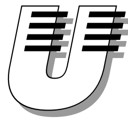



Fachbereich 4: Informatik

# Dynamisches Tone Mapping einer High Dynamic Range Echtzeit 3D-Umgebung mit der Grafik Hardware

## Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von

Philipp Pätzold

Erstgutachter: Prof. Dr. Stefan Müller
Computervisualistik/Arbeitsgruppe Computergraphik

Zweitgutachter: Dipl. Inf. Thorsten Grosch
Computervisualistik/Arbeitsgruppe Computergraphik

Koblenz, im Januar 2007

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

| | Ja | Nein |
|---|---|---|
| Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden. | ☐ | ☐ |
| Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. | ☐ | ☐ |

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(Ort, Datum) (Unterschrift)

## Danksagung

An dieser Stelle möchte ich mich zunächst bei meiner Familie bedanken: Irmgard, Ernst-Günter und Christina, die mich während der Dauer meines gesamten Studiums unterstützt haben und mir in vielen einzelnen Momenten immer tatkräftig zur Seite standen.
Mein ganz besonderer Dank gilt außerdem meiner langjährigen Freundin Parisa, die mir vor allem in schwierigeren Phasen stets zur Seite stand, und ohne die ich mir die Arbeit in dieser Form nicht vorstellen kann.
Weiterhin bedanke ich mich hiermit recht herzlich bei meinem Betreuer Dipl. Inf. Thorsten Grosch, der sich immer genug Zeit nahm, um mir bei Problemen und Fragen behilflich sein zu können und mir dadurch häufig zu neuen Denkansätzen und Ideen verhelfen konnte.
Großer Dank gilt auch den Autoren der Fachliteratur, wobei das insbesondere für diejenigen gilt, die mir für eine Korrespondenz zur Verfügung standen. Dadurch konnten insgesamt viele Verständnisfragen geklärt werden.
Letztendlich spreche ich an dieser Stelle meinen Dank jeder Person aus, die irgendwie an dieser Arbeit beteiligt war und hier namentlich nicht erwähnt ist.

# Inhaltsverzeichnis

**1 Einleitung** **1**
1.1 Motivation . . . . . 1
1.2 Zielsetzung . . . . . 1
1.3 Aufbau . . . . . 2

**2 Tone Mapping** **3**
2.1 Tone Mapping und digitale Bilder . . . . . 3
2.1.1 Berechnung der Leuchtdichten . . . . . 3
2.1.2 Globale Bildgrößen . . . . . 4
2.1.3 Lokale Bildgrößen . . . . . 5
2.1.4 Kompression und Farbbilder . . . . . 6
2.2 Globale Operatoren . . . . . 6
2.3 Lokale Operatoren . . . . . 9

**3 Adaption der menschlichen visuellen Wahrnehmung** **11**
3.1 Aufbau des menschlichen Auges . . . . . 12
3.2 Temporäre Adaption . . . . . 13
3.3 Verlust der Sehschärfe . . . . . 15
3.4 Blendeffekte durch Streulicht . . . . . 16
3.5 Verlust der Farbwahrnehmung . . . . . 18

**4 Moderne 3D-Graphikhardware** **19**
4.1 Effiziente Verarbeitung von Fließkommadaten . . . . . 19
4.2 Programmierbare Graphikpipeline . . . . . 20
4.2.1 Vertexprozessor . . . . . 20
4.2.2 Fragmentprozessor . . . . . 21
4.2.3 Shadersprachen . . . . . 22
4.3 Render-Texturen . . . . . 22

**5 Algorithmen auf der Graphikhardware** **23**
5.1 Grundlagen . . . . . 23
5.2 Parallele Reduktion . . . . . 24
5.3 Effiziente Konvolution von Bilddaten . . . . . 25

**6 Ausgewählte Tone-Mapping-Verfahren** **31**
6.1 Photorezeptor-Tone-Mapping . . . . . 31
6.1.1 Adaptionsmodell für Photorezeptoren . . . . . 31
6.1.2 Benutzerparameter . . . . . 32
6.1.3 Kompression der Bilddaten . . . . . 34
6.1.4 Fazit . . . . . 35
6.2 Histogrammbasiertes Tone Mapping . . . . . 35
6.2.1 Erzeugung des „Fovea-Bildes“ . . . . . 35
6.2.2 Erstellung des Histogramms . . . . . 36

6.2.3 Erzeugung des kumulativen Histogramms . . . . . . . . . 36
6.2.4 Naiver Tone-Mapping-Operator . . . . . . . . . . . . . 37
6.2.5 Histogrammanpassung an einen linearen Schwellwert . . . 37
6.2.6 Histogrammanpassung an die menschliche Kontrastempfindung . . . . . . . . . . . . . . . . . . . . . . . . 39
6.2.7 Fazit . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.3 Tone Mapping auf Basis eines photographischen Modells . . . . . 41
6.3.1 Lineare Skalierung . . . . . . . . . . . . . . . . . . . 42
6.3.2 Globaler Operator . . . . . . . . . . . . . . . . . . . 43
6.3.3 Selektives Aufhellen und Abdunkeln . . . . . . . . . . . 44
6.3.4 Lokaler Operator . . . . . . . . . . . . . . . . . . . . 47
6.3.5 Fazit . . . . . . . . . . . . . . . . . . . . . . . . . . 48

**7 Adaptives Tone Mapping auf der Graphikhardware 51**
7.1 Anforderungen . . . . . . . . . . . . . . . . . . . . . . . . 51
7.2 Entwurf und Design . . . . . . . . . . . . . . . . . . . . . . 52
7.2.1 Globaler Tone Mapper . . . . . . . . . . . . . . . . . . 52
7.2.2 Ergänzung für einen lokalen Operator . . . . . . . . . . . 53
7.2.3 Temporäre Adaptation . . . . . . . . . . . . . . . . . . 59
7.2.4 Verlust der Sehschärfe . . . . . . . . . . . . . . . . . . 63
7.2.5 Blendeffekte durch Streulicht . . . . . . . . . . . . . . . 67
7.2.6 Kompression der Leuchtdichten . . . . . . . . . . . . . 71
7.2.7 Verlust der Farbwahrnehmung . . . . . . . . . . . . . . 72
7.2.8 Gesamtübersicht . . . . . . . . . . . . . . . . . . . . . 73
7.3 Implementierung . . . . . . . . . . . . . . . . . . . . . . . 74
7.3.1 Verwendete Bibliotheken, APIs und Werkzeuge . . . . . . 74
7.3.2 Das *Framework* . . . . . . . . . . . . . . . . . . . . . 74
7.3.3 Die Klasse `PerceptualTonemapper` . . . . . . . . . 77
7.3.4 Fragmentshader . . . . . . . . . . . . . . . . . . . . . 79

**8 Ergebnisse 87**
8.1 Leistungsanalysen . . . . . . . . . . . . . . . . . . . . . . . 87
8.1.1 Leistungsvergleich für die parallele Reduktion . . . . . . 88
8.1.2 Leistungsmessungen für Konvolutionen auf der GPU . . . 88
8.1.3 Leistungsmessungen für das Tone-Mapping-Verfahren . . 90
8.2 Die Testumgebung „TMView“ . . . . . . . . . . . . . . . . . 92
8.3 Integration in einer 3D-Engine . . . . . . . . . . . . . . . . 94

**9 Ausblick 97**

**Literaturverzeichnis 99**

# Abbildungsverzeichnis


1 Resultate verschiedener globaler Operatoren . . . . . . . . . . . . 8
2 Funktionsgraphen globaler Operatoren . . . . . . . . . . . . . . . . 9
3 Lokales Tone Mapping und Halos . . . . . . . . . . . . . . . . . . 10
4 Adaptionszustände und Leuchtdichten . . . . . . . . . . . . . . . . 11
5 Schematischer Aufbau des menschlichen Auges . . . . . . . . . . 13
6 Sensitivität der Stäbchen . . . . . . . . . . . . . . . . . . . . . . 15
7 Funktionsgraph für die Sehschärfeuntersuchungen von Schlaer . . 16
8 Pupillendurchmesser und Hintergrundleuchtdichte . . . . . . . . . 17
9 Die moderne Graphikpipeline von OpenGL 2.0 . . . . . . . . . . 20
10 Parallele Reduktion . . . . . . . . . . . . . . . . . . . . . . . . . 25
11 Separierbare 2D-Faltung . . . . . . . . . . . . . . . . . . . . . . 27
12 Ein 3x3 Binomialfilter mit vier bilinearen Texturzugriffen . . . . . 28
13 Iterative Anwendung des 3x3 Binomialfilters von Kawase . . . . . 29
14 Verschiedene Werte für $f_{exp}$ . . . . . . . . . . . . . . . . . . . 33
15 Photographisches Zonensystem . . . . . . . . . . . . . . . . . . . 42
16 Funktionsgraphen des globalen photographischen Tone-Mapping-Operator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
17 Auswahl einer Center-Region . . . . . . . . . . . . . . . . . . . . 46
18 Relative Leuchtdichten und lokal gemittelte Leuchtdichten . . . . 47
19 Visueller Vergleich zwischen dem lokalen und globalen Operator . 48
20 Filtermasken des Sobel-Operators . . . . . . . . . . . . . . . . . 55
21 Ein Gradientenbild . . . . . . . . . . . . . . . . . . . . . . . . . 56
22 Kantenerhaltender Glättungsfilter . . . . . . . . . . . . . . . . . . 57
23 Visueller Vergleich zweier Tone Mapper . . . . . . . . . . . . . . 58
24 Adaptionszeit für Stäbchen und Zapfen . . . . . . . . . . . . . . 61
25 Temporäre Adaption . . . . . . . . . . . . . . . . . . . . . . . . 62
26 Verschiedene Detailstufen des Sehschärfeverlustes . . . . . . . . 64
27 Auswahl der Mipmap-Stufe . . . . . . . . . . . . . . . . . . . . . 65
28 Wahl der Mipmap-Stufe pro Pixel . . . . . . . . . . . . . . . . . 66
29 Bildartefakte durch angrenzende Mipmap-Stufen . . . . . . . . . 67
30 Verlust der Sehschärfe . . . . . . . . . . . . . . . . . . . . . . . 68
31 Standardabweichung und Pupillengröße . . . . . . . . . . . . . . 69
32 Gaussprofile für die Bildglättung . . . . . . . . . . . . . . . . . . 70
33 Streulichtsimulation . . . . . . . . . . . . . . . . . . . . . . . . . 71
34 Verlust der Farbwahrnehmung . . . . . . . . . . . . . . . . . . . 73
35 Konzeptionelle Gesamtübersicht . . . . . . . . . . . . . . . . . . 75
36 UML-Klassendiagramm des *Framework* . . . . . . . . . . . . . . 78
37 Die Klasse `PerceptualTonemapper` . . . . . . . . . . . . . . 79
38 Leistung der parallelen Reduktion . . . . . . . . . . . . . . . . . 89
39 Leistungsmessungen für Bildkonvolutionen auf der GPU . . . . . 90
40 Leistungsvergleich für verschiedene Tone-Mapping-Operatoren . . 91
41 Leistungsmessung von globalem und lokalem Operator . . . . . . 92

42 Aufwand der einzelnen Teilprozesse . . . . . . . . . . . . . . . . . 93
43 Die Testumgebung „TMView“ . . . . . . . . . . . . . . . . . . . . . 94
44 Integration von Tone Mapping in eine 3D-Engine . . . . . . . . . . 95

# Tabellenverzeichnis


1 Parameter für das Photorezeptor-Tone-Mapping-Verfahren . . . . 32
2 Schnittstelle der Klasse `PerceptualTonemapper` . . . . . . . 79
3 Testkonfiguration für die Leistungsmessungen . . . . . . . . . . . 87
4 Leistungsvergleich für die parallele Reduktion . . . . . . . . . . . 88
5 Leistungsmessung für Bildkonvolutionen auf der GPU . . . . . . 89
6 Leistungsvergleich verschiedener Tone-Mapping-Operatoren . . . 91
7 Leistungsvergleich zwischen globalem und lokalem Operator . . . 92


# Listings


1 `preReduction.fs` . . . . . . . . . . . . . . . . . . . . . . . 80
2 `calcLuminanceData.fs` . . . . . . . . . . . . . . . . . . . . 80
3 `relativeLuminance.fs` . . . . . . . . . . . . . . . . . . . . 81
4 `createAcuityMap.fs` . . . . . . . . . . . . . . . . . . . . . 81
5 `sobelFilter.fs` . . . . . . . . . . . . . . . . . . . . . . . 82
6 `edgePreservingFilter5x5.fs` . . . . . . . . . . . . . . . . 83
7 `glare7x7.fs` . . . . . . . . . . . . . . . . . . . . . . . . . 84
8 `tonemap.fs` . . . . . . . . . . . . . . . . . . . . . . . . . 85

# 1 Einleitung

## 1.1 Motivation

*High Dynamic Range* (HDR) Beleuchtungsverfahren ermöglichen es, dreidimensionale Umgebungen realistisch auszuleuchten. Die dazu notwendigen Berechnungen nutzen intern meist eine hohe numerische Präzision und große Wertebereiche. Bis zum heutigen Zeitpunkt existieren nur wenige Spezialgeräte und Prototypen, die eine direkte Darstellung digitaler HDR-Bilder ermöglichen [SHS+04]. Eine Darstellung von HDR-Bildern auf herkömmlichen Ausgabegeräten, wie beispielsweise auf Standard CRT- oder TFT-Bildschirmen, führt in den meisten Fällen zu schlechten Ergebnissen. Die darstellbaren Leuchtdichtenspektren solcher Geräte fallen deutlich kleiner aus als die für eine korrekte Darstellung benötigten Wertebereiche. Durch eine direkte Abbildung der berechneten Werte auf einem solchen Standardausgabegerät würden sämtliche Werte, die außerhalb des darstellbaren Bereichs liegen, verloren gehen. Dabei könnte der Kontrast- und Helligkeitseindruck des Originalbildes in vielen Fällen nicht ausreichend reproduziert werden.
Dieser Problematik nehmen sich die Verfahren des *Tone Mappings* an [RWPD06]. Dabei werden die Leuchtdichten des HDR-Bildes mit einer geeigneten Technik auf den darstellbaren Bereich des Ausgabegerätes skaliert. Der Skalierungsprozess ist darauf ausgelegt, den subjektiven Helligkeits- und Kontrasteindruck des Originalbildes weitgehend zu erhalten [LRP97]. Hierfür existieren eine Vielzahl unterschiedlicher Verfahren, die jedoch teilweise mit erheblichem Rechenaufwand verbunden sind.
Einige Tone-Mapping-Verfahren simulieren zudem Teile der menschlichen visuellen Wahrnehmung, wodurch der Realismus bei der Darstellung deutlich gesteigert werden kann [LRP97, DD00, KMS05, IFM05]. So kann etwa die Adaption der menschlichen visuellen Wahrnehmung simuliert werden, wodurch Phänomene wie der Sehschärfeverlust im skotopischen Bereich oder Blendeffekte durch Streulicht darstellbar sind.
Hierbei eignen sich die adaptiven Verfahren besonders gut für eine realistische Darstellung interaktiver 3D-Umgebungen mit wechselhaften Beleuchtungsverhältnissen. Für die interaktive Darstellung sind jedoch gleichzeitig hohe Bildwiederholungsraten notwendig. Es wäre daher von Vorteil, wenn die Algorithmen der Tone-Mapping-Verfahren durch die programmierbare Graphikprozessor Einheit (GPU) der Graphikhardware unterstützt würden. Die GPU ist auf Fließkommaarithmetik optimiert und kann viele Berechnungen deutlich schneller ausführen als der Hauptprozessor (CPU) des Hostcomputers.

## 1.2 Zielsetzung

Im Rahmen dieser Arbeit werden zunächst einige Tone-Mapping-Verfahren hinsichtlich Adaption und Echtzeitfähigkeit untersucht. Auf der Basis der gewonnenen Erkenntnisse wird anschließend ein Verfahren implementiert, das einen adapti-

ven Tone Mapper beinhaltet und zudem einige ausgewählte Bereiche der visuellen menschlichen Wahrnehmung simuliert.
Darüber hinaus ist der Tone Mapper ausreichend performant für eine Echtzeitdarstellung einer interaktiven 3D-Umgebung. Um eine hohe Leistung zu gewährleisten, werden große Teile der Berechnung auf die programmierbare GPU der Graphikhardware ausgelagert. Hierbei kommen neue Features aktueller Graphikhardware und moderner Graphik-APIs zum Einsatz. Für die spätere Präsentation und Evaluation des gesamten Verfahrens wird zudem eine geeignete Testumgebung implementiert, mit der eine Navigation in einer 3D-Umgebung möglich ist. Die 3D-Umgebung wird dabei mit statischen HDR-Lightmaps ausgeleuchtet, sodass ein adaptives Tone Mapping sinnvoll ist.

## 1.3 Aufbau

Um in die zugrundeliegende Thematik einzuführen und allgemeine Grundlagen zu schaffen, werden zunächst die Themengebiete „Tone Mapping“ in Kapitel 2 und „Adaption der menschlichen visuellen Wahrnehmung“ in Kapitel 3 behandelt.
Im darauffolgenden Kapitel 4 „Moderne 3D-Graphikhardware“ werden die für diese Arbeit besonders relevanten Features moderner 3D-Graphikhardware beschrieben. Es folgt das Kapitel 5 „Algorithmen auf der Gaphikhardware“, das sich mit einigen Algorithmen befasst, die im Rahmen dieser Arbeit auf der Graphikhardware umgesetzt worden sind.
In Kapitel 6 „Ausgewählte Tone-Mapping-Verfahren“ werden drei Verfahren im Detail behandelt und hinsichtlich der Zielsetzung der Aufgabenstellung untersucht.
In Kapitel 7 „Adaptives Tone Mapping auf der Graphikhardware“ wird ein komplexes adaptives Tone-Mapping-Verfahren mit Unterstützung durch die Graphikhardware vorgestellt. Dabei werden die konzeptionelle Planung und die Implementation des Verfahrens ausführlich dargestellt. Zum Abschluss des Kapitels werden die kommentierten Quellcodes der wichtigsten Shader mit zusätzlichen Erklärungen aufgeführt.
In Kapitel 8 „Ergebnisse“ werden die Resultate zur Evaluation des Verfahrens präsentiert. Darunter befinden sich mehrere Leistungsanalysen und eine Beschreibung der graphischen Testumgebung „TMView“, die ebenfalls im Rahmen dieser Arbeit enstanden ist. Zudem wird erläutert, wie der Tone Mapper in Kombination mit einer bereits bestehenden 3D-Engine eingesetzt worden ist.
Kapitel 9 „Ausblick“ beinhaltet eine kurze kritische Würdigung des im Rahmen dieser Arbeit enstandenen Tone-Mapping-Verfahrens und stellt mögliche Verbesserungen vor.

# 2 Tone Mapping

Damit digitale HDR-Bilder auf Standardausgabegeräten, wie Computerbildschirmen, dargestellt werden können, müssen die Bilddaten skaliert werden. Dafür werden Tone-Mapping-Verfahren benötigt, die eine Kompression der Leuchtdichten vornehmen [RWPD06].
Für die Verfahren sollten jedoch einige Anforderungen gelten, sodass der Gesamteindruck des Ergebnisbildes eine möglichst genaue Reproduktion des Originals darstellt. Nach Ward et al. sollte ein Tone Mapper daher zunächst in der Lage sein, Sichtbarkeiten zu erhalten. Hierbei müssen Objekte, die für einen Betrachter im Originalbild sichtbar sind, ebenfalls im Ergebnisbild vorhanden sein. Es soll keine Information durch Über- oder Unterbelichtung verloren gehen. Weiterhin ist es nach Ward wichtig, dass der subjektive Helligkeits-, Kontrast- und Farbeindruck des Originalbildes weitgehend mit dem Ergebnisbild übereinstimmt [LRP97].
Für die Kompression der Leuchtdichten verwenden die meisten Tone-Mapping-Verfahren spezielle mathematische Operatoren, die auf einer bestimmten Modellvorstellung beruhen. So verwenden viele Verfahren beispielsweise ein vereinfachtes Modell der menschlichen visuellen Wahrnehmung [RD05, RWPD06]. Es existieren aber auch andere Modellvorstellungen, etwa aus dem Bereich der Photographie [RSSF02]. Dabei wird in der Literatur grob zwischen *globalen* und *lokalen* Operatoren unterschieden [RWPD06]. Diese beiden Klassen von Operatoren werden in Kapitel 2.2 und 2.3 dieser Arbeit ausführlicher beschrieben.
Daneben existieren in der Literatur Verfahren, die Phänomene der menschlichen visuellen Wahrnehmung simulieren können, wodurch meist realistischere Ergebnisse erzeugt werden, die eine hohe authentische Reproduktion des Originals darstellen [LRP97, DD00, KMS05, LRP97]. Durch eine Simulation der *temporären Adaption* des Auges ist es möglich, Bildfolgen mit unterschiedlichen Leuchtdichteverteilungen zu verarbeiten und darzustellen. In [DCWP02] geben Devlin et al. eine umfangreiche Übersicht über bestehende Tone-Mapping-Verfahren.
Im Folgenden werden einige grundlegende Größen und Umrechnungsverfahren aufgeführt, die für das Tone Mapping digitaler Bilder von Bedeutung sind.

## 2.1 Tone Mapping und digitale Bilder

Die im Rahmen dieser Arbeit vorgestellten Tone-Mapping-Vefahren verwenden als Eingabe Daten im RGB-Format. Daher werden vorab einige wichtige Verfahren beschrieben, die ein Tone-Mapping auf RGB-Daten ermöglichen. Die nachfolgenden Verfahren und mathematischen Bezeichner werden im Laufe dieser Arbeit immer wieder verwendet.

### 2.1.1 Berechnung der Leuchtdichten

Die Eingabebilddaten sind selten in einem für das Ausgabegerät unabhängigen Format, wie beispielsweise als Komponenten im XYZ-Farbraum, gegeben. Meist lie-

gen die Daten ausschließlich im RGB-Format vor, wobei die HDR-Daten intern mit einer hohen numerischen Präzision gespeichert werden. Mittlerweile existieren zur Speicherung von HDR-Bildern eine Reihe spezieller Datenformate, die eine effiziente Kodierung der Daten bieten und andere Farbräume unterstützen können [RWPD06].
Viele Tone-Mapping-Verfahren leiten daher die Leuchtdichten direkt aus den RGB-Farbkomponenten des Eingabebildes ab [RWPD06]. Die Umrechnung erfolgt dabei über eine Linearkombination der Komponenten des Y-Zeilenvektors einer XYZ-Farbmatrix. Um die photometrische Konsistenz zu bewahren, sollte die XYZ-Matrix individuell für das jeweilige Ausgabegerät und unter Verwendung einer Farbkalibrierung bestimmt werden [Mül05]. Aus pragmatischen Gründen verwenden jedoch viele Tone-Mapping-Verfahren eine standardisierte Umrechnung. In Formel (1) ist die Berechnung einer Leuchtdichte $L_i$ aus RGB-Werten nach dem ITU-R BT.709 Standard aufgeführt [RWPD06]:

$$L_i = 0.2126R + 0.7152G + 0.0722B \tag{1}$$

Nachfolgend wird diese Art der Umrechnung verwendet, wobei die resultierenden Leuchtdichten als einheitslos angesehen werden und ohne die typische Einheit $[cd/m^2]$ angegeben sind.

### 2.1.2 Globale Bildgrößen

Für die Skalierung der Leuchtdichten eines Bildes durch einen Tone-Mapping-Operator kann es nützlich sein, einige globale Bildgrößen zu kennen. Dazu werden häufig die Werte für die *maximale Leuchtdichte* des Bildes $L_{i,max}$ sowie die *minimale Leuchtdichte* des Bildes $L_{i,min}$ verwendet. Zudem wird von vielen Verfahren der Wert für die *durchschnittliche Leuchtdichte* des Bildes $L_{i,avg}$ benötigt [RSSF02]. Der Wert ist als Maß für die Gesamthelligkeit des Bildes zu sehen und kann je nach Verfahren auf unterschiedliche Art berechnet werden [1]. Das *arithmetische Mittel* der Leuchtdichten für ein Bild mit $N = w \cdot h$ Pixel kann hierbei durch Formel (2) berechnet werden [RWPD06]:

$$L_{i,avg} = \frac{1}{N} \sum_{y=0}^{h-1} \sum_{x=0}^{w-1} L_i(x, y) \tag{2}$$

Dabei steht $L_i(x, y)$ für die jeweilige Leuchtdichte eines einzelnen Pixels im Bild. Eine andere Möglichkeit ist durch Formel (3) zur Berechnung des *geometrischen Mittels* gegeben [RWPD06]:

$$L_{i,avg} = \prod_{y=0}^{h-1} \prod_{x=0}^{w-1} (L_i(x, y) + \epsilon)^{\frac{1}{N}}, \quad \epsilon > 0 \tag{3}$$

[1] In der Literatur wird der Wert auch als *Hintergrundleuchtdichte* bezeichnet [RWPD06]

Durch die Addition eines kleinen Wertes $\epsilon$ wird sichergestellt, dass keiner der Produktterme Null wird und so das gesamte Produkt auf Null setzt. In der Praxis wird zudem häufig die *durchschnittliche logarithmische Leuchtdichte* $L_{i,avg}$ verwendet. Diese lässt sich durch Formel (4) berechnen [RWPD06]:

$$L_{i,avg} = \exp\left(\frac{1}{N}\sum_{y=0}^{h-1}\sum_{x=0}^{w-1}(\log(L_i(x,y)+\epsilon))\right), \quad \epsilon > 0 \tag{4}$$

Ein kleiner Wert für $\epsilon$ verhindert, dass der Logarithmus für $L_i(x,y) = 0$ berechnet werden kann. Neben den zuvor beschriebenen globalen Bildgrößen benötigen einige Tone-Mapping-Verfahren oftmals auch lokale Größen, die für einzelne Bildbereiche berechnet werden.

### 2.1.3 Lokale Bildgrößen

Bei Bildern mit einer hohen dynamischen Verteilung der Leuchtdichten können die Helligkeitsunterschiede einzelner Regionen beachtlich sein. Damit der Tone Mapper dennoch gute Ergebnisse erzielt, werden häufig lokale Verfahren eingesetzt [RWPD06]. Die lokalen Operatoren dieser Verfahren verwenden zur Skalierung der einzelnen Leuchtdichten *lokale Mittelwerte*. Die Mittelwerte können dabei aus einer gewichteten Nachbarschaft der jeweiligen Leuchtdichte eines Pixels gewonnen werden. In Formel (5) ist die Berechnung eines solchen Mittelwerts aufgeführt [RWPD06]:

$$L_{i,mean}(p) = \frac{1}{\sum_{i\in\Omega} w(p,i)} \sum_{i\in\Omega} w(p,i) L_i(p) \tag{5}$$

Dabei können die Gewichte $w(p,i)$ für einen Pixel $p$ beispielsweise aus einer Gaussverteilung wie in Formel (6) berechnet werden:

$$w(p,i) = \exp\left(-\frac{||p-i||^2}{s^2}\right) \tag{6}$$

Bei den obigen Gleichungen werden die Nachbarschaftspixel eines Zentrumspixels $p$ durch $\Omega$ repräsentiert. Mit $||p-i||^2$ wird der euklidische Pixelabstand zwischen dem Zentrumspixel und einem Pixel der Nachbarschaft berechnet. Über den Parameter $s$ lässt sich die Gewichtung für unterschiedliche radiale Nachbarschaftsgrößen modifizieren.
Die Konvolution mit Gaussfiltern wird von vielen lokalen Tone-Mapping-Verfahren verwendet [RWPD06]. Weiterhin können Gaussfaltungen für adaptive Verfahren interessant sein. So wird die Gaussfilterung in anderen Teilen dieser Arbeit, beispielsweise zur Simulation von Blendeffekten, genutzt.

### 2.1.4 Kompression und Farbbilder

Nach der Kompression der Leuchtdichten mit Hilfe von Tone-Mapping-Operatoren und globaler oder lokaler Bildgrößen müssen die RGB-Farbkomponenten für die Ausgabe ebenfalls entsprechend skaliert werden. Dabei kann nach Reinhard et al. eine Berechnung der skalierten Farbkomponenten eines Pixels $RGB_d(x, y)$ des Ausgabebildes durch Formel (7) erfolgen [RWPD06]:

$$RGB_d(x, y) = L_d(x, y) \frac{RGB_i(x, y)}{L_i(x, y)} \quad (7)$$

Hierbei ist $L_d(x, y)$ die skalierte Leuchtdichte und $RGB_i(x, y)$ die unkomprimierte RGB-Intensität eines Pixels im HDR-Bild.
Die Sättigung der komprimierten RGB-Farbwerte $RGB_d(x, y)$ kann zudem über einen Exponenten $s$ gesteuert werden. In (8) ist eine entsprechende Formel angegeben [RWPD06]:

$$RGB_d(x, y) = L_d(x, y) \left( \frac{RGB_i(x, y)}{L_i(x, y)} \right)^s, \qquad s \in [0, 1] \quad (8)$$

## 2.2 Globale Operatoren

Globale Tone-Mapping-Operatoren komprimieren einzelne Leuchtdichten unabhängig voneinander mit einer globalen Funktion, die für das gesamte Bild gilt [IFM05]. Dazu können bildabhängige Größen wie die maximale Leuchtdichte $L_{i,max}$, die minimale Leuchtdichte $L_{i,min}$ oder die durchschnittliche Leuchtdichte $L_{i,avg}$ des Bildes aus Kapitel 2.1.2 genutzt werden.
Ein sehr einfacher globaler Tone-Mapping-Operator ist durch die *lineare* Skalierung der Leuchtdichten auf den maximal darstellbaren Bereich des Ausgabegerätes gegeben. In Formel (9) ist die Funktion eines solchen Operators aufgeführt:

$$L_d(x, y) = L_{d,max} \left( \frac{L_i(x, y)}{L_{i,max}} \right) \quad (9)$$

In der obigen Formel ist $L_d(x, y)$ die komprimierte Leuchtdichte eines Pixels, $L_{d,max}$ die maximal darstellbare Leuchtdichte des Ausgabegerätes, $L_i(x, y)$ die unkomprimierte Leuchtdichte des Pixels und $L_{i,max}$ die maximale Leuchtdichte im Bild.
Diese einfache Art der Skalierung führt allerdings nur zu guten Resultaten, wenn die Bilder eine ähnliche Leuchtdichteverteilung aufweisen, die auch auf dem Ausgabegerät darstellbar ist [LRP97]. Bei Bildern mit einer hohen dynamischen Verteilung der Leuchtdichten stößt dieser Operator jedoch schnell an seine Grenzen. Meist ist das Resultat zu dunkel und es werden nur sehr helle Bereiche, wie Lichtquellen oder starke Lichtreflektionen, abgebildet. Eine einfache Verbesserung des Operators lässt sich dadurch erzielen, dass bei der Bestimmung der maximalen

Leuchtdichte $L_{i,max}$ sehr helle Bildpunkte, etwa die der Lichtquellen, ausgenommen werden [Mül05]. Aber selbst mit dieser Vorgehensweise sind die Ergebnisse oftmals nicht zufriedenstellend.
Eine bessere Möglichkeit zur Kompression der Leuchtdichten ist durch eine globale nicht-lineare Skalierung gegeben. Dazu werden in der Praxis häufig *logarithmische* oder *exponentielle* Skalierungsfunktionen verwendet [RWPD06]. In Formel (10) ist die Funktion eines globalen Operators dargestellt, der eine logarithmische Skalierung nutzt [RWPD06]:

$$L_d(x,y) = \frac{log_{10}(1.0 + L_i(x,y))}{log_{10}(1.0 + L_{i,max})} \tag{10}$$

Dieser Operator bildet eine größere Spanne kleinerer Leuchtdichten ab und lässt dadurch das Gesamtergebnis nicht so dunkel erscheinen, wie es etwa bei der linearen Skalierung der Fall ist.
In Formel (11) ist die Funktion für eine exponentielle Skalierung der Leuchtdichten aufgeführt [RWPD06]:

$$L_d(x,y) = L_{d,max}\left(1.0 - exp\left(\frac{-L_i(x,y)}{L_{i,avg}}\right)\right) \tag{11}$$

Dazu wird die durchschnittliche Leuchtdichte des Bildes $L_{i,avg}$ benötigt, wobei Reinhard explizit die Verwendung des arithmetischen Mittels aus Formel (2) vorschlägt [RWPD06].
Diese einfachen Skalierungen können durchaus zu guten Ergebnissen führen, wenn das Eingabebild kein hohes Kontrastverhältnis aufweist [RWPD06]. Daneben existieren in der Literatur weitere mathematische Operatoren für globale Verfahren, die auf anderen Modellvorstellungen basieren [RWPD06]. In Kapitel 6.1 dieser Arbeit wird beispielsweise ein globaler Operator von Reinhard et al. vorgestellt, der auf den komplexen Zusammenhängen der Photorezeptor-Adaption beruht [RD05].
In Abbildung 1 sind die Resultate der vorher aufgeführten globalen Tone-Mapping-Operatoren dargestellt. Die Funktionsgraphen der Tone-Mapping-Operatoren für die lineare, logarithmische und exponentielle Skalierung sind zusätzlich in Abbildung 2 dargestellt. Das Kontrastverhältnis [2], also das Verhältnis der maximalen Leuchtdichte $L_{i,max}$ zur kleinsten Leuchtdichte $L_{i,min}$, lag bei dem Originalbild etwa bei $1 : 1.690.722$. Zusätzlich ist ein Ergebnisbild nach Anwendung des globalen Tone-Mapping-Operators von Reinhard et al. aus [RSSF02] dargestellt. Dieses Verfahren beruht auf einem photographischen Modell und wird ausführlich in Kapitel 6.3 behandelt.
Globale Tone-Mapping-Operatoren lassen sich in vielen Fällen effizient berechnen und werden daher häufig für Echtzeitanwendungen eingesetzt [RWPD06]. Viele der Berechnungen, wie etwa die Bestimmung der durchschnittlichen Leuchtdichte

[2] Wird in der Literatur auch als *Dynamic Range* bezeichnet [RSSF02]

(a) Lineare Skalierung

(b) Logarithmische Skalierung

(c) Exponentielle Skalierung

(d) Globale Skalierung nach Reinhard et al. [RSSF02]

**Abbildung 1:** Resultate verschiedener globaler Tone-Mapping-Operatoren
*Quelle Originalbild: Begleit-DVD [RWPD06]*

$L_{i,avg}$, können zudem effizient durch die programmierbare GPU der Graphikhardware unterstüzt werden. Dies lässt sich durch die Leistungsanalysen aus Kapitel 8.1.1 dieser Arbeit belegen.
Globale Operatoren neigen jedoch bei Bildern, die eine hohe dynamische Verteilung der Leuchtdichten aufweisen, zu Kontrastverlusten [RWPD06]. Durch die Abbildung einzelner Leuchtdichten mit einer globalen Skalierungsfunktion auf den darstellbaren Bereich des Ausgabegerätes wird das verfügbare Spektrum der Leuchtdichten oft nicht optimal ausgenutzt. Daher gibt es eine weitere Klasse von Tone-Mapping-Operatoren, die in den meisten Fällen zu besseren Kontrastverhältnissen führen.

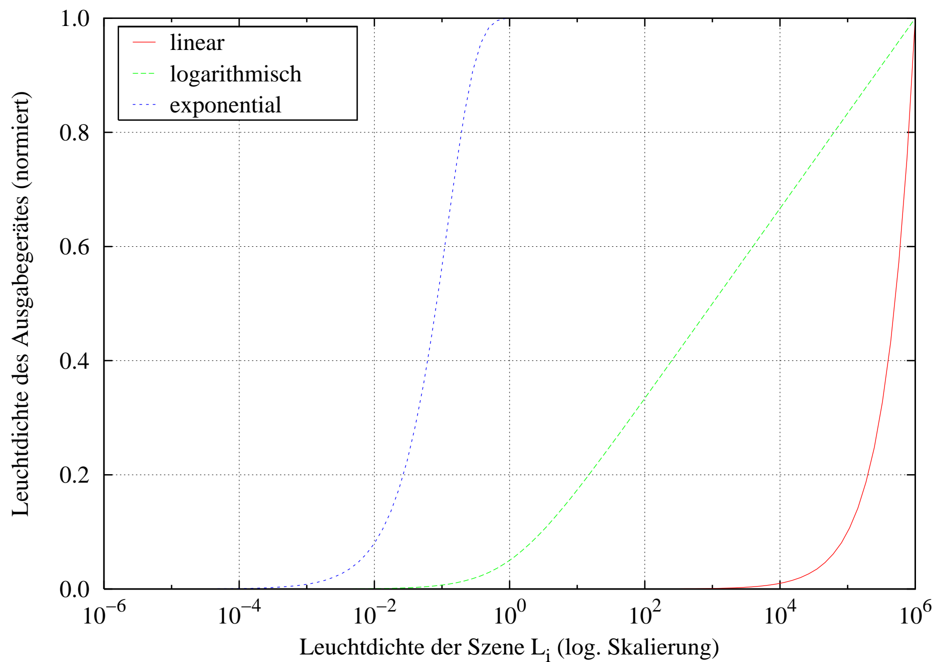



**Abbildung 2:** Funktionsgraphen globaler Tone-Mapping-Operatoren

## 2.3 Lokale Operatoren

Bei der Betrachtung eines Bildes fokussieren die Augen einzelne Bildteile. Man könnte daher annehmen, dass der Betrachter lokal auf Kontraständerungen reagiert und sich entsprechend anpasst. Zu jedem fokussierten Bildpunkt existiert demnach eine umschließende Region, die den Adaptionszustand des Punktes bestimmt [RWPD06]. Es wäre daher naheliegend, statt einer globalen Skalierungsfunktion für alle Leuchtdichten eine variable Funktion zu nutzen, die einzelne Leuchtdichten auf Basis benachbarter Leuchtdichten skaliert. Lokale Tone-Mapping-Verfahren verwenden eine solche Skalierung und können im Vergleich zu den globalen Verfahren häufig zu kontrastreicheren Ergebnisbildern führen [RWPD06].
Lokale Verfahren verwenden zur Kompression der einzelnen Leuchtdichten häufig Mittelwerte, deren Berechnung durch eine gewichtete Mittelung der Leuchtdichten in einer Pixelnachbarschaft erfolgt. Die Mittelwerte können anschließend als Eingabe für eine lokale Operatorfunktion genutzt werden.
Für lokale Tone Mapper ergibt sich oftmals die Fragestellung, wie groß die Pixelnachbarschaft zur Bestimmung eines lokalen Mittels gewählt werden muss, um ein optimales Ergebnis zu erzielen [RWPD06]. Wird die Nachbarschaft zu klein gewählt, führt dies bei den Ergebnisbildern häufig zu lokalen Kontrastverlusten. Bei zu großen Pixelnachbarschaften können hingegen Bildartefakte, sogenannte *Halos*, entstehen. Diese Bildartefakte treten vor allem in Randbereichen von Bild-

(a) Verfahren ohne Artefakte (b) Verfahren mit Artefakten

**Abbildung 3:** Typische Halo-Artefakte eines lokalen Tone-Mapping-Operators
*Quelle Originalbild:*
`http://www.cis.rit.edu/mcsl/icam/hdr/rit_hdr/`

regionen auf, die einen starken Kontrast zum Vorder- oder Hintergrund bilden. Dies ist beispielsweise bei den Lichtquellen in Abbildung 3 der Fall. Dabei werden in der unmittelbaren Nähe von hellen Bildregionen lokale Mittelwerte berechnet, die sowohl Leuchtdichten aus der hellen, als auch aus der dunklen Region enthalten. Im Ergebnisbild führt dies, nach Anwendung der Operatorfunktion, oftmals zu einem unerwünschten sichtbaren Übergang zwischen beiden Bildregionen.
Insgesamt können lokale Tone-Mapping-Verfahren zu kontrastreicheren Bildern führen. Dies gilt insbesondere für Bilder, die über ein hohes Kontrastverhältnis verfügen [RWPD06]. Zur Berechnung der lokalen Mittelwerte werden jedoch teilweise recht große Nachbarschaften eingesetzt, wodurch der Berechnungsaufwand im Vergleich zu globalen Verfahren deutlich höher ausfällt [RSSF02, RWPD06]. Die Verwendung lokaler Operatoren in Kombination mit einer Echtzeitanwendung ist daher stark eingeschränkt. In Kapitel 8.1.3 sind dazu einige Leistungsanalysen aufgeführt.
Im Rahmen dieser Arbeit ist, wie in Kapitel 7.2 ausführlich beschrieben, ein lokales Verfahren entstanden, das zu plausiblen Ergebnissen führt und zugleich eine für Echtzeitanwendungen ausreichende Leistung bietet.

