2.2	Composite Model	39
5	Spectral Mipmapping	41
5.1	Resampling and Downsampling	43
6	Implementation	45
6.1	The Renderer	45
6.1.1	Description of the <i>Minilight</i> Ray Tracing Renderer	45
6.2	Adaptation of an RGB Renderer for Full-Spectral Rendering	47
6.2.1	Optimized Code Design	47
6.2.2	Implementation of the Composite Model	49

6.2.3	Mipmapped Ray Tracing Pipeline with Level Feeler	51
6.2.4	Preprocessing	52
6.2.5	Postprocessing	53
7	Results	55
7.1	General Test Description	55
7.1.1	Questions and Answers (regarding all tests)	56
7.1.2	Time Measurement	56
7.1.3	Test Colors	57
7.2	Tests	57
7.2.1	Standard Cornell Box	57
7.2.2	Colorful Scenes	63
7.2.3	Color Difference Thresholds	66
7.2.4	Operator Times	69
8	Conclusion	73
A	Color and Color Difference	75
A.1	Tonemapping	75
A.2	Color-Matching Functions	75
A.3	Spectra	77
A.3.1	Light Source Spectra	77
A.3.2	Reflector Spectra	78
B	Test Details	81
B.1	Test Design and Time Measurement	81
B.2	Operating System, Compiler and Hardware	83
C	Source Code Listings	85
	List of Figures	88
	List of Tables	90
	Bibliography	91

Chapter 1

Introduction

1.1 Domain Definition

The considerations discussed in this work pertain to the area of *realistic image synthesis*, which implies the use of global illumination techniques.

1.1.1 Realistic Image Synthesis

Realistic image synthesis is a long standing concern of computer graphics the objectives and solutions of which developed in the past thirty years. The objective is to compute photograph-like images of virtual three-dimensional scenes that mimic real-world environments. The expected result is a deceptively real-looking picture, based on convincing light propagation and color appearance. Furthermore, realistic image renderers are used in order to visualize scenes which are impossible in the real world, such as illuminated fractal objects [2] or cartoons, used for animations by the film industry.

Three selected examples for realistic image synthesis in practice:

- Render car paint [3].
- Render interior and exterior design [4].
- Landscape design [5].

1.1.2 Global Illumination

As opposed to “local illumination”, where surfaces are at least culled using a z-buffer, but illuminated regardless of the physical influence of other, ambient surfaces, global illumination renderers are at least equipped with the capabilities listed in table 1.1. Their mathematical functioning is described by Kajiya’s Rendering Equation [6]:

$$I(x, x') = g(x, x') \left[\epsilon(x, x') + \int_S \rho(x, x', x'') I(x', x'') dx'' \right] \quad (1.1)$$

<i>Capability</i>	<i>required for...</i>
Light occlusion	all scenes
Light reflection, indirect illumination	all scenes
Light refraction	transparent objects

Table 1.1: Global Illumination Basics.

The variables x , x' and x'' are points in the three-dimensional scene where x is the final accumulation point of light collection, for instance a pixel on a two-dimensional image of the scene and x' a different point that may send energy (in the form of light) to the point x , either by emitting light (amount defined by $\epsilon(x, x')$) or by transmitting/reflecting light from all other sources in the entire scene x'' , accumulated by the integral term and scaled by the function $\rho(x, x', x'')$. In most cases, ρ acts as attenuator and $g(x, x')$ is 0 if x and x' are mutually invisible, otherwise it attenuates depending on the distance between them.

Ray Tracing and *Radiosity* are the fundamental techniques used for implementing global illumination renderers and can be described in terms of the rendering equation.

Ray Tracing

(figure 1.1)

This technique simulates light propagation as described by the rendering equation along randomly selected paths of photons. To avoid “photons” that miss the camera’s plane (final image pixels), the paths are started at the pixels of the image (the 2D locations of which are transformed to world space), instead of a light source, and to raise the number of light source hits, the path may not only be randomly reflected or refracted but additionally branched directly to one of the light sources. In terms of the rendering equation, the x'' variable is chosen to be on a light source. Further improvements such as *Photon Mapping* can equip a ray tracer with special capabilities for the efficient computation of caustics or diffuse interreflections.

Ray tracing implicitly delivers global illumination because of its lighting model that exhibits similarities to physical light propagation. The scene geometry must be available in the form of a data structure that allows for fast ray intersection in order to find reflecting and emitting objects that contribute to the color of the resulting pixels. Hence, occlusion tests (and thus shadows) are marginal, and indirect illumination is easily achieved by performing recursive random reflections (in combination with direct emitter sampling). Recursive reflections are described by the rendering equation’s recursive definition of $I(x, x'')$.

Radiosity

The concept behind the Radiosity technique [7] comes from heat propagation models. For each pair of individual surfaces in a scene, so-called form factors are precalculated. Form factors determine the exchange of energy between two surface patches if one of the patches relays heat rays. They depend on the size of the patches and their relative orientation. For computer graphics applications, this model can be applied on visible light rays instead of heat rays, due to the fact that both are based on electromagnetic waves.

The investigations of this work are not focused on geometrical issues of light propagation,

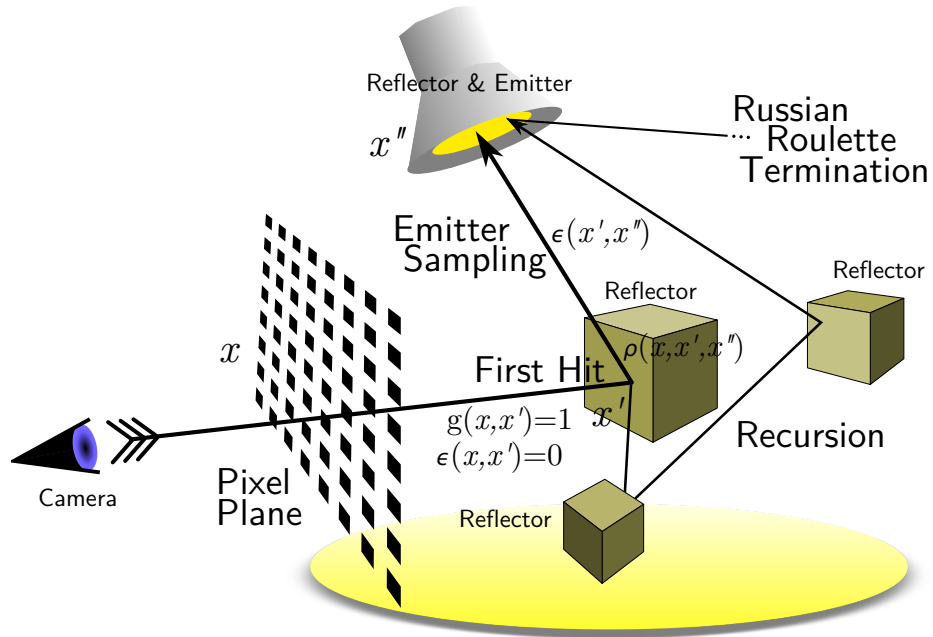


Figure 1.1: Light Transport in a Ray Tracer.

therefore it can easier be emulated using a simple ray tracer instead of a radiosity renderer. Therefore, a radiosity implementation is not regarded. However, the results produced in the course of this work are useful for radiosity renderer implementations as well.

Refraction and Reflection of Light

The light reflection and refraction behavior is defined for each surface in the form of a bidirectional reflectance distribution function (BRDF). The basic concepts of bidirectional reflectance are described in detail by [8]. Light rays that hit an object's surface are redirected by the surface. The simplest BRDFs are the BRDF of a mirror and the BRDF of an ideal diffuser. The former redirects the light ray to a particular direction exactly defined by the ray's incidence direction relative to the surface. In contrast, the latter redirects the light ray into an arbitrary direction, independent of its origin. In principle, ray tracing based global illumination renderers are capable of emulating both specular and diffuse BRDF light reflections, as well as refraction (in contrast, a pure radiosity renderer is limited to diffuse reflection).

1.1.3 Full-Spectral Rendering

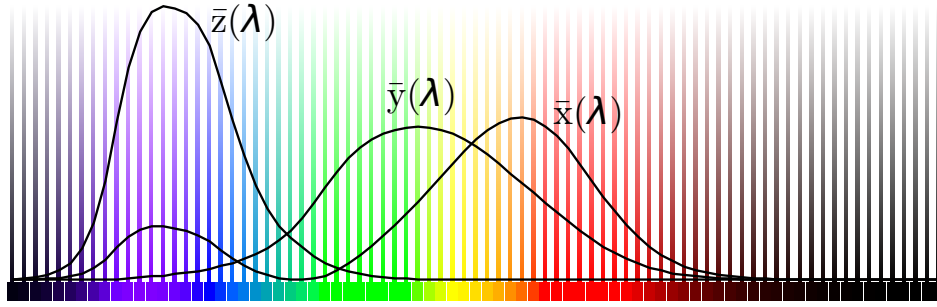


Figure 1.2: Spectral Colors with Color Matching Functions (see section 3.1.1).

Unlike frequently-used three-component based global illumination software that is primarily designed to operate on RGB colors (e.g. *Blender* [9]), a full-spectral renderer is focused on a physically based lighting model. For instance, the core of *LuxRender* [10] is capable of full-spectral rendering color computations. Light propagation computations are performed on digital representations of entire color spectra, which carry considerably more information than three-component colors. This allows for a physically correct simulation of light-surface interactions and physically-correct appearance, in particular if two or more light sources with different spectra illuminate a surface. The major disadvantage of full-spectral rendering are higher computation costs because of the higher amount of processed information. Even with modern hardware, global illumination in general is extremely time-consuming and rendering a complex scene can take several hours or even days. Therefore, a full-spectral rendering implementation should not introduce considerably more computation costs.

This work investigates methods of optimizing full-spectral color computations using a new approach called *Spectral Mipmapping*.



Figure 1.3: Full-Spectral Renderer.

1.2 Objectives

The task underlying this master's thesis is to research approaches of making spectral color operations in a full-spectral renderer faster. This requires a controlled reduction of accuracy

in order to find a reasonable trade-off between accuracy and speed. The number of floating point values representing a scene’s color spectra is to be minimized and thereby, the quality of the rendered image, as perceived by the human standard observer, must not be influenced.

Spectral Mipmapping is a promising approach because it basically enables the renderer to find an individual accuracy versus speed trade-off for each light ray, depending on the spectra the light ray collects during its lifetime. Based on a test implementation, the capability of Spectral Mipmapping is to be investigated.

1.2.1 Essential Questions

Essential questions that are to be answered in the course of the master’s thesis:

- Which kinds of spectral color representations are suited for a full-spectral renderer?
- Can the computer reliably guess the parameters that are necessary for an optimal trade-off between time-efficiency and accuracy?
- Which speed improvements can be accomplished?
- How does the availability of optimization techniques influence the usefulness of promising spectral color representations?
- What must be regarded if a conventional three-component raytracer is to be prepared for full-spectral rendering and Spectral Mipmapping?

1.3 Notation Conventions

Vectors are stated either in the form of $n \times 1$ matrices or in the form transposed $1 \times n$ matrices. Vector symbols are **bold**. Unit vector example:

$$\mathbf{v} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T$$

Colors are either spectra in the visible wavelength range (“spectral color representation”) or metamer classes (“three-component color representation”). Three-component colors are specified in the form of \mathbb{R}^3 vectors.

Scalar Operations are multiplications or additions of two real numbers, i.e. numbers $a, b \in \mathbb{R}$ (floating-point numbers).

Chapter 2

State of the Art

Johnson and Fairchild [11] comment on the necessity of full-spectral color calculations.

Leading rendering software for realistic image synthesis is primarily focused on three-component (RGB) color rendering, although, for example, the core of *LuxRender* [10] can operate on full-spectral colors. If one decides to equip an RGB-only renderer with full-spectral rendering capability, very likely considerable parts of the core source code need to be rewritten in order to allow for efficient color operations, as the adaptation of the Minilight ray tracer this thesis is based on indicates.

Spectral Color Representations Several mathematical methods exist for constructing compressed representations of spectral information. (In fact, a certain amount of information reduction always comes along with compression of spectral data.) Basic but not necessarily effective methods such as point sampling, riemann summation or gaussian quadrature are discussed in various papers, for instance [12], [13], [14]. In the chapter *Spectrum Decomposition and Reconstruction* of the comprehensive reference book *Computational Color Technology* [15], its author points out that few components (scalar real numbers) are found to be sufficient to accurately represent various lights or reflectance spectra. A full-spectral renderer may benefit from the reduced memory consumption if the spectrum is suchlike compressed, in particular when color representations are to be copied between memory locations during the rendering process, provided that they do not complicate other essential color operations, in particular light-surface reflection multiplications. Many application areas in the domain of color-image processing, such as color transformation, white-point conversion and chromatic adaptation are cited, but the usefulness in full-spectral renderers is not treated.

A sophisticated approach for compressing spectral color information has been proposed by Sun [16]. The so called *Composite Model* splits the spectrum into a smooth part and a spiky part, a hybrid representation of spiky spectra. Wilkie [17] argues that the special treatment of added-on spikes in this hybrid approach makes computation more complicated. They advocate the use of evenly-spaced sampling only. Both approaches, evenly-spaced sampling only as well as the Composite model are investigated in the course of this thesis.

2.1 Hardware and Software Optimization

Recent successors of the Intel 8086 processor are equipped with an increasing number of performance-optimizing extensions. Extensions such as *MMX*, *SSE* or *3DNow!* turn an *SISD* x86 CPU more or less into an *SIMD* CPU [18]. *SISD* is the abbreviation for *single-instruction stream–single data-stream*, which basically means that data is processed in series on a single core CPU. *SIMD* extensions (*SIMD* stands for *single-instruction stream–multiple data-stream*) introduce machine instructions that allow for simultaneous execution of basic operations (such as copy operations or arithmetic operations).

Modern PCs that are qualified for computer graphics tasks are usually equipped with a powerful graphics processing unit (GPU) that is specialized in floating point operations. Recent GPUs provide *SIMD* architectures and through *CUDA* [19] or *OpenCL* [20] they can be utilized for costly general purpose processing that was otherwise done by the CPU. Suchlike GPUs are called *GPGPU* (general purpose graphics processing unit). *GPGPU* processing can particularly be useful for realistic image synthesis, for instance to do parallel light propagation calculations inside a raytracing algorithm. *OpenCL* allows for heterogeneous computing using all general purpose processing units that are part of the system. Unlike *CUDA*, *OpenCL* programs are portable between arbitrary hardware platforms that support *OpenCL* and therefore a seminal quasi-standard.

SIMD CPU or *GPGPU* instructions can make a full-spectral renderer that deals with equidistant spectral samples considerably faster, because multiple samples can be handled in parallel.

Many performance-enhancing programming techniques have been proposed that allow for software optimization focused on hardware, one of them is building loops so that they can be unrolled by the compiler as illustrated in figure 2.1. The second step can be the compiler-controlled grouping of successive operations in the form of *SIMD* vector operations. Performance-enhancing techniques should definitely be taken into account as soon as possible when attempts are made to improve a full-spectral renderer, because they can affect a program’s design. A comprehensive range of publications that cover basic concepts as well as various treatments of particular use cases is available these days, for instance [21].

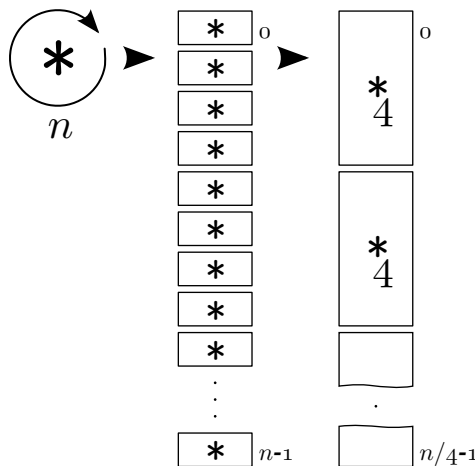


Figure 2.1: Frequently applied Optimizations: Loop Unrolling and Vector Operations.

2.2 Perception and Color Reproduction

The human visual system and appropriate color representations are described in the book *Color Science* [22]. It is a comprehensive reference work that explains basic principles, provides useful data tables and describes how to estimate perceived color differences. However, the integrity of this work should not hide the fact that the human visual system is still not entirely understood, and will probably never be, because the human brain plays an important role in perception models and is only coarsely understood these days due to its complexity. This complexity is also reflected in the ongoing sophistication of the CIE color difference formula and its imperfectness [23].

The manufacturers of color displays and printing devices are still constrained to rely on the fact that metameric colors possessing different spectra share the same appearance under the same illumination conditions, and mix few predefined colors in order to produce arbitrary color impressions.

A color the spectrum of which is known can be accurately transformed into the corresponding metamer produced by tristimulus color mixing through known computational algorithms (given that the result is viewed by the *standard observer*). For verification purposes it must also be possible to perform color comparisons. Perceived chromatic and intensity aberrations are to be estimated computationally. Therefore, uniform color spaces have been created and standardized. Anyhow, color difference formulas, which are based on such uniform color spaces, are still not fully developed. Nevertheless they are very useful and either already standardized or available in the form of a draft publication.

Chapter 3

Fundamentals

3.1 Perception, Color Spaces and Color Difference

Young-Helmholtz Theory The most important color space in the computer graphics area is the *RGB* color space: Based on the Young-Helmholtz theory of the trichromacy of color, arbitrary human-visible colors can be produced by adding or subtracting the so-called *primary colors* red, green and blue, which correspond to three types of photoreceptors that are sensitive to long-wave, middle-wave and short-wave light.

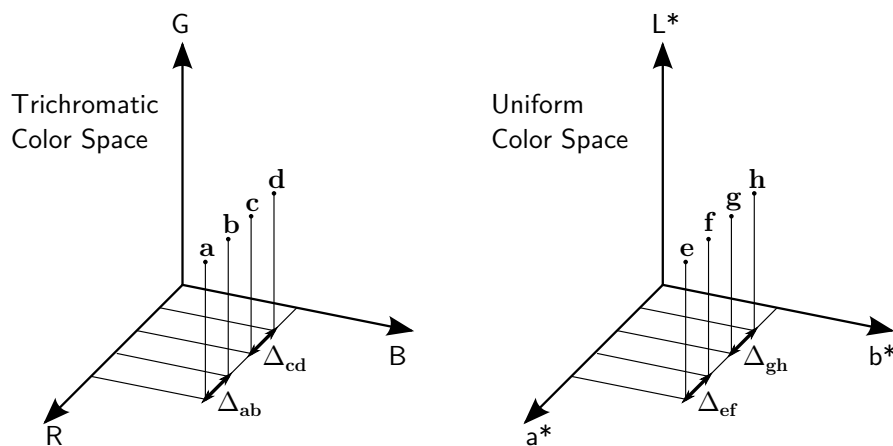


Figure 3.1: 3D plots of a trichromatic color space and a uniform color space.

Opponent-Colors Theory The Young-Helmholtz theory by itself is not sufficient to explain the perception of color differences: Trichromatic color spaces are not suited for straightforward estimation of perceived difference between two colors. If the trichromatic representations of four different colors **a**, **b**, **c**, **d** are plotted in an orthogonal coordinate system as shown in the left part of figure 3.1, the perceived distances between the colors in the pairs (**a**, **b**) and (**c**, **d**), ΔE_{ab} and ΔE_{cd} , are not necessarily equal even if the euclidean distances Δ_{ab} and Δ_{cd} are. Uniform color spaces are an approach of converging euclidean distances to perceived distances. Since such a convergence cannot be achieved in a trichromatic color space, uniform color spaces rely on additional findings that were at first considered to be incompatible with the Young-Helmholtz theory: Ewald Hering proposed the opponent-colors

red	yellow
yellow	green
green	blue
blue	red

Table 3.1: Opponent colors.

L^*	black	light
a^*/u^*	red	green
b^*/v^*	yellow	blue

Table 3.2: $L^*u^*v^*/L^*a^*b^*$ brightness and color pairs.

theory. According to this theory, the color pairs listed in table 3.1 are mixed in the form of opposite neural signals in the human visual system, which produces the actual intermediate hue impressions.

Zone Theory Zone theories assumes that both, the Young-Helmholtz theory as well as the opponent-colors theory apply to the human visual system, in two successive zones:

Zone 1	<i>Young-Helmholtz</i>	trichromatic input	photopigments of cones (eye)
Zone 2	<i>opponent-colors</i>	coding of cone signals	neural network
Zones 3...?	assumed to exist		

In the CIELUV/CIELAB color spaces ($L^*u^*v^*/L^*a^*b^*$), which are based on a zone theory, the color pairs of the original opponent-colors theory appear in a condensed form, supplemented by a brightness component (L^*). Table 3.2 lists the brightness and color pairs. The mathematical differences between CIELUV and CIELAB are shown by [24], [22] and colored illustrations are provided by [25].

For the colors **e**, **f**, **g**, **h** plotted in figure 3.1, the following reasoning applies as opposed to **a**, **b**, **c**, **d**:

$$\mathbf{e} = \begin{pmatrix} L_{\mathbf{e}}^* & a_{\mathbf{e}}^* & b_{\mathbf{e}}^* \end{pmatrix}^T, \mathbf{f} = \begin{pmatrix} L_{\mathbf{f}}^* & a_{\mathbf{f}}^* & b_{\mathbf{f}}^* \end{pmatrix}^T, \mathbf{g} = \begin{pmatrix} L_{\mathbf{g}}^* & a_{\mathbf{g}}^* & b_{\mathbf{g}}^* \end{pmatrix}^T, \mathbf{h} = \begin{pmatrix} L_{\mathbf{h}}^* & a_{\mathbf{h}}^* & b_{\mathbf{h}}^* \end{pmatrix}^T$$

$$\Delta_{\mathbf{ef}} = \|\mathbf{f} - \mathbf{e}\| = \sqrt{(\Delta L_{\mathbf{ef}}^*)^2 + (\Delta a_{\mathbf{ef}}^*)^2 + (\Delta b_{\mathbf{ef}}^*)^2} \quad (3.1)$$

$$\Delta_{\mathbf{gh}} = \|\mathbf{h} - \mathbf{g}\| = \sqrt{(\Delta L_{\mathbf{gh}}^*)^2 + (\Delta a_{\mathbf{gh}}^*)^2 + (\Delta b_{\mathbf{gh}}^*)^2} \quad (3.2)$$

The CIE 1976 color difference formulae are defined as follows:

$$\Delta E_{uv}^* = \sqrt{(\Delta L^*)^2 + (\Delta u^*)^2 + (\Delta v^*)^2} \quad (3.3)$$

$$\Delta E_{ab}^* = \sqrt{(\Delta L^*)^2 + (\Delta a^*)^2 + (\Delta b^*)^2} \quad (3.4)$$

The right sides of equations 3.1, 3.2 and 3.4 are basically the same, therefore:

$$\Delta E_{ab,\mathbf{ef}}^* = \Delta_{\mathbf{ef}}, \Delta E_{ab,\mathbf{gh}}^* = \Delta_{\mathbf{gh}}$$

From this point of view, the $L^*a^*b^*$ color space can be used as straightforward as possible: The color difference is always equal to the euclidean distance of the three components. But recent refinements of the color difference formula clearly point up that this estimation is only a coarse approximation of the much more complex real perception system, i.e. the $L^*u^*v^*$ and $L^*a^*b^*$ color spaces are actually not uniform in terms of their use as color difference estimators.

The CIEDE2000 color difference formula is a very complex set of formulae if applied on colors that are given in the form of $L^*a^*b^*$ components. Several precalculations [26] prepare the parameters for the final main formula which is defined as follows:

$$\Delta E_{00} = \sqrt{\left(\frac{\Delta L'}{K_L S_L}\right)^2 + \left(\frac{\Delta C'}{K_C S_C}\right)^2 + \left(\frac{\Delta H'}{K_H S_H}\right)^2 + R_T \left(\frac{\Delta C'}{K_C S_C}\right) \left(\frac{\Delta H'}{K_H S_H}\right)} \quad (3.5)$$

The input parameters of the precalculations are the $L^*a^*b^*$ components of the colors that are to be compared. In fact, the precalculations transform them into a completely different color space that is reflected in the three quadratic terms of the main formula:

$\boxed{\Delta L'}$ *Lightness* difference, equals ΔL^* .

$\boxed{\Delta C'}$ *Chroma* (saturation), ruled by the magnitudes (pos./neg.) of a^* and b^* .

$\boxed{\Delta H'}$ *Hue* difference: roughly the 2D orientation on the a^*, b^* plane.

A C++ implementation of the CIEDE2000 color difference calculation is provided in the appendix. Implementation notes can be found in [26] and [23] discusses mathematical discontinuities in the formulae.

However, figure 3.2 suggests that even the euclidean distance is still sufficient for many applications, because the resulting difference value is always more conservative (higher) than the CIEDE2000 result for the same color pair. The diagrams show the estimated color difference (ordinate) plotted against changed L^* , a^* and b^* values of the compared colors, which are varied in the range $-150.0 \dots 150.0$ (abscissae). The test source code is listed in the appendix. Even if all 3 components are arbitrarily different, the ΔE_{00} value will be below the ΔE_{ab}^* value, and near 0 they converge. Therefore use of the euclidean distance might be favored over the CIEDE2000 formulae particularly in time-critical applications if the accuracy of high difference estimations is of little importance.

Further information regarding color perception and color difference can be found in [27], [24], [22], [28].

3.1.1 Standard Colorimetric Observers

This section gives a brief review of the pioneering experiments that were conducted in order to scientifically describe the nature of perceived colors and how the XYZ color space was created. The facts are extracted from Wyszecki and Stiles [22].

