Leserstimmen zu vorangegangenen Auflagen:

WOW ... Endlich mal wieder ein Buch, das makelos ist
„So begeistert war ich schon lange nicht mehr von einem Buch. Wer schon mal ein Buch von Herrn Haberäcker gelesen hat, weiß was Ihn erwartet. Fundierte, tiefe Informationen und ein guter, klarer Schreibstil. Dieses Talent scheint auch Herr Nischwitz zu besitzen. Das komplette Buch liest sich flüssig und ist sehr gut verständlich."

M.Sc., Dipl-Inf. Dark Habits, Berlin

Ein „Muss" für Studium und industrielle Praxis!
„Dieses sehr umfassende Buch ersetzt viele andere, die sich entweder nur mit Computergrafik oder aber mit Bildverarbeitung beschäftigen. Vom Aufbau her ist es vorbildlich strukturiert."

Prof. Dr. Ulla Kirch-Prinz, FH München

Aus dem Bereich IT erfolgreich lernen

Lexikon für IT-Berufe
von Peter Fetzer und Bettina Schneider
Grundkurs IT-Berufe
von Andreas M. Böhm und Bettina Jungkunz
Java für IT-Berufe
von Wolf-Gert Matthäus
Prüfungsvorbereitung für IT-Berufe
von Manfred Wünsche
Grundlegende Algorithmen
von Volker Heun
Algorithmen für Ingenieure – realisiert mit Visual Basic
von Harald Nahrstedt
Algorithmen und Problemlösungen mit C++
von Dorina Logofătu
Grundkurs Programmieren mit Delphi
von Wolf-Gert Matthäus
Grundkurs Visual Basic
von Sabine Kämper
Visual Basic für technische Anwendungen
von Jürgen Radel
Grundkurs Smalltalk – Objektorientierung von Anfang an
von Johannes Brauer
Grundkurs Software-Entwicklung mit C++
von Dietrich May
Programmieren lernen mit Java
von Erwin Merker und Roman Merker
Grundkurs Java-Technologien
von Erwin Merker
Java ist eine Sprache
von Ulrich Grude
Middleware in Java
von Steffen Heinzl und Markus Mathes
Grundkurs Computergrafik mit Java
von Frank Klawonn
Das Linux-Tutorial – Ihr Weg zum LPI-Zertifikat
von Helmut Pils
Rechnerarchitektur
von Paul Herrmann
Grundkurs Datenbankentwurf
von Helmut Jarosch
Datenbank-Engineering
von Alfred Moos
Grundkurs Datenbankentwicklung
von Stephan Kleuker
Grundkurs Relationale Datenbanken
von René Steiner
Grundlagen der Rechnerkommunikation
von Bernd Schürmann
Netze – Protokolle – Spezifikationen
von Alfred Olbrich
Grundkurs Spracherkennung
von Stephan Euler
Grundkurs Computernetze
von Jürgen Scherff
Grundkurs Verteilte Systeme
von Günther Bengel
Methoden wissensbasierter Systeme
von Christoph Beierle und Gabriele Kern-Isberner
Grundkurs Mobile Kommunikationssysteme
von Martin Sauter
Grid Computing
von Thomas Barth und Anke Schüll
Grundkurs Codierung
von Wilfried Dankmeier
Grundkurs Wirtschaftsinformatik
von Dietmar Abts und Wilhelm Mülder
Grundkurs Theoretische Informatik
von Gottfried Vossen und Kurt-Ulrich Witt
Anwendungsorientierte Wirtschaftsinformatik
von Paul Alpar, Heinz Lothar Grob, Peter Weimann und Robert Winter
Business Intelligence – Grundlagen und praktische Anwendungen
von Hans-Georg Kemper, Walid Mehanna und Carsten Unger
Grundkurs Geschäftsprozess-Management
von Andreas Gadatsch
ITIL kompakt und verständlich
von Alfred Olbrich
BWL kompakt und verständlich
von Notger Carl, Rudolf Fiedler, William Jórasz und Manfred Kiesel
Grundkurs Mediengestaltung
von David Starmann
Grundkurs Web-Programmierung
von Günter Pomaska
Web-Programmierung
von Oral Avcı, Ralph Trittmann und Werner Mellis
Grundkurs MySQL und PHP
von Martin Pollakowski
Logistikprozesse mit SAP R/3®
von Jochen Benz und Markus Höflinger
Grundkurs SAP R/3®
von André Maassen, Markus Schoenen, Detlev Frick und Andreas Gadatsch
Controlling mit SAP®
von Gunther Friedl, Christian Hilz und Burkhard Pedell
SAP®-gestütztes Rechnungswesen
von Andreas Gadatsch und Detlev Frick
Kostenträgerrechnung mit SAP R/3®
von Franz Klenger und Ellen Falk-Kalms
Masterkurs Kostenstellenrechnung mit SAP®
von Franz Klenger und Ellen Falk-Kalms
Masterkurs IT-Controlling
von Andreas Gadatsch und Elmar Mayer
Computergrafik und Bildverarbeitung
von Alfred Nischwitz, Max Fischer und Peter Haberäcker

www.vieweg.de

Alfred Nischwitz
Max Fischer
Peter Haberäcker

Computergrafik und Bildverarbeitung

Alles für Studium und Praxis – Bildverarbeitungswerkzeuge, Beispiel-Software und interaktive Vorlesungen online verfügbar

Mit 378 Abbildungen

2., verbesserte und erweiterte Auflage

vieweg

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Höchste inhaltliche und technische Qualität unserer Produkte ist unser Ziel. Bei der Produktion und Auslieferung unserer Bücher wollen wir die Umwelt schonen: Dieses Buch ist auf säurefreiem und chlorfrei gebleichtem Papier gedruckt. Die Einschweißfolie besteht aus Polyäthylen und damit aus organischen Grundstoffen, die weder bei der Herstellung noch bei der Verbrennung Schadstoffe freisetzen.

1. Auflage 2004
Diese Auflage erschien unter dem Titel „Masterkurs Computergrafik und Bildverarbeitung".
2., verbesserte und erweiterte Auflage März 2007

Lektorat: Sybille Thelen / Andrea Broßler

Der Vieweg Verlag ist ein Unternehmen von Springer Science+Business Media.
www.vieweg.de

Umschlaggestaltung: Ulrike Weigel, www.CorporateDesignGroup.de
Druck und buchbinderische Verarbeitung: Těšínská Tiskárna, a. s.
Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier.
Printed in Czech Republic

ISBN 978-3-8348-0186-9

Vorwort

Moderne Computer sind in unserer Zeit weit verbreitet. Sie werden kommerziell und privat zur Bewältigung vielschichtiger Aufgaben eingesetzt. Als Schnittstelle zur Hardware dienen grafische Betriebssysteme, mit denen der Benutzer bequem mit dem System kommunizieren kann. Die meisten Anwendungen präsentieren sich ebenfalls in grafischer Form und begleiten so den Anwender bei der Erledigung seiner Aufgaben.

Viele benützen dabei das Computersystem wie ein praktisches Werkzeug und freuen sich, wenn alles gut funktioniert, oder ärgern sich, wenn es nicht funktioniert. Mancher wird sich aber doch fragen, was hinter der grafischen Oberfläche steckt, wie z.B. ein Computerspiel oder ein Bildbearbeitungspaket zu einer Digitalkamera gemacht wird, welche Verfahren dabei ablaufen und welche Programmiersysteme dazu verwendet werden. An diesen Personenkreis wendet sich das vorliegende Buch.

Bei der Implementierung zeitgemäßer Anwendungen nimmt die Programmierung der Computergrafik und Bildverarbeitung einen wesentlichen Teil ein. In den letzten Jahren sind diese beide Bereiche immer stärker zusammengewachsen. Dieser Entwicklung versucht das vorliegende Buch gerecht zu werden.

Der erste Teil des Buches ist der Computergrafik gewidmet. Es werden die wichtigsten Verfahren und Vorgehensweisen erläutert und an Hand von Programmausschnitten dargestellt. Als grafisches System wurde OpenGL verwendet, da es alle Bereiche abdeckt, sich als weltweiter Standard etabliert hat, plattformunabhängig und kostenlos ist, und mit modernen Grafikkarten bestens zusammenarbeitet.

Der zweite Teil befasst sich mit digitaler Bildverarbeitung, die mit ihren Verfahren die Grundlage für praktische Anwendungen von bildauswertenden und bildgenerierenden Systemen in vielen Bereichen bildet. Der Bildverarbeitungsteil ist als Zusammenfassung von zwei Büchern entstanden, die von einem der beiden Autoren vorlagen. Zunächst war geplant, diese Zusammenfassung ausschließlich über das Internet und auf CD-ROM anzubieten. Die permanente Nachfrage ließ es aber doch sinnvoll erscheinen, sie in dieses Buch zu integrieren.

Das Buch wendet sich also an Interessenten, die sich in dieses Gebiet einarbeiten und praktische Erfahrungen sammeln möchten. Deshalb wurde, soweit möglich, auf die Darstellung der oft komplizierten mathematischen Hintergründe verzichtet und häufig eine eher pragmatische Vorgehensweise gewählt. Zur Vertiefung wird das Buch durch ein Internetangebot ergänzt. Hier findet der Leser Übungsaufgaben, vertiefende Kapitel und interaktive Kurse, wie sie auch an Hochschulen angeboten werden. Außerdem wird der Internetauftritt für Korrekturen und die Versionsverwaltung verwendet.

Alfred Nischwitz, Peter Haberäcker, 3. Juni 2004

Vorwort zur 2. Auflage

Aller guten Dinge sind Drei. In diesem Sinne freuen wir uns, dass wir als dritten Co-Autor für die Weiterentwicklung dieses Buchs unseren Kollegen Prof. Dr. Max Fischer gewinnen konnten. Damit wurde der Tatsache Rechnung getragen, dass die sehr dynamischen und immer enger zusammenwachsenden Gebiete der Computergrafik und Bildverarbeitung eines weiteren Autors bedurften, um adäquat abgedeckt zu werden.

Inhaltlich wurde der Charakter der 1. Auflage beibehalten. An einer Reihe von Stellen wurden jedoch Ergänzungen und Aktualisierungen vorgenommen. So ist im Kapitel 2 ein neuer Abschnitt über „Bildverarbeitung auf programmierbarer Grafikhardware“ hinzugekommen, im Kapitel 20 wurde der neuerdings häufig verwendete „Canny-Kantendetektor“ eingefügt, und im Kapitel 34 wurde eine Anwendung des „Run-Length-Coding“ im Umfeld der Objektverfolgung in Echtzeitsystemen ergänzt. Die Literaturhinweise wurden aktualisiert und durch neue Zitate erweitert.

Weiterhin konnten zahlreiche Fehler aus der 1. Auflage korrigiert werden. Dafür sei den kritischen Lesern ganz herzlich gedankt, die sich die Mühe gemacht haben, uns zu schreiben. Ganz besonders möchten wir uns an dieser Stelle bei Fr. Dipl.-Math. Beate Mielke bedanken, die alleine für ca. $\frac{2}{3}$ aller Fehlermeldungen zuständig war.

Alfred Nischwitz, Max Fischer, Peter Haberäcker, 22. Juli 2006

Inhaltsverzeichnis

1 Einleitung 1

2 Zusammenhang Computergrafik – Bildverarbeitung 6
- 2.1 Bildverarbeitung auf programmierbarer Grafikhardware 7
- 2.2 Simulation von kameragesteuerten Geräten 10
- 2.3 Bilddatencodierung 14
- 2.4 Bildbasiertes Rendering 19

I Computergrafik 25

3 Interaktive 3D-Computergrafik 27
- 3.1 Harte Echtzeit 28
- 3.2 Weiche Echtzeit 31

4 Anwendungen interaktiver 3D-Computergrafik 32
- 4.1 Ausbildungs-Simulation 32
- 4.2 Entwicklungs-Simulation 34
- 4.3 Unterhaltung 36
- 4.4 Telepräsenz 36
- 4.5 Daten-Visualisierung 37
- 4.6 Augmented Reality 38
- 4.7 Datenübertragung 39

5 Einführung in die 3D-Computergrafik mit OpenGL 41
- 5.1 OpenGL Kurzbeschreibung 43
- 5.2 Die OpenGL Rendering Pipeline 45
  - 5.2.1 Display Listen 46
  - 5.2.2 Vertex-Operationen 46
  - 5.2.3 Pixel-Operationen 46
  - 5.2.4 Textur-Speicher 47
  - 5.2.5 Rasterisierung 47
  - 5.2.6 Fragment-Operationen 47

5.2.7 Bildspeicher 48
5.3 Die OpenGL Kommando Syntax 48
5.4 Ergänzende Literaturhinweise 49

6 Geometrische Grundobjekte **51**
6.1 3D-Modellierungsmethoden 51
6.1.1 Planare Polygone 51
6.1.2 Gekrümmte Polygone 52
6.1.3 Volumendarstellung 53
6.1.4 Konstruktive Körpergeometrie 54
6.2 Geometrische Grundobjekte in OpenGL 54
6.2.1 Vertex-Definition in OpenGL 56
6.2.2 Grafik-Primitive in OpenGL 57
6.2.3 Programmierbeispiele 59
6.3 Tipps zur Verwendung der Grafik-Primitive 74
6.3.1 Rendering-Geschwindigkeit 74
6.3.2 Vertex Arrays 75
6.3.3 Konsistente Polygon-Orientierung 76
6.3.4 Koordinaten-Berechnungen offline 76
6.3.5 Oberflächen-Tessellierung 77
6.3.6 Lücken 77
6.3.7 Hinweise zum glBegin/glEnd-Paradigma 78
6.4 Modellierung komplexer 3D-Szenarien 79

7 Koordinatensysteme und Transformationen **80**
7.1 Definition des Koordinatensystems 80
7.2 Transformationen im Überblick 81
7.3 Mathematische Grundlagen 83
7.3.1 Homogene Koordinaten 83
7.3.2 Transformations–Matrizen 84
7.4 Modell–Transformationen 86
7.4.1 Translation 86
7.4.2 Rotation 87
7.4.3 Skalierung 89
7.4.4 Reihenfolge der Transformationen 91
7.5 Augenpunkt–Transformationen 93
7.6 Projektions–Transformationen 94
7.6.1 Orthografische Projektion (Parallel–Projektion) 94
7.6.2 Perspektivische Projektion 95
7.6.3 Normierung 100
7.7 Viewport-Transformation 101
7.8 Matrizen-Stapel 103

8 Verdeckung **108**
8.1 Der z-Buffer Algorithmus 109
8.2 Die Implementierung des z-Buffer Algorithmus 112
8.3 Einsatzmöglichkeiten des z-Buffer Algorithmus 113
8.3.1 Entfernen aller Vorderteile 114
8.3.2 Höhenkarten generieren 114
8.3.3 Volumenmessung 116
8.3.4 Oberflächenmessung 117
8.3.5 Entfernungsmessung 118

9 Farbe, Transparenz und Farbmischung **119**
9.1 Das Farbmodell in OpenGL 119
9.2 Modelle der Farbdarstellung 122
9.2.1 Der RGBA–Modus 122
9.2.2 Der Farb-Index–Modus 122
9.2.3 Wahl zwischen RGBA– und Farb-Index–Modus 123
9.2.4 Farbspezifikation im RGBA–Modus 123
9.2.5 Farbspezifikation im Farb-Index–Modus 126
9.3 Transparenz und Farbmischung 127
9.3.1 Farbmischung in OpenGL 127
9.3.2 Beispiele für Farbmischungen 130
9.3.3 Transparente Texturen 136
9.3.4 3D-Probleme bei der Farbmischung 137

10 Anti-Aliasing **140**
10.1 Aliasing-Effekte 140
10.2 Gegenmaßnahmen – Anti-Aliasing 144
10.2.1 Pre-Filterungs-Methode: Flächenabtastung 144
10.2.2 Post-Filterungs-Methoden 146

11 Nebel und atmosphärische Effekte **151**
11.1 Anwendungen 151
11.2 Nebel in OpenGL 153

12 Beleuchtung und Schattierung **159**
12.1 Beleuchtungsmodelle 160
12.1.1 Physikalische Optik und Näherungen der Computergrafik 160
12.1.2 Lokale und globale Beleuchtungsmodelle 169
12.1.3 Das Standard-Beleuchtungsmodell in OpenGL 171
12.2 Schattierungsverfahren 191
12.2.1 Flat-Shading 192
12.2.2 Smooth-/Gouraud-Shading 193
12.2.3 Phong-Shading 199

12.3 Programmierbare Shader 200
12.3.1 Shading Programmiersprachen 202
12.3.2 Realisierung eines Phong-Shaders in Cg 204

13 Texturen **215**
13.1 Foto-Texturen (Image Texturing) 217
13.1.1 Spezifikation der Textur 219
13.1.2 Textur-Filter 228
13.1.3 Gauß-Pyramiden-Texturen (MipMaps) 231
13.1.4 Textur-Fortsetzungsmodus (Texture Wraps) 236
13.1.5 Mischung von Textur- und Beleuchtungsfarbe (Texture Environment) 239
13.1.6 Zuordnung von Texturkoordinaten 242
13.1.7 Einschalten des Texture Mappings 253
13.1.8 Textur-Objekte 254
13.2 Mehrfach-Texturierung (Multitexturing) 256
13.3 Projektive Texturen (Projective Texture) 260
13.4 Umgebungs-Texturen (Environment Maps) 262
13.4.1 Sphärische Texturierung (Sphere Mapping) 263
13.4.2 Kubische Texturierung (Cube Mapping) 266
13.5 Relief-Texturierung (Bump Mapping) 270
13.6 Schatten-Texturierung (Shadow Mapping) 277

14 Animationen **279**
14.1 Animation und Double Buffering 279
14.2 Animationstechniken 283
14.2.1 Bewegung eines starren Objektes – Pfadanimation 284
14.2.2 Bewegung von Gelenken eines Objektes – Artikulation 286
14.2.3 Verformung von Oberflächen – Morphing 288
14.2.4 Bewegung von Objektgruppen: Schwärme und Partikelsysteme 290

15 Beschleunigungsverfahren **293**
15.1 Szenen Graphen 296
15.2 Cull Algorithmen 299
15.2.1 Viewing Frustum Culling 301
15.2.2 Occlusion Culling 303
15.2.3 Backface Culling 307
15.2.4 Portal Culling 308
15.2.5 Detail Culling 309
15.3 Level Of Detail (LOD) 310
15.3.1 Switch LOD 312
15.3.2 Fade LOD 313
15.3.3 Morph LOD 314

15.4 Billboards 315
15.5 Multiprozessorsysteme 319
15.6 Geschwindigkeits-Optimierung 324
15.6.1 Leistungsmessung 324
15.6.2 Optimierungsmaßnahmen 329

Literatur zu Teil I **334**

II Bildverarbeitung **339**

16 Digitale Bilddaten **341**
16.1 Prinzipielle Vorgehensweise 341
16.1.1 Sensoren 341
16.1.2 Digitalisierung 342
16.1.3 Vorverarbeitung der Rohdaten 342
16.1.4 Berechnung von Merkmalen 342
16.1.5 Segmentierung des Bildes 343
16.1.6 Kompakte Speicherung der Segmente 343
16.1.7 Beschreibung der Segmente 343
16.1.8 Synthese von Objekten 343
16.1.9 Ableitung einer Reaktion 344
16.1.10 Schlussbemerkung zu Abschnitt 16.1 344
16.2 Unterabtastung und Quantisierung 344
16.3 Digitalisierung von Schwarz/Weiß-Bilddaten 348
16.4 Digitalisierung von Grautonbildern 352
16.5 Farbbilder 357
16.5.1 Farbe: Physikalische Aspekte 358
16.5.2 Farbe: Physiologische Aspekte 358
16.5.3 Das CIE-Farbdreieck 360
16.5.4 Das RGB-Farbmodell 363
16.5.5 Das CMY-Farbmodell 365
16.5.6 Das YIQ-Farbmodell 366
16.5.7 Das HSI-Farbmodell 366
16.5.8 Mathematisches Modell für Farbbilder 368
16.6 Multispektral- und mehrkanalige Bilder 370
16.7 Bildfolgen 373
16.8 Weitere mathematische Modelle für Bilder 375
16.8.1 Bilder als relle Funktionen zweier reeller Variablen 375
16.8.2 Bilder als (diskrete) Funktionen zweier diskreter Variablen 376
16.8.3 Bilder als Zufallsprozesse 376

16.9 Bildliche Reproduktion
von digitalisierten Bildern 380
16.9.1 Geräte zur Bilddarstellung 380
16.9.2 Ausdrucken der Grauwerte 381
16.9.3 Ausgabe von logischen Bildern 383
16.9.4 Zeilendruckerausgabe von Grauwertbildern 384
16.9.5 Halbtonverfahren 385
16.10 Datenreduktion und Datenkompression 388
16.11 Charakterisierung digitalisierter Bilder 389
16.11.1 Mittelwert und mittlere quadratische Abweichung 389
16.11.2 Histogramme 392
16.11.3 Entropie 394
16.11.4 Grauwertematrix (*co-occurrence*-Matrix) 396

17 Modifikation der Grauwerte **399**
17.1 Anwendungen 399
17.2 Grundlagen der Grauwerttransformation 400
17.3 Lineare Skalierung 401
17.4 Äquidensiten (*gray level slicing*) 406
17.5 Erzeugen von Binärbildern 412
17.6 Logarithmische und exponentielle Skalierung 419
17.7 Ebnen der Grauwerte 421
17.8 Grauwertskalierung mit Hochpassfilterung 426
17.9 Kalibrierung der Grauwerte 428
17.10 Berechnung einer neuen Grauwertmenge 432
17.11 Rekonstruktion des höchstwertigen Bit 434
17.12 Differenzbildung 434

18 Operationen im Ortsbereich **438**
18.1 Anwendungen 440
18.2 Grundlagen: Filteroperationen im Ortsbereich 440
18.3 Glätten der Grauwerte eines Bildes 443
18.4 Differenzenoperatoren 448
18.5 Elimination isolierter Bildpunkte 453
18.6 Elimination gestörter Bildzeilen 454
18.7 Bildakkumulation bei Bildfolgen 456

19 Mathematische Morphologie **457**
19.1 Anwendungen 457
19.2 Grundlagen: Mathematische Morphologie 457
19.3 Median Filter 461
19.4 Dilatation und Erosion im Binärbild 463
19.5 Morphologie im Grauwertbild 465

20 Kanten und Linien **469**
20.1 Anwendungen . . . 469
20.2 Grundlegendes über Kanten und Linien . . . 469
20.3 Einfache Verfahren zur Kantenextraktion . . . 474
20.4 Parallele Kantenextraktion . . . 475
20.5 Gradientenbetrag und Gradientenrichtung . . . 480
20.6 Der Canny-Kantendetektor . . . 486
20.7 Kanten und Linien mit morphologischen Operationen . . . 490
20.7.1 Extraktion des Randes von Segmenten . . . 490
20.7.2 Verarbeitung von Linien . . . 492
20.8 Skelettierung mit morphologischen Operationen . . . 492
20.9 Skelettierung mit der Euler'schen Charakteristik . . . 499
20.10 Relaxation . . . 502
20.11 Houghtransformation . . . 506
20.12 Verallgemeinerte Houghtransformation . . . 512
20.12.1 Parametrisierung der Referenzstruktur . . . 512
20.12.2 Akkumulierende Abbildung der Merkmalspunkte . . . 513
20.12.3 Auswertung des Akkumulators . . . 513
20.13 Erweiterte Houghtransformation . . . 514
20.13.1 Erweiterte Houghtransformation der Randpunkte . . . 514
20.13.2 Erweiterung auf flächenhaft gegebene Segmente . . . 519
20.14 Sequentielle Kantenextraktion, Linienverfolgung . . . 521
20.15 Abschließende Bemerkungen zu Kanten und Linien . . . 526

21 Operationen im Frequenzbereich **527**
21.1 Anwendungen . . . 527
21.2 Lineare Approximation . . . 527
21.3 Trigonometrische Approximationsfunktionen . . . 530
21.4 Diskrete zweidimensionale Fouriertransformation . . . 532
21.5 Digitale Filterung im Ortsfrequenzbereich . . . 534
21.6 Zusammenhang mit dem Ortsbereich . . . 536
21.7 Logarithmische Filterung . . . 542
21.8 Inverse und Wiener Filterung . . . 542
21.9 Diskrete, zweidimensionale Cosinustransformation . . . 543

22 Modifikation der Ortskoordinaten **547**
22.1 Anwendungen . . . 547
22.2 Grundlegende Problemstellung . . . 547
22.3 Vergrößerung, Verkleinerung . . . 548
22.4 Affine Abbildungen . . . 551

22.5 Interpolation mit Polynomen 552
22.5.1 Polynome 552
22.5.2 Ausgleichsrechnung 555
22.5.3 Beurteilung der Qualität 556
22.5.4 Vermessung der Passpunkte 559
22.6 Abschließende Bemerkungen 560

23 Szenenanalyse **561**
23.1 Einleitung, Beispiele, Merkmale 561

24 Merkmale: Grauwert und Farbe **567**
24.1 Anwendungen 567
24.2 Merkmal: Grauwert 568
24.3 Merkmal: Farbe 570
24.4 Reduktion der Farben in einem Farbbild durch Vorquantisierung 573
24.5 Indexbilder 574
24.5.1 Die Farbhäufigkeitsverteilung eines Farbbildes 576
24.5.2 Erstellen einer Farb-Look-Up-Tabelle 578
24.5.3 Abbildung der Farben des Originalbildes in die Farbtabelle 580
24.5.4 Segmentierung des Originalbildes 581
24.5.5 Segmentierung des Originalbildes mit Dithering 582
24.5.6 Unüberwachte Klassifikatoren zur Reduktion der Farben 583

25 Merkmale aus mehrkanaligen Bildern **585**
25.1 Anwendungen 585
25.2 Summe, Differenz, Ratio 585
25.3 Verknüpfung von Kanälen bei logischen Bildern 589
25.4 Die Hauptkomponententransformation 589

26 Merkmale aus Bildfolgen **597**
26.1 Anwendungen 597
26.2 Akkumulation und Differenz 598
26.3 Merkmal: Bewegung 600
26.4 Differentielle Ermittelung von Verschiebungsvektoren 603
26.5 Blockmatching 605

27 Merkmale aus Umgebungen: Texturen **611**
27.1 Anwendungen 611
27.2 Grundlagen zu Texturmerkmalen 611
27.3 Streuung (Varianz) 613

27.4 Gradient . . . 613
27.5 Kantendichte . . . 615
27.6 Autokorrelation . . . 618
27.7 Abschlussbemerkung zu den einfachen Texturmaßen . . . 618

28 Gauß- und Laplace-Pyramiden **621**
28.1 Anwendungen . . . 621
28.2 Begriffe aus der Signaltheorie . . . 622
28.3 Motivation für Gauß- und Laplace-Pyramiden . . . 623
28.4 Der REDUCE-Operator . . . 624
28.5 Der EXPAND-Operator . . . 625
28.6 Rekonstruktion des Originalbildes . . . 629
28.7 Implementierung des REDUCE-Operators . . . 632
28.8 Implementierung des EXPAND-Operators . . . 635
28.9 Frequenzverhalten und Wahl des freien Parameters a . . . 638
28.10 Anwendungsbeispiele zu den Laplace-Pyramiden . . . 642
28.10.1 Verwendung einzelner Schichten . . . 642
28.10.2 Mosaicing . . . 642
28.10.3 Multifokus . . . 645
28.10.4 Glättungsoperationen in Laplace-Pyramiden . . . 648
28.10.5 Texturen und Segmentierung . . . 648

29 Scale Space Filtering **663**
29.1 Anwendungen . . . 663
29.2 Grundlagen: Fraktale Geometrie . . . 663
29.3 Implementierung des Scale Space Filtering . . . 669
29.3.1 Ermittlung der Größe der Grundtextur . . . 672
29.3.2 Ermittlung der Gauß-Filterkerne . . . 673
29.3.3 Berechnung der Oberflächen der Grauwertfunktion . . . 675
29.3.4 Berechnung des Skalenparameters . . . 675
29.3.5 Beispiele und Ergebnisse . . . 676

30 Baumstrukturen **681**
30.1 Anwendungen . . . 681
30.2 Aufbau von Baumstrukturen . . . 681
30.3 Regionenorientierte Bildsegmentierung mit *quad trees* . . . 687

31 Segmentierung und numerische Klassifikation **696**
31.1 Grundlegende Problemstellung . . . 696
31.2 Klassifizierungsstrategien überwacht . . . 700
31.3 Klassifizierungsstrategien unüberwacht . . . 702
31.4 Überwachtes und unüberwachtes Lernen . . . 706

31.5 Der Minimum-Distance-Klassifikator 706
31.6 Maximum-Likelihood-Klassifikator 714
31.7 Der Quader-Klassifikator 719
31.8 Beurteilung der Ergebnisse 722
31.9 Ergänzungen 723

32 Klassifizierung mit neuronalen Netzen **724**
32.1 Grundlagen: Künstliche neuronale Netze 724
32.1.1 Prinzipieller Aufbau 724
32.1.2 Adaline und Madaline 725
32.1.3 Das Perceptron 730
32.1.4 Backpropagation 732
32.2 Neuronale Netze als Klassifikatoren 735
32.2.1 Verarbeitung von Binärbildern 736
32.2.2 Verarbeitung von mehrkanaligen Bildern 742

33 Segmentierung mit Fuzzy Logic **749**
33.1 Anwendungen 749
33.2 Grundlagen: Fuzzy Logic 749
33.2.1 Einführende Beispiele 749
33.2.2 Definitionen und Erläuterungen 751
33.3 Fuzzy Klassifikator 758

34 Run-Length-Coding **764**
34.1 Anwendungen 764
34.2 Run-Length-Codierung 766
34.2.1 Prinzipielle Problemstellung und Implementierung 766
34.2.2 Vereinzelung von Segmenten 769
34.2.3 Effiziente Vereinzelung mit Union-Find-Algorithmen 773

35 Einfache segmentbeschreibende Parameter **776**
35.1 Anwendungen 776
35.2 Flächeninhalt 777
35.3 Flächenschwerpunkt 777
35.4 Umfang 779
35.5 Kompaktheit 779
35.6 Orientierung 779
35.7 Fourier-Transformation der Randpunkte 781
35.8 Chain-Codierung 782
35.9 Momente 784
35.10 Euler'sche Charakteristik 790
35.11 Auswahl mit morphologischen Operationen 791
35.12 Segmentbeschreibung mit Fuzzy Logic 792

36 Das Strahlenverfahren **793**
36.1 Anwendungen . . . 793
36.2 Prinzipieller Ablauf des Strahlenverfahrens . . . 794
36.3 Aufbau des Merkmalsvektors für ein Segment . . . 795
36.4 Klassifizierungs- / Produktionsphase . . . 800
36.5 Strahlenverfahren: Ein Beispiel . . . 800

37 Neuronale Netze und Segmentbeschreibung **804**
37.1 Anwendungen . . . 804
37.2 Prinzipieller Ablauf . . . 804
37.3 Trainingsdaten und Training . . . 806
37.4 Die Produktions- (Recall-) Phase . . . 807
37.5 Ein Beispiel: Unterscheidung von Schrauben . . . 807

38 Kalman-Filter **811**
38.1 Grundidee . . . 811
38.2 Anwendungen . . . 812
38.2.1 Tracking . . . 812
38.2.2 3D-Rekonstruktion aus Bildfolgen . . . 817
38.2.3 Bilddatencodierung . . . 818
38.3 Theorie des diskreten Kalman-Filters . . . 820
38.3.1 Das System . . . 820
38.3.2 Die Messung . . . 820
38.3.3 Die Schätzfehler-Gleichungen . . . 820
38.3.4 Optimales Schätzfilter von Kalman . . . 821
38.3.5 Gleichungen des Kalman-Filters . . . 822
38.3.6 Das erweiterte Kalman-Filter (EKF) . . . 824
38.4 Konkrete Beispiele . . . 825
38.4.1 Schätzung einer verrauschten Konstante . . . 826
38.4.2 Schätzung einer Wurfparabel . . . 829
38.4.3 Bildbasierte Navigation . . . 832

39 Zusammenfassen von Segmenten zu Objekten **840**
39.1 Bestandsaufnahme und Anwendungen . . . 840
39.2 Einfache, heuristische Vorgehensweise . . . 843
39.3 Strukturelle Verfahren . . . 845
39.3.1 Die Mustererkennungskomponente . . . 847
39.3.2 Die statische und dynamische Wissensbasis . . . 848
39.3.3 Die Reaktionskomponente . . . 849
39.3.4 Die Verwaltungskomponente . . . 849
39.3.5 Die Interaktionskomponente . . . 849
39.3.6 Die Dokumentationskomponente . . . 850
39.3.7 Ein Beispiel . . . 850

39.4 Bildverarbeitungssysteme im Einsatz 852

Literatur zu Teil II **854**

Sachverzeichnis **858**

Kapitel 1

Einleitung

Die elektronische Datenverarbeitung hat in den letzten fünf Jahrzehnten eine atemberaubende Entwicklung durchgemacht. Sie wurde ermöglicht durch neue Erkenntnisse in der Hardwaretechnologie, die Miniaturisierung der Bauteile, die Erhöhung der Rechengeschwindigkeit und der Speicherkapazität, die Parallelisierung von Verarbeitungsabläufen und nicht zuletzt die enorm sinkenden Kosten. Ende der 60er Jahre des letzten Jahrhunderts wurde z.B. der Preis für ein Bit Halbleiterspeicher mit etwa 0.50 Euro (damals noch 1.- DM) angegeben. Demnach hätte 1 MByte Hauptspeicher für einen Rechner über 4000000.- Euro gekostet.

Nachdem ursprünglich die elektronischen Rechenanlagen fast ausschließlich zur Lösung numerischer Problemstellungen eingesetzt wurden, drangen sie, parallel zu ihrer Hardwareentwicklung, in viele Gebiete unseres täglichen Lebens ein. Beispiele hierzu sind moderne Bürokommunikationssysteme oder Multimedia-Anwendungen. Aus den elektronischen Rechenanlagen entwickelten sich Datenverarbeitungssysteme, die in kommerziellen und wissenschaftlichen Bereichen erfolgreich eingesetzt werden. Aber auch in den meisten privaten Haushalten sind PCs zu finden, die dort eine immer wichtigere Rolle spielen und, gerade im Multimedia-Bereich, angestammte Geräte wie Fernseher, Stereoanlagen, DVD-Player/Recorder oder Spielekonsolen verdrängen.