# 3 Adaption der menschlichen visuellen Wahrnehmung

Im Alltag können Leuchtdichten von $10^{-6} cd/m^2$, wie etwa bei einem bewölkten Nachthimmel, bis hin zu $10^7 cd/m^2$ bei der Betrachtung eines sonnenbestrahlten Schneefeldes auftreten [HH06]. Das menschliche visuelle System arbeitet *adaptiv* und kann eine große Spanne von Leuchtdichten verarbeiten [DD00].
Beim Übergang von einer hellen Umgebung zu einer dunklen benötigt der Betrachter zunächst eine gewisse Zeitspanne, bis er erneut im Stande ist, feinere Bilddetails auszumachen. Bei dem umgekehrten Vorgang ist der Betrachter innerhalb der hellen Umgebung zunächst geblendet. Das gewohnte Sehvermögen stellt sich erst nach einer kurzen Zeitspanne erneut ein. Diesen Effekt kennt beispielsweise jeder, der sich für längere Zeit in einem dunklen Raum aufgehalten hat und eine helle Lampe einschaltet.
Weiterhin existieren für das menschliche visuelle System noch andere Phänomene. So ist es bei der Betrachtung von sehr hellen Lichtquellen schwierig, selbst starke Kontraste zu erkennen. Dieser Blendeffekt ist scheinbar auch von dem jeweiligen Adaptionszustand abhängig und wird in einer dunklen Umgebung noch verstärkt. So ist der Betrachter bei Tageslicht von einer hellen Lichtquelle, wie einem Autoscheinwerfer, weit weniger stark geblendet, als während der Nacht.
Im photopischen Bereich, dem Tagessehen, können Farben gut voneinander unterschieden werden. Die Sehschärfe ist ebenfalls weitgehend unbeeinträchtigt. Dagegen wird es im mesopischen Bereich, dem Dämmerungssehen und vor allem im skotopischen Bereich, dem Nachtsehen, für den Betrachter zunehmend schwieriger, Farben zu unterscheiden und Konturen scharf zu erkennen. In Abbildung 4 sind die verschiedenen Leuchtdichten der drei Adaptionszustände für Tages-, Dämmerungs- und Nachtsehen auf einer Skala aufgetragen. Daneben sind exemplarisch einige Werte für Leuchtdichten verschiedener Lichtquellen in der Umwelt aufgeführt.
Um die menschliche visuelle Adaption und die damit verbundenen Phänomene

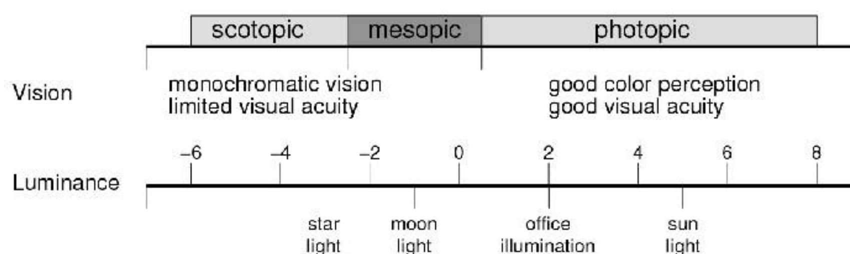



**Abbildung 4:** Skala für Leuchtdichten und die verschiedenen Adaptionszustände
*Quelle: Krawczyk et al.*

nachvollziehen zu können, ist es sinnvoll, zunächst den Aufbau des visuellen Systems im Detail zu beschreiben.

## 3.1 Aufbau des menschlichen Auges

Das optische System des Auges ist ein zusammengesetztes Linsensystem, das auf der Netzhaut ein umgekehrtes, stark verkleinertes Bild der Umwelt entwirft [Sch01]. Hierbei führt der Weg eines Lichtstrahls zunächst durch die Hornhaut, die eine spezifische, unveränderbare Brechkraft besitzt und somit von der Funktion her einer ersten Linse im optischen System entspricht. Danach durchdringt der Lichtstrahl die Pupille, wobei die Pupillenweite auf Basis der einfallenden Lichtmenge reguliert wird. Im Anschluss passiert der Lichtstrahl die bikonvexe elastische Augenlinse, über die eine Fokussierung für Nah- und Fernsehen durchgeführt wird. Bevor der Lichtstrahl schließlich auf die Netzhaut fällt, muss er noch durch den gallertartigen Glaskörper gelangen, der zusammen mit der Linse die letzten zwei Bausteine des dioptrischen Apparats bildet [Sch01].
Die Netzhaut ist ein lichtempfindliches feingegliedertes Häutchen und bedeckt mit etwa $1100mm^2$ einen großen Teil der Innenseite des Augapfels. Sie enthält spezielle Rezeptoren, die auf Photonenreize reagieren. Hierbei gibt es zwei verschiedene Klassen von Photorezeptoren: *Zapfen* und *Stäbchen* [Reh00, Sch01].
Für das Farbsehen sind circa sechs Millionen Zapfen verantwortlich. Sie liegen zumeist konzentriert auf einer kleinen Netzhautfläche und bilden den sogenannten *Gelben Fleck* (Macula lutea). Innerhalb des Gelben Flecks befindet sich eine kleine Vertiefung, die ausschließlich Zapfen enthält und die höchste Ortsauflösung ermöglicht. Die *Sehgrube* (Fovea centralis) liegt im Zentrum des gelben Flecks und ist vor allem für das Scharfsehen im photopischen Bereich verantwortlich, da sie ausschließlich Zapfen enthält [Sch01, Sch98, Reh00].
Die zweite Klasse der Photorezeptoren, die Stäbchen, liegen außerhalb der Fovea centralis. Dabei sind circa 120 Millionen Stäbchen auf der verbleibenden Netzhautfläche vorhanden, wobei ihre Dichte in zunehmendem Abstand von der Fovea centralis wächst. Die Stäbchen sind primär im Dämmerungs- und Nachtsehen als Sensorium für die Helligkeit aktiv, während im mesopischen Bereich beide Klassen von Photorezeptoren aktiv sind [Reh00, Sch01].
Sämtliche Photorezeptoren besitzen spezielle *Photopigmente*. Die Zapfen enthalten drei verschiedene Pigmentarten, die für das trichromatische Farbsehen genutzt werden. Die Zapfenpigmente besitzen verschiedene Absorptionsmaxima, die für Blau bei $440nm$, für Grün bei $540nm$ und für Rot bei $570nm$ liegen. Die Stäbchen besitzen ebenfalls Photopigmente [3], die ein Empfindlichkeitsmaximum bei $510nm$ haben [HH06]. Dabei sind die Stäbchen rund 10.000 mal empfindlicher als die Zapfen [Reh00].
Bei einem Lichteinfall und dem damit verbundenen Kontakt mit Photonen zerfallen die Photopigmente, wobei die nachgeschalteten Nervenzellen unmittelbar informiert werden. Anschließend wird die Information über ein neuronales Netzwerk verarbeitet und an das Gehirn weitergeleitet [Sch01]. Nach diesem Vorgang werden die Photopigmente resynthetisiert. Gleichzeitig wird die Sensitivität der Photorezeptoren erneut aufgebaut [TS97].

[3] Fachbegriffe: Sehpurpur, Rhodopsin

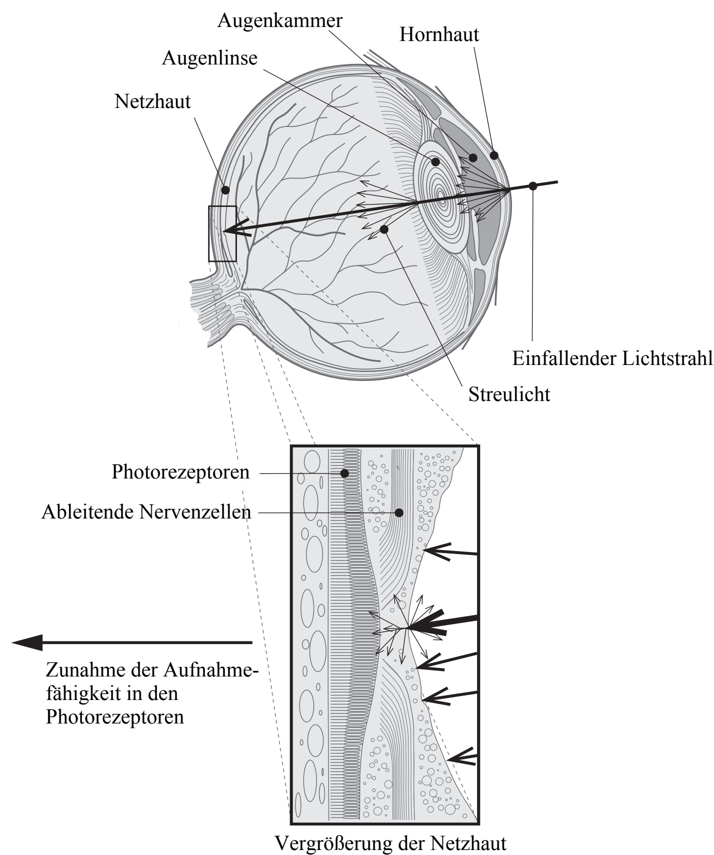



**Abbildung 5:** Schematischer Aufbau des menschlichen Auges
*Quelle: Spencer et al. [SSZG95]*

Die chemischen Photorezeptor-Prozesse sind insofern interessant, da einige Tone-Mapping-Verfahren ein wahrnehmungsbasiertes Modell verwenden, das versucht die Zerfalls- und Resyntheseprozesse mathematisch zu erfassen und in einem Tone-Mapping-Operator zu kapseln [RWPD06, RD05]. In Abbildung 5 ist der Aufbau des menschlichen Auges schematisch dargestellt.

## 3.2 Temporäre Adaption

Gerade bei abrupten Änderungen der Leuchtdichteverhältnisse benötigt das menschliche visuelle System eine gewisse Zeitspanne, bis es wieder die gewohnte Sensitivität erlangt.
Die temporäre Adaption wird von mehreren Komponenten beeinflusst. Dabei sind vor allem die photochemische Komponente der Photorezeptoren und die neuronalen Komponenten relevant [IFM05, TS97]. Die Adaption der Pupillengröße spielt eher eine untergeordnete Rolle [RWPD06].

Die photochemische Komponente ist durch die Zerfalls- und Resyntheseprozesse der Photopigmente innerhalb der Photorezeptoren gegeben. Allgemein hängt die Lichtsensitivität des optischen Systems von der Anzahl der Photopigmente innerhalb der Stäbchen und Zapfen ab. Je mehr Photopigmente zur Verfügung stehen, desto sensibler ist das Auge für den Lichteinfall [She04]. Hierbei wurde beobachtet, dass der zeitliche Verlauf der Adaption für Stäbchen und Zapfen unterschiedlich ist und davon abhängt, ob sich der Betrachter im Zustand einer Hell- oder Dunkeladaption befindet [DD00].
Bei der Dunkeladaption werden in den Stäbchen vermehrt Rhodopsinmoleküle resynthetisiert, wobei der vollständige Resyntheseprozess nach einer kurzen, sehr hellen Beleuchtung der Netzhaut bei Dunkelheit über eine Stunde dauert. Bei diesem Vorgang nimmt die Empfindlichkeit des Auges in den ersten 30 Minuten um fast sechs Zehnerpotenzen zu. Das photopische Sehen der Zapfen geht am Ende der Zapfenadaption nach etwa acht bis zehn Minuten in das skotopische Sehen der Stäbchen über [Sch98].
Wird das Auge hingegen einer hellen Umgebung ausgesetzt, zerfallen viele Rhodopsinmoleküle innerhalb der Stäbchen gleichzeitig. Die Photorezeptorsensitivität wird fast unmittelbar verringert, wodurch der Betrachter für kurze Zeit geblendet werden kann. Dieser Prozess ist im Gegensatz zur Dunkeladaption meist binnen Sekunden abgeschlossen [HH06]. Weiterhin zerfallen die Photopigmente der Stäbchen beim Übergang vom mesopischen zum photopischen Bereich nahezu vollständig. Daher wird angenommen, dass die Stäbchen im photopischen Bereich für den Sehprozess nicht relevant sind [RWPD06]. Die Sensitivität der Stäbchen ist für verschiedene Leuchtdichten empirisch erfasst worden. Nach Hunt kann die Sensitivität eines Stäbchens zu einer gegebenen Leuchtdichte $L$ durch die Funktion $\sigma(L)$ in Formel (12) approximiert werden [Hun85]:

$$\sigma(L) = \frac{0.04}{0.04 + L} \tag{12}$$

In Abbildung 6 ist der Funktionsgraph der Stäbchensensitivität $\sigma(L)$ nach Hunt für verschiedene Leuchtdichten $L$ aufgeführt.
Neben den photochemischen Komponenten der Photorezeptoren spielen neuronale Komponenten für den Adaptionsprozess ebenfalls eine wichtige Rolle. Dabei wird zwischen schneller und langsamer neuronaler Adaption unterschieden [IFM05]. Die schnelle neuronale Adaption wird durch einen zentral-adaptiven Mechanismus gesteuert, der bei der Dunkeladaption die Wahrnehmung des Auges vom Zapfensystem auf das Stäbchensystem umschaltet. Durch den langsamen neuronalen Mechanismus werden die einfallenden Leuchtdichten während der Dunkeladaption ständig gemessen, wobei die Schwellenreizstärke der Stäbchen kontinuierlich angepasst wird. Diese Reizstärke gibt einen Wert an, der überschritten werden muss, damit die Photopigmente der Stäbchen zerfallen und dadurch ein Informationsaustausch mit dem Gehirn stattfindet [TS97].

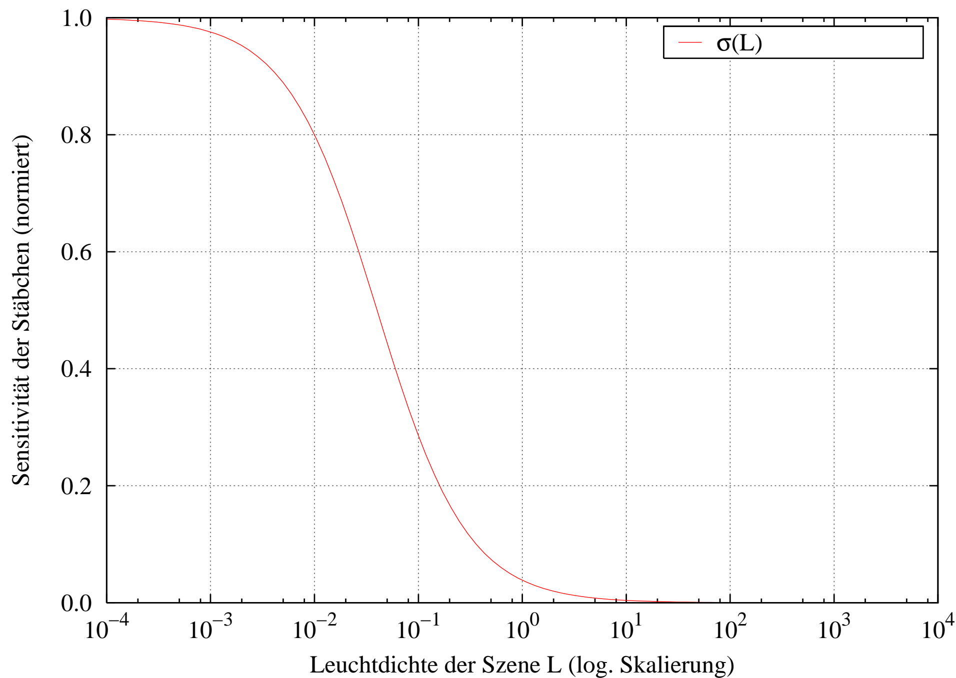



**Abbildung 6:** Sensitivität der Stäbchen nach Hunt [Hun85]

## 3.3 Verlust der Sehschärfe

Im *mesopischen* und *skotopischen* Bereich verliert das menschliche visuelle System zunehmend die Fähigkeit, räumliche Details aufzulösen [LRP97]. Mit dem Verlust der Sehschärfe wirken Konturen verschwommen, und eine Abgrenzung der Objekte in der Umwelt wird erschwert.

Das Sehschärfevermögen ist jedoch nicht überall auf der Netzhaut gleichermaßen vorhanden. Am höchsten ist die Sehschärfe in der Sehgrube innerhalb des Gelben Flecks. Dort sind die für das Scharfsehen verantwortlichen Zapfen in einer hohen Konzentration vorhanden. Die Mehrzahl der Zapfen haben einen dedizierten ableitenden Nerv zum Gehirn. Bei einem Lichteinfall auf die Fovea centralis können daher einzelne Zapfen individuell kodierte Signale an das Gehirn weiterleiten. Die Information wird dadurch unabhängig von benachbarten Zapfen versendet [She04, BB04].

Die Sehschärfe nimmt mit zunehmendem Abstand von der Fovea centralis ab, wobei die Netzhaut zunehmend dichter von Stäbchen bekleidet wird. Die Signale der Stäbchen, die meist zu Bündeln von circa 100 Stück zusammengeschaltet sind, werden durch spezielle Zwischenzellen konvergiert und jeweils an einem Gehirnneuron weitergeleitet [She04]. Dadurch ist die Ortsauflösung der Stäbchen weitaus kleiner als die der Zapfen.

Beim Übergang vom mesopischen zum skotopischen Bereich verlieren die Zapfen

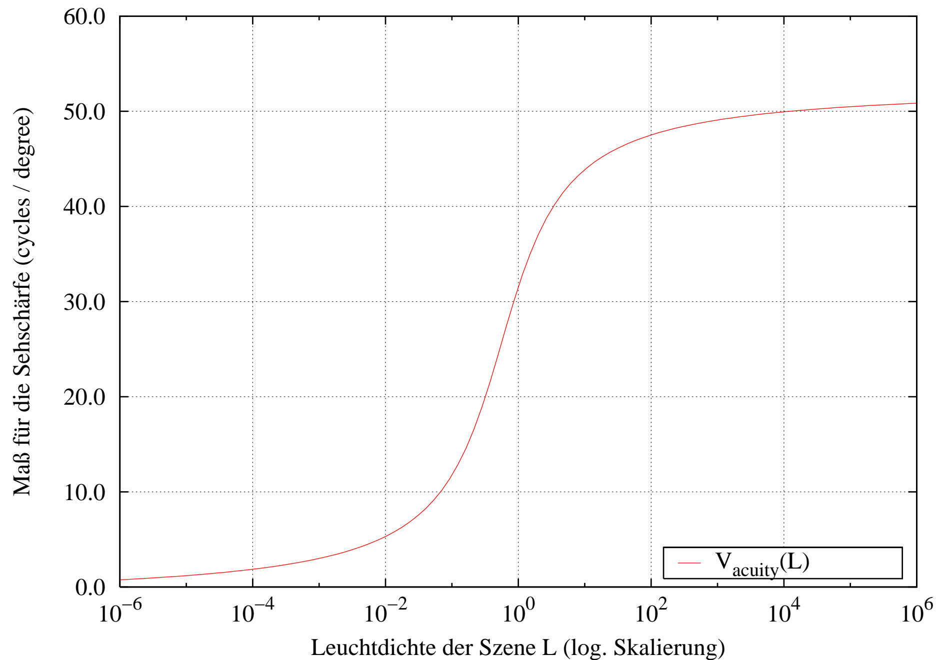



**Abbildung 7:** Eine approximierende Funktion für die Messungen zur Sehschärfe von Shlaer [LRP97]

ihre Sensitivität, bis sie wegen der Umschaltung der schnellen neuronalen Komponenten komplett inaktiv werden [HH06]. Damit ist die Sehschärfe direkt von den Beleuchtungsverhältnissen abhängig. Der Grad der Sehschärfe ist für verschiedene Leuchtdichteverhältnisse von Schlaer in [Shl37] empirisch erfasst worden. Die experimentellen Werte wurden von Ward et al. in [LRP97] durch die Funktion in Formel (13) approximiert.

$$V_{acuity}(L) = 17.25 \cdot arctan(1.4 \log_{10} L + 0.35) + 25.72 \tag{13}$$

Die Funktionswerte von $V_{acuity}(L)$ geben eine Obergrenze für eine räumliche Frequenz an, die zu einer gegebenen Leuchtdichte $L$ noch vom visuellen System aufgelöst werden kann [KMS05]. In Abbildung 7 ist der Funktionsgraph von Ward dargestellt.

## 3.4 Blendeffekte durch Streulicht

Bei der Betrachtung von Objekten in unmittelbarer Nähe heller Lichtquellen fällt auf, dass feinere Strukturen aufgrund des eingeschränkten Kontrastes nur sehr schlecht auszumachen sind. Weiterhin kann ein Blendeffekt auftreten, der die Sichtbarkeit stark einschränkt [Fah05]. Der Effekt tritt dabei im verstärkten Maße im

mesopischen und im skotopischen Bereich auf [SSZG95].
Dieses Phänomen des menschlichen visuellen Systems entsteht durch Streulicht. Dabei wird einfallendes Licht durch die Hornhaut, die Linse und die erste Schicht der Netzhaut gestreut, bevor es schließlich die Photorezeptoren erreicht. Da die Stäbchen keine hohe Ortsauflösung ermöglichen, tritt der Effekt verstärkt im mesopischen und skotopischen Bereich auf, wobei der Pupillenreflex ebenfalls von Bedeutung ist [SSZG95].
Bei einer Zu- oder Abnahme der Leuchtdichte wird die Pupillenweite durch den Pupillenreflex verkleinert beziehungsweise vergrößert. Bei einer schnellen Zunahme der Leuchtdichten wird die Pupille entsprechend schnell verkleinert, wodurch ein erster Schutz ermöglicht wird [Sch98]. Dabei hängt die einfallende Lichtmenge und damit die Streulichtmenge linear von der Pupillenfläche ab. Die Lichtmenge kann sich bis zu 25fach verkleinern, wenn der Pupillendurchmesser von 7,5 auf 1,5 mm abnimmt [Sch98].
Nach empirischen Messungen von Moon und Spencer kann der Pupillendurchmesser in Abhängigkeit von der durchschnittlichen Hintergrundleuchtdichte $L_{avg}$ durch Formel (14) geschätzt werden [SSZG95]:

$$PD(L_{avg}) = 4.9 - 3\tanh\left(0.4\left(\log_{10} L_{avg} + 1.0\right)\right) \tag{14}$$

Die Funktion ist nochmals graphisch in Abbildung 8 dargestellt.

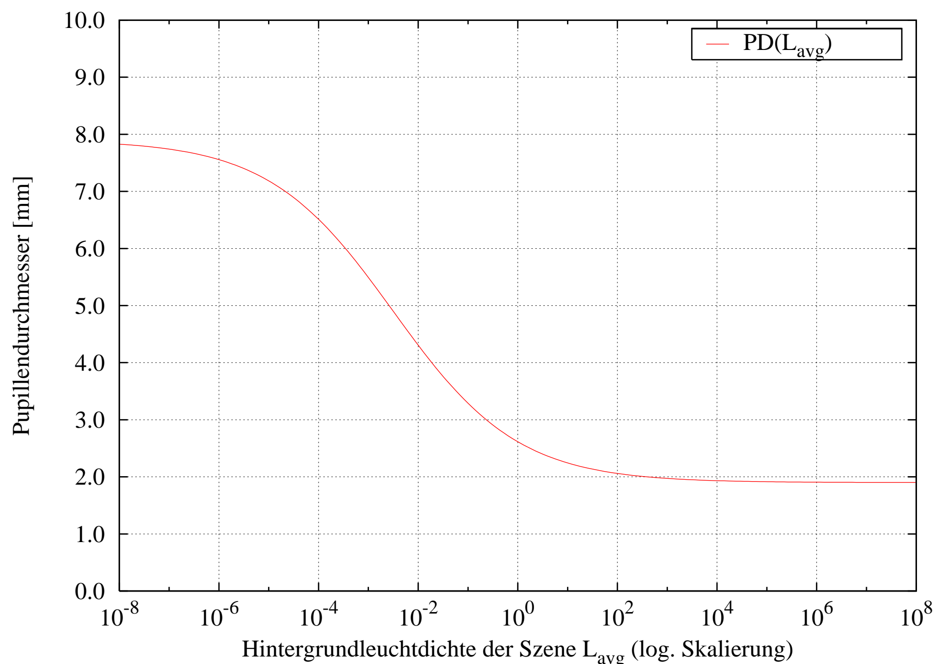



**Abbildung 8:** Pupillendurchmesser für verschiedene durchschnittliche Hintergrundleuchtdichten [RWPD06]

### 3.5 Verlust der Farbwahrnehmung

Die Fähigkeit der menschlichen visuellen Wahrnehmung Farben voneinander zu unterscheiden, ist nur im photopischen und eingeschränkt im mesopischen Adaptionsbereich gegeben.
Beim Übergang vom mesopischen zum skotopischen Bereich werden immer weniger Zapfen angeregt, wobei gleichzeitig jedoch die Sensitivität der Stäbchen steigt. Dadurch verschlechtert sich die Farbwahrnehmung kontinuierlich, bis schließlich im skotopischen Bereich keine echte Farbunterscheidung mehr möglich ist [Sch98].
Neben dem Verlust der Farbwahrnehmung wird angenommen, dass ein weiteres perzeptuelles Phänomen beim Nachtsehen auftreten kann. Damit ist eine leichte Farbverschiebung gemeint, die Nachtszenen häufig leicht blaustichig erscheinen lässt [WJPS$^+$00].
Während der Dunkeladaption verschiebt sich das mittlere Absorptionsmaximum des trichromatischen Sehens vom photopischen Bereich bei $550nm$ bis hin zum Hell-Dunkel-Sehen im skotopischen Bereich bei $510nm$. Aufgrund der Verschiebung des Absorptionsmaximums werden blaue Farbtöne im skotopischen Bereich heller wahrgenommen. Dieser Effekt ist als Purkinje-Phänomen [4] bekannt und wird häufig künstlerisch in Bildern, Photos und Filmen durch eine angepasste Farbpalette für Nachtszenen umgesetzt [HH06, WJPS$^+$00].
Eine weitere Theorie besagt, dass die Stäbchen neuronale Pfade mit einigen Zapfen teilen, die besonders für kurze Wellenlängen sensitiv sind [DD00].

[4] Jan Evangelista Purkinje, tschechischer Physiologe 1787-1869 [Tec05]

# 4 Moderne 3D-Graphikhardware

In den letzten Jahren ist es im Bereich der 3D-Graphikhardware zu einigen bemerkenswerten Innovationen gekommen. Es ist vor allem dem Konkurrenzkampf einiger Hersteller zu verdanken, dass der Kunde regelmäßig mit neuartigen Graphiklösungen versorgt werden konnte. Moderne 3D-Graphik-APIs, wie OpenGL 2.0 oder Direct3D 9.0, brachten außerdem eine hinreichende Unterstützung neuerer Graphikfeatures, sodass sich diese schnell als Standard etablieren konnten. Jede neue Generation von Graphikchips war der vorherigen Generation im Hinblick auf Features und Performanz meist deutlich überlegen. Gleichzeitig hat sich das Preis-Leistungs-Verhältnis moderner 3D-Graphikhardware ebenfalls stetig verbessert. Heutzutage sind schnelle Graphiklösungen für die breite Masse verfügbar und nicht nur Profianwendern mit dem entsprechenden Budget vorbehalten.
Im Folgenden werden einige wichtige Entwicklungen der letzten Jahre aus dem Bereich der Graphikhardware näher beschrieben. Dabei wird besonders auf neuere Features eingegangen, die im Kontext dieser Arbeit relevant sind.

## 4.1 Effiziente Verarbeitung von Fließkommadaten

Für die interaktive Darstellung einer 3D-Szene ist unter anderem die schnelle Verarbeitung von *Fließkommaoperationen* wichtig. Die Transformation und Beleuchtung dreidimensionaler Objekte erfordert eine Vielzahl dieser Operationen, wobei gerade komplexere Oberflächenmaterialien für die 3D-Objekte mehrere Operationen pro Fragment erfordern. Aus diesem Grund wurde die Graphikhardware in den letzten Jahren vor allem hinsichtlich ihrer Fließkommaleistung optimiert. Das Rechenwerk einer modernen Graphikkarte, die *Graphics Programming Unit* (GPU), kann Fließkommaoperationen wesentlich schneller berechnen als ihr Pendant, die *Central Processing Unit* (CPU) des Hostrechners. Weiterhin ermöglicht das hochgradig parallele Design der GPU, mehrere Berechnungsoperationen in einem Taktzyklus gleichzeitig durchzuführen. Auf aktueller Graphikhardware finden sich zudem breite Speicheranbindungen in Kombination mit einem hochgetakteten dedizierten Speicher [5]. Damit ist die theoretische Speicherbandbreite deutlich größer als die des Hostrechners, wodurch ebenfalls Leistungsvorteile enstehen können [Ver04].
Eine weitere Innovation in diesem Bereich ist durch die Unterstützung von Texturdaten im Fließkommaformat gegeben. Hierbei werden durch die Graphik-APIs spezielle Texturformate bereitgestellt, die eine Speicherung der Daten in einem Fließkommaformat direkt auf dem Graphikspeicher erlauben. Solche Texturen stellen mitunter erhebliche Anforderungen an die Speicherausstattung und die Speicherbandbreite der Graphikhardware. Daher werden in der Regel verschiedene Datenformate für 16-Bit, 24-Bit und 32-Bit Präzision pro Farbkanal angeboten. Desweiteren unterstützt aktuelle Hardware spezielle Fließkommaformate, die Texturdaten mit weniger als vier Komponenten speichern können [Ver05].

[5] nVidia 7900 Serie mit 256-Bit Speicherbusbreite und GDDR-RAM3 [nVi06b]

## 4.2 Programmierbare Graphikpipeline

Neben der Forderung einer erweiterten und effizienten Fließkommaunterstützung ist auch die Forderung nach mehr Flexibilität gewachsen. Viele graphische Effekte und neue Beleuchtungsmodelle waren mit der klassischen Graphikpipeline kaum mehr zu realisieren oder mussten durch mehrere Renderdurchläufe (Multipass) berechnet werden [Ros04a]. Durch eine Integration spezialisierter, programmierbarer Bausteine in die GPU, die sogenannten *Vertex- und Fragmentprozessoren*, konnte die Graphikpipeline deutlich flexibler gestaltet werden [Ros04a]. Dabei wurden einzelne Stufen der Graphikpipeline, die zuvor nur über einen festen Funktionsumfang verfügten, durch programmierbare Stufen ersetzt. In Abbildung 9 ist eine solche programmierbare Graphikpipeline am Beispiel von OpenGL 2.0 dargestellt. Die Vertex- und Fragmentprozessoren sind auf Fließkommaoperationen optimiert,

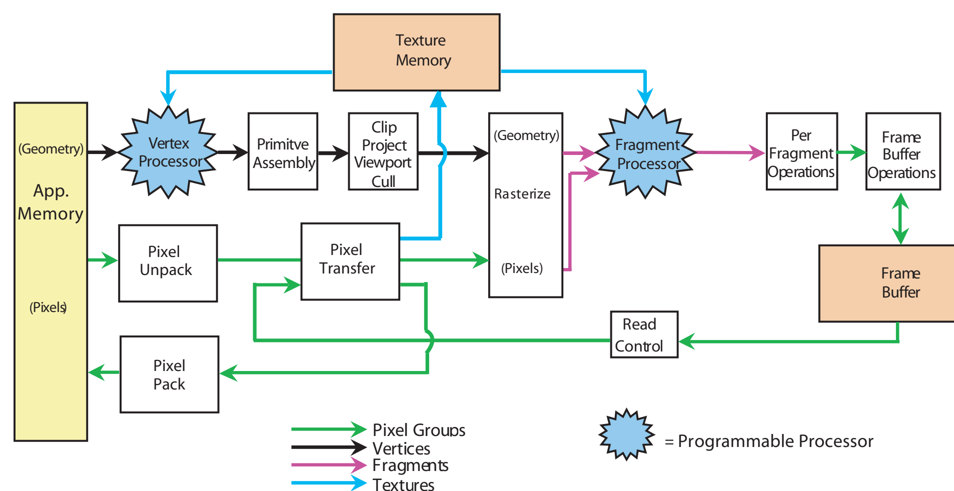



**Abbildung 9:** Die moderne Graphikpipeline von OpenGL 2.0

*Quelle: Randi J. Rost [Ros04b]*

wobei auf aktueller Graphikhardware meist mehrere solcher Einheiten integriert sind [6]. Somit wird eine parallele Verarbeitung der Daten ermöglicht und die Leistung insgesamt gesteigert.

### 4.2.1 Vertexprozessor

Der *Vertexprozessor* übernimmt Aufgaben, die pro Eckpunkt (Vertex) und zugehöriger Vertexattribute wie Normale, Texturkoordinate oder Farbwert in der zu berechnenden Szene anfallen [Ros04a]. Damit sind in der Regel folgende Aufgaben gemeint:

[6] ATI Radeon X1900 verwendet acht Vertex Shader (Vertexprozessoren) und 48 Pixel Shader (Fragmentprozessoren) [BW06]

- Transformation von Eckpunkten und Eckpunktattributen
- Generierung von Texturkoordinaten
- Beleuchtungsberechnungen pro Eckpunkt

Natürlich kann der Vertexprozessor auch andere Berechnungen durchführen. Um eine parallele Verarbeitung durch die Vertexprozessoren zu ermöglichen, ist pro Vertexprozessor immer nur der Zugriff auf einen Eckpunkt und die damit verknüpften Attribute erlaubt. Weiterhin können im Vertexprozessor bestehende Daten lediglich modifiziert werden. Es kann jedoch keine neue Geometrie hinzugefügt oder entfernt werden. Außerdem bleiben einige Aufgaben der „festen Graphikpipeline" vorbehalten und sind nicht über die Vertexprozessoren programmierbar. Unter anderem sind dies:

- Perspektivische Projektion
- Clipping
- Backface Culling [7]

### 4.2.2 Fragmentprozessor

Der *Fragmentprozessor* übernimmt Aufgaben, die pro eingehendes Fragment nach der Rasterisierung der Primitive anfallen [Ros04a]. Diese Aufgaben beinhalten:

- Texturierung pro Fragment
- Beleuchtung pro Fragment

Dabei umfasst ein *Fragment* eine ganze Reihe von Daten:

- Fensterkoordinaten nach der Rasterisierung
- Interpolierte Farbwerte
- Interpolierte Normale
- Interpolierte Texturkoordinate(n)
- Einträge im Tiefenpuffer [8]

Aufgrund der Architektur müssen auch hier sämtliche Berechnungen parallelisierbar bleiben. Pro Fragmentprozessor ist daher nur der Zugriff auf jeweils ein Fragment gleichzeitig möglich. Es können keine neuen Fragmente generiert werden. Zudem lassen sich einige Attribute der Fragmente, wie zum Beispiel die Positionsdaten, nicht mehr verändern.

[7] Entfernung abgewandter Polygone
[8] Depth Buffer bei OpenGL

### 4.2.3 Shadersprachen

Die Programmierung der Vertex- und Fragmentprozessoren wird mit speziellen Programmiersprachen, den sogenannten *Shadersprachen*, vorgenommen. Hierbei haben sich die Hochsprachen Cg von nVidia, HLSL von Microsoft und die OpenGL Shading Language etabliert. Programme, die auf Vertexprozessor und Fragmentprozessor ausgeführt werden, heißen *Vertexshader* beziehungsweise *Fragmentshader* [9] [Ros04a]. Die Programmierung in den Shaderhochsprachen ist stark an die prozedurale Programmiersprache C angelehnt. Bevor sich die Shaderhochsprachen zur Programmierung von Shadern durchsetzen konnten, wurde die Programmierung häufig in assemblerähnlichen Sprachen vorgenommen. Aufgrund der niedrigen Abstraktionsebene, der schlechten Portabilität und Wartbarkeit, ist die direkte Programmierung durch diese maschinennahen Sprachen mehr und mehr in den Hintergrund gedrängt worden.