CIE 1931 Standard Colorimetric Observer The *Commission Internationale de l'Eclairage*, CIE, played the major role in defining standards regarding the link between electromagnetic waves and perceived appearance. The basis for this link was created by Guild and Wright in the first half of the 20th century, who independently performed color matching

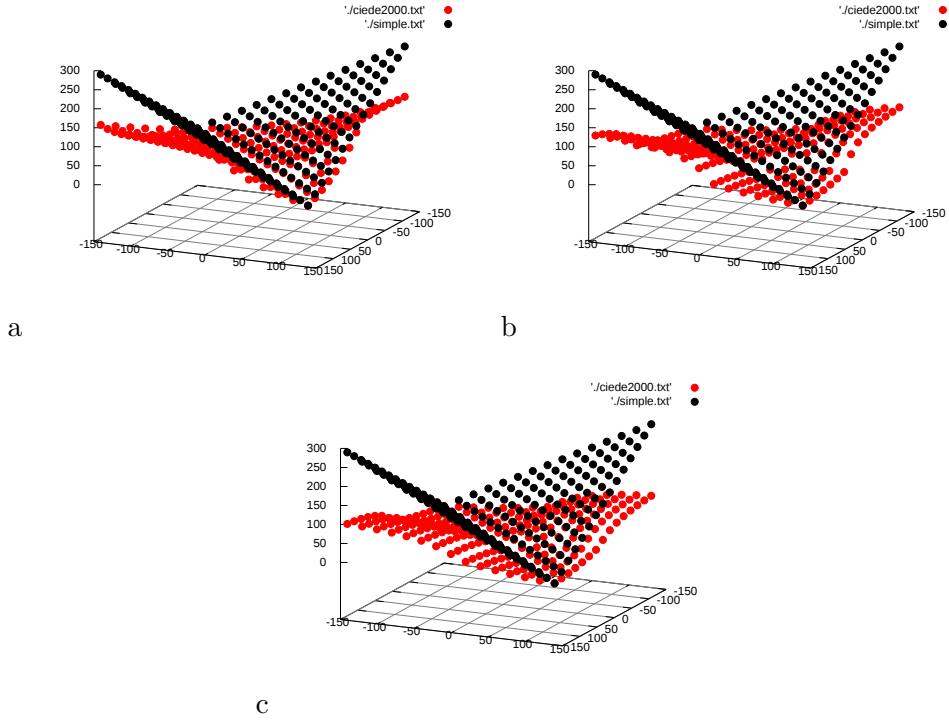


Figure 3.2: CIEDE2000 (red dots) compared to Euclidean (CIE 1976) Perceived Color Difference (black dots) plotted against L^* (plot a), a^* (plot b), b^* (plot c).

experiments with two different trichromatic colorimeters. They presented monochromatic light stimuli in the wavelength range from 400 to 700 nanometers to a couple of observers (Guild: 7 observers, Wright: 10 observers) who had to match them by adjusting three primary lights. Wright used monochromatic primaries having wavelengths of 460 (blue), 530 (green) and 650 (red) nanometers, respectively (Guild produced wavelength bands by filtering the light of an incandescent tungsten lamp). The intensities of the adjusted primary lights were recorded for each of the presented stimuli and can be plotted in a wavelength/intensity diagram in the form of three curves, known as *color-matching functions* (curves in figure 1.2 in the introduction). Guild transformed his and Wright's results to a common system in order to define a “standard observer”, the *CIE 1931 Standard Colorimetric Observer* (thus, the average of overall 17 human observers), the basis for applied colorimetry since 1931.

CIE 1964 Supplementary Standard Colorimetric Observer The color-matching functions of this standard observer are defined in a wider wavelength range (360 to 380 nanometers). It is intended for use when the colors of large angular areas are in question.

The color-matching functions are smooth curves, defined in the form of $1nm$ equidistant samples (for practical reasons). For computational reasons, negative chromaticity coordinates were undesirable. In order to avoid them, the CIE invented new (imaginary [22]) primary stimuli X, Y and Z that allow for non-negative color-matching functions. The original trichromatic system invented by Guild and Wright can be transformed into the XYZ system and back simply by applying a three-dimensional transformation matrix.

The CIE 1931 color-matching functions $\bar{x}(\lambda)$, $\bar{y}(\lambda)$ and $\bar{z}(\lambda)$ are listed in appendix A.

3.2 Color Representations for Rendering

3.2.1 Three Components compared to Full Spectrum

In Computer Graphics, colors are usually represented by three components, namely red, green and blue, short hand *RGB*, that are stored in the form of three integer or floating point values. The red, green and blue components are mixed by synchronous presentation on a small surface patch in order to produce most of the colors the human visual system is able to distinguish (visible colors within the RGB/CMY/... gamut of the device).

Three-component color representations are perception-based and therefore they solely carry the information required to describe the colors that can be distinguished by the human visual system. In fact, they exploit the nature of *metamerism*.

Metamerism Two light sources with different spectra may possess equal appearance. This also applies to the reflectance of surfaces if the illuminator spectrum is left unchanged.

Metamerism is one reason why the usefulness of three-component colors in physically-based rendering tasks is limited compared to color representations that preserve most of the available information, ideally the power intensity at each visible wavelength. A further reason is introduced if realistic wavelength-dependent refraction effects and fluorescence effects are of major importance. Table 3.3 lists the major limits of three-component rendering.

<i>Rendering Operation</i>	<i>RGB/Three Components</i>
Additive Color Mixing	sufficient
Reflection	can be sufficient in case of a single light source
Refraction, Fluorescence Effects	realistic simulation impossible

Table 3.3: Three Component Rendering Constraints.

3.2.2 Full Spectrum Representations

Three different characteristics of spectra can be observed:

Type A: Wavy Spectrum with Noise Possesses waves having turbulent shape, for instance the spectrum of daylight [22].

Type B: Smooth Wavy Spectrum Typically feature one major maximum and different intensity levels dependent on the wavelength. They may result from the sketchy reconstruction of a type A spectrum (for example if measured using a spectrophotometer device).

Types A+S/B+S: Spiky Spectrum Possesses steep gradients at certain wavelength ranges. Spikes of fluorescent lamps occupy wavelength ranges of less than 10 nm. Often, spiky spectra possess a type A or B part that is obviously independent of the spikes.

Ideal Spectrum Representation A spectrum is completely represented in the visible range by a function $f(\lambda)$, $\lambda \in \mathbb{R}$, defined in the interval $[\lambda_{visible,min}, \lambda_{visible,max}]$, which is the range of human-visible wavelengths of light.

Because of the huge amount of data, it is impractical to store a color spectrum in this form.

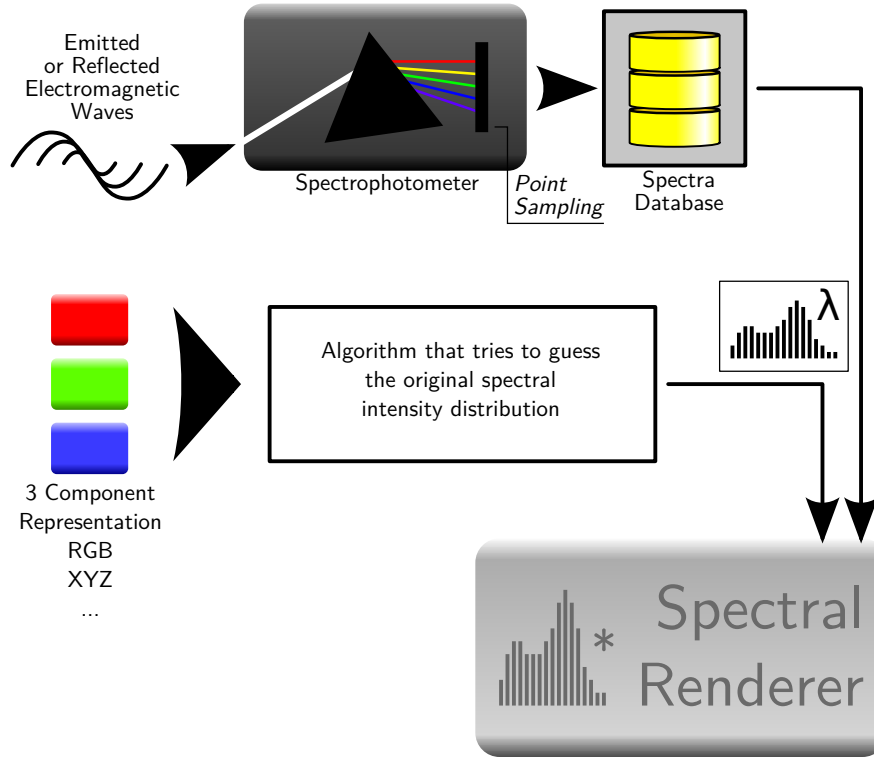


Figure 3.3: Acquisition of Spectral Data for Rendering.

Real Spectrum Representation Spectral color representations can be synthesized, but in most cases they stem from databases or they are measured from in-house patterns, using a *spectrophotometer* device. A spectrophotometer usually samples the color of (reflected) light in equidistant wavelength intervals, for instance $10nm$. Although this results in less than 50 intensity samples, for most spectra only a negligible amount of information is lost. The resulting representation is a *point sampled spectrum*. A spectrophotometer can either be monochromatic or multichromatic, for instance in the form of an image scanner to generate spectral-colored textures or in the form of a camera.

Area Preserving Sampling Using the same number of samples, more accuracy can be achieved if, instead of point sampling, the surroundings of the sample wavelength are regarded by integrating the intensities in the surrounding wavelength range. For instance, to calculate the average intensity round $650nm$, assumed the sampling interval is $10nm$, $\int_{\lambda=645}^{655} f(\lambda)d\lambda$ is the recommended computation. Then possible peaks between $645nm$ and $650nm$ and between $650nm$ and $655nm$ are, as opposed to point sampling, not dropped but incorporated into the total value of the sample. The area formed by the wavelength interval and the intensities is preserved. The ranges beyond $645nm$ and $655nm$ contribute to the neighboring samples at $640nm$ and $660nm$, respectively. Area preserving sampling should be used whenever a spectrum is to be *resampled* (resampling is discussed in section 5.1).

RGB devices can produce most of the colors the human visual system can distinguish. But real colors contain a good deal more information than three floating point values may store. They are mixtures of electromagnetic waves with different amplitudes that possess virtually infinite different light wave frequencies/wavelengths. Thus, assumed that no simplifications can be applied (this is discussed in the following chapter), huge arrays of floating point values, where successive floating point values represent the energy of successive wavelength ranges in the spectrum, are necessary to precisely store arbitrary natural color spectra on a digital device.

3.3 Code Optimization

Since a global illumination renderer is a CPU-bound (or GPU-bound) process, code and compiler optimizations, ideally adapted to specific capabilities of the target processor, are of vital importance in order to provide time efficiency.

Many performance optimization recommendations and techniques can be found in literature. Besides inlining and cache-friendly programming, three of them are of primal importance (provided that C++ is used for programming):

- Avoid unnecessary copies.
- Prefer stack allocations over heap allocations or use a memory pooling technique.
- Allow for (or apply) loop optimizations and the use of vectorization.

Stalls and Branches Modern CPUs preload instructions into their processing pipeline. Branches in the machine code can lead to preloaded instructions that must be discarded because of a decision for a branch that does not use the instructions. This is called a pipeline stall. Thus, branches are precarious constructs and should be avoided if they are not really required.

Where branches cannot be avoided, they can be optimized by the use of jump tables if *switch* statements are used. A jump table uses the switch case integers as indices and provides addresses for further code executions. This minimizes the costs of condition testing. However, this does not combat pipeline stalls.

Loop Optimization A full-spectral renderer inherently requires loops with a considerable control overhead compared to the loop body for color processing. The individual loop iterations are usually independent of each other and the number of iterations is not unpredictable. The compiler is usually capable of detecting these conditions if the source code is written properly (for instance, the count number of a *for* loop should be an integer literal or constant). To reduce the loop overhead, the compiler can put several iterations into a single iteration to be serially performed or in parallel using SIMD instructions (this may require a compiler flag, for example *-msse* if G++ should apply SSE instructions). To force the complete elimination of loop overhead by entirely unrolling a loop, the loop may need to be unrolled manually. But afterwards the resulting machine code instructions should be examined in order to estimate whether they will fit into the instruction cache of the processor.

Chapter 4

Preliminary Investigation

4.1 Mathematics of Light Distribution and Sampling

This section discusses the expected influence of the interaction of sampling and light distribution on the difference $d(s_{r_1}, s_{r_2})$ between the reconstructions of two sample representations $s_{r_1}(i)$ and $s_{r_2}(j)$ of a spectrum, sampled at different rates $r_1 \neq r_2$, $i \in \{1, 2, 3, \dots, n_1\}$, $j \in \{1, 2, 3, \dots, n_2\}$, $n_1 \neq n_2$, in a global illumination renderer.

The following operations are integral parts of a global illumination renderer:

Scaling used for *attenuation* (e.g. dependent on the angle of incidence of a light ray)

Multiplication used for *reflection* (light-surface interaction)

Addition used for power *accumulation* (pixel color)

They are the fundamental operations of light distribution.

Renderer as Digital Signal Processor The question how sampled light and reflection spectra interact with applying these operations can be answered using the signal theory: A plotted light spectrum can be regarded as an electronic signal simply by changing the meaning of the wavelength axis. The intensities at different wavelengths change to the intensities at consecutive points in time, then the plotted spectrum has the meaning of an oscillogram. The oscillogram shows the sum of the signal's oscillations. Actually each signal is a composition of up to infinite different sine and cosine oscillations. Each of these oscillations has a unique frequency and the corresponding wavelength. For the rest of this section, the terms *frequency* ω and *wavelength* λ are defined as follows:

Frequency ω frequency of a signal oscillation (i.e. not the reciprocal of *photometric wavelength*)

Wavelength λ photometric wavelength (= signal time)

A photometric spectrum only carries a finite amount of information, a sampled spectrum no more than the information of its samples (usually floating-point numbers). Therefore, the number of oscillations is also a finite number and the signal is *bandlimited* at f_{max} , the maximum frequency that is part of the oscillation frequencies. Furthermore, the signal's appearance beyond the limits of the visible range is irrelevant and usually unknown if human

individuals are the final information receivers. Thus, it might recur beyond those limits and so it can be considered to be a *periodic* function.

Since the wavelength axis is the equivalent of a signal plot's time axis, a light spectrum is a periodic function of time if it is regarded in *time domain*.

4.1.1 Observations in Time Domain concerning Color Difference

Scaling on Sampled Spectrum/Signal

$$t_{r_1}(i) = s_{r_1}(i) * a$$

$$t_{r_2}(i) = s_{r_2}(i) * a$$

Scaling changes the intensities and does not distort the signal. In the case of attenuation (scaling factor < 1.0) the color difference is expected to decrease: $d(t_{r_1}, t_{r_2}) < d(s_{r_1}, s_{r_2})$. From this point of view this operation is unproblematic and will not further be examined.

Multiplication of Sampled Spectra/Signals

$$t_{r_1}(i) = s_{r_1}(i) * a_{r_1}(i)$$

$$t_{r_2}(i) = s_{r_2}(i) * a_{r_2}(i)$$

Multiplication of two spectra corresponds to an *amplitude modulation* of two signals: One signal modulates the other and the result is a distortion of the modulated signal. If two slopes that are within the same time range in both spectra and the gradients of the slopes both point up or down, the resulting slope will be steeper which might require a higher sampling rate to avoid $d(t_{r_1}, t_{r_2}) > d(s_{r_1}, s_{r_2})$. This problem is discussed in the next section with the aid of the Fourier transform.

Addition of Sampled Spectra/Signals

$$t_{r_1}(i) = s_{r_1}(i) + a_{r_1}(i)$$

$$t_{r_2}(i) = s_{r_2}(i) + a_{r_2}(i)$$

Addition also leads to a result the shape of which differs from the shapes of $s_{r_1}(i)$ and $a_{r_1}(i)$. How this affects $\frac{d(t_{r_1}, t_{r_2})}{d(s_{r_1}, s_{r_2})}$ and $\frac{d(t_{r_1}, t_{r_2})}{d(a_{r_1}, a_{r_2})}$ respectively is discussed in the next section.

4.1.2 Observations in Frequency Domain concerning Color Difference

To examine a sampled signal $s(i)$ in frequency domain, it is to be transformed using the *Discrete Fourier Transform*. Because light spectra can be regarded as periodic functions, the following formula applies to the continuous reconstruction of the spectrum $f(t) = \text{reconst}(s)$:

$$f(t) = \frac{a_0}{2} + \sum_{\nu=1}^{\infty} (a_{\nu} \cos \nu\omega_0 t + b_{\nu} \sin \nu\omega_0 t) \quad (4.1)$$

(formula 4.1 copied from [29])

The index ν is increased in integer steps. The coefficients a_{ν} and b_{ν} describe the amplitudes of oscillations that form the signal. They can be calculated using the sampled signal.

n	...	number of samples
ω_0	=	$\frac{2\pi}{n}$

$$a_{\nu} = \sum_{i=1}^n s(i) \cos \nu\omega_0 \quad (4.2)$$

$$b_{\nu} = - \sum_{i=1}^n s(i) \sin \nu\omega_0 \quad (4.3)$$

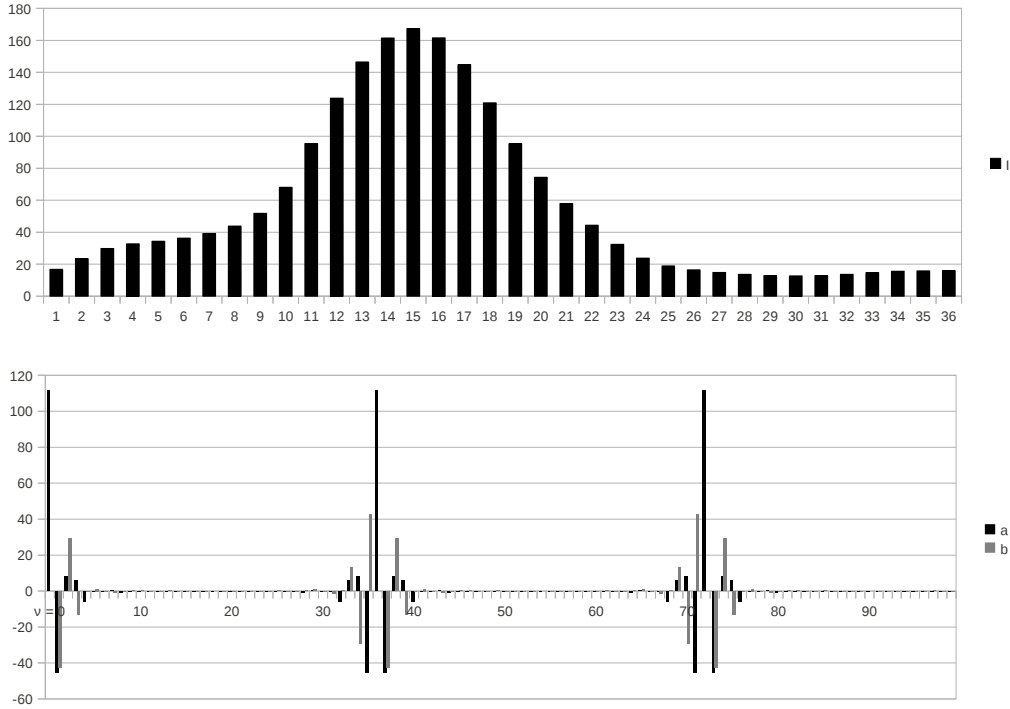


Figure 4.1: Sampled Green Light Spectrum and its Fourier Coefficients.

Figure 4.1 shows the fourier coefficients of an arbitrarily chosen spectrum's signal. The bars in the lower diagram represent the oscillations that contribute to the signal: The abscissa

(horizontal) shows the values of ν which correspond to the oscillation frequency, and the ordinate (vertical) shows the amplitude (absolute value of the oscillation's maximum/minimum). The information represented by the lower diagram is a complete representation in frequency domain of the sampled signal.

The leading bar represents the signal's constant component ($\omega = 0$), the further bars cosine and sine frequencies of $\nu\omega_0$. Noticeable tendency: the higher the frequency the lower the amplitude. Amplitudes beyond $\nu = 10$ almost disappear. The recurrence of amplitudes round $\nu = 36$ can be explained using *Shannon's Sampling Theorem*.

Shannon's Sampling Theorem

A time function $f(t)$ is uniquely defined by a set of equidistant samples if it is bandlimited at ω_{max} (oscillations having $\omega > \omega_{max}$ filtered) and sampled at π/ω_{max} intervals [29](page 236). Hence, after bandlimiting, the highest residual frequency determines the maximum gap between two samples so that the entire information carried by the bandlimited signal is preserved. Frequencies of arbitrary signals that exceed the band limit corresponding to a certain sampling rate get lost when sampling the signals at that rate, thus sampling includes bandlimiting.

Frequency Aliasing The recurring amplitudes in figure 4.1 are aliasing artifacts. The artifact oscillations run through the same ordinate values at the sample locations as the true oscillations do. The first aliasing block starts with $\nu = 18$. Only the oscillations at $\nu = 1...17$ are actually useful. This observation confirms the sampling theorem.

Figure 4.2 shows merely six equidistant samples of the spectrum of figure 4.1. Aliasing blocks considerably overlap in the frequency domain diagram, i.e. major oscillations are lost, and correspondingly in time domain the signal's original shape can only be guessed from the samples. Accurate reconstruction is impossible and the reconstructed signal will notably differ from the original.

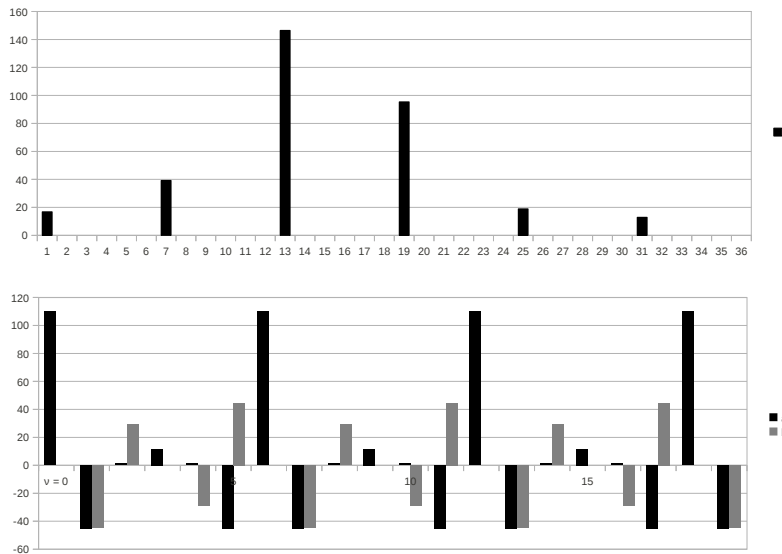


Figure 4.2: Spectrum of figure 4.1, samples dropped.

This example shows that the fourier coefficient plot is a practical aid for evaluating the appropriateness of a certain sampling rate for a certain spectrum. Using this aid, the influence of basic light distribution operations on the color difference can be estimated:

Multiplication of Sampled Spectra/Signals

(Amplitude Modulation)

Figure 4.3 shows the spectrum multiplied by itself and figure 4.4 discloses the consequences of this operation: The range of considerable amplitudes stretches on the abscissa. In fact, new oscillations with sum and difference frequencies of the original oscillations are introduced. This is a known effect of amplitude modulation. Hence, to completely avoid information loss, the sampling rate needs to be doubled.

Consecutive Multiplications It is required to initially sample each spectrum at intervals of $\pi/(\omega_{s,max} + \omega_{t,max}n_{mul})$ in order to avoid further information loss after the initial sampling caused by amplitude modulation, where $\omega_{s,max}$ is the maximal frequency that occurs in factor $s(i)$, $\omega_{t,max}$ is the maximal frequency that occurs in factor $t(i)$ and n_{mul} is the number of multiplications that are to be carried out in succession. For example: Assuming maximum frequencies of $\pi/20$ (frequencies that are preserved if sampled at $20nm$ intervals or less), to achieve a lossless result after 3 reflections, a sample spacing of

$$\frac{\pi}{\frac{\pi}{20} + \frac{3\pi}{20}}nm = \frac{1}{\frac{4}{20}}nm = 5nm$$

is required.

Thus, multiplication is a considerable source of color difference increase candidate.

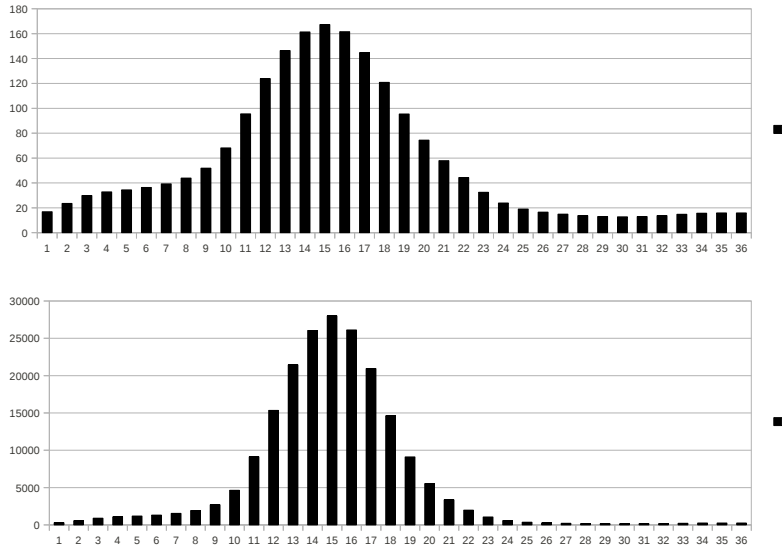


Figure 4.3: Spectrum of figure 4.1 (upper diagram) multiplied by itself (lower diagram).