Die Verarbeitung von visuellen Informationen ist ein wichtiges Merkmal höherer Lebensformen. So ist es nicht verwunderlich, dass schon frühzeitig versucht wurde, auch in diesem Bereich Computer einzusetzen, um z.B. bei einfachen, sich immer wiederholenden Arbeitsvorgängen eine Entlastung des menschlichen Bearbeiters zu erreichen. Ein gutes Beispiel ist die automatische Verarbeitung von Belegen im bargeldlosen Zahlungsverkehr. Hier wurde durch den Einsatz der modernen Datenverarbeitung nicht nur eine Befreiung des Menschen von eintöniger Arbeit erreicht, sondern auch geringere Fehlerhäufigkeit und wesentlich höhere Verarbeitungsgeschwindigkeit erzielt.

Die Benutzer von Datenverarbeitungs- und PC-Systemen werden heute nur mehr mit grafischen Betriebssystemen und grafisch aufbereiteter Anwendungssoftware konfrontiert. Für die Entwickler dieser Software heißt das, dass sie leistungsfähige Programmiersysteme für *Computergrafik* benötigen. OpenGL ist ein derartiges Programmiersystem für grafische Computeranwendungen. Es hat sich als ein weltweiter Standard etabliert und ist weitge-

hend unabhängig von Hard- und Softwareplattformen. Außerdem ist es kostenlos verfügbar.

Damit der Anwender die Computergrafik sinnvoll verwenden kann, benötigt er einen Rechner mit schnellem Prozessor, ausreichendem Hauptspeicher und eine leistungsfähige Grafikkarte. Diese Forderungen erfüllen die meisten PC-Systeme, die überall angeboten werden. Der interessierte Computeraspirant kann sich sogar beim Einkauf neben Butter, Brot und Kopfsalat ein passendes System beschaffen.

Wenn es die Hardware der Grafikkarte erlaubt, verlagert OpenGL die grafischen Berechnungen vom Prozessor des Computers auf die Grafikkarte. Das hat zur Folge, dass grafische Anwendungen den Prozessor nicht belasten und auf der eigens dafür optimierten Grafikkarte optimal ablaufen. Moderne Grafikkarten haben dabei eine Leistungsfähigkeit, die sich mit der von Großrechenanlagen messen kann.

Bei grafischen Anwendungen, etwa bei Simulationen oder bei Computerspielen, wird angestrebt, dass auf dem Bildschirm dem Benutzer ein möglichst realistisches Szenario angeboten wird. Die Bilder und Bildfolgen werden hier ausgehend von einfachen grafischen Elementen, wie Geradenstücke, Dreiecke oder Polygonnetze, aufgebaut. Mit geometrischen Transformationen werden dreidimensionale Effekte erzielt, Beleuchtung, Oberflächengestaltung und Modellierung von Bewegungsabläufen sind weitere Schritte in Richtung realistisches Szenario. Angestrebt wird eine Darstellung, bei der der Betrachter nicht mehr unterscheiden kann, ob es sich z.B. um eine Videoaufzeichnung oder um eine computergrafisch generierte Szene handelt. Der große Vorteil ist dabei, dass der Benutzer interaktiv in das Geschehen eingreifen kann, was bei reinen Videoaufzeichnungen nur eingeschränkt möglich ist. Der Weg der Computergrafik ist also die Synthese, vom einfachen grafischen Objekt zur natürlich wirkenden Szene.

In der *digitalen Bildverarbeitung* wird ein analytischer Weg beschritten: Ausgehend von aufgezeichneten Bildern oder Bildfolgen, die aus einzelnen Bildpunkten aufgebaut sind, wird versucht, logisch zusammengehörige Bildinhalte zu erkennen, zu extrahieren und auf einer höheren Abstraktionsebene zu beschreiben.

Um das zu erreichen, werden die Originalbilddaten in rechnerkonforme Datenformate transformiert. Sie stehen dann als zwei- oder mehrdimensionale, diskrete Funktionen für die Bearbeitung zur Verfügung. Die Verfahren, die auf die digitalisierten Bilddaten angewendet werden, haben letztlich alle die Zielsetzung, den Bildinhalt für den Anwender passend aufzubereiten. Der Begriff „Bildinhalt" ist dabei rein subjektiv: Dasselbe Bild kann zwei Beobachtern mit unterschiedlicher Interessenlage grundsätzlich verschiedene Informationen mitteilen. Aus diesem Grund werden auch die Transformationen, die beide Beobachter auf das Bild anwenden, ganz verschieden sein. Das Ergebnis kann, muss aber nicht in bildlicher oder grafischer Form vorliegen. Es kann z.B. auch eine Kommandofolge zur Steuerung eines Roboters oder einer Fräsmaschine sein.

Mit der digitalen Bildverarbeitung verwandte Gebiete sind die *Mustererkennung* und die *künstliche Intelligenz*. Die Mustererkennung ist im Gegensatz zur digitalen Bildverarbeitung nicht auf bildhafte Informationen beschränkt. Die Verarbeitung von akustischen Sprachsignalen mit der Zielsetzung der Sprach- oder Sprechererkennung ist z.B. ein wichtiger Anwendungsbereich der Mustererkennung. Im Bereich bildhafter Informationen wird mit den Verfahren der Mustererkennung versucht, logisch zusammengehörige Bildinhalte

zu entdecken, zu gruppieren und so letztlich abgebildete Objekte (z.B. Buchstaben, Bauteile, Fahrzeuge) zu erkennen. Um hier zufriedenstellende Ergebnisse zu erzielen, sind in der Regel umfangreiche Bildvorverarbeitungsschritte durchzuführen.

Künstliche Intelligenz ist ein Oberbegriff, der für viele rechnerunterstützte Problemlösungen verwendet wird (z.B. natürlich-sprachliche Systeme, Robotik, Expertensysteme, automatisches Beweisen, Bildverstehen, kognitive Psychologie, Spiele). Im Rahmen der Verarbeitung von bildhafter Information wird die Ableitung eines Sinnzusammenhangs aus einem Bild oder einer Bildfolge versucht. Eine Beschreibung der Art: „Ein Bauteil liegt mit einer bestimmten Orientierung im Sichtbereich", kann dann in eine Aktion umgesetzt werden, etwa das Greifen und Drehen des Bauteils mit einer industriellen Handhabungsmaschine. Hier werden also Systeme angestrebt, die im Rahmen eines wohldefinierten „Modells der Welt" mehr oder weniger unabhängig agieren und reagieren. Diese Selbstständigkeit ist meistens erst nach einer langen Anwendungskette von Bildverarbeitungs- und Mustererkennungsalgorithmen möglich.

Wenn ein Bild oder eine Bildfolge analytisch aufbereitet ist, kann der Informationsgehalt symbolisch beschrieben sein. Bei einer Bildfolge einer Straßenszene könnte das etwa so aussehen:

- Die Bildfolge zeigt eine Straße, die von rechts vorne nach links hinten verläuft.
- Auf der Straße bewegen sich zwei Fahrzeuge in entgegengesetzter Richtung und etwa gleicher Geschwindigkeit.
- Bei dem Fahrzeug, das von rechts vorne nach links hinten fährt, handelt es sich um ein rotes Cabriolet vom Typ X des Herstellers Y.
- Bei dem Fahrzeug, das von links hinten nach rechts vorne fährt, handelt es sich um einen weißen Minitransporter vom Typ XX des Herstellers YY.
- Links neben der Straße ist ein Wiesengelände, rechts ein Nadelwald.
- Den Horizont bildet eine Bergkette, die zum Teil mit Schnee bedeckt ist.
- usw.

Im Rahmen der Beschreibung könnten auch die Nummernschilder der beiden Fahrzeuge vorliegen. Dann wäre es als Reaktion z.B. möglich, beiden Fahrzeughaltern einen Bußgeldbescheid zuzusenden, da ihre Fahrzeuge mit überhöhter Geschwindigkeit unterwegs waren.

Eine andere interessante Möglichkeit wäre es, aus der symbolischen Beschreibung mit Computergrafik eine Szene zu generieren und zu vergleichen, wie realistisch sie die ursprüngliche Straßenszene wiedergibt.

Mit diesem Beispiel wurde gezeigt, wie eng Computergrafik und Bildverarbeitung heute miteinander verknüpft sind. Dieser Tatsache versucht das vorliegende Buch gerecht zu werden. Es gliedert sich in zwei Teile: Der erste Teil befasst sich mit der Thematik „Computergrafik". Nach einem Kapitel, in dem ausführlich auf den Zusammenhang zwischen

Computergrafik, Bildverarbeitung und Mustererkennung eingegangen wird, folgen Kapitel über interaktive 3D-Computergrafik und ihre Anwendungen. Ab Kapitel 6 werden einzelne Bestandteile der Computergrafik, wie Grundobjekte, Koordinatentransformationen, Verdeckung, Farbverarbeitung, Anti-Aliasing, Beleuchtung und Texturen beschrieben. An Hand von zahlreichen Programmfragmenten wird gezeigt, wie OpenGL die jeweiligen Problemstellungen unterstützt. Den Abschluss des Computergrafik-Teils bildet ein Kapitel über „Echtzeit 3D-Computergrafik".

Der zweite Teil des Buches ab Kapitel 16 ist der digitalen Bildverarbeitung gewidmet. Zunächst wird die Digitalisierung von Bilddaten untersucht und, ausgehend vom Binärbild (Zweipegelbild) über das Grauwertbild, das Farbbild bis zur Bildfolge verallgemeinert. Anschließend werden Maßzahlen zur Beschreibung digitalisierter Bilddaten vorgestellt und verschiedene mathematische Modelle für Bilder behandelt. Bei der Diskussion der Speicherung von digitalisierten Bilddaten in Datenverarbeitungssystemen werden Lösungsansätze zur Datenreduktion und Datenkompression vorgestellt. Einem Abschnitt über die bildliche Reproduktion von digitalisierten gespeicherten Bildern schließen sich Verfahren zur Modifikation der Grauwertverteilung an. Weiter folgt die Untersuchung von Operationen im Orts- und Frequenzbereich, von morphologischen Operationen und von Kanten und Linien.

Die weiteren Kapitel sind mehr in Richtung Mustererkennung bei bildhaften Daten orientiert. Nach der grundlegenden Darstellung der Szenenanalyse werden unterschiedliche Techniken zur Merkmalsgewinnung besprochen. Stichworte hierzu sind: Grauwert und Farbe, Merkmale aus mehrkanaligen Bildern und aus Bildfolgen. Der Beschreibung von einfachen Texturmerkmalen schließen sich aufwändigere Verfahren, wie Gauß- und Laplace-Pyramiden, Scale Space Filtering und Baumstrukturen an. In den anschließenden Kapiteln wird die Segmentierung mit klassischen Methoden, mit neuronalen Netzen und mit Fuzzy Logic beschrieben. Nach dem Übergang von der bildpunkt- zur datenstrukturorientierten Bildverarbeitung werden unterschiedliche Verfahren zur Segmentbeschreibung erläutert. Den Abschluss bildet ein Kapitel über Kalman Filter und der Synthese von Objekten aus Segmenten.

Zu diesem Buch liegt auch ein Internetangebot unter folgender Adresse vor:

http://www.cs.fhm.edu/cgbv-buch

Der Zugang zu dieser Webseite ist passwort-geschützt. Den Benutzernamen und das Passwort erhält man, wenn man auf dieser Webseite dem Link „Passwort" folgt. Das Passwort wird regelmäßig geändert.

Der Online-Service umfasst folgende Angebote:

- interaktive Vorlesungen zu den Themen:
  - Computergrafik
  - Bildverarbeitung und Mustererkennung
  - Bilddatencodierung für die Übertragung und Kommunikation
- Übungsaufgaben
- Das Bildverarbeitungswerkzeug IGARIS
- Bilder bzw. Texturen
- Beispiel-Software
- Eine aktuelle Liste interessanter Internet-Links
- Eine aktuelle Literatur-Liste
- Korrekturen zum Buch
- Zusatzkapitel zum Buch

Als zusätzliche Informationsquelle wird eine CD-ROM mit allen hier geschilderten Bestandteilen von den Autoren angeboten.

Kapitel 2

Zusammenhang zwischen Computergrafik und Bildverarbeitung

Warum fasst man die Gebiete Computergrafik und Bildverarbeitung in einem Buch zusammen? Weil sie die zwei Seiten einer Medaille sind: während man in der Computergrafik aus einer abstrakten Objektbeschreibung ein Bild generiert, versucht man in der Bildverarbeitung nach der Extraktion von charakteristischen Merkmalen die im Bild enthaltenen Objekte zu erkennen und so zu einer abstrakten Objektbeschreibung zu kommen (Bild 2.1). Oder anders ausgedrückt: Computergrafik ist die Synthese von Bildern und Bildverarbeitung ist die Analyse von Bildern. In diesem Sinne ist die Computergrafik die inverse Operation zur Bildverarbeitung.

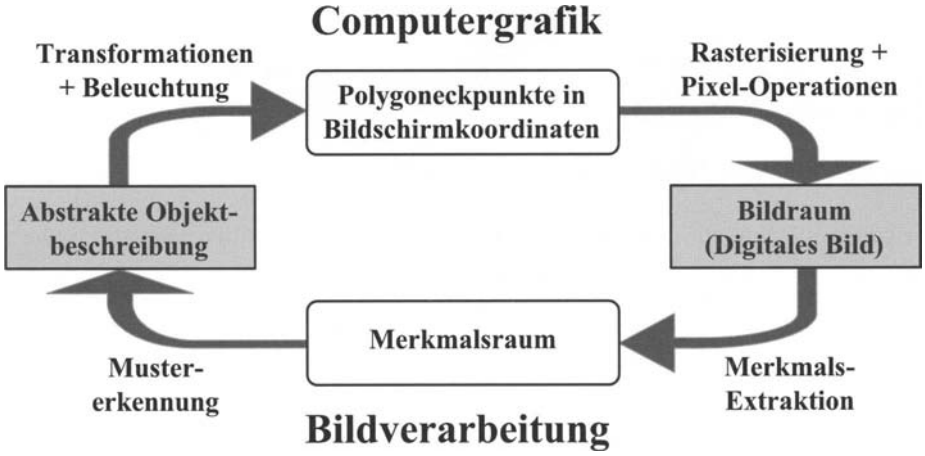


Bild 2.1: Die zwei Seiten einer Medaille: Computergrafik ist die Synthese von Bildern aus einer abstrakten Objektbeschreibung und Bildverarbeitung ist die Analyse von Bildern mit dem Ziel, zu einer abstrakten Objektbeschreibung zu gelangen.

Für Computergrafik und Bildverarbeitung benötigt man in vielen Teilen die gleichen Methoden und das gleiche Wissen. Dies beginnt beim Verständnis für den Orts- und Frequenzbereich, das Abtasttheorem, das Anti-Aliasing und die verschiedenen Farbräume, setzt sich fort bei den Matrizen-Operationen der linearen Algebra (Geometrie-Transformationen wie Translation, Rotation, Skalierung und perspektivische Projektion, Texturkoordinaten-Transformation, Transformationen zwischen den Farbräumen etc.), geht über die Nutzung von Faltungs- und Morphologischen Operatoren bis hin zu Datenstrukturen, wie z.B. Gauß-Pyramiden (MipMaps), quad- bzw. octrees, sowie Graphen zur Szenenbeschreibung.

Nachdem Computergrafik und Bildverarbeitung häufig die gleichen Algorithmen einsetzen und die Grafikhardware in den letzten 5 Jahren einen gigantischen Leistungssprung um den Faktor 100 geschafft hat, liegt es Nahe, teure Spezialhardware zur Bildverarbeitung durch billige PC-Grafikkarten zu ersetzen, wie im folgenden Abschnitt erläutert wird. Bei einer zunehmenden Anzahl von Anwendungen wird Computergrafik und Bildverarbeitung gleichzeitig eingesetzt, wie am Beispiel der Simulation von kameragesteuerten Geräten erläutert wird. Im Multimedia-Bereich entsteht eine immer engere Verzahnung von Computergrafik und Bildverarbeitung. Dies wird anhand moderner Bilddatencodierungsmethoden erklärt. Im Rahmen des im letzten Abschnitt vorgestellten „bildbasierten Renderings“ löst sich die Trennung zwischen klassischer Computergrafik und Bildverarbeitung immer mehr auf.

2.1 Bildverarbeitung auf programmierbarer Grafikhardware

Der mit weitem Abstand größte Leistungszuwachs in der Computerhardware hat in den letzten 5 Jahren im Bereich der Grafikhardware stattgefunden. Eine aktuelle nVIDIA GeForce 7900 GTX Grafikkarte hat etwa die 100-fache Rechenleistung wie die vor 5 Jahren aktuelle GeForce 2, und etwa die 200-fache Floating-Point-Rechenleistung wie die derzeit schnellsten „Pentium IV“-Prozessoren. Zu verdanken ist dieser enorme Leistungszuwachs in der Grafikhardware vor allem den Millionen Kindern, die von den Möglichkeiten interaktiver Computerspiele fasziniert wurden. Seit Anfang 2002 sind diese Grafikkarten auch noch relativ gut durch Hochsprachen (Abschnitt 12.3.1) programmierbar, so dass sie auch für andere Zwecke als nur Computergrafik genutzt werden können. Allerdings sind programmierbare Grafikkarten nicht bei allen Rechenaufgabe schneller als gewöhnliche Prozessoren, sondern nur bei solchen, für die die Grafikhardware optimiert ist, wie z.B. Vektor- und Matrizen-Operationen. Genau diese Operationen werden aber sowohl in der Computergrafik als auch in der Bildverarbeitung sehr häufig benötigt. Ein weiterer wichtiger Grund für die extrem hohe Rechenleistung von Grafikkarten ist der hohe Parallelisierungsgrad in der Hardware. So arbeiten in der bereits erwähnten GeForce 7900 GTX Grafikkarte beispielsweise 24 Pixelprozessoren parallel. Um diese 24 Pixelprozessoren gleichmäßig auszulasten, benötigt man eine Rechenaufgabe, die trivial parallelisierbar ist, wie z.B. die Texturierung

aller Pixel eines Polygons in der Computergrafik (Kapitel 13), oder auch die Faltung eines Bildes mit einem Filterkern in der Bildverarbeitung (Abschnitt 18.2).

Die Grundidee besteht nun darin, sich die riesige Rechenleistung heutiger Grafikkarten für eine schnelle Echtzeit-Bildverarbeitung zu Nutze zu machen und somit teure Spezialhardware (FPGAs = Field Programmable Gate Arrays) zur Bildverarbeitung durch billige und hochsprachen-programmierbare Grafikkarten zu ersetzen. Da FPGAs nie zu einem richtigen Massenprodukt geworden sind, das in millionenfacher Stückzahl hergestellt worden wäre, resultiert ein erheblich höherer Stückpreis als bei PC-Grafikkarten. Außerdem wurde für FPGAs nie eine Hochsprache zur Programmierung entwickelt, so dass sie wie Computer in ihrer Frühzeit durch Assembler-Code gesteuert werden müssen. Da jedes FPGA seine eigene ganz spezifische Assembler-Sprache besitzt, muss für jede neue Generation an FPGAs eine teure Anpassentwicklung durchgeführt werden. All diese Probleme entfallen bei den billigen und hochsprachen-programmierbaren Grafikkarten.

Um die Umsetzbarkeit dieser Grundidee in die Praxis zu überprüfen, wurden im Labor für Computergrafik und Bildverarbeitung (www.cs.fhm.edu/~nischwit/labor.html) an der Fachhochschule München entsprechende Untersuchungen durchgeführt. Dabei wurde zunächst die Implementierbarkeit verschiedener Bildverarbeitungs-Algorithmen auf programmierbaren Grafikkarten (nach *Shader Model 3.0*) sehr erfolgreich getestet. Dazu zählten Faltungsoperatoren (Kapitel 18), wie z.B.

- der gleitende Mittelwert (18.2),
- der Gauß-Tiefpassfilter (18.4 und Bild 2.2-b)
  mit unterschiedlich großen Faltungskernen ($3 \cdot 3, 7 \cdot 7, 11 \cdot 11$),
- der Laplace-Operator (18.29),
- der Sobelbetrags-Operator (18.27),

und morphologische Operatoren (Kapitel 19), wie z.B.

- die Dilatation(19.3),
- die Erosion(19.3 und Bild 2.2-c),
- und der Median-Filter (Abschnitt 19.3).

Weitere Beispiele für komplexe Bildverarbeitungsoperationen, die auf programmierbarer Grafikhardware implementiert wurden, sind z.B.

- die Fourier-Transformation [Suma05],
- die Hough-Transformation[Strz03],
- der Canny-Kantendetektor [Fung05],
- und weitere Tracking-Algorithmen [Fung05].

Häufig werden in der Bildverarbeitung jedoch auch mehrere verschiedene Operatoren hintereinander auf ein Bild angewendet. Dies ist auch auf der programmierbaren Grafikhardware möglich, indem man verschiedene *Shader* in einem sogenannten *Multipass-Rendering-Verfahren* mehrfach über ein Bild laufen lässt (Bild 2.2-d). Die Implementierung kann entweder über die `glCopyTexImage2D`-Funktion (Abschnitt 13.1.1.2) oder über die *Render-to-Texture*-Option (auch *pBuffer*-Funktion genannt) erfolgen, die neuere Grafikkarten bieten, welche das *Shader Model 3.0* erfüllen.

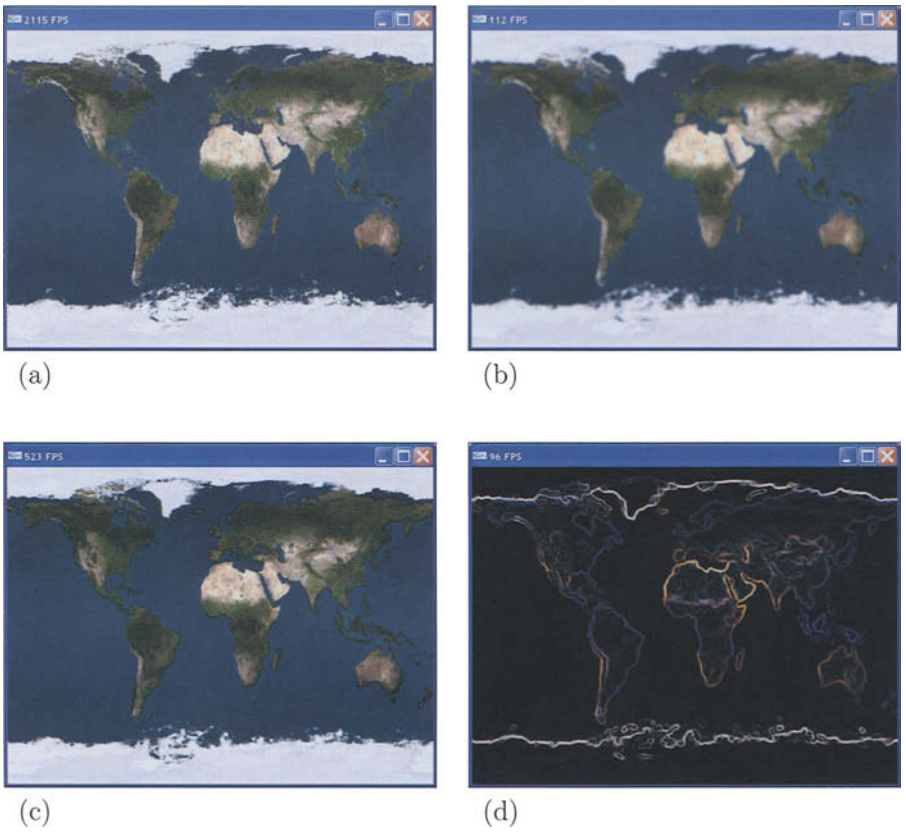


Bild 2.2: Beispiele für Bildverarbeitung auf programmierbarer Grafikhardware: (a) Originalbild. (b) Faltungsoperator: Gauß-Tiefpass mit $7 \cdot 7$-Filterkern und $\sigma = 2$. (c) Morphologischer Operator: Erosion. (d) Kombination aus den drei Operatoren Gauß-Tiefpass $7 \cdot 7$, Sobelbetrag und Erosion.

Ein Leistungsvergleich zwischen Grafikhardware (GPU = *Graphics Processing Unit*) und CPU (*Central Processing Unit*) in Bezug auf einige der oben aufgeführten Bildverarbeitungs-Algorithmen brachte erstaunliche Ergebnisse. So war die Grafikhardware

(nVidia GeForce 7900 GTX Go) bei den Beispielen aus Bild 2.2 um einen Faktor von ca. 100 schneller als die CPU (Intel Pentium M 2,26 GHz)! Die auf der Grafikhardware erzielbaren Bildraten (*FPS = Frames Per Second*) betrugen gigantische 523 Bilder/sec bei morphologischen Operatoren (Bild 2.2-c) und immerhin noch 112 Bilder/sec bei der Anwendung eines Gauß-Tiefpassfilters mit einem nicht separiertem $7 \cdot 7$-Filterkern (Bild 2.2-b). Damit ist nicht mehr die Bildverarbeitung der Flaschenhals in Echtzeit-Anwendungen, sondern der Datentransfer von der Kamera in den Computer. Oder anders ausgedrückt: heutzutage kann man sehr aufwändige Bildverarbeitungs-Operationen in Echtzeit auf Grafikkarten durchführen, die sich fast jeder leisten kann.

Das Grundprinzip der Implementierung ist einfach: es muss nur ein bildschirmfüllendes Rechteck gezeichnet werden, auf das das zu bearbeitende Bild mit Hilfe des *Texture-Mappings* (Kapitel 13) aufgebracht wird. Im Pixel-Shader (Abschnitt 12.3), der den Bildverarbeitungs-Operator enthält, werden die Farbwerte des Bildes jetzt nicht nur Pixel für Pixel kopiert, sondern entsprechend dem verwendeten Operator miteinander verknüpft. Ein lauffähiges Programm-Beispiel inclusive Quell-Code ist auf der von den Autoren angebotenen CD-ROM erhältlich.

Die vielfältigen Möglichkeiten von programmierbaren Grafikkarten wurden mittlerweile auch für eine Reihe weiterer Anwendungen außerhalb der Computergrafik und Bildverarbeitung erkannt. So werden bereits Software-Pakete für Dynamik-Simulationen angeboten, die nicht mehr wie bisher auf der CPU ausgeführt werden, sondern auf der Grafikkarte (GPU). Für besonders anspruchsvolle Computerspiele werden neuerdings spezielle PCs angeboten, die zwei extrem leistungsfähige Grafikkarten enthalten, eine Karte für die Computergrafik und die zweite Karte für die Physik-Simulationen. Weitere Anwendungen basieren auf der Finiten-Elemente-Methode (FEM) bzw. im allgemeinen auf dem Lösen von großen linearen Gleichungssystemen, wie z.B. Aero- und Fluiddynamik-Simulationen, oder auch virtuelle Crash-Tests. Nachdem die Hardware-Architekten derzeit auch bei CPUs auf Parallelisierung setzen (Stichwort *Dual- und Multi-Core CPUs*), wurden schon Überlegungen angestellt ([Owen05]), ob nicht der Impuls für die Weiterentwicklung von Computerhardware generell durch den rasanten Fortschritt der Grafikhardware gesetzt wurde.

2.2 Simulation von kameragesteuerten Geräten

Ein gutes Beispiel für das Zusammenwirken von Computergrafik und Bildverarbeitung ist die Simulation von kameragesteuerten Geräten. Solche Geräte sind z.B. autonome mobile Roboter oder Lenkflugkörper, die eine Videokamera als wesentlichen Sensor besitzen. Um ein solches Gerät in seiner Interaktion mit der Umwelt simulieren zu können, muss der Sensor – in diesem Fall die Videokamera – mit geeigneten Stimuli, d.h. Bildern, versorgt werden. Eine Möglichkeit, solche Bilder zur Verfügung zu stellen, besteht darin, einen Videofilm aufzuzeichnen und ihn später wieder in die Simulation einzuspeisen. Ein großer Vorteil dieser Technik ist die absolute Realitätsnähe der Bilder, da sie ja in der realen Umwelt mit dem realen Sensor aufgezeichnet werden können. Der entscheidende Nachteil dieser Technik ist, dass keine Interaktion zwischen dem Gerät und der Umwelt mehr

möglich ist, d.h. die Regelschleife der Simulation ist offen (*open loop simulation*) . Um die Regelschleife zu schließen (*closed loop simulation*), ist eine interaktive Bildgenerierung notwendig, d.h. aus der bei jedem Zeitschritt neu bestimmten Position und Lage des Geräts muss ein aktuelles Bild generiert werden. Da es unmöglich ist, für alle denkbaren Positionen und Orientierungen der Kamera reale Bilder aufzuzeichnen, muss das Bild aus einer visuellen 3D-Datenbasis mit Hilfe der Computergrafik interaktiv erzeugt werden. Die Computergrafik generiert in diesem Fall direkt den Input für die Bildverarbeitung.

Aus systemtheoretischer Sicht lässt sich die Simulation solcher Systeme zunächst einmal grob in zwei Bereiche unterteilen: Das kameragesteuerte Gerät auf der einen Seite, das mit der Umwelt auf der anderen Seite interagiert (Bild 2.3). Eine Simulation, in der sowohl die Umwelt, wie auch das kameragesteuerte Gerät durch ein rein digitales Modell ersetzt wird, bezeichnet man als „Mathematisch Digitale Simulation" (MDS). Die MDS ist also ein Software-Paket, das auf jedem beliebigen Computer ablaufen kann.

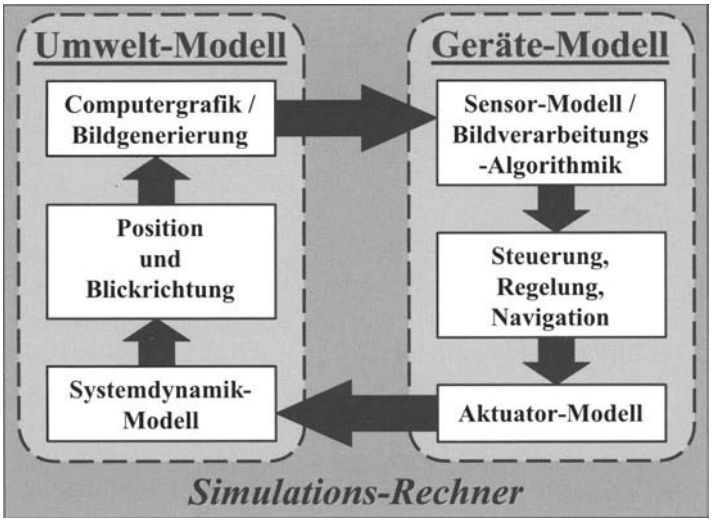


Bild 2.3: Mathematisch Digitale Simulation (MDS) eines kameragesteuerten Geräts. Im Rahmen einer geschlossenen Regelschleife interagieren die Komponenten des Geräts auf der rechten Seite mit den Komponenten der Umwelt auf der linken Seite. Alle Komponenten bestehen aus digitalen Modellen, die gemeinsam auf einem Simulations-Rechner ablaufen.

Das Modell des kameragesteuerten Geräts besteht aus:

- Der Sensorik (hier ein Kamera-Modell) zur Erfassung der Umwelt,
- der Bildverarbeitungs-Algorithmik zur Objekterkennung und -verfolgung,
- der Steuer-, Regel- und Navigations-Algorithmik, die auf der Basis der gestellten Aufgabe (Führungsgröße) und der erfassten Situation (Messgröße) eine Reaktion (Stellgröße) des Geräts ableitet,

- der Aktorik (z.B. Lenkrad bzw. Ruder zur Einstellung der Bewegungsrichtung sowie dem Antrieb zur Regelung der Geschwindigkeit).

Das Modell der Umwelt besteht aus:

- Einem Systemdynamik-Modell, das zunächst aus der Stellung der Aktoren die auf das System wirkenden Kräfte und Drehmomente berechnet, und anschließend aus den Bewegungsgleichungen die neue Position und Orientierung des Geräts,

- einer visuellen Datenbasis, die die 3-dimensionale Struktur und die optischen Eigenschaften der Umwelt enthält. Mit Hilfe der Computergrafik wird aus der errechneten Position und Blickrichtung der Kamera der relevante Ausschnitt aus der visuellen Datenbasis in ein Bild gezeichnet. Danach wird das computergenerierte Bild in das Kamera-Modell eingespeist, und der nächste Durchlauf durch die Simulationsschleife startet.

Die MDS ist eines der wichtigsten Werkzeuge für die Entwicklung und den Test kameragesteuerter Systeme und somit auch für die Bildverarbeitungs-Algorithmik. Ein Vorteil der MDS ist, dass in der Anfangsphase eines Projekts, in der die Ziel-Hardware noch nicht zur Verfügung steht, die (Bildverarbeitungs-) Algorithmik bereits entwickelt und in einer geschlossenen Regelschleife getestet werden kann. In späteren Projektphasen wird die Onboard-Bildverarbeitungs-Software ohne Änderungen direkt in die MDS portiert. Deshalb sind Abweichungen vom realen Verhalten, z.B. aufgrund unterschiedlicher Prozessoren, Compiler bzw. des zeitlichen Ablaufs, sehr gering. Im Rahmen der Validation der MDS mit der *Hardware-In-The-Loop* Simulation bzw. mit dem realen System werden evtl. vorhandene Unterschiede minimiert. Da die MDS somit das Verhalten der realen Bildverarbeitungs-Komponente sehr präzise reproduzieren kann, wird sie am Ende der Entwicklungsphase auch zum Nachweis der geforderten Leistungen des Geräts verwendet. Dies spart Kosten und ist wegen des häufig sehr breiten Anforderungsspektrums (unterschiedliche Landschaften kombiniert mit verschiedenen Wetterbedingungen, Tageszeiten, Störungen, etc.) in der Regel nur noch mit einer großen Anzahl virtueller Versuche (d.h. Simulationen) in der MDS realisierbar.