## 4.3 Render-Texturen

Mit moderner Graphikhardware und 3D-Graphik-APIs ist es möglich, Daten direkt in den Texturspeicher zu schreiben [10]. Der Fragmentprozessor kann dabei die Ergebnisse seiner Berechnungen in eine oder mehrere Texturen schreiben. Damit können beispielsweise Zwischenergebnisse berechnet werden und erneut als Eingabewerte für einen Shader auf dem Fragment- und Vertexprozessor dienen.
Mit diesem *Feedbackmechanismus* sind neue Anwendungen denkbar. So lassen sich viele Algorithmen auf der Graphikhardware durch eine Kombination von Fließkommatexturen und Shaderprogrammierung realisieren. Aufgrund der hohen Fließkommaleistung können solche Algorithmen teilweise um ein Vielfaches schneller ausgeführt werden als auf dem Hauptprozessor des Hostsystems.

[9] In der Literatur auch als „Pixelshader“ bekannt

[10] Wird in der Literatur auch als *Render-To-Texture* (RTT) bezeichnet [Mem06b]

# 5 Algorithmen auf der Graphikhardware

Aufgrund der flexiblen Programmierbarkeit und der hohen Rechengeschwindigkeit moderner Graphikhardware bietet es sich an, traditionelle Algorithmen auf der GPU umzusetzen.
In diesem Kapitel wird beschrieben, wie ausgewählte Algorithmen effektiv auf der Graphikhardware umgesetzt werden können. Es werden zwei Algorithmen, die *parallele Reduktion* und die *Konvolution von Bilddaten*, welche im Rahmen dieser Arbeit besonders relevant waren, im Detail vorgestellt. Im Folgenden werden zunächst Grundlagen und Prinzipien des Programmiermodells der Graphikhardware angesprochen.

## 5.1 Grundlagen

Die *Graphics Processing Unit* der Graphikhardware ist mit ihren programmierbaren Bausteinen, den Vertex- und Fragmentprozessoren, für eine hochgradig parallele Verarbeitung von Daten ausgelegt. Dabei kann die GPU ganze Datenströme verarbeiten, die beispielsweise in Form von Vektordaten für Eckpunkte und Texturen vorliegen können. Das zugrundeliegende Programmiermodell wird daher auch als „Stream Programming Model“ bezeichnet [Ver05].
Dieses Modell unterscheidet sich grundlegend von dem traditionellen Programmiermodell sequentieller Prozessoren. Um Algorithmen von der CPU auf die GPU portieren zu können, sind teilweise erhebliche Modifikationen notwendig. Hierbei ist eine Umsetzung einiger spezieller Standardalgorithmen auf der GPU mitunter wenig sinnvoll [Ver04].
Viele Standardalgorithmen nutzen im Kern zwei Klassen von Operationen: *Scatter* und *Gather*. Als Scatter-Operation werden diejenigen Operationen bezeichnet, die Daten an eine zuvor berechnete Speicheradresse schreiben [Ver05]. Die Operation $a[i] = x$ ist beispielsweise als typische Scatter-Operation anzusehen, wobei ein Wert $x$ an eine zuvor berechnete Adresse $i$ des Datenfeldes $a$ geschrieben wird. Die zum Scattering komplementäre Operation ist die sogenannte Gather-Operation. Damit ist die Klasse derjenigen Operationen gemeint, die Daten von einer zuvor berechneten Adresse auslesen [Ver05]. Analog zu dem vorangegangenen Beispiel ist die Operation $x = a[i]$ als typischer Vertreter der Gather-Operationen anzusehen.
Das Konzept von Gather- und Scatteroperation lässt sich auch auf die programmierbaren Bausteine der GPU übertragen [Ver05]. Der Vertexprozessor der Graphikhardware transformiert Eckpunkte und deren Attribute. Er ist demnach in der Lage, Daten zu verteilen (Scatter). Im Vergleich dazu ist es für den Fragmentprozessor nur eingeschränkt möglich, eine Verteilung von Daten vorzunehmen. Während der Rasterisierung kann der Fragmentprozessor seine Ausgabewerte lediglich an feste sequentielle Rasterpositionen schreiben. Allerdings kann er Texturdaten einlesen, wobei die notwendigen Texturzugriffe über Texturkoordinaten an beliebigen Positionen in der Textur stattfinden können. Diese Vorgänge können demnach als

typische Gatherschritte aufgefasst werden.
Nachfolgend werden zwei Algorithmen, die im Rahmen dieser Arbeit von besonderem Interesse sind, detailliert beschrieben.

## 5.2 Parallele Reduktion

In vielen Fällen ist es notwendig, aus einer gegebenen Menge von Werten einen Einzelwert zu berechnen, der in einer mathematischen Beziehung zu den anderen Werten steht. Ein typisches Beispiel ist die Berechnung eines Mittelwerts aus einer gegebenen Menge von Werten. Auf einem sequentiellen Prozessor, wie der CPU, stellt diese Aufgabe keine besondere Herausforderung dar. Es müssen lediglich alle Werte eines Datenfeldes sequentiell von Anfang bis Ende durchlaufen und in einer globalen Variablen akkumuliert werden. Nach dem Durchlaufen der Schleife kann das arithmetische Mittel durch eine einfache Division der Akkumulatorvariablen mit der Anzahl der Elemente berechnet werden.
Für das parallele Programmiermodell der GPU ist der sequentielle Algorithmus jedoch nicht geeignet. Das liegt vor allem daran, dass aktueller Graphikhardware ein spezielles Hardwareregister zur Akkumulation von Daten fehlt [CDPS03].
Es existiert jedoch ein Algorithmus, mit dem die Graphikhardware effizient genutzt werden kann, um aus einem Vektor oder einer Matrix von Eingangswerten einen einzelnen Ergebniswert zu berechnen. Dazu wird der Algorithmus der *parallelen Reduktion* verwendet. Hierbei werden mehrere Datenelemente der Matrix schrittweise über einen Shader, der die Rechenoperation kapselt, zusammengefasst und in eine neue verkleinerte Matrix kopiert. Somit findet eine schrittweise Reduktion der ursprünglichen Datenmenge statt, wobei der Prozess so lange fortgesetzt wird, bis ein einzelner Wert übrig bleibt.
Damit der Algorithmus von der Graphikhardware ausgeführt werden kann, müssen zunächst alle zu reduzierenden Datenelemente in eine quadratische Textur der Größe $2^N x 2^N$ kopiert werden. Die Textur wird in jedem Renderdurchlauf sukzessiv in X- und Y-Richtung halbiert, wobei in jedem Schritt jeweils vier Texturelemente (Texel) über einen Fragmentshader zu einem neuen Element zusammengefasst und in eine Render-Textur geschrieben werden. Der Reduktionsvorgang benötigt daher zwei Texturen, die alternierend für Lese- und Schreibzugriffe verwendet werden [11]. Der komplette Vorgang ist in insgesamt $\log_2 N$ Renderdurchläufen abgeschlossen. Dabei werden maximal $4/3N^2$ Texturzugriffe benötigt [CDPS03]. In Abbildung 10 ist der Vorgang der parallelen Reduktion am Beispiel einer Maximabestimmung schematisch dargestellt.
Mit dem Verfahren der parallelen Reduktion können einzelne Werte recht effizient aus einer großen Wertemenge heraus bestimmt werden. Bei einer Speicherung der Eingangswerte in einer 2D-Textur und den heute üblichen Texturauflösungen von maximal $4096x4096$ [Ver05] in einem RGBA-Fließkommaformat sind maximal $4 \cdot 4096^2 = 67108864$ Eingangswerte möglich. Dabei können die Anforderungen

[11] Dieses Verfahren ist auch als Ping-Pong-Rendering bekannt [Ver04]

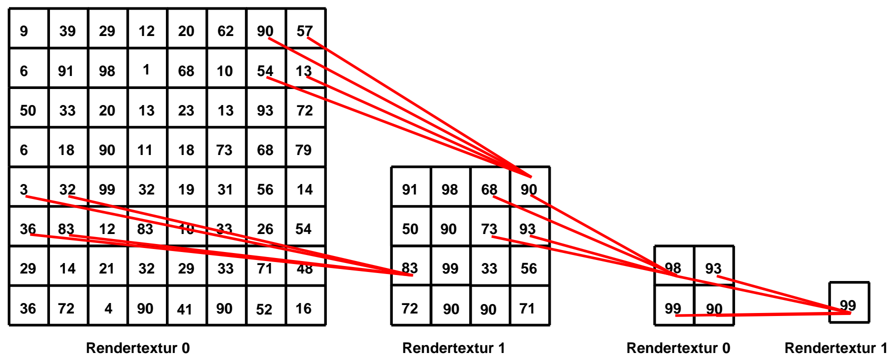



**Abbildung 10:** Parallele Reduktion zur Bestimmung des größten Wertes in einer 8x8 Matrix

an den Texturspeicher und die Bandbreite bei großen Eingangsmengen im Fließkommaformat jedoch beachtlich sein [CDPS03].
Das Verfahren der parallelen Reduktion wurde im Rahmen dieser Arbeit eingesetzt, um die minimale, maximale und durchschnittliche logarithmische Leuchtdichte aus einer Matrix von Leuchtdichten zu bestimmen. In Kapitel 8.1.1 ist die Leistung der parallelen Reduktion im Vergleich zu einer traditionellen Berechnung auf der CPU aufgeführt.

## 5.3 Effiziente Konvolution von Bilddaten

Viele Algorithmen der Bildverarbeitung verwenden eine Konvolution von Bilddaten. Dazu wird jeder Pixel des Bildes abhängig von seinen Pixelnachbarn und mit den Gewichten einer speziellen Filtermaske gefaltet. Die Pixelnachbarschaften und Filtermasken können dabei recht groß werden, was wiederum eine Vielzahl von Lesezugriffen erfordert. Eine Umsetzung auf dem Fragmentprozessor bietet sich an, da die Algorithmen zur Bildkonvolution im Kern viele Gather-Operationen verwenden.
Die Bilddaten werden zunächst in einer Textur gespeichert, die als Eingabeparameter für einen Fragmentshader dient. Die Daten sollten dabei möglichst in einem Fließkommaformat mit ausreichender Genauigkeit abgelegt werden, sodass keine numerischen Überläufe entstehen und Werte abgeschnitten werden. Dabei kann eine mehrkomponentige Textur erhebliche Anforderungen an Speicherplatz und Speicherbandbreite stellen, wie die Leistungsanalysen in Kapitel 8.1.2 zeigen.
Zusätzlich zu den Bilddaten benötigt der Fragmentshader eine Reihe von Filtergewichten, die für die Faltung genutzt werden. Für eine Bereitstellung der Filtergewichte existieren hierbei mehrere Möglichkeiten. So können die Gewichte vor jedem Renderdurchgang vom Hauptprogramm zum Fragmentprozessor kopiert werden. Im Rahmen dieser Arbeit hat sich jedoch herausgestellt, dass eine Schleife im Fragmentshader über ein Datenfeld, das als Parameter im Fragmentshader ver-

wendet wird, mit Problemen verbunden ist. Diese Beobachtung wurde auch von Markus Fahlén in [Fah05] gemacht und ist mit großer Wahrscheinlichkeit auf eine fehlende Unterstützung seitens der OpenGL Shading Language zurückzuführen. Eine weitere Möglichkeit, die Filtergewichte als Parameter zu übergeben, kann durch eine vorherige Speicherung in einer zusätzlichen Textur erfolgen. Die Textur kann dann neben den eigentlichen Bilddaten ebenfalls als Eingabeparameter verwendet werden, wobei es in diesem Fall innerhalb des Fragmentshaders notwendig ist, während der Faltung für jeden Nachbarschaftspixel eine entsprechende Texturkoordinate für das korrespondierende Filtergewicht zu berechnen. Zudem wird mit dieser Vorgehensweise ein weiterer Texturzugriff zum Lesen des Filtergewichts benötigt, wodurch die Berechnungsgeschwindigkeit der Konvolution weiter verringert wird. Eine andere Möglichkeit ist die direkte Speicherung der Filtergewichte als Konstanten im Fragmentshader. Diese Lösung bietet weniger Flexibilität, da der Fragmentshader nun eine feste diskrete Filtergröße verwenden muss. Im Rahmen dieser Arbeit hat sich diese Vorgehensweise jedoch als schnellste und einfachste Lösung etabliert. Dies liegt nicht zuletzt daran, dass der Compiler bei der Verwendung von Konstanten in den meisten Fällen in der Lage ist, den Fragmentshader bei der Übersetzung weiter zu optimieren [12]. Dazu wurde während der Implementationsphase eine Klasse geschrieben, die auf der Basis einiger Parameter, wie zum Beispiel dem diskreten Filterradius und der Standardabweichung, dynamisch einen entsprechenden Shader mit konstanten Filtergewichten generiert. Dadurch konnte ein Teil der Flexibilität zurückgewonnen werden, wobei gleichzeitig eine hohe Leistung möglich war.

Bevor der Fragmentshader die Faltung durchführen kann, müssen zunächst Fragmente generiert werden. Dazu wird ein mit den Eingabedaten texturiertes, bildschirmfüllendes Rechteck [13] so gezeichnet, dass jede Texelposition der Bilddaten genau auf eine korrespondierende Fragmentposition der Render-Textur abgebildet wird, welche die Ergebnisse der Konvolution speichert. Diese Eins-zu-Eins Abbildung kann leicht über eine einfache orthographische Projektion realisiert werden, die sicherstellt, dass der Fragmentshader für jedes einzelne Texel der Eingabetextur während des Rasterisierungsvorgangs ausgeführt wird. Der Vorgang ist vom Standpunkt der sequentiellen Berechnung aus analog zu einer doppelten `FOR`-Schleife in X- und Y-Richtung über die Bilddaten zu sehen. Der Fragmentshader stellt den Berechnungskern der Faltung dar und ist nun in der Lage, für jedes Pixel der Eingabetextur eine feste Pixelnachbarschaft einzulesen und mit den Filtergewichten zu multiplizieren. Das Ergebnis wird zum Abschluss normiert, indem die aufsummierten und gewichteten Nachbarschaftswerte durch die Summe der Filtergewichte geteilt und in die Render-Textur geschrieben werden.

Je nach Größe der Bilddaten und Filtermasken ist im Kontext der Graphikhardwareprogrammierung eine unterschiedliche Anzahl von Texturzugriffen notwendig. In der Regel sind ohne weitere Optimierungen für eine quadratische $NxN$

[12] Eine mögliche Optimierung ist hierbei durch *Loop Unrolling* gegeben

[13] OpenGL Primitiventyp: GL_QUADS

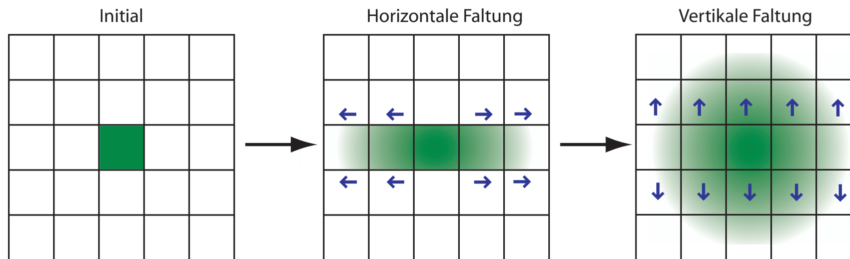



**Abbildung 11:** Separierbare 2D-Faltung

Filtermaske $N^2$ Texturzugriffe pro Fragment erforderlich. Steigt die Anzahl der Texturzugriffe, werden die Anforderungen an die Speicherbandbreite der Graphikhardware ebenfalls größer, wobei die maximale Füllrate bei großen Filtermasken deutlich reduziert werden kann [Ver04]. Damit ein Algorithmus zur Konvolution möglichst effizient von der Graphikhardware ausgeführt wird, sollte eine Minimierung der notwendigen Texturzugriffe einen ersten Ansatz zur Optimierung darstellen.
Um die Anzahl der Texturzugriffe zu reduzieren, kann die lineare Abhängigkeit einiger Filterkerne ausgenutzt werden. So sind beispielsweise Filterkerne mit einer Gaußschen-Verteilung separierbar [RWPD06]. Mathematisch lässt sich eine separierbare 2D-Funktion als Produkt zweier 1D-Funktionen schreiben. Dieser Zusammenhang ist in Formel (15) und (16) dargestellt:

$$G(x,y) = G_x(x)G_y(y) \tag{15}$$

$$\frac{1}{\pi s^2}\exp^{\left(-\frac{x^2+y^2}{s^2}\right)} = \frac{1}{\pi s^2}\exp^{\left(-\frac{x^2}{s^2}\right)} \cdot \frac{1}{\pi s^2}\exp^{\left(-\frac{y^2}{s^2}\right)} \tag{16}$$

Praktisch entspricht damit eine 1D-Faltung in horizontaler Richtung, gefolgt von einer 1D-Faltung in vertikaler Richtung, der kompletten 2D-Faltung. Die Konvolution der Bilddaten kann demnach in zwei aufeinanderfolgenden Renderdurchgängen, einmal für die horizontale und einmal für die vertikale Richtung, durchgeführt werden. Dieser Vorgang ist in Abbildung 11 schematisch visualisiert. Dadurch lässt sich die Anzahl der Texturzugriffe bei einer $NxN$ Filtermaske effektiv auf $2N$-Texturzugriffe verringern. Bezogen auf die Texturzugriffe entspricht dies einer Reduktion eines quadratischen auf einen linearen Aufwand, wodurch eine erhebliche Verbesserung der Leistung erzielt wird. Allerdings gilt die Einschränkung, dass diese Optimierung nur für separierbare Filterkerne anwendbar ist.

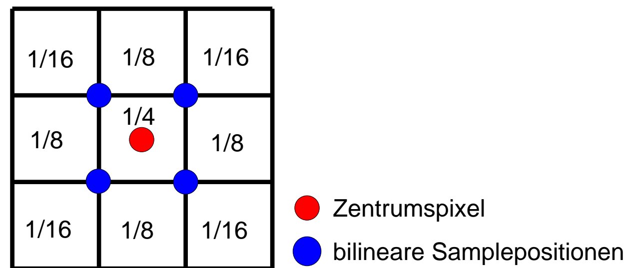



**Abbildung 12:** Ein 3x3 Binomialfilter mit vier bilinearen Texturzugriffen

Durch die Ausnutzung der bilinearen Texturfilterung der Graphikhardware können bei der Konvolution unter Umständen weitere Texturzugriffe eingespart werden. So nutzt Masaki Kawase [Mas03] die bilineare Filterung effektiv aus, um einen 3x3 Binomialfilter mit lediglich vier Texturzugriffen in nur einem Renderdurchlauf zu berechnen. Dazu macht er sich die lineare Abhängigkeit eines 3x3 Binomialfilterkerns aus Formel (17) zu nutze:

$$B^2 = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{17}$$

Damit lässt sich die Filtermaske nach Formel (18) wie folgt umschreiben [Fah05]:

$$B^2 = \frac{1}{4}\left(\frac{1}{4}\begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \frac{1}{4}\begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} + \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} + \frac{1}{4}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix}\right) \tag{18}$$

Durch Texturzugriffe an Texelkreuzungen und unter Verwendung der bilinearen Texturfilterung können die Texel entsprechend gemittelt werden. Abbildung 12 zeigt die Texturzugriffe an den entsprechenden Texturkreuzungen sowie die späteren Gewichtungen auf Basis der Binomialverteilung. Der Filter von Kawase kann zudem iterativ angewendet werden, um stärkere Bildglättungen durch größere Filterkerne herbeizuführen [Mas03]. Dabei werden die Samplepositionen in jedem Schritt etwas weiter nach außen verschoben. Dieser Vorgang ist schematisch in Abbildung 13 dargestellt. Der Binomialfilter von Kawase aus [Mas03] ist äußerst effizient und ist besonders für den Einsatz in Computerspielen geeignet [Car05].
Die bilineare Texturfilterung der Graphikhardware kann auch für andere separierbare Filterkerne verwendet werden. In dem Buch GPU Gems 2 [Ver05] wird ein Verfahren beschrieben, mit dem Konvolutionen allgemein als Summe von mehreren linearen Interpolationen berechnet werden können. Die Filtergewichte der Filterkerne sowie die Samplepositionen für die Texturzugriffe müssen dazu angepasst werden. Mit dieser Optimierung ist es möglich, die Anzahl der Texturzugriffe

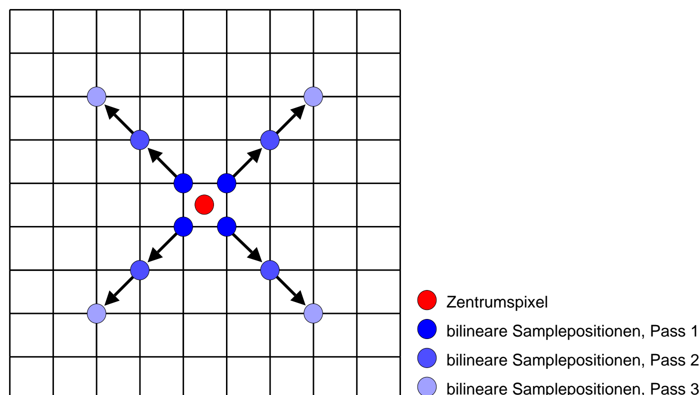



**Abbildung 13:** Iterative Anwendung des 3x3 Binomialfilters von Kawase

von $2N$ nochmals auf $(n+1)\ mod\ 2 \approx n$ zu reduzieren [Fah05].
Leider ist die Verwendung der bilinearen Filterung der Graphikhardware auf bestimmte Texturformate beschränkt. Dies trifft insbesondere bei Fließkommaformaten zu, wodurch eine Verwendung dieser Optimierungen im Rahmen der Arbeit stark eingeschränkt ist. Ein bilinearer Filter lässt sich ohne weiteres über einen Fragmentshader realisieren. Dazu sind jedoch vier weitere Texturzugriffe pro Texel notwendig, die den Geschwindigkeitsvorteil schnell zunichte machen.
Eine weitere Möglichkeit zur Optimierung ergibt sich durch die Verwendung einer Approximation für größere Filtermasken. In [KMS05] wird von Krawczyk et al. vorgeschlagen die Bilddaten zunächst herunterzuskalieren. Die eigentliche Faltung erfolgt dann auf den kleineren Bildern und erfordert deshalb wesentlich weniger Texturzugriffe. Abschließend wird das Bild wieder vergrößert. Bei der Vergrößerung wird eine bilineare Filterung verwendet, um eine genauere Approximation zu erhalten. Krawczyk et al. implementieren dazu die bilineare Filterung für nicht-unterstützte Texturformate in einem Fragmentshader, der zur abschließenden Skalierung eingesetzt wird [KMS05]. Eine ähnliche Vorgehensweise wird von Masaki Kawase in [Mas04] genutzt, um Blendeffekte durch helle Lichtquellen zu visualisieren.
Für diese Arbeit wurden einige der zuvor vorgestellten Algorithmen zur Konvolution von Bilddaten implementiert. In Kapitel 8.1.2 sind dazu einige Leistungsmessungen aufgeführt.

# 6 Ausgewählte Tone-Mapping-Verfahren

In diesem Kapitel werden verschiedene Tone-Mapping-Verfahren auf Basis der Aufgabenstellung untersucht. Dabei wurden drei Verfahren im Detail betrachtet, die jeweils auf unterschiedlichen Modellvorstellungen beruhen.

## 6.1 Photorezeptor-Tone-Mapping

In [RD05] beschreiben Reinhard et al. ein globales Tone-Mapping-Verfahren, das auf einem wahrnehmungsbasierten Modell aufsetzt. Das Modell von Reinhard versucht hierbei jedoch nicht das komplette visuelle System, sondern vielmehr die erste Stufe der visuellen Verarbeitung, die der Photorezeptoren, nachzubilden [RD05]. Im Folgenden werden die einzelnen Teilprozesse des Verfahrens im Detail beschrieben.

### 6.1.1 Adaptionsmodell für Photorezeptoren

Elektrophysiologische Untersuchungen belegen, dass eine visuelle Adaption bereits in den frühen Stufen der menschlichen visuellen Wahrnehmung erfolgt. Die Grundlage des Tone-Mapping-Verfahrens von Reinhard bildet daher ein vereinfachtes Adaptionsmodell für die Photorezeptoren.
Die Photorezeptoren sind in der Lage Signale in Form von elektrochemischen Potentialen an Neuronen weiterzuleiten [Sch98]. Empirische Untersuchungen zeigen, dass gemessene Potentialstärken beschränkt sind. Die Funktion aus Formel (19) beschreibt das Potential $V$, das die Zapfen bei einer einfallenden Lichtintensität $I$ produzieren [RD05]:

$$V = \frac{I}{I + \sigma(I_{adp})} V_{max} \tag{19}$$

Reinhards Verfahren nutzt das Potential $V$ im Kontext eines Tone Mappers als skalierten Wert für die Darstellung auf einem Ausgabegerät. Die Formel (19) bildet daher die Grundlage für den Tone-Mapping-Operator [RWPD06]. Die Funktion $\sigma(I_{adp})$ aus Formel (19) beschreibt den Adaptionsvorgang der Photorezeptoren und hängt von der Adaptionsgröße $I_{adp}$ ab. Diese Größe beschreibt den aktuellen Adaptionszustand eines Photorezeptors und ist von der aktuellen Lichtintensität abhängig [RD05]. Der Wert von $V_{max}$ ist ein globaler Skalierungsfaktor und gibt das maximale Potential an, das durch eine Lichtintensität im Photorezeptor erreicht werden kann. Für ein typisches Ausgabegerät setzt Reinhard $V_{max} = 1$, sodass die resultierenden Werte von $V$ in einem Wertebereich von $[0, 1]$ liegen. Weiterhin verwendet Reinhard für die Funktion $\sigma(I_{adp})$ die folgende Formel (20) [RD05]:

$$\sigma(I_{adp}) = (f I_{adp})^m \tag{20}$$

Die beiden Konstanten $f$ und $m$ sind frei wählbar, wobei im nächsten Kapitel einige Vorschläge für die Wertebereiche der Parameter gemacht werden, mit denen das Verfahren in den meisten Fällen zu plausiblen Ergebnissen führt [RD05].

### 6.1.2 Benutzerparameter

Das Tone-Mapping-Verfahren von Reinhard verwendet insgesamt vier Benutzerparameter, mit denen die Gesamthelligkeit, der Kontrast und die Adaptionsvorgänge für den skotopischen und photopischen Adaptionsbereich unabhängig voneinander gesteuert werden können. Die einzelnen Parameter sind in einer Übersicht in Tabelle 1 aufgeführt [RD05]:

| **Parameter** | **Beschreibung** | **Initialer Wert** | **Wertebereich** |
|---|---|---|---|
| $m$ | Kontrast | $0.3 + 0.7k^{1.4}$ | $[0.3, 1.0]$ |
| $f$ | Helligkeit | 0.0 | $[-8.0, 8.0]$ |
| $c$ | Chromatische Adaption | 0.0 | $[0.0, 1.0]$ |
| $a$ | Helligkeitsadaption | 1.0 | $[0.0, 1.0]$ |

**Tabelle 1:** Die verschiedenen Parameter des Verfahrens

Über den Parameter $m$ kann der globale Kontrast im Ergebnisbild modifiziert werden. Dabei kann es für bestimmte Anwendungen durchaus sinnvoll sein, die verschiedenen Parameter des Tone-Mapping-Operators automatisch zu berechnen und nicht für jedes Einzelbild manuell zu setzen [RD05]. Dies ist besonders bei dem im Rahmen dieser Arbeit entstandenen adaptiven Tone-Mapping-Verfahren von Bedeutung. Für eine automatische Berechnung des Parameters $m$ verwendet Reinhard daher folgende Formel (21) [RD05, RWPD06]:

$$m = 0.3 + 0.7k^{1.4} \tag{21}$$

Die Berechnung des Kontrastparameters $m$ erfordert zusätzlich eine Konstante $k$, die ein Maß für die Gesamthelligkeit des Bildes darstellt und damit analog zum Szenen-Key des photographischen Tone Mappers aus Kapitel 6.3 zu sehen ist. Nach Reinhard kann ein Wert für $k$ aus bildabhängigen Größen abgeschätzt und durch Formel (22) berechnet werden [RD05]:

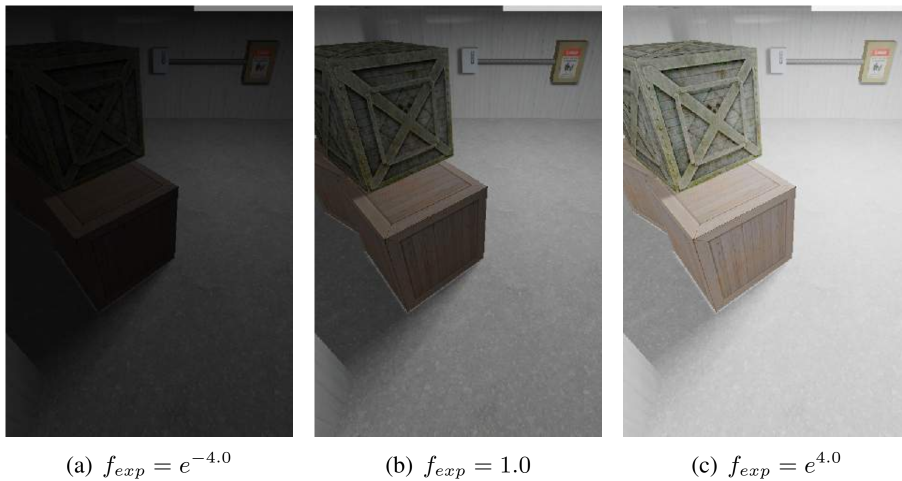
$$k = \frac{\log(L_{i,max}) - L_{i,avg}}{\log(L_{i,max}) - \log(L_{i,min})} \tag{22}$$

In dieser Formel gibt $L_{i,min}$ die minimale, $L_{i,max}$ die maximale und $L_{i,avg}$ die durchschnittliche logarithmische Leuchtdichte des Bildes aus Kapitel 2.1.2 an.

Reinhard empfiehlt in [RD05] den Wert $m$ aus Formel (21) auf einen Wertebereich von $[0.3, 1.0]$ zu begrenzen. Dies entspricht in etwa den Wertebereichen, die von elektrophysiologischen Studien zur Funktion der Photorezeptoren belegt sind [RWPD06].
Über den Faktor $f$ kann die Gesamthelligkeit des Ergebnisbildes gesteuert werden. Nach Reinhard können eine Vielzahl an Werten des Parameters $f$ zu plausiblen Ergebnissen führen, wobei auch hier explizit ein Wertebereich von $[-8.0, 8.0]$ für $f$ vorgeschlagen wird [RD05]. Eine automatische Berechnung ist nicht vorgesehen. Reinhard modifiziert vielmehr den Paramter $f$ vor der eigentlichen Verwendung durch die einfache Exponentialfunktion in Formel (23):

$$f_{exp} = e^{-f} \tag{23}$$

Kleinere Werte für $f_{exp}$ führen zu dunkleren Ergebnisbilder, während größere Werte ein helleres Gesamtergebnis erzeugen [RD05]. In Abbildung 14 sind dazu einige Ergebnisbilder nach Anwendung des Tone Mappers für verschiedene Werte von $f_{exp}$ dargestellt.
In der Literatur wird die adaptierte Größe $I_{adp}$ aus Formel (19) häufig auf den Wert



(a) $f_{exp} = e^{-4.0}$ (b) $f_{exp} = 1.0$ (c) $f_{exp} = e^{4.0}$

**Abbildung 14:** Ergebnisbilder für verschiedene Werte des Parameters $f_{exp}$

der durchschnittlichen logarithmischen Leuchtdichte $L_{i,avg}$ des Bildes aus Formel (4) gesetzt. In [RWPD06] und [RD05] schlägt Reinhard allerdings eine andere Vorgehensweise zur Berechnung vor, die zu einem späteren Zeitpunkt zusätzliche Modifikationen für die chromatische Adaption und die Helligkeitsadaption ermöglichen. So können starke Farbstiche einzelner Pixel im Bild beispielsweise durch eine lokale Farbkorrektur nach Formel (24) reduziert werden [RD05]:

$$I_{adp}(x, y) = cRGB_i(x, y) + (1 - c)L_i(x, y) \tag{24}$$

Mit der Interpolationsgröße $c$ kann linear zwischen der Leuchtdichte eines Pixels $L_i(x, y)$ und dem RGB-Wert des Pixels $RGB_i(x, y)$ interpoliert werden. Für $c = 0$ findet keine Farbkorrektur statt.
Die Adaptionsgröße $I_{adp}(x, y)$ ist nach Reinhard zudem von der aktuellen und der vergangenen Lichtintensität abhängig, die der Photorezeptor ausgesetzt war. Da das optische System beim Betrachten eines Bildes viele verschiedene Punkte zufällig abtastet, nimmt Reinhard an, dass die aktuelle Adaptionsgröße $I_{adp}(x, y)$ als Funktion lokaler und globaler Intensitäten ausgedrückt werden kann. Als globale Größe verwendet Reinhard das arithmetische Mittel aller RGB-Werte des Bildes, wobei die Berechnung der Adaptionsgröße $I_{adp}(x, y)$ durch Formel (25) erfolgt [RWPD06]:

$$I_{adp}(x, y) = aRGB_i(x, y) + (1 - a)RGB_{i,avg} \tag{25}$$

Mit der Interpolationsgröße $a$ kann linear zwischen dem RGB-Wert eines Pixels $RGB_i(x, y)$ und dem globalen durchschnittlichen RGB-Wert $RGB_{i,avg}$ des Bildes interpoliert werden.
Die Formeln (24) und (25) zur Berechnung der lokalen Adaptionsgröße $I_{adp}(x, y)$ können über die Interpolation in den Formeln (26), (27) und (28) kombiniert werden [RD05]:

$$I_{adp,local}(x, y) = cRGB_i(x, y) + (1 - c)L_i(x, y) \tag{26}$$

$$I_{adp,global} = cRGB_{i,avg} + (1 - c)L_{i,avg} \tag{27}$$

$$I_{adp}(x, y) = aI_{adp,local} + (1 - a)I_{adp,global} \tag{28}$$

Über die Interpolationsgröße $a$ lässt sich festlegen, wie stark der Einfluss der lokalen und globalen Komponenten $I_{adp,local}$ und $_{adp,global}$ auf die Berechnung der Adaptionsgröße $I_{adp}$ ausfällt. Dabei kann durch eine Modifikation des Parameters $a$ bei manchen Bildern eine Kontrastverbesserung erzielt werden. Nachdem die Adaptionsgröße $I_{adp}(x, y)$ durch die obige Formel berechnet wurde, kann die eigentliche Kompression der Bilddaten vorgenommen werden.

### 6.1.3 Kompression der Bilddaten

Durch die Anwendung von Formel (19) lässt sich mit Hilfe der Adaptionsgröße $I_{adp}(x, y)$ für jeden Pixel ein Wert für das Photorezeptorpotential bestimmen. Daraus lassen sich die skalierten Pixelwerte für das Ausgabegerät $RGB_d(x, y)$ durch Formel (29) berechnen:

$$RGB_d(x, y) = \frac{1}{L_{i,max} - L_{i,min}} \left( \frac{RGB_i(x, y)}{RGB_i(x, y) + (f_{exp} I_{adp}(x, y))^m} - L_{i,min} \right) \tag{29}$$

In der Formel wird implizit eine Normierung vorgenommen, sodass die resultierenden RGB-Farbwerte in einem Intervall von $[0, 1]$ liegen [RD05].

### 6.1.4 Fazit

Das globale Tone-Mapping-Verfahren von Reinhard et al. wurde im Rahmen dieser Arbeit implementiert. Dabei konnte eine Verlagerung vieler Berechnungen auf die GPU vorgenommen werden, wodurch eine hohe Leistung möglich war. So lassen sich die Werte für die durchschnittliche logarithmische Leuchtdichte $L_{i,avg}$ und die arithmetischen Mittelwerte der Rot-, Grün- und Blaukanäle $RGB_{i,avg}$ effizient über den Algorithmus der parallelen Reduktion aus Kapitel 5.2 bestimmen. Weiterhin verwendet Reinhards Verfahren im Kern viele lineare Vektorinterpolationen pro Pixel, die ebenfalls sehr effizient von der Graphikhardware berechnet werden können.
In dem Originalpaper „Dynamic Range Reduction inspired by Photoreceptor Physiology“ beschreiben Reinhard et al., wie ihr Verfahren angepasst werden könnte, damit eine temporäre Adaption möglich wird. Dazu schlägt Reinhard vor, die Adaptionsgröße $I_{adp}$ über eine Bildfolge hinweg durch eine geeignete Mittelung zu berechnen. Leider konnten keine Ansätze gefunden werden, durch die das bereits vorgestellte Verfahren von Reinhard für die Simulation der anderen Phänomene der menschlichen visuellen Wahrnehmung modifiziert werden kann.

## 6.2 Histogrammbasiertes Tone Mapping

Die *Histogram Equalization* ist ein bekanntes statistisches Verfahren aus dem Bereich der Bildverarbeitung, wobei das Histogramm eines digitalen Grauwertbildes zur Kontrastverbesserung des Originalbildes genutzt wird [Reh00].
In [LRP97] wenden Ward et al. eine ähnliche Technik im Kontext eines globalen Tone-Mapping-Operators an. Neben der Histogram Equalization werden weitere Verfahren zur Histogrammanpassung eingesetzt. Dabei wird das kumulative Histogramm als Skalierungsfunktion zur Abbildung der Leuchtdichten auf den darstellbaren Bereich des Ausgabegeräts verwendet. Im Kern verwendet das Verfahren somit einen globalen Tone-Mapping-Operator.
Im Folgenden werden die einzelnen Teilprozesse des Verfahrens von Ward et al. aus [LRP97] detailliert behandelt. In dem Originalpaper „A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes“ werden außerdem einige Techniken zur Simulation verschiedener Phänomene der menschlichen visuellen Wahrnehmung beschrieben. Diese Techniken wurden im Kontext dieser Arbeit nicht verwendet und werden daher nicht weiter erläutert.