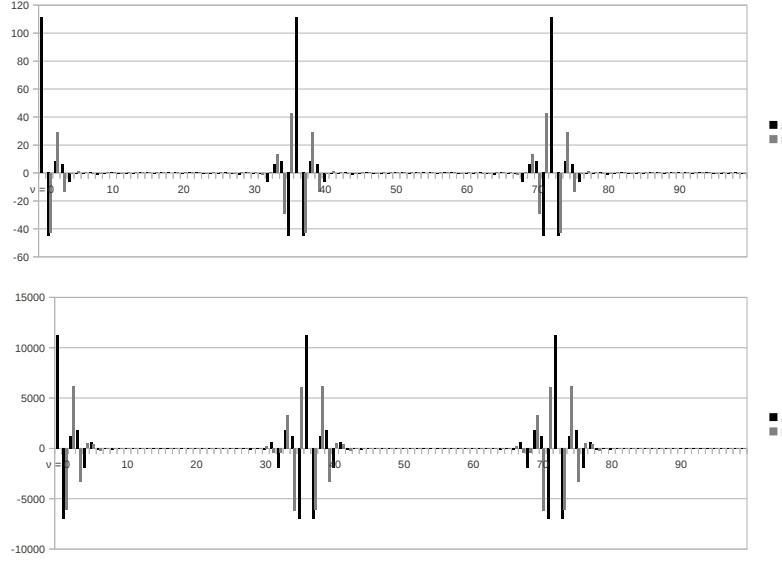


Figure 4.4: Fourier coefficients of the figure 4.3 spectra.

Addition of Sampled Spectra/Signals

The oscillations of the summands are added. Apparently, no new oscillations emerge and the initial sampling rate will be adequate for the resulting signal. Therefore, the choice of the sampling rate will not systematically moderate the influence of additions on color difference.

Conclusion

Successive multiplications of spectra as they usually occur in global illumination renderers (particularly in the recursive part of a ray tracer) cause information loss during the rendering process. Therefore, the color difference between two images generated at distinct sampling rates is expected to increase with the number of successive multiplications.

4.1.3 Color Multiplication and Color Difference

If the sample spacing is chosen by the renderer itself for performance optimization and a particular color difference threshold is given to assure an expected minimum quality for the resulting image, color difference pretests are required to decide whether the sample spacing is actually dense enough.

In order to design a meaningful pretest, the developing characteristic of the color difference over successive spectrum-by-spectrum multiplications (surface reflections) is of vital importance. Figure 4.5 shows the development of color differences between the results of reference multiplications and the results of multiplications of resampled spectra. The reference spectra are reflectances of real surfaces measured using the *GretagMacbeth EyeOne* spectrophotometer (see [30]) and stored in the form of 36 equidistant samples ($\Delta\lambda = 10nm$). They are resampled at $\Delta\lambda = 30nm$ in order to produce the test colors. Two tests were performed:

Test 1: Single Spectrum Sequence 20 randomly chosen spectra are multiplied by themselves 30 times. Simulates successive reflections at equally colored surfaces.

Test 2: Multiple Spectra Sequence Randomly chosen spectra are multiplied in 20 sequences, 30 times per sequence. Simulates successive reflections at differently colored surfaces.

The resulting color differences are listed in tables 4.1 and 4.2 in the form of CIEDE1976 measures.

Test procedure for one sequence:

1. Select the first spectrum $s_1(i)$, $i \in \{1, 2, \dots, 36\}$ and resample it at 1/3 of the original rate ($s_{1,resamp}(j)$, $j \in \{1, 2, \dots, 12\}$).
2. Illumination: Scale $s_1(i)$ and $s_{1,resamp}(j)$ by 1000.
3. Calculate XYZ representations of the resulting spectra and multiply the XYZ representation by a scale factor to fit into the range $\begin{pmatrix} 0.0 & 0.0 & 0.0 \end{pmatrix}^T \dots \begin{pmatrix} 0.95047, 1.0, 1.08883 \end{pmatrix}^T$ (black...reference white) if it is outside this range in order to produce realistic $L^*a^*b^*$ values.
4. Calculate the color difference ΔE (CIEDE1976) and store it.
5. Multiply the resulting spectra by the second spectrum in the sequence and its resampled counterpart.
6. Perform steps 3 and 4 again.
7. Continue with the third spectrum in the sequence...

The $L^*a^*b^*$ values underlying the color differences are listed in tables A.2 and A.3 in the appendix.

Discussion of the Test Results

Test 1 The color errors show differing tendencies until the tenth multiplication is reached (figure 4.5.a): They stay at a moderate level round the just noticeable difference ($\Delta E = 1.0 \dots 2.0$), slowly increase, or drastically increase, reach a peak and then converge to zero. During the first iterations, the color difference in almost every sequence rises. Particularly, spectra with steep gradients show drastic color difference rises. This can be explained using the signal theory: Steep gradients imply high frequency components with considerable amplitudes, and therefore huge sum and difference frequencies after an amplitude modulation (multiplication). Hence, the constant bandlimit defined by the sampling rate becomes too low during the iterations.

Test 2 Random spectrum sequences in most cases generate less aggressive color difference increases (figure 4.5.b) because for most multiplications, steep gradients of the factors do not overlap regarding their wavelength ranges.

Conclusion

The subsequent multiplication of a spectrum by itself can be a good indicator for the required color difference level, although this approach does not allow for an *accurate* prediction of the maximum color error. Pretests in a global illumination renderer may classify spectra according to this finding if they include a reasonable tolerance between the estimated maximum color difference and the desired threshold.

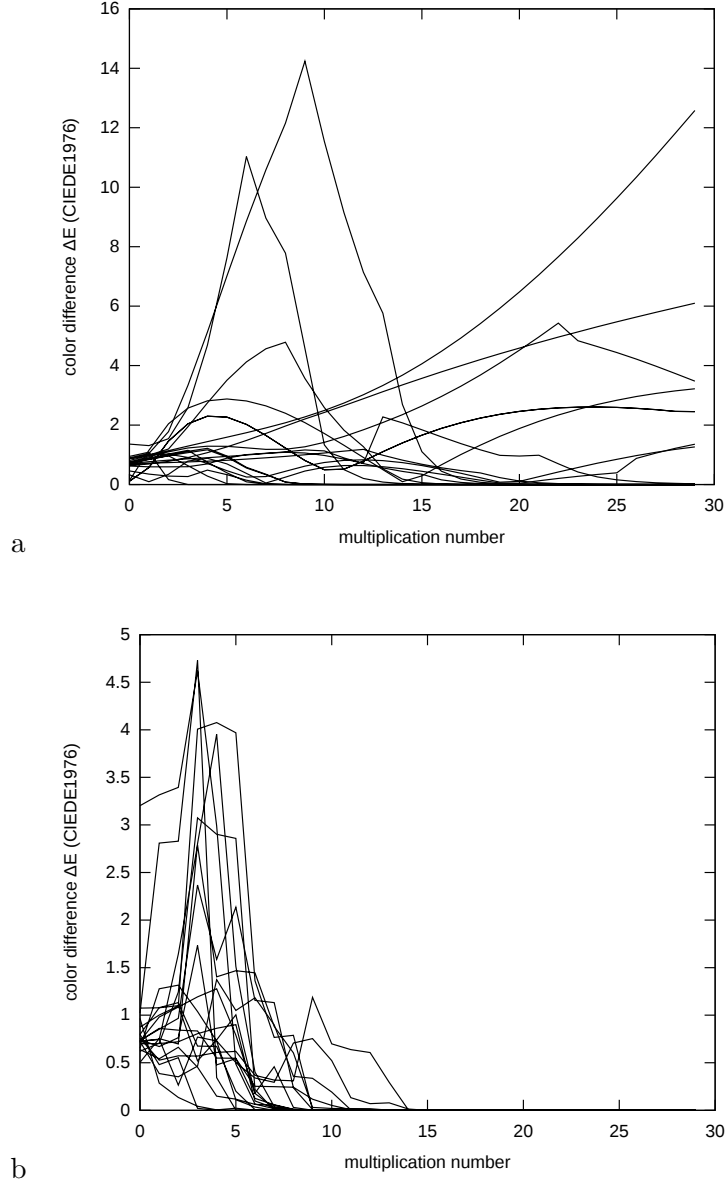


Figure 4.5: Development of the color difference between spectra sampled at two rates, (a) Test 1 plots (single spectrum sequences), (b) Test 2 plots (randomly chosen spectra).













































































































































 0.873	 0.992	 1.193	 1.943	 2.745	 3.511	 4.126
 0.700	 0.770	 0.820	 0.862	 0.902	 0.946	 0.993
 0.621	 0.596	 0.595	 0.599	 0.693	 0.887	 1.006
 0.758	 0.935	 1.102	 1.228	 1.291	 1.286	 1.235
 0.739	 0.875	 1.000	 1.114	 1.217	 0.959	 0.579
 0.718	 0.842	 0.967	 0.635	 0.284	 0.026	 0.002
 0.893	 1.045	 1.780	 3.337	 5.126	 7.017	 8.861
 0.456	 0.351	 0.283	 0.269	 0.487	 0.330	 0.146
 0.744	 0.888	 1.026	 1.153	 0.748	 0.393	 0.090
 0.948	 1.086	 0.167	 0.002	 0.000	 0.000	 0.000
 0.110	 0.572	 1.458	 2.043	 2.300	 2.266	 2.023
 0.728	 0.853	 0.968	 1.073	 1.167	 0.921	 0.555
 1.361	 1.308	 1.542	 2.582	 4.685	 7.630	 11.032
 0.807	 0.978	 1.139	 1.293	 1.443	 1.591	 1.743
 0.676	 0.716	 0.739	 0.759	 0.784	 0.816	 0.854
 0.873	 1.013	 1.020	 0.909	 0.710	 0.466	 0.254
 0.110	 0.572	 1.458	 2.043	 2.300	 2.266	 2.023
 0.638	 0.692	 0.781	 0.908	 1.071	 1.265	 1.483
 0.329	 0.096	 0.345	 0.790	 0.883	 0.693	 0.374
 0.135	 1.151	 2.041	 2.569	 2.814	 2.877	 2.813

Table 4.1: Test 1 results.













































































































































 0.700	 0.702	 0.720	 0.805	 0.863	 0.901	 0.098
 0.758	 0.527	 0.574	 0.571	 0.608	 0.620	 0.368
 0.504	 0.754	 0.698	 2.803	 3.952	 1.514	 0.402
 0.966	 0.286	 0.135	 0.044	 0.005	 0.022	 0.002
 0.725	 0.745	 1.643	 2.788	 1.403	 1.468	 1.445
 0.873	 1.002	 1.096	 4.007	 4.076	 3.968	 1.363
 3.203	 3.315	 3.395	 4.620	 2.991	 0.487	 0.066
 0.733	 0.853	 0.968	 2.368	 1.587	 2.136	 1.155
 0.797	 1.079	 1.133	 0.676	 0.671	 0.201	 0.008
 0.744	 0.670	 0.763	 1.734	 0.478	 0.543	 0.252
 0.651	 1.278	 1.316	 1.039	 0.729	 0.111	 0.035
 0.638	 0.542	 0.663	 0.453	 0.149	 0.117	 0.062
 0.797	 0.480	 0.554	 0.021	 0.000	 0.000	 0.000
 0.728	 0.981	 1.085	 1.192	 1.278	 0.789	 0.181
 1.074	 1.078	 1.101	 0.520	 0.740	 1.004	 0.178
 0.729	 0.748	 0.264	 0.770	 0.729	 0.502	 0.339
 0.733	 0.862	 0.842	 0.834	 0.549	 0.547	 0.126
 1.023	 2.810	 2.829	 4.729	 0.344	 0.002	 0.000
 0.621	 0.706	 1.243	 3.074	 2.902	 2.859	 0.220
 0.724	 0.387	 0.353	 0.465	 1.374	 1.050	 1.183

Table 4.2: Test 2 results.

4.2 Evaluation of Alternatives to Plain Equidistant Sampling

4.2.1 Basis functions

The sine or cosine functions of a fourier-transformed signal are only one of many types of basis functions. If sampled signals with n samples are regarded as n -dimensional feature vectors, given multiple training signals, *principal component analysis (PCA)* can be used to find *orthonormal vectors* that point into the directions of greatest variance in the point cloud of signal vectors in R^n . Each orthonormal vector is an orthonormal basis function. The basis functions can be scaled by *weights* and added up in order to reconstruct the individual signals. The less signals are involved and the more similar they are, the less basis functions are required for a sufficient signal reconstruction. That is, a PCA transform allows for superior compression rates of color spectra if few colors are being used and if they possess similar spectra. In many cases, five basis functions (and five weights per color spectrum) or less provide enough information for a proper signal reconstruction. *Gaussian quadrature* and further transformation methods based on basis functions can also be used for compression in similar ways. Compression using basis function usually comprises information loss.

Basis Functions and Color Operations

Spectra transformed using basis functions are tempting as an efficient replacement for three component colors if they can be represented using few weights [12]. A comparison is given in table 4.3. But the number of required basis functions (and weights) depends on the number of different spectra in a scene and also their distinctiveness. If the algorithm decides to use an amount of three basis functions, a multiplication already requires nine scalar multiplications (plus six scalar additions). Five basis functions require 25 multiplications, even though the majority of spectra could be sufficiently represented by 20 samples or less.

Reflection using Basis Functions:

- m basis functions $E_1(\lambda) \dots E_m(\lambda)$
- per light spectrum L : m basis function weights $\epsilon_{L,1} \dots \epsilon_{L,m}$
- per reflectance spectrum R : m basis function weights $\epsilon_{R,1} \dots \epsilon_{R,m}$

Reflectance matrix calculation for three coefficients and diffuse reflection:

$$M_R = \begin{pmatrix} \int_{\lambda} \epsilon_{R,1}(\lambda) E_1(\lambda) d\lambda & \int_{\lambda} \epsilon_{R,1}(\lambda) E_2(\lambda) d\lambda & \int_{\lambda} \epsilon_{R,1}(\lambda) E_3(\lambda) d\lambda \\ \int_{\lambda} \epsilon_{R,2}(\lambda) E_1(\lambda) d\lambda & \int_{\lambda} \epsilon_{R,2}(\lambda) E_2(\lambda) d\lambda & \int_{\lambda} \epsilon_{R,2}(\lambda) E_3(\lambda) d\lambda \\ \int_{\lambda} \epsilon_{R,3}(\lambda) E_1(\lambda) d\lambda & \int_{\lambda} \epsilon_{R,3}(\lambda) E_2(\lambda) d\lambda & \int_{\lambda} \epsilon_{R,3}(\lambda) E_3(\lambda) d\lambda \end{pmatrix} \quad (4.4)$$

Application of the reflectance matrix:

$$\begin{pmatrix} \epsilon_{out,1} \\ \epsilon_{out,2} \\ \epsilon_{out,3} \end{pmatrix} = M_R \begin{pmatrix} \epsilon_{in,1} \\ \epsilon_{in,2} \\ \epsilon_{in,3} \end{pmatrix} \quad (4.5)$$

<i>Operation</i>	<i>RGB/Three Components</i>	<i>Basis Function (n weights)</i>
Addition	3 fp additions	n fp additions
Multiplication	3 fp multiplications	n^2 fp multiplications $n(n - 1)$ fp additions

Table 4.3: Basis Functions: Computational Costs (fp...floating point).

4.2.2 Composite Model

Based on the observation that spikes occur sparse in the visible wavelength range of certain spiky spectra (see table 4.4), it is a promising approach to cut off the spikes and handle them separately from the rest of the spectrum, which is often wavy and free of steep gradients. Sun [16] proposed this approach in his *Composite Model*.

Daylight	< 10 major peaks
Mercury Vapor Lamp	5 spikes
Common Fluorescent Lamp	4 spikes
High Pressure Xenon Lamp	4 major spikes
Zirconium Concentrated-Arc Lamp	3 major spikes

Table 4.4: Selection of Spiky Light Sources (see [22]).

In fact, the Composite Model is a savvy decomposition of spectra that are obviously the sum of two distinct signals:

Smooth Signal	low frequency signal continuous entire wavelength range
Spike Signal	high frequency and amplitude > 0 at a certain wavelength or narrow wavelength range

The smooth part and the individual spikes can be considered as if they stem from separate light sources. For the smooth part, the sampling rate can be drastically reduced compared to the sampling rate required for the spectrum inclusive of spikes (in most cases 1/4 or less).

The Composite Model is tested in the implementation part of this work.

Chapter 5

Spectral Mipmapping

The idea of *Spectral Mipmapping* is to provide several accuracy levels for each color. When the color is used in a calculation, the appropriate level can be chosen. The term *Mipmapping* (The acronym “mip” is from the Latin phrase “multum in parvo,” meaning “many things in a small place.” [31]) is borrowed from *texture mipmapping*. Texture mipmaps are precalculated resolution levels of twodimensional images which are to be rendered in different sizes.

A typical use case of mipmaps are texture images on threedimensional objects that are placed in different distances to the viewer: If a texel is smaller than a pixel on the screen, the texel may be dropped and its information gets lost, there exist spacial frequencies ω in the texture image that are higher than the sampling frequency π/ω_g , which also leads to frequency aliasing if the human observer tries to reconstruct the image. In other words: Frequency aliasing influences the image appearance, hence bandlimiting should be applied. This is a costly process and therefore better done before the actual render process is started: Copies of the texture image having a resolution of 2^n are generated at resolution levels of 2^{n-1} , 2^{n-2} ,... During the render process, for each pixel, the color of the nearest texel per each of two best-matching resolution levels is taken and the final color is interpolated according to the level suitable for the texel’s distance. This approach is much faster than applying a bandlimiting filter for each pixel.

The Spectral Mipmapping approach translates this the concept of texture mipmapping from the image domain into the domain of spectral color representations, but instead of bandlimiting, the primal objective is time-efficiency: The rendering process of a full-spectral raytracer should become faster by reducing the number of scalar operations that are part of color calculations as much as possible, without affecting the visual appearance of the resulting image.

Based on the experience that equidistant samples of color spectra can be used in a straightforward way in sample-by-sample scalar operations that allow for compiler optimizations, they are the feedstock of this approach. Light and reflection spectra that occur in the scene are available in the form of high-resolution representations, comparable to the original image that is to be filtered using texture mipmapping. For instance, such a representation consists of 36 floating point numbers (or even 81 to represent spiky/turbulent spectra). At first glance, it makes sense to reduce the number of floating point numbers as far as possible, depending on the dominance of high frequency oscillations in the signal of each individual spectrum. This is a reasonable approach, but it poses the question how two colors represented by different

Level 0	Class of length-32 vectors
Level 1	Class of length-16 vectors
Level 2	Class of length-8 vectors
Level 3	Class of length-4 vectors

Table 5.1: Levels as Vector-Length Classes.

numbers of floating point values should interact, e.g. equal white light, that requires less than four samples for a sufficient representation, reflected by a green surface that is bandlimited to the range of 530 to 550 nm, with a reflectance of nearly 1.0, requiring 16 or more samples. (The light-surface interaction is an element-by-element multiplication of two floating point vectors.) There are two possible solutions to this question:

- Either try to find a way how to multiply two vectors of different lengths,
- or provide a color representation for the light's color that holds the same number of floating point numbers as the surface color's representation does and vice versa.

The first solution has not turned out satisfactory, particularly for reflection multiplications. For instance, if a 16-sample light color is to be multiplied by an 8-sample surface color, this can be done in an efficient way if each surface sample is spread onto two light samples. This results in a 16-sample reflected light. Unfortunately, due to the missing interpolation, sawtooth artifacts are introduced that considerably impair the result's accuracy, even though it provides 16 samples for reconstruction.

The latter solution remedies this problem. It implies that for each class of n_{SCL} occurring classes of *surface color levels*, each light color that occurs in the scene must be represented by n_{SCL} vectors of the according lengths, and vice versa for each class of n_{LCL} occurring *light color levels*. If a scene has two lights and 15 surface colors, each light color is represented by 15 sample vectors or less with different numbers of samples, and each surface color of no more than two with different numbers of samples. Suchlike dimensions are reasonable. But for optimization reasons, a predefined number of vector length classes is preferred (e.g. four classes: 4, 8, 16 and 32 floating point values).

Thus, each light or surface color, they are known before the rendering process starts, is to be classified. A color is provided in the form of $Cl_{surf} \in Cl$, where Cl is the set of vector length classes and Cl_{surf} is the set of vector length classes with vector lengths (numbers of samples) that sufficiently represent the color. Hence a mipmapped color consists of the following parts:

- One or more vectors of floating point numbers that represent the color's spectrum. (A color of level l must also be available for levels $0 \dots l - 1$.)
- An integer number that identifies the class of the color's minimum length vector, called *level* (see table 5.1).

5.1 Resampling and Downsampling

If the sample spacing or phasing of color spectra taken from a database does not correspond to the sampling defined for the most accurate level (*Level 0*), the resampling method described by figure 5.1 can be applied.

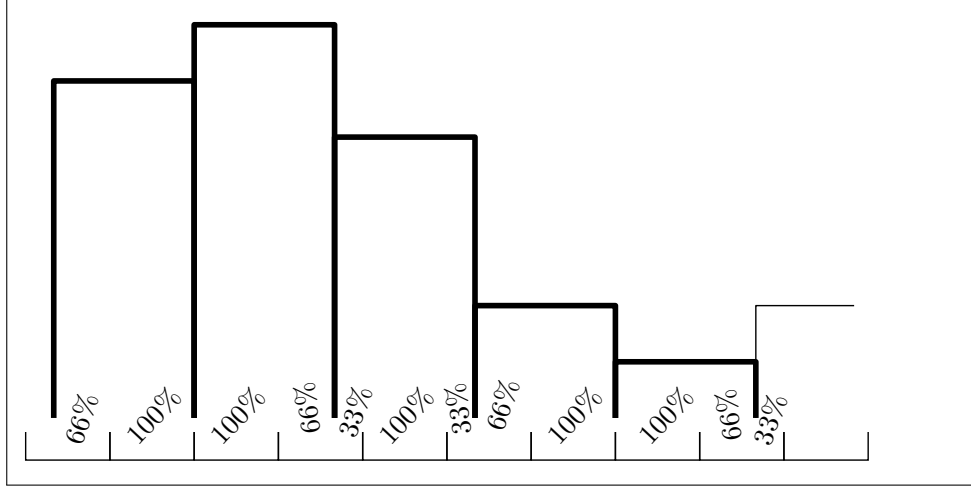


Figure 5.1: Level Generation: Resampling.

It is used in the test implementation (chapter 6). The percentages denote the intensity flow from sample to sample. It is applicable for downsampling as well as upsampling.

Downsampling: Advantage over Dropping Samples If samples are simply dropped, the intensities of the dropped samples disappear entirely and do not contribute to the new samples, which results in unnecessary information loss. See also section 3.2.2.

Downsampling for Level Generation

Figure 5.2 depicts the relation between two possible mipmapping levels, for instance *Level 0*, $l = 0$ (upper samples) and *Level 1*, $l = 1$ (lower samples). The upper samples are spaced by intervals of $\Delta\lambda_{l=0} = 10nm$, the lower samples by intervals of $\Delta\lambda_{l=1} = 2\Delta\lambda_{l=0}$, shifted by $\Delta\lambda_{l=0}/2$.

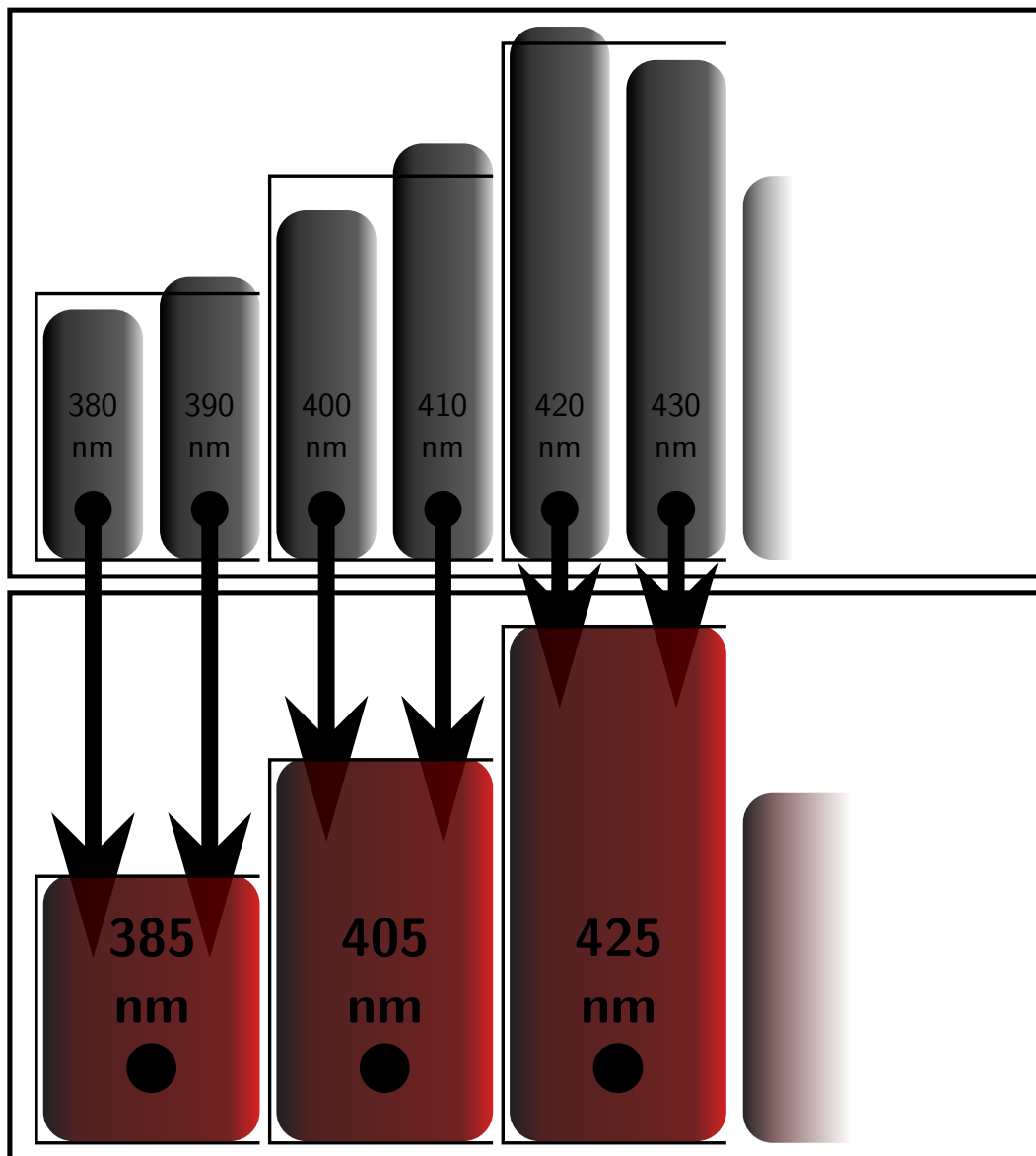


Figure 5.2: Level Generation: Downsampling.