Voraussetzung für eine verlässliche Simulation ist aber nicht nur eine 1:1-Abbildung der Bildverarbeitungs-Software in der MDS, sondern auch die interaktive Generierung möglichst realistätsnaher Bilder mit Hilfe der Computergrafik. Das Ziel der Computergrafik muss sein, dass das Ergebnis der Bildverarbeitung bei computergenerierten Bildern das Gleiche ist wie bei realen Bildern. Und dabei lässt sich ein Algorithmus nicht so leicht täuschen wie ein menschlicher Beobachter. Schwierigkeiten bei der Generierung möglichst realitätsnaher Bilder bereitet die natürliche Umwelt mit ihrem enormen Detailreichtum und den komplexen Beleuchtungs- und Reflexionsverhältnissen, sowie das zu perfekte Aussehen computergenerierter Bilder (die Kanten sind zu scharf und zu gerade, es gibt keine Verschmutzung und keine Störungen). Dies führt in der Regel dazu, dass die Bildverarbeitungs-Algorithmik in der Simulation bessere Ergebnisse liefert als in der Realität. Um diesen Lerneffekt nach den ersten Feldtests zu vermeiden, ist eine

Validation der computergenerierten Bilder anhand realer Videoaufzeichnungen erforderlich. Erst wenn das Ergebnis der Bildverarbeitung im Rahmen bestimmter Genauigkeitsanforderungen zwischen synthetisierten und real aufgenommenen Bildern übereinstimmt, kann der Simulation genügend Vertrauen geschenkt werden, um damit Feldversuche einzusparen bzw. den Leistungsnachweis zu erbringen.

Damit computergenerierte Bilder kaum noch von realen Bildern unterscheidbar sind, muss fast immer ein sehr hoher Aufwand betrieben werden. Um die 3-dimensionale Gestalt der natürlichen Umwelt möglichst genau nachzubilden, ist eine sehr große Anzahl an Polygonen erforderlich (Kapitel 6). Zur realistischen Darstellung komplexer Oberflächen- und Beleuchtungseffekte sind sehr viele Foto-Texturen, Relief-Texturen, Schatten-Texturen usw. notwendig (Kapitel 13), sowie aufwändige Beleuchtungsrechnungen (Kapitel 12). Die Glättung zu scharfer Kanten kann mit Hilfe des Anti-Aliasing, d.h. einer rechenaufwändigen Tiefpass-Filterung erreicht werden (Kapitel 10). Luftverschmutzung und andere atmosphärische Effekte können mit Hilfe von Nebel simuliert werden (Kapitel 11). Die Liste der Maßnahmen zur Steigerung des Realitätsgrades computergenerierter Bilder könnte noch um viele weitere und zunehmend rechenintensivere Punkte ergänzt werden. Wichtig ist aber nicht, dass das computergenerierte Bild in allen Eigenschaften exakt dem realen Bild entspricht, sondern dass diejenigen Merkmale, die die Bildverarbeitung später für die Objekterkennung nutzt, möglichst gut reproduziert werden. Für einen effizienten Einsatz der Ressourcen bei der Computergrafik ist es deshalb unerlässlich, zu verstehen, mit welchen Algorithmen die Bildverarbeitung die erzeugten Bilder analysiert.

In diesem Zusammenhang ist es ein Vorteil der MDS als reinem Software-Paket, dass die Simulation auch langsamer als in Echtzeit erfolgen kann. Wenn eine Komponente, wie z.B. die Bildgenerierung, sehr viel Rechenzeit benötigt, warten die anderen Simulationskomponenten, bis das Bild fertig gezeichnet ist. Dadurch hat man in der MDS die Möglichkeit, auch sehr detailreiche Szenarien in die Simulation zu integrieren, so dass die computergenerierten Bilder auch höchsten Anforderungen genügen. Deshalb ist die MDS ideal geeignet, um den statistischen Leistungsnachweis für Subsysteme (z.B. die Bildverarbeitungs-Komponente) und das Gesamtsystem mit höchster Genauigkeit durchzuführen.

Der Vorteil der MDS, wegen der nicht vorhandenen bzw. „weichen" Echtzeit-Anforderung (Abschnitt 3.2) beliebig detailreiche Szenarien darstellen zu können, wandelt sich aber im Hinblick auf die Verifikation der Bildverarbeitungs-Software auf dem Zielrechner in einen Nachteil um. Denn auf dem Onboard-Rechner des Geräts muss die Bildverarbeitungs-Software in Echtzeit getestet werden, damit deren Funktionstüchtigkeit und das Zusammenspiel mit anderen Komponenten nachgewiesen werden kann. Deshalb wird in späteren Phasen eines Entwicklungsprojekts, in denen die Komponenten des Geräts bereits als reale Prototypen zur Verfügung stehen, eine *Hardware-In-The-Loop* (HIL)-Simulation (Bild 2.4) aufgebaut, mit der die gesamte Regelschleife des Geräts in Echtzeit getestet wird. In einer HIL-Simulation muss die Umwelt weiterhin simuliert werden, allerdings in Echtzeit. Für die Computergrafik bedeutet dies eine hohe Anforderung, da für die Bildgenerierrate in einer HIL-Simulation eine „harte" Echtzeitanforderung gilt (Abschnitt 3.1). Verarbeitet die Kamera z.B. 50 Bilder pro Sekunde, muss die Computergrafik mindestens mit dieser Rate neue Bilder generieren. Dies bedeutet, dass pro Bild maximal 20 Millisekunden Rechenzeit

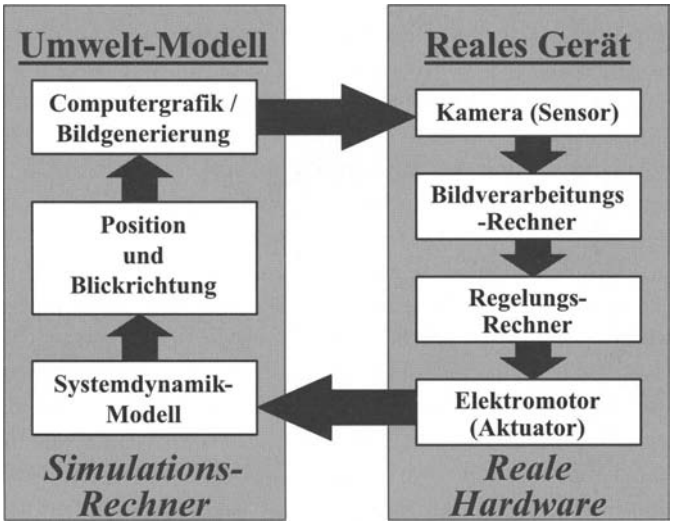


Bild 2.4: Hardware-In-The-Loop (HIL) Simulation eines kameragesteuerten Geräts. Das reale Gerät (oder evtl. nur einzelne Komponenten davon) interagiert in Echtzeit mit den simulierten Komponenten der Umwelt.

zur Verfügung stehen und deshalb gewisse Einschränkungen bei der Bildqualität in Kauf genommen werden müssen. Durch den enormen Fortschritt bei der Leistungsfähigkeit von Grafikhardware und neue Beschleunigungsverfahren für Echtzeit-3D-Computergrafik (Kapitel 15) ist zu erwarten, dass bald ein ausreichend hoher Realitätsgrad der Bilder bei interaktiven Generierraten (50 Bilder pro Sekunde und mehr) erzielt werden kann, damit der Leistungsnachweis auch in einer Echtzeit-Simulation erbracht werden kann.

2.3 Bilddatencodierung

Die Videocodierstandards MPEG-1 (Moving Picture Experts Group, ISO/IEC Standard 11172) und MPEG-2 (ISO/IEC Standard 13818) dienen der komprimierten digitalen Repräsentation von audiovisuellen Inhalten, die aus Sequenzen von rechteckigen 2-dimensionalen Bildern und den zugehörigen Audiosequenzen bestehen. Der Erfolg dieser Videocodierstandards verhalf einigen Produkten, wie Video-CD, DVD-Video und MP3-Geräten (MPEG-1 Audio layer 3), zum kommerziellen Durchbruch[1]. Methoden der 3D-Computergrafik, der Bildverarbeitung und der Mustererkennung kommen dabei (fast[2]) nicht zum Einsatz. Den Videocodierstandards MPEG-1 und MPEG-2 liegt das Paradigma des passiven Zuschauers bzw. Zuhörers zugrunde, genau wie beim Fernsehen oder Radio. Der

[1]Digitales Fernsehen und Digitales Radio, die ebenfalls auf den MPEG-Standards beruhen, konnten sich bisher noch nicht kommerziell durchsetzen.

[2]Bis auf das Blockmatching zur Berechnung von Bewegungsvektorfeldern in Bildsequenzen (Abschnitt 26.5).

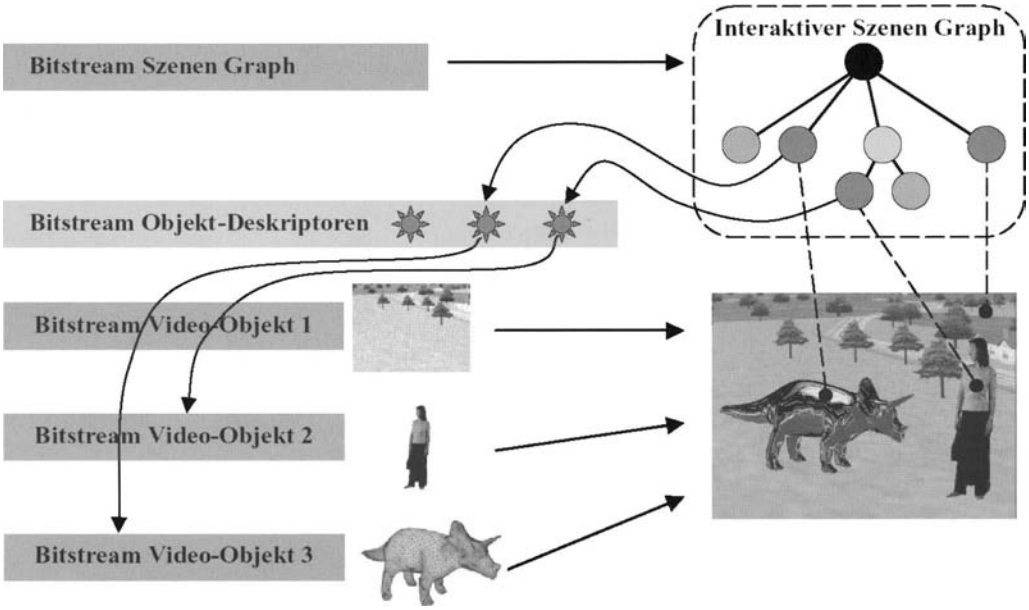


Bild 2.5: Szenen Graph zur Beschreibung der Video-Objekte in MPEG-4.

Konsument hat in diesem Fall keine Möglichkeit zur Interaktion mit den audiovisuellen Inhalten bzw. mit dem Anbieter dieser Inhalte.

Genau in diese Lücke stößt der neuere Codierstandard MPEG-4 (ISO/IEC Standard 14496). Er stellt ein objekt-orientiertes Konzept zur Verfügung, in dem eine audiovisuelle Szene aus mehreren Video-Objekten zusammengesetzt wird. Die Eigenschaften der Video-Objekte und ihre raum-zeitlichen Beziehungen innerhalb der Szene werden mit Hilfe einer Szenenbeschreibungssprache in einem dynamischen Szenen Graphen definiert (Bild 2.5). Dieses Konzept ermöglicht die Interaktion mit einzelnen Video-Objekten in der Szene, denn der Szenen Graph kann jederzeit verändert werden [Pere02]. Die Knoten des Szenen Graphen enthalten sogenannte „Objekt-Deskriptoren", die die individuellen Eigenschaften der Objekte beschreiben. Ein Objekt-Deskriptor kann z.B. ausschließlich eine Internet-Adresse (URL = *Uniform Resource Locator*) enthalten. Dadurch ist es möglich, die Inhalte einer audiovisuellen Szene durch Objekte zu ergänzen, die auf Media-Servern rund um die Welt verteilt sind. Voraussetzung für die Darstellung solcher Szenen ist dann natürlich ein entsprechender Internet-Zugang. Im Normalfall enthält ein Objekt-Deskriptor das eigentliche Video-Objekt (einen sogenannten „*elementary stream*"), sowie Zusatzinformationen zu dem Video-Objekt (eine Beschreibung des Inhalts und der Nutzungsrechte).

Die äußere Form der Video-Objekte kann, im Unterschied zu MPEG-1/-2, variieren. Während bei MPEG-1/-2 nur ein einziges Video-Objekt codiert werden kann, das aus einer Sequenz von rechteckigen Bildern besteht, erlaubt MPEG-4 die Codierung mehrerer Video-Objekte, die jeweils eine Sequenz von beliebig geformten 2-dimensionalen Bildern

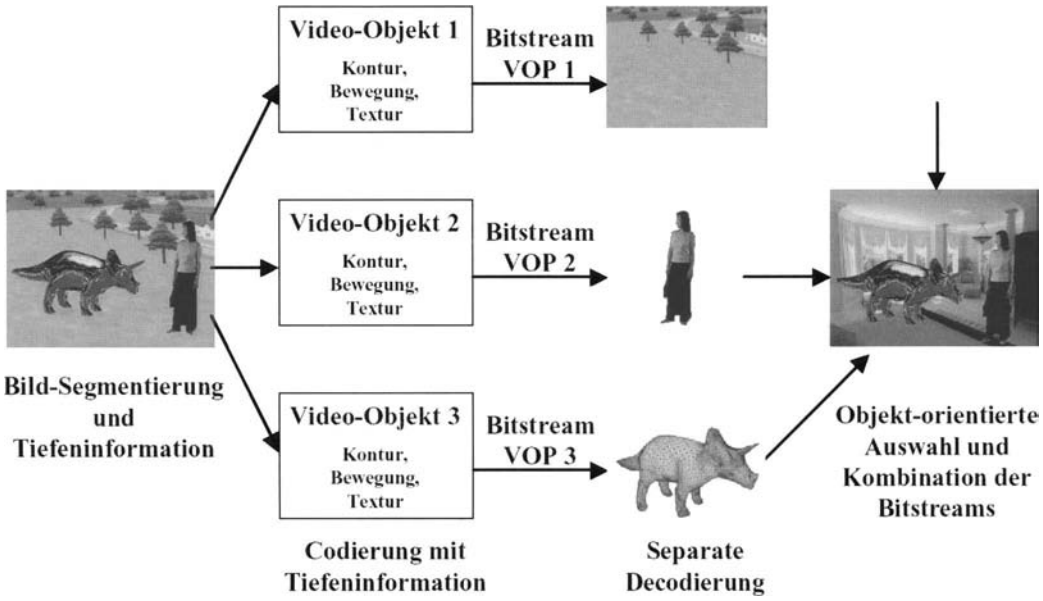


Bild 2.6: Zerlegung einer Bildsequenz in einzelne MPEG-4 Video-Objekte (*Video Object Planes* = VOPs). Nach der Decodierung der einzelnen Video-Objekte besteht die Möglichkeit, die Objekte neu zu kombinieren. Somit kann die Szene interaktiv und objektbezogen verändert werden.

enthalten. MPEG-4 erlaubt daher die Zerlegung eines Bildes in einzelne Segmente, die sich unterschiedlich bewegen. Jedes Segment ist ein Video-Objekt, das von Bild zu Bild in seiner Form veränderlich ist. In Bild 2.5 ist eine Szene gezeigt, die in drei Video-Objekte zerlegt werden kann: Einen statischen Hintergrund, eine bewegte Person und ein bewegtes Tier. Für jedes Video-Objekt wird die Kontur, die Bewegung und die Textur in einem eigenen Bitstream codiert. Der Empfänger decodiert die Video-Objekte und setzt sie wieder zur vollständigen Szene zusammen. Da der Empfänger jedes Video-Objekt separat decodiert und er den Szenen Graph interaktiv verändern kann, hat er die Möglichkeit, die einzelnen Video-Objekte nach seinen Vorstellungen zu kombinieren. In Bild 2.6 ist diese Möglichkeit illustriert: Hier werden zwei Video-Objekte aus Bild 2.5 mit einem neuen Video-Objekt für den Hintergrund kombiniert, so dass die Person und der Dinosaurier nicht mehr im Freien stehen, sondern in einem Zimmer.

Der Inhalt eines Video-Objekts kann eine „natürliche" Videosequenz sein, d.h. ein sogenanntes „*natural video object*", das mit einer Kamera in einer „natürlichen" Umgebung aufgezeichnet wurde (wie die Person in Bild 2.5 bzw. 2.6), oder ein künstliches Objekt (*synthetic video object*), aus dem der Konsument mit Hilfe von interaktiver 3D-Computergrafik eine synthetische Videosequenz generieren kann (der Dinosaurier in Bild 2.5 bzw. 2.6). Ein einfaches synthetisches Video-Objekt kann z.B. ein Text sein, der später einem Bild überlagert wird (*text overlay*). Anspruchsvollere synthetische Video-Objekte sind 3-dimensionale

Computergrafik-Modelle, die im Wesentlichen aus Polygonnetzen, Farben, Normalenvektoren und Texturen bestehen. Ein großer Vorteil von synthetischen Video-Objekten gegenüber natürlichen Video-Objekten besteht darin, dass der Blickwinkel, unter dem sie dargestellt werden, interaktiv und frei wählbar ist. Der Grad an Interaktivität, den synthetische Video-Objekte bieten, ist demnach sehr viel höher, als der von natürlichen Video-Objekten.

In MPEG-4 können also nicht nur klassische Videofilme, sondern auch 3D-Computergrafik-Modelle und Kombinationen von beiden effizient codiert werden. Somit ist es z.B. möglich, in eine natürliche 2D-Videosequenz ein computergeneriertes 3D-Modell eines Dinosauriers einzublenden. Diese Technik, die in der Filmproduktion für Trickeffekte mittlerweile Standard ist, kann mit MPEG-4 von jedem Konsumenten genutzt werden. Umgekehrt ist es ebenso machbar, in eine 3-dimensionale virtuelle Szene eine natürliche 2D-Videosequenz einzublenden (dies ist genau die gleiche Idee, wie bei animierten Texturen auf Billboards in der Echtzeit-3D-Computergrafik, Abschnitt 15.4). Die Kombination von synthetischen und natürlichen Bildinhalten im Rahmen einer effizienten Codierung ist ein sehr mächtiges Werkzeug, das im MPEG-Fachjargon „*Synthetic and Natural Hybrid Coding* (SNHC)“ genannt wird ([Pere02]).

Voraussetzung für eine effiziente Bilddatencodierung in MPEG-4 ist die Kombination anspruchsvoller Methoden der 3D-Computergrafik, der Bildverarbeitung und der Mustererkennung. MPEG-4 bietet mehrere Möglichkeiten an, um sehr hohe Kompressionsraten bei vorgegebener Bildqualität für natürliche Videosequenzen zu erreichen:

- Der Einsatz blockbasierter Verfahren zur Codierung von Bewegungsvektoren, Texturen und Objektkonturen.

- Der Einsatz von 2-dimensionalen Polygonnetzen zur Codierung von Objektkonturen und -bewegungen in Verbindung mit einer Abbildung von Texturen auf das Polygonnetz.

- Der Einsatz von 3-dimensionalen Polygonnetzen zur Codierung von Objektkonturen und -bewegungen in Verbindung mit einer Textur-Abbildung.

Der erste Schritt bei allen drei Verfahren ist die Zerlegung eines Bildes in einzelne Segmente. Die dafür nötigen Bildverarbeitungsalgorithmen werden in MPEG-4 bewusst nicht festgelegt, sondern den Entwicklern eines Codecs[3] überlassen. Typischerweise werden zunächst verschiedene Bildmerkmale extrahiert, wie z.B. Bewegungsvektorfelder (Kapitel 26) und Texturmerkmale (Kapitel 27). Auf dieser Basis erfolgt nun die Bildsegmentierung, für die die Bildverarbeitung eine ganze Reihe von Verfahren zur Verfügung stellt, wie z.B. Minimum-Distance- und Maximum-Likelihood-Klassifikatoren (Kapitel 31), Neuronale Netze (Kapitel 32), oder Fuzzy Logic (Kapitel 33). Da in der Regel nicht nur Einzelbilder segmentiert werden, sondern Bildfolgen, lohnt es sich, Kalman-Filter zur Schätzung der Segmentbewegungen in aufeinander folgenden Bildern einzusetzen (Kapitel 38).

[3]Codec = Software zur *Co*dierung und *Dec*odierung.

Nach dem Segmentierungsschritt trennen sich die Wege der drei Codierverfahren. Beim blockbasierten Verfahren wird genau wie bei transparenten Texturen in der 3D-Computergrafik (Abschnitt 9.3.3) ein vierter Farbkanal, der sogenannte Alpha- oder Transparenz-Kanal, eingeführt. Jeder Bildpunkt des gesamten rechteckigen Bildfeldes bekommt einen Alpha-Wert zugewiesen, wobei Bildpunkte, die zum segmentierten Video-Objekt gehören, den Alpha-Wert 1 (nicht transparent), und alle anderen Bildpunkte den Alpha-Wert 0 (transparent)[4] bekommen. Die Alpha-Bitmasken der einzelnen Video-Objekte werden nun blockweise arithmetisch codiert.

Beim 2D-netzbasierten Verfahren wird jedem Segment ein 2-dimensionales Netz aus verbundenen Dreiecken (Abschnitt 6.2.3.6) nach bestimmten Optimierungskriterien zugewiesen. Die Topologie des Dreiecksnetzes für ein Segment darf sich innerhalb einer Bildsequenz nicht ändern, nur seine Form. Aus diesem Grund genügt es, beim ersten Bild die Anzahl und die 2D-Positionen der Eckpunkte zu codieren, für alle folgenden Bilder des Video-Objekts muss man nur noch die Verschiebungsvektoren für die Eckpunkte codieren. Die Verfolgung der Netz-Eckpunkte in einer Bildfolge, d.h. die Bestimmung der Verschiebungsvektoren kann wieder sehr gut mit Hilfe eines Kalman-Filters durchgeführt werden. Die Bilddaten des Video-Objekts werden als Textur auf das 2D-Dreiecksnetz abgebildet. Da sich die Textur innerhalb der Bildsequenz eines Video-Objekts kaum ändert, können die geringfügigen Texturänderungen zwischen zwei aufeinander folgenden Bildern sehr gut komprimiert werden. Die Codierung der Objekte erfolgt nun in zwei Abschnitten:

- Für das erste Bild einer Folge muss die Geometrie-Information in Form von Polygonnetzen und die Bildinformation in Form von Texturen codiert werden. Dies erfordert zu Beginn eine hohe Datenrate.

- Für die folgenden Bilder der Folge müssen nur noch die Verschiebungsvektoren der Netzeckpunkte und die geringfügigen Texturänderungen codiert werden, so dass bei den Folgebildern nur noch eine relativ niedrige Datenrate nötig ist.

Auf der Empfängerseite werden die Bilder mit Hilfe von 3D-Computergrafik schließlich wieder in Echtzeit erzeugt.

Die Codierung 3-dimensionaler Polygonnetze in MPEG-4 dient eigentlich nicht dem Zweck, noch höhere Kompressionsraten bei natürlichen Video-Objekten zu erreichen, als mit 2D-Netzen, sondern dazu, statische 3D-Modelle für interaktive Computergrafik oder hybride Anwendungen (SNHC) effizient zu codieren und so deren Verbreitung über Kommunikationnetze (Internet, Rundfunk, Mobilfunk etc.) zu fördern. Es gibt allerdings über MPEG-4 hinausgehende Ansätze, bei denen versucht wird, die 3D-Geometrie segmentierter Video-Objekte aus der Bildfolge mit Hilfe eines Kalman-Filters zu rekonstruieren und Bewegungen des 3D-Netzes zu verfolgen (Abschnitt 38.2.3 und [Calv00]). MPEG-4 bietet zwar keinen allgemeinen Ansatz für die Verwendung von 3D-Netzen zur Komprimierung natürlicher Video-Objekte, aber für die wirtschaftlich bedeutenden Anwendungsfelder Bildtelefonie und Videokonferenzen können zwei spezielle 3D-Netzmodelle vordefiniert werden:

[4]Es gibt auch einen Modus in MPEG-4, der 256 (8 bit) verschiedene Transparenzwerte zulässt.

Je ein Prototyp für einen menschlichen Kopf (*face animation*) und einen menschlichen Körper (*body animation*). MPEG-4 lässt zur Bewegung dieser 3D-Netze nur einen eingeschränkten Satz an Animationsparametern zu (68 *face* bzw. 168 *body animation parameters*). Zu Beginn der Übertragung muss deshalb kein komplexes 3D-Netz codiert werden, sondern nur ein Satz von Animationsparametern. Während der laufenden Übertragung müssen nicht mehr die Verschiebungsvektoren aller Eckpunkte des 3D-Netzes codiert werden, sondern nur noch die Änderungen der Animationsparameter. Mit diesem Konzept lassen sich extrem niedrige Bitraten (*very low bitrate coding*) erzielen.

Die geschilderten Codierverfahren bei MPEG-4 und die Entwicklungstendenzen bei den modernen Codierstandards MPEG-7 (ISO/IEC 15938, *Multimedia Content Description Interface*) und MPEG-21 (ISO/IEC 21000, *Multimedia Framework*) zeigen deutlich die immer engere Verquickung zwischen Computergrafik, Bildverarbeitung und Codierung.

2.4 Bildbasiertes Rendering

In der klassischen 3D-Computergrafik werden Objekte durch Polygon-Netze beschrieben (Kapitel 7). Dies ermöglicht die Betrachtung bzw. das Rendering[5] der Objekte aus beliebigen Blickwinkeln, ohne dass sich dabei die polygonale Repräsentation der Objekte ändert. Ein entscheidender Vorteil dieser Methode ist, dass sich der Betrachter interaktiv durch eine 3-dimensionale Szene bewegen kann. Der Nachteil dieser Technik ist, dass eine realitätsgetreue Darstellung von Szenen sehr schwierig bzw. sehr aufwändig ist. Probleme bereiten dabei vor allem komplexe Oberflächen von natürlichen Objekten (Gelände, Pflanzen, Lebewesen etc.). So ist es mit einem rein polygonalen Modell fast unmöglich, z.B. eine echt wirkende Gras- oder Felloberfläche darzustellen.

In der Bildverarbeitung hat man es in der Regel mit 2-dimensionalen Bildern oder Bildfolgen zu tun, die mit einer Kamera aufgenommen wurden. Der Vorteil dabei ist, dass auch komplexeste Szenen unter schwierigsten Lichtverhältnissen exakt eingefangen werden. Ein wesentlicher Nachteil der Fotografie ist, dass die Betrachtung der Szene nur aus einem einzigen Blickwinkel – dem der aufnehmenden Kamera – möglich ist. Eine interaktive Bewegung des Betrachters durch eine 3-dimensionale Szene ist daher nicht realisierbar.

Die Kombination von polygon-orientierter 3D-Computergrafik und Bildverarbeitung ist schon seit langem im Rahmen des „Textur Mappings" (Kapitel 13), also dem „Aufkleben" von Foto-Texturen auf Polygone, etabliert. Das Neue an der Idee des bildbasierten Renderings (*Image Based Rendering* (IBR)) ist der völlige Verzicht auf eine polygonale Repräsentation der Szene. In seiner Reinkultur wird beim bildbasierten Rendering eine Szene, durch die sich der Beobachter beliebig bewegen kann, ausschließlich aus Bilddaten aufgebaut. Dies setzt aber voraus, dass an jeder möglichen Beobachterposition (x, y, z), für jede mögliche Blickrichtung (θ, ϕ) und zu jedem Zeitpunkt (t), der darstellbar sein soll, ein Foto für jede Wellenlänge λ gespeichert werden muss. Man bezeichnet die Summe dieser unendlich vielen Fotos als *plenoptische Funktion* p. Sie ordnet jedem Strahl, der durch das

[5]Rendering = engl. Fachbegriff für Bild mit dem Computer generieren oder zeichnen

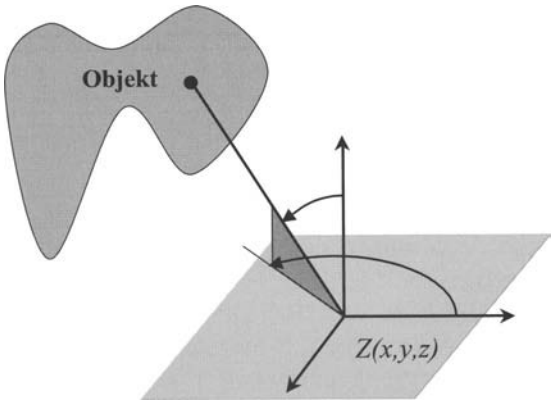


Bild 2.7: Die plenoptische Funktion p definiert für jeden Raumpunkt (x, y, z), jede Blickrichtung (θ, ϕ), jeden Zeitpunkt t und jede Wellenlänge λ eine Lichtintensität I.

Zentrum $Z = (x, y, z)$ einer virtuellen Kamera geht, eine Intensität I zu (Bild 2.7):

$$I = p(x, y, z, t, \theta, \phi, \lambda) \tag{2.1}$$

Die plenoptische Funktion enthält also sieben Parameter, und entspricht daher einer 7-dimensionalen Textur. Selbst bei einer groben Quantisierung der sieben Dimensionen wäre für die Erzielung einer akzeptablen Bildqualität eine gigantische Speicherkapazität[6] von ca. 35.000.000 *TeraByte* erforderlich. Der kontinuierliche Bereich der Spektralfarben wird in der Computergrafik und Bildverarbeitung an drei Stellen (Rot, Grün und Blau) abgetastet. Dadurch lässt sich die plenoptische Funktion als Vektor darstellen, dessen drei Komponenten nur noch von sechs Dimensionen abhängen.
Für statische Szenen reduziert sich die plenoptische Funktion weiter auf fünf Dimensionen:

$$\mathbf{p}(x, y, z, \theta, \phi) = \begin{pmatrix} p_R(x, y, z, \theta, \phi) \\ p_G(x, y, z, \theta, \phi) \\ p_B(x, y, z, \theta, \phi) \end{pmatrix} \tag{2.2}$$

In dem dargestellten Rechenexempel reduziert sich der Speicherplatzbedarf um einen Faktor 1800 (1 Minute bei 30 Hz) auf 20.000 *TeraByte*. Dies sind Größenordnungen, die sicher noch für einige Jahrzehnte eine praktische Realisierung der plenoptischen Funktion verhindern werden. Die plenoptische Funktion ist daher ein idealisierter Grenzfall des bildbasierten Renderings, der die Weiterentwicklung der Computergrafik in eine neue Richtung lenken kann. Von praktischer Relevanz sind daher Spezialfälle der plenoptischen Funktion, die mit der klassischen, polygonalen 3D-Computergrafik kombiniert werden.

[6] Quantisierung des Raums: $x, y, z = 1km$ á $1m = 1000$, der Zeit: $t = 1min$ á $30Hz = 1800$, des Azimutwinkels: $\theta = 360°$ á $0,1° = 3600$, des Elevationswinkels: $\phi = 180°$ á $0,1° = 1800$ und des Wellenlängenspektrums: $\lambda = R, G, B = 3$, d.h. $10^3 \cdot 10^3 \cdot 10^3 \cdot 1,8 \cdot 10^3 \cdot 3,6 \cdot 10^3 \cdot 1,8 \cdot 10^3 \cdot 3 \approx 3,5 \cdot 10^{19}$ Bildpunkte $\approx 35.000.000$ *TeraByte*, bei $1Byte = 256$ Intensitätswerten.

Beschränkt man die plenoptische Funktion auf einen festen Standort, lässt aber den Blickwinkel frei wählbar, so erhält man ein Panorama der Szene. Die plenoptische Funktion vereinfacht sich dabei auf zwei Dimensionen:

$$\mathbf{p}(\theta,\phi) = \begin{pmatrix} p_R(\theta,\phi) \\ p_G(\theta,\phi) \\ p_B(\theta,\phi) \end{pmatrix} \tag{2.3}$$

Das von der Firma Apple eingeführte System „Quicktime VR" legt die Szene in Form eines zylindrischen Panoramafotos ab. Der Beobachter kann sich in (fast[7]) allen Richtungen interaktiv umschauen und er kann zoomen. Allerdings kann er keine translatorischen Bewegungen ausführen. In der 3D-Computergrafik hat sich eine andere Variante des Panoramafotos etabliert: Die „kubische Textur" (Abschnitt 13.4.2). Sie wird dadurch erzeugt, dass man die Umgebung aus der Position des Beobachter im Zentrum des Kubus' sechs mal mit einem Öffnungswinkel von 90° fotografiert oder rendert, und zwar so, dass die sechs Würfelflächen genau abgedeckt werden. Die sechs Einzeltexturen müssen also an den jeweiligen Rändern übergangslos zusammen passen. In Bild 13.20 ist ein Beispiel für eine kubische Textur dargestellt. Der Speicherplatzbedarf ist mit ca. 10 *MegaByte* moderat (sechs 2D-Texturen) und die Erzeugung verhältnismäßig einfach. Kubische Texturen können als „*Sky Boxes*" eingesetzt werden, um den komplexen Hintergrund durch ein Panoramafoto zu ersetzen. Davor können konventionell modellierte 3D-Objekte aus Polygon-Netzen platziert und animiert werden. In der Praxis wird die Panorama-Technik auch bei translatorisch bewegten Beobachtern eingesetzt. Solange sich der Beobachter nur in einem eingeschränkten Bereich innerhalb des Kubus' bzw. Zylinders bewegt und die in den Panoramafotos abgebildeten Objekte relativ weit von der Kamera entfernt waren, sind die Bildfehler, die aufgrund von Parallaxenveränderungen entstehen, praktisch vernachlässigbar. Kubische Texturen eignen sich außerdem ausgezeichnet, um Spiegelungs- oder Brechungseffekte zu simulieren (Abschnitt 13.4.2).