### 6.2.1 Erzeugung des „Fovea-Bildes“

Zunächst sieht Ward die Erzeugung eines sogenannten *Fovea-Bildes* vor. Dieses Bild repräsentiert spezielle Fixpunkte der Sehgrube des Auges [LRP97]. Die Bildgröße wird dabei so gewählt, dass jeder Pixel des Fovea-Bildes in etwa einem Grad

des Gesichtsfeldes des Betrachters entspricht. Ward berechnet hierbei die Breite $F_{width}$ und Höhe $F_{height}$ des Bildes durch Formel (30) und (31) wie folgt:

$$F_{width} = 2.0 \tan(0.5 \cdot \theta_h) / 0.01745 \tag{30}$$
$$F_{height} = 2.0 \tan(0.5 \cdot \theta_v) / 0.01745 \tag{31}$$

Der konstante Wert $0.01745$ entspricht hierbei einem Grad im Bogenmaß. Zusätzlich müssen die Werte für den horizontalen und vertikalen Öffnungswinkel $\theta_h$ und $\theta_v$ der Projektion bekannt sein, die zur Aufnahme oder Berechnung des Originalbildes verwendet wurden. Nachdem die Bildgröße für das Fovea-Bild berechnet worden ist, können die Leuchtdichten des Originalbildes hineinkopiert werden.

### 6.2.2 Erstellung des Histogramms

Von dem Fovea-Bild kann nun ein Histogramm erzeugt werden. Das Histogramm dient der Repräsentation von Häufigkeitsverteilungen einzelner Leuchtdichten. Ward verwendet eine logarithmische Leuchtdichtenskala, wobei als obere und untere Histogrammgrenze die kleinste beziehungsweise größte Leuchtdichte $L_{i,min}$ und $L_{i,max}$ des Fovea-Bildes genutzt werden [LRP97]. Damit der Wert für $L_{i,min}$ nicht zu klein gewählt wird, schlägt Ward einen Minimalwert von $10^{-4}$ vor.
Die Häufigkeitsverteilungen der Leuchtdichten werden in mehreren *Bins* gleichmäßig auf das logarithmische Leuchtdichtenspektrum verteilt. Ward nutzt hierbei 100 Bins, um eine ausreichend akkurate Darstellung der Leuchtdichteverteilung zu erhalten [LRP97]. Für ein Histogramm mit $N$-Bins errechnet sich die Größe eines einzelnen Bins $\Delta b$ durch Formel (32):

$$\Delta b = \frac{(\log(L_{i,max}) - \log(L_{i,min}))}{N} \tag{32}$$

Nach der Erzeugung des Histogramms können die eigentlichen Berechnungen zur Histogram Equalization erfolgen.

### 6.2.3 Erzeugung des kumulativen Histogramms

Für die Histogram Equalization muss zunächst das *kumulative Histogramm* bestimmt werden. Das kumulative Histogramm $H_{cum}(b)$ enthält zu jedem Bin $b$ die relative Summenhäufigkeit des Bildhistogramms $H(b)$ und kann durch Formel (33) berechnet werden [LRP97, Reh00]:

$$H_{cum}(b) = \frac{1}{T} \sum_{b_i < b} H(b_i) \tag{33}$$

Hierbei gibt $T$ die Gesamtanzahl aller Elemente des Histogramms $H(b_i)$ an und

kann durch Formel (34) berechnet werden:

$$T = \sum_{b_i} H(b_i) \tag{34}$$

Durch eine Skalierung der einzelnen Leuchtdichten des Bildes $L_i(x, y)$ mit dem kumulativen Histogramm $H_{cum}(b)$, ist eine Gleichverteilung der Leuchtdichten zu erreichen. Nach der Skalierung sind alle Leuchtdichten im Bild mit derselben Häufigkeit vertreten, wodurch eine Kontrastverbesserung erzielt wird [LOPR97]. Unter Verwendung der Histogram Equalization, lässt sich bereits ein erster Tone-Mapping-Operator entwickeln.

### 6.2.4 Naiver Tone-Mapping-Operator

Nach Ward kann das kumulative Histogramm $H_{cum}(b)$ in einem ersten Ansatz zur Skalierung der Leuchtdichten des Bildes $L_i(x, y)$ genutzt werden. Die auf dem jeweiligen Ausgabegerät darstellbaren komprimierten Leuchtdichten $L_d(x, y)$ lassen sich durch Formel (35) berechnen [LRP97]:

$$\log(L_d(x, y)) = \log(L_{d,min}) + (\log(L_{d,max}) - \log(L_{d,min})) \cdot H_{cum}(\log(L_i(x, y))) \tag{35}$$

Dabei sind die minimal und maximal darstellbaren Leuchtdichten des Ausgabegerätes durch $L_{d,min}$ und $L_{d,max}$ gegeben.
Die Anwendung dieses Operators kann jedoch zu sehr starken Kontrastverhältnissen im Ergebnisbild führen, die nicht dem natürlichen Kontrasteindruck des Originalbildes entsprechen [LRP97]. Durch die Gleichverteilung der Grauwerte werden einige Bereiche nicht etwa komprimiert, sondern teilweise expandiert. Es sollten jedoch gerade die Leuchtdichten komprimiert werden, die in einer großen Häufigkeit vorkommen und einen Peak im Histogramm erzeugen [RWPD06]. Dieser Problematik nimmt sich Ward an, indem eine schrittweise Verbesserung des Verfahrens vorgenommen wird.

### 6.2.5 Histogrammanpassung an einen linearen Schwellwert

Ward macht zunächst die Beobachtung, dass ein Tone-Mapping-Operator mit einer einfachen linearen Skalierung keine übertrieben starke Kontrastverhältnisse verursacht und bei vielen Bildern zu einem zufriedenstellendem Ergebnis führt [LRP97]. Daher schlägt Ward vor, das Bildhistogramm $H(b)$ so zu modifizieren, dass bei einer späteren Kompression der Leuchtdichten keine Kontrastverhältnisse entstehen können, die eine lineare Skalierung übersteigen [LRP97]. Dabei wird durch Formel (36) folgende Einschränkung geltend gemacht:

$$\frac{dL_d(x, y)}{dL_i(x, y)} \leq \frac{L_d}{L_i} \tag{36}$$

Damit das Kontrastverhältnis nicht über dem einer linearen Skalierung liegt, muss der Quotient aus den abgeleiteten skalierten Leuchtdichten $L_d(x,y)$ und der Ableitung der Leuchtdichten im Originalbild $dL_i(x,y)$ kleiner oder gleich dem Quotienten aus skalierter Leuchtdichte $L_d(x,y)$ und den Originalleuchtdichten $L_i(x,y)$ des Bildes sein.
Damit aus der Ungleichung in (36) ein entsprechender Schwellwert gewonnen werden kann, muss zunächst eine Ableitung für $dL_d(x,y)$ gefunden werden. Dazu wird die Funktion des Tone-Mapping-Operators aus Formel (35) abgeleitet. Zuvor muss jedoch die Ableitung für das kumulative Histogramm $H_{cum}(b)$ bestimmt werden. Dabei ist das kumulative Histogramm $H_{cum}(b)$ als numerische Integration des Bildhistogramms zu sehen. Die Ableitung kann demnach aus der Funktion des Bildhistogramms $H(b)$ und durch einen geeigneten Normalisierungsfaktor berechnet werden. Dieser Zusammenhang ist in Formel (37) dargestellt [LRP97]:

$$\frac{dH_{cum}(b)}{db} = \frac{H(b)}{T\Delta b} \tag{37}$$

Zur Normalisierung verwendet Ward die Gesamtzahl aller Elemente des Histogramms $T$ aus Formel 34 und die Größe der Bins $\Delta b$, die durch Formel (32) berechnet werden kann. Unter Anwendung der Kettenregel und durch das Einsetzen von Formel (35) in Formel (36) kann folgende Ungleichung (38) aufgestellt werden:

$$\exp\left(\log(L_d(x,y))\right) \cdot \frac{H(\log(L_i(x,y)))}{T\Delta b} \cdot \frac{(\log(L_{d,max}) - \log(L_{d,min}))}{L_i(x,y)} \leq \frac{L_d(x,y)}{L_i(x,y)} \tag{38}$$

Durch weitere Umformungen und Vereinfachungen wird die Ungleichung in Formel (39) überführt:

$$H(b) \leq \frac{T\Delta b}{(\log(L_{d,max}) - \log(L)_{d,min})} \tag{39}$$

Solange die Elementanzahl aller Bins des Histogramms $H(b)$ unterhalb des Schwellwertes liegt, der durch die obige Ungleichung angegeben wird, können nach einer Skalierung der Leuchtdichten durch Formel (35) in dem Ergebnisbild keine Kontrastverhältnisse enstehen, die über denen einer lineare Skalierung liegen.
Dazu muss das Histogramm des Bildes angepasst werden, wobei Ward vorschlägt, diejenigen Bins auf den Schwellwert zu setzen, deren Elementanzahl oberhalb des Wertes liegt [LRP97]. Eine Reduzierung der Elementanzahl eines Bins, verändert jedoch gleichzeitig die Gesamtanzahl aller Elemente des Histogramms. Dabei wird der Wert $T$ wiederum für eine Berechnung des Schwellwertes in Formel (41) benötigt [RWPD06]. Um dieses nicht-lineare Problem zu lösen, nutzt Ward einen iterativen Algorithmus, der die Gesamtzahl $T$ und den Schwellwert in jedem Schritt

neu berechnet und so lange eine Kürzung enstprechender Histogrammbins vornimmt, bis ein vorher festgelegtes Toleranzkriterium erreicht wird [IFM05].
Nachdem das Histogramm angepasst wurde, wird das kumulative Histogramm berechnet und in dem Tone-Mapping-Operator aus Formel (35) verwendet.

### 6.2.6 Histogrammanpassung an die menschliche Kontrastempfindung

Das lineare Schwellwertverfahren aus dem vorherigen Kapitel setzt vorraus, dass die menschliche visuelle Wahrnehmung Kontrasteindrücke gleichmäßig über das gesamte Spektrum der Leuchtdichten wahrnimmt. Dies entspricht jedoch nicht dem natürlichen Verhalten [RWPD06]. Dabei ist gerade im skotopischen Bereich keine kontrastreiche Wahrnehmung möglich.
Aus diesem Grund entwickelt Ward ein weiteres Verfahren zur Histogrammanpassung, welches ebenfalls eine Trimmung einzelner Histogrammbins vorsieht. Diesmal wird jedoch keine einfache lineare Schwelle genutzt, sondern Schwellwerte, die aus einem empirischen Modell für die natürliche Kontrastempfindung der menschlichen visuellen Wahrnehmung stammen [LRP97].
Dazu verwendet Ward eine Funktion $\Delta L_t(L_a)$, die zu einer Leuchtdichte $L_a$ die kleinste Schwelle angibt, die für eine Unterscheidung zweier Leuchtdichten in diesem Adaptionszustand überschritten werden muss. Die Funktion ist durch eine stückweise Approximation einiger empirischer Daten zu Kontrastuntersuchungen gegeben und wird durch Formel (40) beschrieben [LRP97, FPSG96]:

$$\Delta L_t(L_a) = \begin{cases} -2.86 \Leftrightarrow \log_{10}(L_a) < -3.94 \\ (0.405 \log_{10}(L_a) + 1.6)^{2.18} - 2.86 \Leftrightarrow -3.94 \leq \log_{10}(L_a) < -1.44 \\ \log_{10}(L_a) - 0.395 \Leftrightarrow -1.44 \leq \log_{10}(L_a) < -0.0184 \\ (0.294 \log_{10}(L_a) + 0.65)^{2.7} - 0.72 \Leftrightarrow -0.0184 \leq \log_{10}(L_a) < 1.9 \\ \log_{10}(L_a) - 1.255 \Leftrightarrow \log_{10}(L_a) \geq 1.9 \end{cases} \tag{40}$$

Mit Hilfe der Funktion aus (40) lässt sich die Ungleichung aus Formel (41) angeben:

$$\frac{dL_d(x,y)}{dL_i(x,y)} \leq \frac{\Delta L_t(L_d(x,y))}{\Delta L_t(L_i(x,y))} \tag{41}$$

Die Einschränkung der obigen Ungleichung besagt, dass zwei Leuchtdichten im Originalbild, die als nicht-unterschiedlich wahrgenommen werden, im Ergebnisbild auf dieselbe Leuchtdichte abgebildet werden [IFM05]. Die Ungleichung lässt sich analog zu der vorherigen Vorgehensweise auflösen, wobei sich der Schwellwert für die maximal erlaubte Binanzahl durch Formel (42) ergibt:

$$H(b) \leq \frac{T \Delta b L_i(x,y)}{(\log(L_{d,max}) - \log(L_{d,min}))\, L_d(x,y)} \cdot \frac{\Delta L_t(L_d(x,y))}{\Delta L_t(L_i(x,y))} \tag{42}$$

Hierbei werden ebenfalls Bins gekürzt, deren Elementanzahl über dem Schwellwert liegen [LRP97]. Dabei kommt erneut der iterative Algorithmus, der die Gesamtanzahl der Histogrammelemente in jedem Schritt neu berechnet, in einer leicht modifizierten Version zum Einsatz [LRP97]. Nach der Histogrammanpassung, kann das kumulative Histogramm ermittelt und in dem Tone-Mapping-Operator aus Formel (35) verwendet werden.

### 6.2.7 Fazit

Das zuvor beschriebene Tone-Mapping-Verfahren von Ward et al. wurde im Rahmen dieser Arbeit teilweise implementiert. Zu einem späteren Zeitpunkt wurde davon abgesehen das Verfahren als Grundlage für einen Echzeit-Tone-Mapper zu nutzen.
Die Implementation beinhaltet den ersten Algorithmus zur Histogrammanpassung, wobei einige Teile der Berechnung auf die GPU ausgelagert werden konnten. Insbesondere konnte die Berechnung der Histogramme auf der GPU effizient realisiert werden. Gleichzeitig war jedoch die Umsetzung vieler Teilprozesse des Verfahrens nicht für das Programmiermodell der GPU geeignet. So mussten die Berechnungen zur Erzeugung des kumulativen Histogramms und der Algorithmus des iterativen Verfahrens zur ersten Histogrammanpassung komplett auf der CPU durchgeführt werden. Diese einfachen Algorithmen konnten allerdings auch auf der CPU ausreichend schnell ausgeführt werden und brachten keine großen Leistungseinbußen mit sich. Eine Ausnahme bildete der iterative Algorithmus zur zweiten Histogrammanpassung aus Kapitel 6.2.6, der die Modifikation des Histogramms auf Basis der empirischen Daten des menschlichen Kontrastempfindens vornimmt. Dieser Algorithmus benötigt direkten Zugriff auf die Leuchtdichten des Bildes, die jedoch aus Gründen der Effizienz zusammen mit den Originalbilddaten in einem dynamischen Texturobjekt direkt im Graphikspeicher abgelegt sind. Für die Berechnung der Histogrammanpassung auf der CPU müssten die Daten vorher komplett zurückgelesen werden, wodurch der Graphikbus zwischen Graphikhardware und CPU stark belastet wird. Ob mit dieser Vorgehensweise eine Echtzeitanwendung überhaupt möglich ist, kann durchaus angezweifelt werden.
Das zuvor beschriebene Tone-Mapping-Verfahren ist zunächst nur für die Kompression von Leuchtdichten einzelner Bilder geeignet und nicht für eine Simulation der menschlichen visuellen Adaption im Rahmen einer Echtzeit 3D-Umgebung. Bei der Anwendung des Verfahrens konnte beobachtet werden, dass die Histogrammgrenzen in dynamischen Szenen mit wechselnden Leuchtdichteverteilungen stark varieren können. Das kumulative Histogramm, das im Prinzip die Skalierungsfunktion zur Abbildung der Leuchtdichten auf das Ausgabegerät darstellt, kann unter solchen Bedingungen für zwei aufeinanderfolgende Bilder ebenfalls stark unterschiedlich sein. Dadurch können Diskontinuitäten bei der späteren Darstellung der Bildfolge auftreten. Der iterative Algorithmus zur Modifikation des Histogramms kann ebenfalls zu solchen Diskontinuitäten führen [IFM05]. Dieser

Problematik haben sich Irawan et al. in [IFM05] angenommen, indem eine modifizierte Version von Wards Verfahren vorgestellt wird, mit der ein adaptives Tone Mapping ohne Diskontinuitäten möglich ist. Im Kern verwendet Irawan dazu ebenfalls einen Algorithmus zur Histogrammanpassung, wobei er jedoch nach der Trimmung einzelner Histogrammbins eine Neuverteilung der abgeschnittenen Werte auf die verbleibenden Bins vornimmt. Zusätzlich verwendet Irawan ein komplexes Modell der temporären Adaption, das die feste Schwellwertfunktion von Ward für die Verwendung in einer dynamischen Umgebung mit wechselnden Leuchtdichteverhältnissen anpasst. Das Verfahren von Irawan wurde im Rahmen dieser Arbeit jedoch nicht evaluiert, da es lediglich als Offline-Verfahren in einer MATLAB-Implementation existiert [IFM05]. Hierbei ergibt sich ebenfalls die Problematik, dass viele der von Irawan verwendeten Algorithmen nicht effizient auf der GPU umgesetzt werden können.

## 6.3 Tone Mapping auf Basis eines photographischen Modells

In [RWPD06] und [RSSF02] beschreiben Reinhard et al. einen lokalen Tone Mapper, der auf einem photographischen Modell basiert. Die meisten Bildträger können nur ein begrenztes Spektrum an Leuchtdichten darstellen [RWPD06]. Die Komprimierung von Leuchtdichten und die damit verbundenen Problematiken haben ihren Ursprung im Bereich der Phototechnik [LRP97].
Als Grundlage für das Modell von Reinhard dient das *Zonensystem* von Ansel Adams [RSSF02]. Dieses System sieht eine Unterteilung des gesamten Leuchtdichtespektrums einer Bildszene auf eine feste Anzahl von Druckzonen vor. Dazu sind in der Regel elf Zonen definiert, die in der Literatur mit 0 bis X durchnummeriert werden. Zone 0 steht dabei für Schwarz und Zone X für Weiß. Jede nachfolgende Zone hat die doppelte Intensität der vorherigen [RSSF02]. Dabei fasst jede Zone einen Teil des gesamten Leuchtdichtespektrums zusammen [Fah05]. Durch eine direkte Abbildung der Leuchtdichten auf die einzelnen Zonen werden mehrere Leuchtdichten auf einen einzelnen Wert gesetzt. Dadurch findet effektiv eine Kompression der Leuchtdichten statt. In Abbildung 15 ist dieser Zusammenhang graphisch dargestellt. Durch diese einfache Art der Kompression können einige Bildbereiche im Ergebnisbild jedoch kontrastarm erscheinen. Besonders bei Bildern, die über eine hohe Dynamik in der Leuchtdichteverteilung verfügen, werden viele Werte entweder auf Zone 0 (Schwarz) oder Zone X (Weiß) abgebildet. Um den Kontrasteindruck des Bildes nachträglich zu verbessern, ist ein weiterer Arbeitsschritt notwendig. Dabei werden einzelne Bildbereiche wieder aufgehellt oder abgedunkelt. Dieser Prozess wird in der Phototechnik häufig während der Entwicklung vorgenommen und ist unter dem Begriff des „Dodging and Burning“ bekannt [RSSF02].
Analog zum Zonenmodell und der Technik des „Dodging and Burning“ beschreiben Reinhard et al. in [RWPD06] und [RSSF02] ein Verfahren, mit dem eine Kompression der Leuchtdichten bei digitalen Bildern durchgeführt werden kann. Im Folgenden werden die einzelnen Teilprozesse des Verfahrens beschrieben.

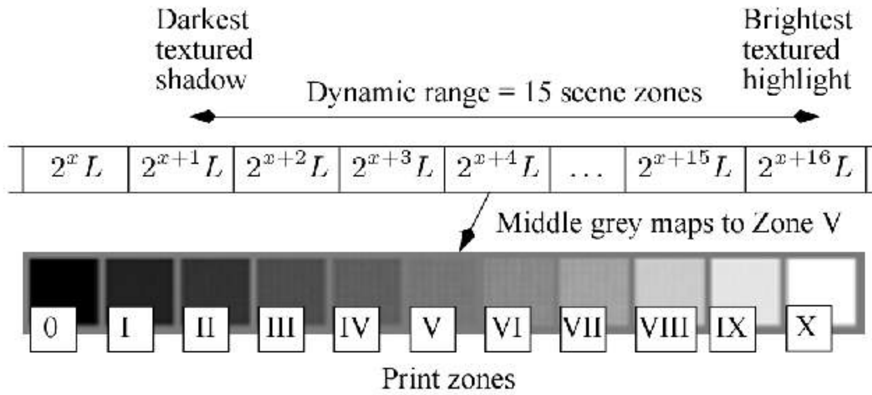



**Abbildung 15:** Das Zonensystem von Ansel Adams
*Quelle: Reinhard et al. [RSSF02]*

### 6.3.1 Lineare Skalierung

Zunächst wird eine lineare Skalierung der Leuchtdichten $L_i(x, y)$ des Bildes durch Formel (43) vorgenommen. Dies würde im Bereich der Photographie dem Einstellen der Belichtungszeit einer Kamera entsprechen [RWPD06]. Dazu nutzt Reinhard die durchschnittliche logarithmische Leuchtdichte $L_{i,avg}$ die bereits aus Formel (4) bekannt ist und berechnet die skalierten Werte durch Formel (43):

$$\overline{L_i}(x, y) = \frac{\alpha}{L_{i,avg}} L_i(x, y) \tag{43}$$

Die Skalierung zur Bestimmung der relativen Leuchtdichten $\overline{L_i}(x, y)$ aus Formel (43) verwendet einen weiteren Parameter $\alpha$, der die Gesamthelligkeit des Ergebnisbildes steuert. Für den Parameter $\alpha$ werden häufig Werte zwischen $[0, 1]$ gewählt [RJIH04]. In [Rei02] stellt Reinhard einen Ansatz vor, mit dem sich Werte für $\alpha$ aus bildabhängigen Größen abschätzen lassen. Dazu werden in Formel (44) und (45) verschiedene Werte für $\alpha$ aus der maximalen Leuchtdichte $L_{i,max}$, der minimalen Leuchtdichte $L_{i,min}$ und der durchschnittlichen logarithmischen Leuchtdichte $L_{i,avg}$ des Bildes berechnet:

$$k = \left( \frac{2log_2 L_{i,avg} - log_2 L_{i,min} - log_2 L_{i,max}}{log_2 L_{i,max} - log_2 L_{i,min}} \right) \tag{44}$$

$$\alpha = 0.18 \cdot 4^k \tag{45}$$

Es ensteht der Vorteil, dass der Tone-Mapping-Prozess weitgehend automatisierbar bleibt und der Parameter $\alpha$ nicht manuell gesetzt werden muss. Dies ist im Kontext

dieser Arbeit besonders wichtig, da das hier vorgestellte Tone-Mapping-Verfahren adaptiv und für ganze Bildfolgen arbeiten soll. Zur dynamischen Berechnung des Parameters $\alpha$ existieren in der Literatur weitere Ansätze [KMS05, RJIH04], die speziell für interaktive Umgebungen mit stark wechselnden Leuchtdichtespektren geeignet sind.

### 6.3.2 Globaler Operator

Nach der Berechnung der relativen Leuchtdichten ist es möglich die Leuchtdichten $L_i(x, y)$ zu komprimieren. Dies würde effektiv einem globalen Tone-Mapping-Operator entsprechen. Dazu bietet Reinhard eine einfache Funktion an, welche die relativen Leuchtdichten $\overline{L_i}(x, y)$ des Bildes auf einen Wertebereich zwischen $[0, 1[$ abbildet. Die Operatorfunktion aus Formel (46) skaliert kleine Leuchtdichtewerte linear, während größere Werte stärker komprimiert werden [RWPD06]:

$$L_d(x, y) = \frac{\overline{L_i}(x, y)}{\overline{L_i}(x, y) + 1.0} \tag{46}$$

In Abbildung 16 sind mehrere Graphen für den globale Tone-Mapping-Operator für verschiedene Werte von $\alpha$ dargestellt. Hierbei lässt sich erkennen, dass die Modifikation des Parameters $\alpha$ effektiv eine Verschiebung der Graphen in X-Richtung bewirkt, wobei das Bild insgesamt heller oder dunkler dargestellt wird. Da in dem Originalbild keine beliebig großen Leuchtdichten enthalten sind, gilt für die skalierten Leuchtdichten $L_d(x, y) < 1$. Daher bietet Reinhard alternativ eine zweite Funktion für einen globalen Operator an, der einen Burn-Out-Effekt von sehr hellen Bildbereichen erzielen kann [RWPD06]. Dazu benötigt man jedoch einen weiteren Parameter $L_{i,white}$, der die kleinste Leuchtdichte angibt, auf die Weiß (1.0) abgebildet werden soll. Die vollständige Operatorfunktion ist in Formel (47) dargestellt:

$$L_d(x, y) = \frac{\overline{L_i}(x, y) \left(1.0 + \frac{\overline{L_i}(x,y)}{L_{i,white}^2}\right)}{1.0 + \overline{L_i}(x, y)} \tag{47}$$

Nach Reinhard kann diese Operatorfunktion zu besseren Ergebnissen führen als die Funktion aus (46). Dies gilt insbesondere bei Bildern, die eine niedrige Dynamik in ihrem Leuchtdichtespektrum aufweisen und für die die Bedingung $L_{i,max} < 1$ gilt [RSSF02]. Normalerweise wird der Parameter $L_{i,white}$ auf den Wert der maximalen Leuchtdichte des Bildes $L_{i,max}$ gesetzt. Eine Abschätzung des Parameters aus bildabhängigen Größen ist ebenfalls möglich [RWPD06]. Hierfür nutzt Reinhard die folgende Formel (48) [Rei02]:

$$L_{i,white} = 1.5 \cdot 2.0 \left(log_2 L_{i,max} - log_2 L_{i,min} - 5.0\right) \tag{48}$$

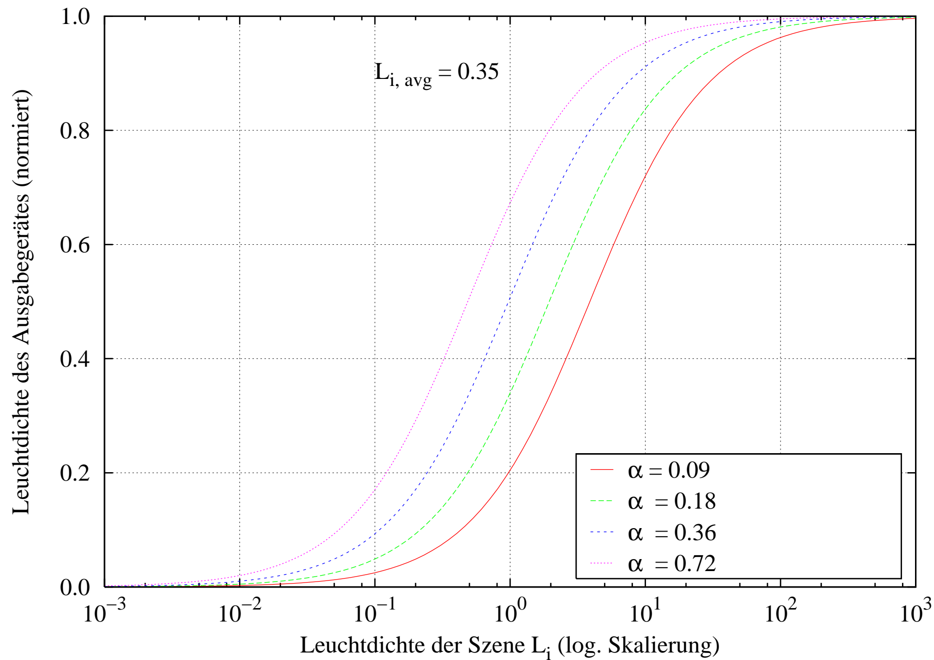



**Abbildung 16:** Funktionsgraphen für den globalen Operator für verschiedene Werte für $\alpha$

Beide Operatoren haben jedoch den Nachteil, dass in Szenen mit einer hohen Beleuchtungsdynamik Kontrast verloren gehen kann. Daher nutzt Reinhard einen lokalen Tone-Mapping-Operator der den Gesamtkontrast des Bildes verbessert.

### 6.3.3 Selektives Aufhellen und Abdunkeln

In Anlehnung an die photographische Technik des „Dodging and Burning“ wird von Reinhard et al. in [RWPD06] und [RSSF02] ein Algorithmus vorgestellt, mit dem einzelne Bildbereiche selektiv aufgehellt oder abgedunkelt werden können. Das Verfahren arbeitet direkt auf den relativen Leuchtdichten $\overline{L_i}(x, y)$ des Bildes. Der Algorithmus versucht, für jeden Pixel im Bild der relativen Leuchtdichten $\overline{L_i}(x, y)$ eine umschließende homogene Region zu finden, deren Größe maximal ist und in der keine großen Kontrastunterschiede vorliegen [RWPD06]. Diese Regionen können schließlich als lokale Mittelwerte in einem Tone-Mapping-Operator verwendet werden, um eine Kontrastverbesserung herbeizuführen.
Dafür muss das Bild der relativen Leuchtdichten $\overline{L_i}(x, y)$ zunächst mit einem festen Satz an Gaussfiltern $G_j(x, y, s)$ gefaltet werden. Hierbei werden die Filtergrößen $s$ sukzessiv vergrößert. Reinhard verwendet für die Faltung spezielle radialsymmetrische Gaussprofile aus Formel (49):

$$G_j(x,y,s) = \frac{1.0}{\pi(\gamma_j s)^2} \exp\left(-\frac{x^2+y^2}{(\gamma_j s)^2}\right) \tag{49}$$

Reinhard wählt als initialen Wert für den Skalierungsfaktor $\gamma_1 = 1/2\sqrt{2} \approx 0.35$. Die verbleibenden Skalierungsfaktoren werden durch $\gamma_{i+1} = 1.6\gamma_{i+1}$ inkrementiert, wobei nach Reinhard für manche HDR-Bilder kleine Änderungen an den Werten vorgenommen werden können. Insgesamt verwendet Reinhard in [RSSF02] acht diskrete Profilgrößen für $G_j(x,y,s)$, wobei jedes Profil um den Faktor 1.6 vergrößert wird. Daraus ergeben sich acht verschiedene Filtermasken mit diskreten Größen von 1x1 bis 43x43 [RSSF02].
Das gaussgewichtete Bild der relativen Leuchtdichten $\overline{L}_{g,s}(x,y)$ aus Formel (50) ergibt sich durch eine Faltung von $\overline{L}_i(x,y)$ mit dem jeweiligen Gaussprofil $G(x,y,s)$ aus Formel (49). Das Ergebnis der ersten Faltung $L_{g,1.0}$ entspricht hierbei dem Bild der relativen Leuchtdichten $\overline{L}_i(x,y)$.

$$\overline{L}_{g,s}(x,y) = \overline{L}_i(x,y) \otimes G_j(x,y,s) \tag{50}$$

Die einzelnen gesuchten Regionen dürfen allerdings nicht zu groß gewählt werden, da es sonst zu den bekannten Halo-Artefakten kommt [RSSF02]. Deshalb wird in jedem Schritt ein Maß für den Unterschied der Kontrastverhältnisse zwischen der vorherigen und der neuen, größeren Region ermittelt. Hierfür verwendet Reinhard die Funktion aus Formel (51):

$$V(x,y,s) = \frac{\overline{L}_{g,s}(x,y) - \overline{L}_{g,s+1}(x,y)}{2^{\phi}\alpha/s^2 + \overline{L}_{g,s}(x,y)} \tag{51}$$

Die Funktion gehört der Klasse der sogenannten „Center-Surround-Funktionen“ an. Diese Funktionen werden häufig zur Berechnung von Kontrastunterschieden zweier Bildregionen verwendet [RSSF02]. Dabei wird ein Maß für den Kontrastunterschied zwischen einer kleineren Region (Center) und einer größeren, umschließenden Region (Surround) berechnet. Die Regionen stehen repräsentativ für die gewichteten Pixelnachbarschaften unterschiedlicher Größen $s$ nach der Faltung durch Formel (50). Durch die Skalierung $2^{\phi}\alpha/s^2 + \overline{L}_{g,s}(x,y)$ im Zähler von Formel (51) wird eine Normierung vorgenommen. Dies ist zu einem späteren Zeitpunkt wichtig, um einen absoluten Schwellwert für alle Profilgrößen $s$ nutzen zu können [RWPD06]. Der Parameter $\phi$ aus Formel (51) kann dabei verwendet werden, um im Ergebnisbild kleinere Kontrastverbesserungen herbeizuführen. Dabei sollten jedoch für $\phi$ nicht zu große Werte gewählt werden, da dies ebenfalls zu Halo-Artefakten führen kann. Reinhard empfiehlt den Paramter auf $\phi = 8.0$ zu setzen [RWPD06].
Um festzustellen, ob sich Center- und Surround-Regionen stark in ihrem Kontrastverhältnis unterscheiden, werden die Ergebnisse der Center-Surround-Berechnung

**Abbildung 17:** Auswahl einer Center-Region
*Quelle Originalbild: Greg Ward*

aus Formel (51) mit einem Schwellwert $C_{max}$ verglichen. Dazu verwendet Reinhard Formel (52):

$$|V(x, y, s)| < C_{max} \tag{52}$$

Als Schwellwert gibt Reinhard einen Wert von $C_{max} = 0.05$ an [RSSF02]. Liegen die Werte der „Center-Surround-Funktion“ aus Formel (51) über dem Schwellwert $C_{max}$, ist ein größerer Unterschied in den Kontrastverhältnissen beider Regionen vorhanden. In diesem Fall enthält die kleinere Region (Center) einen homogenen Bildbereich mit der maximalen Größe $s$, der keine starken Kontrastunterschiede aufweist. Die Region kann als lokales Mittel für den jeweiligen Zentrumspixel verwendet werden. Die größere Region (Surround) enthält bereits Diskontinuitäten, die etwa durch Licht- oder Schattenkanten und damit stark wechselnde Leuchtdichten gegeben sind.

Liegt der Wert aus Formel (52) jedoch unter dem Schwellwert $C_{max}$, so ist es eventuell möglich die Center-Region weiter zu vergrößern. Wenn zu kleine Regionen als lokales Mittel genutzt werden, kann es im Ergebnisbild und nach Anwendung des lokalen Tone-Mapping-Operators lokale Kontrastverluste geben. Das Ziel

(a) Relative Leuchtdichten (b) Gemittelte lokale Leuchtdichten

**Abbildung 18:** Relative Leuchtdichten und lokal gemittelte Leuchtdichten
*Quelle Originalbild: Begleit-DVD [RWPD06]*

ist für jeden Pixel eine Center-Region zu finden, deren Größe $s$ maximal ist und die eine homogene Verteilung von Leuchtdichten ohne große Kontraständerungen aufweist. In Abbildung 17 ist eine Auswahl verschiedener Center-Regionen exemplarisch dargestellt.
Mit den maximalen Center-Regionen für jedes Pixel im Bild der relativen Leuchtdichten $\overline{L_i}(x, y)$ sind gleichzeitig die zugehörigen lokalen Mittelwerte $\overline{L_m}(x, y)$ bekannt. In Abbildung 18 ist neben dem Bild der relativen Leuchtdichten $\overline{L_i}(x, y)$ ein Bild mit den dazugehörigen lokalen Mittelwerten dargestellt. Sind die lokalen Mittelwerte erst einmal berechnet, kann ein lokaler Tone-Mapping-Operator angewendet werden.

### 6.3.4 Lokaler Operator

Durch eine einfache Modifikation kann der bereits bekannte globale Tone-Mapping-Operator aus Formel (46) in einen lokalen Operator überführt werden. Reinhard verwendet in [RWPD06] und [RSSF02] die folgende Formel (53):

$$L_d(x, y) = \frac{\overline{L_i}(x, y)}{\overline{L_m}(x, y) + 1.0} \tag{53}$$

Bei Szenen mit einer hohen Dynamik der Leuchtdichteverteilung erzeugt der lokale

(a) Globaler Operator (b) Lokaler Operator

**Abbildung 19:** Visueller Vergleich zwischen dem lokalen und globalen Operator
*Quelle Originalbild: Begleit-DVD [RWPD06]*

Operator von Reinhard kontrastreichere Bilder. In der Detailvergrößerung in Abbildung 19 sind exemplarisch zwei Ausschnitte von Ergebnisbildern des einfachen globalen Operators aus Formel (46) und des lokalen Operators aus Formel (53) dargestellt. Zur Berechnung der relativen Leuchtdichten $\overline{L}_i(x, y)$ aus Formel (43) wurde für den Szenen-Key der Wert $\alpha = 0.38$ genutzt. Der lokale Operator verwendete die von Reinhard in [RSSF02] empfohlenen Parameter.

### 6.3.5 Fazit

Im Rahmen dieser Arbeit wurde der zuvor beschriebene photographische Tone Mapper von Reinhard et al. implementiert. Dabei konnten große Teile zur Berechnung auf die GPU ausgelagert werden, sodass eine vergleichsweise hohe Leistung zu den Angaben der CPU-basierten Umsetzung aus [RSSF02] möglich war.
Es hat sich herausgestellt, dass der globale Operator von Reinhard effizient zu berechnen ist und sich damit für eine Echtzeitanwendung im Sinne der Aufgabenstellung eignet. Dabei lässt sich der globale Operator auch mit anderen Tone-Mapping-Verfahren kombinieren. So nutzen Barladian et al. in [BVGK04] ein hybrides Verfahren, das den Tone Mapper von Tumblin-Rushmeier aus [Tum99] mit der globalen Operatorfunktion aus Formel (47) von Reinhard kombiniert.
Reinhards lokaler Operator benötigt wesentlich größere Leistungsressourcen, was

vor allem durch die Bildkonvolutionen mit großen Filtermasken zu erklären ist. Daher empfiehlt Reinhard zunächst die relativen Leuchtdichten $\overline{L_i}(x,y)$ und die Filterkerne der Gaussprofile aus Formel (54) mit Hilfe einer Fouriertransformation in den Frequenzraum zu überführen. Die eigentliche Faltung erfordert dann lediglich eine einfache Multiplikation pro Bildpunkt [RSSF02]. Von dieser Vorgehensweise wurde im Rahmen dieser Arbeit abgesehen, da die Berechnungsgeschwindigkeit der Fouriertransformation auf der GPU zum gegenwärtigen Zeitpunkt nicht ausreicht, um eine Echtzeitanwendung mit höheren Bildauflösungen zu ermöglichen [MA03].
Es hat sich gezeigt, dass die Berechnungsgeschwindigkeit des lokalen Operators bei einer Umsetzung auf der GPU deutlich höher ist als bei der CPU-basierten Implementation von Reinhard et al. aus [RSSF02]. Hierbei konnten vor allem die Berechnungen zur Konvolution mit Gaussfiltern von einer Implementation auf dem Fragmentprozessor der Graphikhardware profitieren. Um die Leistung weiter zu optimieren, wurde die Separierbarkeit der Gaussfiltermasken ausgenutzt, sodass insgesamt weniger Texturzugriffe notwendig waren. In Kapitel 8.1.2 werden dazu einige Leistungsanalysen präsentiert.
Für das im Rahmen dieser Arbeit vorgestellte Tone-Mapping-Verfahren wurde der globale Operator von Reinhard auf der GPU realisiert. Als lokaler Operator kam jedoch ein anderes Verfahren zum Einsatz, das eine Echtzeitanwendung auch bei höheren Bildauflösungen ermöglicht. An dieser Stelle sollte erwähnt werden, dass es in der Literatur durchaus Arbeiten gibt, die den lokalen Operator von Reinhard für eine Echtzeitanwendung optimiert haben [KMS05, GWWH03].
Das zuvor beschriebene Tone-Mapping-Verfahren ist zunächst nur für einzelne Bilder geeignet und nicht für eine Simulation der menschlichen visuellen Adaption in einer Echtzeit 3D-Umgebung vorgesehen. In der Literatur existieren dazu jedoch einige Ansätze, die das Verfahren von Reinhard modifizieren, sodass ein adaptives Tone Mapping möglich ist [KMS05, GWWH03, RJIH04]. Für das hier umgesetzte adaptive Verfahren waren ebenfalls einige Modifkationen und Erweiterungen des Originalverfahrens notwendig, die in Kapitel 7 ausführlich vorgestellt werden.
Der photographische Tone Mapper von Reinhard et al. ist durch Krawczyk et al. in [KMS05] bereits für die Simulation der Phänomene zur Sehschärfe aus Kapitel 3.3, der Blendeffekte aus Kapitel 3.4 und für den Verlust der Farbwahrnehmung aus Kapitel 3.5 angepasst worden. Daher bot es sich an, das Verfahren von Reinhard als Grundlage für das adaptive Tone-Mapping-Verfahren im Rahmen dieser Arbeit zu nutzen.