Chapter 6

Implementation

The implementation serves the following purposes:

- Gain information about the performance impact induced by full-spectral colors.
- Eliminate specific bottlenecks (instruction count and instruction cache faults).
- Test the Spectral Mipmapping approach in combination with the Composite model by Sun [16].
- Appraise the relevance of color difference measures applied in pretests for performance optimization purposes.

6.1 The Renderer

Since this work is focused on fundamental research regarding spectral rendering, a simple ray tracer with the essential capabilities of a global illumination renderer and a well arranged source code structure was chosen for the test implementation. *Minilight* [32] is a renderer that largely fulfills these requirements.

6.1.1 Description of the *Minilight* Ray Tracing Renderer

Minilight is a basic global illumination ray tracing renderer that features Monte-Carlo path-tracing transport. It is currently available on the web at <http://www.hxa.name/minilight/> [32] under the “New BSD License”, inclusive of the entire source code in various programming languages. The C++ implementation was chosen for the research this thesis is based on and preferred over the C version for human readability reasons.

The C++ source code of the renderer can be compiled by every standard-compliant C++ compiler, for instance G++. No additional libraries are required.

The scenes are defined in human-readable scene files by specifying geometric objects in the form of absolute triangle vertex positions and their metameric colors in the form of RGB triples: emitted light colors as well as diffuse surface colors (reflecting colors). During the rendering process, which iteratively invokes the raytracing algorithm a predefined number of times, the result is successively refined and stored in a PPM format image file, at most

after each iteration. The PPM format allows for plain text storage with arbitrary debug information in the form of comment lines.

The C++ source code of Minilight is structured and compact and thus can easily be extended for academic and research purposes with minimum effort. Extensions (such as full-spectral color computations) can be added in the form of additional classes requiring a minimal number of changes in the preexisting code.

<i>Geometric Object Representation</i>	triangle primitives
<i>Spatial Data Structure</i>	Octree
<i>Surface Type (Emitter, Reflector)</i>	ideal diffuse
<i>Light Propagation</i>	Monte-Carlo, Importance Sampling
<i>Ray Termination</i>	Russian Roulette based on reflectance magnitude
<i>Scene Input</i>	plain text
<i>Image Output</i>	tone-mapped RGB to PPM (binary or plain text)

Table 6.1: Minilight Renderer Capabilities Summary.

Ray Tracing Algorithm and Tone Mapping

Through each pixel of the resulting image, one ray starting at the imaginary camera is sent into the scene. For example, for an image size of 320×320 pixels, 102400 rays are sent. The ray is handed over to the raytracing algorithm, which intersects it with the objects in the scene. At the first intersection point, the following color calculations are executed in order to determine the overall radiance at the intersection point:

- The radiance is set to the light *emitted* from the triangle surface.
- *Emitter Sampling*: One emitter of the scene is randomly selected, its light is added to the radiance.
- *Monte Carlo path tracing*: Starting at the intersection point, the ray is successively reflected at further intersection points in the scene in random directions (*cosine-weighted importance*) in recursive function calls. Recursion levels are added until the reflectance magnitude of the currently intersected surface is below a random threshold that differs from recursion level to recursion level (*russian roulette*). This ensures the recursion will most likely terminate with a finite number of levels, i.e. in finite time. Emitter sampling, as described above, is performed in each recursion level, but local emission directly at the intersection point is solely added once in the first recursion level.

This procedure is repeated in multiple iterations until the user terminates the process or a predefined iteration limit is reached.

The pixels of the scene image are multiplied by a tone mapping factor before they are written to the result image file (*linear tone mapping*). Therefore, the final luminance of each pixel depends on all other pixels. As tone mapping affects lightness, it influences the perception of color differences/the just-noticeable difference threshold between two images rendered at different quality settings. In other words: The quality of an image is difficult to estimate before the tone mapping step.

Color calculations

The original Minilight renderer deals with three-component colors (RGB triples) that represent the light and surface colors in the scene. They are stored in three-dimensional vector objects the class of which provides operators for element-wise addition/subtraction and multiplication as well as multiplication/division by scalars, and further important methods such as dot product calculation. Basically, all color related operations and calculations are encapsulated in this vector class. If color computation is to be adapted, almost all changes can be done inside a copy of this class (the original vector class must be kept unmodified for calculations in the geometric part of the renderer), unless performance optimizations are to be done. In the rest of the code, solely the type declarations are to be adjusted to the name of the new color class. However, performance-related weaknesses, predominantly unnecessary object constructions and copies, most likely necessitate substantial changes to the code to allow for meaningful performance tests.

6.2 Adaptation of an RGB Renderer for Full-Spectral Rendering

The following explanation is based on Minilight C++.

6.2.1 Optimized Code Design

Because of their compactness, three-component color objects (as well as 3D vectors in geometric computations), containing three floating point values, are handled like atomic data type variables. They are constructed in stack-based memory and copied between procedures. But when dealing with full-spectral colors, one color may consist of up to 80 or even more floating point values in a vector representing the spectrum's point or area samples. Hence, each copy operation is to be uncovered and scrutinized.

Unnecessary color object copies usually occur...

- when incoming light colors are accumulated at certain points in the raytracing process (the colors are copied from subroutines)
- and, in general, whenever overloaded *binary operators* such as `*` or `+` are applied.

Binary Operators

For color calculations, the *Color+Color*, *Color*Color* and *Color*float* operators are defined and used for RGB operations. They are way too inefficient if applied as arbitrary-dimensional vector operations as it is the case in spectral color computations, because the operation's result is stored on stack-based memory and copied to the target object when the operator method returns.

Therefore, each occurrence of an overloaded binary operator should be replaced by a method of the form

```
void (TypeC& out, const TypeA& inA, const TypeB& inB)
```

which gets passed a reference to the target object. In most cases, an *in-place operation* of the form

```
void (TypeC& inOut, const TypeB& inB)
```

is applicable.

Color-Centered View of the Renderer

In other words, the hierarchy of functions that handle color information is to be flattened.

A common ray tracing algorithm can be regarded as an information pipeline with one main thread and some branches. Indirect illumination requires the main thread to be recursively called, i.e. it is a branch of itself. In each branch, color information is gathered from surfaces or lights, mixed by color-by-color multiplications and accumulated in the main thread.

The Minilight renderer is a prime example of a ray tracer implementation which entirely ignores the room for improvement that arises from this aspect.

Original design of the Minilight ray tracer:

- `getEmission` equation 6.1
- `sampleEmitters` equation 6.2
- `getRadiance` equation 6.3

The following equations describe the color information flow for a single pixel over multiple iterations. All bold symbols name \mathbb{R}^3 vectors which carry either geometrical positions or RGB triples.

$$\mathbf{p}_{e,i,j,k} = \mathbf{e}_{i,j,k} \frac{A_{etri,i,j,k}(-\mathbf{v}_{eray,i,j,k} \cdot \mathbf{n}_{etri,i,j,k})}{d^2} \quad (6.1)$$

$$\mathbf{p}_{se,i,j} = (\mathbf{p}_{e,i,j} n_e) \mathbf{r}_{rtri,i,j} \frac{|\mathbf{v}_{ray,i,j} \cdot \mathbf{n}_{rtri,i,j}|}{\pi} \quad (6.2)$$

$$\mathbf{p}_{i,j} = \mathbf{p}_{e,i,j} + \mathbf{p}_{se,i,j} + \frac{\mathbf{r}_{rtri,i,j}}{\mathbf{r}_{rtri,i,j} \cdot \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}^T} \mathbf{p}_{i,j+1} \quad (6.3)$$

$$\mathbf{p}_i = \mathbf{p}_{i-1} + \mathbf{p}_{i,0} \quad (6.4)$$

(Symbol descriptions are given on the following page.)

The variable $\mathbf{e}_{i,j,k}$ together with the numerator in the fraction of equation 6.1 correspond to the ϵ term of the rendering equation, whereas the denominator represents the g term. The fraction in 6.2 corresponds to the ρ term. The I term of the rendering equation is represented by \mathbf{p}_{\dots} variables. (The rendering equation is described in section 1.1.2.)

$i \in \{1, 2, \dots, n_{iter}\}$	iteration index
$j \in \mathbb{N}_0$	recursion index
$k \in \{1, 2\}$	$k = 1$: local emitter $k = 2$: sampled emitter
$\mathbf{e}_{i,j,k}$	emitter color
$A_{etri,i,j,k}$	emitting triangle area
d	reflector-emitter distance
n_e	number of emitters
$\mathbf{r}_{rtri,i,j}$	reflector color
$\mathbf{v}_{ray,i,j,k}$	ray to emitter
$\mathbf{v}_{ray,i,j}$	ray to reflector
$\mathbf{n}_{etri,i,j,k}$	normal vector of emitting triangle
$\mathbf{n}_{rtri,i,j}$	normal vector of reflecting triangle
$\mathbf{p}_{...}$	“power” (intermediate colors)

Pipeline Optimization: Recommendations for Improvement

The values of $\mathbf{p}_{e,i,j,k}$, $(\mathbf{p}_{e,i,j}n_e)$, $\mathbf{p}_{se,i,j}$, $(\mathbf{r}_{rtri,i,j}/\mathbf{r}_{rtri,i,j} \cdot (1 \ 1 \ 1)^T)$ and $\mathbf{p}_{i,j}$ are stored in intermediate stack variables in the ray tracing algorithm.

The following improvements are recommended for color representations with considerable memory consumption (e.g. spectral samples):

- Replace the mentioned intermediate stack variables by preallocated class variables that can be reused over all iterations.
- The pipeline structure allows for a throughout use of in-place operations.
- If equidistant samples are used, combine assignment or addition and scaling in hybrid operations. This saves one iteration over the sample set per pass.

6.2.2 Implementation of the Composite Model

(see also section 4.2.2)

Spike Detection Sun suggests to use the gradient information in order to detect spike occurrences. This obviously works well with spikes the energy of which is completely represented by one sample. Wider peaks may possess foothills with gradients smaller than the threshold of the detector. That is, even though the foothills belong to a spike, they remain among the rest of the spectrum. High foothills unnecessarily introduce high-frequency oscillations and thus may prevent resampling at low rates. Therefore, the gradient threshold should be as low as possible: slightly higher than the steepest gradient of the smooth part of the spectrum, hence individually set for each (spiky) spectrum. In the test implementation, a more sophisticated two-pass approach is used: A Gauss filter is applied to blur the spectrum. The result is compared to the unblurred spectrum in order to classify each sample as spike/no spike. After the peaks are removed, the result is blurred once again to detect the remaining foothills and classify them as parts of a spike.

Sun expresses spikes in terms of *delta functions*. In fact, information about one spike (or foothill) is completely represented by two floating point values, the first of which represents the spike location, i.e. its *wavelength*, and the second represents the spike's *height* (the distance between the smooth part of the spectrum and the spike's peak).

A color object contains the samples of a color spectrum in a field of floating point values that is allocated on the heap. The Composite model can be added on without changing this basic design, in further heap space.

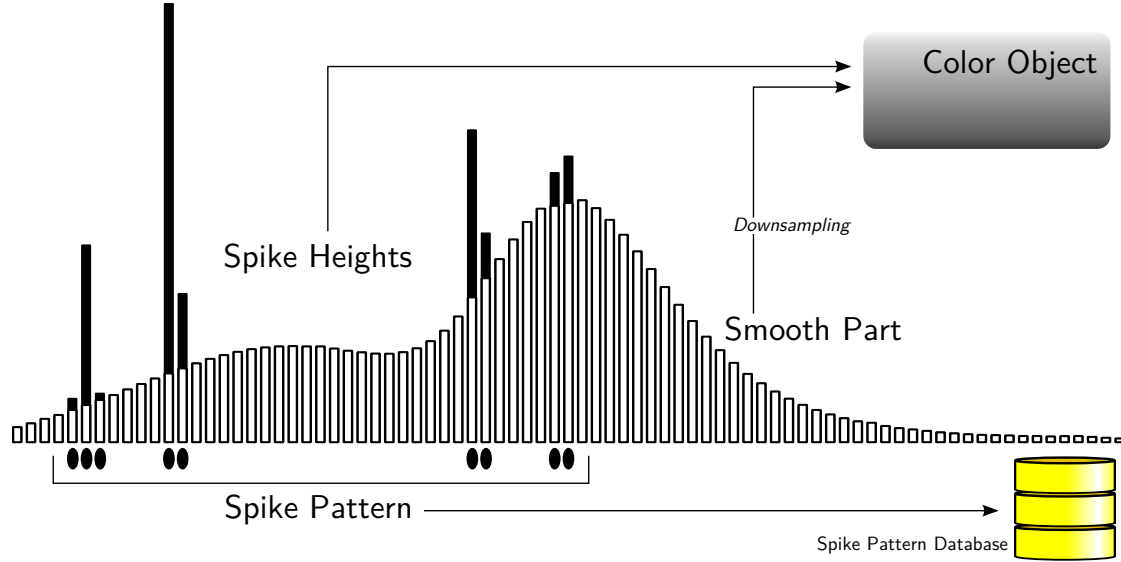


Figure 6.1: Spike Separation.

Pattern Database As described above, a spike/foothill is represented by one location and one height value. Height values have a different scope than location values: Spike heights are changed over and over again in the information flow during the raytracing process, whereas location values never change. Rather they are fixed *patterns* that can be stored outside the color objects in a small database that holds each spike pattern that occurs in the scene once in a field of floating point values. The spike patterns are directly accessible by the raytracing procedures.

Consequently, color objects only allocate storage space for spike heights, but not for spike locations.

Preparation of Spectra Assuming the source colors are provided to the renderer in the form of equidistant samples with small distances of 1, 5 or 10 nm, spikes are to be separated as described above and afterwards, the smooth parts can be resampled at lower rates (please refer to section 5.1). (The preparation process can try to resample the spectrum even if no spikes are detected.) After the preparation process, spiky spectra hold two heap addresses and one pattern identifier (field index or heap address): the former address the sample field and spike height field, respectively, whereas the latter references the spike location pattern field.

Colors with multiple Spike Patterns occur if more than one light source with a spiky spectrum is part of the scene. They emerge from additions of light colors which stem from

different spiky light sources. *So, a color object is capable of storing more than one spike height field.*

All operator methods that are already provided for flat and wavy spectrum operations (operations on equidistant samples) are to be equipped with appropriate spike operations. One peculiarity is common for spike operations in general: Since one color object may hold multiple spike height fields, the spike operation must be capable of handling two or more height fields consecutively. Apart from that, the operations are quite different:

Addition Color $c = \text{Color } a + \text{Color } b$ If the addition is not in-place (i.e. $c \neq a$), place the spike heights of a in newly allocated heap space owned by c . Add the spike heights of b to the spike heights of c for patterns that occur in both a and b or else place them in newly allocated heap space owned by c .

Multiplication Color $c = \text{Color } a * \text{Color } b$ The multiplication operation for spikes is considerably more expensive, as both the spiky part and the smooth part of the color are involved. If one can assume that surface colors are free of spikes, a is the light and b is the surface color, it is sufficient to multiply the spikes of a with the samples of b . Otherwise, this must also be done the other way round and, moreover, it might occur that spikes of the surface color overlap with spikes in the light. Thus, spike-by-spike multiplications must be carried out if necessary.

The most important part of the multiplication operation with spikes is the spikes-by-samples multiplication. Since a spike may be located between two samples (as a result of resampling the spectrum at a lower rate), the value of the spectrum's smooth part at the spike's location must be interpolated. For performance reasons, *linear interpolation* is the sole possible choice: It does not require information about more than two neighboring samples and is way less expensive than spline-interpolation, which requires a system of equation to be solved, or more complicated interpolation methods.

For each spike, the spike-by-sample multiplication requires at least 4 scalar-by-scalar multiplications plus 4 scalar-by-scalar additions/subtractions.

Multiplication Color $c = \text{Color } a * \text{float } b$ This operation is a simple scaling of the spike heights.

6.2.3 Mipmapped Ray Tracing Pipeline with Level Feeler

The level (vector-length class) of operation is determined for an entire pixel iteration, as it is described by the equations 6.1–6.4 on page 48, to avoid a costly parallel processing of multiple levels. Thus, all operations and intermediate variables between scene color queries and final pixel accumulation are single-level.

Level Feeler

At the beginning of each pixel iteration, the level variable is set to the least accurate level (minimum vector length). That followed, in the course of the ray tracing process, the level feeler collects references to emission and reflection colors and lifts the level variable value whenever a color requires a more accurate representation. All geometrical calculations are performed and the calculated attenuation factors are kept in scalar variables. Finally the

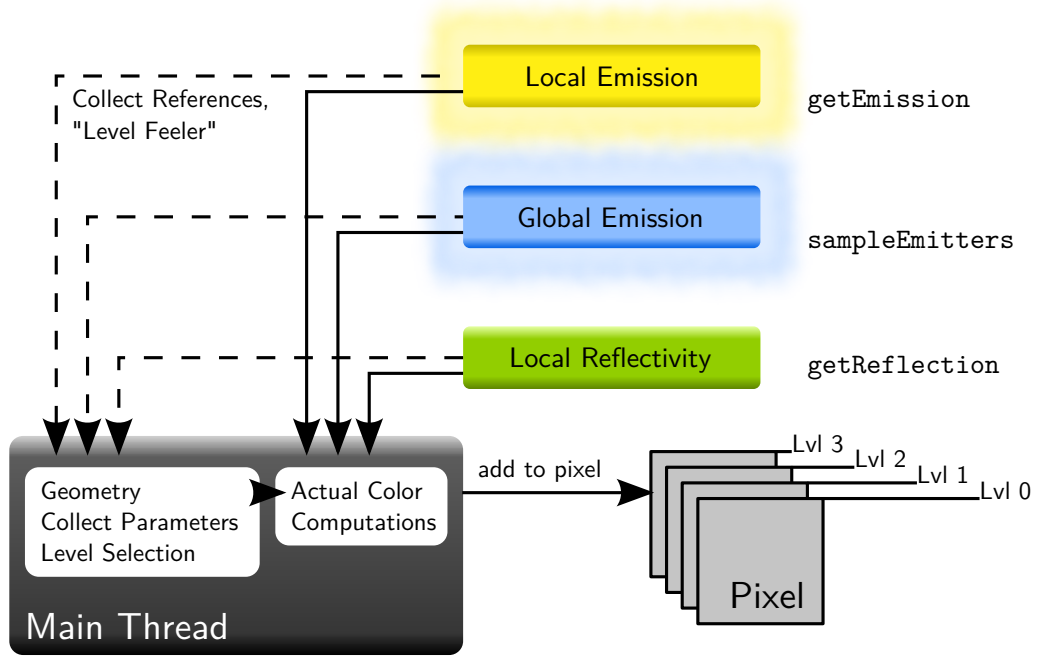


Figure 6.2: Spectral Mipmapping Color Pipeline Thread.

required level, references to all contributing colors and attenuation factors are known and the actual color computations can be performed in a compact code block using the selected level.

Summing up, the level feeler probes the level required for the color operations, based on the most accurate level among the levels of contributing colors in the pixel iteration.

Since the level is determined individually per iteration, after multiple iterations one pixel contains one or more sample vectors with different sample counts.

6.2.4 Preprocessing

The preprocessing phase takes place before the actual rendering process starts. Basically, a scene's light and surface spectra are resampled at the minimum required accuracy level and the levels with higher accuracy.

The minimum required level is determined by pretests, for instance:

- A** Compare resampled spectrum to reference and compare the resulting color difference to a predefined threshold, for instance $\Delta E = 2.0$.
- B** Multiply the spectrum several times by itself, as the conclusion of section 4.1.3 suggests, perform **A** on the result.
- C** Do **B** with randomly selected spectrum sequences.

Pretests of type **A** and type **B** can be accomplished with reasonable computational effort. **C** is especially promising if a huge amount of random sequences is appended to each spectrum occurring in the scene.

6.2.5 Postprocessing

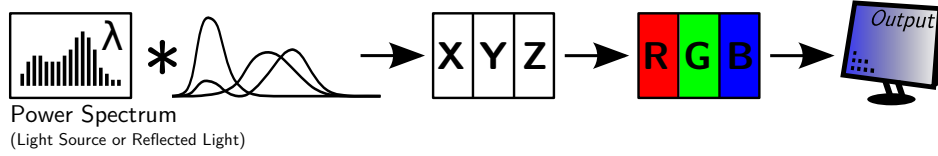


Figure 6.3: Postprocessing.

Postprocessing is done after the final spectral colors are stored for each pixel. If Spectral Mipmapping is implemented, each pixel may contain distinct spectra at different levels, as well as spikes height fields of different patterns.

Spectrum to XYZ

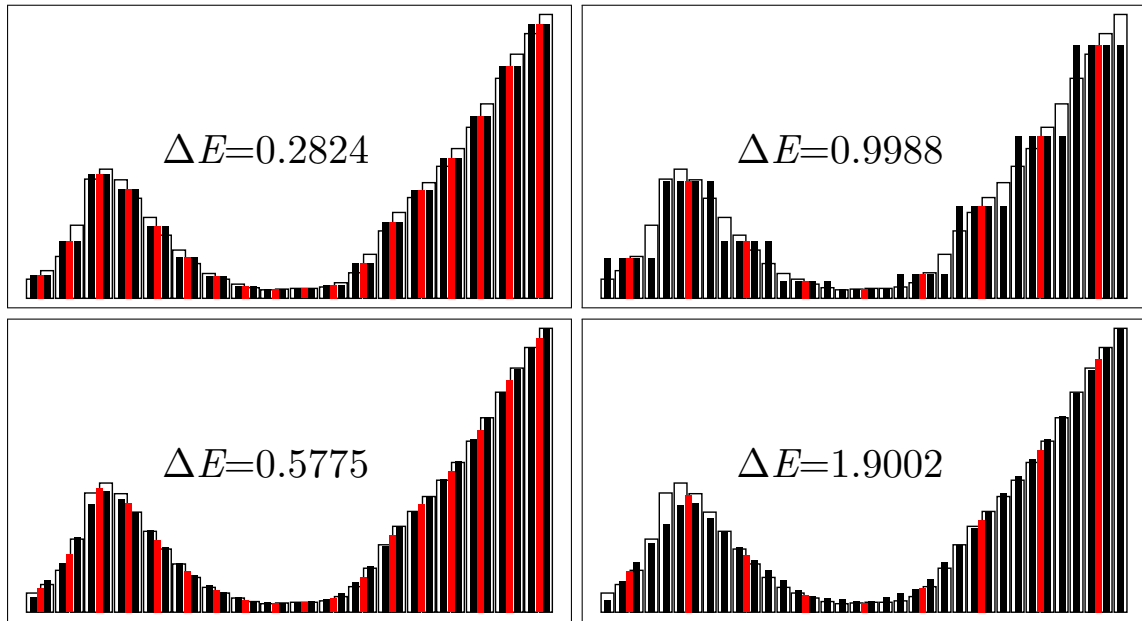


Figure 6.4: Reconstruction of a Spectrum with and without Interpolation.

Figure 6.4 illustrates reconstruction examples. The unfilled bars show the original spectrum, the red bars show the downsampled equivalent and the black bars the reconstructed spectrum. For calculating the ΔE values, the XYZ triples were transformed into the $L^*a^*b^*$ color space. The ΔE statements inform about the perceived difference between the original spectrum and its reconstruction. The plots in the upper row show simple reconstructions whereas the lower row plots show linearly interpolated reconstructions. The interpolation acts as a smoothing filter and extends the sphere of influence of the downsampled representation's samples beyond

their wavelength interval, which amplifies the aberration. Therefore, linear interpolation is inapt for reconstruction.

Spikes to XYZ

The spike height field is iterated and, according to the spike pattern, the appropriate X, Y and Z values are extracted from the color matching function table. Potentially, spike patterns define arbitrary spike positions in the form of wavelengths. To account for this, X, Y and Z values can be interpolated during reconstruction in order to accomplish maximum accuracy.

Final Output Conditioning

The resulting XYZ values are added up to gain the final pixel color. The final value is converted to RGB and tone mapped as described in the appendix A.1. *Caution:* Tone mapping considerably rescales the three-component color.

Chapter 7

Results

7.1 General Test Description

The following tests were run under three optimization conditions on an *Intel Core 2 Duo* processor. For a detailed description of the processor, please refer to the appendix B.

No Optimization Compiler optimizations are disabled.

Compiler Optimization The maximum possible optimization level (*-O3*) is requested from the compiler to ensure inlining and the actual use of SIMD operations in the *Compiler+Vector Optimization* condition.

Compiler+Vector Optimization Co-operation of compiler and vector optimizations.

The pure *Vector Optimization* condition was dropped because vector optimization is done by the compiler solely if compiler optimization is turned on.

Each test is headed by a reference run with a constant number of 81 equidistant samples per color (Spectral Mipmapping and the Composite Model are disabled in this run). The reference and test cases are described below in section 7.2.1.

Pretest Each spectrum is resampled at the rates of the levels. The reference white for XYZ-to-L*a*b* transformation is

$$\begin{pmatrix} x_{rwhite} \\ y_{rwhite} \\ z_{rwhite} \end{pmatrix} = \begin{pmatrix} 0.4124564 & 0.3575761 & 0.1804375 \\ 0.2126729 & 0.7151522 & 0.0721750 \\ 0.0193339 & 0.1191920 & 0.9503041 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

where the unit vector represents the brightest possible RGB output. The matrix is the inverse of the XYZ-to-sRGB matrix (D65 illuminant).

Pixel Color Difference: Blur Filter In order to reduce noise artifacts, all images are compared in blurred form. The applied blur filter is 5×5 pixels wide.