Die Umkehrung des Panoramafotos ist das blickpunktabhängige „*Billboarding*" (Abschnitt 15.4): Der Blickwinkel ist immer fest auf ein Objekt gerichtet, aber der Ort des Beobachters ist (in gewissen Grenzen) frei. Falls die Ausdehnung des betrachteten Objekts klein im Verhältnis zum Abstand Objekt – Beobachter ist, kommt es nur auf den Raumwinkel an, aus dem der Beobachter das Objekt betrachtet. Der Beobachter sitzt gewissermaßen an einer beliebigen Stelle auf einer kugelförmigen Blase, die das Objekt weiträumig umschließt, und blickt in Richtung Objektmittelpunkt. Die Position des Beobachters auf der Kugeloberfläche kann durch die beiden Winkel α und β eines Kugelkoordinatensystems beschrieben werden (anstatt der drei kartesischen Koordinaten x, y, z). Die (eingeschränkte) Blickrichtung wird weiterhin durch die Winkel (θ, ϕ) festgelegt. Damit reduziert sich die

[7]Bei einem zylindrischen Panoramafoto kann der Beobachter einen beliebigen Azimutwinkel wählen, aber nur einen eingeschränkten Bereich an Elevationswinkeln.

statische plenoptische Funktion von fünf auf vier Dimensionen:

$$\mathbf{p}(\alpha,\beta,\theta,\phi) = \begin{pmatrix} p_R(\alpha,\beta,\theta,\phi) \\ p_G(\alpha,\beta,\theta,\phi) \\ p_B(\alpha,\beta,\theta,\phi) \end{pmatrix} \tag{2.4}$$

Das plenoptische Modell des Objekts besteht also aus einer Ansammlung von Fotos des Objekts, die (mit einer gewissen Quantisierung) von allen Punkten der Kugeloberfläche in Richtung des Objekts aufgenommen wurden (Bild 15.15). Dieses plenoptische Objekt kann nun in eine konventionell modellierte 3D-Szene aus Polygon-Netzen integriert werden. Tritt das plenoptische Objekt ins Sichtfeld, wird das Foto ausgewählt, dessen Aufnahmewinkel den Betrachterwinkeln am nächsten liegen. Diese Foto-Textur wird auf ein Rechteck gemappt, das senkrecht auf dem Sichtstrahl steht, und anschließend in den Bildspeicher gerendert. Die Teile der rechteckigen Foto-Textur, die nicht zum Objekt gehören, werden als transparent gekennzeichnet und somit nicht gezeichnet. Damit lässt sich bei komplexen Objekten, die sonst mit aufwändigen Polygon-Netzen modelliert werden müssten, sehr viel Rechenzeit während der interaktiven Simulation einsparen. Allerdings geht dies auf Kosten eines stark vergrößerten Texturspeicherbedarfs in der Größenordnung von *GigaByte*[8]. Mit modernen Codierverfahren (JPEG, MPEG) sind aufgrund der großen Ähnlichkeit benachbarter Bilder hohe Kompressionsfaktoren (ca. 100) möglich, so dass der Texturspeicherbedarf eher akzeptabel wird. Eine weitere Technik zur Reduktion des Speicherplatzbedarfs, bei der Bildverarbeitung und Mustererkennung eine wesentliche Rolle spielt, ist die Interpolation des Blickwinkels [Watt02]. Die Grundidee ist dabei die Gleiche, wie bei der Bewegungskompensation in modernen Codierverfahren (Abschnitt 38.2.3): Man berechnet aus zwei benachbarten Bildern ein Verschiebungsvektorfeld (Abschnitt 26.5), das angibt, durch welchen Verschiebungsvektor jedes Pixel aus dem ersten Bild in das korrespondierende Pixel des zweiten Bildes überführt wird. Jeder Blickwinkel zwischen den Winkeln, unter denen die beiden Bilder aufgenommen wurden, kann jetzt durch eine Interpolation der Verschiebungsvektoren näherungsweise erzeugt werden. Damit lässt sich die Anzahl der Bilder, die für die plenoptische Modellierung eines Objekts notwendig ist, deutlich senken. Im Gegenzug steigt natürlich der Rechenaufwand während der laufenden Simulation wieder an.

Eine ähnliche Technik wie das blickpunktabhängige Billboarding ist das Lichtfeld-Rendering (*Light Fields* bzw. *Lumigraph*, [Watt02]). Im Gegensatz zum Billboarding, bei dem eine bestimmte Anzahl an Fotos von einer Kugeloberfläche in Richtung des Mittelpunkts erzeugt werden, wird die Kamera beim Lichtfeld-Rendering in einer Ebene parallel verschoben. Dabei werden in äquidistanten Abständen Fotos aufgenommen, die zusammen eine 4-dimensionale plenoptische Funktion definieren. Mit gewissen Einschränkungen bzgl. Blickwinkel und Position ist damit eine freie Bewegung einer virtuellen Kamera durch die Szene darstellbar, da für jede zugelassene Position und Orientierung des Beobachters die notwendigen Bilddaten gespeichert sind.

[8]Ein plenoptisches Objekt, bei dem die Einzelbilder eine Auflösung von $1280 \cdot 1024$ Pixel zu je 24 bit besitzen und das im Azimutwinkel 64 mal bzw. im Elevationswinkel 32 mal abgetastet wird, benötigt ca. 8 *GigaByte* an Speicherplatz.

Aufgrund des sehr hohen Speicherplatzbedarfs von blickpunktabhängigen Billboards werden schon seit Längerem bestimmte Spezialfälle eingesetzt, die entweder nur unter gewissen Einschränkungen anwendbar sind, oder bei denen Abstriche bei der Bildqualität hingenommen werden müssen. In vielen Anwendungen kann sich der Beobachter z.B. nur auf einer Ebene bewegen, so dass die Blickwinkelabhängigkeit des Billboards um eine Dimension reduziert werden kann. Die plenoptische Funktion hängt in diesem Fall nur noch von drei Winkeln ab:

$$\mathbf{p}(\alpha, \theta, \phi) = \begin{pmatrix} p_R(\alpha, \theta, \phi) \\ p_G(\alpha, \theta, \phi) \\ p_B(\alpha, \theta, \phi) \end{pmatrix} \tag{2.5}$$

In diesem Fall werden nur Fotos von einem Kreis um das Objekt benötigt (Bild 15.15). Falls das Objekt rotationssymmetrisch ist (wie in guter Näherung z.B. Bäume), oder die Darstellung des Objekts aus einem Blickwinkel ausreicht, genügt zur Modellierung ein einziges Foto des Objekts. Dies ist das klassische Billboard, bei dem eine Foto-Textur mit Transparenz-Komponente (Kapitel 9) auf ein Rechteck gemappt wird, das sich immer zum Beobachter hin ausrichtet (Bild 15.14). Bewegte Objekte mit inneren Freiheitsgraden, wie z.B. Fußgänger, können durch ein Billboard mit animierten Texturen, d.h. einer kurzen Bildsequenz, dargestellt werden. Die plenoptische Funktion eines solchen Modells enthält als Variable die Zeit t: $\mathbf{p} = \mathbf{p}(t, \theta, \phi)$. Einen Mittelweg zwischen vorab gespeicherten blickpunktabhängigen Billboards und zur Laufzeit berechneten 3D-Geometriemodellen stellen *Impostors* dar. Darunter versteht man die Erzeugung der Foto-Textur eines Objekts während der Laufzeit durch klassisches Rendering der Geometrie, sowie das anschließende Mapping auf ein normales Billboard. Solange sich der Beobachter nicht allzu weit von der Aufnahmeposition entfernt hat, kann ein solcher Impostor anstelle des komplexen 3D-Modells für eine gewisse Zeit genutzt werden. Bewegt sich der Beobachter weiter weg, muss ein neuer Impostor gerendert werden.

Wie dargestellt, existiert mittlerweile ein nahezu kontinuierliches Spektrum an Möglichkeiten, was den Anteil an Geometrie bzw. Bilddaten bei der Modellierung von 3D-Szenen betrifft. Inwieweit eher der klassische, polygonbasierte Ansatz oder mehr die plenoptische Funktion für das Rendering genutzt werden, hängt von der Anwendung ab: Sind Geometrie-Daten leicht zugänglich (wie bei CAD-Anwendungen) oder eher Fotos? Welche Hardware steht zur Verfügung? Was sind die Anforderungen an die Darstellungsqualität? Welche Einschränkungen gelten für die Position, den Blickwinkel und die Bewegung des Beobachters? In jedem Fall verschwimmt die strikte Trennlinie zwischen klassischer Computergrafik und Bildverarbeitung bei der Bildgenerierung immer mehr.

In diesen vier Beispielen für den immer enger werdenden Zusammenhang zwischen Computergrafik und Bildverarbeitung tauchen viele Begriffe auf, die dem Leser möglicherweise (noch) nicht geläufig sind. Dies soll jedoch nicht abschrecken, sondern vielmehr Appetit auf die kommenden Kapitel machen.

Teil I

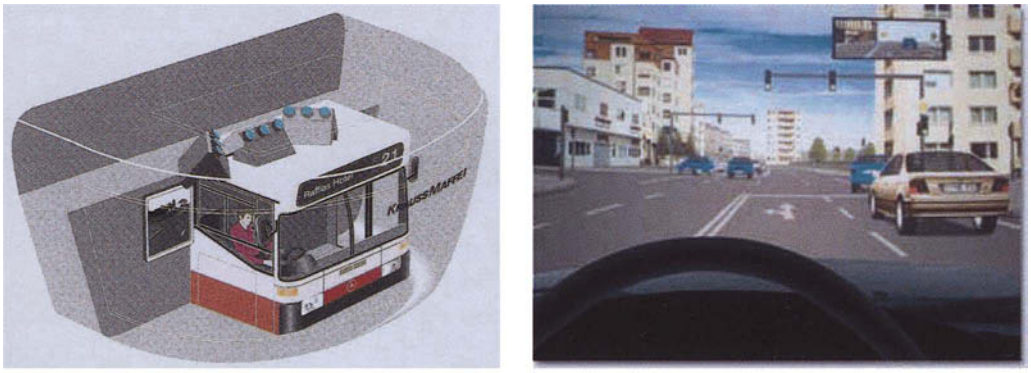
Computergrafik

Kapitel 3

Interaktive 3D-Computergrafik

Die Computergrafik im allgemeinen lässt sich zunächst in zwei Kategorien einteilen: einerseits in die *Interaktive 3D-Computergrafik*, auch *Echtzeit-3D-Computergrafik* genannt, und andererseits in die Nichtechtzeit-3D-Computergrafik. Wie der Name schon sagt, ist das wesentliche Element der Echtzeit-3D-Computergrafik die Interaktivität, d.h. dass die Reaktion des Systems - sprich das computergenerierte Bild - auf Eingaben, wie z.B. von einer Computer-Maus, innerhalb kurzer Zeit erscheint. Idealerweise erfolgt die Reaktion auf Eingaben so schnell, dass ein menschlicher Beobachter Verzögerungen aufgrund der rechenzeitintensiven Bildgenerierung nicht bemerkt.

Am Beispiel eines LKW-Fahrsimulators (Bild 3.1) soll dies genauer erläutert werden.



(a) (b)

Bild 3.1: (a) Typischer Aufbau eines LKW-Fahrsimulators bestehend aus einer Kabinennachbildung und einer Außensichtprojektion. (b) Blick von der Fahrerposition durch die Frontscheibe. Fotos: KraussMaffei Wegmann GmbH.

Wenn ein Fahrer das Lenkrad des LKWs dreht, erwartet er, dass sich die Außensicht entgegen der Drehrichtung des Lenkrads von ihm wegdreht. Dabei muss die gesamte Signalver-

arbeitungskette, vom Sensor für den Lenkradeinschlag über die Weiterleitung des Signals an einen Computer bzw. eine LKW-Dynamik-Simulation, die aus LKW-Geschwindigkeit und Lenkradeinschlag eine neue Position und Lage des LKWs errechnet bis zur Berechnung der neuen Außensicht im Grafikcomputer, so schnell durchlaufen werden, dass der Fahrer keine unnatürliche Verzögerung beim Aufbau des gedrehten Bildes bemerkt. Die Bildgenerierung im Grafikcomputer, meist die größte Komponente in dieser Signalverarbeitungskette, darf also für eine ausreichende Interaktivität nur Bruchteile einer Sekunde benötigen. Folglich müssen die Bildgenerierraten in diesem Bereich der 3D-Computergrafik deutlich über 1 *Hz* (d.h. 1 Bild/Sekunde) liegen. Als Folge dieser starken Rechenzeitbeschränkung, müssen bei der Detailtiefe und dem Realismus von Interaktiver 3D-Computergrafik gewisse Abstriche gemacht werden. Andererseits wurden aus dieser zeitlichen Restriktion heraus eine Reihe von interessanten Techniken entwickelt, auf die später noch genauer eingegangen wird, um trotz der Echtzeit-Anforderung einigermaßen realistätsnahe Bilder generieren zu können.

Im Gegensatz dazu stehen Verfahren der Nichtechtzeit-3D-Computergrafik, wie z.B. aufwändige Raytracing- oder Radiosity-Rendering[1]-Verfahren, deren oberstes Ziel ein möglichst realitätsnahes Aussehen computergenerierter Bilder ist. Die Generierung eines einzigen Bildes für einen Film wie Toy Story kann mehrere Stunden Rechenzeit auf einer sehr leistungsstarken Workstation benötigen. Für die Berechnung aller Bilder eines ganzen Films werden bisher sogenannte Rendering-Farmen, bestehend aus mehreren hundert Multiprozessor-Workstations, über Monate hinweg ausgebucht. Die Bildgenerierraten liegen in diesem Bereich der 3D-Computergrafik typischerweise unter $0,001$ *Hz* (Bilder/Sekunde).

Deshalb ergibt sich als willkürliche Trennlinie zwischen Interaktiver und Nichtechtzeit-3D-Computergrafik ein Wert von 1 *Hz*, d.h.:

- Bildgenerierrate > 1 *Hz* : Interaktive 3D-Computergrafik
- Bildgenerierrate < 1 *Hz* : Nichtechtzeit 3D-Computergrafik

In diesem Buch wird der Schwerpunkt auf die *Interaktive 3D-Computergrafik* gelegt.

Die Interaktive 3D-Computergrafik lässt sich je nach Anforderungen weiter in harte und weiche Echtzeit unterteilen.

3.1 Harte Echtzeit

Unter harter Echtzeit-Anforderung versteht man eine Bildgenerierrate ≥ 60 *Hz*. Typische Anwendungen, bei denen harte Echtzeitanforderungen vorliegen, sind Ausbildungssimulatoren, wie z.B. Fahr- oder Flugsimulatoren. Zum besseren Verständnis wird das obige Beispiel eines LKW-Simulators noch einmal etwas intensiver betrachtet: Was würde passieren, wenn die Bildgenerierrate z.B. nur halb so hoch, also 30 Hz wäre? Bei einer Bewegung durch eine virtuelle Landschaft würden Doppelbilder auftreten, da in diesem Fall

[1]Rendering = engl. Fachbegriff für Bild mit dem Computer generieren oder zeichnen

die Bildschirm-Refresh-Rate mit 60 Hz doppelt so groß wäre[2], wie die Bildgenerierrate, so dass jedes aus einer bestimmten Position und Lage berechnete Bild zweimal am Bildschirm dargestellt werden müsste, das erste Mal an der „richtigen" Position und das zweite Mal an der „falschen" (Bild 3.2).

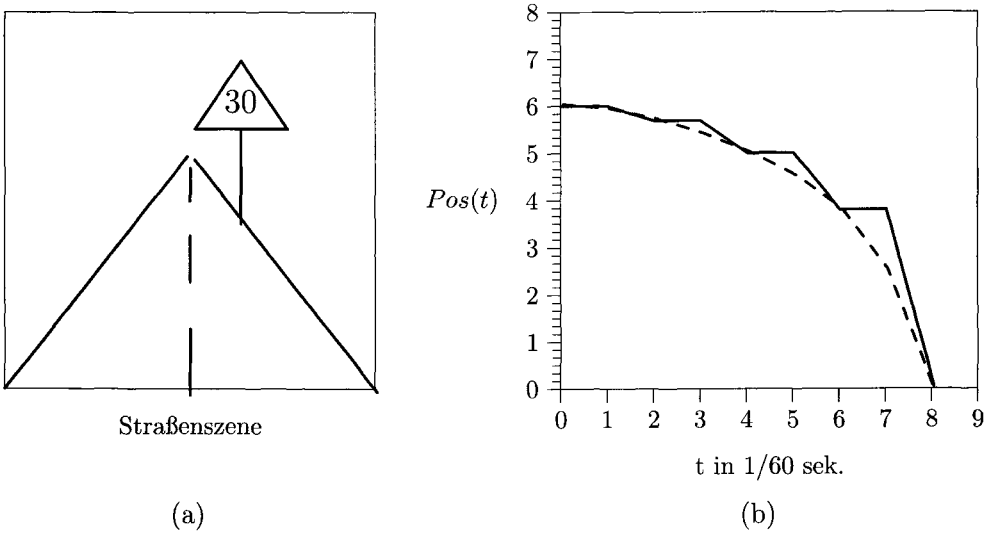


Bild 3.2: Erklärung der Doppelbilder bei 30 *Hz* Bildgenerierrate:
(a) Schematische Straßenszene mit Verkehrsschild, durch die sich der Beobachter bewegt
(b) Position des Verkehrsschildes auf der Fluchtgeraden in Abhängigkeit von der Zeit. Gestrichelt (60 Hz Bildgenerierrate): exakte Positionen des Verkehrsschildes bei jedem Refresh-Zyklus des Bildschirms. Durchgezogen (30 Hz Bildgenerierrate): man sieht Doppelbilder, da bei jedem zweiten Refresh-Zyklus $(1, 3, 5, 7)$ das vorherige Bild und damit das Verkehrsschild an den falschen Ort projiziert wird. Da die Relativgeschwindigkeit des Verkehrsschildes im Bildfeld kurz vor dem Vorbeifahren am höchsten ist, tritt der verwischende Effekt der Doppelbilder leider genau an der Stelle am stärksten auf, an der der Inhalt des Verkehrsschildes aufgrund der perspektivischen Vergrößerung am ehesten erkennbar wäre.

Da die Relativgeschwindigkeit von Objekten im Bildfeld kurz vor dem Vorbeifahren am höchsten ist, tritt der verwischende Effekt der Doppelbilder unglücklicherweise genau an der Stelle am stärksten auf, an der die Objekte aufgrund der perspektivischen Vergrößerung am ehesten erkennbar wären. Bei Ausbildungssimulatoren ist das Erkennen von Verkehrszeichen eine Grundvoraussetzung, so dass allein schon aus diesem Grund harte Echtzeitanforderungen gelten. Der Leser könnte sich jetzt fragen, warum man denn im

[2]30 Hz Bildschirm-Refresh-Rate sind bei den heutigen Bildschirmen aus guten Grund meist nicht mehr einstellbar, da in diesem Fall sehr unangenehme Helligkeitsschwankungen Auge und Gehirn des Betrachters irritieren und ermüden würden. Um diesen Effekt zu minimieren, wird versucht, mit möglichst hohen Bildschirm-Refresh-Raten zu arbeiten, wie z.B. bei den sogenannten „100Hz-Fernsehern"

Fernsehen keine Doppelbilder sieht, obwohl doch in diesem Fall die Bilder nur mit Raten von ca. 25 *Hz* auf den Bildschirm geworfen werden? Erstens, weil auf europäischen Bildschirmen, die nach der PAL-Norm arbeiten, Halbbilder (d.h. jede zweite Bildzeile) mit 50 *Hz* auf den Bildschirm geworfen werden, so dass nach zwei Halbbildern alle Bildzeilen erneuert sind, was einer Vollbildrate von 25 *Hz* entspricht. Dadurch werden Objekte mit hohen Relativgeschwindigkeiten in der Bildebene verzerrt. Zweitens, weil sich bei Aufnahmen mit Fernsehkameras Objekte mit hohen Relativgeschwindigkeiten in der Bildebene innerhalb der Blendenöffnungszeit ein Stückchen weiter bewegen, so dass sie über das Bild verwischt werden. Diese beiden Effekte bewirken, dass man beim Fernsehen keine Doppelbilder wahrnehmen kann. Allerdings sind wegen der Verwischungen z.B. auch keine Verkehrsschilder lesbar, was aber meist nicht auffällt.

Ein anderes Beispiel für harte Echtzeitanforderungen sind *Hardware-In-The-Loop* (HIL)-Simulationen , wie z.B. beim Test der gesamten Lenkregelschleife eines Flugkörpers in Echtzeit. Die Regelschleife besteht in diesem Fall zum Einen aus den entsprechenden Komponenten des Flugkörpers, wie Sensorik (z.B. Infrarot-Kamera), Informationsverarbeitung (z.B. Bildverarbeitung und Mustererkennung, Lenk-, Regel- und Navigations-Software) sowie evtl. noch Aktorik (z.B. Ruder und Triebwerk), und zum Anderen aus den Komponenten der Umwelt-Simulation, wie z.B. der Flugdynamik-Simulation und den computergenerierten Infrarot-Bildern als Input für die Kamera. Falls der Infrarot-Suchkopf beispielsweise mit einer Rate von 60 *Hz* Bilder aufnimmt bzw. verarbeitet, müssen natürlich auch die synthetischen Bilder auf dem Grafikcomputer mit dieser Rate erzeugt werden, und zwar aus der jeweils aktuellen Position und Lage des Suchkopfes. Sobald auch nur ein einziges Bild nicht rechtzeitig fertig wird (ein sogenannter *Frame-Drop*), ist der gesamte Simulationslauf unbrauchbar, da aus der Bewegung des Ziels im Bildfeld die entsprechenden Lenkbefehle abgeleitet werden. Bei einem *Frame-Drop* wird entweder das vorher berechnete Bild zum zweiten Mal in den Bildverarbeitungsrechner eingespeist oder gar keines, was zu einem verfälschten oder gar keinem Lenkbefehl führt. In jedem Fall wird das simulierte Verhalten des Flugkörpers vom realen Verhalten abweichen, was natürlich unerwünscht ist.

Ein weiterer wichtiger Grund für hohe Bildgenerierraten ist die sogenannte „Transport-Verzögerung" (*transport delay*). Darunter versteht man die Zeitspanne für den Durchlauf der Signalverarbeitungskette von der Eingabe (z.B. Lenkradeinschlag) bis zur Ausgabe (computergeneriertes Bild z.B. auf dem Bildschirm oder der Leinwand). Erfahrungswerte zeigen, dass die Wahrnehmungsschwelle von Menschen bei ca. 50 *msec* liegt (bei professionellen Kampfpiloten, die besonders reaktionsschnell sind, liegt die Wahrnehmungsschwelle bei ca. 30 *msec*). Bei Simulatoren, die mit einem Berechnungstakt von 60 *Hz* arbeiten, wird meist ein Zeittakt (*Frame*), d.h. beim betrachteten Beispiel $1/60 \approx 0,0166$ *sec* für die Sensorsignaleingabe und die Dynamik-Simulation benötigt, mindestens ein weiteres *Frame* für die Berechnung des Bildes im Grafikcomputer und noch ein weiteres *Frame* bis es vom Bildspeicher (*Framebuffer*) des Grafikcomputers ausgelesen und z.B. auf den Bildschirm oder die Leinwand geworfen wird (Kapitel 14). Insgesamt treten also mindestens drei *Frames* Transport-Verzögerung auf, d.h. bei einem 60 *Hz*-Takt also 50 *msec*, so dass man gerade noch an der Wahrnehmungsschwelle des Menschen liegt. Befindet sich in der Informationsverarbeitungsschleife kein Mensch, sondern, wie z.B. bei einer HIL-Simulation, eine

Kamera und ein Bildverarbeitungsrechner, wirkt sich die Transport-Verzögerung direkt als zusätzliche Totzeit in der Lenkregelschleife des simulierten Flugkörpers aus. Bisher haben sich zwei (auch kombinierbare) Strategien etabliert, um diesen schädlichen Simulationseffekt abzumildern: einerseits noch höhere Bildgenerierraten (z.B. 120 *Hz* oder 180 *Hz*), so dass sich die Transport-Verzögerung entsprechend auf 25 *msec* bzw. 16,6 *msec* reduziert, und andererseits der Einbau von Prädiktionsfiltern[3] in die Dynamik-Simulation.

3.2 Weiche Echtzeit

Unter weicher Echtzeitanforderung versteht man eine Bildgenerierrate im Bereich 1 *Hz* – 60 *Hz*. In diesem Bereich wird die Echtzeitanforderung je nach Anwendung immer weniger wichtig: möchte man bei Unterhaltungs-Simulatoren (PC-Spiele, Spielekonsolen, Simulatoren in Spielhallen oder Museen) oder Virtual Reality (*CAVEs, Power-Walls, Virtual Showrooms*) möglichst noch 30 *Hz* Bildgenerierrate erreichen, so begnügt man sich aufgrund der meist extrem komplexen Modelle bei CAD-Tools (*Computer Aided Design*), Architektur-Visualisierung oder Medizinischer Visualisierung oft schon mit 15 *Hz* Bildgenerierrate oder sogar darunter. Außerdem sind vereinzelte *Frame-Drops*, die sich als kurzes „Ruckeln" in einer Bildsequenz äußern, zwar nicht erwünscht, aber tolerabel.

Ein zur HIL- bzw. Echtzeit-Simulation komplementäres Werkzeug in der Entwicklung technischer Systeme ist die Nichtechtzeit-Simulation (auch „Mathematisch Digitale Simulation" oder kurz „MDS" genannt), die vor allem zur Auslegung und zum Leistungsnachweis dient. Bei der MDS wird, im Gegensatz zur *Hardware-In-The-Loop*-Simulation, jede Komponente des technischen Systems durch ein digitales Modell simuliert, und zwar in Nichtechtzeit. Dies kann bedeuten, dass die simulierte Zeit schneller oder langsamer als die reale Zeit abläuft, je nach Komplexität des Simulationsmodells eben gerade so lange, wie der eingesetzte Computer für die Abarbeitung benötigt. Für die Simulationskomponente „Bildgenerierung" bedeutet dies, dass es keine harte zeitliche Beschränkung für die Generierung eines Bildes gibt, sondern dass die restlichen Simulationskomponenten einfach warten, bis das Bild vollständig gerendert ist. Dadurch hat man in der MDS die Möglichkeit, auch beliebig detailreiche Szenarien in die Simulation zu integrieren, so dass die computergenerierten Bilder auch höchsten Anforderungen, wie z.B. im Leistungsnachweis, genügen. Bei der MDS gibt es folglich keine harte Anforderung an die Bildgenerierrate, aber dennoch darf es nicht Stunden (wie in der Filmproduktion) dauern, bis ein Bild gerendert ist, denn sonst würde eine Simulation von zehn Minuten Echtzeit über vier Jahre dauern!

[3]Ein Prädiktionsfilter liefert einen Schätzwert für den zukünftigen Systemzustand (Kapitel 38).

Kapitel 4

Anwendungen interaktiver 3D-Computergrafik

Die enorme Leistungssteigerung von Grafikcomputern hat dazu geführt, dass heutzutage 3D-Computergrafik in nahezu allen Anwendungsgebieten von Computern Einzug gehalten hat. In diesem Kapitel wird auf eine Auswahl derzeitiger Anwendungsbeispiele interaktiver 3D-Computergrafik eingegangen, die einen gewissen Überblick verschaffen soll.

4.1 Ausbildungs-Simulation

Die erste starke Triebfeder für die Entwicklung der Computergrafik war die Flugsimulation im militärischen Umfeld (Bild 4.1-a). Zwei wesentliche Argumente sprachen für den Einsatz von Flugsimulatoren für die Ausbildung bzw. Weiterbildung von Piloten: einerseits das Kostenargument, denn bei Kampfflugzeugen lag bzw. liegt der Systempreis in der Größenordnung von 100 Millionen Euro, so dass sich Simulatoren, die zu Beginn ca. die Hälfte dessen kosteten, ganz einfach rentierten, und andererseits die Problematik, dass sich gewisse kritische Situationen in der Realität nicht trainieren lassen, wie z.B. eine Notlandung oder ein Luftkampf. Heutzutage werden Flugsimulatoren nicht nur für die Ausbildung, sondern u.a. auch für die Missionsvorbereitung genutzt: steht ein Kampfeinsatz bevor, wird zunächst das entsprechende Gelände mit Hilfe von Aufklärungsdaten detailgetreu nachgebildet, d.h. die Oberfläche wird mit Bewuchs, Bebauung, feindlichen Abwehrstellungen und Zielen in 3D-Computergrafik modelliert, anschließend wird die Mission vorab im Flugsimulator so oft trainiert, bis alle Handlungsabläufe sitzen, und erst danach wird der reale Einsatz geflogen. Mittlerweile ist die Flugsimulation in alle Bereiche der Luft- und Raumfahrt vorgedrungen: für nahezu alle zivilen Flugzeugtypen gibt es Flugsimulatoren für die Aus- und Weiterbildung, die praktisch jede größere Fluggesellschaft in Betrieb hat (Bild 4.1-b); entsprechende Flugsimulatoren gibt es selbstverständlich auch für Helicopter, bemannte und unbemannte Raketen, Luftschiffe und andere Flugobjekte.

Der Einsatz von 3D-Computergrafik in der Fahrsimulation begann Anfang der neunziger Jahre des letzten Jahrhunderts, nachdem der Preis für leistungsfähige Grafikcomputer in

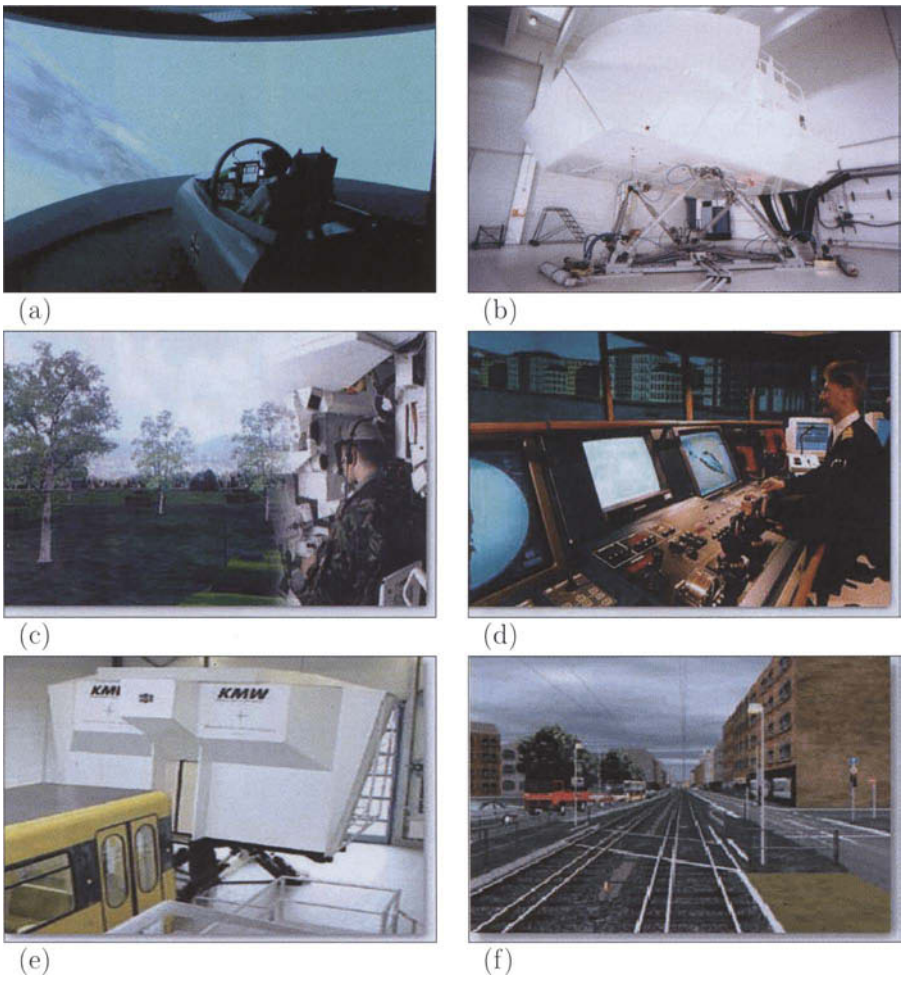
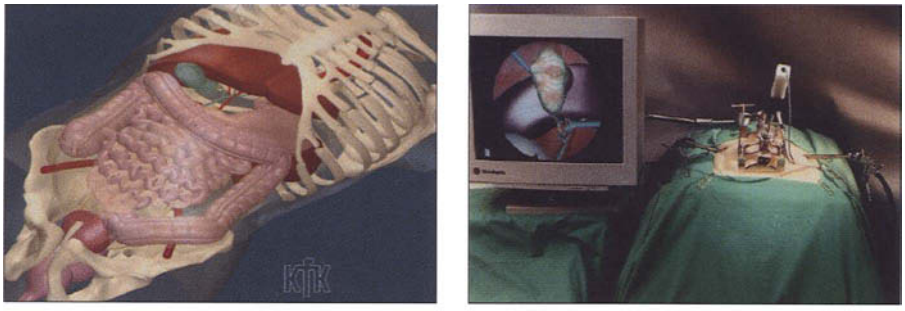


Bild 4.1: Flug- und Fahrsimulatoren: (a) Dome-Simulator Eurofighter Typhoon. Quelle: EADS-Military Aircraft. (b) Airbus A330/A340 Flugsimulator. Quelle: Zentrum für Flugsimulation Berlin. (c) Panzer-Simulator Challenger. (d) Schiffs-Simulator. Quelle: Rheinmetall Defence Electronics. (e) Ausbildungssimulator für die Stuttgarter Stadtbahn. (f) Blick von der Fahrerposition. Quelle: Krauss-Maffei Wegmann.

die Größenordnung von einer Million Euro gefallen war. Damit rentierte sich der Ersatz von realen Fahrzeugen, deren Systempreis in der Größenordnung von zehn Millionen Euro lag, durch entsprechend günstigere Simulatoren. Waren es zu Beginn wieder militärische Fahr- und Gefechtssimulatoren für geländegängige gepanzerte Fahrzeuge (Bild 4.1-c), sowie für Schiffe (Bild 4.1-d) und Unterseeboote, kamen später Simulatoren für Schienenfahrzeuge wie Lokomotiven, U- und S-Bahnen, Straßenbahnen etc. hinzu (Bild 4.1-e,f). Wesentliche Argumente für den Einsatz von Schienenfahrzeug-Simulatoren sind neben geringeren Ko-

sten die Entlastung des meist dicht befahrenen Schienennetzes vom Ausbildungsbetrieb und das flexible Training von Gefahrensituationen. Zuletzt wurde auch der Bereich großer oder spezieller Straßenfahrzeuge wie LKW (Bild 3.1), Polizei- und Notdienstfahrzeuge durch die Aus- und Weiterbildungssimulation erschlossen.

Die Ausbildung und das Training von Medizinern wie z.B. Chirurgen und Orthopäden ist sehr zeitaufwändig, teuer und nicht selten mit einem erheblichen Risiko für die ersten Patienten verbunden. Deshalb wird in der Medizin von der theoretischen Grundausbildung (z.B. interaktive Anatomie-Atlanten, Bild 4.2-a) bis hin zum Erwerb handwerklicher Fähigkeiten (z.B. Laparoskopietrainer, Bild 4.2-a) verstärkt auf Simulation mit interaktiver 3D-Computergrafik gesetzt. Ein Überblick über Virtual Reality Techniken in der Aus- und Weiterbildung bei minimalinvasiven Eingriffen ist in ([Cakm00]) zu finden. Bei Schönheitsoperationen ist es mittlerweile üblich, dass die gewünschten Veränderungen vor der Operation in einer 3D-Computergrafik interaktiv von Arzt und Patient modelliert werden.