# 7 Adaptives Tone Mapping auf der Graphikhardware

Ziel dieser Arbeit ist die Umsetzung eines Tone-Mapping-Operators auf der Graphikhardware. Dabei sollte der Tone-Mapping-Operator echtzeitfähig sein und die adaptive menschliche visuelle Wahrnehmung simulieren.
Nach Evaluation der verschiedenen Tone Mapper aus Kapitel 6 hinsichtlich dieser Zielsetzung fiel die Wahl auf eine modifizierte Version des Tone-Mapping-Verfahrens von Reinhard et al. aus Kapitel 6.3, das auf einem photographischen Modell basiert. Der globale Operator dieses Verfahrens lässt sich relativ einfach auf der GPU realisieren und kann, wie viele andere globale Operatoren auch, durchaus als echtzeitfähig bezeichnet werden. Dabei wurde zusätzlich ein Algorithmus für einen lokalen Operator implementiert, der jedoch nicht mit dem lokalen Operator von Reinhard verwandt ist. Weiterhin ist das im Rahmen dieser Arbeit umgesetzte Tone-Mapping-Verfahren für eine Simulation der menschlichen visuellen Adaption erweitert worden.
Im Verlauf dieser Arbeit wurde ebenfalls begonnen, den histogrammbasierten Tone Mapper von Ward et al. aus Kapitel 6.2 umzusetzen. Nach einigen technischen Durchstichen wurde jedoch davon abgesehen die Umsetzung weiterzuführen. Als nützlichen Nebeneffekt konnte zumindest die Klasse `GPUHistogram` zum Erzeugen von Histogrammen durch die GPU für die spätere Testumgebung weiter verwendet werden. Der in Kapitel 6.1 vorgestellte Photorezeptor-Tone-Mapper wurde ebenfalls nicht für eine Implementation vorgesehen.
Das folgende Kapitel beschreibt die konzeptionelle Planung und Implementation des im Rahmen dieser Arbeit entstandenen Tone-Mapping-Verfahrens. Es folgt zunächst eine Auflistung von Anforderungen, die an das Verfahren gestellt werden und für die weitere Planung hilfreich waren.

## 7.1 Anforderungen

Für die konzeptionelle Planung und Realisierung des Tone Mappers ist es sinnvoll, einige Anforderungen natürlich sprachlich und auf Basis der Aufgabenstellung zu formulieren. Dadurch können die verschiedenen Teilaufgaben des Systems später leichter konkretisiert und weiter verfeinert werden.
Das hier umgesetzte Tone-Mapping-Verfahren sieht einen *globalen* und einen *lokalen* Operator vor. Der globale Operator wird auf Basis des photographischen Modells von Reinhard et al. aus Kapitel 6.3 umgesetzt. Der lokale Operator verwendet ein eigenes Verfahren, das die Echtzeitfähigkeit bei einer Bildschirmauflösung von maximal 1024x768 Pixeln erhält. Als echtzeitfähig ist hierbei eine Bildwiederholrate von mindestens 20 Hz gemeint.
Das Tone-Mapping-Verfahren wird um die Simulation einiger ausgewählter Teile der menschlichen visuellen Adaption ergänzt. Diese Teile umfassen:

- Temporäre Adaption
- Verlust der Sehschärfe

- Blendeffekte durch Streulicht
- Verlust der Farbwahrnehmung

Die einzelnen Teile lassen sich selektiv an- und abschalten und sind über verschiedene Parameter steuerbar. Bei der kompletten Simulation aller Teile bleibt die Echtzeitfähigkeit weiter erhalten.
Das gesamte Design ist modular ausgelegt, sodass eine einfache Erweiterbarkeit gegeben ist. Hierbei wird die Implementation des Tone-Mapping-Verfahrens durch ein bereits bestehendes Framework unterstützt. Viele berechnungsintensive Algorithmen werden auf Basis der programmierbaren Graphikhardware implementiert, wobei insbesondere neue Features der Hardware zum Einsatz kommen.
Weiterhin wird eine Testumgebung mit einer graphischen Umgebung (GUI) realisiert. Die Umgebung ermöglicht eine einfache und intuitive Evaluation des gesamten Verfahrens. Dabei soll es dem Anwender möglich sein in einer 3D-Umgebung zu navigieren, wobei verschiedene Szenarien eingeladen werden können. Die 3D-Umgebungen sind voll texturiert und nutzen statische HDR-Lightmaps, sodass ein adaptives Tone Mapping sinnvoll ist.
Die Implementation erfolgt in der Programmiersprache C und C++, wobei die Shader für die Graphikhardware in der OpenGL Shading Language realisiert werden. Darüber hinaus ist der gesamte Quelltext sinnvoll und ausreichend kommentiert.

## 7.2 Entwurf und Design

Auf Basis der vorangegangenen Anforderungen lässt sich ein Entwurf ausarbeiten. Im Folgenden werden die relevanten Teile des Verfahrens spezifiziert, wobei bereits eine stärkere technische Sichtweise gewählt wird. Zudem sind die wichtigsten Designentscheidungen näher erläutert.

### 7.2.1 Globaler Tone Mapper

Zunächst wird der globale Tone-Mapping-Operator von Reinhard et al. aus Kapitel 6.3 implementiert. Dazu müssen die relativen Leuchtdichten $\overline{L}_i(x, y)$ aus den Leuchtdichten $L_i(x, y)$ des Bildes bestimmt werden. Für die notwendige lineare Skalierung nach Formel (43) muss vorab die durchschnittliche logarithmische Leuchtdichte $L_{i,avg}$ aus Formel (4) bestimmt werden. Diese bildabhängige Größe kann durch eine parallele Reduktion auf der GPU gewonnen werden. Das Verfahren wurde bereits in Kapitel 5.2 vorgestellt. Vor der eigentlichen Reduktion müssen die logarithmischen Leuchtdichten in einem separaten Renderdurchlauf ermittelt werden. Die logarithmischen Leuchtdichten werden jeweils als eine Komponente in einer RGB-Textur abgespeichert. Die verbleibenden Komponenten der Textur speichern zusätzlich die Leuchtdichten $L_i(x, y)$ des Originalbildes. Die Textur dient dann als Eingabe für die nachfolgende parallele Reduktion. Dabei werden neben der durchschnittlichen logarithmischen Leuchtdichte $L_{i,avg}$ zusätzlich die minimale und maximale Leuchtdichte des Bildes $L_{i,min}$ und $L_{i,max}$ berechnet.

Diese Werte können später genutzt werden, um den Szenen-Key $\alpha$ und den Wert von $L_{i,white}$ nach Formel (45) und (48) automatisch zu berechnen.
Nach der Reduktion werden die relativen Leuchtdichten $\overline{L}_i(x, y)$ in einem weiteren Renderdurchlauf durch einen Shader berechnet und in eine Render-Textur geschrieben. Dabei werden die Leuchtdichten des Bildes $L_i(x, y)$, der Key $\alpha$ und die durchschnittliche logarithmische Leuchtdichte $L_{i,avg}$ als Eingabeparameter verwendet.
Zu diesem Zeitpunkt ist es bereits möglich den globalen Tone-Mapping-Operator nach Reinhard et al. aus Kapitel 6.3 umzusetzen. Nach den Anforderungen wird zusätzlich ein lokaler Operator implementiert, der im Folgenden näher beschrieben ist.

### 7.2.2 Ergänzung für einen lokalen Operator

Damit es bei den lokalen Tone-Mapping-Operatoren nicht zu den typischen Halo-Artefakten kommt, sollten die lokalen Mittelwerte der einzelnen Pixel nicht über starke Kanten hinweg berechnet werden. Dies kann durch eine selektive Bildglättung erreicht werden. Dabei werden homogene Bildregionen stärker geglättet, während Kanten weitgehend erhalten bleiben. Im Kontext des Tone Mappings werden dazu häufig bilaterale Filter eingesetzt, die eine solche Filterung der Bilddaten erlauben [RWPD06]. In [PY02] beschreiben Pattanaik und Yee ein Verfahren für einen weiteren kantenerhaltenden Glättungsfilter im Kontext des Tone Mappings. In der Bildverarbeitung existieren ebenfalls einige digitale Filter, die Bilddaten glätten, ohne dabei starke Kanten zu zerstören [Reh00]. Diese Filter können für lokale Tone-Mapping-Operatoren besonders interessant sein.
Der Minimum-Varianz-Filter von Kuwahara [KHEK76] ist ein Beispiel für einen kantenerhaltenden Glättungsfilter. Dieser nicht-lineare Filter unterteilt für jeden Pixel des Bildes eine Pixelnachbarschaft fester Größe in vier leicht überlappende Teilnachbarschaften. Für jede Teilregion wird die Abweichung vom Mittelwert der Gesamtregion, die sogenannte Varianz, berechnet. Zusätzlich wird für jede Teilregion ein Mittelwert bestimmt. Im Ergebnisbild wird der Pixel auf den Mittelwert derjenigen Teilregion gesetzt, welche die kleinste Varianz besitzt. Dadurch werden homogene Bildregionen stärker geglättet, während Kanten weitgehend intakt bleiben.
Mit einem lokalen Tone-Mapping-Operator auf Basis des Kuwahara-Filters konnte jedoch nicht ganz die Qualität des lokalen Operators von Reinhard aus Kapitel 6.3 erreicht werden. Außerdem ist die Leistung bei Filtergrößen ab 5x5 nicht mehr für eine Echtzeitberechnung in höheren Auflösungen geeignet. Mit kleineren Filtermasken, wie 3x3, konnten homogene Bildregionen nicht mehr ausreichend stark geglättet werden. In diesem Fall kam es zu sichtbaren lokalen Kontrastverlusten. Der starke Leistungseinbruch bei der Verwendung von großen Filtermasken lässt sich durch die hohe Anzahl der notwendigen Texturzugriffe pro Pixel erklären. Dabei ist ein genereller Nachteil dieser digitalen Filter, dass sie nicht linear sind. Im Unterschied zu den Gaussfiltern sind sie nicht separierbar,

wodurch sie insbesondere bei großen Filtermasken nur bedingt für eine GPU-Implementation geeignet sind. Bei der im Rahmen dieser Arbeit entstandenen Implementation des Kuwahara-Filters auf dem Fragmentprozessor der Graphikhardware ist es beispielsweise technisch nicht möglich, größere Filtermasken als 5x5 zu verwenden. Bei einem 5x5 Filter werden selbst bei einer effizienten Programmierung bereits 25 Texturzugriffe pro Pixel benötigt, wobei die Werte anschließend in den temporären Registern des Fragmentprozessors gehalten werden müssen. Bei größeren Filtermasken konnte der Fragmentshader nicht mehr kompiliert werden, da laut Compiler die maximale Anzahl der temporären Register im Fragmentshader überschritten wurde. Ein weiterer Leistungsnachteil ensteht für die GPU durch den Prozess zur Bestimmung der Teilregion mit der kleinsten Varianz. Hierfür sind mehrere bedingte Sprünge durch *IF*-Abfragen notwendig. Die Verwendung von bedingten Sprüngen ist aufgrund des parallelen Programmiermodells der GPU relativ teuer.
Mit der Implementation des Kuwahara-Filters in Verbindung mit einem lokalen Tone-Mapping-Operator konnte eine deutliche Verbesserung des Kontrastes im Vergleich zu dem globalen Operator von Reinhard festgestellt werden. Mit dieser Vorgehensweise war jedoch eine Echtzeitfähigkeit gemäß den Anforderungen nicht gegeben.
In [WJPS$^+$00] beschreiben Henrik Wann Jensen et al. ein weiteres Verfahren zur Glättung von Bilddaten. Auch dieses Verfahren ermöglicht es Bildkanten weitgehend zu erhalten. Für die Glättung von homogenen Bildregionen wird dazu ein Glättungsfilter mit einer Gaussverteilung verwendet. In Formel (54) ist eine typische Gaussverteilung mit einer Standardabweichung $d$ für einen Punkt an den relativen Koordinaten $x, y$ aufgeführt [KMS05]:

$$G(x,y) = \frac{1}{\pi d^2} \exp^{\left(-\frac{x^2+y^2}{d^2}\right)} \tag{54}$$

Bei der Konvolution der Bilddaten wird die Standardabweichung $d$ allerdings nicht global für das komplette Bild festgelegt, sondern für jeden Bildpunkt neu berechnet. Dazu nutzt Jensen den *Gradientenbetrag* der einzelnen Bildpunkte. Dieser Wert kann als Maß für die Stärke einer Kante in einem Bildpunkt verwendet werden [Reh00]. In Formel (55) ist die dynamische Berechnung der Standardabweichung $d$ nach Jensen aufgeführt [WJPS$^+$00]:

$$d = d_{max} \left(grad_{i,max} - grad_i(x,y)\right) / grad_{i,max} \tag{55}$$

Der Parameter $d_{max}$ gibt die maximal mögliche Standardabweichung für die Gaussverteilung zur Glättung an. Der Parameter $grad_{i,max}$ ist als kleinste untere Schwelle für den Gradientenbetrag $grad_i(x,y)$ eines Pixels im Bild zu sehen, für den keinerlei Glättung erfolgen soll. Eine dynamische Berechnung der Standardabweichung $d$ führt zu unterschiedlichen Gaussgewichten der einzelnen Pixel. Dadurch

$S_x$

| | | |
|---|---|---|
| 1 | 0 | -1 |
| 2 | 0 | -2 |
| 1 | 0 | -1 |

horizontale Richtung

$S_y$

| | | |
|---|---|---|
| 1 | 2 | 1 |
| 0 | 0 | 0 |
| -1 | -2 | -1 |

vertikale Richtung

**Abbildung 20:** Die Filtermasken des Sobel-Operators

ist es möglich, homogene Bildregionen stärker zu glätten, während Kantenpixel eine geringere Gewichtung erhalten und damit weniger geglättet werden. Das Verfahren kann in einer veränderten Form in diese Arbeit übernommen werden.
Damit das Bild der relativen Leuchtdichten abhängig von den lokalen Kantenstärken geglättet werden kann, müssen die Gradientenbeträge $grad_i(x, y)$ für jeden Bildpunkt bekannt sein. In einem separaten Renderdurchlauf wird über einen Shader zunächst das Gradientenbild erzeugt. Hierbei wird für jeden Pixel aus dem Bild der relativen Leuchtdichten $\overline{L}_i(x, y)$ ein entsprechender Gradientenbetrag berechnet und in eine Render-Textur geschrieben. In [WJPS$^+$00] beschreibt Jensen jedoch nicht, welches Verfahren zur Bestimmung der Gradientenbeträge eingesetzt werden soll.
Aus diesem Grund wurde zunächst mit verschiedenen *Kantendetektoren* aus der Bildverarbeitung experimentiert. Diese speziellen Operatoren können Grenzen zwischen homogenen Bildregionen erkennen, indem Diskontinuitäten zwischen benachbarten Pixelwerten bestimmt werden [Reh00]. Ein Maß für die Stärke einer Diskontinuität ist durch die erste oder zweite diskrete Ableitung der Pixelwerte an der jeweiligen Bildposition gegeben [Reh00]. Ein bekannter Operator zur Detektion von Kanten in horizontaler und vertikaler Richtung ist der Sobel-Operator [Reh00]. Dieser Operator basiert auf der ersten diskreten Ableitung und nutzt eine spezielle Filtermaske zur Differenzierung. Dabei werden üblicherweise zwei 3x3 Filtermasken verwendet, die neben der eigentlichen Kantendetektion in horizontaler und vertikaler Richtung eine zusätzliche Glättung quer zur Differenzierungsrichtung durchführen. Aufgrund einer speziellen Binomial-Verteilung der Filterkoeffizienten ist dieser Kantendetektor insgesamt weniger anfällig für Bildrauschen [Reh00]. In Abbildung 20 sind die beiden Filtermasken $S_x$ und $S_y$ des Sobel-Operators zur Detektion von horizontalen und vertikalen Kanten dargestellt. Die Erstellung des Gradientenbildes erfordert nun für jeden Bildpunkt eine Differenzierung in horizontaler und vertikaler Richtung durch eine Faltung des Bildes mit den beiden Filtermasken $S_x$ und $S_y$. Nach der Faltung sind für jeden Bildpunkt zwei Gradientenwerte $g_x(x, y)$ und $g_y(x, y)$ vorhanden. Die beiden Gradi-

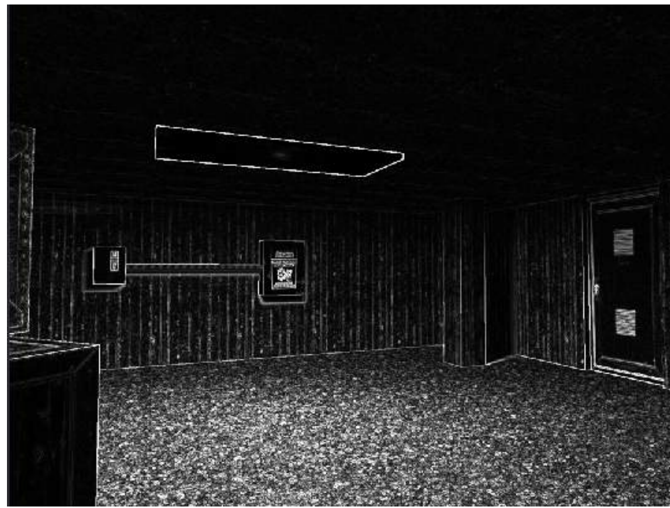
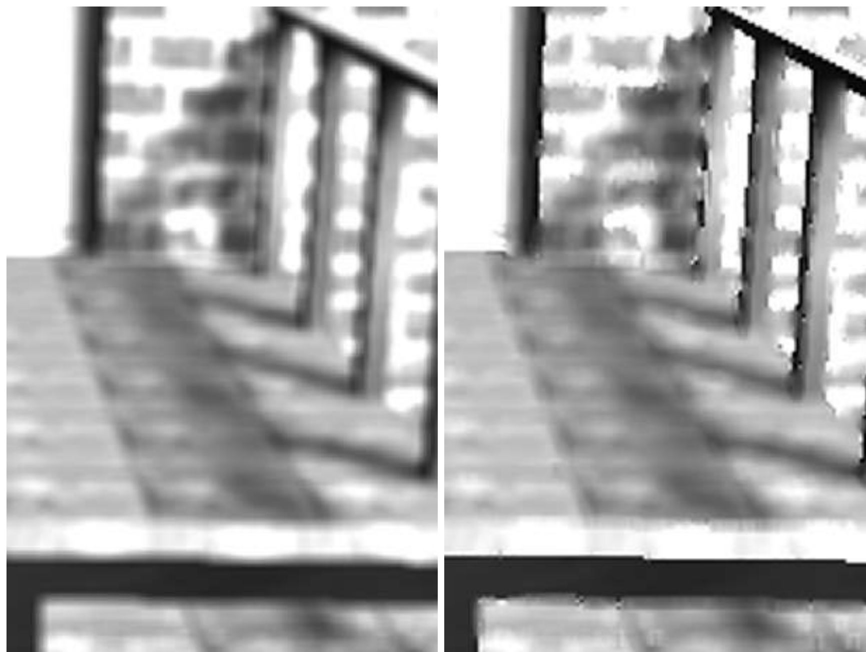


**Abbildung 21:** Visualisierung des Gradientenbildes nach Anwendung des Sobel-Operator

entenwerte können durch Formel (56) zu einem Gradientenbetrag $grad_i(x, y)$ für jeden Pixel zusammengefasst werden:

$$grad_i(x, y) = \sqrt{g_x(x, y)^2 + g_y(x, y)^2} \tag{56}$$

Dadurch ergibt sich das Gradientenbild $grad_i(x, y)$, das jedem Pixel aus dem Bild der relativen Leuchtdichten $\overline{L}_i(x, y)$ einen entsprechenden Gradientenbetrag zuweist. Diese Daten können nun als Eingabe für den dynamischen Glättungsfilter genutzt werden.
Nach der Auswertung verschiedener Kantendetektoren wird der Sobel-Operator in die Implementation übernommen. Der Operator lässt sich effizient in nur einem Renderdurchlauf mit sechs Texturzugriffen pro Pixel realisieren und ist daher auch bei höheren Bildauflösungen ausreichend schnell. Als zusätzlicher Parameter wird eine globale Skalierungsgröße für die Gradientenbeträge genutzt, wodurch Gradienten hervorgehoben oder unterdrückt werden können. Der Parameter bewirkt in einigen Fällen eine weitere Kontrastverbesserung. Abbildung 21 zeigt ein typisches Gradientenbild nach Anwendung des Sobel-Operators. Nach der Erzeugung des Gradientenbildes kann die eigentliche Glättung durch den dynamischen Gaussfilter vorgenommen werden. Für das ursprüngliche Verfahren von Jensen ist jedoch eine weitere Modifikation angedacht. Dabei wird ein globaler Schwellwert eingesetzt, über den entschieden werden kann, ob ein Pixelwert auf einer Kante überhaupt geglättet werden soll. Damit wird der kantenerhaltende Charakter des Filters verstärkt und eine Optimierung des Fragmentshaders ermöglicht. Das Verfahren wird in zwei Renderdurchläufen, einmal für die horizontale und einmal für



(a) 5x5 Gaussfilter mit $d = 8.0$ (b) 5x5 kantenerhaltender Filter

**Abbildung 22:** Vergleich zwischen Standardglättungsfilter und kantenerhaltenden Glättungsfilter

die vertikale Richtung, durchgeführt. Wie bereits in Kapitel 5.3 beschrieben, lässt sich unter Ausnutzung der Separierbarkeit eines Filters ein quadratischer Aufwand bezogen auf die Anzahl der Texturzugriffe vermeiden. Diese Optimierung kann angewandt werden, da die Berechnung der Gradientenbeträge nicht während der Faltung durchgeführt wird, sondern vorab in einem eigenen Renderdurchlauf.

In Listing 1 ist der gesamte Algorithmus in einer vereinfachten Form als Pseudo-Code aufgeführt. In Abbildung 22 ist der kantenerhaltende Glättungsfilter im Vergleich zu einem einfachen Glättungsfilter mit Gaussverteilung in einer Vergrößerung dargestellt. Abschließend werden die lokalen Mittelwerte der relativen Leuchtdichten in eine Render-Textur geschrieben, die als Eingabe für die lokale Operatorfunktion zur Skalierung der Leuchtdichten aus Kapitel 7.2.6 verwendet werden kann.

Mit dem zuvor vorgestellten Verfahren sind insgesamt gute Ergebnisse zu erzielen. Daher wird es in die Implementation übernommen. Die typischen Halo-Artefakten an Kantenübergängen mit starken Kontrastunterschieden sind bei Anwendung des Verfahrens kaum sichtbar. Rein subjektiv betrachtet sind die Ergebnisbilder in vielen Fällen sogar qualitativ mit den Resultaten des lokalen Tone-Mapping-Operators von Reinhard aus Kapitel 6.3 vergleichbar. Bei hohen Bildauflösungen kann der Operator von Reinhard jedoch teilweise zu kontrastreicheren Ergebnissen führen. Dabei sind durch Modifikationen an den verfügbaren Parametern des hier um-

**Algorithm 1** Algorithmus für den lokalen Tone-Mapping-Operator

$gradientImage \leftarrow createGradientImage(inputImage)$
**for all** $pixel \in inputImage$ **do**
  $grad \leftarrow gradientImage(pixel)$
  **if** $grad \leq threshold$ **then**
    $scaling \leftarrow 0$
    $weightedSum \leftarrow 0$
    **for all** $npixel \in neightbourPixels(pixel)$ **do**
      $grad \leftarrow gradientImage(npixel)$
      $deviation \leftarrow calculateDeviationFromGradient(grad)$
      $gaussianWeight \leftarrow calculateGaussianWeight(npixel, deviation)$
      $scaling \leftarrow scaling + gaussianWeight$
      $weightedSum \leftarrow weightedSum + (gaussianWeight \cdot npixel)$
    **end for**
    $outputImage(pixel) \leftarrow weightedSum/scaling$
  **else**
    $outputImage(pixel) \leftarrow pixel$
  **end if**
**end for**

(a) Verfahren nach Reinhard et al. (b) Eigenes Verfahren

**Abbildung 23:** Ergebnisbilder der beiden Tone-Mapping-Verfahren
*Quelle Originalbild: Begleit-DVD [RWPD06]*

gesetzten Verfahrens ebenfalls weitere Kontrastverbesserungen möglich. Abbildung 23 zeigt die Ergebnisse zum Vergleich.
Die Leistungsmessungen aus Kapitel 8.1.3 belegen, dass die Geschwindigkeit des zuvor vorgestellten Operators im Vergleich zu Reinhards lokalem Operator bei einer Umsetzung auf der GPU deutlich höher sein kann. Der hier vorgestellte lokale Operator benötigt allerdings mehr Leistungsressourcen als die meisten globalen Operatoren. Eine Verbesserung bringt die Verwendung einer kleineren 3x3 Filtermaske während der Glättung. Dadurch werden Texturzugriffe eingespart, wobei gleichzeitig jedoch eine geringere Kontraststeigerung erreicht wird. Gute Ergebnisse, die mit Reinhards „Dodging and Burning"-Algorithmus konkurrieren können, werden mit Filtergrößen ab 5x5 Pixeln erzielt.

### 7.2.3 Temporäre Adaptation

Das Tone-Mapping-Verfahren auf Basis des photographischen Modells von Reinhard et al. aus Kapitel 6.3 ist zunächst nur für einzelne Bilder mit statischer Beleuchtung geeignet. Zur Simulation der visuellen temporären Adaption in einer 3D-Umgebung mit wechselnden Beleuchtungsverhältnissen ist es notwendig das Verfahren zu modifizieren. Dazu existieren in der Literatur einige Ansätze, wobei eine Anpassung der linearen Skalierung zur Bestimmung der relativen Leuchtdichten $\overline{L}_i(x, y)$ üblich ist [KMS05, GWWH03, RJIH04].
Bei einer Bildfolge mit wechselnden Beleuchtungsverhältnissen können sich die einzelnen Leuchtdichteverteilungen pro Einzelbild stark voneinander unterscheiden. In einem Bild kann beispielsweise eine dunkle Ecke eines Raumes betrachtet werden, während im nächsten Bild eine sehr helle und große Lichtquelle sichtbar ist. Die durchschnittliche logarithmische Leuchtdichte $L_{i,avg}$ wäre für beide Bilder stark unterschiedlich. Das kann bei der späteren Darstellung der Bildfolge zu Diskontinuitäten führen.
Der Wert für den Szenen-Key $\alpha$ ist ebenfalls für die Berechnung der relativen Leuchtdichten relevant. Bei einer Bildfolge wäre eine automatische Anpassung von $\alpha$ hilfreich. Dazu wurde in Kapitel 6.3.1 bereits die Formel (45) von Reinhard vorgestellt, mit der eine Abschätzung von $\alpha$ aus verschiedenen bildabhängigen Größen möglich ist. Es hat sich jedoch herausgestellt, dass durch die Verwendung dieser Formel ebenfalls Diskontinuitäten auftreten können. Das lässt sich damit begründen, dass die Formel die beiden bildabhängigen Größen $L_{i,min}$ und $L_{i,max}$ zur Bestimmung von $\alpha$ verwendet. Die beiden Werte können in Bildfolgen für verschiedene Einzelbilder jedoch stark variieren, wodurch wiederum Diskontinuitäten während der temporären Adaption auftreten können.
Damit solche Probleme vermieden werden, sollte der gesamte Skalierungsfaktor $\frac{\alpha}{L_{i,avg}}$ zur Bestimmung der relativen Leuchtdichten aus Formel (43) kontinuierlich und über die gesamte Bildfolge hinweg angepasst werden. Für eine geeignete Vorgehensweise werden dazu einige Verfahren betrachtet.
In [RJIH04] verwenden Ramsey et al. als Adaptionsgrößen einen Mittelwert für die durchschnittliche logarithmische Leuchtdichte $L_{i,avg}$ und den Szenen-Key $\alpha$

für Bildfolgen fester Länge. Dieses Verfahren zur temporären Adaption wird dabei eher im Kontext eines HDR-Videoplayers genutzt und hat keine empirische Grundlage. Daher wird von einer Implementation abgesehen.
Für die Simulation der temporären Adaption des menschlichen visuellen Systems existieren mittlerweile komplexe Modelle [KMS05]. In [IFM05] beschreiben Irawan et al. ein Modell zur temporären Adaption, das die verschiedenen Teilprozesse einzeln und unabhängig voneinander simuliert. Dabei wird zunächst grob zwischen neuronalen und photochemischen Prozessen der Adaption unterschieden, wobei diese beiden Teilprozesse nochmals verfeinert werden. Nach Krawczyk et al. ist es jedoch bei der Simulation der temporären Adaption wichtiger, den Prozess als Ganzes aufzufassen und zu simulieren [KMS05]. Das Modell von Irawan wird aufgrund der hohen Komplexität nicht in die Implementation übernommen.
In [KMS05] verwenden Krawczyk et al. ein kompaktes Modell zur Simulation der temporären Adaption. Das Modell basiert auf dem physiologischen Modell von Durand und Dorsey aus [DD00]. Krawczyk nutzt hierbei für jedes Einzelbild einer Bildfolge eine adaptierte Größe als durchschnittliche logarithmische Leuchtdichte. Die adaptierte Größe $L_{i,adp}$ wird pro Einzelbild berechnet und später zur linearen Skalierung der Leuchtdichten genutzt. Krawczyk verwendet die spezielle Exponentialfunktion aus Formel (57) von Durand und Dorsey [DD00]:

$$L_{i,adp} = L_{i,adp}^{old} + (L_{i,avg} - L_{i,adp}^{old})(1 - e^{-\frac{T}{\tau(L_{i,avg})}}) \tag{57}$$

Die Funktion modelliert den zeitlichen Verlauf der Adaption. Der Wert für die adaptierte Größe $L_{i,adp}$ konvergiert mit einer bestimmten Geschwindigkeit gegen die durchschnittliche Leuchtdichte $L_{i,avg}$. Aus Kapitel 3.2 ist bereits bekannt, dass die Geschwindigkeit der temporären Adaption unter anderem vom Adaptionszustand des Betrachters abhängt. Dabei kann sich der Betrachter im Zustand einer Hell-Dunkel- oder Dunkel-Hell-Adaption befinden, wobei der zeitliche Verlauf für beide Zustände unterschiedlich ist. In Formel (57) ist daher die adaptierte Größe $L_{i,adp}^{old}$ des vorherigen Bildes und die durchschnittliche logarithmische Leuchtdichte $L_{i,avg}$ des aktuellen Bildes enthalten.
Über den Exponenten $\frac{T}{\tau(L_{i,avg})}$ wird die Geschwindigkeit des Adaptionsprozesses gesteuert. Der Parameter $T$ gibt die diskrete Zeitdauer an, die für die Darstellung und Berechnung des vorherigen Einzelbildes benötigt wurde. Dadurch ist eine gleichmäßige Adaptionsgeschwindigkeit unabhängig von der Bildwiederholfrequenz möglich. Die Geschwindigkeit der temporären Adaption ist außerdem davon abhängig, ob der Betrachter die Szene hauptsächlich mit den Stäbchen im skotopischen Bereich oder mit den Zapfen im photopischen Bereich wahrnimmt. Dabei ist die Adaptionsgeschwindigkeit für Stäbchen und Zapfen unterschiedlich [DD00]. Es entsteht die Schwierigkeit, dass die Simulation für Stäbchen und Zapfen getrennt durchgeführt werden muss [KMS05]. Um dieses Problem zu umgehen, nutzt Krawczyk die Funktion aus Formel (58):

$$\tau(L_{i,avg}) = \sigma(L_{i,avg})\tau_{rod} + (1 - \sigma(L_{i,avg}))\tau_{cone} \tag{58}$$

Die Funktion interpoliert zwischen zwei Zeitkonstanten für die Stäbchen $\tau_{rod}$ und die Zapfen $\tau_{cone}$, wobei als Interpolationsgröße die Funktion $\sigma(L_{i,avg})$ zur Approximation der Stäbchensensitivität nach Hunt aus Kapitel 3.2 verwendet wird. Krawczyk nutzt für die beiden Konstanten die Werte $\tau_{rod} = 0.4$ und $\tau_{cone} = 0.1$, die ebenfalls in der Arbeit von Durand und Dorsey vorgeschlagen werden [DD00]. Der Funktionsgraph $\tau(L_{i,avg})$ ist in Abbildung 24 dargestellt.

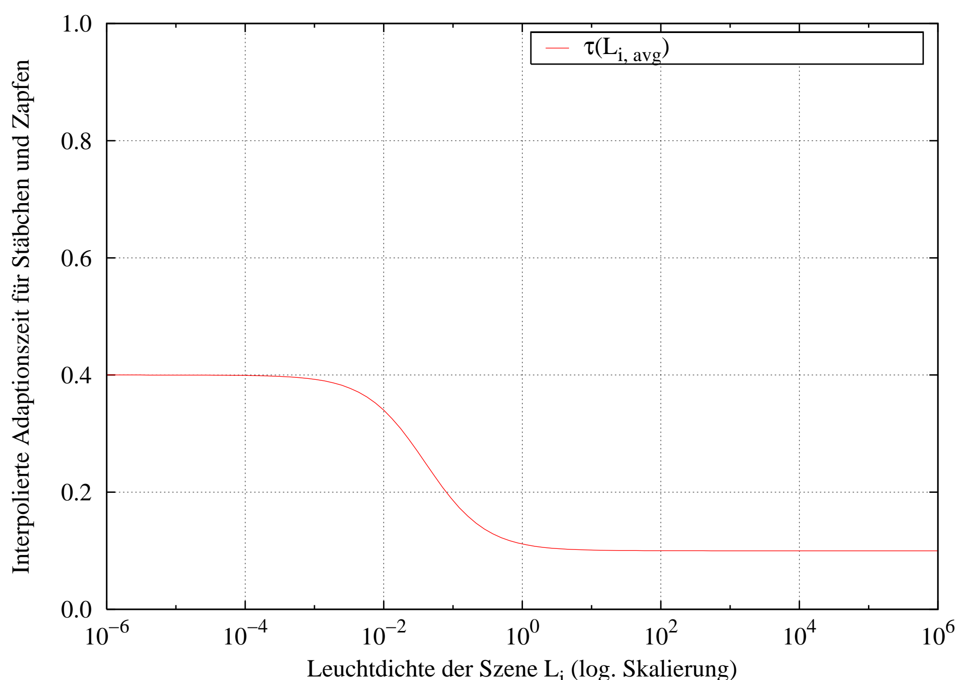



**Abbildung 24:** Interpolierte Adaptionszeit für Stäbchen und Zapfen

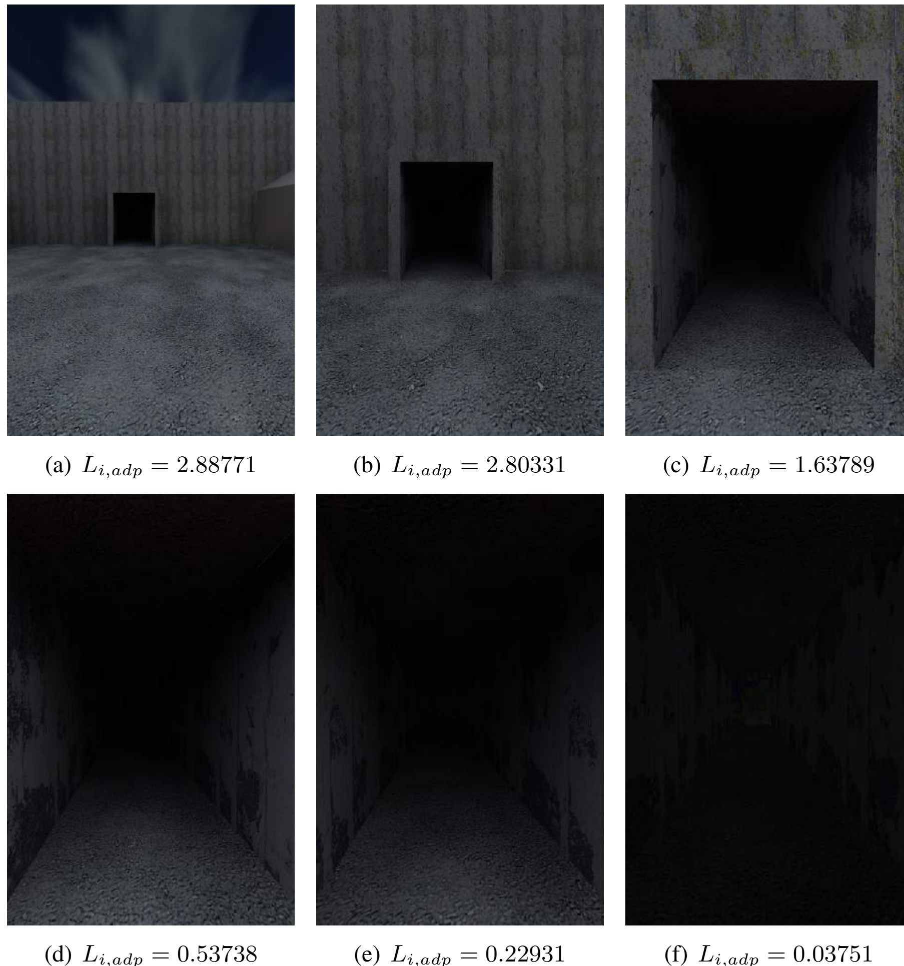
Aus dem Wert für die adaptierte durchschnittliche Leuchtdichte $L_{i,adp}$ berechnet Krawczyk in Formel (59) einen adaptierten Wert für den Szenen-Key $\alpha_{adp}$:

$$\alpha_{adp} = 1.03 - \frac{2}{2 + log_{10}(L_{i,adp} + 1)} \tag{59}$$

Insgesamt führt das Verfahren von Krawczyk zu plausiblen Ergebnissen und wird daher in die Implementation übernommen. Die Werte für die adaptierten Größen $L_{i,adp}$ und $\alpha_{adp}$ werden einmal pro Einzelbild direkt auf der CPU berechnet. Dabei wird der Wertebereich von $L_{i,adp}$ durch einen Minimal- und Maximalwert zusätzlich beschränkt, sodass numerische Berechnungsfehler, etwa durch Überläufe,

vermieden werden. Ward et al. nutzen dazu in [LRP97] als minimale Leuchtdichte, für die ein Adaptionsvorgang erfolgt, einen unteren Grenzwert von $10^{-4}$. Die adaptierten Größen $L_{i,adp}$ und $\alpha_{adp}$ können schließlich als Eingabeparameter an den Fragmentshader für die lineare Skalierung zur Bestimmung der relativen Leuchtdichten $\overline{L}_i(x,y)$ übergeben werden.