7.1.1 Questions and Answers (regarding all tests)

Q: Why the sample counts 80, 40 and 20?

A: 80 samples are used for spiky spectra, 40 and 20 for wavy spectra. For most spectra, less than 20 samples are sufficient, some require more (less than 40). The chosen sample counts are multiples of four in order to optimally exploit SIMD 4-parallel vector instructions. Consequently, further recommended sample counts are 8, 16 and 32.

Q: Why is the minimal sample count not less than 20? For many spectra, far less samples are sufficient. Why no more than three levels?

A: Because the raytracer is controlled by random parameters, reflections and emitter hits and their colors cannot be predicted and thus not sorted by level. Hence the current level will often vary from ray to ray. Therefore, level-specific instructions for each level should be kept in the instruction cache in order to avoid time-consuming cache-faults. The less levels are generated, the less level-specific code must fit into the instruction cache of the CPU. If vector optimization is turned on, the number of consecutive multiplications drops to 5 at the 20-sample level. This is a reasonable instruction count, in particular if the avoidance of further accuracy levels reduces the risk of cache faults.

In the following tests, the times of three accuracy levels (80, 40 and 20 samples) are compared. These results can be used for extrapolations of expected times for levels with less than 20 samples, if such an implementation is intended.

Q: Why a threshold of $\Delta E_{ab,thresh}^ = 5.0$?*

A: This threshold is a good choice for the test computations insofar as it is well above the average just-noticeable difference threshold of $\Delta E_{ab}^* = 1.0...2.0$: Aberrations of $\Delta E_{ab}^* > 5.0$ are expected to stand out in the test images.

7.1.2 Time Measurement

Source code optimizations were exclusively performed in the color-related, geometry-independent part of the program, in order to evaluate the performance gains induced by Spectral Mipmapping as well as spike separation using the Composite Model. Therefore, the following stopwatches are regarded as representative:

- Color-related code chunk stopwatches
- Per-operation stopwatches

The former measure the time consumed by the color-related code chunks, the sum of which represents the entire (time-critical) color-related code of the renderer. The latter measure the time consumption of atomic operations such as color-by-color multiplications or spikes-by-color multiplications.

The stopwatches make use of a high-resolution per-process timer from the CPU. Time that elapses between two timer queries is determined prior to each time measurement and subtracted from the result in order to minimize adulterations induced by the stopwatches. A detailed description of the stopwatch implementation is given in appendix B.

RDTS The time measurements are validated using cycle count measurements provided by the *rdtsc* machine code instruction (*rdtsc* is the abbreviation for *read time stamp counter*), which is described in detail in the appendix.

The times are listed in seconds for all test results.

7.1.3 Test Colors

The used colors and their spectra are listed in the appendix. In the test descriptions, they are referred to by L_{\dots} (light color) and R_{\dots} (reflector color), respectively.

Light Colors One equal light (L_1) and one non-equal, spiky light (L_2) are used as light colors, except for the multi light scene. L_2 is provided to the renderer at an accuracy of 81 equidistant samples.

Surface Colors The surface colors are sampled *Macbeth Colour Checker* patches.

7.2 Tests

7.2.1 Standard Cornell Box

This test is focused on *visible errors* and *time efficiency*. The resulting images are shown in figure 7.3. All times and cycle counts (*rdtsc*) averaged over 200 iterations. Color difference threshold in pretest: $\Delta E_{ab,thresh} = 5.0$.

<i>Level</i>	<i>Sample Count</i>	<i>Times Emit.</i>	<i>Times Refl.</i>
Level 0	80	0	0
Level 1	40	0	0
Level 2	20	2	32

Table 7.1: Standard Cornell Box: Level Decisions.

Spectra The scene coloring consists of light-wavy spectra only (R_1 , R_2 and the rest of the walls as well as the boxes R_3). Hence, without exception, the pretest always decides for level 2 (least-accurate level), for reflectors as well as emitters (table 7.1). The intention of this scene is to exhaust the maximum possible performance gain induced by the mipmapping approach.

Visible Errors Figures 7.3.c and 7.3.f respectively show the color difference distribution on the image plane. In the pretest, the color errors for the red and orange walls reached $\Delta E_{ab,max}^{*}(pretest) = 3.31$ and 3.03 respectively. The color errors in the resulting image are greater than 6.0 for 12 pixels at the area of the red wall and less for all other pixels. This result is within the expectations based on the findings in section 4.1.3. The mean error is $\Delta E_{ab,mean}^{*} = 0.563$.

Time efficiency The color-related operation times are listed in table 7.2. If the compiler translates the C++ code command-by-command into machine instructions (case *No Opt.*), roughly 140 milliseconds are spent on processing color-related code (see figure 7.1). Since the scene colors are all sampled at the lowest rate (20 samples), in the test cases (*ML* and *ML-C*), the computation costs considerably drop to less than 50 milliseconds, which is roughly a third, even though not a quarter as expected if the sample count ratio of 20/81 is regarded. *ML* and *ML-C* roughly differ because the scene is spikeless. For this scene, the Spectral Mipmapping approaches are obviously superior to fixed sample count implementations. Compiler optimization *Compil.Opt.* brings a performance improvement of more than 3/4 and an overall improvement of roughly 90% if combined with sample reduction. The contribution of vector operations is marginal *C+Vec.Opt.* if compared to compiler optimization in general, but nevertheless they are a welcome bonus that is ease to achieve. Considering that the vector operations merely parallelize four operations, a greater amount of parallel operations as provided by recent GPUs can particularly be beneficial for high-sample count spectra.

Figure 7.2, stemming from an earlier stage of development, gives an impression of the influence of function calls on the relative differences between the reference and test cases: For this measurement, operations are located in functions and the function call overhead considerably diminishes the relative improvement by the lower sampling rate. This suggests that the renderer should be designed using aggressive inlining instead of function calls.

	<i>No Opt.</i>	<i>Compil.Opt.</i>	<i>C+Vec.Opt.</i>
<i>Ref. (seconds)</i>	0.128971	0.0262309	0.0283894
<i>ML (seconds)</i>	0.0385935	0.0133063	0.0132475
<i>ML-C (seconds)</i>	0.0390271	0.0126055	0.012462
<i>Ref. (rdtsc)</i>	3.08742e+08	5.11619e+07	5.43243e+07
<i>ML (rdtsc)</i>	6.34737e+07	2.23608e+07	2.33346e+07
<i>ML-C (rdtsc)</i>	6.72401e+07	2.37181e+07	2.50209e+07

Table 7.2: Standard Cornell Box: Average single-iteration time in seconds and cycles measured by *rdtsc* spent processing color-related code.

Description of the table rows:

***Ref.* = Reference Run** 81 samples (fixed).

***ML* = Multi-Level** Spectral Mipmapping algorithm, levels automatically chosen: 20...80 samples.

***ML-C* = Multi-Level Composite** Spectral Mipmapping algorithm with Composite Model implementation: levels automatically chosen, 20 or 40 samples, spikes separated.

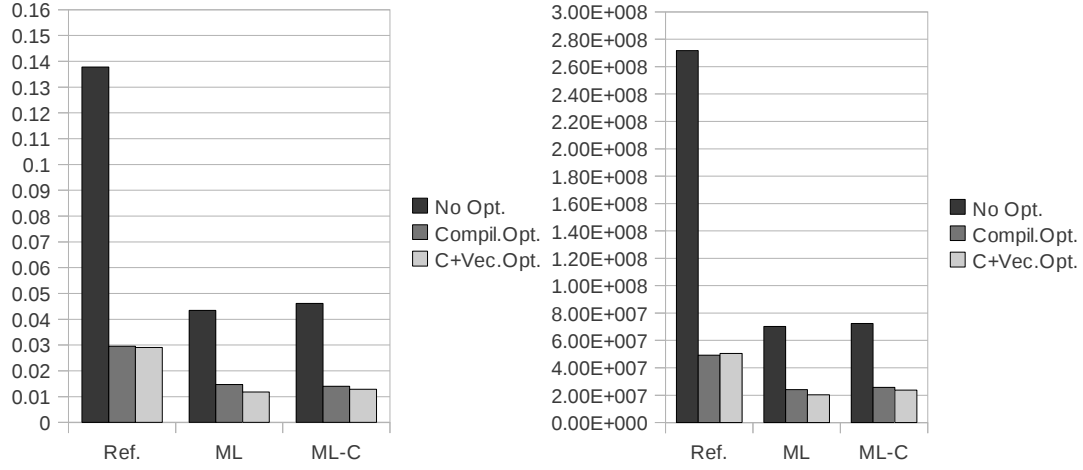


Figure 7.1: Standard Cornell Box: Charts of table 7.2.

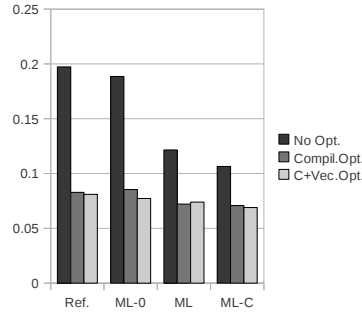


Figure 7.2: Standard Cornell Box: Overhead of Function Calls.

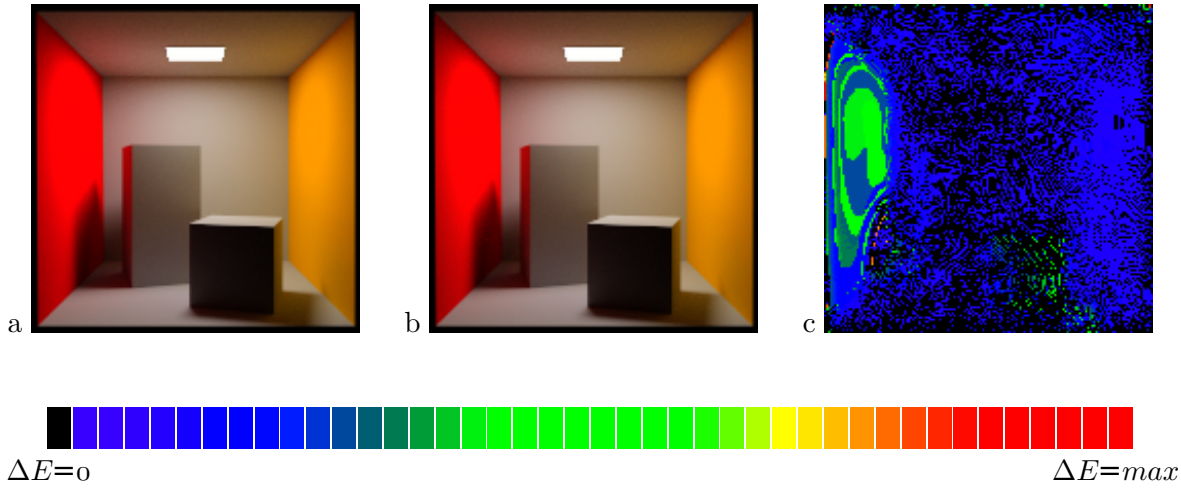


Figure 7.3: Standard Cornell Box. Reference image (a, d), *ML* result (b), *ML-C* result (e), color differences ΔE (c: a vs. b; f: d vs. e). The horizontal bar shows the color coding of ΔE .

Spiky light in the Cornell Box

To clarify the question how prominent the influence of the spiky spectrum parts is under average conditions, figure 7.5 compares the original scene (first column, i.e. 7.5.a, d, g) with the same scene, but spikes dropped (second column, i.e. 7.5.b, e, h). The last column shows the color differences.

All times and cycle counts are averaged over 200 iterations. Color difference threshold in pretest: $\Delta E_{ab,thresh} = 5.0$.

Evaluation of the Composite Model In the *No Opt.* condition of the spiky-light scene, the Composite model (*ML-C*) scores better than equidistant sampling (*ML*), as chart 7.4.a suggests. But the charts hint at a weakness of the Composite model: If compiler optimizations and in particular vector operations come into play, the benefit disappears or *ML* is even faster because the Composite model implementation actually cannot make use of SIMD instructions.

Interpretation of the resulting images:

- The prominent spikes below $440nm$ desaturate the light. This is why the second image appears slightly more yellowish.
- For two reasons, the color difference reaches its maximum at the light source:
 - The light source is the brightest part of the image.
 - Spikes are not filtered by reflections. (In the ray tracing process, the light-source pixels are directly illuminated.)
- As expected, the red wall is relatively insensitive to missing spikes, because the highest density in redish spectra is above $600nm$, distant to the spikes below $440nm$. This finding can be seen as a hint for future full-spectral renderer implementations: If the Composite Model is implemented, spikes in ranges that are not covered by high densities in the scene’s reflection colors can actually be dropped without noticeable influence on the perceived image appearance.

<i>Level</i>	<i>Sample Count</i>	<i>Times Emit.</i>	<i>Times Refl.</i>
Level 0	80	2 if spikes present else 0	0
Level 1	40	2 if spikes dropped else 0	0
Level 2	20	0	32

Table 7.3: Spiky-Light Cornell Box: Level Decisions for *ML* (= equidistant sampling only).

<i>Level</i>	<i>Sample Count</i>	<i>Times Emit.</i>	<i>Times Refl.</i>
Level 0	40	2	0
Level 1	20	0	32

Table 7.4: Spiky-Light Cornell Box: Level Decisions for *ML-C* (= equidistant sampling + Composite Model).

	<i>No Opt.</i>	<i>Compil. Opt.</i>	<i>C+ Vec. Opt.</i>
<i>Ref. (seconds)</i>	0.139787	0.0285388	0.0284563
<i>ML (seconds)</i>	0.094341	0.0250236	0.0199028
<i>ML-C (seconds)</i>	0.0790766	0.0259461	0.0263739
<i>Ref. (rdtsc)</i>	2.63287e+08	5.00027e+07	5.00512e+07
<i>ML (rdtsc)</i>	1.5827e+08	5.80304e+07	3.65298e+07
<i>ML-C (rdtsc)</i>	1.461e+08	4.6761e+07	4.39784e+07

Table 7.5: Spiky-Light Cornell Box: Average single-iteration time in seconds spent processing color-related code.

	<i>No Opt.</i>	<i>Compil. Opt.</i>	<i>C+ Vec. Opt.</i>
<i>Ref. (seconds)</i>	0.154131	0.0301018	0.0288795
<i>ML (seconds)</i>	0.0572969	0.0182942	0.0185523
<i>ML-C (seconds)</i>	0.0602847	0.0189722	0.0189955
<i>Ref. (rdtsc)</i>	2.90461e+08	4.98838e+07	5.27637e+07
<i>ML (rdtsc)</i>	1.10911e+08	3.24872e+07	3.01975e+07
<i>ML-C (rdtsc)</i>	9.81445e+07	3.21536e+07	3.19167e+07

Table 7.6: Spiky-Light Cornell Box (spikes dropped): Average single-iteration time in seconds spent processing color-related code.

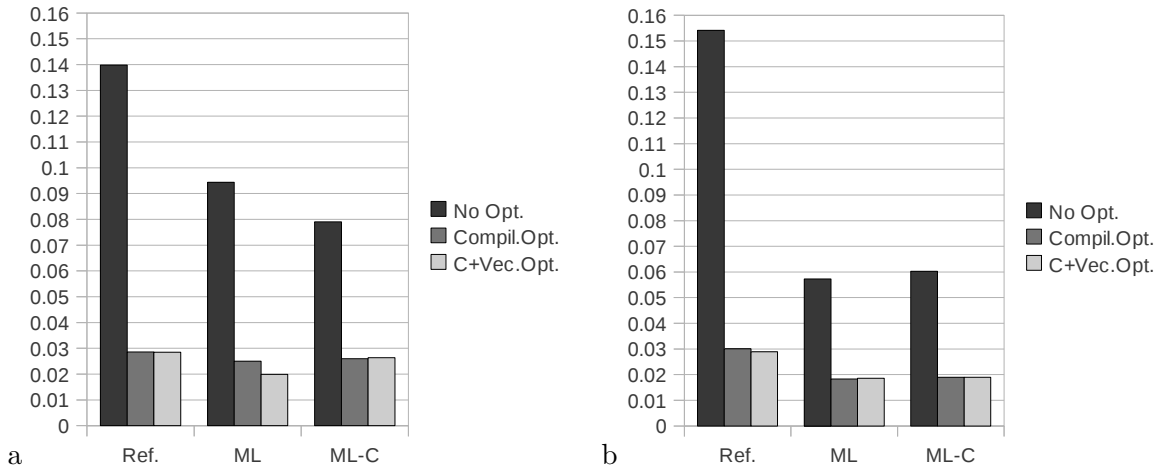


Figure 7.4: Spiky-Light Cornell Box: Charts of tables 7.5 (chart a) and 7.6 (chart b).

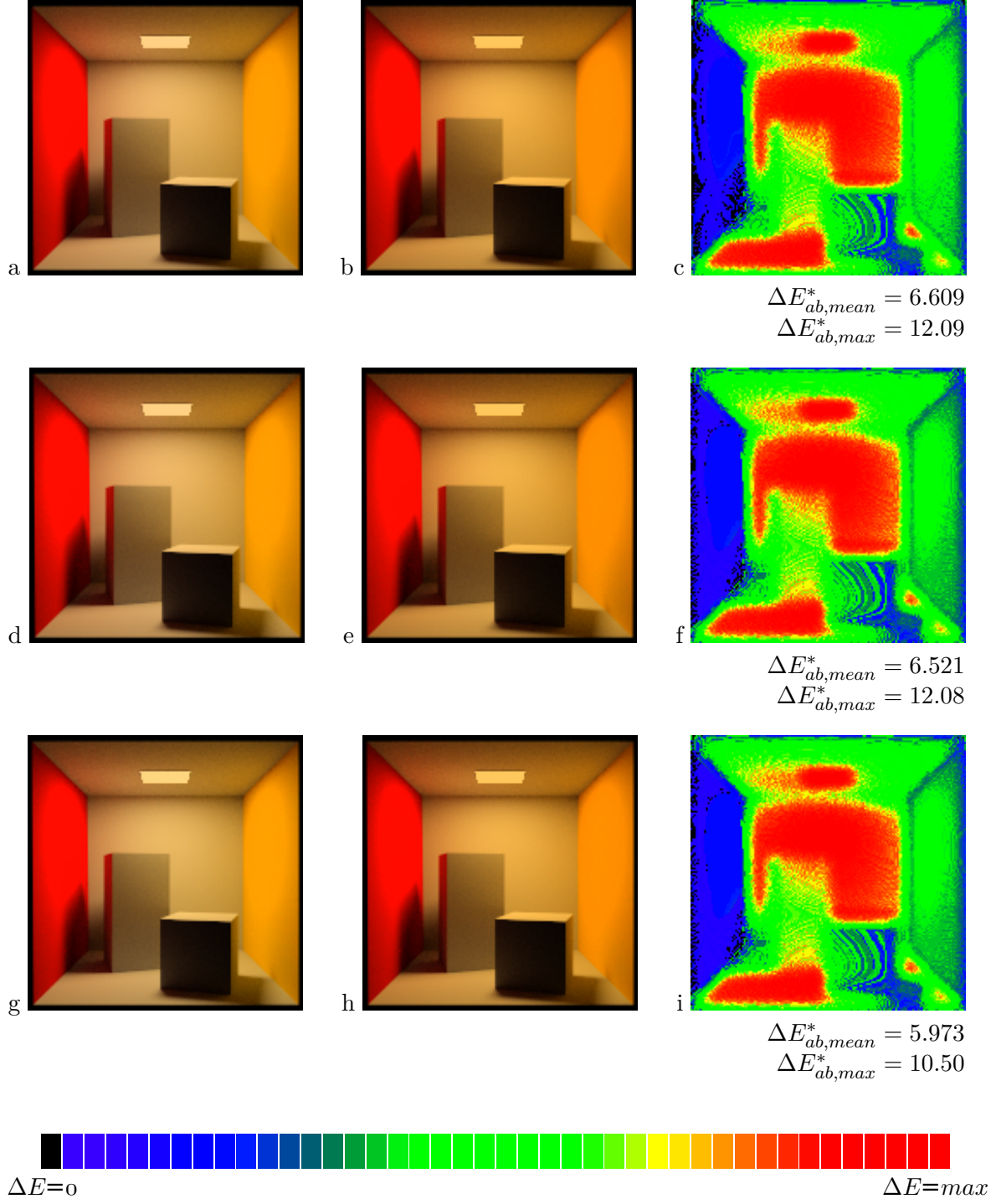


Figure 7.5: Spiky-Light Cornell Box. *Ref.* (a-c), *ML* (d-f), *ML-C* (g-i).

7.2.2 Colorful Scenes

The tests are focused on *visible errors* and *time efficiency* in more heterogeneous light and surface color environments. The resulting images are shown in figures 7.7 and 7.8. All times are stated in seconds, averaged over 200 iterations. Color difference threshold in pretest: $\Delta E_{ab,thresh} = 5.0$.

The following spectra are applied in the multi-light scene (figure 7.7):

- L_1 and L_2 are the light spectra.
- $R_1...R_8$ are the reflector spectra.

The following spectra are applied in the colorful Cornell box scene (figure 7.8):

- $L_3...L_{10}$ are the light spectra.
- R_3 is an equal spectrum and describes all reflectors in the scene.

The iteration time charts (figure 7.6) largely conform to the Standard Cornell Box time chart 7.1. In general, the Composite model appears to be slightly faster than plain equidistant sampling, but the *ML-C* case of the multi-light scene is ambivalent: The inferiority of the optimized *ML-C* cases might be due to the fact that rays with more than one accumulated spiky spectra need to access their patterns and height fields separately for each subsequent operation which induces time penalties. But also memory cache effects might be a cause, because the equidistant sample color representation implies less spread memory accesses than the retrieval of a spiky spectrum's smooth part, its spike pattern and its spike heights. For the multi-light scene, the light sources are all hybrid and their entire information is used randomly during the entire rendering process. However, the more realistic setting of the colorful Cornell box with two distinct light sources (one smooth, the other with spikes) and colors with sample counts at all levels casts a positive light on the Composite model approach.

Even though the color difference threshold is $\Delta E_{ab,thresh}^* = 5.0$, for the multi-light scene, the maximum color error among the pixel errors does not exceed 7.35 for *ML* and 9.18 for *ML-C* respectively, the mean errors are 0.15 and 0.59 respectively. The maximum color error of the colorful Cornell box scene is 9.09 for *ML* and 8.11 for *ML-C* respectively and the mean errors are 0.42 and 0.81. The spatial error distributions are shown in figures 7.7.c, 7.8.c, 7.8.f.

<i>Level</i>	<i>Sample Count</i>	<i>Times Emit.</i>	<i>Times Refl.</i>
Level 0	80	96	0
Level 1	40	0	0
Level 2	20	0	784

Table 7.7: Multi-Light Scene: Level Decisions for *ML* (= equidistant sampling only).

<i>Level</i>	<i>Sample Count</i>	<i>Times Emit.</i>	<i>Times Refl.</i>
Level 0	80	2	20
Level 1	40	0	6
Level 2	20	2	8

Table 7.8: Colorful Cornell Box: Level Decisions for *ML* (= equidistant sampling only).

	<i>No Opt.</i>	<i>Compil. Opt.</i>	<i>C+ Vec. Opt.</i>
<i>Ref. (seconds)</i>	0.0454618	0.0121505	0.0120491
<i>ML (seconds)</i>	0.024926	0.00770605	0.00770262
<i>ML-C (seconds)</i>	0.0226315	0.0084848	0.00978644
<i>Ref. (rdtsc)</i>	8.99419e+07	2.1872e+07	2.23589e+07
<i>ML (rdtsc)</i>	4.37697e+07	1.40397e+07	1.34597e+07
<i>ML-C (rdtsc)</i>	3.69719e+07	1.6986e+07	1.72476e+07

Table 7.9: Multi-Light Scene: Average single-iteration time in seconds spent processing color-related code.

	<i>No Opt.</i>	<i>Compil. Opt.</i>	<i>C+ Vec. Opt.</i>
<i>Ref. (seconds)</i>	0.117355	0.0245464	0.0235157
<i>ML (seconds)</i>	0.0710397	0.0187738	0.0178573
<i>ML-C (seconds)</i>	0.0548471	0.0175328	0.0149872
<i>Ref. (rdtsc)</i>	2.29202e+08	4.26293e+07	4.43836e+07
<i>ML (rdtsc)</i>	1.13546e+08	3.37784e+07	3.93586e+07
<i>ML-C (rdtsc)</i>	1.04791e+08	3.35788e+07	3.03262e+07

Table 7.10: Colorful Cornell Box: Average single-iteration time in seconds spent processing color-related code.

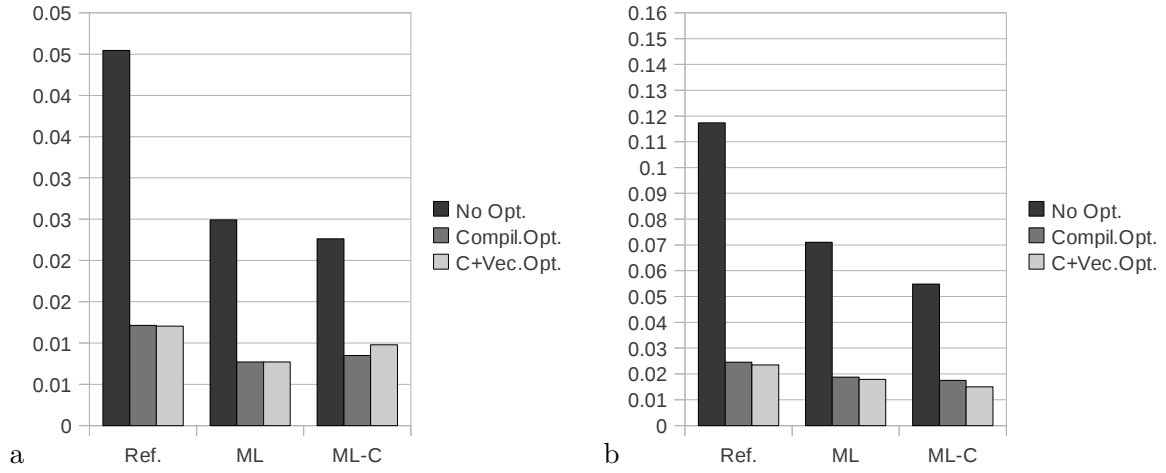


Figure 7.6: Colorful Scenes: Charts of tables 7.9 (chart a) and 7.10 (chart b).