(a) (b)

Bild 4.2: 3D-Grafik in der Medizinischen Simulation: (a) Anatomie-Atlas. (b) Laparoskopietrainer. Quelle: U.Kühnapfel, Forschungszentrum Karlsruhe IAI.

4.2 Entwicklungs-Simulation

Ein extrem vielfältiges Gebiet für Anwendungen der interaktiven 3D-Computergrafik ist die Entwicklungs-Simulation. Da die Entwicklung komplexer technischer Produkte wie z.B. Autos, Flugzeuge oder Fabriken immer Milliarden-Euro-Projekte sind, die möglichst schnell, flexibel und kostengünstig bewältigt werden sollen, haben sich hier Verfahren des *Rapid* bzw. *Virtual Prototyping* durchgesetzt. Die Entwicklung eines virtuellen Prototypen läuft dabei in der Regel über mehrere Stufen. Zunächst werden in einer Vorsimulation verschiedene Produktideen entwickelt, die nach bestimmten Selektionskriterien in ein

Grobkonzept münden. Anschließend werden die einzelnen Komponenten des Produkts mit Hilfe von *Computer-Aided-Desing* (CAD)-Software als virtuelle Prototypen entwickelt, unabhängig davon, ob es sich dabei um einen Motor, ein Chassis oder einen elektronischen Schaltkreis handelt. Jede Komponente für sich benötigt schon die verschiedensten Simulationen und Visualisierungen, wie am Beispiel der Entwicklung eines Fahrzeug-Chassis verdeutlicht werden soll: neben der rein geometrischen Konstruktion in einem CAD-Werkzeug (Bild 4.3-a), werden statische und dynamische Kraftwirkungen ebenso wie elektromagnetische Abschirmeigenschaften des Chassis durch Falschfarben visualisiert, die Ergonomie beim Sitzen bzw. Ein- und Aussteigen wird durch virtuelle Cockpits und Personen dargestellt (Bild 4.3-b), die Verformbarkeit des Chassis wird in virtuellen Crash-Tests ermittelt (Bild 4.3-c) und die Attraktivität des Aussehens muss vorab mit Hilfe hochwertiger Grafiksimulationen bei den Käufern erkundet werden (Bild 4.3-d).

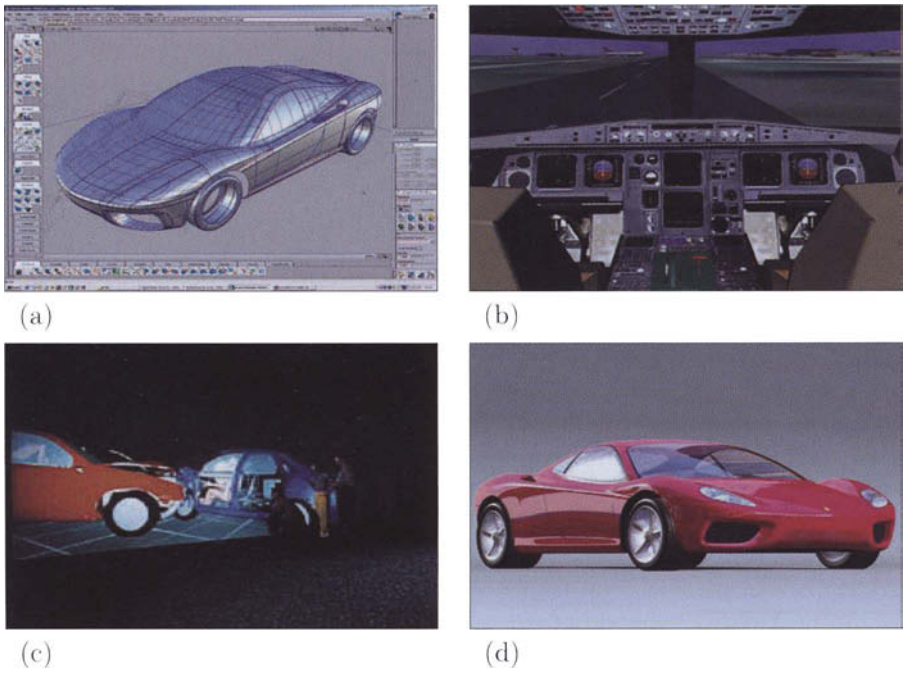


Bild 4.3: 3D-Grafik in der Entwicklungssimulation: (a) Computer-Aided Design eines Ferrari 360. (b) Virtuelles Cockpit in der Flugzeugentwicklung. (c) virtueller Crash-Test. (d) Qualitativ hochwertige 3D-Grafik mit Spiegelungseffekten.

Die virtuellen Komponenten des Produkts werden anschließend in einer Mathematisch Digitalen Simulation (MDS) in Nichtechtzeit zu einem detaillierten Gesamtsystem zusammengefügt, getestet und optimiert (Abschnitt 2.2). Bis zu diesem Zeitpunkt ist kein einziges

Stück Metall bearbeitet worden, das 3-dimensionale Aussehen und Verhalten des Produktes wird bis dahin ausschließlich durch die Computergrafik bestimmt. Aber auch nachdem die ersten Komponenten auf der Basis von CAD-Plänen in Hardware gefertigt sind, behält die Simulation und damit auch die 3D-Computergrafik ihre wesentliche Rolle im weiteren Entwicklungsprozess, denn einerseits werden allfällige Optimierungsschritte immer zuerst in der Simulation getestet, bevor sie in Hardware realisiert werden, und andererseits wird mit der realen Verfügbarkeit erster Komponenten eine *Hardware-In-The-Loop* Simulation aufgebaut, um das Zusammenwirken der einzelnen Komponenten in Echzeit überprüfen zu können.

In zunehmendem Maße wird auch für den Nachweis der Leistungsfähigkeit von Produkten die Simulation eingesetzt: z.B. bei Lenkflugkörpern gingen auf Grund des breiten Anforderungsspektrums (unterschiedliche Einsatzgebiete kombiniert mit verschiedenen Wetterbedingungen und Zieltypen bzw. -manövern) und der geringen Stückzahlen oft 20-40% der gesamten Produktion allein für den Leistungsnachweis verloren, da für jede Anforderungskombination mindestens ein Flugkörper verschossen werden musste. Deshalb begnügt man sich mit wenigen realen Flugversuchen, anhand derer die Simulationen validiert werden. Das gesamte Spektrum der Anforderungen wird dann mit Hilfe von vielen virtuellen Flugversuchen abgedeckt. Damit einer solchen Simulation von der Auftraggeberseite genügend Vertrauen entgegen gebracht wird, ist eine sehr aufwändige und detaillierte Nachbildung ausgewählter Einsatzgebiete in Echtzeit-3D-Computergrafik nötig.

4.3 Unterhaltung

Nachdem bis Mitte der 1990iger Jahre vor allem der schmale Markt für Simulation und *Computer-Aided-Design* mit seinen relativ wenigen professionellen Anwendern die Haupttriebfeder für die Leistungssteigerungen in der 3D-Computergrafik waren, dominiert seither der Millionen-Markt für Computerspiele am Heim-PC, auf Spielekonsolen und neuerdings auch auf Handys die geradezu sprungartige Weiterentwicklung. Computerspiele wie der *Microsoft Train Simulator* oder der *Flight Simulator* mit ihren länderspezifischen und fotorealistischen Szenarien stehen heute professionellen Simulatoren in Punkto 3D-Computergrafik kaum noch nach. Aufgrund des rapiden Preisverfalls für Grafikkarten - eine vergleichbare Leistung einer aktuellen 3D-Grafikkarte für 500 Euro kostete vor zehn Jahren noch ca. eine Million Euro - eröffnet sich ein riesiges Potential für eine schnelle Ausbreitung weiterer Computergrafikanwendungen, nicht nur in der Unterhaltungsindustrie.

4.4 Telepräsenz

Interaktive 3D-Computergrafik erlaubt auch neue Möglichkeiten der Überwindung von Raum und Zeit. Ein Anwendungsbeispiel dafür sind ferngesteuerte Roboter: die Firma Norske Shell z.B. betreibt eine Gaspipeline in einem Tunnel unter der Nordsee, die von der Bohrplattform im Meer bis zum Festland reicht. Wegen des extrem hohen Drucks unter

dem Meeresgrund ist der Tunnel geflutet und somit für Menschen unzugänglich. Notwendige Inspektionsarbeiten an der Pipeline werden mit Hilfe ferngesteuerter Inspektionsroboter durchgeführt, die mit verschiedenen Sensoren zur Fehlererkennung und Navigation ausgestattet sind. Aus den Sensorsignalen kann u.a. Position und Lage der Inspektionsroboter im Tunnel berechnet werden. Die Teleoperateure an Land erhalten daraus mit Hilfe von Grafikcomputern in Echtzeit eine 3D-Sicht aus dem Blickwinkel des Roboters und können diesen somit fernsteuern. Das gesamte 3D-Szenario wurde aus den Tunnel-Bauplänen generiert. Ähnliche Anwendungen der Telerobotik gibt es auch bei Einsätzen in kontaminiertem Gelände, im Weltraum, oder in der Fern- und Präzisionschirurgie.

Der Nutzen von Interaktiver 3D-Computergrafik zur Überwindung von Raum und Zeit wird auch deutlich bei Virtuellen Stadtbesichtigungen (interaktiver Stadtrundgang z.B. durch das Mexiko City von Heute bis zurück zum Mexiko City der Azteken), virtuellen Museen (z.B. Besichtigung der 3200 Jahre alten Grabkammer der ägyptischen Königin Nefertari, die für Besucher gesperrt ist), oder virtuellen Verkaufsflächen (z.B. Zusammenstellung eines Personenwagens aus der Vielfalt möglicher Ausstattungsvarianten und interaktive 3D-Darstellung).

4.5 Daten-Visualisierung

In der Exploration großer Datenmengen aus den Bereichen Wissenschaft, Forschung und Entwicklung, sowie Finanz- und Wirtschaftswesen hat sich die interaktive 3D-Visualisierung als Standardmethode etabliert. Als Beispiel hierfür seien große Ölkonzerne genannt, die zur Erhöhung der Entdeckungswahrscheinlichkeit großer Ölvorkommen aus seismischen und Bohrkern-Daten die Echtzeit-3D-Computergrafik als Erfolgsfaktor zu schätzen gelernt haben. Nachdem in den letzten Jahrzehnten in der Chemie zunehmend erkannt wurde, dass nicht nur die atomare Zusammensetzung von Molekülen und Verbindungen, sondern vor allem auch die räumliche Anordnung der einzelnen Atome innerhalb eines Moleküls eine entscheidende Rolle für die chemischen Eigenschaften spielt, wurde die interaktive 3D-Visualisierung als Schlüssel für ein tieferes Verständnis benutzt. Um einen noch besseren 3-dimensionalen Eindruck zu bekommen, wird hier teilweise auch die sogenannte „*CAVE*"-Technik (*Cave Automatic Virtual Environment*, Bild 4.4-a,b) genutzt, bei der man durch Stereoprojektion auf die umgebenden Wände eines Würfels und einer optionalen Kraftrückkopplung beim Hantieren mit einem Molekül ein multimodales Gefühl für dessen räumliche Eigenschaften bekommt.

Architekten haben immer ein gewisses Vermittlungsproblem: ihre Planungen, egal ob für Inneneinrichtungen, Gebäude oder ganze Stadtviertel, sind immer 2-dimensionale Schnitte oder Ansichten, bei denen ungeschulte Laien sich meist nicht gut vorstellen können, wie das Ganze später einmal in der 3-dimensionalen Wirklichkeit aussehen wird. Da mit Interaktiver 3D-Computergrafik diese Vorstellungsprobleme elegant behoben werden können, hat sich der Bereich der Architekturvisualisierung als fruchtbares Anwendungsgebiet erwiesen (Bild 4.4-c,d).

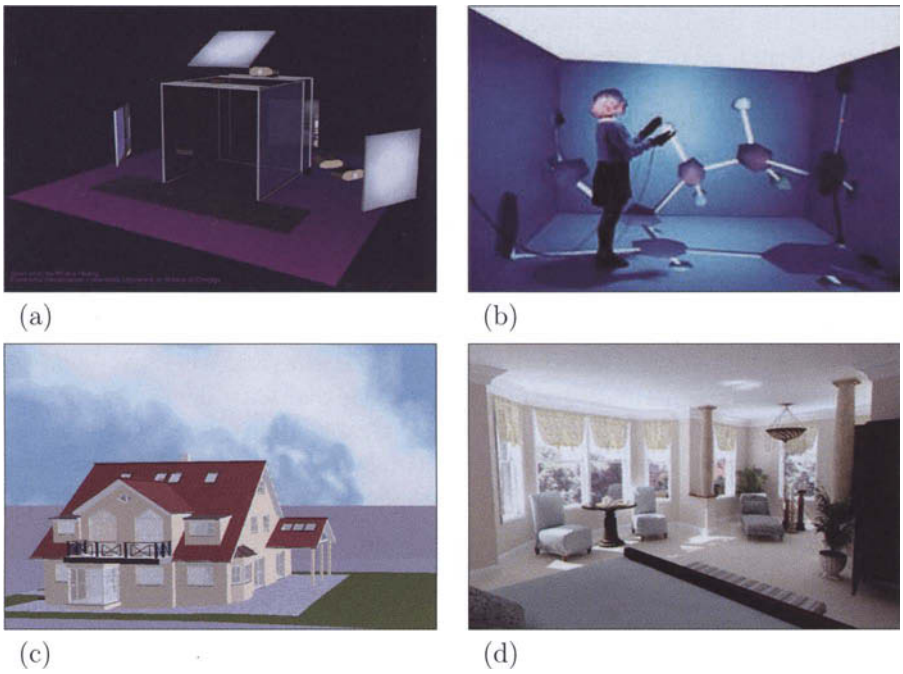


Bild 4.4: 3D-Grafik in der Daten-Visualisierung: (a) Prinzip einer CAVE (Cave Automated Virtual Environment). (b) Molekül-Visualisierung in einer CAVE. (c) Architektur-Visualisierung eines geplanten Gebäudes. (d) Architektur-Visualisierung einer geplanten Inneneinrichtung.

4.6 Augmented Reality

Sehr interessant ist auch die Überlagerung von realen wahrgenommenen Szenen mit computergenerierten Bildern z.B. mit Hilfe eines halbdurchlässigen Glases, im Fachjargon auch „*Augmented Reality*" genannt. Damit können wichtige Informationen in das Gesichtfeld z.B. von Fahrern oder Piloten eingeblendet werden, ohne dass diese ihren Blick von der Frontscheibe abwenden müssen. Am Lehrstuhl für Flugmechanik und Flugregelung der Technischen Universität München wurde ein System zur Verbesserung der Flugführung bei schlechter Außensicht (z.B. Nacht oder Nebel) entwickelt ([Kloe96]), welches dem Piloten in Echtzeit eine computergenerierte Landschaft und einen Flugkorridor einblendet. Die 3-dimensionale Oberfläche der überflogenen Landschaft ist ja vorab bekannt und kann somit in ein entsprechendes Computergrafik-Modell gebracht werden. Aus der navigierten Position und Lage des Flugzeugs kann folglich die korrekte Aussensicht mit einem Grafikcomputer in Echtzeit berechnet werden (Bild 4.5).

(a) (b)

Bild 4.5: 3D-Grafik und *Augmented Reality*: (a) Testflug mit synthetischer Sicht des Piloten (im Experiment mit rein virtueller Sicht über ein *Helmet Mounted Device* und noch nicht über ein halbdurchlässiges Glas). (b) Computergeneriertes Bild beim Testflug übers Altmühltal mit Flugkorridor. Quelle: Lehrstuhl für Flugmechanik und Flugregelung der Technischen Universität München.

Im Forschungsbereich der Siemens AG gibt es ein Projekt, in dem *Augmented Reality* in einem etwas weiteren Sinne eingesetzt wird: im Mobilfunk. Die Idee dabei ist, auf einem Pocket-PC oder Handy, welches mit einer Navigationsausrüstung versehen ist, ein schematisches 3D-Bild der augenblicklichen Szene mit Zusatzinformationen einzublenden. So könnte ein Tourist in einer Stadt nicht nur auf einem Rundgang geführt werden, sondern er könnte z.B. zu jedem Gebäude beispielsweise auch noch interessante Hintergrundinformationen abfragen.

4.7 Datenübertragung

In der Datenübertragung wie z.B. beim Codierstandard MPEG-4 ist die Interaktive 3D-Computergrafik einerseits ein Werkzeug, das eingesetzt wird, um zu noch höheren Kompressionsraten bei der Codierung von natürlichen Videobildern zu kommen, und andererseits werden die Inhalte von Interaktiver 3D-Computergrafik selbst zum Objekt der Codierung und Datenübertragung. Dieses Konzept erlaubt die Überlagerung und die effiziente Codierung von synthetischen und natürlichen Bildinhalten, im MPEG-Fachjargon auch „*Synthetic and Natural Hybrid Coding* (SNHC)“ genannt ([Pere02]).

Da in Zukunft immer mehr internetbasierte 3D-Grafikanwendungen mit synthetischem Inhalt zu erwarten sind, wie z.B. vernetzte Computerspiele oder 3D-Online-Shopping, enthält der MPEG-4 Standard auch Werkzeuge, um Polygon-Netze und Texturen, also die wesentlichen Inhalte von 3D-Computergrafik, in verschiedenen Detaillierungsstufen effizient zu codieren. Damit wird abhängig von der Leistungsfähigkeit der Datenleitung und der Grafikhardware beim Endgerät eine adäquate Bildqualität für netzwerkbasierte, interaktive 3D-Grafikanwendungen sichergestellt.

Die Grundidee beim Einsatz von Interaktiver 3D-Computergrafik bei der Codierung von natürlichen Videobildern besteht darin, einzelne Objekte aus der Videoszene mit Hilfe von Bildverarbeitungsalgorithmen zu segmentieren, deren 2D- oder 3D-Geometrie zu rekonstruieren und eventuelle Bewegungen zu verfolgen. Die Codierung der Objekte erfolgt nun in drei Teilen:

- Speicherung der Geometrie-Information in Form von Polygonnetzen.
- Speicherung der Bildinformation in Form von Texturen, die auf die Polygone gemappt werden.
- Speicherung von Bewegungen, in dem Verschiebungsvektoren für die Polygoneckpunkte (die sogenannten Vertices) angegeben werden.

Die Übertragung der so codierten Bildsequenzen erfordert nur zu Beginn eine hohe Datenrate, bis die gesamten Geometrie- und Texturinformationen übermittelt sind, danach ist nur noch eine relativ niedrige Datenrate nötig, um die inkrementellen Verschiebungsvektoren zu senden. Auf der Empfängerseite werden die Bilder mit Hilfe von 3D-Computergrafik schließlich wieder in Echtzeit erzeugt (Abschnitt 2.3).

Kapitel 5

Einführung in die 3D-Computergrafik mit OpenGL

Bei einer praxisnahen Einführung in die 3D-Computergrafik stellt sich zunächst einmal die Frage, auf der Basis welcher Grafik-Programmierschnittstelle (*Application Programmers Interface* (API)) dies geschehen soll. Auf der einen Seite würden sich hier international normierte Standards anbieten, wie GKS (*Graphical Kernel System*) [Iso85] bzw. GKS-3D [Iso88] oder PHIGS (*Programmer's Hierarchical Interactive Graphics System*) [Iso89] bzw. PHIGS+ [Iso91], die in vielen Standard-Lehrbüchern über Computergrafik (siehe [Fole96],[Enca96],[Jans96]) benützt werden. Allerdings konnten sich diese Grafikstandards in der Praxis nie durchsetzen, da sie viel zu allgemein bzw. schwerfällig sind und die Möglichkeiten der Beschleunigung durch Grafikhardware nicht oder nur unzureichend ausnutzen. Auf der anderen Seite stehen de facto Standards, die die Industrie gesetzt hat: Direct3D von der Firma Mircosoft und OpenGL von der Firma Silicon Graphics bzw. einem Firmenkonsortium, das sich in einem *Architecture Review Board* (ARB) zusammengeschlossen hat.

In der folgenden Tabelle wird die historische Entwicklung der wichtigsten Grafikprogrammierschnittstellen dargestellt:

Jahr	Internationale Normen	Industrie-Standards	
1982		IrisGL (Silicon Graphics)	
1985	GKS		
1988	GKS-3D		
1989	PHIGS		
1992		OpenGL 1.0 (ARB)	
1995		OpenGL 1.1 (ARB)	Direct3D 1.0 (Microsoft)
1998		OpenGL 1.2 (ARB)	Direct3D 4.0 (Microsoft)
2001		OpenGL 1.3 (ARB)	Direct3D 7.0 (Microsoft)
2003		OpenGL 1.5 (ARB)	Direct3D 9.0 (Microsoft)
2004		OpenGL 2.0 (ARB)	Direct3D 9.0b (Microsoft)
2006		OpenGL 2.1 (ARB)	Direct3D 10 (Microsoft)

Dieses Buch führt anhand von OpenGL in die 3D-Computergrafik ein, da sich OpenGL als der *offene* Grafikstandard durchgesetzt hat. Im Folgenden sind die Gründe dafür zusammengestellt:

- OpenGL ist das 3D-Grafik-API mit der größten Marktdurchdringung.
- OpenGL ist unter allen wesentlichen Betriebssystemen verfügbar, z.B. für Mircosoft Windows95/98/2000/NT/XP, für viele UNIX-Derivate (Linux, IRIX von SGI, Solaris von SUN, HP-UX und Ultrix von HP, OS/2 und AIX von IBM usw.), für Mac OS, OPENStep und BeOS. Dieser Punkt ist eine drastische Einschränkung bei Direct3D von Microsoft, da es nur unter Windows läuft.
- OpenGL besitzt Sprachanbindungen an alle wesentlichen Programmiersprachen, wie C, C++, Java, Ada und Fortran. Beispiele in diesem Buch halten sich an die C-Syntax.
- OpenGL ist unabhängig von der Hardware, da für nahezu jede Grafikhardware entsprechende OpenGL-Treiber zu Verfügung stehen. OpenGL-Treiber müssen Konformitätstests bestehen, die eine einheitliche Funktionalität sicherstellen. Aus diesem Grund sind OpenGL-Programme leicht auf andere Hardware portierbar.
- OpenGL ist sehr gut skalierbar, d.h. es läuft auf den schwächsten PCs, auf Workstations und ebenso auf Grafik-Supercomputern. Eine Erweiterung nach unten hin wurde mit „OpenGL ES“ (Embedded Systems) geschaffen, in dem eine Teilmenge wesentlicher OpenGL-Basisfunktionen ausgeklammert wurde, die 3D-Grafikanwendungen auch auf portablen, leistungsschwachen elektronischen Geräten, wie z.B. Handys, Palm Tops oder PocketPCs ermöglicht. Eine Erweiterung nach oben hin wurde mit „OpenGL SL“ (Shading Language) geschaffen, einer Vertex- und Pixel-Shader-Sprache für anspruchsvolle Oberflächeneffekte, die als OpenGL Extension auf den neuesten Grafikkarten nutzbar ist.
- Die Erweiterung OpenML (Media Library) bietet Unterstützung für die Erfassung, Übertragung, Verarbeitung, Darstellung und die Synchronisierung digitaler Medien, wie z.B. Audio und Video.
- OpenGL hat den mächtigsten 3D-Graphik-Sprachumfang.
- OpenGL bietet Hardware-Herstellern zur Differenzierung ihrer Produkte die Möglichkeit, hardware-spezifische Erweiterungen des Sprachumfangs, die sogenannten OpenGL *Extensions* zu entwickeln und somit innovative Hardware-Funktionen optimal zu nutzen. Viele frühere OpenGL *Extensions* sind in den Standard-Sprachumfang neuerer OpenGL-Versionen eingeflossen.
- OpenGL wird kontrolliert und ständig weiterentwickelt durch ein *Architecture Review Board* (ARB), in dem die bedeutendsten Firmen der Computergrafik-Industrie

vertreten sind (siehe nachfolgende Tabelle). Das ARB hat bis auf weiteres beschlossen, jedes Jahr eine neue Version von OpenGL zu spezifizieren, um dem rasanten Fortschritt in der Computergrafik Rechnung zu tragen.

In der folgenden Tabelle sind die derzeitigen[1] Mitglieder des OpenGL ARB aufgelistet:

Architecture Review Board (ARB)		
Gründungsmitglieder	später aufgenommene Mitglieder	
Silicon Graphics / SGI	Evans&Sutherland	Apple
(Microsoft)	nVidia	SUN Microsystems
IBM	ATI	Dell
Intel	Matrox	3DLabs
	Quantum3D	Spinor GmbH

Eine Implementierung von OpenGL ist meistens ein Treiber für eine Hardware (z.B. eine Grafikkarte), der für das Rendering, d.h. für die Umsetzung von OpenGL Funktionsaufrufen in Bilder, sorgt. In selteneren Fällen ist eine OpenGL-Implementierung auch eine Software Bibliothek, die das Gleiche sehr viel langsamer bewerkstelligt. Abgesehen von den Mitgliedsfirmen des ARB (siehe oben), gibt es noch eine große Zahl weiterer Firmen, die Lizenzen für OpenGL von SGI erwerben, um für Ihre eigenen (Hardware-)Produkte eine OpenGL-Implementierung zu entwickeln und zu vermarkten.

Allerdings muss an dieser Stelle nochmal darauf hingewiesen werden, dass die vorliegende Abhandlung kein OpenGL-Handbuch ist, in dem alle Funktionalitäten von OpenGL erläutert werden, dies ist den Referenzhandbüchern ([Shre04],[Shre05],[Leng03],[Sega04]) vorbehalten. Diese Abhandlung führt zwar anhand von OpenGL in die wichtigsten Grundlagen der interaktiven 3D-Computergrafik ein, geht aber mit Themen wie Vertex- und Pixel-Shader, Aufbau eines Szenengraphen, Cull-Algorithmen oder Mehrprozessorsysteme darüber hinaus (Kapitel 12, 13, 15).

5.1 OpenGL Kurzbeschreibung

Im Folgenden wird ein stichpunktartiger Überblick über OpenGL gegeben:

- OpenGL steht für *Open Graphics Library*
- OpenGL ist eine Grafikprogrammierschnittstelle (Grafik-API), die:
  - unabhängig ist von der Grafik-Hardware.

[1]Microsoft hat 2003 das ARB verlassen, unterstützt OpenGL aber weiterhin und liefert mit jedem Betriebssystem auch OpenGL-Treiber aus

  - unabhängig ist von der Bildschirmfenster-Verwaltung und der Benutzerschnittstelle (z.B. Maus und Tastatur).
  - einen Sprachumfang von ca. 200 Befehlen besitzt.

- OpenGL besteht aus drei Teilen:
  - der OpenGL *Library* (opengl.dll unter Unix bzw. opengl32.dll unter Windows), die die Implementierung der OpenGL-Basisbefehle enthält und mit der Zeile „`#include <GL/gl.h>`“ in ein C/C++ – Programm eingebunden wird.
  - der OpenGL *Utility Library* (glu.dll unter Unix bzw. glu32.dll unter Windows), die eine Reihe von Befehlen enthält, um komplexere geometrische Objekte wie z.B. Ellipsoide, Splines oder NURBS (*non-uniform rational b-splines*) zu rendern, um konkave Polygone in konvexe aufzuteilen (Kapitel 6) und um Geometrie-Transformationen mit Hilfe von Matrizenoperationen durchzuführen (Kapitel 7). Mit der Zeile „`#include <GL/glu.h>`“ wird die GLU-Library in ein C/C++ – Programm eingebunden.
  - einer Erweiterung für die Window-Verwaltung:
    * GLX für das X-Window-System
    * WGL für Microsoft Windows
    * AGL für Apple Mac OS
    * PGL für OS/2 von IBM
    * GLUT (OpenGL *Utility Toolkit*, glut.dll unter Unix bzw. glut32.dll unter Windows), einer vom *Window*-System unabhängigen Bibliothek, auf die bei den hier vorgestellten Beispielen zurückgegriffen wird. GLUT stellt Befehle bereit, um ein Bildschirmfenster (*window*) zu öffnen, um interaktive Benutzereingaben z.B. von der Tastatur zu bearbeiten, um insbesondere bei bewegten Bildern die veränderten Bildinhalte berechnen zu lassen (mit „*Display Callback*“ – Funktionen) und um komplexere geometrische Objekte wie z.B. Kegel, Kugel, Torus oder eine Teekanne zeichnen zu lassen. Eine genauere Beschreibung der GLUT-Befehle wird in [Shre04] und [Kilg96] bzw. im Internet[2] gegeben. Mit der Zeile „#include <GL/glut.h>“ wird die GLUT-Library in ein C/C++ – Programm eingebunden.

- Die Grundstruktur eines OpenGL-Programms ist einfach, sie besteht nur aus zwei Teilen:
  - Der Initialisierung eines Zustandes, um festzulegen, in welchem Modus die Objekte gerendert werden sollen.
    Wie wird gerendert.
  - Der Festlegung der 3D-Geometrie der Objekte, die gerendert werden sollen.
    Was wird gerendert.

[2] Auf der Interseite zu diesem Buch findet man entsprechende Links

- OpenGL ist ein Zustandsautomat (*state machine*):
  Der Automat befindet sich immer in einem bestimmten Zustand. Dies gilt auch, wenn nichts festgelegt wurde, denn jede Zustandsvariable hat einen sogenannten „Default"–Wert, der standardmäßig angenommen wird. Wird für eine Zustandsvariable, wie z.B. die Farbe, ein bestimmter Wert festgelegt, so werden alle folgenden Objekte mit dieser Farbe gerendert, bis im Programm eine neue Farbe spezifiziert wird. Andere Zustandsvariablen, die festlegen, wie bestimmte geometrische Objekte gerendert werden sollen, wären z.B. Materialeigenschaften von Oberflächen, Eigenschaften von Lichtquellen, Nebeleigenschaften oder Kantenglättung (*Anti-Aliasing*). Die Zustandsvariablen legen also einen Zeichenmodus fest, der mit den folgenden OpenGL-Kommandos ein- bzw. ausgeschaltet werden kann:

  `glEnable(...)` : Einschalten
  `glDisable(...)` : Ausschalten

5.2 Die OpenGL Rendering Pipeline

Zur Generierung synthetischer Bilder auf einem Computer sind viele einzelne Rechenschritte in einer bestimmten Reihenfolge notwendig, die nach dem Fließbandprinzip in Form einer sogenannten „*Rendering Pipeline*" ausgeführt werden. Diese Reihenfolge der Operationen, wie sie in Bild 5.1 in der Art eines Blockschaltbildes dargestellt ist, vermittelt einen groben Überblick über die prinzipiellen Abläufe bei der 3D-Computergrafik, der nicht nur für OpenGL, sondern generell gilt. In diesem Abschnitt werden die einzelnen Blöcke der *Rendering Pipeline* kurz erläutert, um ein erstes Gefühl für die Zusammenhänge zu vermitteln. In den weiteren Kapiteln wird die Funktionsweise der einzelnen Blöcke genau erklärt.

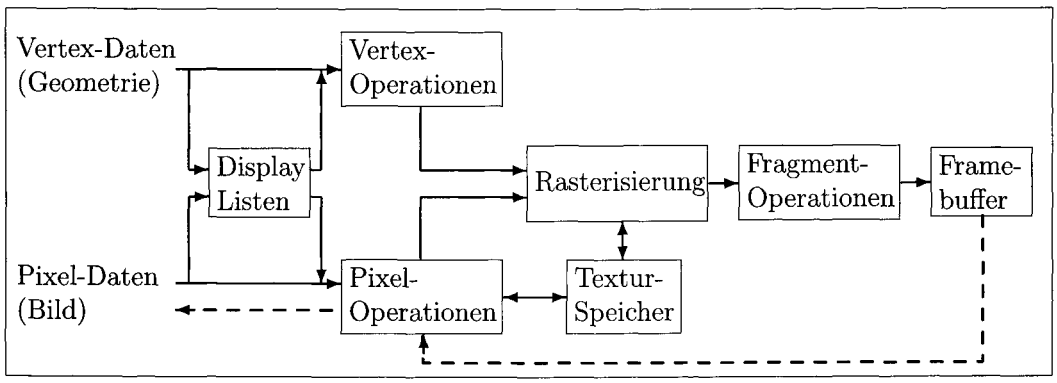


Bild 5.1: Die *Rendering Pipeline* beschreibt die Reihenfolge der Operationen bei der Generierung synthetischer Bilder

Zunächst gibt es zwei verschiedene Daten-Pfade, denn einerseits werden 3D Geometrie-Daten (Vertices, Linien, Polygone) bearbeitet (transformiert, beleuchtet, zugeschnitten) und andererseits können auch Bild-Daten verarbeitet (gefiltert, gezoomt, gruppiert) und in einem Textur-Speicher abgelegt werden. Beide Datenarten werden dann rasterisiert, d.h. in Bildpunkte (sog. Fragmente) umgewandelt und im Rahmen der Fragment-Operationen zusammengeführt, bevor sie schließlich in den Bildspeicher (engl. Framebuffer) geschrieben werden.

5.2.1 Display Listen

Display Listen dienen dazu, geometrische Primitive (Vertices, Linien, Polygone) oder Bild-Daten zu komplexeren Objekten zusammenzufassen und diese komplexen Objekte auch mehrfach wiederverwenden zu können. Dies geschieht einfach dadurch, dass alle benötigten Daten, egal ob Geometrie- oder Bild-Daten, in einer sogenannten *Display* Liste unter einem bestimmten Namen gespeichert werden und entweder sofort oder zu einem späteren Zeitpunkt wieder abgerufen werden können. Der Zeichenmodus, bei dem komplexere Objekte abgespeichert und später wieder aufgerufen werden, heißt *Display List Modus* oder auch *Retained Mode*. Werden dagegen alle Daten unmittelbar an die *Rendering Pipeline* übergeben, spricht man vom *Immediate Mode*.