Die automatische Berechnung des Wertes für $\alpha_{adp}$ resultiert teilweise in sehr klei-



(a) $L_{i,adp} = 2.88771$ (b) $L_{i,adp} = 2.80331$ (c) $L_{i,adp} = 1.63789$

(d) $L_{i,adp} = 0.53738$ (e) $L_{i,adp} = 0.22931$ (f) $L_{i,adp} = 0.03751$

**Abbildung 25:** Temporäre Adaption für eine Bildfolge

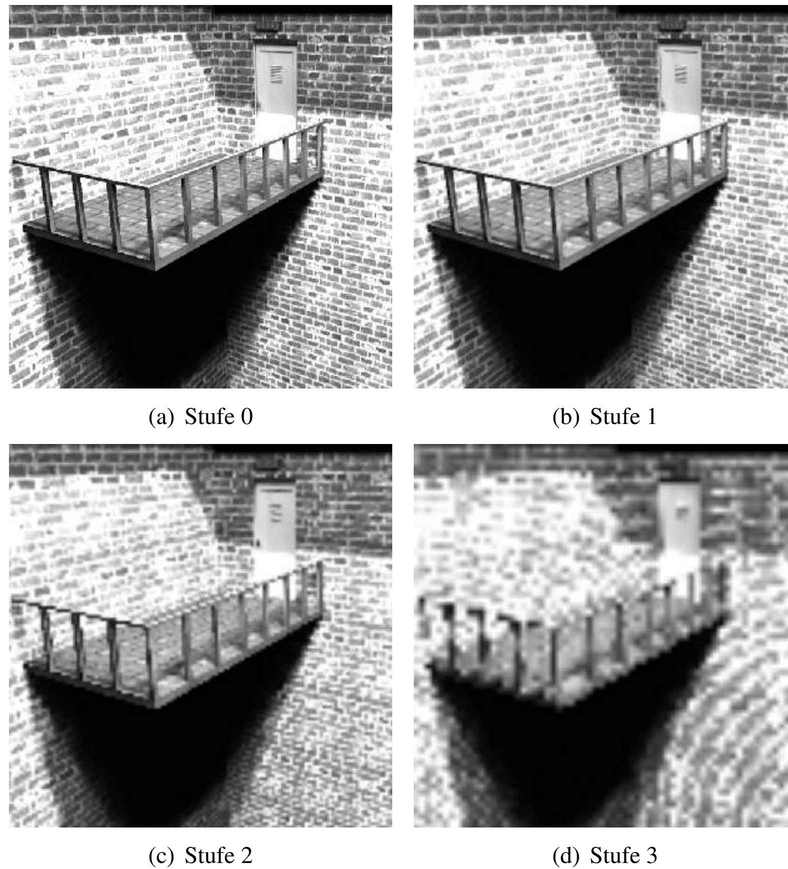
nen Werten. Das führt im Endresultat mitunter zu sehr dunklen Bildern. Hier wäre eine andere Berechnungsmethode eventuell besser geeignet. Leider existieren für eine empirisch gestützte Berechnung eines zeitlich adaptierten Wertes für $\alpha_{adp}$ keine umfangreichen experimentellen Daten [KMS05]. Eine Mittelung mehrerer Werte zur Bestimmung von $\alpha_{adp}$, wie sie etwa in [RJIH04] von Ramsey et al. verwendet wird, könnte vorteilhaft sein. In Abbildung 25 ist die temporäre Adaption

über eine Bildfolge hinweg unter Anwendung des zuvor beschriebenen Verfahrens dargestellt. Dabei sind jeweils die vollständig konvergierten Werte für $L_{i,adp}$ angegeben.

### 7.2.4 Verlust der Sehschärfe

Die Simulation zum Verlust der Sehschärfe erfolgt in der Literatur häufig durch eine Konvolution der Bilddaten mit Gaussfiltern [WJPS$^+$00, FPSG96, KMS05]. Dadurch wird eine Glättung des Bildes erreicht, wobei feinere Bilddetails verloren gehen. Die Stärke der Glättung kann durch den Filterradius und die Standardabweichung des Gaussfilters beeinflusst werden. Damit die Simulation des Sehschärfeverlustes möglichst realistisch ist, kann die Glättungsstärke an den empirischen Daten verschiedener Sehschärfeexperimente ausgerichtet werden [LRP97]. In Kapitel 3.3 wurde dazu die Funktion aus Formel 30 von Ward et al. vorgestellt, die empirische Messdaten der Experimente zur Sehschärfe von Shlaer [Shl37] approximiert. Die Funktion wird beispielsweise von Ward et al. in [LRP97] und Krawczyk et al. in [KMS05] für die Simulation des Sehschärfeverlustes genutzt.
Diese Simulation erfordert teilweise große Filterkerne [KMS05]. Aus Kapitel 5.3 ist bekannt, dass große Filterkerne viele Texturzugriffe benötigen und dadurch die Gesamtleistung reduzieren. In [FPSG96] verwendet Ferwerda et al. zur Simulation des Sehschärfeverlustes eine globale Glättung des Bildes. Dabei wird das gesamte Bild global mit einem Gaussfilter gefaltet, wobei allerdings Details in den Bildbereichen verloren gehen, die aufgrund ihrer Leuchtdichteverteilung scharf dargestellt werden sollten. Daher wäre es besser, die Glättung durch einen Filter lokal vorzunehmen [LRP97]. Eine selektive Glättung des Bildes erschwert jedoch eine Separierung des Gaussfilters, wodurch unter Umständen die Anzahl der Texturzugriffe während der Faltung nochmals steigt. Daher wird im Folgenden ein Verfahren vorgestellt, das keine Filter zur Konvolution verwendet.
Anstatt Bilddetails durch Glättungsfilter zu entfernen, kann für die verschiedenen Detailstufen während der Sehschärfesimulation eine Mipmapping-Pyramide eingesetzt werden. Die oberste Mipmap-Stufe ist direkt durch das Bild mit den relativen Leuchtdichten $\overline{L}_i(x, y)$ gegeben. Jede weitere Stufe wird durch eine sukzessive Verkleinerung der vorherigen Stufe berechnet, wobei die Auflösung jeweils in horizontaler und vertikaler Richtung halbiert wird. Vor der eigentlichen Berechnung des Sehschärfeverlustes werden die einzelnen Stufen wieder auf die Ausgangsgröße hochskaliert, wobei eine bilineare Filterung genutzt wird. Dadurch erhält man eine Reihe von Bildern, die verschiedene Detailstufen des Originalbildes repräsentieren. In Abbildung 26 sind vier solcher Detailstufen exemplarisch dargestellt.
Die komplette Mipmap-Pyramide wird für jedes Einzelbild einer Bildfolge neu aufgebaut. Eine moderne Graphik-API wie OpenGL 2.0, bietet die Möglichkeit, automatisch Mipmaps von Render-Texturen zu generieren [nVi05]. Dabei berechnet die Hardware sämtliche Mipmap-Stufen neu, wenn die erste Stufe der Pyramide, also die Textur mit der höchsten Auflösung, geändert wird. Dieser Me-

chanismus wird für das Verfahren zur Simulation der Sehschärfe genutzt, um die Mipmap-Pyramide automatisch und effizient pro Einzelbild neu zu erzeugen.
Ein weiteres Kernfeature von OpenGL 2.0 ist die volle Unterstützung für Tex-



(a) Stufe 0 (b) Stufe 1

(c) Stufe 2 (d) Stufe 3

**Abbildung 26:** Verschiedene Detailstufen des Sehschärfeverlustes

turdaten, die nicht auf eine Auflösung von $2^N x 2^M$ beschränkt sind [Mem06a]. Die „Non-Power-Of-Two-Textures“ unterstützen bilineare und trilineare Texturfilterungen, die für das hier beschriebene Verfahren wichtig sind. Leider bietet die im Rahmen dieser Arbeit verwendete Graphikhardware keine Unterstützung für bilineare und trilineare Filterungen bei einkomponentigen Texturen und bei Texturen mit 32-Bit-Genauigkeit an. Daher müssen die relativen Leuchtdichten in einem anderen Texturformat abgelegt werden, das eine hinreichende Unterstützung für die Filterarten bietet. Hierfür wird ein dreikomponentiges Texturformat mit 16-Bit-Fließkommagenauigkeit [14] gewählt. Dieses Format bietet eine ausreichende numerische Präzision, ohne die Speicherbandbreite unnötig zu belasten.
Nach dem automatischen Aufbau der Mipmap-Pyramide können die eigentlichen Berechnungen zur Simulation des Sehschärfeverlustes durchgeführt werden. Das

[14] Format: GL_RGB16F_ARB

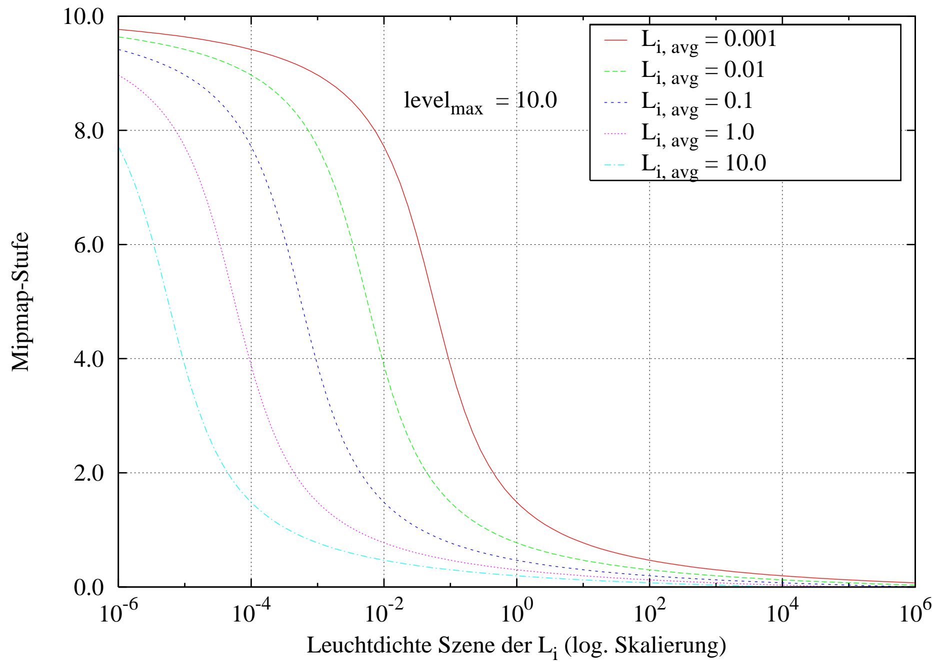


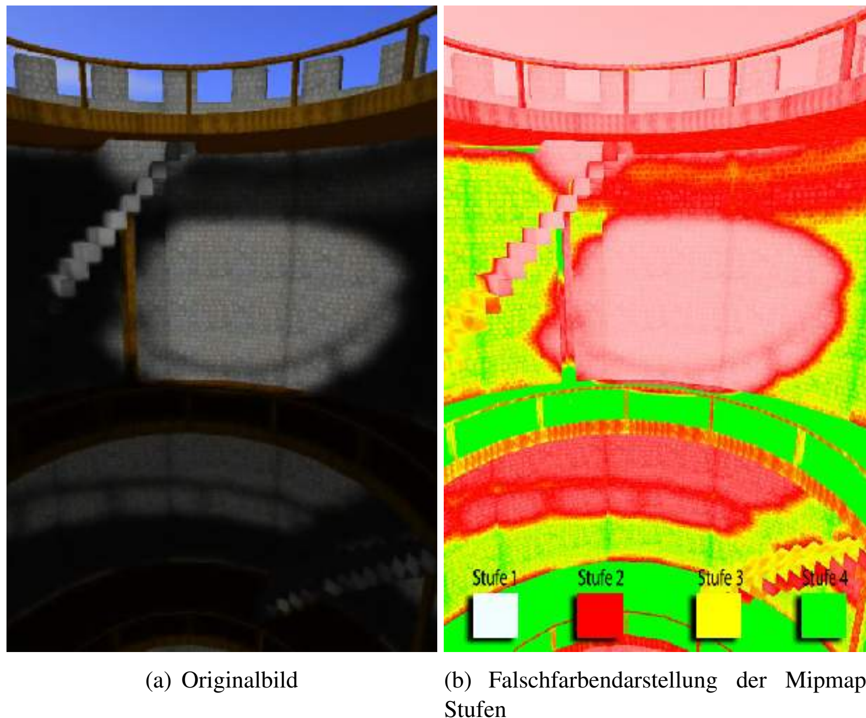

**Abbildung 27:** Auswahl der Mipmap-Stufe

Verfahren wählt dabei für jeden Pixel aus dem Bild der relativen Leuchtdichten $\overline{L}_i(x, y)$ ein entsprechendes Pixel aus einer der Mipmap-Stufen und schreibt es in eine separate Render-Textur. Die Auswahl der Mipmap-Stufe wird dabei durch die einzelnen Leuchtdichten der Pixel $\overline{L}_i(x, y)$ und auf Basis der Funktion aus Formel (13) zur Sehschärfe von Ward et al. aus Kapitel 3.3 vorgenommen. Zusätzlich wird die trilineare Filterung der Graphikhardware genutzt, um eine Auswahl von Pixeln zwischen zwei benachbarten Detailstufen zu interpolieren. Zunächst ist jedoch ein geeignetes Verfahren zur Umrechnung der Funktionswerte auf die verschiedenen Mipmap-Stufen zu finden. Hierbei ergibt sich die Schwierigkeit, dass in der Literatur keine experimentellen Daten für diese spezielle Vorgehensweise existieren. Nach einigen experimentellen Versuchen wird die folgende Funktion aus Formel (60) zur Auswahl der Mipmap-Stufen $level(\overline{L}_i(x, y))$ genutzt:

$$level(\overline{L}_i(x, y)) = level_{max} - level_{max}\left(V_{acuity}\left(10^4 \cdot L_{i,avg} \cdot \overline{L}_i(x, y)\right)/51.5\right) \tag{60}$$

In der Formel wird neben der Funktion zur Berechnung der Sehschärfe $V_{acuity}(L)$ aus Kapitel 3.3 zusätzlich die durchschnittliche logarithmische Leuchtdichte der Szene $L_{i,avg}$ verwendet. Hierdurch kann global pro Einzelbild festgelegt werden, wie die einzelnen Mipmap-Stufen zu wählen sind. Dabei fallen die Effekte zum

Sehschärfeverlust bei helleren Bildern weniger stark aus, während dunklere Bilder lokal unschärfer dargestellt werden. Ist die Simulation der temporären Adaption aktiviert, wird in Formel (60) statt $L_{i,avg}$ die adaptierte durchschnittliche logarithmische Leuchtdichte $L_{i,adp}$ aus Kapitel 7.2.3 genutzt. Durch den Parameter $level_{max}$ wird die maximale Mipmap-Stufe angegeben, die als niedrigste Detailstufe verwendet wird. Dabei ist die maximal mögliche Stufe in der Mipmap-Pyramide von der Bildauflösung abhängig, wobei die unterste Mipmap-Stufe immer einer 1x1 Textur und damit einer einzelnen Leuchtdichte entspricht.
Die Funktion aus Formel (60) führte bei vielen Bildern zu guten Ergebnissen.




(a) Originalbild

(b) Falschfarbendarstellung der Mipmap-Stufen

**Abbildung 28:** Wahl der Mipmap-Stufen pro Pixel

Es wäre jedoch in Zukunft wünschenswert, wenn die Funktion zur Auswahl der Mipmap-Stufe durch empirische Daten gestützt werden könnte. Bei einigen digitalen HDR-Photographien führte die Funktion teilweise zu sehr unscharfen Bildern. In Abbildung 27 sind verschiedene Graphen für die Funktion aus Formel (60) exemplarisch dargestellt. Hierbei ist gut zu erkennen, dass der Parameter $L_{i,avg}$ eine Verschiebung des Graphen in X-Richtung bewirkt.
In Abbildung 28 sind die Zuweisungen der einzelnen Pixel zu ihren korrespondierenden Detailstufen aus der Mipmap-Pyramide schematisch dargestellt.
Das zuvor vorgestellte Verfahren kann jedoch zu Bildartefakten führen. Diese Artefakte in Abbildung 29 treten hauptsächlich an Übergängen auf, in denen zwei unterschiedliche Mipmap-Stufen abrupt aneinander stoßen und nicht kontinuier-

(a) Originalbild

(b) Falschfarbendarstellung der Mipmap-Stufen

**Abbildung 29:** Bildartefakte durch angrenzende Mipmap-Stufen

lich ineinander übergehen.
Das Auftreten der Artefakte lässt sich dadurch begründen, dass nach der Skalierung Diskontinuitäten zwischen zwei korrespondierenden Werten beider Mipmap-Stufen entstehen können. Im Ausblick dieser Arbeit in Kapitel 9 werden verschiedene Möglichkeiten zur Reduzierung der Artefakte angesprochen. Die Ansätze wurden jedoch nicht mehr in die Implementation übernommen.
Das zuvor beschriebene Verfahren zur Simulation der Sehschärfe führt zu plausiblen Ergebnissen und ist ausreichend performant für eine Echtzeitdarstellung gemäß den Anforderungen. Es wird daher in die Implementation übernommen. Zukünftig sollte das Verfahren jedoch optimiert werden, sodass die Bildartefakte reduziert werden. In Abbildung 30 sind die Ergebnisse des Verfahrens zu sehen.

### 7.2.5 Blendeffekte durch Streulicht

Für die Visualisierung von Blendeffekten bei hellen Lichtquellen in Computerspielen wird häufig ein Effekt eingesetzt, der als *Bloom* bezeichnet wird [Mas03,Car05, Jas03]. Hierbei werden zunächst besonders helle Bildbereiche in einem extra Renderdurchlauf ermittelt. Die Bildanteile können durch ein Schwellenwertverfahren über einen Fragmentshader extrahiert und in eine kleinere Render-Textur kopiert werden. So werden nachfolgende Berechnungen nicht in voller Bildauflösung

**Abbildung 30:** Verlust der Sehschärfe bei niedrigen Leuchtdichten

durchgeführt [Jas03]. Die Render-Textur mit den hellen Bildbereichen wird nun mit einem geeigneten Filter geglättet. Dazu kann beispielsweise der Glättungsfilter von Kawase aus Kapitel 5.3 genutzt werden. Abschließend wird die Render-Textur wieder auf die volle Auflösung skaliert und additiv mit den Originalpixelwerten kombiniert. Bei der Skalierung sollte jedoch ein bilinearer Filter oder eine andere höherwertige Filtermethode eingesetzt werden, sodass Interpolationsartefakte reduziert werden. Durch die additive Überlagerung entsteht ein Blendeffekt, wobei helle Bildbereiche in dunklere übergehen und diese teilweise überlagern. Das Verfahren wurde im Rahmen dieser Arbeit in einem ersten technischen Durchstich geprüft. Dabei wirkten die Ergebnisbilder jedoch insgesamt weichgezeichnet, wobei besonders Lichtquellen übertrieben stark dargestellt wurden. Zudem ist das Verfahren für eine Simulation von Blendeffekten durch Streulicht nicht ausreichend durch empirische Daten gestützt. So gibt es beispielsweise keinen Ansatz für eine dynamische Berechnung des Schwellwertes für die Extraktion der hellen Bildbereiche. Weiterhin ist unklar, wie stark die Glättung des extrahierten Bildes erfolgen soll. Das Verfahren wurde daher nicht in die Implementation übernommen. Für dramatische Lichteffekte in Computerspielen kann das Verfahren jedoch durchaus als praktikabel angesehen werden, zumal es sich effizient realisieren lässt.

Eine umfassende Darstellung zur physikalischen Simulation von Blendeffekten durch starke Lichtquellen ist durch Spencer et al. in [SSZG95] gegeben. Dabei wird unter anderem ein kontrastreduzierender Effekt beschrieben, der durch Streulicht

im Auge hervorgerufen wird. Der Effekt wird in der Literatur ebenfalls als *Bloom* oder *Glare* bezeichnet [SSZG95, KMS05, RWPD06]. In der Arbeit von Spencer sind noch weitere komplexere visuelle Effekte beschrieben, die bei einer Simulation von hellen Lichtquellen auftreten können. Von der Simulation dieser Effekte wurde jedoch abgesehen, da sich eine Berechnung auf der GPU nicht angeboten hat. Zudem beschränken sich viele Arbeiten in der Literatur auf die reine Simulation von Bloom-Effekten [KMS05, LRP97, Fah05].
Nach Spencer wird das Licht durch die Hornhaut, die Augenlinse und die erste Schicht der Netzhaut in einer schmalen kegelförmigen Verteilung auf die Photorezeptoren der Netzhaut gestreut. Dabei kann die Verteilung der Streuung durch Gaussprofile approximiert werden [SSZG95]. Aus Kapitel 3.4 ist bekannt, dass der Kontrastverlust durch Streulicht im skotopischen Bereich in einem stärkeren Maße vorhanden ist als im photopischen und mesopischen Bereich. Diese Beobachtungen werden nachfolgend für die Simulation von Blendeffekten durch Streulicht genutzt.
Um einen globalen Kontrastverlust durch Streulicht herbeizuführen, wird das Bild der relativen Leuchtdichten $\overline{L}_i(x, y)$ mit einem Gaussfilter gefaltet. Dazu können etwa die Gaussprofile in Formel (54) aus Kapitel 7.2.2 eingesetzt werden. Die Glättungsstärke wird durch eine Anpassung der Standardabweichung $d$ festgelegt. Die Standardabweichung wird dynamisch pro Einzelbild berechnet, wobei als

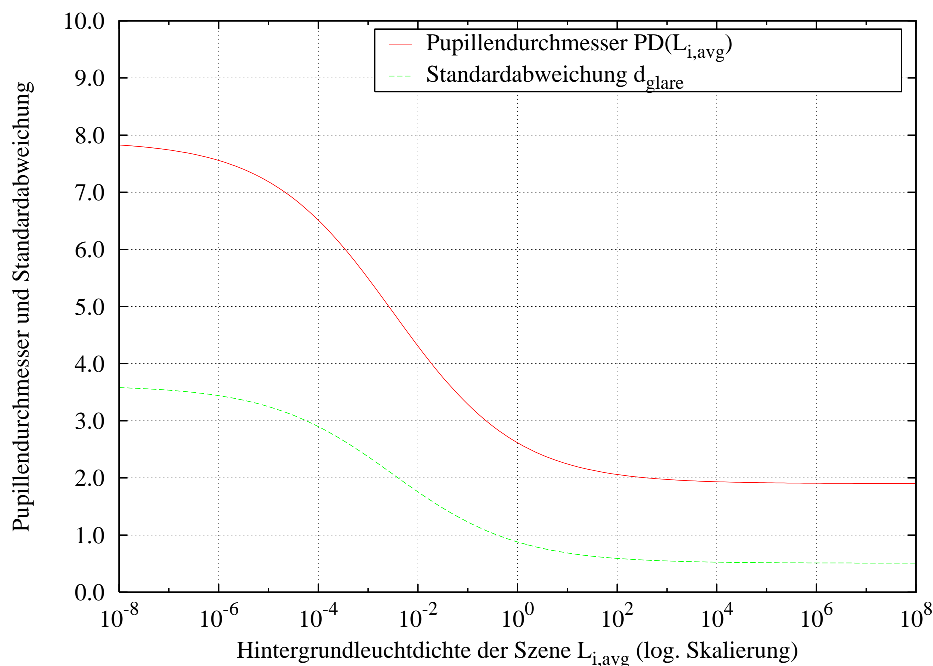



**Abbildung 31:** Standardabweichung im Verhältnis zur Pupillengröße

empirische Basis die Funktion $PD(L_{i,avg})$ aus Formel (14) zur Abschätzung des Pupillendurchmessers aus Kapitel 3.4 dient. Der Pupillendurchmesser ist von der aktuellen Hintergrundleuchtdichte abhängig. Als geeignetes Maß kann die durchschnittliche logarithmische Leuchtdichte $L_{i,avg}$ des Bildes genutzt werden. Dabei werden dunkle Bilder stärker geglättet, während für hellere Bilder eine kleinere Standardabweichung berechnet wird und damit eine schwächere Glättung erfolgt. In Formel (61) ist die Funktion zur Berechnung der Standardabweichung dargestellt:

$$d_{glare}(L_{i,avg}) = d_{min} + \frac{1}{6}\left|PD(L_{i,avg}) - 2\right|(d_{max} - d_{min}) \tag{61}$$

Der Wert $d_{min}$ gibt dabei eine untere Grenze für die kleinste mögliche Standardabweichung an, während $d_{max}$ den größtmöglichen Wert darstellt. Wird die Simulation zur temporären Adaption aus Kapitel 7.2.3 durchgeführt, kann in Formel (61) statt $L_{i,avg}$ die adaptierte durchschnittliche logarithmische Leuchtdichte $L_{i,adp}$ genutzt werden.

In Abbildung 31 ist der Funktionsgraph zur Berechnung der Standardabweichung zusammen mit dem Funktionsgraph der Pupillengröße zu sehen. Die Werte für die minimale und maximale Standardabweichung $d_{min}$ und $d_{max}$ orientieren sich

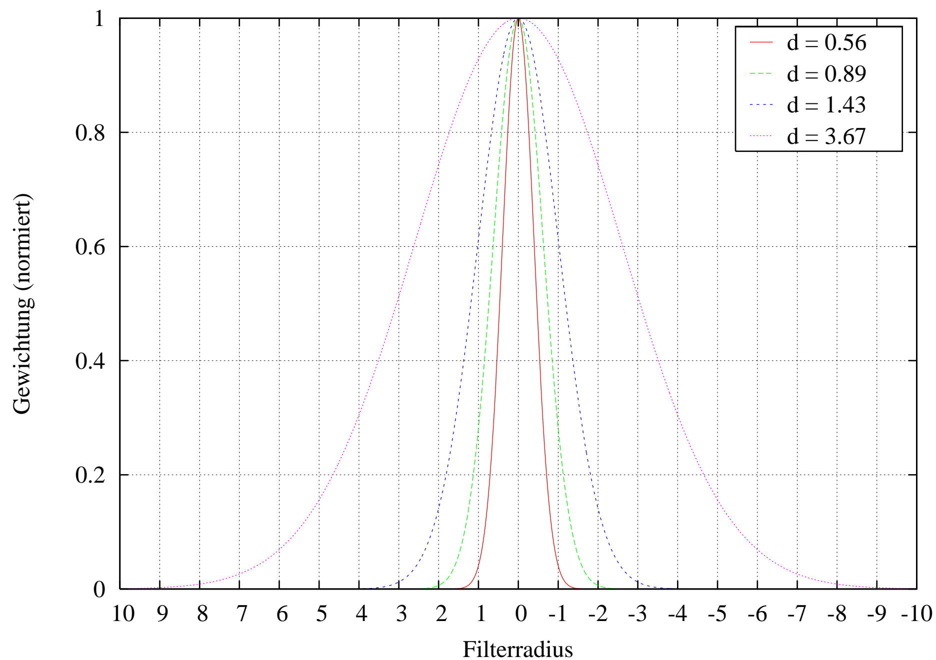
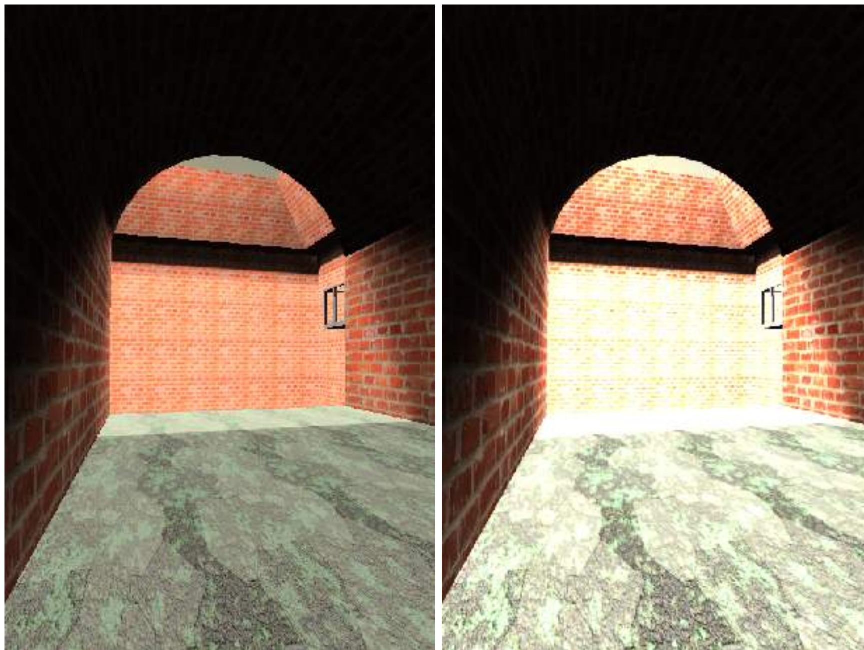



**Abbildung 32:** Verschiedene Gaussprofile zur Bildglättung für die Streulichtsimulation



(a) Ohne Streulicht (b) Mit Streulicht

**Abbildung 33:** Simulation von Streulicht

weitgehend an Beispielen aus der Literatur [KMS05]. In Abbildung 32 sind die Graphen einiger zur Bildglättung verwendeter Gaussprofile visualisiert.
Damit die Berechnungen zur Simulation von Streulicht möglichst effizient sind, erfolgt die Faltung durch einen separierbaren Gaussfilter in zwei Renderdurchläufen. Basierend auf der aktuellen Standardabweichung werden pro Einzelbild mehrere Gaussgewichte berechnet, wobei diese an einen Fragmentshader zur Berechnung der Konvolution übergeben werden. Für die Implementation ist zudem eine feste Filtergröße zur Faltung vorgesehen. Das gefilterte Bild wird abschließend in eine eigene Render-Textur geschrieben, die zu einem späteren Zeitpunkt von der Operatorfunktion in Kapitel 7.2.6 verwendet wird.
In [KMS05] verwendet Krawczyk et al. eine ähnliche Berechnung zur Simulation der Blendeffekte durch Streulicht. Dabei werden ebenfalls die relativen Leuchtdichten gefaltet, um eine Kontrastreduktion herbeizuführen.
In Abbildung 33 sind die Ergebnisse des oben vorgestellten Verfahrens zur Simulation von Blendeffekten durch Streulicht zu sehen.

### 7.2.6 Kompression der Leuchtdichten

Nach der Simulation der einzelnen Teile der menschlichen visuellen Wahrnehmung kann die eigentliche Kompression der Leuchtdichten vorgenommen werden. Dazu wird der Operator aus Formel (62) von Krawczyk et al. aus [KMS05] verwendet:

$$L_d(x,y) = \frac{L_{acuity}(x,y) + L_{glare}(x,y)}{1 + L_{mean}(x,y)} \tag{62}$$

Die komprimierten Leuchtdichten $L_d(x,y)$ werden aus den Render-Texturen für den Verlust der Sehschärfe $L_{acuity}(x,y)$, für die Streulichtsimulation $L_{glare}(x,y)$ und der Mittelwerte des lokalen Operators $L_{mean}(x,y)$ gewonnen. Sind die Teile für den lokalen Operator und die Simulation der Sehschärfe nicht aktiv, werden die Render-Texturen $L_{acuity}(x,y)$ und $L_{mean}(x,y)$ auf die Render-Textur mit den relativen Leuchtdichten $\overline{L}_i(x,y)$ gesetzt. In diesem Fall entspricht das Verfahren effektiv dem globalen Tone-Mapping-Operator von Reinhard et al. aus Kapitel 6.3.2. Nach der Kompression der Leuchtdichten wird die Simulation zum Verlust der Farbwahrnehmung durchgeführt, wobei die skalierten RGB-Werte für das Ausgabegerät bestimmt werden.

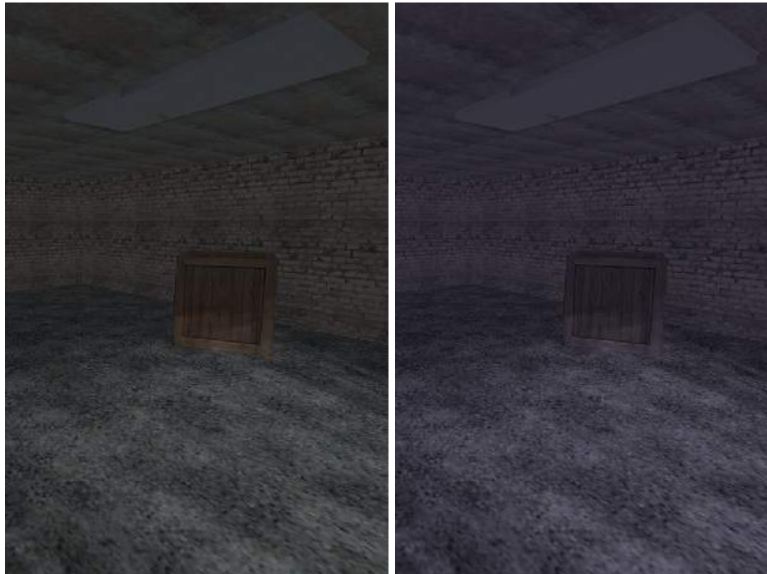
### 7.2.7 Verlust der Farbwahrnehmung

Eine einfache Möglichkeit den Verlust der Farbwahrnehmung aus Kapitel 3.5 zu simulieren, ist durch eine lineare Interpolation der Sättigungen der Pixelfarben, basierend auf ihren Leuchtdichten, gegeben. Dazu könnte man den Parameter $s$ aus Formel (8) linear zwischen $[0,1]$ interpolieren. Dieses einfache Modell ist jedoch empirisch nicht gestützt und wurde daher nicht in der Implementation verwendet. Desweiteren ist aus Kapitel 3.5 bekannt, dass die menschliche visuelle Perzeption bei dunklen Szenen, insbesondere während der Nacht, zu einer leichten Blauverschiebung neigt. Dieser Effekt wäre über eine einfache Sättigungsinterpolation ebenfalls schwer zu realisieren.
Nach Durand et al. und Krawczyk et al. kann der Verlust der Farbwahrnehmung über eine Interpolation zwischen zwei Funktionen für die unterschiedlichen Adaptionsbereiche simuliert werden [DD00,KMS05]. Die komprimierten RGB-Farbwerte der Pixel $RGB_d(x,y)$ werden aus einer Kombination der ursprünglichen RGB-Farbwerte des HDR-Bildes $RGB_i(x,y)$ und der monochromatischen Intensität, proportional zu der Funktion der skotopischen Sensitivität $\sigma(L)$ nach Hunt aus Kapitel 3.2, berechnet. Die Funktion ist in Formel (63) dargestellt:

$$RGB_d(x,y) = RGB_i(x,y)\frac{L_d(x,y)(1-\sigma(L_i(x,y)))}{L_i(x,y)} + \begin{pmatrix} 1.05 \\ 0.97 \\ 1.27 \end{pmatrix} \cdot L_d(x,y) \cdot \sigma(L_i(x,y)) \tag{63}$$

Die konstanten Koeffizienten $\{1.05, 0.97, 1.27\}$ des monochromatischen Teils führen zu einer leichten Blauverschiebung bei sehr dunklen Bildern [KMS05]. In der Implementation sind die Koeffizienten frei wählbar.
Mit dem vorgestellten Verfahren zur Simulation des Verlustes der Farbwahrnehmung lassen sich gute Ergebnisse erzielen. Es wird daher in die Implementation übernommen. Die Berechnung erfolgt nach Anwendung des Tone-Mapping-

Operators zur Kompression der Leuchtdichten in einem Shader auf der Graphikhardware. Abbildung 34 zeigt zwei Ergebnisbilder ohne und mit Simulation der Farbwahrnehmung.