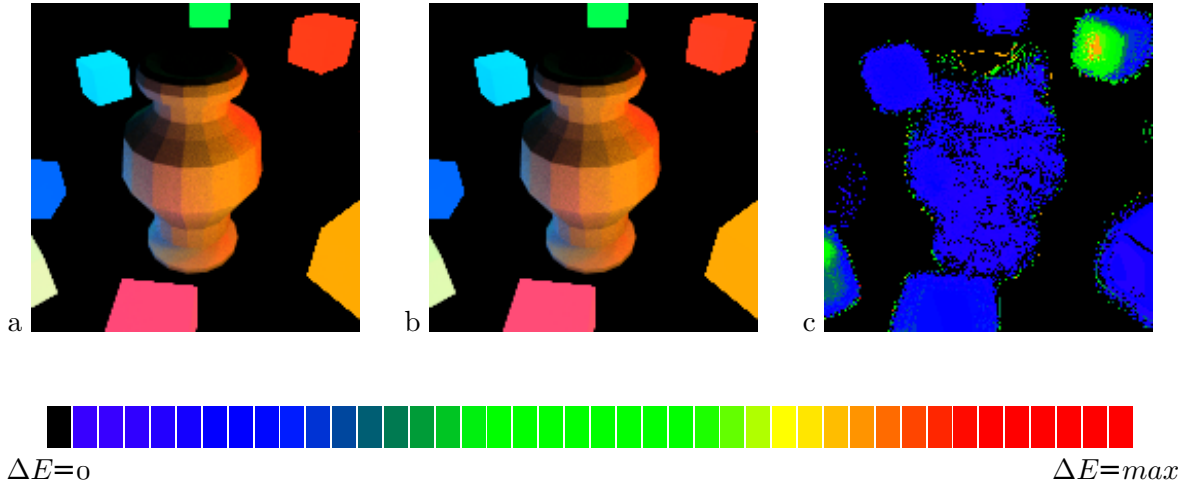


Figure 7.7: Multi-Light Scene. Reference image (a), $ML-C$ result (b), color differences ΔE (c). The horizontal bar shows the color coding of ΔE .

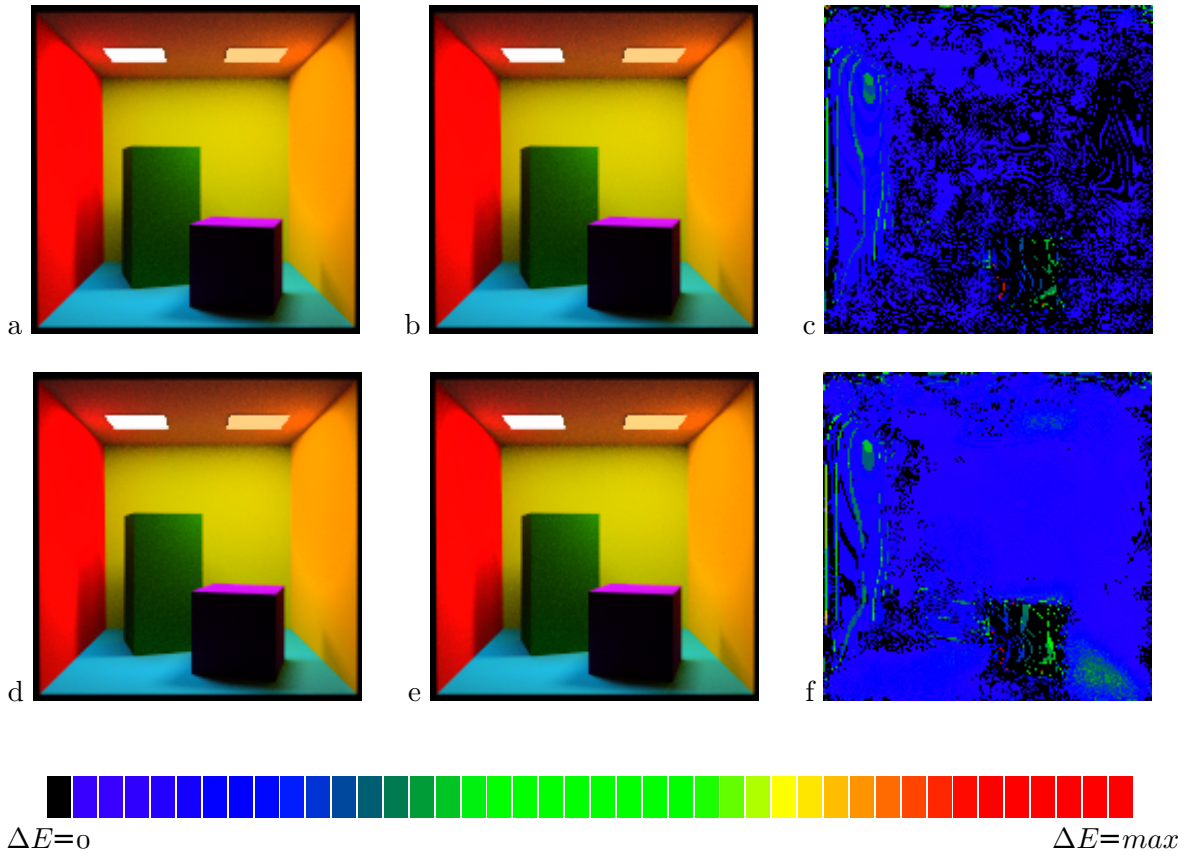


Figure 7.8: Colorful Cornell Box. Reference image (a, d), ML result (b), $ML-C$ result (e), color differences ΔE (c: a vs. b; f: d vs. e). The horizontal bar shows the color coding of ΔE .

7.2.3 Color Difference Thresholds

This test is focused on *color difference* versus *speed*. Resulting images are shown in figure 7.10. All times are stated in seconds, averaged over 200 iterations. Color difference threshold in pretest: $\Delta E_{ab,thresh} = 5.0$.

Level decision distributions are tabulated in table 7.11.

The plot of relative developments (figure 7.9.b) shows that in all optimization conditions, the color difference considerably stronger rises than the computation times drops (the *No Opt.* condition considerably fluctuates, presumably due to measuring inaccuracy, but the overall tendency is a slight decrease). In other words: A slightly higher computation time allows for significantly more accurate results. Anyhow, even in the $\Delta E_{ab,thresh}^* = 7.0$ case (figure 7.10.e, f, g), actually no difference is visually identifiable between the reference image and the test case image. Thus, if accuracy is not of overall importance, the performance gain will most likely outweigh the accuracy loss, which argues for a mipmapping implementation.

ΔE Thresh.	Level	Sample Count	Times Emit.	Times Refl.
3.0	Level 0	80	2	22
	Level 1	40	0	8
	Level 2	20	2	4
4.0	Level 0	80	2	22
	Level 1	40	0	4
	Level 2	20	2	8
5.0	Level 0	80	2	20
	Level 1	40	0	6
	Level 2	20	2	8
6.0	Level 0	80	2	20
	Level 1	40	0	4
	Level 2	20	2	10
7.0	Level 0	80	2	10
	Level 1	40	0	12
	Level 2	20	2	12

Table 7.11: Comparison of Color Difference Thresholds: Development of Level Decisions for *ML* (equidistant sampling only).

$\Delta E_{ab,thresh}^*$	$\Delta E_{ab,mean}^*$	Number of Pixels with $\Delta E_{ab}^* \dots$					
		< 1.0	< 2.0	< 3.0	< 4.0	< 5.0	≥ 5.0
3.0	0.236	24282	865	328	91	23	0, 1, 11
4.0	0.379	22859	1862	493	301	31	43, 1, 10
5.0	0.384	22844	1866	496	307	31	45, 1, 10
6.0	0.396	22718	1969	494	325	34	48, 2, 10
7.0	0.443	22455	2153	548	342	37	50, 2, 8, 5

Table 7.12: Comparison of Color Difference Thresholds: Color Errors in the final image.

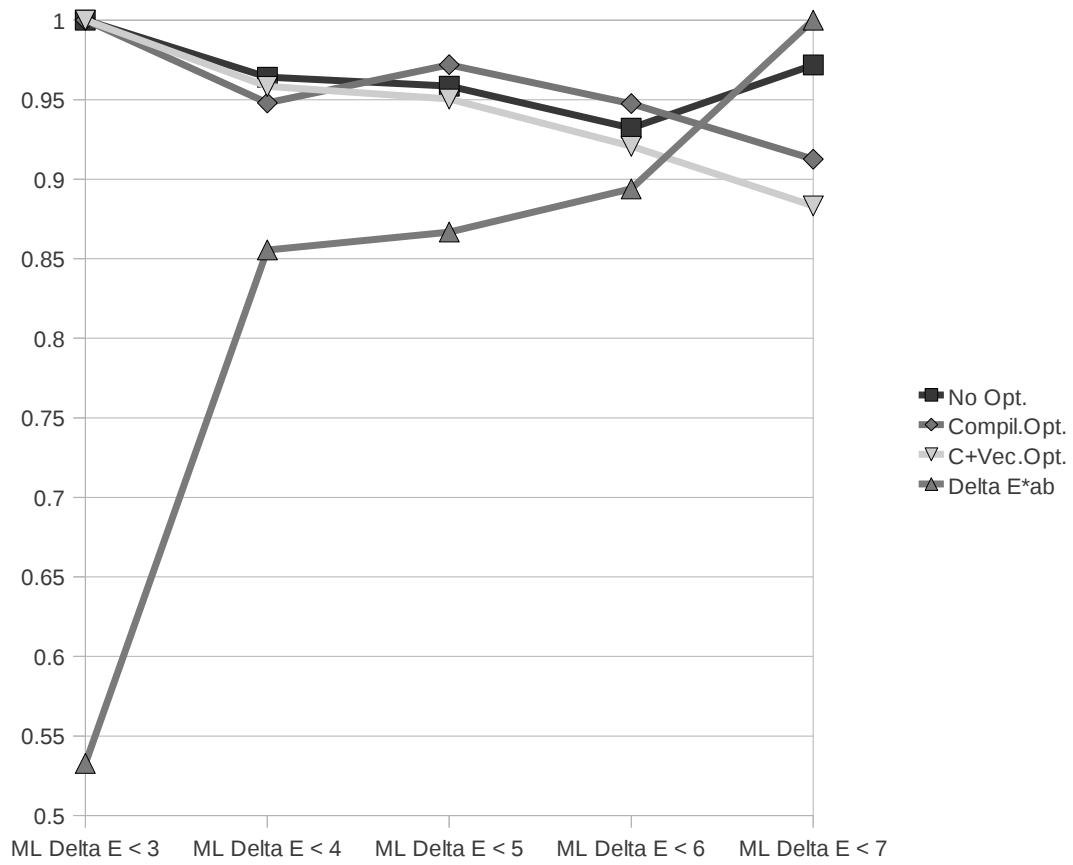


Figure 7.9: Comparison of Color Difference Thresholds: Development of processing times and resulting mean color differences $\Delta E_{ab,mean}^*$ against rising color difference threshold (abscissa). The processing times are plotted relative to the computation time at $\Delta E_{ab,thresh}^* = 3.0$ (*No Opt.*: 72millisec; *Compil.Opt.*: 21millisec; *C+Vec.Opt.*: 21millisec) and color differences are plotted relative to $\Delta E_{ab,mean}^* = 0.443$ (table 7.12).

Color differences of figure 7.10 on the next page:

Figure	ΔE_{ab}^*	$\Delta E_{ab,max}^*$	$\Delta E_{ab,mean}^*$
7.10.a	3.0	0.236	7.977
7.10.b	4.0	0.379	7.977
7.10.c	5.0	0.384	7.977
7.10.d	6.0	0.396	7.977
7.10.e	7.0	0.443	8.106

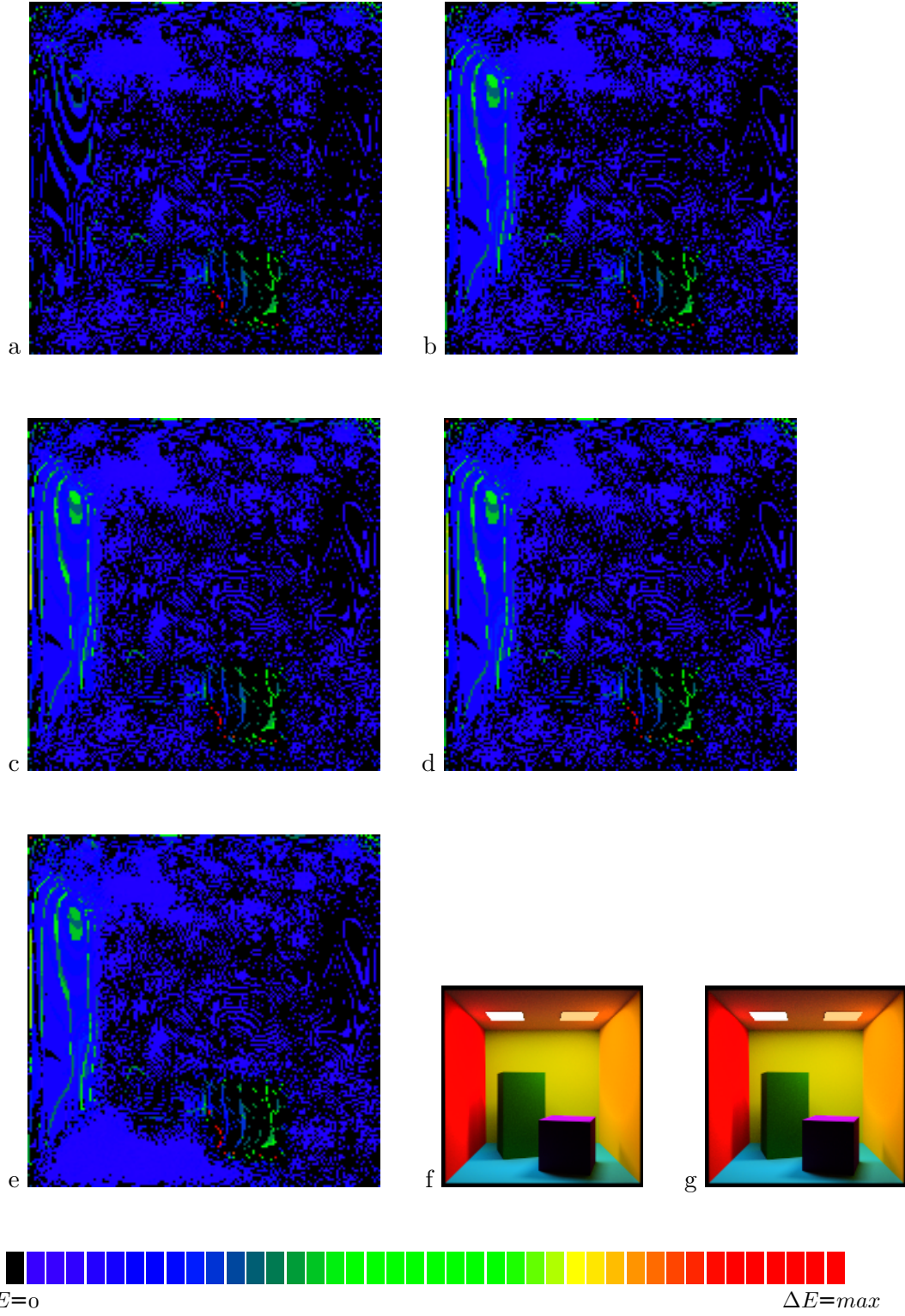


Figure 7.10: (a–e) pixel color errors for pretest color difference thresholds of $\Delta E_{ab,thresh}^* = 3.0\text{--}7.0$, (f) *Ref.* and (g) *ML* are the results with $\Delta E_{ab,thresh}^* = 7.0$.

7.2.4 Operator Times

The following time-critical operations/queries (“fast operation”, identifier *fop*; “fast query”, identifier *fq*) are part of the full-spectrum color class in the adapted Minilight ray tracer:

fop-ipl:mulsource-c Multiplication of the spectrum by a reflector or light source spectrum, inclusive of spikes (in-place operation).

fop-ipl:addsource-cs Addition of a scaled reflector or light source spectrum, inclusive of spikes (in-place operation).

fop-ipl:addsingle-c Addition of an arbitrary single-level spectrum, inclusive of spikes (in-place operation).

fop-ipl:addsingle-cs Addition of an arbitrary scaled single-level spectrum, inclusive of spikes (in-place operation).

fop-ipl:clearsingle Clear single level and spikes. Used for global intermediate variables.

fop-ipl:setsource-c Set the spectrum to a reflector or light source spectrum.

fop-ipl:setsource-cs Set the spectrum to a scaled reflector or light source spectrum.

The test scene is the spiky-light Cornell box of section 7.2.1. All times are accumulated during 200 ray tracing iterations and stated in seconds.

Multiplication

(Table 7.13)

The use of vector operations drastically reduces the difference between the time costs of the reference calculation *Ref.* and the test cases *ML* and *ML-C*. However, they are still somewhat faster. The spike part is pretty costly, because of the separate handling. This points out a weakness of the Composite model.

fop-ipl:mulsource-c					
		<i>Call Count (Compil.Opt.)</i>	<i>No Opt.</i>	<i>Compil.Opt.</i>	<i>C+Vec.Opt.</i>
<i>Ref.</i>		8145468	6.826	0.931	0.655
<i>ML</i>	Level 0	6653696	4.359	0.804	0.626
	Level 1	0	0	0	0
	Level 2	428187	0.061	0.015	0.014
		7081883	4.421	0.820	0.640
<i>ML-C</i>	Level 0	6679898	2.071	0.408	0.508
	Level 1	431476	0.067	0.019	0.017
		7111374	2.138	0.426	0.525
	Spikes		2.461	1.252	0.878

Table 7.13: Spectrum-by-Spectrum Multiplication.

Addition, Clear and Set

(Tables 7.14 and 7.15)

Except for *fop-ipl:addsingle-cs*, the ratio between reference and test cases is similar to that of the multiplication operation. The third table (addition of scaled spectrum) suggests that the scaling of spikes using the Composite model is a costly task.

fop-ipl:addsource-cs					
		<i>Call Count (Compil.Opt.)</i>	<i>No Opt.</i>	<i>Compil.Opt.</i>	<i>C+Vec.Opt.</i>
<i>Ref.</i>		3863954	3.549	0.481	0.489
<i>ML</i>	Level 0	29327	0.020	0.004	0.003
	Level 1	0	0	0	0
	Level 2	0	0	0	0
		29327	0.020	0.004	0.003
<i>ML-C</i>	Level 0	29298	0.011	0.002	0.002
	Level 1	0	0	0	0
		29298	0.011	0.002	0.002
	Spikes		0.004	0.001	0.001

fop-ipl:addsingle-c					
		<i>Call Count (Compil.Opt.)</i>	<i>No Opt.</i>	<i>Compil.Opt.</i>	<i>C+Vec.Opt.</i>
<i>Ref.</i>		5120000	5.077	1.525	1.453
<i>ML</i>	Level 0	3385668	2.444	0.579	0.540
	Level 1	0	0	0	0
	Level 2	1734332	0.368	0.146	0.139
		5120000	2.812	0.724	0.679
<i>ML-C</i>	Level 0	3386530	1.258	0.343	0.321
	Level 1	1733470	0.389	0.135	0.138
		5120000	1.647	0.478	0.459
	Spikes		0.620	0.145	0.222

fop-ipl:addsingle-cs					
		<i>Call Count (Compil.Opt.)</i>	<i>No Opt.</i>	<i>Compil.Opt.</i>	<i>C+Vec.Opt.</i>
<i>Ref.</i>		4629492	4.219	0.559	0.535
<i>ML</i>	Level 0	4290740	3.191	0.896	0.609
	Level 1	0	0	0	0
	Level 2	0	0	0	0
		4290740	3.191	0.896	0.609
<i>ML-C</i>	Level 0	4299832	1.772	0.577	0.574
	Level 1	0	0	0	0
		4299832	1.772	0.577	0.574
	Spikes		0.731	0.260	0.334

Table 7.14: Spectrum-by-Spectrum Additions.

fop-ipl:clearsingle					
		<i>Call Count (Compil.Opt.)</i>	<i>No Opt.</i>	<i>Compil.Opt.</i>	<i>C+Vec.Opt.</i>
<i>Ref.</i>		3900524	1.561	0.276	0.250
<i>ML</i>	Level 0	2982665	0.822	0.173	0.160
	Level 1	0	0	0	0
	Level 2	1067068	0.168	0.055	0.036
		4049733	0.989	0.228	0.196
<i>ML-C</i>	Level 0	2980720	0.451	0.128	0.089
	Level 1	1063832	0.162	0.043	0.031
		4044552	0.613	0.171	0.120
	Spikes		0.118	0.028	0.011

fop-ipl:setsource-c					
		<i>Call Count (Compil.Opt.)</i>	<i>No Opt.</i>	<i>Compil.Opt.</i>	<i>C+Vec.Opt.</i>
<i>Ref.</i>		1219476	0.804	0.178	0.172
<i>ML</i>	Level 0	403003	0.342	0.119	0.130
	Level 1	0	0	0	0
	Level 2	667264	0.171	0.088	0.076
		1070267	0.514	0.207	0.205
<i>ML-C</i>	Level 0	405810	0.216	0.100	0.093
	Level 1	669638	0.165	0.085	0.069
		1075448	0.382	0.185	0.162
	Spikes		0.038	0.011	0.016

fop-ipl:setsource-cs					
		<i>Call Count (Compil.Opt.)</i>	<i>No Opt.</i>	<i>Compil.Opt.</i>	<i>C+Vec.Opt.</i>
<i>Ref.</i>		8145468	7.315	1.173	1.224
<i>ML</i>	Level 0	6653696	3.821	0.941	0.659
	Level 1	0	0	0	0
	Level 2	428187	0.063	0.033	0.024
		7081883	3.884	0.974	0.683
<i>ML-C</i>	Level 0	6679898	2.124	0.637	0.629
	Level 1	431476	0.085	0.028	0.024
		7111374	2.209	0.665	0.652
	Spikes		0.816	0.273	0.279

Table 7.15: Clear and Set Operations.

Chapter 8

Conclusion

Full-spectral global illumination renderer implementations require a different design strategy than implementations of three-component based renderers. A properly designed renderer that is programmed using advanced performance programming techniques with the objective of efficient machine code generation allows for fast operations on sampled spectra. The performance gain induced by the support of aggressive compiler optimizations, can be 75% or more. Further considerable performance gains can be achieved by the use of *Spectral Mipmapping*, but they are only noticeable if function calls are eliminated by manual inlining. The effectiveness of Spectral Mipmapping always depends on a certain heterogeneity of spectra: Few spectra with high sample counts and a huge amount of spectra with low sample counts best exhibit the potential of Spectral Mipmapping. The Spectral Mipmapping approach allows for a fine-tuning of sample counts. This is especially useful for scenes possessing multiple light sources or surface colors with differently shaped spectra with regard to the required number of samples. Depending on the hardware characteristics, the reduction of samples may lead to a considerable performance gain, particularly if few of a scene's spectra require 80 samples or more, such as spiky or other turbulent types of spectra.

Under certain conditions, the Composite model by Sun [16] is more efficient than plain equidistant sampling, even if 4-parallel SIMD instructions are used which are not necessarily beneficial for a Composite model implementation. However, one spike already necessitates eight scalar operations if multiplied by a smooth spectrum (please refer to section 6.2.2) and the spectrum of a common fluorescence lamp possesses four major spikes, not to mention their foothills, at least 32 not necessarily parallelizable scalar operations must be performed, merely to handle the spikes without the smooth part which requires further scalar operations. If, in a ray tracer, a recursive ray collects more than one spike pattern with four or more spikes, belonging to different light sources, 64 or more scalar operations are required solely to handle the spikes.

Particularly with regard to the growing parallel processing capabilities of modern computer systems, concerning both CPUs and Graphical Processing Units, which can handle large sample vectors entirely parallel for applying a common scalar operation on their components, plain sample representations of spectral colors are advantageous. But even if massive parallelization is possible, Spectral Mipmapping can help using the resources more efficiently.

The weak point of the Spectral Mipmapping multi-level approach is the pretest phase. The color difference threshold can only roughly define the effective maximum pixel error in the final image if the pretests do not cover all possible light-surface interactions that may occur during the rendering process, which is roughly as time-consuming as the entire rendering process and therefore not an option. However, forecasts can be made based on likely color difference developments (please refer to section 4.1.3 for details) and a color difference (ΔE_{ab}^*) threshold of 5.0 seems to be a good choice. In order to decide for a threshold that is appropriate for a certain scene, a preview can be rendered at a low resolution and few iterations and compared to a reference preview rendered from the same scene in blurred form (blurring eliminates the noise).

The number of vector length levels for a Spectral Mipmapping implementation should be chosen wisely: Although many levels provide more choices to appropriately classify the individual spectra, if the machine code is aggressively optimized for performance, each individual level gets its own machine code procedures. Since the level chosen for processing varies from pixel iteration to pixel iteration, the procedures of all levels need to be available and should therefore reside in the instruction cache. Otherwise, cache faults may induce considerable performance penalties. In addition, the levels are restricted by the set of vector operation instructions. If SIMD instructions are used, which handle four scalars per operand in parallel, the level pool can be restricted to vector lengths of multiples of four. Three levels, for instance 80, 40 and 20 samples, are a suitable choice.

Methods based on basis functions depend on the shape and number of spectra that occur in a scene since each spectrum contributes to the shape of the basis functions. The information loss stemming from the basis function representation therefore depends on the individual scene and is hard to estimate. Besides, light-surface interactions are matrix multiplications and thus, akin to the Composite model, less suitable for SIMD vector operations.