5.2.2 Vertex-Operationen

Vertex-Daten sind einfach nur Koordinaten von Punkten im 3-dimensionalen Raum. Zusammen mit der Information, welche Punkte miteinander verbunden sind, können so Linien oder Polygone definiert werden. Vertices sind also die Eckpunkte von Polygonen und Linien, oder einfach nur 3D-Punkte. Die Vertex-Daten müssen zunächst über mehrere Stufen transformiert werden: durch Modelltransformationen werden die Objekte richtig positioniert (glTranslate), gedreht (glRotate) und skaliert (glScale); anschließend wird eine Projektionstransformation durchgeführt, um die 3D-Koordinaten auf eine 2D-Bildebene zu projizieren, danach erfolgt eine Normierung und die Abbildung auf 2D-Bildschirmkoordinaten (Viewport Transformation). Außerdem werden in dieser Stufe gegebenenfalls die Normalenvektoren und die Texturkoordinaten generiert und transformiert. Falls der Beleuchtungsmodus aktiviert ist, wird in dieser Verarbeitungsstufe aus den Vertices, Normalenvektoren und Materialeigenschaften der Objekte auf der einen Seite und den Eigenschaften der Lichtquellen auf der anderen Seite für jeden Vertex die Beleuchtungsrechnung durchgeführt und somit eine Farbe ermittelt. Sollten zusätzliche Schnittebenen (*Clipping Planes*) definiert sein, werden die Geometrie-Daten eliminiert, die sich im weggeschnittenen Halbraum befinden.

5.2.3 Pixel-Operationen

Pixel-Daten sind die Farbwerte der einzelnen Bildpunkte von 2D-Bildern, auch Texturen genannt (möglich sind auch 1-dimensionale oder 3-dimensionale Texturen). Texturen

können z.B. Fotografien sein, die auf Polygone quasi aufgeklebt werden. Damit lassen sich sehr realistisch aussehende Computergrafiken erzeugen. Der erste Bearbeitungsschritt bei Pixel-Daten ist das Entpacken, falls die Daten komprimiert vorliegen. Anschließend können die Farbwerte skaliert, begrenzt, mit einem Offset versehen oder mit Hilfe einer Lookup-Tabelle transformiert werden. Zuletzt können die Texturen gruppiert und, falls hardwareseitig vorhanden, in einem sehr schnellen Textur-Speicher abgelegt werden.

5.2.4 Textur-Speicher

Texturen müssen erst einmal vom Hauptspeicher des Computers über ein Bussystem auf die Grafikhardware geladen werden. Sie durchlaufen dann, wie im vorigen Absatz erläutert, eine Reihe von Operationen, bevor sie auf die Polygone projiziert werden können. Wegen der häufigen Wiederverwendung von Texturen bietet es sich deshalb an, sie nach der Vorverarbeitung in einem Textur-Speicher im Grafiksubsystem abzuspeichern, auf den sehr schnell zugegriffen werden kann. Falls in einer Anwendung mehr Texturen benötigt werden, als in den Textur-Speicher passen, müssen möglichst in Perioden geringerer Belastung der Grafikhardware Texturen vom Hauptspeicher in den Textur-Speicher nachgeladen werden. Bei Echtzeit-Anwendungen erfordert dies einen vorausschauenden, prioritätsgesteuerten Textur-Nachlademechanismus.

5.2.5 Rasterisierung

Nach den Vertex-Operationen liegen für jeden Eckpunkt eines Polygons die korrekt berechneten Farbwerte vor. Was jetzt noch für die Darstellung am Bildschirm fehlt, sind die Farbwerte für jeden Leuchtpunkt der Bildschirm-Maske, den das Polygon bedeckt. Diese Interpolation der Farbwerte zwischen den Vertices eines Polygons für jeden Rasterpunkt des Bildschirms heißt Rasterisierung. In dieser Stufe der Rendering Pipeline werden zur besseren Unterscheidbarkeit die Rasterpunkte auch Fragmente genannt, d.h. jedes Fragment entspricht einem Pixel im Framebuffer und dieses entspricht einem Raster- oder Leuchtpunkt des Bildschirms. Texturen liegen nach den Pixel-Operationen ebenfalls in Form von Farbwerten für die einzelnen Texturbildpunkte, auch Texel (***tex**ture **el**ement*) genannt, vor. Da Texturen meist auf Polygone projiziert werden, stimmt die Größe der Textur praktisch nie mit der Größe des Polygons am Bildschirm überein. Deshalb muss auch für Texturen eine Rasterisierung auf die Fragmente durchgeführt werden.

5.2.6 Fragment-Operationen

Nach der Rasterisierung liegen für jedes Fragment Farbwerte sowohl von den Vertex-Daten (Polygonen etc.) als auch von den Pixel-Daten (Texturen) vor. In der ersten Stufe der Fragment-Operationen werden jetzt abhängig vom eingestellten Textur Mapping Modus, für jedes Fragment die Farbwerte der Vertex-Daten ersetzt durch oder gemischt mit den Farbwerten der Texturen. In den weiteren Stufen der Fragment-Operationen werden die Nebelberechnungen, das *Anti-Aliasing* (Kantenglättung), der Sichtbarkeitstest bzgl. des

Bildschirms bzw. *Viewports* (Fenster), der *Alpha*-Test bzgl. transparenter Bildpunkte, der *Stencil*-Test für Maskierungseffekte, der *Z-Buffer*-Test (Verdeckungsrechnung), das Mischen (*blending*) der Farbe mit einer Hintergrundfarbe, das *Dithering* (die Erhöhung der Farbauflösung auf Kosten der räumlichen Auflösung), sowie logische Operationen und das abschließende maskierte Schreiben in den Bildspeicher (*Framebuffer*) durchgeführt.

5.2.7 Bildspeicher

Der Bildspeicher (*Framebuffer*) enthält in digitaler Form die endgültigen Farbwerte für jeden Bildpunkt (Pixel von ***pic**ture **el**ement*) des computergenerierten Bildes. Das fertige Bild im *Framebuffer* kann jetzt für verschiedene Zwecke benutzt werden:

- Umwandlung des Bildes mit Hilfe eines Digtal-Analog-Converters (DAC) in ein elektrisches Signal, um das Bild z.B. auf einem Bildschirm ausgeben zu können
- Zurückkopieren des Bildes in den Hauptspeicher und abspeichern auf einem Datenträger (*Offscreen-Rendering*)
- Zurückkopieren des Bildes in die Rendering Pipeline und weitere Benutzung als Textur (*Multipass Rendering*)

5.3 Die OpenGL Kommando Syntax

In OpenGL gelten bestimmte Konventionen für die Schreibweise von Kommandos und Konstanten. Außerdem wurden zur Sicherung der Plattformunabhängigkeit eigene Datentypen definiert.
Kommandos enthalten immer ein Prefix, an dem man erkennen kann, aus welcher Bibliothek der Befehl stammt, gefolgt von Wörtern, die jeweils mit Großbuchstaben beginnen:

`gl*`	: Basis-Befehl der OpenGL Library (opengl32.dll)	z.B.: `glShadeModel()`
`glu*`	: Befehl der OpenGL Utility Library (glu32.dll)	z.B.: `gluNurbsSurface()`
`glut*`	Befehl des OpenGL Utility Toolkit (glut32.dll)	z.B.: `glutCreateWindow()`

Manche Kommandos gibt es in verschiedenen Ausprägungen, die durch weitere Zeichen nach dem Kommandonamen festgelegt werden, wie in Bild 5.2 dargestellt.

Konstanten in OpenGL beginnen immer mit dem Prefix „`GL_*`“, die Buchstaben aller Wörter werden groß geschrieben und mehrere Wörter in einer Konstante werden durch „_“ (Unterstrich, *underscore*) getrennt, wie bei dem Beispiel `GL_COLOR_BUFFER_BIT`.

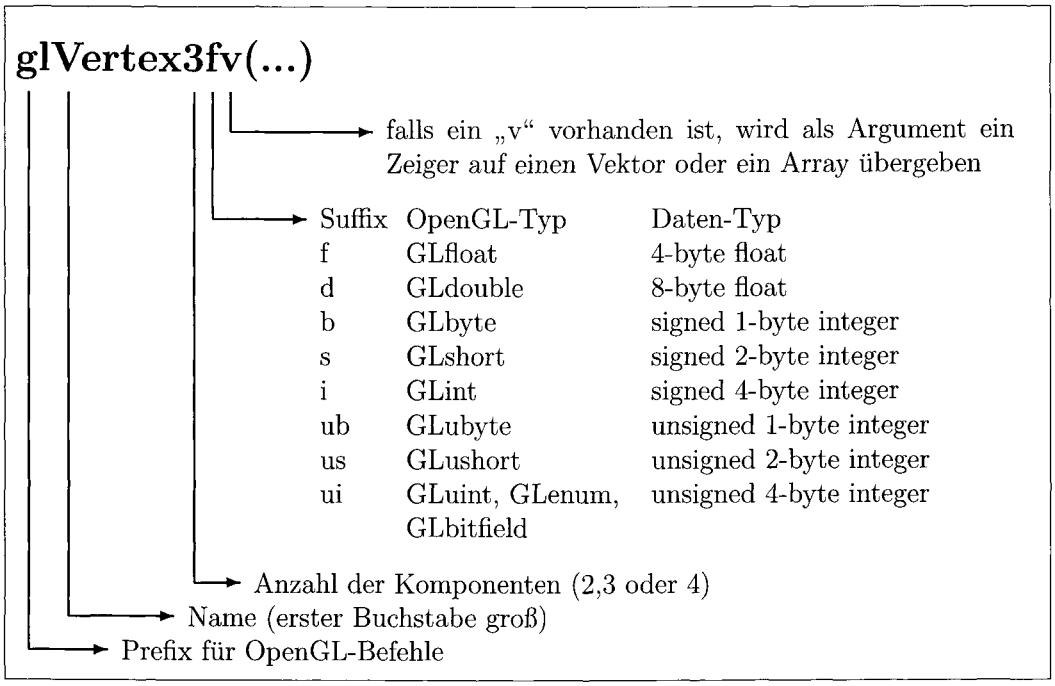


Bild 5.2: Die OpenGL Kommando Syntax

5.4 Ergänzende Literaturhinweise

Im Anhang findet man das gesamte Literaturverzeichnis, getrennt nach den Gebieten Computergrafik und Bildverarbeitung. An dieser Stelle wird nur eine kleine Auswahl kommentierter Literaturempfehlungen zur Computergrafik als Ergänzung zu diesem Werk vorgestellt.

Computergrafik allgemein:

- [Fole96] Foley J.D., van Dam A., Feiner S.K., Hughes J.F.: *Computer graphics: principles and practice. 2nd ed. in C.* Addison-Wesley, Reading, 1996.
  Das Standardwerk zur Computergrafik: allumfassend, tiefgehend, nicht mehr ganz aktuell, aber dennoch sehr gut, dick und teuer.

- [Watt02] Watt A.: *3D-Computergrafik. 3. Auflage.* Pearson Studium, München, 2002.
  Sehr gutes Buch: grundlagenorientiert, mathematisch, aktuell.

- [Aken02] Akenine-Möller T., Haines E.: *Real-Time Rendering, Second Edition.* A.K. Peters, Natick, 2002.

Das Buch für fortgeschrittene Grafikprogammierer: deckt die neuesten Entwicklungen der Computergrafik vollständig ab, bespricht Vor- und Nachteile aller Verfahren.

Offizielle Standardwerke zu OpenGL:

- [Shre05] Shreiner D., Woo M., Neider J., Davis T., OpenGL ARB: *OpenGL Programming Guide, 5th edition: The Official Guide to Learning OpenGL, Version 2.0.* Addison-Wesley, Reading, 2005.

  Das „rote" Buch: sehr schöne Einführung in die Bedienung von OpenGL, zeigt zum Teil auch Hintergründe auf und ist deshalb sehr gut verständlich.

- [Shre04] Shreiner D., OpenGL ARB: *OpenGL Reference Manual, 4th edition.* Addison-Wesley, Reading, 2004.

  Das „blaue" Buch: ein reines Referenzhandbuch, in dem die OpenGL-Befehle incl. Parametern und Konstanten aufgelistet und kurz erklärt sind, nicht mehr ganz aktuell.

- [Rost04] Rost R.J.: *OpenGL Shading Language.* Addison-Wesley, Reading, 2004

  Das „ockergelbe" Buch: sehr schöne Einführung in die OpenGL Shading Language, mit der man programmierbare Grafikhardware ansteuern kann.

- [Leng03] Lengyel E.: *The OpenGL Extensions Guide.* Charles River Media, Hingham, 2003.

  Das „grüne" Buch: ein reines Referenzhandbuch, in dem die OpenGL-Extensions incl. Parametern und Konstanten aufgelistet und kurz erklärt sind.

- [Sega04] Segal M., Akeley K.: *The OpenGL Graphics System: A Specification, Version 2.0.* Silicon Graphics Inc., Mountain View, 2004.

  Die offizielle Referenz für alle Entwickler von OpenGL-Treibern. Die kostenlose Alternative zum „blauen" und „grünen" Buch.

Im Internet sind sehr viele Informationen und Programmbeispiele zu OpenGL zu finden. Allerdings sind die Änderungszyklen im Internet sehr schnell, so dass gedruckte Links häufig in kürzester Zeit veraltet sind. Deshalb findet man eine aktuelle Linkliste auf der Webseite zu diesem Buch:

http://www.cs.fhm.edu/cgbv-buch

Der Zugang zu dieser Webseite ist passwort-geschützt. Den Benutzernamen und das Passwort erhält man, wenn man auf dieser Webseite dem Link „Passwort" folgt. In Kapitel 1 ist beschrieben, welche Bonusmaterialien zu diesem Buch auf der Webseite zur Verfügung gestellt werden.

Kapitel 6

Geometrische Grundobjekte

Die 3D–Computergrafik hat konstruktiven Charakter: als Ausgangspunkt des Rendering muss zunächst eine abstrakte Beschreibung von 3–dimensionalen Objekten erstellt werden, bevor eine 2-dimensionale Projektion der gesamten Szene aus einem bestimmten Blickwinkel für einen flachen Bildschirm berechnet werden kann. Für die Modellierung von 3–dimensionalen Objekten wurden in der 3D–Computergrafik, je nach Anwendungsfall, unterschiedliche Methoden entwickelt, die im Folgenden erläutert werden.

6.1 3D-Modellierungsmethoden

6.1.1 Planare Polygone

Da die allermeisten 3D–Objekte in unserer natürlichen Umgebung nicht transparent, d.h. opak[1] sind, genügt es für die Bildgenerierung, nur die Oberflächen der 3D–Objekte zu modellieren. Da sich Lichtstrahlen in transparenten Medien, in denen der Brechungsindex im gesamten Medium konstant ist, wie z.B. in Luft oder Glasscheiben, geradlinig ausbreiten, genügt es in diesem Fall, nur die Oberflächen zu modellieren, denn die Brechung des Lichts einer bestimmten Wellenlänge wird durch die Grenzfläche zwischen den Medien und deren Brechungsindizes bestimmt (Kapitel 9). In der interaktiven 3D–Computergrafik bei weitem am häufigsten eingesetzt wird die Annäherung von 3D–Objekten durch Netze aus ebenen, d.h. planaren Polygonen, insbesondere durch Netze aus Dreiecken (*triangles*) und Vierecken (*quads*). Die Einschränkung auf planare Polygone erleichtert die Rasterisierung und die Verdeckungsrechnung wesentlich, da nur noch lineare Gleichungen vorkommen, die extrem schnell gelöst werden können und sich gut für die Hardware-Beschleunigung eignen. Außerdem kann man alle denkbaren Formen von 3D–Objekten durch Netze aus planaren Polygonen mit beliebiger Genauigkeit darstellen (Bild 6.1). Auf dieser einfachen und schnellen Methode beruht die Modellierung von 3D–Objekten bei OpenGL. Aber dennoch muss man einen Preis für die Einschränkung auf planare Polygone bezahlen: bei gekrümmten

[1] Der Begriff „opak“ bedeutet „undurchsichtig“ oder „lichtundurchlässig“.

Oberflächen benötigt man für eine bestimmte Genauigkeit eine höhere Polygonauflösung als mit den in Abschnitt 6.1.2 erwähnten gekrümmten Oberflächenelementen.

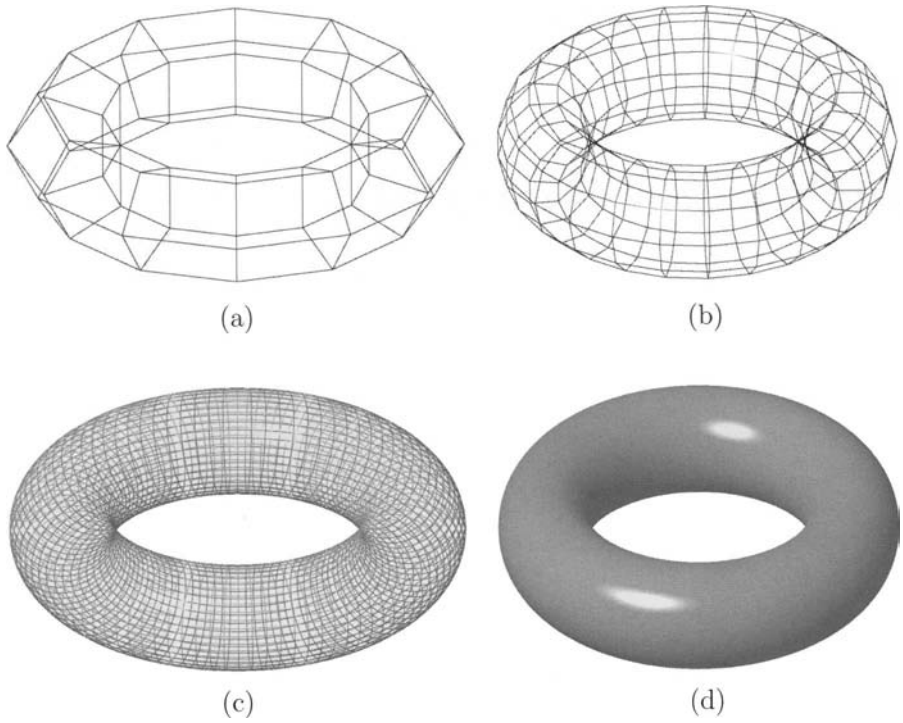


Bild 6.1: Tori in verschiedenen Polygonauflösungsstufen: (a) 50 Quads (b) 200 Quads (c) 3200 Quads (d) 819200 Quads

6.1.2 Gekrümmte Polygone

Der einzige Unterschied zu dem Ansatz mit planaren Polygonen ist, dass jetzt als grafische Grundbausteine auch räumlich gekrümmte Polygone, sogenannte bikubische parametrische Patches (z.B. Bezier–Flächen oder NURBS (*Non-Uniform Rational B-Splines*)) zugelassen sind. Dieser Ansatz bietet den Vorteil, dass komplexe 3–dimensional gekrümmte Oberflächen durch wenige Parameter (sogenannte Kontrollpunkte) *exakt* modelliert werden können. Die Speicherung einiger Kontrollpunkte nimmt sehr viel weniger Platz ein, als hunderte oder tausende planarer Polygone. Außerdem könnten bei sehr hohen Genauigkeitsanforderungen selbst tausend planare Polygone zu wenig sein, die Kontollpunkte dagegen beschreiben die gekrümmte Oberfläche exakt. Allerdings bringt dieser Ansatz auch Nachteile mit sich: erstens sind bikubische parametrische Patches sehr rechenaufwändig

und daher echtzeitschädlich, und zweitens ist es problematisch, die Form eines Patches in einem Netzwerk zu verändern, da dann in der Regel die Tangenten zwischen benachbarten Patches nicht mehr übereinstimmen, d.h. es entstehen oft unerwünschte Kanten an den Grenzen. Eine ausführlichere Darstellung der Mathematik von parametrischen Kurven und Flächen wird in den allgemeinen Computergrafikbüchern [Fole96] und [Watt02] gegeben. Diese Art der 3D–Modellierung wird in einigen Bereichen des *Computer Aided Design* (CAD) angewendet, da sie eine elegante Art der Gestaltung von Freiformflächen erlaubt. OpenGL bietet im Rahmen der GLU-Bibliothek (OpenGL *Utility Library*) Unterstützung für diese Art der 3D–Modellierung ([Shre05]). Allerdings werden die bikubischen parametrischen Patches in OpenGL vor dem eigentlichen Rendering durch sogenannte Evaluatoren in Abhängigkeit von den eingestellten Genauigkeitsanforderungen in eine mehr oder weniger große Anzahl von planaren Polygonen umgerechnet.

6.1.3 Volumendarstellung

Problematisch werden die polygonalen Ansätze erst in Fällen, in denen die Transparenz und/oder der Brechungsindex im Medium schwanken, wie z.B. bei Wolken und Bodennebel oder wenn die Anwendung einen Einblick in den inneren Aufbau des 3D–Objekts ermöglichen soll, wie z.B. in der Medizin bei nichtinvasiven Analysemethoden (Computertomogramm, Kernspintomogramm etc.) oder der Darstellung von Temperatur- oder Druckverläufen innerhalb von Werkstücken. In solchen Fällen eignet sich die Volumendarstellung besonders gut, deren Grundidee die Unterteilung des Raumes in elementare Volumenelemente, sogenannte „*Voxel*“ (***vo**lume **el**ement*) ist - in Analogie zur Darstellung von 2-dimensionalen Bildern durch ein Raster von Pixeln. Ebenso wie man den Pixeln eines Bildes z.B. Farbwerte zuordnet, werden auch den Voxeln eines Raumes bestimmte Farbwerte zugeordnet, die z.B. die Temperatur, den Druck oder die Dichte des Mediums repräsentieren. Da in dieser Darstellungsart für jeden Raumpunkt ein Farbwert gegeben ist, kann ein Betrachter beliebige Schnittebenen (*clipping planes*) oder allgemeine Schnittflächen durch das Volumen legen und sich somit durch eine Reihe von 2-dimensionalen Bildern einen Eindruck von den inneren Eigenschaften des 3–dimensionalen Objektes machen. Der Nachteil dieses Ansatzes ist, dass bei einer den polygonalen Ansätzen vergleichbaren Auflösung eine ungleich höhere Datenmenge für die Beschreibung der Objekte anfällt. Während bei den polygonalen Ansätzen nur die relativ spärlich im Raum verteilten Eckpunkte der Polygone (die Vertices) und allenfalls noch ein paar Kontrollpunkte für gekrümmte Oberflächen nötig sind, müssen beim Volumenansatz alle Rasterpunkte im Raum mit Werten belegt sein. Auf der anderen Seite sind es aber gerade die räumlich dicht liegenden Informationen, die bei bestimmten Anwendungen wertvoll sind. OpenGL unterstützt seit 1998 (d.h. seit der Version 1.2) auch diese Art der 3D–Modellierung durch die Möglichkeit der Definition und Darstellung von 3–dimensionalen Texturen. Darauf aufbauend gibt es u.a. von der Firma Silicon Graphics Inc. ein *high-level* Werkzeug zum *Volume Rendering*, den OpenGL Volumizer ([Jone05]).

6.1.4 Konstruktive Körpergeometrie

Konstruktive Körpergeometrie (*Constructive Solid Geometry* (CSG)) ist ebenfalls eine volumetrische Methode, denn komplexere 3D–Objekte werden aus elementaren Körpern („Bauklötzen"), wie z.B. Quadern, Kugeln, Kegeln oder Zylindern zusammengesetzt. Zusammensetzen bedeutet hier aber nicht nur das Neben- oder Aufeinandersetzen undurchdringbarer „Bauklötze", sondern die Kombination der Grundkörper durch Boole'sche Mengen-Operatoren oder lineare Transformationen. Die Grundidee für die CSG-Methode kommt aus der Fertigung: man nehme einen Rohling (z.B. einen Quader) und bohre ein Loch (zylinderförmig) hinein, oder in der CSG-Methode ausgedrückt, vom Grundkörper Quader wird der Grundkörper Zylinder subtrahiert. Durch weitere Bearbeitungsvorgänge, wie z.B. Fräsen, Drehen, Schneiden oder auch Zusammenfügen entsteht schließlich ein komplexes 3–dimensionales Endprodukt. Bei der CSG-Methode werden alle Grundkörper und deren Kombinationen in einer Baumstruktur gespeichert: die Grundkörper sind die Blätter des Baums und die Kombinationen sind die Knoten. Auf diese Art und Weise wird nicht nur das Endprodukt gespeichert, sondern auch alle Herstellungsschritte. Aus diesem Grund eignet sich die CSG-Methode besonders gut für *Computer Aided Manufactoring* (CAM)-Anwendungen, wie z.B. computergesteuerte Fräs- und Bestückungsautomaten. Ein großer Nachteil des CSG-Ansatzes ist es, dass es sehr rechenaufwändig ist, ein gerendertes Bild eines 3D–Objektes zu bekommen. Meistens wird der Weg über die Annäherung der Oberfläche des CSG-Modells durch Netze von planaren Polygonen gewählt, um schnell zu einer qualitativ hochwertigen Darstellung des CSG-Modells zu kommen. Damit sind wir jedoch wieder einmal an den Ausgangspunkt der Betrachtungen zurückgekehrt.

6.2 Geometrische Grundobjekte in OpenGL

In OpenGL werden alle geometrischen Objekte durch einen Satz von *Grafik-Primitiven*, bestehend aus Punkten, Linien und planaren Polygonen beschrieben. Komplexere 3-dimensionale Objekte, wie z.B. die gekrümmte Oberfläche eines Torus' oder eine Schraubenlinie werden durch eine für die Genauigkeitsanforderungen ausreichende Anzahl von planaren Polygonen bzw. geraden Linienstücken angenähert (Bild 6.1). Die Polygone und Linienstücke ihrerseits sind definiert durch einen geordneten Satz von Vertices, d.h. Punkten im 3–dimensionalen Raum, die die Eckpunkte der Polygone oder die Endpunkte der Linien darstellen, sowie durch die Information, welche Vertices miteinander verbunden sind, oder anders ausgedrückt, um welchen Typ von Grafik-Primitiv es sich handelt. Im konstruktiven Sinne kann man also mehrere Abstraktionsebenen bei der Nachbildung von 3–dimensionalen Objekten durch planare Polygone unterscheiden (Bild 6.2):

- Es wird ein geordneter Satz von Vertices definiert.
- Durch die Angabe des Grafik-Primitiv-Typs für einen Block von Vertices wird festgelegt, welche Vertices miteinander zu verbinden sind. Vertices können gleichzeitig zu verschiedenen Grafik-Primitiven gehören. Ein solcher Block legt also eine facettierte

Oberfläche einer bestimmten Form fest (z.B. einer Kreisscheibe als Polygon, eines Kegelmantels als Fächer aus Dreiecken (*Triangle_Fan*) oder eines Zylindermantels aus verbundenen Vierecken (*Quad_Strip*)), falls es sich um polygonale Grafik-Primitive handelt.

- Die facettierte Oberfläche eines solchen Blocks ist eine Annäherung an eine mathematisch ideale Oberfläche (z.B. glatter Kegel- oder Zylindermantel).
- Entsprechende Oberflächenteile können z.B. die geschlossene Oberfläche eines 3–dimensionalen Objektes nachbilden.

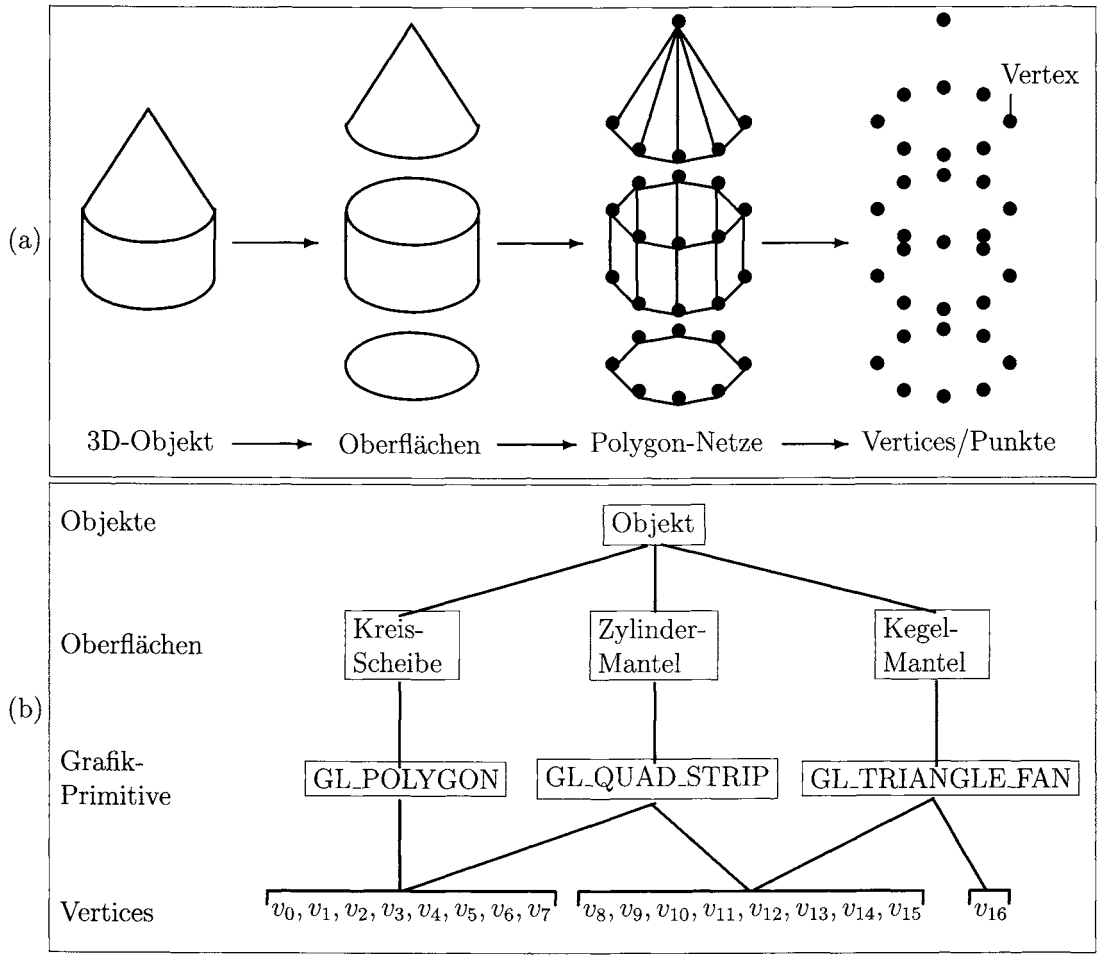


Bild 6.2: Darstellung eines 3–dimensionalen Objektes durch Oberflächen, Polygon-Netze bzw. Vertices. (a) Perspektivische Darstellung (b) hierarchische Baumstruktur

6.2.1 Vertex-Definition in OpenGL

In OpenGL werden alle geometrischen Objekte durch einen geordneten Satz von *Vertices* beschrieben. Ein Vertex ist ein Punkt im 3–dimensionalen Raum, der entweder die Eckpunkte eines Polygons, die Endpunkte von Linien oder einfach nur einen Punkt an sich darstellt. Zur Spezifikation von Vertices wird der Befehl `glVertex*(TYPE coords)` benützt, der in verschiedenen Ausprägungen existiert, wie in der folgenden Tabelle dargestellt:

Skalar-Form	Vektor-Form	z-Koordinate	w (inverser Streckungsfaktor)
`glVertex2f(x,y)`	`glVertex2fv(vec)`	0.0	1.0
`glVertex2d(x,y)`	`glVertex2dv(vec)`	0.0	1.0
`glVertex2s(x,y)`	`glVertex2sv(vec)`	0.0	1.0
`glVertex2i(x,y)`	`glVertex2iv(vec)`	0.0	1.0
`glVertex3f(x,y,z)`	`glVertex3fv(vec)`	z.d.	1.0
`glVertex3d(x,y,z)`	`glVertex3dv(vec)`	z.d.	1.0
`glVertex3s(x,y,z)`	`glVertex3sv(vec)`	z.d.	1.0
`glVertex3i(x,y,z)`	`glVertex3iv(vec)`	z.d.	1.0
`glVertex4f(x,y,z,w)`	`glVertex4fv(vec)`	z.d.	z.d.
`glVertex4d(x,y,z,w)`	`glVertex4dv(vec)`	z.d.	z.d.
`glVertex4s(x,y,z,w)`	`glVertex4sv(vec)`	z.d.	z.d.
`glVertex4i(x,y,z,w)`	`glVertex4iv(vec)`	z.d.	z.d.

z.d. = zu definieren

In der Skalar–Form des Befehls müssen die Koordinaten im entsprechenden Datenformat (z.B. `f = GLfloat`) direkt übergeben werden, wie im folgenden Beispiel gezeigt:

```
glVertex3f(3.8, 0.47, -4.1);
```

In der Vektor–Form des Befehls wird nur ein Zeiger auf ein Array übergeben, das die Koordinaten im entsprechenden Datenformat (z.B. `f = GLfloat`) enthält, wie im folgenden Beispiel gezeigt:

```
GLfloat vec[3] = {3.8, 0.47, -4.1};
glVertex3fv(vec);
```

Meistens bringt es Vorteile die Vektorform des Befehls zu benützen, da es erstens schneller geht, nur einen Wert (den Zeiger) an die Grafikhardware zu übergeben anstatt mehrerer Werte (die Koordinaten), und zweitens eine größere Flexibilität besteht, die beim Einlesen von Vertices aus Datenfiles benötigt wird.

OpenGL arbeitet mit *Homogenen Koordinaten* im 3-dimensionalen Raum (Einführung in Kapitel 7), so dass für interne Berechnungen alle Vertices durch vier Koordinaten (x, y, z, w) dargestellt werden. Solange der inverse Streckungsfaktor „w“ ungleich 0 ist, entsprechen diese vier Koordinaten im 3–dimensionalen Euklidischen Raum dem Punkt

$(x/w, y/w, z/w)$. Bei einem inversen Streckungsfaktor „$w = 0$" definieren diese vier Koordinaten keinen Punkt, sondern eine Richtung im Euklidischen Raum. Wird ein Vertex mit einem Befehl `glVertex2*()` spezifiziert, der nur die beiden (homogenen) Koordinaten x und y festlegt, wird automatisch die z–Koordinate auf 0 und die w–Koordinate auf 1 gesetzt. Entsprechend wird bei einem Befehl `glVertex3*()`, der nur die drei (homogenen) Koordinaten x, y und z festlegt, die w–Koordinate automatisch auf 1 gesetzt.