(a) Ohne Verlust der Farbwahrnehmung (b) Mit Verlust der Farbwahrnehmung

**Abbildung 34:** Verlust der Farbwahrnehmung

#### 7.2.8 Gesamtübersicht

In diesem Teil der Arbeit wird eine Gesamtübersicht des Tone-Mapping-Verfahrens präsentiert. Dabei wird zunächst eine Zerlegung des Verfahrens in kleinere Bausteine vorgenommen, wodurch eine geeignete Sichtweise auf das gesamte Verfahren ermöglicht wird.
Eine Zerlegung des Verfahrens in einzelne Bausteine ist durch die Aufgabenverteilung der verschiedenen Shader gegeben. Weitere Bausteine können durch die Render-Texturen repräsentiert werden, die als Datenspeicher zu sehen sind. Zwischen den Shadern und den Render-Texturen ist ein unidirektionaler Datenfluss vorhanden. Die Shader lesen Daten aus den Render-Texturen und schreiben ihre Ausgaben wiederum in weitere Render-Texturen. Aus dieser Sichtweise heraus wurde das Diagramm in Abbildung 35 erstellt. Es zeigt eine mögliche Gesamtübersicht für das Tone-Mapping-Verfahren. Dabei repräsentieren die abgerundeten Formen einzelne Shader, während die rechteckigen Formen Render-Texturen darstellen. Lese- und Schreibzugriffe von Shadern sind mit gestrichelten beziehungsweise durchgängigen Pfeilen angedeutet. Der Teilprozess zur Simulation der temporären Adaption ist gesondert visualisiert, um anzuzeigen, dass die Berech-

nung auf der CPU erfolgt.

## 7.3 Implementierung

Dieser Teil der Ausarbeitung beschäftigt sich mit der Implementierung des zuvor vorgestellten Tone-Mapping-Operators. Zunächst folgt eine knappe Übersicht der für die Implementation verwendeten Bibliotheken, APIs und Werkzeuge.

### 7.3.1 Verwendete Bibliotheken, APIs und Werkzeuge

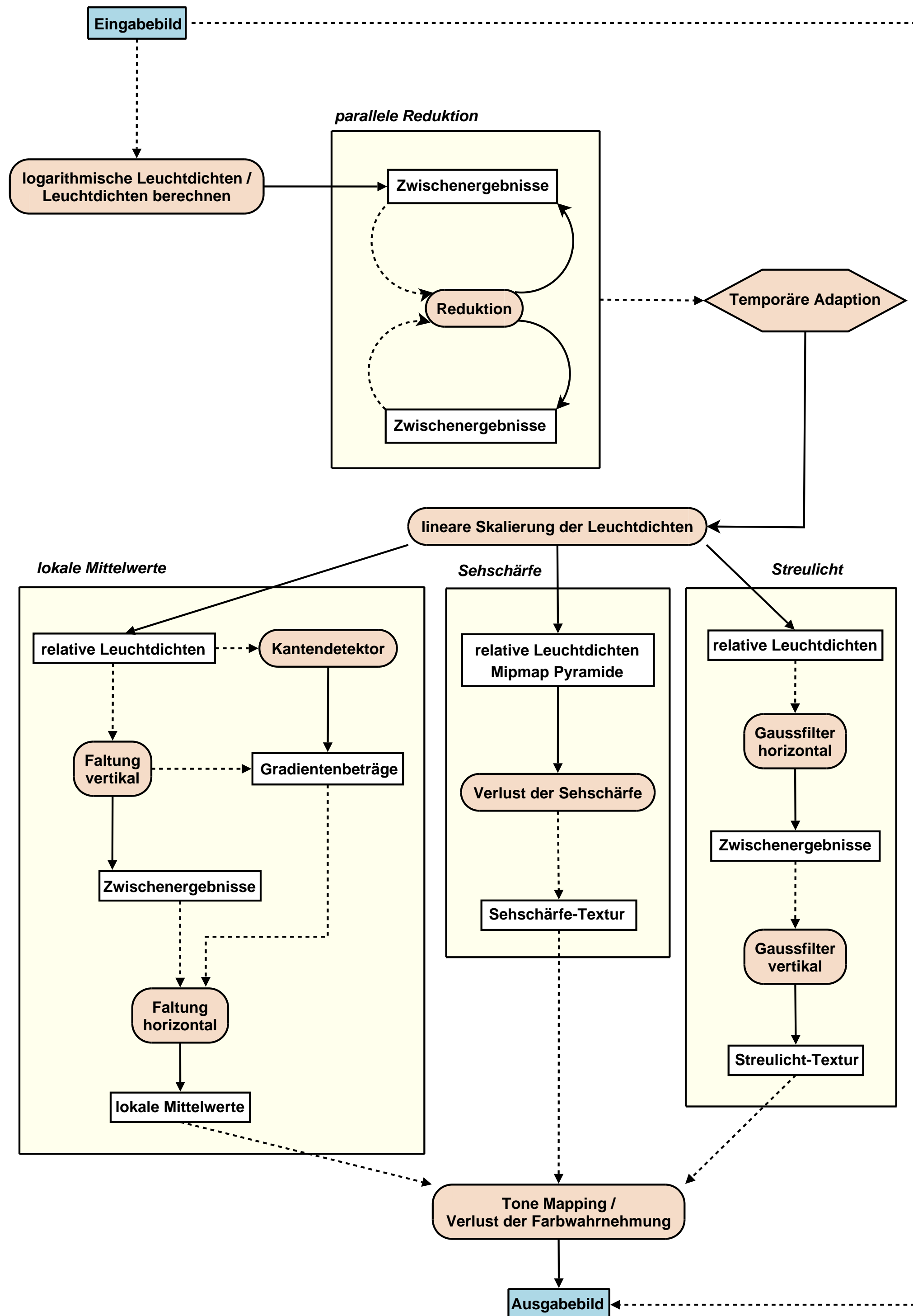
Während der Implementationsphase kamen verschiedene Bibliotheken, APIs und Werkzeuge zum Einsatz. Als Programmiersprache wurde C++ genutzt. Dabei wurde die Entwicklungsumgebung „Microsoft Visual Studio 2005“ unter Windows XP verwendet. Als Graphik-API kam OpenGL 2.0 zum Einsatz. Die Shader wurden allesamt in der OpenGL Shading Language (GLSL) implementiert. Da ebenfalls zahlreiche OpenGL-Erweiterungen genutzt wurden, kam die plattformunabhängige *GLEW*-Bibliothek [15] zum Einsatz. Zudem wurde die *GLUT*-Bibliothek [16] genutzt. Die graphische Benutzeroberfläche (GUI) des Testprogramms „TMView“ wurde mit der ebenfalls plattformunabhängigen Bibliothek *GLUI* programmiert. Während der Implementation wurde auf ein selbsterstelltes *Framework* zurückgegriffen, das zu einem späteren Zeitpunkt ausführlicher vorgestellt wird. Dabei ist die Mehrzahl der Klassen des Frameworks plattformunabhängig konzipiert. Sie sollten nach kleineren Modifikationen unter anderen Architekturen lauffähig sein. Die Kommentierung des Quellcodes konnte durch das Werkzeug *doxygen* [17] automatisiert werden. Zum Einlesen und Parsen von XML-Dateien, in denen Geometrie- und Materialdaten der verschiedenen 3D-Szenen gespeichert sind, wurde eine Bibliothek [18] von Frank Vanden Berghen genutzt. Das Einlesen verschiedener Bilddaten für Texturen wurde zudem durch die *DevIL*-Bibliothek [19] unterstützt. Diese Bibliothek ist seit Version 1.6.8 RC1 in der Lage, Bilder im Radiance HDR-Format einzulesen und war deshalb für diese Arbeit besonders hilfreich.

### 7.3.2 Das *Framework*

Für die Implementation des Tone-Mapping-Verfahrens konnte auf ein objektorientiertes Framework zurückgegriffen werden. Das Framework entstand bereits im Hauptstudium während der Implementationsphase im Rahmen der Studienarbeit. Es beinhaltet Klassen und Methoden, die für verschiedene Aufgaben der Computergraphik nützlich sind. Das Framework wurde während des Studiums gepflegt und erweitert. Es ist modular aufgebaut, wobei die Klassen logisch auf mehrere

[15] http://glew.sourceforge.net/
[16] http://www.xmission.com/˜nate/glut.html
[17] http://www.doxygen.org
[18] http://iridia.ulb.ac.be/ fvandenb/tools/xmlParser.html
[19] http://openil.sourceforge.net/

**Abbildung 35:** Konzeptionelle Gesamtübersicht für den Tone Mapper



Namensräume verteilt sind. Jeder Namensraum kapselt die Funktionalität für eine bestimmte Aufgabe oder ein Aufgabengebiet. Zur Implementation des Tone Mappers wurden neue Namensräume geschaffen oder bereits bestehende durch das Hinzufügen weiterer Klassen verfeinert. Im Folgenden werden der Aufbau des Frameworks und die darin enthaltenen Klassen sowie ihre Funktionalität kurz beschrieben.

Der oberste Namensraum *framework* enthält alle weiteren Namensräume. Zudem sind im obersten Namensraum die Klassen Clock, FPSCounter, Logger und Camera abgelegt. Die Klasse Clock realisiert eine einfache Uhr für Zeitmessungen. Diese Funktionalität wird unter anderem in der Klasse FPSCounter für Leistungsmessungen und Zeitnahmen verwendet. Die Klasse Logger ermöglicht ein Aufzeichnen benutzerdefinierter Textmeldungen und ist vor allem in der Entwicklungsphase als zusätzliches Debugwerkzeug hilfreich. Dabei sind die Klassen Logger und Clock als *Singletons* implementiert und besitzen zur Laufzeit nur jeweils eine Instanz. Im obersten Namensraum ist zusätzlich die Klasse Camera implementiert. Sie stellt die Funktionaltät einer Kamera zur Navigation in einer 3D-Szene zur Verfügung.

Im Namensraum math sind die Klassen Vector2<T>, Vector3<T> für Vektorarithmetik und Matrix4x4<T> für Matrizenalgebra abgelegt. Diese Klassen sind unabhängig vom Datentyp als *Templateklassen* realisiert. So können Fließkommaberechnungen mit einfacher (float) oder doppelter Genauigkeit (double) durchgeführt werden. Zusätzlich ist die Klasse Quaternion zur Berechnung von Quaternionenalgebra im gleichen Namensraum abgelegt. Diese Funktionalität wird von der Kameraklasse Camera genutzt, um die Kameratransformationen berechnen zu können.

Im Namensraum *graphics* ist unter anderem die Klasse Image realisiert. Diese Klasse bietet eine einfache Funktionalität zum Einlesen verschiedenster Bildformate, die beispielsweise als Texturen genutzt werden können. Zur Repräsentation von RGB- beziehungsweise RGBA-Farbtupel sind die beiden Klassen Color3<T> und Color4<T> erstellt. Auch hier sind verschiedene Datentypen durch Templateklassen möglich. Der Namensraum *graphics* beinhaltet die beiden Namensräume *gpuservices* und *effects*.

Der Namensraum *gpuservices* enthält Klassen für graphikhardwarespezifische Anwendungen. Dazu gehört die Verwaltung von OpenGL-Texturobjekten durch die Klasse TextureManager. Sie verwendet intern die Klasse Image und bietet Funktionalität für das Einlesen von Bilddaten zur Verwendung als Texturen. Die Verwaltung von Vertex- und Fragmentshadern in der *OpenGL Shading Language* ist durch die Klasse ShaderManager realisiert. Die Render-To-Texture-Mechanismen von OpenGL 2.0 sind in der Klasse FramebufferManager gekapselt. Sämtliche Manager-Klassen werden zur Laufzeit nur einmal instanziiert und sind als Singletons implementiert.

Der Namensraum *gpuservices* umfasst zusätzlich den Namensraum *gpgpu*. In diesem Namensraum sind verschiedene Klassen und Funktionalitäten für Algorithmen auf der Graphikhardware realisiert. Die Klassen GaussianBlurUtilities

und GaussianBlur enthalten Methoden und Attribute zur Erzeugung und Verwaltung von Shadern, die für Gauss-Konvolutionen auf Texturdaten genutzt werden können. Dabei ist es möglich, Shader für verschiedene Kernelgrößen und Standardabweichungen dynamisch zu generieren. Die Klasse KawaseFilter implementiert den in Kapitel 5.3 vorgestellten Binomial-Filter nach Kawase [Mas03] auf der Graphikhardware. Weiterhin ist durch die Klasse SobelFilter der Sobel-Operator zur Kantendetektion auf der Graphikhardware implementiert. Die Klasse GPUReduction realisiert generische Reduktionen für ein-, drei- oder vierkomponentige Texturen aus Kapitel 5.2 mit Unterstützung durch die Graphikhardware. Die Klasse GPUHistogram beinhaltet Methoden zur Erzeugung von Histogrammen für die Messung von Leuchtdichteverteilungen direkt auf der Graphikhardware. Die Klasse GPGPUUtilities kapselt einige einfache Hilfsroutinen, die von den anderen Klassen genutzt werden.
Im Namensraum *effects* unter *graphics* ist die Kernfunktionalität des Tone Mappers implementiert. Die Klasse PerceptualTonemapper ist im Rahmen dieser Arbeit besonders interessant und wird im nächsten Kapitel betrachtet. Weiterhin enthält der Namensraum die Klassen WorldMesh und SkySphere. Damit können unter anderem XML-Datensätze für vorgefertigte 3D-Umgebungen eingeladen und dargestellt werden. Abbildung 36 zeigt eine Übersicht des Frameworks mit den einzelnen Namensräumen und Abhängigkeiten in einem UML-Klassendiagramm.

Aufgrund des modularen Aufbaus des Frameworks ließ sich der Tone Mapper relativ leicht integrieren. Während der Implementierung der Klassen und Methoden zur Darstellung der 3D-Szenen konnte zudem auf die bereits vorhandenen und bewährten Klassenbibliotheken des Frameworks zurückgegriffen werden.

### 7.3.3 Die Klasse **PerceptualTonemapper**

Die Realisierung der Kernfunktionalität des Tone Mappers ist durch die Klasse PerceptualTonemapper gegeben, die als UML-Klassendiagramm in Abbildung 37 dargestellt ist.

Zur Steuerung des Tone-Mapping-Verfahrens sind eine Vielzahl von Parametern notwendig, was dazu führt, dass die Klasse PerceptualTonemapper eine große Zahl an Attributen beinhaltet. Ein Ziel während der Designphase des Frameworks war es, die Schnittstellen der einzelnen Klassen möglichst schmal und konsistent zu halten. Bei dem Design der Klasse PerceptualTonemapper wurde ebenso vorgegangen. Die einzelnen Attribute der Klasse lassen sich über wenige, allgemeine Zugriffsmethoden (get/set) auslesen oder setzen. Mit Hilfe des speziellen Aufzählungstypen PT_PARAM_TYPE werden dabei die jeweiligen Attribute über einen Parameter in den Zugriffsmethoden eindeutig unterschieden. Durch diese Vorgehensweise musste nicht für jedes Attribut eine entsprechende get- und set-Zugriffsmethode geschrieben werden. So wird die Wartbarkeit und Erweiterbarkeit der Klasse erhalten, während die Schnittstelle relativ schmal bleibt. In Tabelle 2 ist die Schnittstelle der Klasse im Detail dokumentiert.

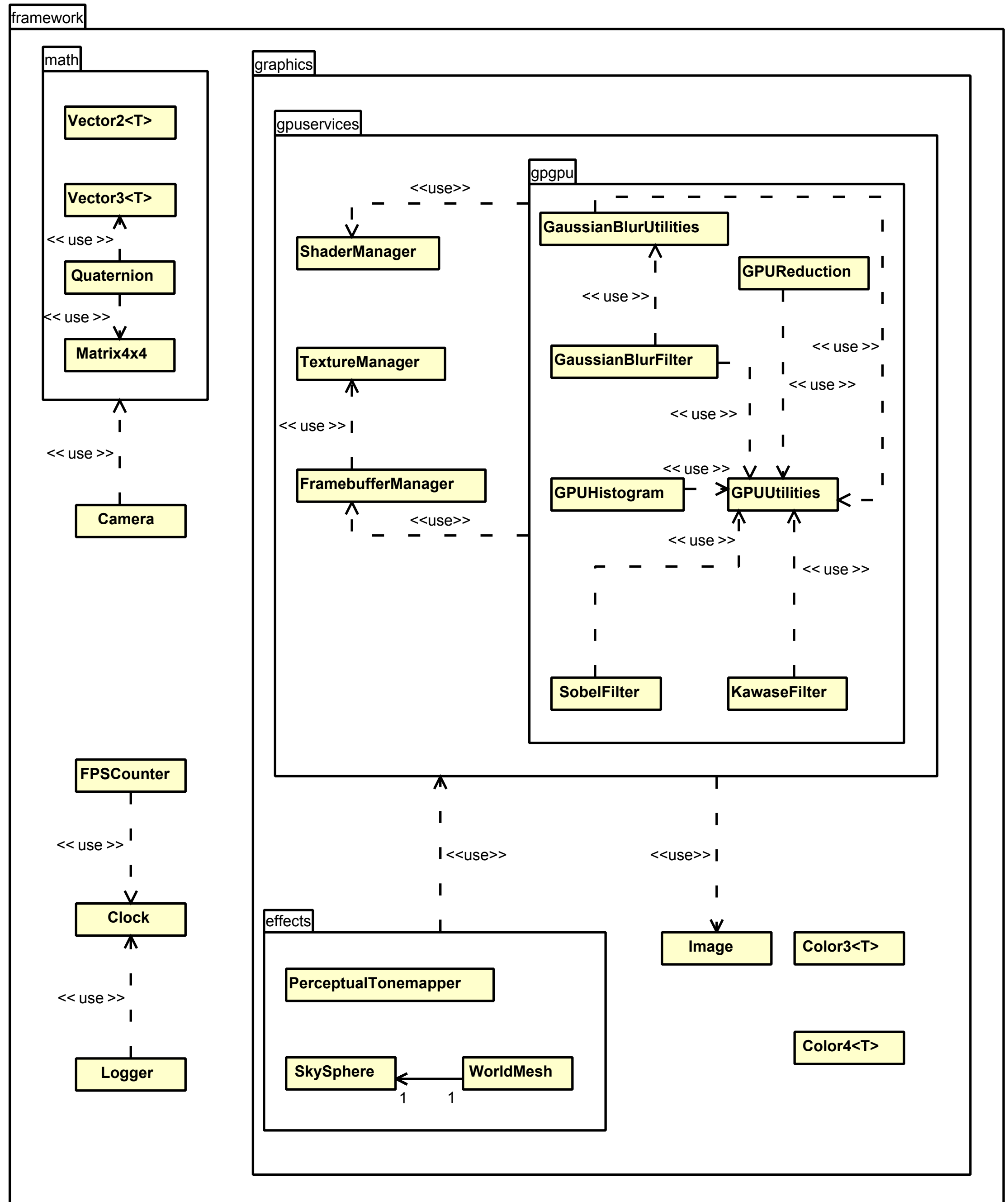



**Abbildung 36:** Ein UML-Klassendiagramm des gesamten *Framework*

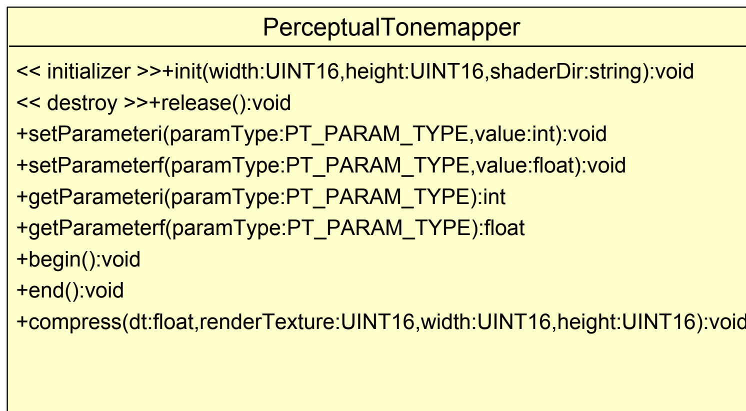



**Abbildung 37:** Die Klasse `PerceptualTonemapper`

| Methode | Beschreibung |
|---:|---|
| `init` | Initialisiert die Ressourcen des Tone-Mapping-Operators. Diese Methode muss vor der ersten Verwendung aufgerufen werden. |
| `getParameter*` | Erlaubt das Auslesen eines Parameters. |
| `setParameter*` | Erlaubt das Setzen eines Parameters. |
| `begin` | Leitet das Rendering in die interne Fließkommatextur des Tone Mappers um. |
| `end` | Deaktiviert die Umleitung. |
| `compress` | Führt den Tone-Mapping-Prozess aus. |

**Tabelle 2:** Schnittstelle der Klasse `PerceptualTonemapper`

Große Teile des Tone-Mapping-Verfahrens sind über externe Fragmentshader realisiert, die im folgenden Kapitel aufgeführt werden.

### 7.3.4 Fragmentshader

Im Folgenden sind die wichtigsten Fragmentshader des Tone-Mapping-Verfahrens in der OpenGL Shading Language aufgeführt. Darüber hinaus ist jedem Shader eine kurze Funktionsbeschreibung beigefügt.
Der Fragmentshader `preReduction.fs` in Listing 1 bereitet die Daten für die nachfolgende parallele Reduktion auf der Graphikhardware vor. Dabei wird für jeden RGB-Pixel der HDR-Textur die logarithmische Leuchtdichte berechnet und in den Rotkanal einer RGB-Fließkommatextur geschrieben. Damit zusätzlich die maximale und minimale Leuchtdichte des Bildes im nachfolgenden Reduktionsschritt berechnet werden kann, werden außerdem die Standardleuchtdichten in die Grün- und Blaukanäle der Textur kopiert.

**Listing 1:** `preReduction.fs`

```
// contains original hdr RGB-pixels
uniform samplerRect hdrTexture;

// Y component of XYZ conversion matrix;
// for calculating luminance from a rgb pixel via a dot product
uniform vec4 yVec;

// small offset to avoid log(0)
const float DELTA = 0.00001;

void main(void) {
        // calculate pixel luminance
        float pixelLum = dot(yVec, textureRect(hdrTexture, vec2(gl_TexCoord[0].st)));

        // calculate log luminance
        float logLum = log(DELTA + pixelLum);

        // write log luminance and luminance to rgb components
        gl_FragColor = vec4(logLum, pixelLum, pixelLum, 0.0);
}
```

Der Fragmentshader `calcLuminanceData.fs` in Listing 2 führt die eigentliche Reduktion durch. Dabei wird die durchschnittliche logarithmische Leuchtdichte durch das Aufsummieren der einzelnen logarithmischen Leuchtdichten berechnet. Der abschließende Normierungsschritt wird nicht durch einen Shader vorgenommen, sondern erfolgt im Hauptprogramm auf der CPU. Zusätzlich werden die maximale und minimale Leuchtdichten ermittelt. In jedem Reduktionsschritt werden die Ergebnisse in eine RGB-Fließkommatextur geschrieben.

**Listing 2:** `calcLuminanceData.fs`

```
// holds log luminance in red channel and luminance values in blue
// and green channel
uniform samplerRect reductionSet;

void main(void)
{
        // get four reduction samples
        vec4 a = textureRect(reductionSet, gl_TexCoord[0].st);
        vec4 b = textureRect(reductionSet, gl_TexCoord[0].st + vec2(0, 1));
        vec4 c = textureRect(reductionSet, gl_TexCoord[0].st + vec2(1, 0));
        vec4 d = textureRect(reductionSet, gl_TexCoord[0].st + vec2(1, 1));

        // sum up all four log luminance values
        float logLum = a.r + b.r + c.r + d.r;

        // calculate maximum luminance
        float maxLum = max(max(a.g, b.g), max(c.g, d.g));

        // calculate minimum luminance
        float minLum = min(min(a.b, b.b), min(c.b, d.b));

        // write to output buffer
        gl_FragColor = vec4(logLum, maxLum, minLum, 0.0);
}
```

Der Fragmentshader `relativeLuminance.fs` in Listing 3 berechnet die relativen Leuchtdichten nach Formel (43). Hierzu werden die Leuchtdichten des Originalbildes entsprechend skaliert, wobei die durchschnittliche logarithmische Leuchtdichte und der Key-Wert als globale Parameter im Shader verwendet werden. Die relativen Leuchtdichten werden in eine RGB-Fließkommatextur geschrieben. Die Textur kann später zum Aufbau der Mipmap-Pyramide vom Algorithmus zur Berechnung der Sehschärfe genutzt werden.

**Listing 3:** `relativeLuminance.fs`

```
1  // contains original hdr RGB-pixels
2  uniform samplerRect hdrTexture;
3
4  // log average luminance
5  uniform float avgLum;
6
7  // reinhard key value
8  uniform float key;
9
10 // Y component of XYZ conversion matrix;
11 // for calculating luminance from a rgb pixel via a dot product
12 uniform vec4 yVec;
13
14 void main(void) {
15         // get pixel color
16         vec4 hdrPixel = textureRect(hdrTexture, gl_TexCoord[0].st);
17
18         // calculate pixel luminance
19         float pixelLum = dot(hdrPixel, yVec);
20
21         // perform global scaling; calculate relative luminance
22         float relativeLum = (key / avgLum) * pixelLum;
23
24         // write to output buffer
25         gl_FragColor = vec4(relativeLum, relativeLum, relativeLum, 1.0);
26 }
```

Der Fragmentshader `createAcuityMap.fs` in Listing 4 berechnet aus der Mipmap-Pyramide der relativen Leuchtdichten eine Textur, die zur Simulation des Sehschärfeverlustes genutzt werden kann. Abhängig von der relativen Leuchtdichte und der globalen logarithmischen Leuchtdichte wird pro Fragment ein Texel aus der entsprechenden Mipmap-Stufe gewählt und in eine Render-Textur geschrieben.

**Listing 4:** `createAcuityMap.fs`

```
1  // contains relative luminances in a mipmap pyramide
2  uniform sampler2D acuityMipmaps;
3
4  // maximum mipmap level
5  uniform float maxLevel;
6
7  // log average luminance of the scene
8  uniform float avgLum;
9
10 // needed for calculation of log10
```

```
11 const float tmp = log(10.0);
12
13 void main(void)
14 {
15         // get relative luminance from mipmap base level
16         float relLum = texture2D(acuityMipmaps, gl_TexCoord[0].st, 0.0);
17
18         // calculate max cycles per degree for current pixel using Shaler's data
19         float RF = (17.25 * atan(1.4 * (log(1e4 * relLum * avgLum) / tmp) + 0.35) + 25.72);
20
21         // calculate a suitable mipmap level for the current pixel luminance also taking
22         // into account the overall scene log average luminance
23         float level = maxLevel - maxLevel * (RF / 51.5);
24
25         // make sure the level is in a valid range
26         level = clamp(level, 0.0, maxLevel);
27
28         // write pixel from mip map level to acuity map
29         gl_FragColor = texture2D(acuityMipmaps, gl_TexCoord[0].st, level);
30 }
```

Der Fragmentshader `SobelFilter.fs` in Listing 5 implementiert den Sobel-Operator auf der GPU. Dabei werden zunächst die Gradientenwerte für die horizontale und vertikale Richtung berechnet. Abschließend werden die beiden Gradientenwerte zu dem Gradientenbetrag zusammengefasst und in eine einkomponentige Render-Textur geschrieben. Über den Parameter `edgeScale` können die Gradientenbeträge global skaliert werden, um die einzelnen Werte hervorzuheben oder zu unterdrücken.

**Listing 5:** `sobelFilter.fs`

```
1 // input texture
2 uniform samplerRect texture;
3
4 // edge scaling
5 uniform float edgeScaling;
6
7 void main(void)
8 {
9         // get all required samples
10        vec4 tl = textureRect(texture, gl_TexCoord[0].st + vec2(-1, -1));
11        vec4 tr = textureRect(texture, gl_TexCoord[0].st + vec2(+1, -1));
12        vec4 br = textureRect(texture, gl_TexCoord[0].st + vec2(+1, +1));
13        vec4 bl = textureRect(texture, gl_TexCoord[0].st + vec2(-1, +1));
14        vec4 l = textureRect(texture, gl_TexCoord[0].st + vec2(-1, 0));
15        vec4 r = textureRect(texture, gl_TexCoord[0].st + vec2(+1, 0));
16        vec4 b = textureRect(texture, gl_TexCoord[0].st + vec2(0, +1));
17        vec4 t = textureRect(texture, gl_TexCoord[0].st + vec2(0, -1));
18
19        // calculate x-gradient
20        vec4 gx = (-tl -2.0 * l -bl) + (tr +2.0 * r + br);
21
22        // calculate y-gradient
23        vec4 gy = (-tl -2.0 * t -tr) + (bl +2.0 * b + br);
24
25        // calculate gradient magnitude approx. vec4 gm = sqrt(gx * gx + gy * gy);
26        vec4 gm = length(gx) + length(gy);
27
```

```
28          // scale result and write to buffer
29          gl_FragColor = edgeScaling * gm;
30  }
```

Der Fragmentshader `edgePreservingFilter5x5.fs` in Listing 6 implementiert den bereits in Kapitel 7.2.2 vorgestellten kantenerhaltenden Filter für eine 5x5 Filtergröße. Dabei können über verschiedene Parameter leichte Kontrastverbesserungen herbeigeführt werden. Als zusätzliche Optimierung wird über einen globalen Schwellwert über den maximalen Gradientenbetrag entschieden, ob der Leuchtdichtewert eines Pixels unverändert bleiben soll. In diesem Fall ist es nicht notwendig die Schleife über die Pixelnachbarn zu durchlaufen. So können weitere Texturzugriffe eingespart werden, wodurch die Ausführungsgeschwindigkeit insgesamt erhöht wird. Allerdings ist hierbei ein bedingter Sprung durch eine IF-Anweisung notwendig, der ebenfalls einige Zyklen kostet. Eine weitere Optimierungsstrategie wäre durch ein manuelles Aufrollen der Schleife möglich, wodurch ein redundanter Texturzugriff für den Gradientenbetrag des Zentrumspixels eingespart werden könnte.

**Listing 6:** `edgePreservingFilter5x5.fs`

```
1   // holds the input pixels
2   uniform samplerRect texture;
3
4   // holds the gradient strengths
5   uniform samplerRect edges;
6
7   // determines the convolution direction
8   uniform vec2 dir;
9
10  // max gradient threshold, where smoothing occurs
11  uniform float maxGrad;
12
13  // max deviation allowed during smoothing
14  uniform float maxDeviation;
15
16  // min deviation allowed during smoothing
17  uniform float minDeviation;
18
19  // half mask size of filter
20  const int hwidth = 2;
21
22  // pi constant
23  const float PI = 3.14159265;
24
25  void main(void)
26  {
27          // holds the convolution sum
28          vec4 sum = vec4(0.0, 0.0, 0.0, 0.0);
29
30          // sum of weights
31          float weights = 0.0;
32
33          // get pixel gradient strength
34          vec4 grad = textureRect(edges, gl_TexCoord[0].st);
35
36          // keep strong edges intact and try to avoid the loop
```

```
37         if (grad.r < maxGrad)
38         {
39                 // loop over pixel neighbours in a given direction
40                 for (int i = -hwidth; i <= hwidth; i++)
41                 {
42                         // calculate texture coordinate for current direction
43                         vec2 tc = gl_TexCoord[0].st + dir * vec2(i, i);
44 
45                         // get pixel gradient strength
46                         vec4 grad = textureRect(edges, tc);
47 
48                         // calculate deviation based on pixel gradient strength
49                         float scaling = max((maxGrad - grad) / maxGrad, 0.0);
50                         float deviation = max(minDeviation, scaling * maxDeviation);
51                         float sd = pow(deviation, 2.0);
52 
53                         // calculate gaussian weight dynamically
54                         float weight = (1.0 / (PI * sd)) * exp(-(float)(i * i) / sd);
55 
56                         // sum up weighted samples
57                         sum += weight * textureRect(texture, tc);
58 
59                         // sum up weights
60                         weights += weight;
61                 }
62                 // scale result and write to output buffer
63                 gl_FragColor = sum / weights;
64         } // simple pass through; preserve edge
65         else gl_FragColor = textureRect(texture, gl_TexCoord[0].st);
66 }
```

Der Fragmentshader `glare7x7.fs` in Listing 7 implementiert einen generischen separierbaren 7x7 Filter, der zur Simulation des Streulichts aus Kapitel 7.2.5 genutzt wird. Über den Parameter `dir` kann zwischen horizontaler und vertikaler Faltung pro Renderdurchlauf umgeschaltet werden. Die Filterkoeffizienten können über einen weiteren `uniform` Parameter frei gewählt werden.

**Listing 7:** `glare7x7.fs`

```
1  // kernel size
2  const int kernelSize = 7;
3 
4  // half kernel size
5  const int hwidth = kernelSize / 2;
6 
7  // filter weights
8  uniform float weights[kernelSize];
9 
10 // holds input texture
11 uniform samplerRect texture;
12 
13 // direction (horizontal or vertical)
14 uniform vec2 dir;
15 
16 // sum of weights for normalization
17 uniform float weightSum;
18 
19 void main(void)
20 {
```

```
21        // holds the convolution sum
22        vec4 sum = vec4(0.0, 0.0, 0.0, 0.0);
23
24        // loop over pixels
25        for (int i = -hwidth; i <= hwidth; i++)
26        {
27                // calculate texture coordinates
28                vec2 tc = gl_TexCoord[0].st + dir * vec2(i, i);
29
30                // sum up weighted samples
31                sum += weights[i + hwidth] * textureRect(texture, tc);
32        }
33        // scale result and write to buffer
34        gl_FragColor = sum / weightSum;
35 }
```

Der Fragmentshader `tonemap.fs` in Listing 8 implementiert den Tone-Mapping-Operator der in Kapitel 7.2 ausführlich vorgestellt wurde. Die Simulation zum Verlust des Farbsehens wird direkt innerhalb dieses Shaders durchgeführt.

**Listing 8:** `tonemap.fs`

```
1  // texture containing the whole hdr spectrum
2  uniform samplerRect hdrTexture;
3
4  // local adaptation values (local mean values)
5  uniform samplerRect localAdaptationMap;
6
7  // texture containing glare values
8  uniform samplerRect glareMap;
9
10 // contains the acuity values
11 uniform samplerRect acuityMap;
12
13 // for calculating the luminances via dot product (D65 white point)
14 uniform vec4 yVec;
15
16 // scaling for scotopic effects
17 uniform vec4 scotopicScale;
18
19 void main(void) {
20        // get hdr pixel color (sum up hdr portion and blurred portion)
21        vec4 worldCol = textureRect(hdrTexture, gl_TexCoord[0].st);
22
23        // calculate luminance value from both pixel values
24        float worldLum = dot(worldCol, yVec);
25
26        // get sample from glaremap
27        vec4 glareLum = textureRect(glareMap, gl_TexCoord[0].st);
28
29        // get sample from acuitymap
30        vec4 acuityLum = textureRect(acuityMap, gl_TexCoord[0].st);
31
32        // get sample from local adaptation map
33        vec4 localLum = textureRect(localAdaptationMap, gl_TexCoord[0].st);
34
35        // compress luminance
36        float compressedLum = (acuityLum + glareLum) / (1.0 + localLum);
37
38        // calculate visual sensitivity for scotopic range
```

```
        float scotopicSens = 0.04 / (0.04 + worldLum);

        // compress color and account for loss of color sensitivity for scotopic vision
        vec4 colFinal = compressedLum * (1.0 - scotopicSens) / worldLum * worldCol;

        // create a slight color shift toward scotopicScale for scotopic vision
        // can be used to simulate a blue shift for scotopic vision
        colFinal += scotopicScale * compressedLum * scotopicSens;

        // write to output buffer
        gl_FragColor = colFinal;
}
```

# 8 Ergebnisse

Im Rahmen dieser Arbeit wurde ein Tone-Mapping-Verfahren implementiert, das die Anforderungen aus Kapitel 7.1 erfüllt. Um die Implementation evaluieren und präsentieren zu können, ist die Testumgebung „TMView“ entstanden. Die Anwendung wird in Kapitel 8.2 im Detail beschrieben. Weiterhin sind umfangreiche Leistungsuntersuchungen und Leistungsvergleiche durchgeführt worden, wobei die Ergebnisse im folgenden Teil der Ausarbeitung tabellarisch und graphisch präsentiert werden.

## 8.1 Leistungsanalysen

Die Leistungsanalysen wurde auf einem Standard-PC und unter Windows XP [20] durchgeführt. Als Graphikhardware kam eine GeForce 6800 GT (NV40) der Firma nVidia zum Einsatz [21].
Die genaue Testkonfiguration ist in Tabelle 3 aufgeführt. Als Quellen für diese Daten dienten die Herstellerseiten im Internet [nVi06a] und [Adv06]. Die Leistungsmessungen wurden innerhalb einer einfachen GLUT-Anwendung realisiert. Dadurch konnten zusätzliche Latenzen durch nicht-relevante Komponenten, wie etwa die GUI-Komponenten von „TMView“, möglichst ausgeschlossen werden. Für die Zeitmessungen wurde eine spezielle Klasse `Clock` innerhalb des Frameworks implementiert. Die Klasse kapselt die notwendige Funktionalität, um Zeitmessungen in Millisekunden durchzuführen. Intern verwendet sie eine wesentlich

| | **Testkonfiguration** |
|---|---|
| **Hauptprozessor** | AMD Athlon XP 3000+ (Barton)<br>2100 MHz realer Takt<br>400 MHz Front Side Bus |
| **Hauptplatine (Chipsatz)** | nVidia nForce2 |
| **Speicherausstattung** | 1536 MB PC-3200 DDR-400<br>Dualchannel |
| **Graphikhardware** | Nvidia Geforce 6800 GT (NV40)<br>6 Vertexshader, 16 Fragmentshader<br>256 MB GDDR3<br>256 Bit Speicherinterface<br>350 MHz Chiptakt<br>500 (1000) MHz Speichertakt<br>AGP 8x |

**Tabelle 3:** Testkonfiguration für die Leistungsmessungen

[20] Windows XP Professional, Version 2002 mit Service Pack 2
[21] ForceWare 84.21 Treiberversion

höhere Auflösung. Der *Read Time Stamp Counter* ist über den Assembler Befehl `rdtsc` nutzbar und ermöglicht hochauflösende Zeitmessungen auf Basis der Prozessortaktung. Dieses spezielle Register für Zeitnahmen ist seit der Intel Pentium Prozessorarchitektur verfügbar [Int97]. Um Außreißer in den Messwerten weitgehend zu unterbinden, wurden pro Durchlauf mehrere Zeitmessungen genommen und in einem Vektor gespeichert. Das Vektorfeld wurde schließlich sortiert, sodass ein Median als durchschnittlicher Messwert bestimmt werden konnte. Nachfolgend sind sämtliche Ergebnisse der Zeitmessungen entweder in Millisekunden oder als Bilder pro Sekunde angegeben.