Future Work

The peculiarities of different processor types should be investigated in order to recommend or advise against a multi-level approach. Fully developed global illumination renderers that are based on full-spectral rendering, such as LuxRender [10], could be equipped with a multi-level approach as a method of choice. A guide for new renderer projects and modeling tools may help the developer team make the renderer's design full-spectrum ready and appropriately optimized. The required effort is marginal but considerably upvalues the final product.

Particularly the automated accuracy choice for Spectral Mipmapping needs improvement, possibly based on experience with sampling rates for known spectra and a classification algorithm for newly measured or synthesized spectra that classifies spectra by shape comparison.

Appendix A

Color and Color Difference

A.1 Tonemapping

Since a full-spectral renderer inherently operates on device-independent colors in the form of spectra, the result is to be tonemapped in order to ensure a realistic appearance on the output device (i.e. computer monitor, printer,...). In the test scenes, the surface luminance highly varies between emitting surfaces and non-emitting surfaces that are illuminated.

Tonemapping requirements:

- Preserve chromacity appearance.
- Avoid clamping.

Tonemapping (RGB color space):

$$\begin{pmatrix} r_o \\ g_o \\ b_o \end{pmatrix} = factor \cdot \begin{pmatrix} r \\ g \\ b \end{pmatrix} \quad (\text{A.1})$$

The values r_o , g_o and b_o are the RGB colors sent to the output device.

The following formula satisfies the requirements mentioned above with minimal effort:

$$\begin{aligned} factor &= \frac{1}{a + luminance} \\ luminance &= 0.27r + 0.67g + 0.06b \\ 0.27 + 0.67 + 0.06 &= 1 \end{aligned} \quad (\text{A.2})$$

The tone mapping formula is based on formula (3) discussed by [33].

For all rendered images, a was set to 5.

A.2 Color-Matching Functions

The Color-Matching Functions were used to perform spectrum-to-XYZ color space conversions for color difference tests and rendering. The values are copied from [22].

Wavelength λ	$\bar{x}(\lambda)$	$\bar{y}(\lambda)$	$\bar{z}(\lambda)$				
380nm	0.0014	0.0000	0.0065	580nm	0.9163	0.8700	0.0017
385nm	0.0022	0.0001	0.0105	585nm	0.9786	0.8163	0.0014
390nm	0.0042	0.0001	0.0201	590nm	1.0263	0.7570	0.0011
395nm	0.0076	0.0002	0.0362	595nm	1.0567	0.6949	0.0010
400nm	0.0143	0.0004	0.0679	600nm	1.0622	0.6310	0.0008
405nm	0.0232	0.0006	0.1102	605nm	1.0456	0.5668	0.0006
410nm	0.0435	0.0012	0.2074	610nm	1.0026	0.5030	0.0003
415nm	0.0776	0.0022	0.3713	615nm	0.9384	0.4412	0.0002
420nm	0.1344	0.0040	0.6456	620nm	0.8544	0.3810	0.0002
425nm	0.2148	0.0073	1.0391	625nm	0.7514	0.3210	0.0001
430nm	0.2839	0.0116	1.3856	630nm	0.6424	0.2650	0.0000
435nm	0.3285	0.0168	1.6230	635nm	0.5419	0.2170	0.0000
440nm	0.3483	0.0230	1.7471	640nm	0.4479	0.1750	0.0000
445nm	0.3481	0.0298	1.7826	645nm	0.3608	0.1382	0.0000
450nm	0.3362	0.0380	1.7721	650nm	0.2835	0.1070	0.0000
455nm	0.3187	0.0480	1.7441	655nm	0.2187	0.0816	0.0000
460nm	0.2908	0.0600	1.6692	660nm	0.1649	0.0610	0.0000
465nm	0.2511	0.0739	1.5281	665nm	0.1212	0.0446	0.0000
470nm	0.1954	0.0910	1.2876	670nm	0.0874	0.0320	0.0000
475nm	0.1421	0.1126	1.0419	675nm	0.0636	0.0232	0.0000
480nm	0.0956	0.1390	0.8130	680nm	0.0468	0.0170	0.0000
485nm	0.0580	0.1693	0.6162	685nm	0.0329	0.0119	0.0000
490nm	0.0320	0.2080	0.4652	690nm	0.0227	0.0082	0.0000
495nm	0.0147	0.2586	0.3533	695nm	0.0158	0.0057	0.0000
500nm	0.0049	0.3230	0.2720	700nm	0.0114	0.0041	0.0000
505nm	0.0024	0.4073	0.2123	705nm	0.0081	0.0029	0.0000
510nm	0.0093	0.5030	0.1582	710nm	0.0058	0.0021	0.0000
515nm	0.0291	0.6082	0.1117	715nm	0.0041	0.0015	0.0000
520nm	0.0633	0.7100	0.0782	720nm	0.0029	0.0010	0.0000
525nm	0.1096	0.7932	0.0573	725nm	0.0020	0.0007	0.0000
530nm	0.1655	0.8620	0.0422	730nm	0.0014	0.0005	0.0000
535nm	0.2257	0.9149	0.0298	735nm	0.0010	0.0004	0.0000
540nm	0.2904	0.9540	0.0203	740nm	0.0007	0.0003	0.0000
545nm	0.3597	0.9803	0.0134	745nm	0.0005	0.0002	0.0000
550nm	0.4334	0.9950	0.0087	750nm	0.0003	0.0001	0.0000
555nm	0.5121	1.0002	0.0057	755nm	0.0002	0.0001	0.0000
560nm	0.5945	0.9950	0.0039	760nm	0.0002	0.0001	0.0000
565nm	0.6784	0.9786	0.0027	765nm	0.0001	0.0000	0.0000
570nm	0.7621	0.9520	0.0021	770nm	0.0001	0.0000	0.0000
575nm	0.8425	0.9154	0.0018	775nm	0.0000	0.0000	0.0000
				780nm	0.0000	0.0000	0.0000

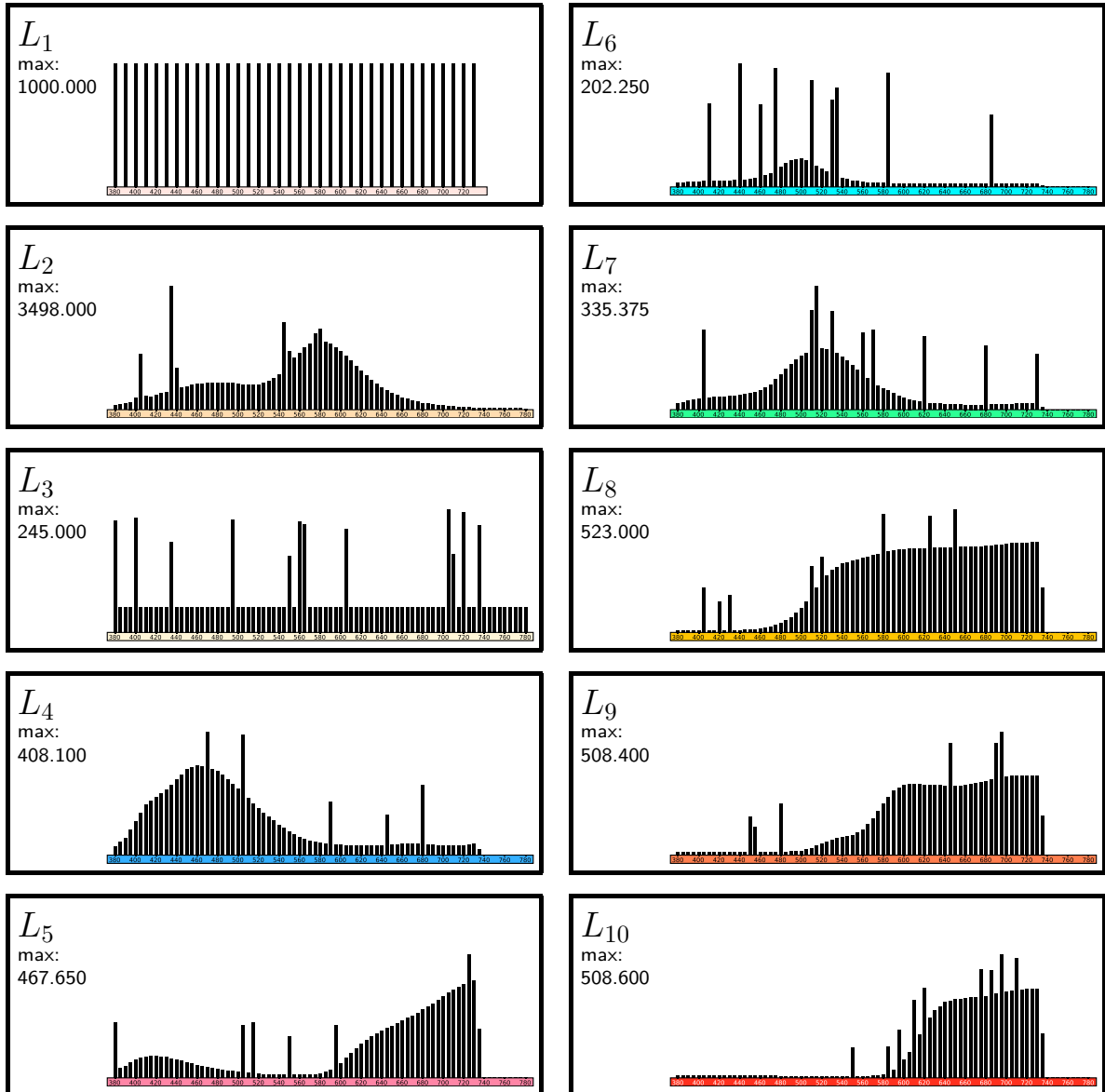
Table A.1: CIE 1931 Standard Colorimetric System: Color-Matching Functions.

A.3 Spectra

The first sample of each spectrum is centered at $380nm$.

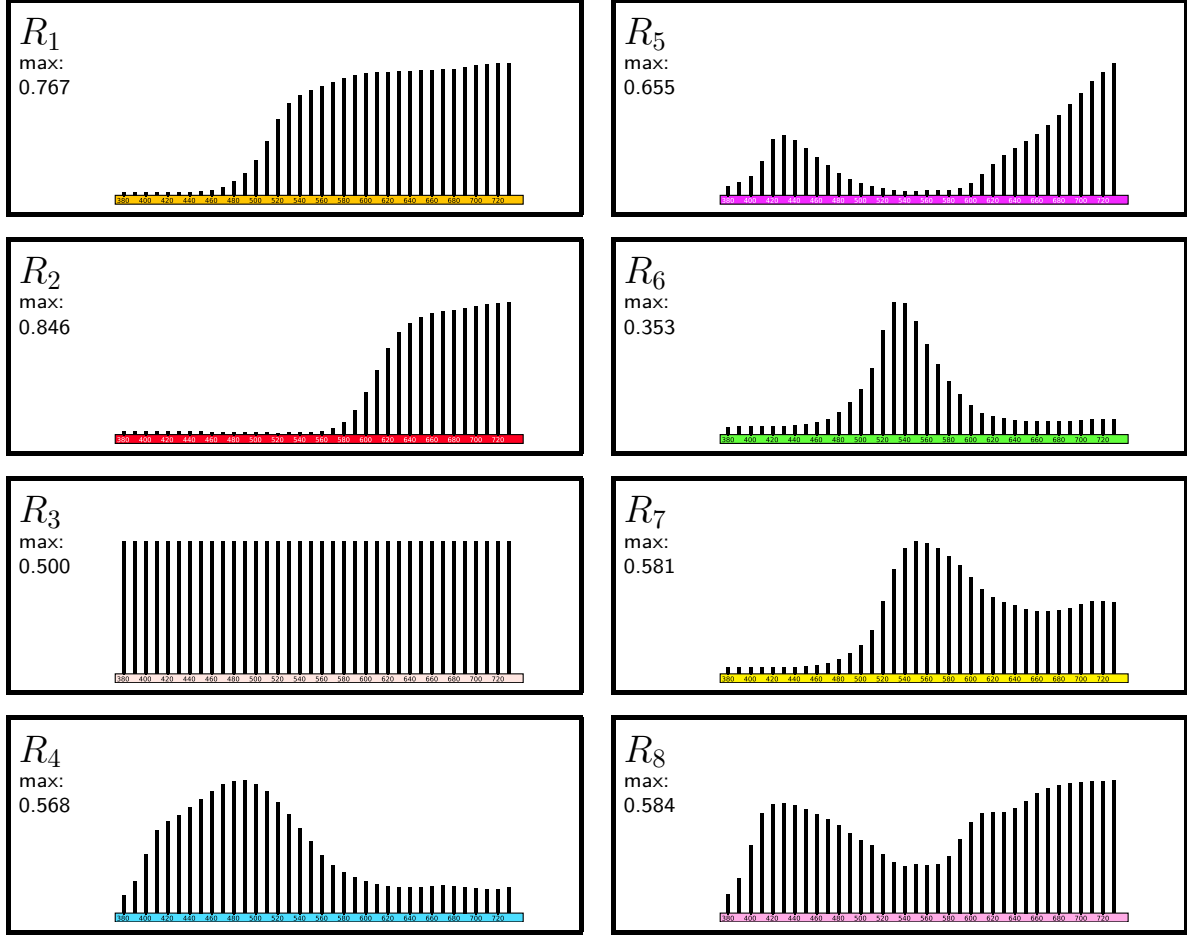
A.3.1 Light Source Spectra

L_1 is an artificial equal spectrum and L_2 is the spectrum of a fluorescent lamp. $L_2...L_{10}$ are resampled reflector spectra measured from the *Macbeth Colour Checker* (see [30]) and resampled at 81 samples. The spikes of $L_2...L_{10}$ are random generated. Except for L_1 , all light source spectra are sampled at $5nm$ intervals.



A.3.2 Reflector Spectra

The reflector spectra are sampled from the *Macbeth Colour Checker* and sampled at 36 equidistant locations at intervals of $10nm$.



L*a*b*	97.8 9.7 86.9 97.9 8.9 87.3	96.3 15.8 130.0 96.5 14.9 129.9	95.3 20.4 148.0 95.3 20.3 146.9	94.5 23.6 153.2 94.3 24.6 151.5	94.0 26.0 154.2 93.5 28.1 152.5
ΔE	0.873	0.992	1.193	1.943	2.745
L*a*b*	98.1 8.4 5.3 97.9 9.0 5.1	98.0 8.4 5.0 97.9 9.1 4.7	98.0 8.5 4.5 97.8 9.3 4.2	98.0 8.7 4.1 97.8 9.5 3.8	98.0 8.8 3.6 97.8 9.7 3.3
ΔE	0.700	0.770	0.820	0.862	0.902
L*a*b*	94.0 25.7 29.1 93.9 26.3 28.8	89.0 47.5 49.3 88.9 48.0 49.2	83.8 69.6 66.4 83.7 70.2 66.6	79.3 89.4 80.9 79.2 89.7 81.4	75.6 105.2 93.1 75.6 105.2 93.8
ΔE	0.621	0.596	0.595	0.599	0.693
L*a*b*	100.0 -5.8 36.4 100.0 -5.0 36.2	100.0 -15.9 61.6 100.0 -15.1 61.4	100.0 -23.6 81.8 100.0 -22.5 81.5	100.0 -29.6 97.7 100.0 -28.4 97.4	100.0 -34.6 109.9 100.0 -33.3 109.6
ΔE	0.758	0.935	1.102	1.228	1.291
L*a*b*	98.2 7.9 5.7 98.0 8.6 5.4	98.2 7.6 5.9 98.1 8.4 5.5	98.3 7.3 6.0 98.1 8.2 5.5	98.4 7.1 6.1 98.1 8.0 5.6	98.4 6.9 6.2 98.2 7.9 5.6
ΔE	0.739	0.875	1.000	1.114	1.217
L*a*b*	98.1 8.1 4.4 98.0 8.7 4.1	98.2 7.9 3.2 98.0 8.6 2.8	98.2 7.8 2.0 98.0 8.6 1.5	51.0 4.5 0.5 50.9 5.1 0.1	11.0 1.8 -0.1 10.9 2.1 -0.3
ΔE	0.718	0.842	0.967	0.635	0.284
L*a*b*	100.0 -15.3 86.7 100.0 -16.2 86.9	100.0 -29.5 131.4 100.0 -30.4 130.9	100.0 -40.5 150.2 100.0 -40.0 148.5	100.0 -49.3 155.8 100.0 -46.7 153.6	100.0 -56.3 157.4 100.0 -51.5 155.6
ΔE	0.893	1.045	1.780	3.337	5.126
L*a*b*	89.4 45.6 52.4 89.3 46.0 52.4	82.4 75.8 87.0 82.3 76.1 87.1	78.0 94.8 110.4 78.0 95.0 110.5	75.4 106.0 123.4 75.4 106.2 123.3	35.7 65.0 61.3 35.9 65.4 61.5
ΔE	0.456	0.351	0.283	0.269	0.487
L*a*b*	98.2 7.9 5.0 98.0 8.6 4.7	98.2 7.6 4.4 98.1 8.4 4.0	98.3 7.3 3.7 98.1 8.2 3.2	98.4 7.1 3.0 98.1 8.1 2.5	51.9 4.1 1.4 51.8 4.7 1.0
ΔE	0.744	0.888	1.026	1.153	0.748
L*a*b*	98.1 8.4 2.6 97.9 9.2 2.3	80.1 7.2 -0.3 80.1 8.2 -0.7	3.8 0.9 -0.3 3.7 1.0 -0.4	0.0 0.0 -0.0 0.0 0.0 -0.0	0.0 0.0 -0.0 0.0 0.0 -0.0
ΔE	0.948	1.086	0.167	0.002	0.000
L*a*b*	97.3 11.7 115.5 97.3 11.6 115.4	94.8 22.5 146.2 94.7 22.8 145.7	93.3 29.0 152.0 93.0 30.2 151.2	92.2 33.4 152.6 91.8 35.2 151.8	91.5 36.7 152.4 91.0 38.8 151.7
ΔE	0.110	0.572	1.458	2.043	2.300
L*a*b*	98.2 7.9 5.7 98.0 8.6 5.4	98.2 7.6 5.9 98.1 8.3 5.5	98.3 7.3 6.0 98.1 8.1 5.5	98.4 7.0 6.1 98.2 8.0 5.6	98.4 6.9 6.2 98.2 7.9 5.6
ΔE	0.728	0.853	0.968	1.073	1.167
L*a*b*	100.0 -87.1 33.3 100.0 -85.8 32.9	100.0 -147.1 57.7 100.0 -145.8 57.9	100.0 -180.5 72.6 100.0 -179.1 73.3	100.0 -199.9 80.9 100.0 -197.4 81.6	100.0 -212.1 85.6 100.0 -207.5 86.2
ΔE	1.361	1.308	1.542	2.582	4.685
L*a*b*	98.0 -6.7 -3.5 97.8 -6.0 -3.8	93.5 -20.7 -11.2 93.3 -19.7 -11.5	90.0 -33.0 -17.3 89.8 -31.9 -17.6	87.2 -43.9 -22.1 87.0 -42.6 -22.5	85.0 -53.2 -25.8 84.8 -51.9 -26.3
ΔE	0.807	0.978	1.139	1.293	1.443
L*a*b*	98.0 8.5 5.5 97.9 9.2 5.3	98.0 8.8 5.3 97.8 9.5 5.1	97.9 9.1 5.1 97.7 9.8 4.9	97.8 9.5 4.8 97.6 10.2 4.6	97.7 9.8 4.5 97.5 10.6 4.3
ΔE	0.676	0.716	0.739	0.759	0.784
L*a*b*	94.9 22.0 12.9 94.7 22.8 12.6	91.8 35.3 20.6 91.6 36.2 20.2	88.9 47.7 28.5 88.7 48.6 28.2	86.3 59.0 36.6 86.1 59.8 36.4	84.0 68.9 44.7 83.8 69.6 44.7
ΔE	0.873	1.013	1.020	0.909	0.710
L*a*b*	97.3 11.7 115.5 97.3 11.6 115.4	94.8 22.5 146.2 94.7 22.8 145.7	93.3 29.0 152.0 93.0 30.2 151.2	92.2 33.4 152.6 91.8 35.2 151.8	91.5 36.7 152.4 91.0 38.8 151.7
ΔE	0.110	0.572	1.458	2.043	2.300
L*a*b*	98.3 7.2 6.7 98.2 7.8 6.5	98.6 6.2 7.4 98.4 6.9 7.3	98.8 5.3 8.1 98.6 6.0 7.9	99.0 4.4 8.6 98.8 5.3 8.4	99.2 3.6 9.0 98.9 4.6 8.7
ΔE	0.638	0.692	0.781	0.908	1.071
L*a*b*	75.1 107.5 77.2 75.0 107.6 77.5	70.4 127.6 117.6 70.4 127.7 117.6	69.0 133.6 118.7 69.1 133.3 118.8	68.3 136.7 117.6 68.4 136.0 117.9	67.8 138.7 116.9 68.0 137.9 117.1
ΔE	0.329	0.096	0.345	0.790	0.883
L*a*b*	92.8 30.9 101.2 92.9 30.8 101.1	88.4 49.8 138.6 88.2 50.7 137.9	85.6 61.9 143.8 85.2 63.7 142.9	83.7 70.2 142.1 83.2 72.5 141.2	82.3 76.2 140.3 81.7 78.8 139.3
ΔE	0.135	1.151	2.041	2.569	2.814

Table A.2: Test 1 results (see section 4.1.3).

L*a*b*	98.1 8.4 5.3 97.9 9.0 5.1	94.0 25.8 28.7 93.9 26.4 28.4	96.4 15.4 54.7 96.3 16.0 54.4	96.6 14.5 54.8 96.5 15.2 54.4	96.8 13.6 53.9 96.7 14.3 53.6
ΔE	0.700	0.702	0.720	0.805	0.863
L*a*b*	100.0 -5.8 36.4 100.0 -5.0 36.2	80.4 84.5 44.7 80.3 85.0 44.6	80.5 84.2 44.4 80.3 84.7 44.2	80.6 83.7 44.9 80.4 84.3 44.9	80.8 82.7 44.7 80.7 83.3 44.6
ΔE	0.758	0.527	0.574	0.571	0.608
L*a*b*	79.2 89.8 -19.3 79.1 90.3 -19.2	69.4 97.4 -52.8 69.4 98.2 -52.8	52.0 87.8 -82.7 51.9 88.4 -82.9	88.6 49.2 13.6 88.0 51.7 12.5	65.6 42.6 -59.3 64.6 46.0 -61.1
ΔE	0.504	0.754	0.698	2.803	3.952
L*a*b*	98.0 8.6 3.3 97.8 9.4 2.8	100.0 -17.8 104.8 100.0 -18.1 104.7	56.0 -11.5 64.5 56.1 -11.6 64.5	24.8 -6.6 35.3 24.8 -6.6 35.3	3.4 -1.6 5.2 3.4 -1.6 5.2
ΔE	0.966	0.286	0.135	0.044	0.005
L*a*b*	98.2 8.0 5.8 98.0 8.6 5.5	98.4 6.9 6.7 98.2 7.6 6.5	81.2 -13.8 -32.4 80.9 -12.3 -33.0	82.7 -89.7 -29.7 82.2 -87.1 -30.6	100.0 -163.4 2.8 100.0 -162.2 2.2
ΔE	0.725	0.745	1.643	2.788	1.403
L*a*b*	94.9 22.0 12.9 94.7 22.8 12.6	100.0 -11.5 96.5 100.0 -10.5 96.2	100.0 -12.1 96.6 100.0 -11.0 96.3	100.0 -66.8 31.4 100.0 -63.1 29.9	100.0 -66.9 30.8 100.0 -63.1 29.3
ΔE	0.873	1.002	1.096	4.007	4.076
L*a*b*	75.1 -35.9 -42.9 74.3 -33.2 -44.3	74.5 -34.7 -43.9 73.6 -31.9 -45.4	73.3 -24.1 -46.0 72.4 -21.2 -47.5	73.1 -103.4 23.3 72.4 -99.1 21.8	27.2 -48.1 -11.2 26.8 -45.4 -12.3
ΔE	3.203	3.315	3.395	4.620	2.991
L*a*b*	98.2 8.0 5.7 98.0 8.6 5.4	98.2 7.6 6.0 98.1 8.4 5.6	98.3 7.5 4.8 98.1 8.3 4.3	77.9 -35.6 -38.1 77.3 -33.5 -39.1	41.3 50.8 -101.2 40.9 52.2 -101.8
ΔE	0.733	0.853	0.968	2.368	1.587
L*a*b*	98.1 8.3 2.9 97.9 9.0 2.7	98.3 7.3 -1.4 98.1 8.3 -1.9	78.3 22.0 21.6 78.3 23.1 21.3	37.4 12.2 12.3 37.4 12.8 12.1	35.7 11.4 12.2 35.7 12.1 12.0
ΔE	0.797	1.079	1.133	0.676	0.671
L*a*b*	98.2 7.9 5.0 98.0 8.6 4.7	94.6 23.2 5.1 94.5 23.8 5.0	94.7 22.7 5.2 94.6 23.4 5.0	79.4 0.0 -35.6 79.1 1.6 -36.1	50.6 73.3 -85.2 50.7 73.7 -85.0
ΔE	0.744	0.670	0.763	1.734	0.478
L*a*b*	98.0 8.6 5.6 97.9 9.2 5.4	100.0 -30.8 70.5 100.0 -31.9 71.0	100.0 -47.4 41.2 100.0 -48.4 42.0	100.0 -27.1 56.8 100.0 -27.8 57.6	52.6 -16.5 32.4 53.0 -16.8 33.0
ΔE	0.651	1.278	1.316	1.039	0.729
L*a*b*	98.3 7.2 6.7 98.2 7.8 6.5	98.3 7.4 2.7 98.2 7.9 2.6	98.3 7.2 1.5 98.2 7.8 1.2	37.0 3.0 -1.3 36.9 3.3 -1.6	9.9 28.1 2.8 9.8 28.2 2.7
ΔE	0.638	0.542	0.663	0.453	0.149
L*a*b*	98.1 8.3 2.9 97.9 9.0 2.7	78.9 6.9 0.4 78.9 7.3 0.2	47.6 4.4 47.0 47.8 3.9 47.2	1.1 0.2 1.5 1.1 0.2 1.5	0.0 0.1 0.0 0.0 0.1 0.0
ΔE	0.797	0.480	0.554	0.021	0.000
L*a*b*	98.2 7.9 5.7 98.0 8.6 5.4	95.0 21.5 13.0 94.8 22.3 12.6	95.1 21.0 13.1 94.9 21.9 12.7	95.2 20.6 13.2 95.0 21.6 12.7	95.3 20.2 13.3 95.0 21.3 12.8
ΔE	0.728	0.981	1.085	1.192	1.278
L*a*b*	62.4 96.1 -64.8 62.8 96.9 -64.2	62.1 95.6 -65.3 62.4 96.5 -64.8	73.6 114.0 -27.1 73.4 114.5 -26.2	44.9 110.8 -95.0 44.9 111.3 -95.0	44.4 110.5 -95.9 44.2 111.2 -96.1
ΔE	1.074	1.078	1.101	0.520	0.740
L*a*b*	98.2 7.9 5.7 98.0 8.6 5.4	98.4 6.9 6.7 98.2 7.6 6.5	93.6 27.4 29.0 93.6 27.6 28.9	87.5 53.7 107.8 87.7 53.0 108.0	87.8 52.7 108.0 87.9 52.0 108.2
ΔE	0.729	0.748	0.264	0.770	0.729
L*a*b*	98.2 8.0 5.7 98.0 8.6 5.4	98.2 7.6 5.9 98.1 8.4 5.5	98.2 7.7 2.4 98.0 8.4 1.9	98.4 6.7 3.2 98.3 7.4 2.8	91.2 37.8 31.5 91.1 38.2 31.2
ΔE	0.733	0.862	0.842	0.834	0.549
L*a*b*	100.0 -75.1 77.5 100.0 -74.1 77.3	100.0 -112.6 53.0 100.0 -109.8 52.4	100.0 -112.5 52.0 100.0 -109.8 51.2	57.4 -17.5 -23.5 56.6 -13.1 -25.2	1.7 -1.1 -2.1 1.6 -0.8 -2.3
ΔE	1.023	2.810	2.829	4.729	0.344
L*a*b*	94.0 25.7 29.1 93.9 26.3 28.8	94.2 24.9 29.1 94.1 25.5 28.8	50.6 47.5 -85.2 50.3 48.7 -85.6	53.7 4.0 -79.9 53.0 6.8 -81.1	53.4 5.1 -80.3 52.8 7.7 -81.4
ΔE	0.621	0.706	1.243	3.074	2.902
L*a*b*	98.0 8.4 5.1 97.9 9.1 4.8	98.0 8.5 2.6 97.9 8.9 2.5	92.2 33.5 39.8 92.2 33.8 39.5	92.4 32.9 39.3 92.3 33.2 39.0	40.2 -15.8 -8.7 40.0 -14.6 -9.3
ΔE	0.724	0.387	0.353	0.465	1.374

Table A.3: Test 2 results (see section 4.1.3).