6.2.2 Grafik-Primitive in OpenGL

Um Vertices zu Polygonen und Linien zu verbinden oder sie einfach nur in Punkte umzuwandeln muss die entsprechende Information OpenGL jetzt noch übergeben werden. Dazu dient das sogenannte „*Begin/End-Paradigma*": alle Vertices, die zu einem Grafik-Primitiv-Typ gehören, werden zwischen einen Aufruf von `glBegin()` und `glEnd()` platziert und somit eingeklammert. Das Argument „`type`", das an den Befehl `glBegin(GLenum type)` übergeben wird, legt einen der zehn möglichen Grafik–Primitiv–Typen in OpenGL fest und somit das aus den Vertices geformte Objekt. In Bild 6.3 sind Beispiele für die jeweiligen Grafik-Primitive dargestellt.

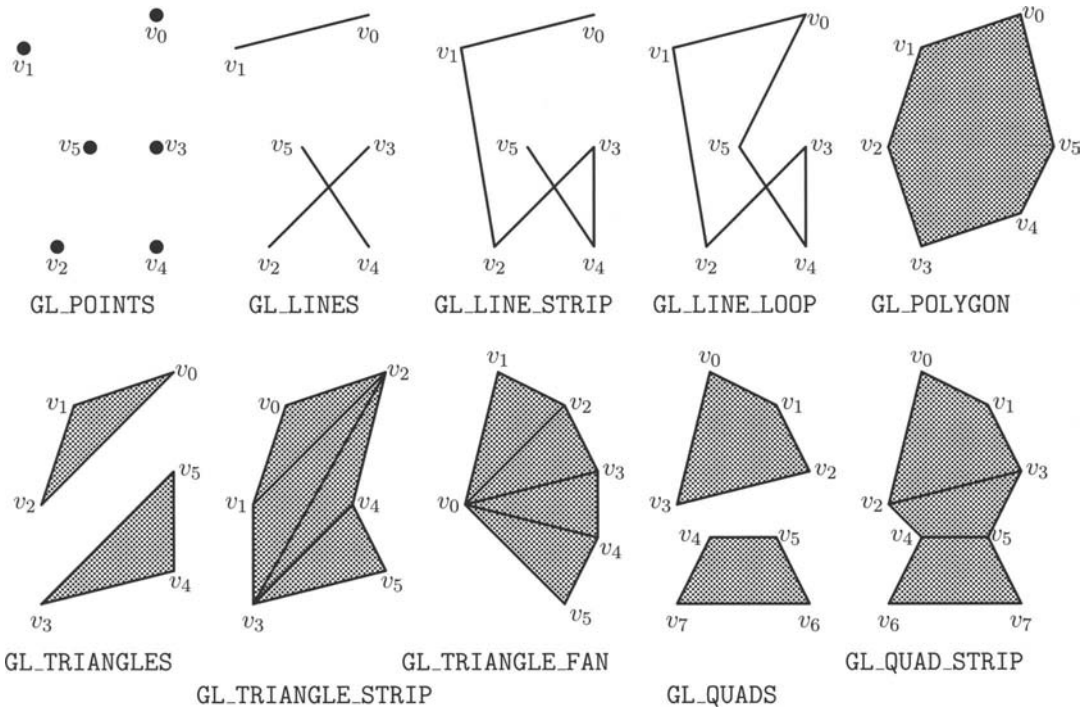


Bild 6.3: Beispiele für die Grafik-Primitive in OpenGL

type	**Grafik-Primitive in OpenGL**
GL_POINTS	Punkte: für jeden Vertex wird ein Punkt gerendert
GL_LINES	Linien (nicht verbunden): Liniensegmente werden zwischen den Vertices v_0 und v_1, v_2 und v_3 usw. gerendert. Bei einer ungeraden Anzahl von Vertices wird der letzte Vertex ignoriert
GL_LINE_STRIP	Linien (verbunden): Liniensegmente werden bei n Vertices zwischen v_0 und v_1, v_1 und v_2 usw. bis v_{n-2} und v_{n-1} gerendert. Der erste Vertex v_0 wird also mit dem letzten Vertex v_{n-1} nicht verbunden
GL_LINE_LOOP	vollständig geschlossener Linienzug: wie GL_LINE_STRIP, allerdings wird der erste Vertex v_0 mit dem letzten Vertex v_{n-1} verbunden
GL_POLYGON	konvexes Polygon (Vieleck): ein Polygon wird gerendert zwischen den Vertices v_0, v_1, usw. bis v_{n-1}. Bei weniger als 3 Vertices wird nichts gezeichnet
GL_TRIANGLES	Dreiecke (nicht verbunden): das erste Dreieck wird gerendert aus den Vertices v_0, v_1 und v_2, das zweite aus den Vertices v_3, v_4 und v_5 usw.; falls die Anzahl der Vertices kein ganzzahliges Vielfaches von drei ist, werden die überschüssigen Vertices ignoriert
GL_TRIANGLE_STRIP	Dreiecke (verbunden): das erste Dreieck wird gerendert aus den Vertices v_0, v_1 und v_2, das zweite aus den Vertices v_2, v_1 und v_3 (in dieser Reihenfolge), das dritte aus den Vertices v_2, v_3 und v_4 usw.; die Reihenfolge der Vertices ist wichtig, damit benachbarte Dreiecke eines Triangle-Strips die gleiche Orientierung bekommen und somit sinnvolle Oberflächen formen können (siehe auch den Abschnitt 6.2.3.4 über Polygone). Bei weniger als drei Vertices wird nichts gezeichnet
GL_TRIANGLE_FAN	Fächer aus Dreiecken: wie GL_TRIANGLE_STRIP, nur mit anderer Vertex-Reihenfolge: erstes Dreieck aus v_0, v_1 und v_2, das zweite aus v_0, v_2 und v_3, das dritte aus v_0, v_3 und v_4 usw.
GL_QUADS	Vierecke (nicht verbunden): das erste Viereck wird gerendert aus den Vertices v_0, v_1, v_2 und v_3, das zweite aus den Vertices v_4, v_5, v_6 und v_7 usw.; falls die Anzahl der Vertices kein ganzzahliges Vielfaches von vier ist, werden die überschüssigen Vertices ignoriert
GL_QUAD_STRIP	Vierecke (verbunden): das erste Viereck wird gerendert aus den Vertices v_0, v_1, v_3 und v_2, das zweite aus den Vertices v_2, v_3, v_5 und v_4 (in dieser Reihenfolge), das dritte aus den Vertices v_4, v_5, v_7 und v_6 usw.; bei einer ungeraden Anzahl von Vertices wird der letzte Vertex ignoriert

6.2.3 Programmierbeispiele

Im Folgenden werden praktische Programmierbeispiele für die jeweiligen Grafik–Primitiv–Typen von OpenGL angegeben. Dabei wird nur der jeweils relevante Ausschnitt des Code-Listings vorgestellt. Ein vollständig lauffähiges OpenGL-Programm auf der Basis der GLUT-Bibliothek zur Window- und Interaktionssteuerung ist auf den Webseiten zu diesem Buch verfügbar.

6.2.3.1 Punkte (GL_POINTS)

Punkte sind die einfachsten Grafik-Primitive in OpenGL. Durch das Argument `GL_POINTS` wird beim Befehl `glBegin()` festgelegt, dass alle nachfolgenden Vertices als Punkte gerendert werden, bis die Sequenz durch den Befehl `glEnd()` beendet wird (Bild 6.4). Man muss also nicht für jeden einzelnen Punkt die Befehlssequenz `glBegin()/glEnd()` aufrufen, sondern man sollte, wenn möglich, um Rechenzeit zu sparen, alle Punkte, die in einer Szene vorkommen, innerhalb einer `glBegin()/glEnd()`–Klammer auflisten.

```
GLfloat vec[3] = {20.0, 20.0, 0.0};

glBegin(GL_POINTS);
    glVertex3fv(vec);
    glVertex3f(80.0, 80.0, 0.0);
    glVertex3f(30.0, 80.0, 0.0);
    glVertex3f(70.0, 10.0, 0.0);
glEnd();
```

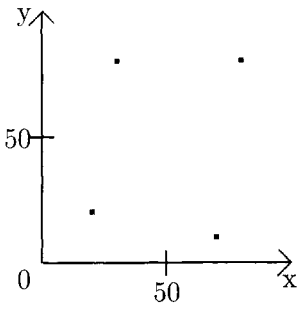


Bild 6.4: Punkte in OpenGL: links der relevante Programm-Code für die vier Punkte (ohne das Koordinatensystem), rechts die Bildschirmausgabe (die im Bild dargestellten Punkte sind wegen der besseren Sichtbarkeit 5 x 5 Pixel groß dargestellt; standardmäßig sind Punkte in OpenGL ein Pixel groß und quadratisch).

Punktgröße:

Punkte in OpenGL sind standardmäßig ein Pixel groß und quadratisch. Zur Veränderung der Größe eines Punktes dient der Befehl `glPointSize(GLfloat size)`. Das Argument „`size`" gibt den Durchmesser bzw. die Kantenlänge des Punktes in Pixel an. Nicht ganzzahlige Werte von size werden auf- oder abgerundet. Da die unterstützte Punktegröße hardwareabhängig ist, sollte man die zulässige untere und obere Schranke für die Punktegröße, sowie das Inkrement vom System abfragen. Dazu dient die folgende Befehlssequenz:

```
GLfloat sizes[2];
GLfloat incr;

glGetFloatv(GL_POINT_SIZE_RANGE, sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &incr);
```

Im Folgenden Bild 6.5 sieht man die Auswirkung des `glPointSize`-Befehls:

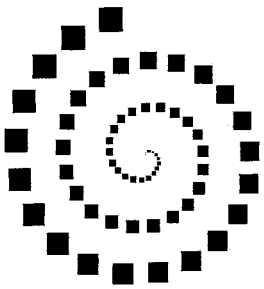
```
GLfloat x,y,z,r,phi,size; // Variablen
// Setze die Anfangswerte
size = sizes[0];
z = -80.0;
r = 0.0;

// Schleife über drei Kreisumläufe
for(phi = 0.0; phi <= 3.1415*6.0; phi += 0.3)
{
    // berechne x,y Werte auf einer Spirale
    x = r*sin(phi);
    y = r*cos(phi);

    // lege die Punktegröße fest
    glPointSize(size);

    // rendere den Punkt
    glBegin(GL_POINTS);
        glVertex3f(x, y, z);
    glEnd();

    // erhöhe z, r und size
    z += 2.5;
    r += 1.5;
    size += 3*incr;
}
```



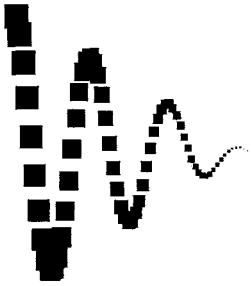


Bild 6.5: Unterschiedlich große Punkte in OpenGL: links der relevante Programm-Code, rechts oben die Bildschirmausgabe in der x–y–Ebene, rechts unten die Bildschirmausgabe in der y–z–Ebene (d.h. Drehung um die y–Achse um 90 Grad)

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_BLEND);
glBlendFunc(    GL_SRC_ALPHA,
                GL_ONE_MINUS_SRC_ALPHA);
```

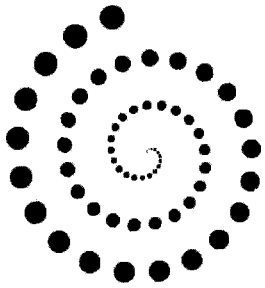


Bild 6.6: Runde Punkte in OpenGL: links der relevante Programm-Code zum Einschalten des Anti-Aliasing und der Transparenzberechnung, rechts die Bildschirmausgabe in der x–y–Ebene

Die bisher gerenderten Punkte waren alle quadratisch. Um runde Punkte zu bekommen, muss man in OpenGL den Zustand *Anti-Aliasing* (Kapitel 10) mit Hilfe des Befehls `glEnable(GL_POINT_SMOOTH)`, sowie die Transparenzberechnung (Kapitel 9) mit dem Befehl `glEnable(GL_BLEND)` einschalten. Um dem Auge einen Kreis vorzutäuschen, werden beim Anti-Aliasing die Randpixel relativ langsam vom Punkt zum Hintergrund übergeblendet. In diesem Modus ist es auch möglich, Punkte mit einem nichtganzzahligen Pixel–Durchmesser (z.B. 2.5 Pixel) darzustellen (das „halbe" Pixel erhält 50% seines Farbwertes vom Punkt und den Rest vom Hintergrund). In Bild 6.6 sieht man „runde" Punkte.

6.2.3.2 Linien (GL_LINES)

Durch das Argument `GL_LINES` wird beim Befehl `glBegin()` festgelegt, dass aus je zwei der nachfolgenden Vertices Linien gerendert werden, bis die Sequenz durch den Befehl `glEnd()` beendet wird (Bild 6.7).

```
GLfloat vec[3] = {20.0, 20.0, 0.0};

glBegin(GL_LINES);
    glVertex3fv(vec);
    glVertex3f(80.0, 80.0, 0.0);
    glVertex3f(30.0, 80.0, 0.0);
    glVertex3f(70.0, 10.0, 0.0);
glEnd();
```

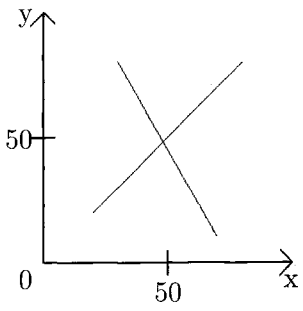


Bild 6.7: Linien in OpenGL: links der relevante Programm-Code, rechts die Bildschirmausgabe (die im Bild dargestellten Linien sind 1 Pixel breit)

Linienbreite:

Linien in OpenGL sind standardmäßig ein Pixel breit und stufenförmig (wenn sie nicht zufällig horizontal oder vertikal verlaufen). Zur Veränderung der Breite von Linien dient der Befehl `glLineWidth(GLfloat width)`. Das Argument „`width`" gibt den Querschnitt der Linie in x- oder y-Richtung in Pixel an. Nicht ganzzahlige Werte von `width` werden auf- oder abgerundet. Auch die unterstützte Linienbreite ist hardwareabhängig. Deshalb sollte man die zulässige untere und obere Schranke für die Linienbreite, sowie das Inkrement vom System abfragen. Dazu dient die folgende Befehlssequenz:

```
GLfloat sizes[2];
GLfloat incr;

glGetFloatv(GL_LINE_WIDTH_RANGE, sizes);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY, &incr);
```

In Bild 6.8 sieht man die Auswirkung des `glLineWidth`-Befehls.

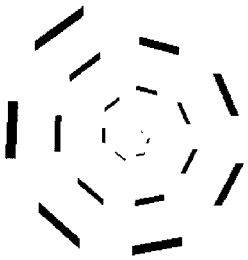


Bild 6.8: Unterschiedlich breite Linien in OpenGL: wie man sehen kann, sind Linien entweder in x–Richtung eine bestimmte Anzahl von Pixeln breit oder in y–Richtung. Deshalb sehen die Linienstücke wie kleine Parallelogramme aus. Der Programm-Code ist analog wie bei Bild 6.5.

Linienmuster (*Line Stipple*):

Linien kann man noch ein Muster aufprägen, d.h. man kann gepunktete, gestrichelte oder kombiniert gepunktete/gestrichelte Linien rendern. Dazu muss man bei dem Zustandsautomanten OpenGL zunächst den entsprechenden Schalter `GL_LINE_STIPPLE` mit dem Befehl `glEnable()` umlegen und anschließend mit dem Befehl `glLineStipple(GLint factor, GLushort pattern)` den Linien ein Muster (*pattern*) aufprägen. Das „`pattern`" ist eine 16-stellige Binärzahl, die pixelweise interpretiert wird: ist eine Binärstelle 1, wird das ent-

sprechende Pixel durch die Linie bedeckt, ist die Binärstelle 0, geschieht nichts. Ist die Linie länger als 16 Pixel, wird das Muster wiederholt. Falls verbundene Linien gerendert werden (Abschnitt 6.2.3.3), wird das Muster über die Ecken hinweg fortgesetzt, solange die Vertices innerhalb einer `glBegin/glEnd`–Klammer stehen. Das „`pattern`" wird nicht bitweise, sondern hexadezimal (0,1,..,9,A,B,C,D,E,F) eingegeben, jede Hex-Zahl repräsentiert vier bit, so dass das 16-stellige Bitmuster durch vier Hex-Zahlen spezifiziert werden kann. Das „`pattern`" „0x3FC9" z.B. repräsentiert das Bitmuster „0011 1111 1100 1001", d.h. das erste Pixel einer Linie wird gezeichnet, danach 2 Pixel nicht, das vierte Pixel wird wieder gezeichnet, danach ist wieder eine Lücke von 2 Pixeln, anschließend werden 6 Pixel gezeichnet, dann kommt wieder eine Lücke von 2 Pixeln und von da an beginnt es wieder von vorne. Der zweite Parameter von `glLineStipple()`, „`factor`" kann ganzzahlige Werte zwischen 1 und 255 annehmen. Er streckt das „`pattern`" eben um einen „`factor`" in die Länge. In Bild 6.9 sind einige Beispiele von Punkt- und Strichmustern dargestellt.

```
GLfloat y;
GLint factor = 1;
GLushort pattern = 0x3FC9;

// Stippling einschalten
glEnable(GL_LINE_STIPPLE);

for(y = -90.0; y <= 90.0; y += 20.0) {
    if(y > 0.0)
        pattern = 0x5555;
    // Streckungsfaktor und Muster setzen
    glLineStipple(factor,pattern);
    glBegin(GL_LINES);
        glVertex2f(-80.0, y);
        glVertex2f(80.0, y);
    glEnd();
    factor++; }
```

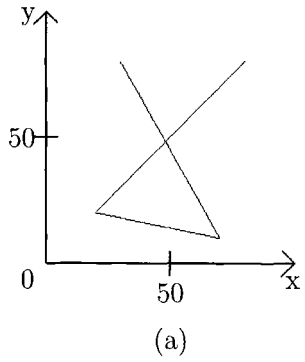
Bild 6.9: Linienmuster in OpenGL: links der relevante Programm-Code, rechts die Bildschirmausgabe. Die unteren 5 Linienmuster sind mit dem pattern 0x3FC9 (0011 1111 1100 1001) gerendert, wobei der Streckungsfaktor von 1 auf 5 ansteigt. Die oberen 5 Linienmuster sind mit dem pattern 0x5555 (0101 0101 0101 0101) gerendert, wobei der Streckungsfaktor von 6 auf 10 ansteigt.

6.2.3.3 Verbundene Linien (GL_LINE_STRIP, GL_LINE_LOOP)

Durch das Argument GL_LINES_STRIP wird beim Befehl glBegin() festgelegt, dass aus den nachfolgenden Vertices ein verbundener Linienzug zu rendern ist (Bild 6.10-a). Mit dem Argument GL_LINES_LOOP wird festgelegt, dass ein geschlossener Linienzug gerendert wird, bei dem auch noch der erste mit dem letzten Vertex durch ein Linienstück verbunden ist (Bild 6.10-b).

```
GLfloat vec[3] = {80.0,80.0,0.0};
glBegin(GL_LINE_STRIP);
    glVertex3fv(vec);
    glVertex3f(20.0,20.0,0.0);
    glVertex3f(70.0,10.0,0.0);
    glVertex3f(30.0,80.0,0.0);
glEnd();
```

```
GLfloat vec[3] = {80.0,80.0,0.0};
glBegin(GL_LINE_LOOP);
    glVertex3fv(vec);
    glVertex3f(20.0,20.0,0.0);
    glVertex3f(70.0,10.0,0.0);
    glVertex3f(30.0,80.0,0.0);
glEnd();
```



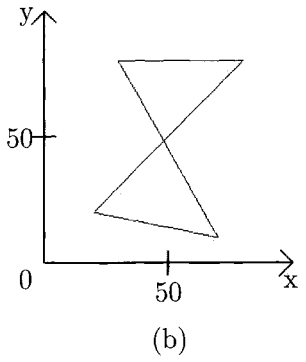


Bild 6.10: Linienzüge in OpenGL: (a) verbundene Linienzüge (GL_LINE_STRIP) und darüber der relevante Programm-Code, (b) geschlossene Linienzüge (GL_LINE_LOOP) und darüber der relevante Programm-Code

Außerdem kann man an den Beispielen in den Bildern 6.11-a,-b und -c sehen, dass man mit einer zunehmenden Zahl von geraden Linienstücken auch gekrümmte Linien immer besser approximieren kann. Um noch glattere Linien zu erhalten (Bild 6.11-d), kann man, wie schon bei den runden Punkten, den Zustand *Anti-Aliasing* (Kapitel 10) mit Hilfe des Befehls glEnable(GL_LINE_SMOOTH) einschalten.

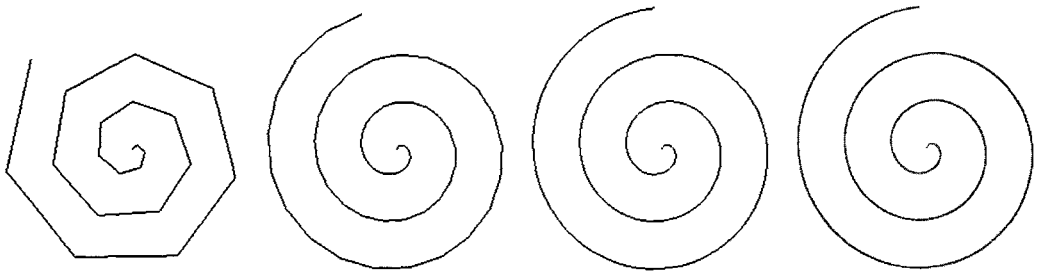


Bild 6.11: Linienzüge in OpenGL: jede gekrümmte Linie kann beliebig genau durch eine entsprechend große Anzahl gerader Linienstücke angenähert werden. (a) 20 Linien, (b) 62 Linien, (c) 188 Linien, (d) 188 Linien mit Anti-Aliasing.

6.2.3.4 Polygone (GL_POLYGON)

Durch das Argument `GL_POLYGON` wird beim Befehl `glBegin()` festgelegt, dass aus den nachfolgenden Vertices ein gefülltes Polygon (Vieleck) zu rendern ist, bis die Sequenz durch den Befehl `glEnd()` beendet wird (Bild 6.12).

```
GLfloat vec[3] = {80.0, 80.0, 0.0};

glBegin(GL_POLYGON);
    glVertex3fv(vec);
    glVertex3f(40.0, 80.0, 0.0);
    glVertex3f(10.0, 60.0, 0.0);
    glVertex3f(20.0, 20.0, 0.0);
    glVertex3f(70.0, 10.0, 0.0);
glEnd();
```

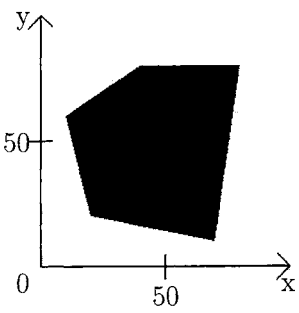


Bild 6.12: Ein gefülltes Polygon in OpenGL: links der relevante Programm-Code, rechts die Bildschirmausgabe

Konkave und Nicht-planare Polygone:

OpenGL macht einige Einschränkungen, was die Form von Polygonen angeht.

- Polygone dürfen nur konvex sein, d.h. nach außen gewölbt bzw. alle Innenwinkel kleiner 180 Grad, oder topologisch allgemeiner ausgedrückt in einer Definition:
  Ein Polygon ist konvex, falls alle Linien, die zwei beliebige Punkte des Polygons verbinden, vollständig innerhalb des Polygons liegen (Bild 6.13-a,-b).
- Polygone müssen planar sein, d.h. alle Eckpunkte müssen auf einer Ebene im Raum liegen (Bild 6.13-c,-d).

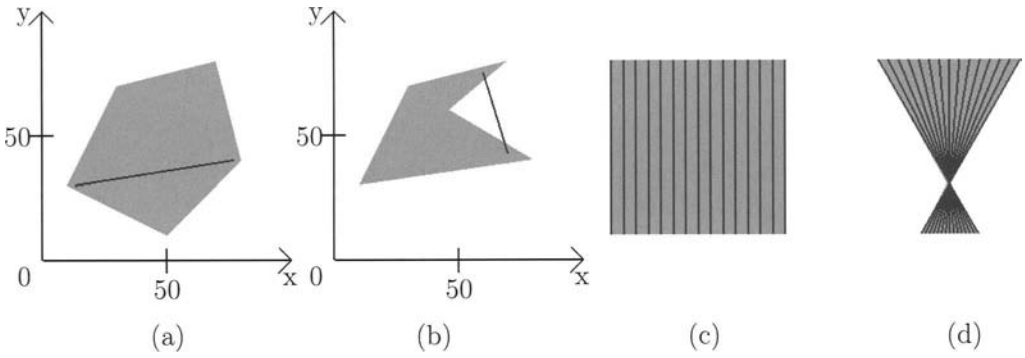


Bild 6.13: Erlaubte und verbotene Polygone in OpenGL: (a) erlaubt: konvexes Polygon (b) verboten: konkaves Polygon (c) erlaubt: planares Polygon (d) verboten: nicht-planares Polygon (entsteht aus dem planaren Polygon links daneben durch Drehung der unteren Kante um die Hochachse)

Der Grund für diese Einschränkungen beim Modellieren mit Polygonen (dies gilt ebenso für Vierecke (*Quads*), nicht jedoch für Dreiecke (*Triangles*), denn drei Punkte im Raum liegen immer in einer Ebene und bilden immer ein konvexes Polygon) liegt darin, dass die Algorithmen zur Rasterisierung und Verdeckungsrechnung unter der Annahme konvexer und planarer Polygone sehr einfach und daher schnell sind. Da außerdem konkave Polygone und nicht-planare Oberflächen beliebig genau durch einfache Polygone approximiert werden können, beschränkt man sich bei OpenGL auf diese Klasse von Polygonen, um eine maximale Rendering-Geschwindigkeit erreichen zu können. Um konkave Polygone, wie das in Bild 6.13-b gezeigte, korrekt rendern zu können, unterteilt man sie einfach in mehrere konvexe Polygone (Dreiecke und/oder Vierecke). Allerdings stürzt ein OpenGL-Programm nicht gleich ab, wenn man konkave oder nicht-planare Polygone definiert, sondern das Rendering-Ergebnis ist evtl. unerwartet, sowie blickrichtungs- und implementierungsabhängig. Da im Endeffekt jedes Polygon OpenGL-intern durch Dreiecks-Fächer (*Triangle Fan*) dargestellt wird, ist das Ergebnis bei der Eingabe eines konkaven oder nicht-planaren Polygons durchaus sicher vorhersagbar (Bild 6.14).

```
glBegin(GL_POLYGON);
    glVertex3f(80.0, 80.0, 0.0);
    glVertex3f(10.0, 10.0, 0.0);
    glVertex3f(70.0, 40.0, 0.0);
    glVertex3f(10.0, 60.0, 0.0);
    glVertex3f(40.0, 10.0, 0.0);
glEnd();
```

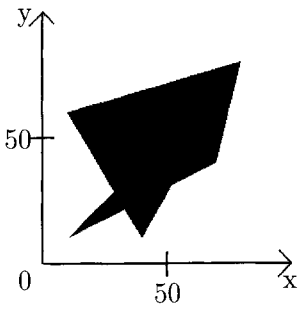


Bild 6.14: Konkaves Polygon in OpenGL: links der relevante Programm-Code, rechts die Bildschirmausgabe. Da OpenGL-intern jedes Polygon durch einen Dreiecks-Fächer dargestellt wird, bilden die Vertices Nr. 1,2,3 das erste Dreieck, die Vertices Nr. 1,3,4 das zweite Dreieck und die Vertices Nr. 1,4,5 das dritte Dreieck.

Polygonmuster (*Polygon Stipple*):

Ebenso wie Linien kann man Polygonen über eine 2-dimensionale binäre Maske der Größe 32 x 32 bit einfache Muster aufprägen (Bild 6.15). Dazu dient der Befehl `glPolygonStipple(GLubyte *mask)`. Außerdem muss noch der entsprechende Zustand in OpenGL aktiviert werden mit dem Befehl `glEnable(GL_POLYGON_STIPPLE)`. Da in OpenGL auch das bedeutend vielseitigere *Texture Mapping* (Kapitel 13) möglich ist, spielen Polygonmuster nur eine untergeordnete Rolle.

Bild 6.15: Polygonmuster in OpenGL

Polygon-Orientierung und *Backface-Culling*:

Polygone im 3-dimensionalen Raum können wie ein bedrucktes Blatt Papier unterschiedliche Vorder- und Rückseiten besitzen. Um die Vorder- und Rückseite in OpenGL unterscheiden zu können, wird per Konvention festgelegt, dass bei Polygonen, deren Vertices gegen den Uhrzeigersinn (*counterclockwise* (CCW)) auf dem Bildschirm erscheinen, die Vorderseite sichtbar ist (Bild 6.16). Durch die Reihenfolge der Vertices im Programm-Code wird also die Drehrichtung (*winding*) des Polygons und damit dessen Vorder- bzw. Rückseite festgelegt. Um eine „sinnvolle" Oberfläche für ein 3D-Objekt zu konstruieren, muss auf eine

konsistente Polygon-Orientierung geachtet werden, z.B. dass alle Vorderseiten des Objekts nach außen zeigen.

```
glBegin(GL_POLYGON);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
    glVertex3fv(v4);
glEnd();
```

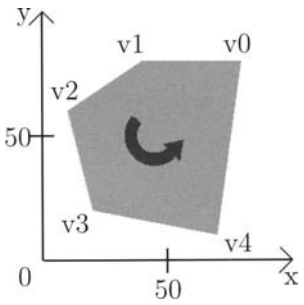


Bild 6.16: Polygon-Orientierung: links der relevante Programm-Code, rechts die Bildschirmausgabe (der Pfeil deutet die Drehrichtung (*winding*) des Polygons an; standardmäßig ist bei einer Drehrichtung gegen den Uhrzeigersinn (*counterclockwise* (CCW)) die Vorderseite des Polygons sichtbar)

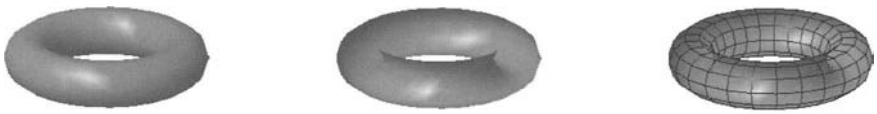
Die Konvention bzgl. Vorder- und Rückseite kann in OpenGL aber auch durch den Befehl `glFrontFace(GLenum mode)` umgedreht werden, indem das für das Argument „`mode`" die Konstante `GL_CW` (für *clockwise*, d.h. im Uhrzeigersinn) eingesetzt wird. Bei allen Polygonen, die nach diesem Befehl gezeichnet werden, ist die Vorderseite sichtbar, wenn die Vertices im Uhrzeigersinn auf dem Bildschirm angeordnet sind. Auf die Standardeinstellung zurückschalten kann man, wenn der Befehl `glFrontFace()` mit der Konstanten `GL_CCW` (für *counterclockwise*, d.h. gegen den Uhrzeigersinn) ausgeführt wird.

Bei einem undurchsichtigen 3-dimensionalen Objekt ist nur die Außenseite der Oberfläche sichtbar, solange man das Objekt nur von außen betrachtet. Oder anders ausgedrückt, bei einer konsistenten Polygon-Orientierung der Oberfläche sind die Rückseiten der Polygone nie sichtbar. Warum sollte man sie also rendern, solange man sich außerhalb des Objekts befindet? Ist der Betrachter dagegen innerhalb des Objekts, sind ausschließlich die Innenseiten der Oberfläche, d.h. die Rückseiten der Polygone sichtbar, so dass man die Vorderseiten der Polygone nicht rendern müsste. Standardmäßig werden in OpenGL immer beide Seiten, also Vorder- und Rückseite eines Polygons gerendert, da OpenGL ja zunächst nichts über die Position des Beobachters weiß. Außerdem gibt es Fälle in der 3D-Computergrafik, in denen sowohl Vorderseiten als auch Rückseiten sichtbar sind, wie z.B. bei halbtransparenten Oberflächen, durch die man ins Innere eines 3D-Objekts sehen kann, oder auch bei dem eingangs erwähnten Blatt Papier, das man ja einfach nur umzudrehen braucht, um die Rückseite zu sehen. Um OpenGL anzuweisen, die Vorder- oder Rückseiten von Polygonen wegzulassen (*culling*), dient der Befehl `glCullFace(GLenum mode)`. Als Argument „`mode`" zugelassen sind die Konstanten `GL_FRONT`, `GL_BACK` und `GL_FRONT_AND_BACK`, die festlegen, dass die Vorderseiten, Rückseiten oder beide Seiten weg-

gelassen werden. Außerdem muss noch der entsprechende Zustand in OpenGL aktiviert werden mit dem Befehl `glEnable(GL_CULL_FACE)`. Wie OpenGL-intern das sogenannte „*Backface-Culling*" durchgeführt wird, ist in Abschnitt 15.2 erläutert.

Polygonfüllung:

Um einen Blick ins Innere eines ansonsten undurchsichtigen 3D-Objekts werfen zu können, gibt es mehrere Möglichkeiten. Einerseits bietet sich der im vorigen Abschnitt erwähnte Befehl `glCullFace()` mit dem Argument `GL_FRONT` an, bei dem die Vorderseiten weggelassen werden und folglich der Blick auf die Rückseiten der Polygone, d.h. auf die Innenseiten des Objekts frei wird, wie in Bild 6.17-b dargestellt (ein ähnlicher Effekt könnte durch das Aufschneiden des Objekts mit zusätzlichen Schnittebenen (*clipping planes*) erreicht werden).



(a) (b) (c)

Bild 6.17: Polygonfüllung und *Culling* in OpenGL: (a) Torus im Normalmodus, d.h. Vorderseiten der Polygone zeigen nach außen und werden gefüllt, (b) Torus mit *Front-Face-Culling*, d.h. Vorderseiten der Polygone werden weggelassen, so dass der Blick auf die Rückseiten bzw. das Innere des Torus frei wird, (c) Torus mit Vorderseiten im Drahtgittermodell, Rückseiten gefüllt

Andererseits kann man bei den Vorderseiten die Füllung der Polygone weglassen und nur die Verbindungslinien zwischen den Vertices oder nur die Vertices selber rendern. Diese Methode erlaubt einen Blick ins Innere des 3D-Objekts und erhält gleichzeitig die Vorstellung von der äußeren Form (Bild 6.17-c).

Die Festlegung, in welchem Füll-Modus Polygone gerendert werden, geschieht mit dem Befehl `glPolygonMode(GLenum face, GLenum mode)`. Der Parameter `mode` kann die Werte `GL_FILL` (Standardwert), `GL_LINE` und `GL_POINT` annehmen, die festlegen, dass die entsprechenden Polygonseiten (*faces*) gefüllt, als Drahtgittermodell (*wireframe*) oder nur als Punkte gerendert werden. Für welche Polygonseiten der Füll-Modus gilt, wird mit dem Parameter `face` festgelegt, der die Werte `GL_FRONT` (Vorderseite), `GL_BACK` (Rückseite) oder `GL_FRONT_AND_BACK` annehmen kann. Der Effekt in Bild 6.17-c wird erreicht durch die Einstellung `glPolygonMode(GL_FRONT,GL_LINE)`, `glPolygonMode(GL_BACK,GL_FILL)`. Die Umschaltung auf ein vollständiges Drahtgittermodell, wie in Bild 6.1 dargestellt, erfolgt mit dem Befehl `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`.

6.2.3.5 Einzelne Dreiecke (GL_TRIANGLES)

Durch das Argument `GL_TRIANGLES` wird beim Befehl `glBegin()` festgelegt, dass aus je drei der nachfolgenden Vertices einzelne Dreiecke gerendert werden. Die Orientierung der beiden Dreiecke in Bild 6.18 ist unterschiedlich: während beim oberen Dreieck die Vertex-Reihenfolge bzgl. des Bildschirms gegen den Uhrzeigersinn läuft, ist es beim unteren Dreieck umgekehrt. Folglich sieht man bei der standardmäßigen Orientierungseinstellung von OpenGL (`glFrontFace(GL_CCW)`) die Vorderseite des oberen Dreiecks und die Rückseite des unteren.