### 8.1.1 Leistungsvergleich für die parallele Reduktion

In diesem Leistungsvergleich wird das Berechnungsverfahren der parallelen Reduktion auf der Graphikhardware aus Kapitel 5.2 einer traditionellen sequentiellen Berechnung auf der CPU gegenübergestellt. Dabei wurden aus einem Datensatz der Minimalwert, Maximalwert und logarithmische Durchschnittswert berechnet. Die Eingabedaten verwendeten ein 32-Bit Fließkommaformat. Die Zeitmessungen wurden für verschiedene Datensatzgrößen durchgeführt und sind in Tabelle 4 aufgeführt. In Abbildung 38 sind die Daten nochmals in graphischer Form aufbereitet.

| **Texturgröße** | **CPU [ms]** | **GPU [ms]** | **Leistung GPU zu CPU** |
|---|---:|---:|---:|
| 32 x 32 | 0,0929 | 0,2295 | 0,4 |
| 64 x 64 | 0,3710 | 0,2481 | 1,5 |
| 128 x 128 | 1,4899 | 0,3698 | 4,03 |
| 256 x 256 | 5,9705 | 0,8484 | 7,04 |
| 512 x 512 | 25,1346 | 3,0104 | 8,35 |
| 1024 x 1024 | 98,9679 | 13,0104 | 7,61 |

**Tabelle 4:** Leistungsvergleich zwischen paralleler Reduktion auf der GPU und sequentieller Berechnung auf der CPU

Die Zeitmessungen zeigen, dass eine Berechnung mit Hilfe der parallelen Reduktion auf der GPU in fast allen Fällen deutlich schneller ist als eine sequentielle Berechnung auf der CPU. Dabei sind die Geschwindigkeitsvorteile ab einer gewissen Datensatzgröße erheblich. In Abbildung 38 ist gut zu sehen, dass die CPU mit jeder Vergrößerung der Datensätze eine lineare Steigerung in der Berechnungszeit aufweist. Der Algorithmus für die parallele Reduktion auf der GPU skaliert hingegen deutlich besser.

### 8.1.2 Leistungsmessungen für Konvolutionen auf der GPU

Bei dieser Messung wurde die Leistungsfähigkeit von Bildkonvolutionen auf der GPU untersucht. Dabei sind separierbare Gaussfilter mit verschiedenen Filterradi-

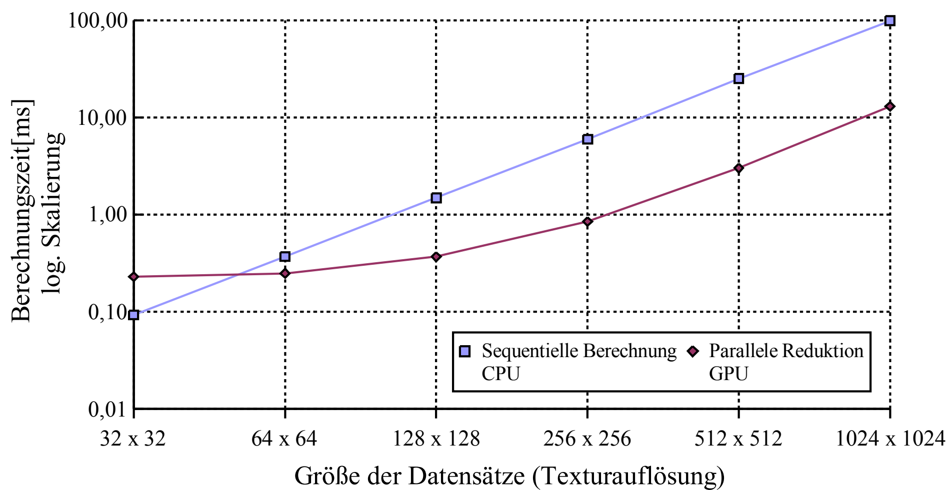



**Abbildung 38:** Leistungsmessung einer Reduktion auf der GPU im Vergleich zu einer sequentiellen Berechnung auf der CPU

en und einer Standardabweichung von $\sigma = 3.0$ verwendet worden. Die diskreten Größen der Filtermasken orientieren sich an den Profilgrößen, die Reinhards photographischer Tone Mapper in Kapitel 6.3 verwendet. Als Bilddaten diente eine Textur in einer Auflösung von $1024x768$ Pixel. Weiterhin wurde die komplette Messreihe für zwei verschiedene Texturformate durchgeführt. Hierbei kam ein 32-Bit-Fließkommaformat mit einer Komponente [22] und mit drei Komponenten [23] zum Einsatz. Die Ergebnisse der Zeitmessungen sind in Tabelle 5 aufgeführt.

| Größe der Filtermaske | 1-Komponente [Hz] | 3-Komponenten [Hz] |
|---|---:|---:|
| 3x3 | 520 | 86 |
| 5x5 | 333 | 57 |
| 7x7 | 267 | 42 |
| 11x11 | 166 | 28 |
| 17x17 | 115 | 19 |
| 27x27 | 75 | 12 |
| 43x43 | 48 | 7 |

**Tabelle 5:** Leistungsmessung für Bildkonvolutionen auf der GPU

Dabei ist interessant, dass die Konvolutionen auf dem einkomponentigen Texturformat im direkten Vergleich fast mehr als sechsmal so schnell berechnet werden,

[22] Format: GL_FLOAT_R32_NV
[23] Format: GL_FLOAT_RGB32_NV

als auf dem dreikomponentigen Format. Hier wurde aufgrund der unterschiedlichen Bandbreitenanforderungen eher ein Faktor von drei oder vier erwartet. Die bilineare Texturfilterung wird auf der verwendeten Graphikhardware für Fließkommaformate mit 32-Bit Genaugigkeit leider nicht unterstützt. Daher konnten die Optimierungen aus Kapitel 5.2 nicht angewendet werden. Die Leistungsmessungen belegen, dass eine Konvolution mit großen Filtermasken hohe Anforderungen an die Speicherbandbreite stellt. Aus diesem Grund wurde im Rahmen dieser Arbeit nach Alternativen zur Konvolution gesucht. In Abbildung 39 sind die Daten zusätzlich in graphischer Form aufbereitet.

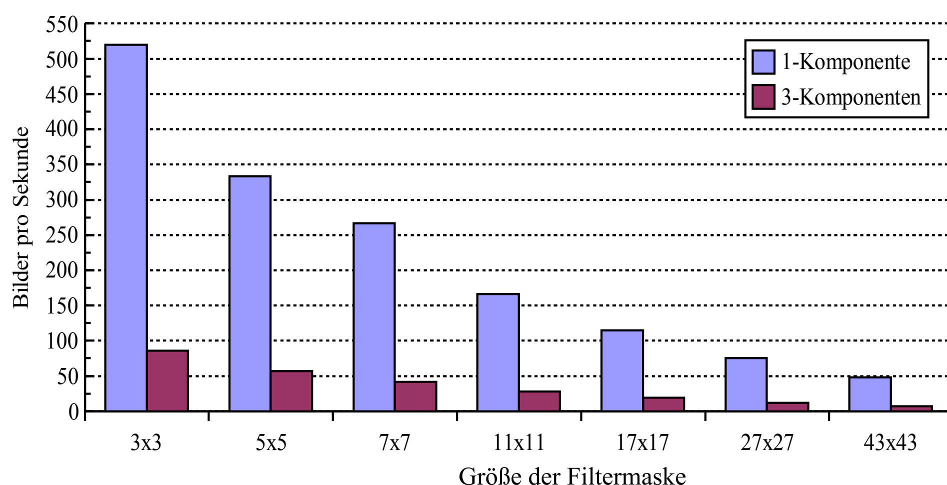



**Abbildung 39:** Leistungsmessungen für Bildkonvolutionen auf der GPU

### 8.1.3 Leistungsmessungen für das Tone-Mapping-Verfahren

In diesem Teil werden ausführliche Leistungsmessungen für das zuvor vorgestellte Tone-Mapping-Verfahren präsentiert. Zunächst wurde die Leistung des lokalen Tone-Mapping-Verfahrens mit dem globalen und lokalen Tone Mapper von Reinhard et al. aus Kapitel 6.3 verglichen. Dabei wurden für den lokalen Operator von Reinhard insgesamt acht Konvolutionen mit diskreten Filtergrößen von 3x3 bis 43x43 verwendet. Damit ein fairer Vergleich möglich war, sind bei dem eigenen Verfahren die Teile zur Simulation der menschlichen visuellen Wahrnehmung deaktiviert worden. Die Leistung der verschiedenen Verfahren wurde für verschiedene Bildauflösungen gemessen und die Ergebnisse in Tabelle 6 aufgeführt.
Der globale Tone Mapper von Reinhard kann auf der GPU effizient berechnet werden und ist für jede Bildauflösung deutlich schneller als die beiden lokalen Verfahren. Das eigene Verfahren ist im direkten Vergleich zu Reinhards lokalem Operator ebenfalls performanter. Außerdem liegt der lokale Operator von Reinhard bei einer

Auflösung von $1024x768$ bereits unterhalb der in den Anforderungen angegebenen Grenze für die Bildwiederholrate von $20Hz$. In Abbildung 40 ist die Messreihe in graphischer Form aufbereitet.

| Bildgröße | Reinhard (global) [Hz] | Reinhard (lokal) [Hz] | Eigener Operator (lokal) [Hz] |
|---|---|---|---|
| 160x120 | 653 | 277 | 543 |
| 320x240 | 595 | 127 | 379 |
| 640x480 | 408 | 36 | 119 |
| 800x600 | 336 | 23 | 78 |
| 1024x768 | 274 | 14 | 51 |

**Tabelle 6:** Leistungsvergleich verschiedener Tone-Mapping-Operatoren

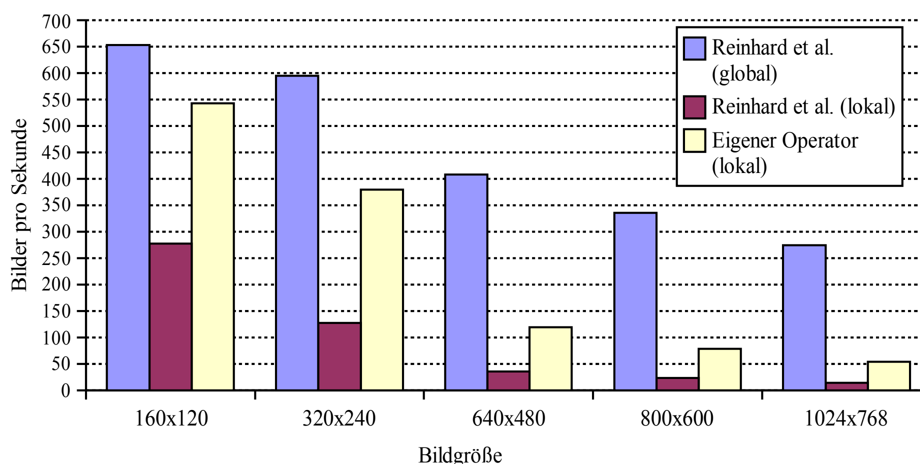



**Abbildung 40:** Leistungsvergleich für verschiedene Tone-Mapping-Operatoren

Die Teilprozesse zur Simulation der menschlichen visuellen Wahrnehmung benötigen ebenfalls ein gewisses Leistungsmaß. Dazu ist in Tabelle 7 ein Leistungsvergleich zwischen dem lokalen und dem globalen Operator des eigenen Tone-Mapping-Verfahrens aufgeführt. Für die Messungen wurden sämtliche Teile zur Simulation der menschlichen Wahrnehmung aktiviert.
In Abbildung 41 sind die tabellarischen Daten zusätzlich in graphischer Form dargestellt. In Abbildung 42 sind die einzelnen Leistungskosten für den lokalen Operator und die Teile zur Simulation der menschlichen Wahrnehmung für das Tone-Mapping-Verfahren aufgeschlüsselt. Die Kosten sind hierbei prozentual im Verhältnis zum globalen Operator ohne Simulation der menschlichen Wahrnehmung angegeben. Die Berechnungen für den lokalen Operator benötigen die größten

Leistungsressourcen. Dabei ist die Simulation für den Verlust der Farbwahrnehmung weniger rechenintensiv als die Simulation der beiden anderen Teile der menschlichen visuellen Wahrnehmung. Dies lässt sich dadurch erklären, dass die Berechnungen für den Verlust der Farbwahrnehmung wesentlich weniger Texturzugriffe als die anderen Simulationsteile benötigen.

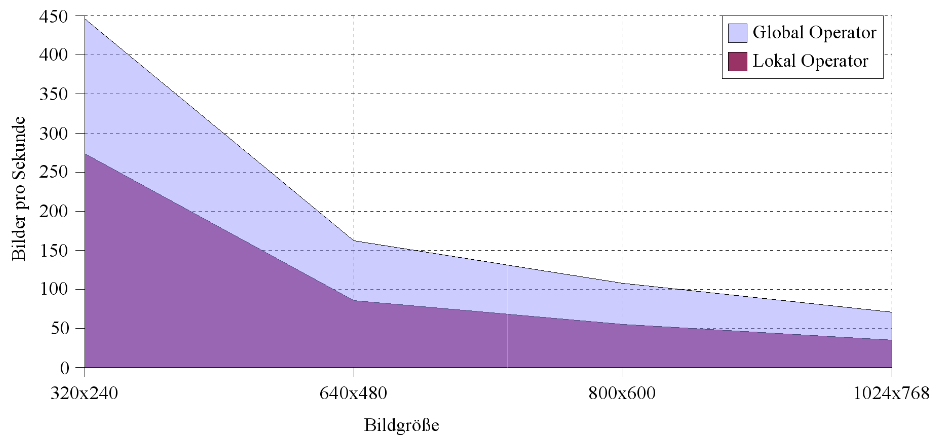



**Abbildung 41:** Leistungsmessung von globalem und lokalem Operator

| Bildgröße | Globaler Operator [Hz] | Lokaler Operator [Hz] |
|---|---:|---:|
| 320x240 | 446 | 274 |
| 640x480 | 162 | 86 |
| 800x600 | 108 | 55 |
| 1024x768 | 71 | 35 |

**Tabelle 7:** Leistungsvergleich zwischen globalem und lokalem Operator

## 8.2 Die Testumgebung „TMView“

Zur Evaluation und Präsentation des im Rahmen dieser Arbeit entstandenen Tone-Mapping-Verfahrens wurde die Testumgebung „TMView“ geschaffen. In Abbildung 43 ist ein Screenshot der Anwendung zu sehen.

Innerhalb der Anwendung können 3D-Szenarien aus einem eigenen XML-Format über einen Dateibrowser eingeladen werden. Damit das Tone-Mapping-Verfahren auch auf Einzelbildern und digitalen Photographien anwendbar ist, lassen sich ebenfalls Bilddaten im .hdr, .png, .tga oder .jpg Format einlesen. Die Navigation

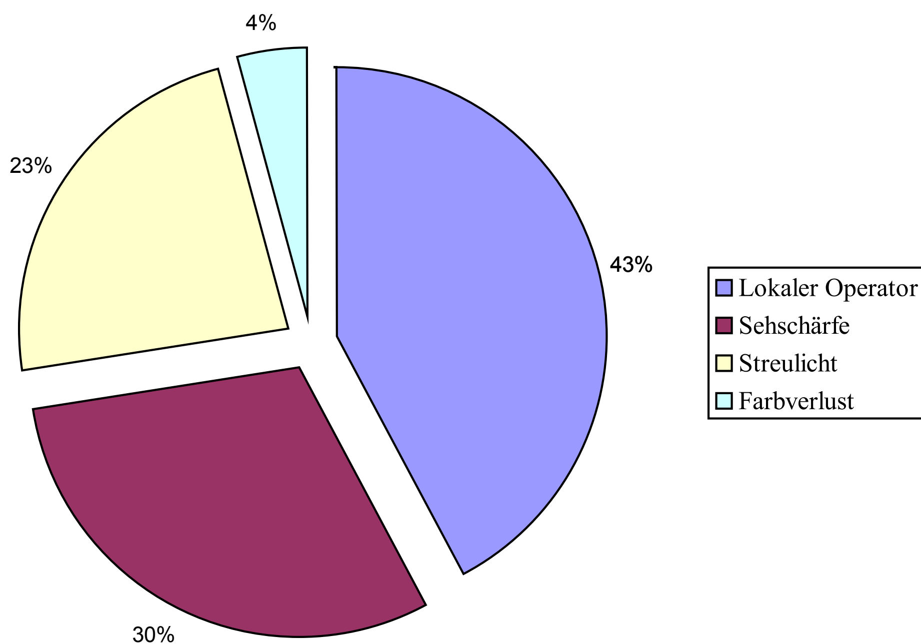



**Abbildung 42:** Prozentualer Aufwand der einzelnen Teilprozesse des Tone-Mapping-Verfahrens

innerhalb der 3D-Umgebung ist über eine einfache Maus-Tastatur-Steuerung realisiert. Neben den Ergebnisbildern nach Anwendung des Tone-Mapping-Verfahrens können zur Veranschaulichung einige interne Render-Texturen im Hauptfenster dargestellt werden. Weiterhin werden die wichtigsten statistischen Daten über Beleuchtungsverhältnisse und bildabhängige Größen angezeigt. So ist es unter anderem möglich, ein Histogramm für die Verteilung der Leuchtdichten im Hauptfenster einzublenden.
Im linken Teil der graphischen Oberfläche befindet sich ein umfangreiches Optionsfeld. Darüber lassen sich zahlreiche Einstellungen und Parameter des Tone-Mapping-Verfahrens modifizieren. So ist beispielsweise die Möglichkeit gegeben, zwischen dem lokalen und dem globalen Operator zu wechseln oder sogar die komplette Deaktivierung des Operators vorzunehmen. Die einzelnen Teile zur Simulation der menschlichen Wahrnehmung lassen sich ebenfalls einzeln an- oder abschalten. Das Ergebnisbild wird dabei interaktiv im Hauptfenster der Anwendung aktualisiert.
Die Testumgebung „TMView“ erlaubt eine einfache und intuitive Bedienung des Tone Mappers, wodurch eine zügige Evaluierung und Fehleranalyse möglich ist.

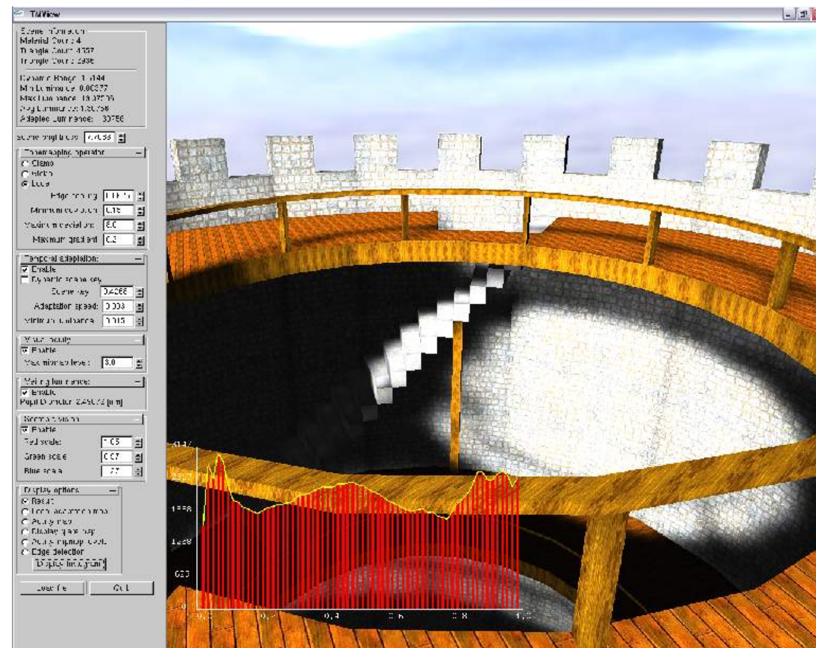


**Abbildung 43:** Die Testumgebung „TMView“

## 8.3 Integration in einer 3D-Engine

Das modulare Design und die einfachen Schnittstellen der Implementation legen es nahe, das vorgestellte Tone-Mapping-Verfahren in einer bestehenden 3D-Engine einzusetzen. Dazu wurde der Tone Mapper zu Demonstrationszwecken in Kombination mit der frei verfügbaren Open Source 3D-Engine „Irrlicht“ [24] eingesetzt.
Diese 3D-Engine ist plattformunabhängig konzipiert und bietet darüber hinaus eine Unterstützung für OpenGL. Dabei ist die Schnittstelle von der Handhabung her an das Konzept der Szenegraphen angelehnt, wobei die Realisierung einer einfachen Maus-Tastatur-Steuerung zur interaktiven Navigation in einer 3D-Szene leicht zu implementieren ist. Weiterhin gehört zum Funktionsumfang der Engine eine direkte Unterstützung für einige bekannte 3D-Formate. Dabei war besonders die Unterstützung von statischen 3D-Modellen im BSP-Format des Computerspiels Quake III für diese Arbeit von Interesse.
Die 3D-Umgebungen des BSP-Formats sind voll texturiert und verwenden für die Beleuchtung zusätzlich statische Lightmaps, die in einem 8-Bit RGB-Format vorliegen. Da der hier umgesetzte Tone Mapper hauptsächlich für die Kompression von Leuchtdichten von HDR-Umgebungen ausgelegt ist, wurden die Pixelwerte vor dem eigentlichen Tone-Mapping-Prozess skaliert.
In Abbildung 44 ist ein Screenshot eines Quake III Levels zu sehen, dessen Pixel-

[24] http://irrlicht.sourceforge.net

**Abbildung 44:** Integration des Tone-Mapping-Verfahrens in eine 3D-Engine
*3D-Level von „Tymo“ (http://lvlworld.com)*

werte mit dem hier vorgestellten Tone-Mapping-Verfahren komprimiert wurden. Dabei sind visuelle Artefakte zu sehen, die einige Bildbereiche umschließen. Die typischen Farbbänder entstehen durch die nachträgliche Skalierung, die bei dem damaligen Prozess zur Erstellung der Lightmaps nicht vorgesehen war. Dies zeigt, dass eine Verwendung von Tone Mapping bei Anwendungen, die nicht explizit während der Produktion darauf ausgelegt waren, problematisch sein kann [Car05].

# 9 Ausblick

Im Rahmen dieser Arbeit ist ein komplexes Tone-Mapping-Verfahren entstanden, das einen lokalen Tone-Mapping-Operator beinhaltet und Teile der menschlichen visuellen Wahrnehmung simulieren kann. Dabei werden große Teile zur Berechnung des Verfahrens durch die Graphikhardware unterstützt, sodass die Gesamtleistung für viele Echtzeitanwendungen ausreicht. Hierbei lässt sich der Tone Mapper leicht in bestehende Umgebungen integrieren, wodurch das Verfahren eventuell auch für die Verwendung in anderen Projekten interessant sein dürfte.
Der lokale Operator aus Kapitel 7.2.2 stellt eine effiziente Möglichkeit zur Kontraststeigerung in HDR-Bildern dar. Dabei ist der Operator bei einer Umsetzung auf der GPU im direkten Vergleich deutlich schneller als der lokale Operator von Reinhard et al. aus Kapitel 6.3. Bei hohen Auflösungen wurde jedoch beobachtet, dass der Operator von Reinhard teilweise zu größeren Kontraststeigerungen führt. Das lässt sich dadurch erklären, dass Reinhards Operator wesentlich größere Pixelnachbarschaften zur Berechnung der lokalen Mittelwerte verwendet. In dem hier umgesetzten dynamischen Glättungsfilter werden lediglich Maskengrößen von 5x5 genutzt, während Reinhard Filtermasken bis 43x43 verwendet. Für eine erhöhte Kontraststeigerung kann der dynamische Glättungsfilter jedoch ohne weiteres für größere Nachbarschaften angepasst werden, was nur einer minimalen Änderung der Implementation bedarf.
Bei einer 43x43-Filterung mit dem dynamischen Glättungsfilter konnten teilweise bessere Ergebnisse erzielt werden als mit Reinhards lokalem Operator. Aktuelle Graphikhardware bietet hohe Speicherbandbreiten und Fließkommaleistungen, wodurch eine hohe Leistung selbst bei einer Bildkonvolution mit größeren Filtermasken möglich ist. Die vor wenigen Tagen erschienene nVidia GeForce 8800 GTX kann bereits mit einen 384-Bit Speicherbus und einer Anbindung an einen schnellen GDDR3-Speicher, der mit 900 Mhz getaktet ist, aufwarten [nVi06c].
Das Verfahren des hier umgesetzten lokalen Operators verwendet eine Reihe von Parametern, durch die weitere Kontrastverbesserungen herbeigeführt werden können. Allerdings ist eine manuelle Modifikation der Parameter für Bildfolgen nur bedingt geeignet. Eine automatische Bestimmung der Parameter, etwa aus bildabhängigen Größen, könnte zukünftig untersucht werden.
Während der Entwicklung des Verfahrens hat sich gezeigt, dass es durchaus lohnenswert sein kann, neuere Features moderner Graphikhardware und 3D-APIs einzusetzen. Zukünftig könnten neue Entwicklungen zu weiteren Leistungssteigerungen und Verbesserung des bestehenden Verfahrens führen. So wird bei der aktuellen Implementation zur Simulation der Sehschärfe mit einer dreikomponentigen Render-Textur gearbeitet. Hier würde bereits eine einkomponentige Textur zur Speicherung der relativen Leuchtdichten ausreichen. Leider gab es auf der im Rahmen dieser Arbeit genutzten Graphikhardware keine ausreichende Unterstützung für den automatischen Aufbau einer Mipmap-Pyramide bei einer solchen Render-Textur. Darum musste in der Implementation eine mehrkomponentige Textur verwendet werden, die jedoch wesentlich mehr Speicherbandbreite benötigt. Die Ver-

wendung von 16-Bit Fließkommagenauigkeit in der Implementation stellt dabei einen Kompromiss dar. Zukünftig wäre eine volle Unterstützung von Mipmapping bei einkomponentigen Texturen mit 32-Bit Fließkommagenauigkeit für das hier vorgestellte Verfahren zur Simulation der Sehschärfe von Vorteil.
Wie bereits in Kapitel 7.2.4 angedeutet, können bei dem Verfahren zur Simulation der Sehschärfe Bildartefakte auftreten. Im Rahmen dieser Arbeit entstanden verschiedene Ideen und Ansätze, wie eine Reduzierung der Artefakte vorgenommen werden kann. Eine Möglichkeit ist durch eine Glättung der „harten“ Übergänge zwischen zwei Mipmap-Stufen mit einer geeigneten Filterung gegeben. Dazu kann zunächst eine Render-Textur erstellt werden, die für jeden Pixel einen Fließkommaindex in die entsprechende Mipmap-Stufe enthält. Die Textur kann dann beispielsweise mit einem Gaussfilter geglättet werden, wobei die Indizes auf harten Kanten entsprechend gemittelt werden. Eine weitere Möglichkeit bestünde darin, ein Gradientenbild zur Erkennung der Mipmap-Übergänge zu nutzen. Dazu könnte das Kantenbild, das bereits für den lokalen Operator in Kapitel 7.2.2 genutzt wurde, erneut Anwendung finden. An Bildpositionen mit hohen Gradientenwerten könnten die Artefakte durch eine geeignete Mittelung mehrerer Leuchtdichten aus zwei benachbarten Mipmap-Stufen reduziert werden. Weiterhin könnte die Verwendung einer höherwertigen Interpolationsmethode für den Skalierungsprozess der Mipmap-Stufen zu einer Reduktion der Bildartefakte führen.
Für die verschiedenen Teile zur Simulation der menschlichen visuellen Adaption wäre es zukünftig interessant, andere empirische Modelle zu nutzen. So existieren in der Literatur zur Simulation der temporären visuellen Adaption komplexe Modelle, die einen realistischen Zeitverlauf für die Adaption an die verschiedenen Beleuchtungsverhältnisse beschreiben [IFM05]. Die beiden im Rahmen dieser Arbeit enstandenen Verfahren zur Simulation der Sehschärfe und der Blendeffekte könnten ebenfalls von einer soliden empirischen Basis profitieren.

# Literaturverzeichnis


[Adv06] ADVANCED MICRO DEVICES: *AMD Athlon XP*. Website. 2006. – http://www.amd.com

[BB04] BARTELS, Rut ; BARTELS, Heinz: *Physiologie*. 7. Urban & Fischer, 2004

[BVGK04] BARLADIAN, B.Kh. ; VOLOBOI, A.G. ; GALAKTIONOV, V.A. ; KOPYLOV, E.A.: An Effective Tone Mapping Operator for High Dynamic Range Images,. In: *Programming and Computer Software* 30 (2004), S. 266–272

[BW06] BERTUCH, Manfred ; WEINER, Laurenz: Kronräuber - ATIs 3D-Chip Radeon 1900 mit 48 Pixel-Shadern. In: *ct* (2006), Nr. 4, S. 70–71

[Car05] CARSTEN WENZEL: *Far Cry and DirectX*. Game Developers Conference Presentation Slides. 2005. – www.ati.com/developer/gdc/D3DTutorial08_FarCryAndDX9.pdf

[CDPS03] COMBA, Joao L. D. ; DIETRICH, Carlos A. ; PAGOT, Christian A. ; SCHEIDEGGER, Carlos E.: Computation on GPUs: From a Programmable Pipeline to an Efficient Stream Processor. In: *Revista de Informática Teórica e Aplicada* 10-1 (2003), Nr. ISSN 0103-4308, S. 41–70

[DCWP02] DEVLIN, Kate ; CHALMERS, Alan ; WILKIE, Alexander ; PURGATHOFER, Werner: Tone Reproduction and Physically Based Spectral Rendering. In: *State of the Art Reports,* Eurographics Symposium on Rendering, 2002, S. 101–123

[DD00] DURAND, Frédo ; DORSEY, Julie: Interactive Tone Mapping, Proceedings of the Eurographics Workshop on Rendering Techniques, 2000, S. 219–230

[Fah05] FAHLÉN, Markus. *Illumination for Real-Time Rendering of Large Architectural Enviroments*. 2005

[FPSG96] FERWERDA, James A. ; PATTANAIK, Sumanta N. ; SHIRLEY, Peter ; GREENBERG, Donald P.: A model of visual adaptation for realistic image synthesis. In: *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. New York, NY, USA : ACM Press, 1996, S. 249–258

[GWWH03] GOODNIGHT, Nolan ; WANG, Rui ; WOOLLEY, Cliff ; HUMPHREYS, Greg: Interactive Time-Dependent Tone Mapping Using Programmable Graphics Hardware, Eurographics Symposium on Rendering, 2003, S. 1–13

[HH06] HICK, Christian ; HICK, Astrid: *Intensivkurs Physiologie*. 5. Urban & Fischer, 2006

[Hun85] HUNT, Robert: *The Reproduction of Colour in Photography, Printing and Television*. 5. Fountain Press, 1985

[IFM05] IRAWAN, Piti ; FERWERDA, James A. ; MARSCHNER, Stephen R.: Perceptually Based Tone Mapping of High Dynamic Range Image Streams, In Proceedings of Eurographics Symposium on Rendering, 2005

[Int97] INTEL CORPORATION: *Using the RDTSC Instruction for Performance Monitoring*. Website. 1997. –
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/timers/timerreference/timerfunctions/queryperformancecounter.asp

[Jas03] JASON L. MITCHELL: *"Real-Time 3D Scene Post-processing"*. Game Developers Conference Presentation Slides. 2003. –
http://www.ati.com/developer/gdc/GDC2003_ScenePostprocessing.pdf

[KHEK76] KUWAHARA, M. ; HACHIMURA, K. ; EIHO, S. ; KINOSHITA, M.: *Digital Processing of Biomedical Images*. New York, N.Y. : Plenum Press, 1976. – 187–203 S

[KMS05] KRAWCZYK, Grzegorz ; MYSZKOWSKI, Karol ; SEIDEL, Hans-Peter: Perceptual Effects in Real-Time Tone Mapping. In: *Spring Conference on Computer Graphics 2005*. Budmerice, Slovakia : ACM, 2005

[LOPR97] LEHMANN, Thomas ; OBERSCHELP, Walter ; PELIKAN, Erich ; REPGES, Rudolf: *Bildverarbeitung für die Medizin*. Springer, 1997

[LRP97] LARSON, Gregory W. ; RUSHMEIER, Holly ; PIATKO, Christine: A Visibility Matching Tone Reproduction Operator for High Dynamic Range Scenes. In: *IEEE Transactions on Visualization and Computer Graphics* 3 (1997), Nr. 4, S. 291–306

[MA03] MORELAND, Kenneth ; ANGEL, Edward: The FFT on a GPU. In: *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, 2003. – ISBN 1–58113–739–7, S. 112–119

[Mas03] MASAKI KAWASE: *Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless)*. Game Developers Conference Presentation Slides. 2003. –
http://www.daionet.gr.jp/~masa/

[Mas04] MASAKI KAWASE: *Practical Implementation of High Dynamic Range Rendering*. Game Developers Conference Presentation Slides. 2004. –
http://www.daionet.gr.jp/~masa/

[Mem06a] MEMBERS OF THE ARCHITECTURE REVIEW BOARD (ARB): *ARB_texture_non_power_of_two*. Website. 2006. –
http://www.opengl.org/registry/specs/ARB/texture_non_power_of_two.txt

[Mem06b] MEMBERS OF THE ARCHITECTURE REVIEW BOARD (ARB): *EXT_framebuffer_object*. Website. 2006. –
www.opengl.org/registry/specs/EXT/framebuffer_object.txt

[Mül05] MÜLLER, PROF. DR. STEFAN: *Photorealistische Computergraphik*. Vorlesungsunterlagen. Sommersemester 2005. – Universität Koblenz-Landau; Institut für Computergraphik

[nVi05] NVIDIA CORPORATION: *State of NVIDIA Support for FBO*. Website. 2005. –
download.nvidia.com/developer/presentations/2005/SIGGRAPH/fbo-status-at-siggraph-2005.pdf

[nVi06a] NVIDIA CORPORATION: *GeForce 6 Series Techspecs*. Website. 2006. –
http://www.nvidia.com/object/geforce6_techspecs.html

[nVi06b] NVIDIA CORPORATION: *GeForce 7 Series Overview*. Website. 2006. –
http://www.nvidia.com/object/IO_30459.html

[nVi06c] NVIDIA CORPORATION: *GeForce 8800*. Website. 2006. –
http://www.nvidia.com/page/geforce_8800.html

[PY02] PATTANAIK, Sumanta ; YEE, Hector: Adaptive gain control for high dynamic range image display. In: *SCCG '02: Proceedings of the 18th spring conference on Computer graphics*. New York, NY, USA : ACM Press, 2002, S. 83–87

[RD05] REINHARD, Erik ; DEVLIN, Kate: Dynamic Range Reduction Inspired by Photoreceptor Physiology, IEEE Transactions on Visualization and Computer Graphics, 2005

[Reh00] REHRMANN, DR. VOLKER: *Vorlesung Digitale Bildverarbeitung*. Wintersemester 1999/2000. – Foliensammlung

[Rei02] REINHARD, Erik: Parameter Estimation for Photographic Tone Reproduction. In: *journal of graphics tools* 7 (2002), Nr. 1, S. 45–52

[RJIH04] RAMSEY, Shaun D. ; JOHNSON III, J. T. ; HANSEN, Charles: Adaptive Temporal Tone Mapping. In: *CGIM 2004*, ACTA Press, 2004

[Ros04a] ROST, Randi J.: *OpenGL Shading Language*. Addison-Wesley, 2004

[Ros04b] ROST, Randi J.: OpenGL Shading Language Master Class. In: *GLSL Master Class* (2004)

[RSSF02] REINHARD, Erik ; STARK, Michael ; SHIRLEY, Peter ; FERWERDA, James: Photographic Tone Reproduction for Digital Images, ACM Transactions on Graphics, 2002

[RWPD06] REINHARD, Erik ; WARD, Greg ; PATTANAIK, Sumanta ; DEBEVEC, Paul: *High Dynamic Range Imaging*. Addison-Wesley, 2006

[Sch98] SCHMIDT, Robert F.: *Neuro- und Sinnesphysiologie*. 2. Springer, 1998

[Sch01] SCHMIDT, Robert F.: *Physiologie kompakt*. 4. Springer, 2001

[She04] SHERWOOD, Lauralee: *Human Physiology: From Cells to Systems*. 5. Wadsworth, 2004

[Shl37] SHLAER, Simon: The Relation Between Visual Acuity and Illumination. In: *The Journal of General Physiology* 21 (1937), S. 165–188

[SHS+04] SEETZEN, Helge ; HEIDRICH, Wolfgang ; STUERZLINGER, Wolfgang ; WARD, Greg ; WHITEHEAD, Lorne ; TRENTACOSTE, Matthew ; GHOSH, Abhijeet ; VOROZCOVS, Andrejs: High dynamic range display systems. In: *ACM Trans. Graph.* 23 (2004), Nr. 3, S. 760–768. – ISSN 0730–0301

[SSZG95] SPENCER, Greg ; SHIRLEY, Peter ; ZIMMERMAN, Kurt ; GREENBERG, Donald P.: Physically Based Glare Effects for Digital Images, Proceedings of the 22nd annual conference on Computer graphics and interactive techniques, 1995, S. 326–334

[Tec05] TECHNISCHE UNIVERSITÄT DRESDEN: *Sammlung Farbenlehre*. Website. 2005. – `http://www.arch.tu-dresden.de/iggd/gl/aktuell/flyer2.pdf`

[TS97] THEWS, Gerhard ; SCHMIDT, Robert F.: *Physiologie des Menschen*. 27. Springer Verlag, 1997

[Tum99] TUMBLIN, (Jack) John E.: *Three methods of detail-preserving contrast reduction for displayed images*, Diss., 1999. – Director-Greg Turk

[Ver04] VERSCHIEDENE: *GPU Gems*. Addison-Wesley, 2004

[Ver05] VERSCHIEDENE: *GPU Gems 2*. Addison-Wesley, 2005

[WJPS+00] WANN JENSEN, Henrik ; PREMOZE, Simon ; SHIRLEY, Peter ; THOMPSON, William B. ; FERWERDA, James A.: Night Rendering / Computer Science Department, University of Utah. 2000. – Forschungsbericht

[WND98] WOO, Mason ; NEIDER, Jackie ; DAVIS, Tom: *OpenGL Programming Guide Second Edition*. Addison-Wesley, 1998