Appendix B

Test Details

B.1 Test Design and Time Measurement

To minimize function call overhead, all non-recursive functions (C++ methods) within measurement periods are inlined using the **inline** statement.

Listing 1 shows the timer macros that were applied for performance measurement. The `clock_gettime` function, if called with the `CLOCK_PROCESS_CPUTIME_ID` argument, delivers the current value of a high-resolution per-process timer. To debunk measurement errors induced by the design of this timing function, the macros are provided in the form of a different implementation using the `rdtsc` machine instruction, which returns the number of cycles since the process start. Due to power-saving capabilities of modern processors, it can not be considered reliable and therefore it is primarily used for verification purposes.

The `RDTSCLL` macro is inline assembly in order to minimize latency times. The `__volatile__` keyword ensures that the code sticks to its location during compilation. The `cpuid` call precedes the `rdtsc` call in order to circumvent out-of-order execution of machine code instructions at run-time. The `eax` register is deleted in order to provide the `cpuid` instruction with a fixed parameter, the output of `cpuid` affects registers `rax...rdx`, therefore they are given in the clobber list. The following line finally executes the `rdtsc` instruction and stores the result in `val` (`=A` → registers `eax` and `edx`).

Delays induced by the time/cycle queries are determined before and after each time measurement by querying the current time/cycle counter twice without intermediate instructions and then subtracted from the measured time.

Listing 1 Time Measurement.

```
1  #include <time.h>
2
3  /*
4   * RDTSC Macro
5   */
6  #define RDTSCLL(val) \
7      __asm__ __volatile__ ("xor %%eax, %%eax; cpuid::: \"%rax\", \"%rbx\", \"%rcx\", \"%rdx\" ); \
8      __asm__ __volatile__ ("rdtsc" : "=A" (val) );
9
10 /*
11  * Operation Timer Macros
12  *
13  * These provide system calls for short-time measurements.
14  *
15  */
16 #ifndef RDTSC_TIME
17
18 #define TIMER_VARIABLES \
19     struct timespec tmeas_start, tmeas_end; \
20     double tmeas_d0, tmeas_d1;
21
22 #define TIMER_START \
23 { \
24     /* first latency measurement */\
25     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tmeas_start); \
26     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tmeas_end); \
27     \
28     tmeas_d0 = (double)(tmeas_end.tv_sec-tmeas_start.tv_sec) + \
29         ((double)(tmeas_end.tv_nsec-tmeas_start.tv_nsec))*0.000000001; \
30     \
31     /* start actual measurement */\
32     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tmeas_start); \
33 }
34 #define TIMER_STOP( timevar ) \
35 { \
36     /* end actual measurement */\
37     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tmeas_end); \
38     timevar += (double)(tmeas_end.tv_sec-tmeas_start.tv_sec) + \
39         ((double)(tmeas_end.tv_nsec-tmeas_start.tv_nsec))*0.000000001; \
40     \
41     /* second latency measurement */\
42     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tmeas_start); \
43     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &tmeas_end); \
44     \
45     tmeas_d1 = (double)(tmeas_end.tv_sec-tmeas_start.tv_sec) + \
46         ((double)(tmeas_end.tv_nsec-tmeas_start.tv_nsec))*0.000000001; \
47     timevar -= (tmeas_d0+tmeas_d1) * 0.5; \
48 }
49
50 #else
51
52 #define TIMER_VARIABLES \
53     unsigned long long tmeas_start, tmeas_end; \
54     double tmeas_d0, tmeas_d1;
55
56 #define TIMER_START \
57 { \
58     /* first latency measurement */\
59     RDTSCLL(tmeas_start); \
60     RDTSCLL(tmeas_end); \
61     \
62     tmeas_d0 = (double)(tmeas_end-tmeas_start); \
63     \
64     /* start actual measurement */\
65     RDTSCLL(tmeas_start); \
66 }
67 #define TIMER_STOP( timevar ) \
68 { \
69     /* end actual measurement */\
70     RDTSCLL(tmeas_end); \
71     timevar += (double)(tmeas_end-tmeas_start); \
72     \
73     /* second latency measurement */\
74     RDTSCLL(tmeas_start); \
75     RDTSCLL(tmeas_end); \
76     \
77     tmeas_d1 = (double)(tmeas_end-tmeas_start); \
78     timevar -= (tmeas_d0+tmeas_d1) * 0.5; \
79 }
80
81 #endif
```

B.2 Operating System, Compiler and Hardware

Operating System (kernel): Linux version 2.6.32-24-generic (builddpalmer) (gcc version 4.4.3 (Ubuntu 4.4.3-4ubuntu5)) #43-Ubuntu SMP Thu Sep 16 14:17:33 UTC 2010 (Ubuntu 10.04 *Lucid Lynx*)

C++ Compiler: g++ (Ubuntu 4.4.3-4ubuntu5) 4.4.3

The tests were run in a single process on a dual-core CPU.

Output of `cat /proc/cpuinfo`:

<pre>processor : 0 vendor_id : GenuineIntel cpu family : 6 model : 15 model name : Intel(R) Core(TM)2 CPU T5500 1.66GHz stepping : 6 cpu MHz : 1667.000 cache size : 2048 KB physical id : 0 siblings : 2 core id : 0 cpu cores : 2 apicid : 0 initial apicid : 0 fdiv_bug : no hlt_bug : no f00f_bug : no coma_bug : no fpu : yes fpu_exception : yes cpuid level : 10 wp : yes flags : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe lm constant_tsc arch_perfmon pebs bts aperfmpperf pni dtes64 monitor ds_cpl est tm2 ssse3 cx16 xtpr pdcm lahf_lm bogomips : 3325.28 clflush size : 64 cache_alignment : 64 address sizes : 36 bits physical, 48 bits virtual power management:</pre>	<pre>processor : 1 vendor_id : GenuineIntel cpu family : 6 model : 15 model name : Intel(R) Core(TM)2 CPU T5500 1.66GHz stepping : 6 cpu MHz : 1667.000 cache size : 2048 KB physical id : 0 siblings : 2 core id : 1 cpu cores : 2 apicid : 1 initial apicid : 1 fdiv_bug : no hlt_bug : no f00f_bug : no coma_bug : no fpu : yes fpu_exception : yes cpuid level : 10 wp : yes flags : fpu vme de pse tsc msr pae mce cx8 apic mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe lm constant_tsc arch_perfmon pebs bts aperfmpperf pni dtes64 monitor ds_cpl est tm2 ssse3 cx16 xtpr pdcm lahf_lm bogomips : 3325.13 clflush size : 64 cache_alignment : 64 address sizes : 36 bits physical, 48 bits virtual power management:</pre>
---	---

Speed Stepping During the tests, speed stepping was disabled by setting the CPU frequency governor to “performance”.

Appendix C

Source Code Listings

The following code snippet (listing 2) shows the algorithm that was applied for all XYZ-to-L*a*b* conversions in the course of this work.

Listing 2 XYZ-to-L*a*b* Conversion (C++).

```
1  float ColorCalc::labFunc( float t, float epsilon, float kappa )
2  {
3      if( t > epsilon )
4          return ::powf( t, 1.f/3.f );
5      else
6          return ( kappa*t + 16.f ) / 116.f;
7  }
8
9  void ColorCalc::xyzToLab
10 {
11     float& l, float& a, float& b,
12     float x, float y, float z,
13     float refWhiteX, float refWhiteY, float refWhiteZ
14 }
15 {
16     float epsilon = 216.f/24389.f;
17     float kappa = 24389.f/27.f;
18
19     float fx = labFunc( x / refWhiteX, epsilon, kappa );
20     float fy = labFunc( y / refWhiteY, epsilon, kappa );
21     float fz = labFunc( z / refWhiteZ, epsilon, kappa );
22
23     l = 116.f*fy - 16.f;
24     a = 500.f * (fx-fy);
25     b = 200.f * (fy-fz);
26 }
```

Listing 3 on the following page shows a C++ implementation of the CIEDE2000 formula. The listing is intended for verification purposes and as a complement to the mathematical description for analyzing purposes.

Listing 3 CIEDE2000 (C++).

```

1  float ColorCalc::deltaEcie2000
2  (
3      float l1, float a1, float b1,
4      float l2, float a2, float b2
5  )
6  {
7      float lBarPrime = ( l1 + l2 ) * 0.5f;
8      float c1 = ::sqrtf( a1*a1 + b1*b1 );
9      float c2 = ::sqrtf( a2*a2 + b2*b2 );
10     float cBar = ( c1 + c2 ) * 0.5f;
11     float cBarPow7 = ::powf( cBar, 7 );
12     float g = ( 1.f - ::sqrtf( cBarPow7 / ( cBarPow7 + 6103515625.f ) ) ) * 0.5f;
13     float a1Prime = a1 * ( 1.f + g );
14     float a2Prime = a2 * ( 1.f + g );
15     float c1Prime = ::sqrtf( a1Prime*a1Prime + b1*b1 );
16     float c2Prime = ::sqrtf( a2Prime*a2Prime + b2*b2 );
17     float cBarPrime = ( c1Prime + c2Prime ) * 0.5f;
18     float h1Prime = ::atan2f( b1, a1Prime );
19     if( h1Prime < 0.f ) h1Prime += 2.f*M_PI;
20     float h2Prime = ::atan2f( b2, a2Prime );
21     if( h2Prime < 0.f ) h2Prime += 2.f*M_PI;
22     float hBarPrime = ( ( h1Prime-h2Prime ) > M_PI )?
23         ( h1Prime + h2Prime + 2.f*M_PI ) * 0.5f:
24         ( h1Prime + h2Prime ) * 0.5f;
25     float t = 1.f
26         - 0.17f * ::cosf( hBarPrime - ( 30.f*M_PI ) / 180.f )
27         + 0.24f * ::cosf( 2.f * hBarPrime )
28         + 0.32f * ::cosf( 3.f * hBarPrime + ( 6.f*M_PI ) / 180.f )
29         - 0.20f * ::cosf( 4.f * hBarPrime - ( 63.f*M_PI ) / 180.f )
30     ;
31     float dhPrime = ( ( h2Prime-h1Prime ) <= M_PI )?
32         h2Prime - h1Prime:
33         ( h2Prime <= h1Prime )?
34             h2Prime - h1Prime + 2.f*M_PI:
35             h2Prime - h1Prime - 2.f*M_PI
36     ;
37     float dLPrime = l2 - l1;
38     float dCPrime = c2Prime - c1Prime;
39     float dHPrime = 2.f * ::sqrtf( c1Prime*c2Prime ) * ::sinf( dhPrime*0.5f );
40     float lBarPrimeMinus50 = lBarPrime - 50.f;
41     float lBarPrimeMinus50Sq = lBarPrimeMinus50*lBarPrimeMinus50;
42     float sL = 1.f +
43         ( 0.015f * lBarPrimeMinus50Sq ) /
44         ::sqrtf( 20.f + lBarPrimeMinus50Sq );
45     float sC = 1.f + 0.045f*cBarPrime;
46     float sH = 1.f + 0.015f*cBarPrime*t;
47     float tmp = ( hBarPrime - ( 275.f*M_PI ) / 180.f ) / 25.f;
48     float dTheta = 30.f * ::expf( -(tmp*tmp) );
49     float cBarPrimePow7 = ::powf( cBarPrime, 7 );
50     float rC = ::sqrtf( cBarPrimePow7 / ( cBarPrimePow7 + 6103515625.f ) );
51     float rT = -2.f * rC * ::sinf( 2.f*dTheta );
52     float kL = 1.f;
53     float kC = 1.f;
54     float kH = 1.f;
55
56     // return deltaE (CIE 2000):
57     float tmpa = dLPrime / ( kL * sL );
58     float tmpb = dCPrime / ( kC * sC );
59     float tmpc = dHPrime / ( kH * sH );
60     return ::sqrtf(
61         tmpa*tmpa + tmpb*tmpb + tmpc*tmpc +
62         rT * ( dCPrime / ( kC * sC ) ) * ( dHPrime / ( kH * sH ) )
63     );
64 }

```

List of Figures

1.1	Light Transport in a Ray Tracer.	13
1.2	Spectral Colors with Color Matching Functions (see section 3.1.1).	14
1.3	Full-Spectral Renderer.	14
2.1	Frequently applied Optimizations: Loop Unrolling and Vector Operations. . .	18
3.1	3D plots of a trichromatic color space and a uniform color space.	21
3.2	CIEDE2000 (red dots) compared to Euclidean (CIE 1976) Perceived Color Difference (black dots) plotted against L^* (plot a), a^* (plot b), b^* (plot c). . .	24
3.3	Acquisition of Spectral Data for Rendering.	26
4.1	Sampled Green Light Spectrum and its Fourier Coefficients.	31
4.2	Spectrum of figure 4.1, samples dropped.	32
4.3	Spectrum of figure 4.1 (upper diagram) multiplied by itself (lower diagram). .	33
4.4	Fourier coefficients of the figure 4.3 spectra.	34
4.5	Development of the color difference between spectra sampled at two rates, (a) Test 1 plots (single spectrum sequences), (b) Test 2 plots (randomly chosen spectra).	36
5.1	Level Generation: Resampling.	43
5.2	Level Generation: Downsampling.	44
6.1	Spike Separation.	50
6.2	Spectral Mipmapping Color Pipeline Thread.	52
6.3	Postprocessing.	53
6.4	Reconstruction of a Spectrum with and without Interpolation.	53
7.1	Standard Cornell Box: Charts of table 7.2.	59
7.2	Standard Cornell Box: Overhead of Function Calls.	59
7.3	Standard Cornell Box. Reference image (a, d), ML result (b), $ML-C$ result (e), color differences ΔE (c: a vs. b; f: d vs. e). The horizontal bar shows the color coding of ΔE	59
7.4	Spiky-Light Cornell Box: Charts of tables 7.5 (chart a) and 7.6 (chart b). . .	61
7.5	Spiky-Light Cornell Box. <i>Ref.</i> (a–c), ML (d–f), $ML-C$ (g–i).	62
7.6	Colorful Scenes: Charts of tables 7.9 (chart a) and 7.10 (chart b).	64
7.7	Multi-Light Scene. Reference image (a), $ML-C$ result (b), color differences ΔE (c). The horizontal bar shows the color coding of ΔE	65
7.8	Colorful Cornell Box. Reference image (a, d), ML result (b), $ML-C$ result (e), color differences ΔE (c: a vs. b; f: d vs. e). The horizontal bar shows the color coding of ΔE	65

7.9	Comparison of Color Difference Thresholds: Development of processing times and resulting mean color differences $\Delta E_{ab,mean}^*$ against rising color difference threshold (abscissa). The processing times are plotted relative to the computation time at $\Delta E_{ab,thresh}^* = 3.0$ (<i>No Opt.</i> : 72millisec; <i>Compil.Opt.</i> : 21millisec; <i>C+Vec.Opt.</i> : 21millisec) and color differences are plotted relative to $\Delta E_{ab,mean}^* = 0.443$ (table 7.12).	67
7.10	(a–e) pixel color errors for pretest color difference thresholds of $\Delta E_{ab,thresh}^* = 3.0$ –7.0, (f) <i>Ref.</i> and (g) <i>ML</i> are the results with $\Delta E_{ab,thresh}^* = 7.0$	68

List of Tables

1.1	Global Illumination Basics.	12
3.1	Opponent colors.	22
3.2	$L^*u^*v^*/L^*a^*b^*$ brightness and color pairs.	22
3.3	Three Component Rendering Constraints.	25
4.1	Test 1 results.	37
4.2	Test 2 results.	37
4.3	Basis Functions: Computational Costs (fp...floating point).	39
4.4	Selection of Spiky Light Sources (see [22]).	39
5.1	Levels as Vector-Length Classes.	42
6.1	Minilight Renderer Capabilities Summary.	46
7.1	Standard Cornell Box: Level Decisions.	57
7.2	Standard Cornell Box: Average single-iteration time in seconds and cycles measured by <i>rdtsc</i> spent processing color-related code.	58
7.3	Spiky-Light Cornell Box: Level Decisions for <i>ML</i> (= equidistant sampling only). 60	
7.4	Spiky-Light Cornell Box: Level Decisions for <i>ML-C</i> (= equidistant sampling + Composite Model).	60
7.5	Spiky-Light Cornell Box: Average single-iteration time in seconds spent processing color-related code.	61
7.6	Spiky-Light Cornell Box (spikes dropped): Average single-iteration time in seconds spent processing color-related code.	61
7.7	Multi-Light Scene: Level Decisions for <i>ML</i> (= equidistant sampling only). . .	63
7.8	Colorful Cornell Box: Level Decisions for <i>ML</i> (= equidistant sampling only). 63	
7.9	Multi-Light Scene: Average single-iteration time in seconds spent processing color-related code.	64
7.10	Colorful Cornell Box: Average single-iteration time in seconds spent processing color-related code.	64
7.11	Comparison of Color Difference Thresholds: Development of Level Decisions for <i>ML</i> (equidistant sampling only).	66
7.12	Comparison of Color Difference Thresholds: Color Errors in the final image. .	66
7.13	Spectrum-by-Spectrum Multiplication.	69
7.14	Spectrum-by-Spectrum Additions.	70
7.15	Clear and Set Operations.	71
A.1	CIE 1931 Standard Colorimetric System: Color-Matching Functions.	76
A.2	Test 1 results (see section 4.1.3).	79

A.3 Test 2 results (see section 4.1.3).	80
---	----

Bibliography

- [1] T. Gockel, *Form der wissenschaftlichen Ausarbeitung*. Springer-Verlag, Heidelberg, 2008, begleitende Materialien unter <http://www.formbuch.de>.
- [2] T. Martyn, “Efficient ray tracing affine IFS attractors,” *Computers & Graphics*, vol. 25, no. 4, pp. 665–670, 2001.
- [3] J. Günther, T. Chen, M. Goesele, I. Wald, and H.-P. Seidel, “Efficient Acquisition and Realistic Rendering of Car Paint,” in *Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV) 2005*, G. Greiner, J. Hornegger, H. Niemann, and M. Stamminger, Eds. Akademische Verlagsgesellschaft Aka GmbH, Nov. 2005, pp. 487–494.
- [4] Trinetram, “3D Rendering Studio,” online source, 2009, <http://www.3drendering-studio.co.uk>.
- [5] Drafix Software Inc., “PRO Landscape,” online source, 2010, <http://www.prolandscape.com>.
- [6] J. T. Kajiya, “The Rendering Equation,” in *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. ACM, 1986, pp. 143–150.
- [7] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, “Modeling the Interaction of Light Between Diffuse Surfaces,” in *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*. ACM, 1984, pp. 213–222.
- [8] R. L. Cook and K. E. Torrance, “A Reflectance Model for Computer Graphics,” *ACM Transactions on Graphics*, vol. 1, no. 1, pp. 7–24, 1982.
- [9] Blender Foundation, “Blender,” online source, 2010, <http://www.blender.org>.
- [10] LuxRender, “LuxRender GPL Physically Based Renderer,” online source, 2010, <http://www.luxrender.net>.
- [11] G. M. Johnson and M. D. Fairchild, “Full-Spectral Color Calculations in Realistic Image Synthesis,” *Computer Graphics and Applications*, vol. 19, no. 4, pp. 47–53, 1999.
- [12] M. S. Peercy, “Linear Color Representations for Full Spectral Rendering,” in *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*. ACM, 1993, pp. 191–198.
- [13] M. S. Peercy and L. Hesselink, “Wavelength selection for true-color holography,” *Applied*

- Optics*, vol. 33, no. 29, pp. 6811–6817, 1994.
- [14] G. Rougeron and B. Péroche, “An Adaptive Representation of Spectral Data for Reflectance Computations,” in *Eurographics Rendering Workshop*. Springer, 1997, pp. 127–138.
 - [15] H. R. Kang, *Computational Color Technology*. The International Society for Optical Engineering, 2006.
 - [16] Y. Sun, “A Spectrum-based Framework for Realistic Image Synthesis,” Ph.D. dissertation, 2000, adviser-F. David Fracchia.
 - [17] A. Wilkie, R. F. Tobler, and W. Purgathofer, “Raytracing of Dispersion Effects in Transparent Materials,” in *Winter School of Computer Graphics Conference Proceedings*, 2000.
 - [18] M. J. Flynn, “Some Computer Organizations and Their Effectiveness,” *IEEE Transactions on Computers*, vol. 21, no. 9, pp. 948–960, 1972.
 - [19] NVIDIA, “CUDA Zone,” online source, 2010, http://www.nvidia.com/object/cuda_home_new.html.
 - [20] Khronos Group, “OpenCL,” online source, 2010, <http://www.khronos.org/opencl>.
 - [21] D. Bulka and D. Mayhew, *Efficient C++*. Addison-Wesley, 2000.
 - [22] G. Wyszecki and W. S. Stiles, *Color Science: Concepts and Methods, Quantitative Data and Formulae*, 2nd ed. Wiley-Interscience, 1982.
 - [23] G. Sharma, W. Wu, E. N. Dalal, and M. U. Celik, “Mathematical Discontinuities in CIEDE2000 Color Difference Computations,” in *Twelfth Color Imaging Conference*. The Society for Imaging Science and Technology, 2004.
 - [24] G. Sharma and H. J. Trussell, “Digital Color Imaging,” *IEEE Transactions on Image Processing*, vol. 6, no. 7, pp. 901–932, 1997.
 - [25] B. Hill, T. Roger, and F. W. Vorhagen, “Comparative Analysis of the Quantization of Color Spaces on the Basis of the CIELAB Color-Difference Formula,” *ACM Transactions on Graphics*, vol. 16, no. 2, pp. 109–154, 1997.
 - [26] G. Sharma, W. Wu, and E. N. Dalal, “The CIEDE2000 Color-Difference Formula: Implementation Notes, Supplementary Test Data, and Mathematical Observations,” *Color Research & Application*, vol. 30, no. 1, pp. 21–30, 2005.
 - [27] J. P. J. Pinel, *Biopsychology*, 6th ed. Pearson Education, 2006.
 - [28] F. H. Imai, M. R. Rosen, and R. S. Berns, “Comparative Study of Metrics for Spectral Match Quality,” in *CGIV 2002*. The Society for Imaging Science and Technology, 2002.
 - [29] O. Föllinger, *Laplace-, Fourier- und z-Transformation*, 9th ed. Hüthig-Verlag, 2007.
 - [30] X-Rite Inc., “X-Rite,” online source, 2010, <http://www.xrite.com/home.aspx>.
 - [31] L. Williams, “Pyramidal Parametrics,” *SIGGRAPH Computer Graphics*, vol. 17, no. 3, 1983.

- [32] “Minilight A Minimal Global Illumination Renderer,” online source, 2010, <http://www.hxa.name/minilight>.
- [33] E. Reinhard, M. Stark, P. Shirley, and J. Ferwerda, “Photographic Tone Reproduction for Digital Images,” *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 267–276, 2002.