```
glBegin(GL_TRIANGLES);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
    glVertex3fv(v4);
    glVertex3fv(v5);
glEnd();
```

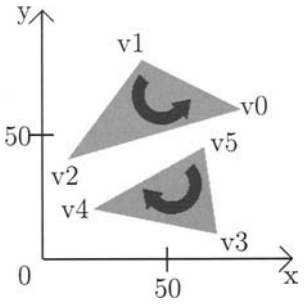


Bild 6.18: Einzelne Dreiecke in OpenGL: links der relevante Programm-Code, rechts die Bildschirmausgabe. Das obere Dreieck hat eine Orientierung gegen den Uhrzeigersinn, das untere im Uhrzeigersinn, d.h. bei der Standardeinstellung von OpenGL sieht man die Vorderseite des oberen Dreiecks und die Rückseite des unteren.

6.2.3.6 Verbundene Dreiecke (GL_TRIANGLE_STRIP)

Verbundene Dreiecke sind die am häufigsten verwendeten Grafik-Primitive in der Interaktiven 3D-Computergrafik. Dies hat eine Reihe von Gründen:

- Alle Oberflächen können durch einen Satz von Triangle-Strips beliebig genau approximiert werden.
- Es ist das einfachste Grafik-Primitiv zum Zeichnen komplexer Oberflächen.
- Es ist das speichersparendste (teil-planare) Grafik-Primitiv zum Zeichnen komplexer Oberflächen.
- Es ist das schnellste Grafik-Primitiv zum Zeichnen komplexer Oberflächen.
- Grafikhardware mit Geometrie-Beschleunigung ist fast immer auf Triangle-Strips optimiert. Eine klassisches Maß für die Leistung von Grafikhardware ist die Angabe

„triangles/sec“, gemeint ist die „Anzahl verbundener Dreiecke, die pro Sekunde“ gerendert werden können.

Durch das Argument `GL_TRIANGLE_STRIP` wird beim Befehl `glBegin()` festgelegt, dass aus den nachfolgenden Vertices verbundene Dreiecke gerendert werden. Für das erste Dreieck benötigt man drei Vertices, für jedes weitere Dreieck nur noch jeweils einen Vertex. Für n Dreiecke benötigt man also $n + 2$ Vertices. Dagegen benötigt man für n unabhängige Dreiecke $3n$ Vertices. Für eine große Anzahl von Dreiecken benötigt man folglich fast die dreifache Menge an Vertices, wenn man anstatt *Triangle-Strips* nur einzelne Dreiecke als Grafik-Primitiv verwendet. Auf dieser Einsparung beruht die hohe Effizienz von *Triangle-Strips* hauptsächlich.
Um die Orientierung aller Dreiecke eines *Triangle-Strips* konsistent zu halten, werden die Vertices nicht in der Reihenfolge verwendet, in der sie im Programm-Code spezifiziert sind (Bild 6.19), sondern bei jedem zweiten Dreieck wird die Reihenfolge der ersten beiden Vertices vertauscht, d.h. das erste Dreieck wird gerendert aus den Vertices v_0, v_1 und v_2, das zweite aus den Vertices v_2, v_1 und v_3 (in dieser Reihenfolge), das dritte aus den Vertices v_2, v_3 und v_4, das vierte aus den Vertices v_4, v_3 und v_5, usw..

```
glBegin(GL_TRIANGLE_STRIP);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
    glVertex3fv(v4);
    glVertex3fv(v5);
glEnd();
```

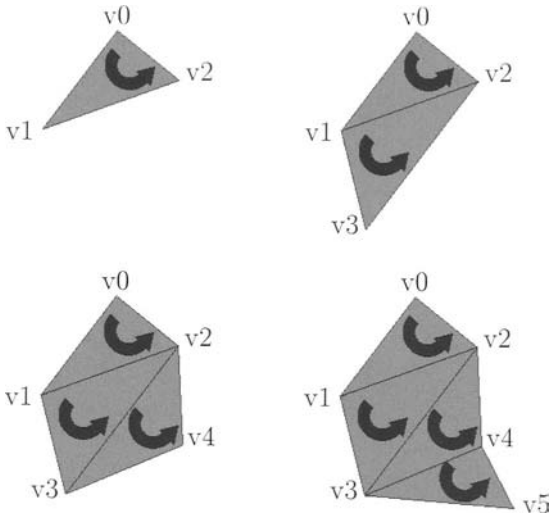


Bild 6.19: Verbundene Dreiecke (*Triangle-Strips*) in OpenGL: links der relevante Programm-Code, rechts die Bildschirmausgabe. Beim ersten Dreieck (links oben) ergibt sich die Orientierung aus den Vertices v_0, v_1 und v_2, beim zweiten Dreieck (rechts oben) aus den Vertices v_2, v_1 und v_3, beim dritten Dreieck (links unten) aus den Vertices v_2, v_3 und v_4 und beim vierten Dreieck (rechts unten) aus den Vertices v_4, v_3 und v_5. Durch diese Reihenfolge der Vertex-Verwendung bleibt die Orientierung aller Dreiecke eines *Triangle-Strips* konsistent.

6.2.3.7 Dreiecks-Fächer (GL_TRIANGLE_FAN)

Ein ebenso effektives Grafik-Primitiv wie der *Triangle-Strip* ist der Dreiecks-Fächer (*Triangle-Fan*). Dreiecks-Fächer werden in erster Linie zur Darstellung runder oder kegelförmiger Flächen benötigt. Durch das Argument `GL_TRIANGLE_FAN` wird beim Befehl `glBegin()` festgelegt, dass aus den nachfolgenden Vertices verbundene Dreiecke gerendert werden. Beim *Triangle-Fan* haben alle Dreiecke den ersten spezifizierten Vertex gemeinsam, so dass dieser Vertex auch das Zentrum des Fächers markiert (Bild 6.20). Auch hier wird die Orientierung aller Dreiecke eines *Triangle-Fans* konsistent gehalten, indem die Vertices in der folgenden Reihenfolge verwendet werden: das erste Dreieck wird gerendert aus den Vertices v_0, v_1 und v_2, das zweite aus den Vertices v_0, v_2 und v_3, das dritte aus den Vertices v_0, v_3 und v_4 usw..

```
glBegin(GL_TRIANGLE_FAN);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
    glVertex3fv(v4);
    glVertex3fv(v5);
glEnd();
```

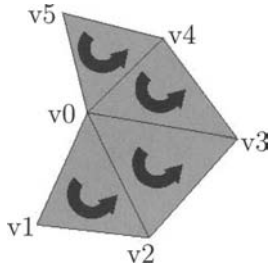


Bild 6.20: Dreiecks-Fächer (*Triangle-Fan*) in OpenGL: links der relevante Programm-Code, rechts die Bildschirmausgabe. Beim ersten Dreieck ergibt sich die Orientierung aus den Vertices v_0, v_1 und v_2, beim zweiten Dreieck aus den Vertices v_0, v_2 und v_3, beim dritten Dreieck aus den Vertices v_0, v_3 und v_4 usw. Durch diese Reihenfolge der Vertex-Verwendung bleibt die Orientierung aller Dreiecke eines *Triangle-Fans* konsistent.

6.2.3.8 Einzelne Vierecke (GL_QUADS)

Einzelne Vierecke (*Quads*) werden als Grafik-Primitiv relativ häufig benützt, z.B. zur Modellierung von Hauswänden, Wald- oder Baumkulissen, oder anderen quader- bzw. rechteckförmigen Objekten. Durch das Argument `GL_QUADS` wird beim Befehl `glBegin()` festgelegt, dass aus je vier der nachfolgenden Vertices einzelne Vierecke gerendert werden. Die Orientierung der beiden Vierecke in Bild 6.21 ist einheitlich: die Vertex-Reihenfolge bzgl. des Bildschirms läuft gegen den Uhrzeigersinn. Folglich sieht man bei der standardmäßigen Orientierungseinstellung von OpenGL (`glFrontFace(GL_CCW)`) die Vorderseite der beiden Vierecke.

```
glBegin(GL_QUADS);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
    glVertex3fv(v4);
    glVertex3fv(v5);
    glVertex3fv(v6);
    glVertex3fv(v7);
glEnd();
```

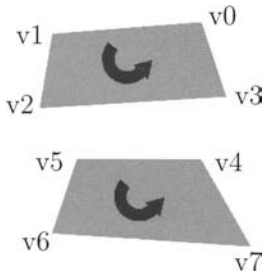


Bild 6.21: Einzelne Vierecke (*Quads*) in OpenGL: links der relevante Programm-Code, rechts die Bildschirmausgabe. Die beiden Vierecke haben die gleiche Orientierung, und zwar gegen den Uhrzeigersinn, d.h. bei der Standardeinstellung von OpenGL sind die Vorderseiten der Vierecke sichtbar.

6.2.3.9 Verbundene Vierecke (GL_QUAD_STRIP)

Durch das Argument `GL_QUAD_STRIP` wird beim Befehl `glBegin()` festgelegt, dass aus den nachfolgenden Vertices verbundene Vierecke gerendert werden. Für das erste Viereck benötigt man vier Vertices, für jedes weitere Viereck nur noch jeweils zwei Vertices. Für n verbundene Vierecke benötigt man also $2n + 2$ Vertices, also genau so viele Vertices, wie für $2n$ verbundene Dreiecke, die die Vierecke exakt ersetzen könnten. *Quad-Strips* sind also bzgl. der Vertex-Anzahl ebenso effektiv wie *Triangle-Strips*, allerdings bieten sie etwas weniger Flexibilität bei der Oberflächenmodellierung. Ein orientierungskonsistenter Kubus kann z.B. aus einem einzigen *Triangle-Strip* modelliert werden, *Quad-Strips* benötigt man aber drei (bzw. einen *Quad-Strip* und zwei einzelne *Quads*).
Um die Orientierung aller Vierecke eines *Quad-Strips* konsistent zu halten, werden die Vertices nicht in der Reihenfolge verwendet, in der sie im Programm-Code spezifiziert sind

(Bild 6.22), sondern bei jedem Viereck wird die Reihenfolge der letzten beiden Vertices vertauscht, d.h. das erste Viereck wird gerendert aus den Vertices v_0, v_1, v_3 und v_2, das zweite aus den Vertices v_2, v_3, v_5 und v_4 (in dieser Reihenfolge), das dritte aus den Vertices v_4, v_5, v_7 und v_6 usw..

```
glBegin(GL_QUAD_STRIP);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glVertex3fv(v3);
    glVertex3fv(v4);
    glVertex3fv(v5);
    glVertex3fv(v6);
    glVertex3fv(v7);
glEnd();
```

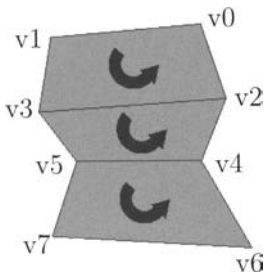


Bild 6.22: Verbundene Vierecke (*Quad-Strips*) in OpenGL: links der relevante Programm-Code, rechts die Bildschirmausgabe. Beim ersten Viereck ergibt sich die Orientierung aus den Vertices v_0, v_1, v_3 und v_2, beim zweiten aus den Vertices v_2, v_3, v_5 und v_4 und beim dritten aus den Vertices v_4, v_5, v_7 und v_6. Durch diese Reihenfolge der Vertex-Verwendung bleibt die Orientierung aller Vierecke eines *Quad-Strips* konsistent.

6.3 Tipps zur Verwendung der Grafik-Primitive

6.3.1 Rendering-Geschwindigkeit

Die 10 verschiedenen Grafik–Primitiv–Typen von OpenGL, die in diesem Kapitel ausführlich erklärt wurden, benötigen unterschiedlich lange Rechenzeiten, die in erster Linie von der mittleren Anzahl von Vertices pro Flächen- bzw. Linienelement abhängen (siehe nachfolgende Tabelle). In zweiter Linie hängt die Rendering-Geschwindigkeit auch noch von der jeweiligen OpenGL–Implementierung und der Grafikhardware ab: heutzutage bieten zwar fast alle Systeme Hardware-Beschleunigung für die Transformation von Triangle-Strips, aber nicht alle Systeme beschleunigen z.B. Linien. Deshalb kann es auf manchen Systemen günstiger sein, zu Linien degenerierte *Triangle-Strips* zu rendern, anstatt der eigentlich von OpenGL für diesen Zweck zur Verfügung gestellten Linien–Grafik–Primitive. Um wirklich sicher zu gehen, sollte man die Rechengeschwindigkeit für jeden Grafik–Primitiv–Typ auf seinem System messen. In der folgenden Tabelle sind qualitative Angaben zur Rendering-Geschwindigkeit der Grafik–Primitiv-Typen von OpenGL zusammengefasst, die auf Erfahrung beruhen:

Grafik-Primitiv-Typ	Rendering–Geschwindigkeit	Vertices pro Viereck bzw. Linienstück
GL_POINTS	langsam	–
GL_LINES	mittel	2
GL_LINE_STRIP	schnell	1.1 (bei 10 Linienstücken)
GL_LINE_LOOP	sehr schnell	1.0 (bei 10 Linienstücken)
GL_POLYGON	langsam	4.0
GL_TRIANGLES	schnell	6.0
GL_TRIANGLE_STRIP	am schnellsten	2.2 (bei 10 Vierecken)
GL_TRIANGLE_FAN	schnell	2.2 (bei 10 Vierecken)
GL_QUADS	schnell	4.0
GL_QUAD_STRIP	sehr schnell	2.2 (bei 10 Vierecken)

Immer wenn Oberflächen zu modellieren sind, sollte man in erster Linie *Triangle-Strips* verwenden, denn sie sind am schnellsten, flexibelsten und speicherplatzsparend. In den eher wenigen Fällen, in denen runde oder kegelförmige Oberflächen nötig sind, sollte man die relativ schnellen *Triangle-Fans* verwenden. Diese beiden Grafik-Primitive sind im Prinzip ausreichend, da mit ihnen jede beliebige Oberfläche modelliert werden kann. *Quad-Strips* sind zwar auch sehr schnell, bieten aber außer einer manchmal etwas einfacheren Modellierung keinen Vorteil gegenüber *Triangle-Strips*. Einzelne Flächen-Primitive, d.h. Polygone, *Triangles* und *Quads* sollten möglichst vermieden werden, denn sie sind meist deutlich langsamer. Außerdem kann jedes planare und konvexe Polygon durch einen *Triangle-Fan* dargestellt werden und jedes *Quad* durch einen *Triangle-Strip* aus zwei verbundenen Dreiecken. Bei Linien gilt im Prinzip das Gleiche wie bei Flächen: die verbundenen Varianten sind schneller und sollten wenn immer möglich bevorzugt werden. Punkte sollte man sehr spärlich einsetzen, denn sie sind oft ein echter Bremser. Anstatt Punkten kann man auch entsprechend kleine *Quads* einsetzen, oder wenn runde Punkte erforderlich sind, auch *Triangle-Fans*, die zu einem Kreis geformt sind.

6.3.2 Vertex Arrays

Eine weitere Möglichkeit die Rendering-Geschwindigkeit zu steigern, bietet OpenGL im Rahmen von sogenannten „Vertex Arrays“ an. Die Grundidee dabei ist, mit möglichst wenig Funktionsaufrufen auszukommen, um ein 3D-Objekt zu rendern, denn Funktionsaufrufe sind mit viel Overhead verbunden und daher Rechenzeitfresser. Um z.B. einen Kubus zu rendern, benötigt man bei geschickter Modellierung, d.h. unter Verwendung eines Triangle-Strips, 14 Vertices: die 6 Kubus-Vierecke unterteilt man in 12 Dreiecke, wobei man für das erste Dreieck 3 Vertices braucht und für die 11 weiteren Dreiecke 11 Vertices (bei ungeschickter Modellierung, d.h. unter Verwendung von *Quads* benötigt man 24 Vertices, je 4 Vertices für die 6 Kubus-Vierecke). Jeder Vertex muss mit einem Funktionsaufruf `glVertex*()` spezifiziert werden. Inclusive der beiden Funktionsaufrufe für `glBegin()` und `glEnd()` benötigt man also mindestens 16 Funktionsaufrufe für einen ein-

fachen Kubus. Ein weiterer Nachteil kommt noch hinzu: ein Kubus wird durch 8 Eckpunkte im Raum aufgespannt, d.h. 6 Vertices müssen auch bei geschickter Modellierung zweifach spezifiziert werden. Bei komplexeren 3D-Objekten, oder wenn zusätzlich noch Normalen-Vektoren und Textur-Koordinaten für jeden Vertex spezifiziert werden, vervielfacht sich die Zahl der Funktionsaufrufe. OpenGL reduziert mit Hilfe von Vertex Arrays die Zahl der Funktionsaufrufe drastisch:

- Einschalten des Vertex–Array–Modus mit dem Befehl `glEnableClientState(GL_VERTEX_ARRAY)`
- Anordnung der Vertex–Daten in einem Array, dessen Elemente direkt nacheinander im Speicher liegen und Übergabe eines Zeigers auf das erste Element des Arrays mit dem Befehl `glVertexPointer()`
- Dereferenzierung des Vertex-Daten z.B. mit dem Befehl `glDrawElements(GL_TRIANGLE_STRIP, 14` (Anzahl der Vertices), `GL_UNSIGNED_BYTE, vertices` (Zeiger auf das Vertex–Array))

Diese 3 Funktionsaufrufe haben die gleiche Wirkung wie die 16 Aufrufe in der standardmäßigen glBegin()/glEnd()–Darstellung beim zeichnen eines Kubus'. Eine ausführliche Beschreibung der vielfältigen Möglichkeiten von Vertex Arrays ist in [Shre05] und [Wrig05] zu finden.

6.3.3 Konsistente Polygon-Orientierung

Bei der Konstruktion von geschlossenen Oberflächen sollte darauf geachtet werden, dass alle Polygon-Orientierungen konsistent sind, d.h. dass alle Vorderseiten von Polygonen von außerhalb der 3D-Objekte sichtbar sind und alle Rückseiten nur von innerhalb (oder umgekehrt). Dadurch ist es möglich, in bestimmten Anwendungen das *Backface-Culling* zu benutzen, bei dem durch Weglassen der Polygon-Rückseiten die Rendering-Geschwindigkeit gesteigert werden kann. Falls Vorder- und Rückseiten von Polygonen gleichzeitig sichtbar sind, aber unterschiedliche Materialeigenschaften aufweisen (z.B. Farbe oder Textur), ist eine konsistente Polygon-Orientierung ebenso Grundvoraussetzung.

6.3.4 Koordinaten-Berechnungen offline

Der z.B. in Bild 6.5 dargestellte Programm-Code enthält eine Sinus- und eine Cosinus-Berechnung je Vertex zur Bestimmung der Vertexkoordinaten. Dies kostet natürlich Rechenzeit, die vor allem dann sehr stark zu Buche schlagen kann, wenn das definierte Objekt mehrfach in einem Bild gerendert werden muss. Deshalb sollte man Koordinaten-Berechnungen generell offline, d.h. in einer Initialisierungs–Routine, oder außerhalb des OpenGL-Programms durchführen. Zusätzlich bietet OpenGL die Möglichkeit, Befehle in sogenannten „*Display Listen*“ zusammenzufassen und in einer compilierten Form zwischenzuspeichern. Bei mehrfacher Nutzung eines Objekts, das in einer *Display Liste* gespeichert

wird, müssen alle Berechnungen nur einmal durchgeführt werden, die Ergebnisse können man dann beliebig oft nutzen. Eine ausführliche Beschreibung von *Display Listen* ist in [Shre05] zu finden.

6.3.5 Oberflächen-Tessellierung

Je feiner die Tessellierung einer Oberfläche, d.h. je kleiner die Mosaiksteinchen, sprich die Polygone, aus denen eine Oberfläche zusammengesetzt ist, desto genauer kann die Form von 3D-Objekten nachgebildet werden und desto realistischer erscheinen die Beleuchtungseffekte (vergleiche Bild 6.17-a mit Bild 6.1-d). Allerdings zahlt man dafür natürlich einen Preis in Form einer höheren Rechendauer. Letztendlich muss man also abwägen zwischen Bildqualität und Bildgenerierrate. Die Kunst besteht darin, bei einer geforderten Bildgenerierrate eine möglichst hohe Bildqualität zu erreichen. Dazu nur ein paar generelle Hinweise:

- Falls ein 3D-Objekt nur auf wenigen Pixeln am Bildschirm erscheint, sollte es auch nur aus wenigen Polygonen bestehen.
- Falls man relativ flache Oberflächen modelliert, sollte man wenige große Polygone benutzen, in raueren Gebieten muss man viele kleinere Polygone benutzen (oder *Bump Mapping* einsetzen, wie in Abschnitt 13.5 beschrieben).
- Falls ein Objekt rotiert, sollte man versuchen, die Silhouette des 3D-Objektes genauer zu modellieren als das Innere.

6.3.6 Lücken

Aufgrund von numerischen Ungenauigkeiten kommt es bei ungeschickter Modellierung manchmal zu kleinen Lücken in geschlossenen Oberflächen. Deshalb sollte man ein T-förmiges Aneinanderstoßen von Polygonen (Dreiecken, Vierecken) vermeiden. Wie in Bild 6.23 gezeigt, könnte der Punkt v4 der oberen beiden Dreiecke aufgrund numerischer Ungenauigkeiten an einem anderen Pixel erscheinen als ein entsprechender Punkt des unteren Dreiecks. Um dies zu vermeiden, sollte man daher das untere Dreieck teilen, so dass alle vier Dreiecke einen gemeinsamen Vertex v4 besitzen.

Weitere Fehler werden häufig bei der Definition geschlossener Kreise gemacht, wie an dem Programm–Code in Bild 6.5 zu sehen ist: die FOR-Schleife

```
for(angle = 0.0; angle <= 3.1415; angle += 0.3)
glVertex2f(cos(angle), sin(angle));
```

enthält zwei Fehler: erstens endet die Laufvariable „`angle`“ nicht bei 3.1415 sondern bei 3.0 (dies kann man einfach dadurch beheben, dass man ein korrektes Inkrement wählt: `angle += 3.1415/incr`), und zweitens ist selbst nach dieser Korrektur nicht garantiert, dass z.B. `cos(3.1415)` das gleiche Ergebnis liefert wie `cos(0.0)`, denn die Kreiszahl π ist ja eine

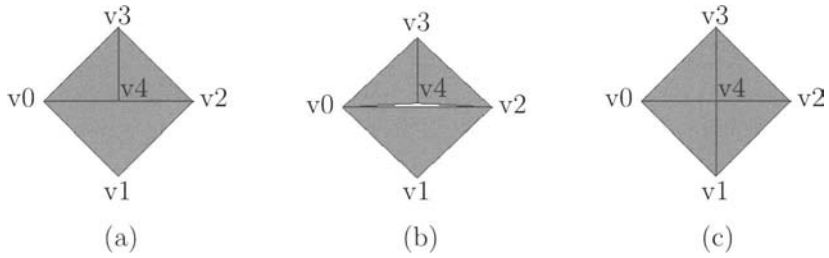


Bild 6.23: Problem T-förmiges Aneinanderstoßen von Polygonen: (a) T-förmige Polygonschnittstelle am Punkt v4 (b) wie das Bild links, aber aus einem anderen Blickwinkel, so dass eine kleine Lücke sichtbar wird (c) korrekt modellierte Dreiecke (alle 4 Dreiecke teilen sich den Vertex v4)

reelle Zahl, die nicht durch eine endliche Zahl von Dezimalstellen nach dem Komma darstellbar ist. Beheben kann man diesen Mangel dadurch, dass man die FOR–Schleife beim vorletzten Inkrement abbricht (d.h. `angle < 3.1415`) und das letzte Inkrement durch das Nullte ersetzt, d.h. am Ende noch einmal `angle = 0.0` setzt. Generell sollte man darauf achten, dass Vertices benachbarter Flächen nicht unabhängig voneinander berechnet, sondern gemeinsam genutzt werden.

6.3.7 Hinweise zum glBegin/glEnd-Paradigma

Die geometrische Modellierung von 3-dimensionalen Objekten in OpenGL basiert auf planaren Polygonen, die durch ihre Vertices definiert sind. Darauf aufbauend laufen sehr viele weitere Berechnungsverfahren, wie Beleuchtung, Texturzuordnung und Nebel einmal pro Vertex ab. Deshalb ist die Grundidee, alle Vertex-bezogenen Daten, die zu einem Grafik-Primitiv-Typ gehören, in eine `glBegin/glEnd`-Klammer zu setzen. Außer den räumlichen Koordinaten (`glVertex*()`) können jedem Vertex noch folgende weitere Eigenschaften zugewiesen werden: Farbe (`glColor*()`, `glSecondaryColor*()`, `glIndex*()`), Texturkoordinaten (`glTexCoord*()`, `glMultiTexCoord*()`), Normalenvektoren (`glNormal*()`) und Nebelkoordinaten (`glFogCoord*()`). Bis auf wenige Ausnahmen sind alle anderen Befehle von OpenGL innerhalb einer `glBegin/glEnd`-Klammer entweder nicht erlaubt oder nicht sinnvoll. Außerdem ist zu beachten, dass `glBegin/glEnd`-Klammern nicht ineinander verschachtelbar sind.

6.4 Modellierung komplexer 3D-Szenarien

Wie man sicher beim Lesen dieses Kapitels schon bemerkt hat, ist die Modellierung von komplexen 3D-Objekten oder sogar ganzer 3D-Szenarien in OpenGL ein sehr aufwändiges Unterfangen. Zudem ist die Modellierung in OpenGL nicht besonders anschaulich, da die Eingabe von 3D-Objekten ausschließlich über alphanumerische Programmzeilen erfolgt, die auch noch compiliert werden müssen, bevor man das Ergebnis der Modellierung am Bildschirm sehen kann. OpenGL selbst stellt auch keine Funktionen für das Abspeichern und Einlesen von 3D-Szenarien oder Texturen zur Verfügung, um die Plattform–Unabhängigkeit zu erhalten. OpenGL stellt die grundlegenden Funktionalitäten zur Verfügung, auf der andere Werkzeuge aufbauen.

Aus diesem Grund wurden für die verschiedenen Anwendungsgebiete der 3D-Computergrafik eine Reihe von Modellier–Werkzeugen geschaffen, die über eine grafische Benutzeroberfläche die interaktive Generierung und Abspeicherung von 3D-Szenarien gestatten. Jedes Anwendungsgebiet hat dabei seine Spezifika, die sich in eigenen 3D-Datenformaten niederschlagen. Es gibt mittlerweile sicher deutlich mehr als hundert verschiedene 3D-Datenformate und noch mehr Modellier–Werkzeuge. Ein kleine Auswahl davon für die Anwendungsgebiete Simulation und Animation ist in folgender Liste zu finden:

- „3d studio max“ von discreet.
- „MAYA“ von Alias|Wavefront.
- „Creator“ von Multigen-Paradigm Inc.
- „Terra Vista“ von Terrex Inc.
- „Softimage|XSI“ von AVID Technology.
- „Cinema 4D“ von Maxon Computer GmbH.

Kapitel 7

Koordinatensysteme und Transformationen

Im vorhergehenden Kapitel wurde gezeigt, wie man geometrische Objekte in der 3D-Computergrafik modelliert. In diesem Kapitel wird dargestellt, wie man diese Objekte in einer Szene positioniert, wie man die Position und die Blickwinkel einer Kamera festlegt, die die Szene quasi fotografiert, und wie man schließlich die Ausmaße des fertigen Bildes spezifiziert, das in einem Fenster des Bildschirms dargestellt werden soll. All diese Aktionen werden durch entsprechende Koordinatentransformationen erreicht.

7.1 Definition des Koordinatensystems

In OpenGL wird ein 3-dimensionales Euklidisches Koordinatensystem verwendet, dessen 3 senkrecht aufeinander stehende Achsen mit x, y und z bezeichnet werden. Standardmäßig sitzt der Betrachter (auch Augenpunkt genannt) im Ursprung ($x = 0, y = 0, z = 0$), die positive x–Achse zeigt bzgl. des Beobachters bzw. des Bildschirms nach rechts, die positive y–Achse zeigt nach oben und die positive z–Achse zeigt in Richtung des Beobachters, d.h. sie steht senkrecht auf dem Bildschirm in Richtung des Augenpunkts[1] (Bild 7.1).

Dadurch ist ein sogenanntes „rechtshändiges" Koordinatensystem definiert, bei dem der Daumen der rechten Hand die positive x–Achse repräsentiert, der Zeigefinger die positive y–Achse und der auf den beiden anderen senkrecht stehende Mittelfinger die positive z–Achse.

[1] Achtung: in der Bildverarbeitung wird traditionell ein anders ausgerichtetes Koordinatensystem verwendet. Der Ursprung sitzt im Bild links oben, die positive x–Achse zeigt nach unten, die positive y–Achse zeigt nach rechts und die positive z–Achse zeigt in Richtung des Beobachters, d.h. sie steht senkrecht auf dem Bildschirm in Richtung des Augenpunkts. Dieses Koordinatensystem, das auch im Bildverarbeitungsteil dieses Werks verwendet wird, geht aus dem Koordinatensystem der Computergrafik durch Drehung um $-90°$ bzgl. der z–Achse hervor.

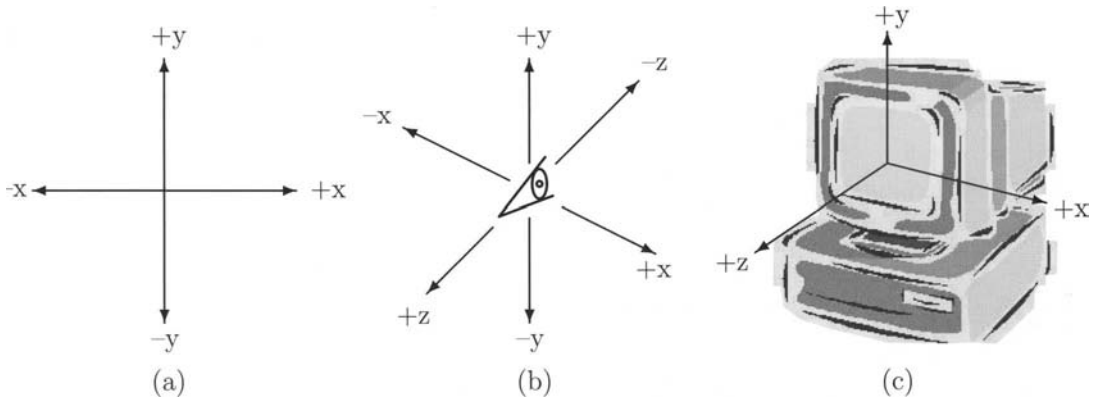


Bild 7.1: Die Definition des Koordinatensystems in OpenGL aus verschiedenen Perspektiven: (a) aus der Perspektive des Augenpunkts, (b) aus der Perspektive eines dritten Beobachters, der nach rechts oben versetzt ist und auf den Ursprung des Koordinatensystems blickt, in dem der Augenpunkt standardmäßig sitzt. (c) aus einer ähnlichen Perspektive wie in der Mitte, aber jetzt mit Blick auf den Bildschirm

7.2 Transformationen im Überblick

Der Titel des vorigen Abschnitts „Definition des Koordinatensystems" klingt so, als ob es in der 3D-Computergrafik nur ein einziges Koordinatensystem gäbe. Tatsächlich aber werden mehrere Koordindatensysteme unterschieden:

- *Weltkoordinaten* sind die Koordinaten, die für die gesamte Szene gelten und im vorigen Abschnitt definiert wurden. Manchmal werden diese Koordinaten auch Augenpunktkoordinaten (*eye coordinates*) genannt[2].
- *Objektkoordinaten* (*object coordinates*) sind die Koordinaten, in denen die 3D-Objekte lokal definiert werden.
- *Projektionskoordinaten* (*clip coordinates*) sind die Koordinaten, die nach der perspektivischen oder parallelen Projektionstransformation gelten.
- *normierte Koordinaten* (*normalized device coordinates*) sind die Koordinaten, die nach der Division der Projektionskoordinaten durch den inversen Streckungsfaktor w entstehen.
- *Bildschirmkoordinaten* (*window coordinates*) sind die Koordinaten am Ende der ganzen Transformations–Kette, die die Szene in der gewählten Fenstergröße darstellen.

[2]Manchmal wird zwischen Weltkoordinaten und Augenpunktkoordinaten auch stärker unterschieden. In diesem Fall bezeichnet man die Koordinaten nach der Modelltransformation als Weltkoordinaten und erst nach der Augenpunkttransformation als Augenpunktkoordinaten. Da in OpenGL diese beiden Transformationen jedoch in der ModelView-Matrix zusammengefasst sind, werden hier Welt- und Augenpunktkoordinaten äquivalent benutzt

Das Bild 7.2 zeigt einen Überblick über die Transformationsstufen, durch die man die Vertices in die zugehörigen Koordinatensysteme überführt.

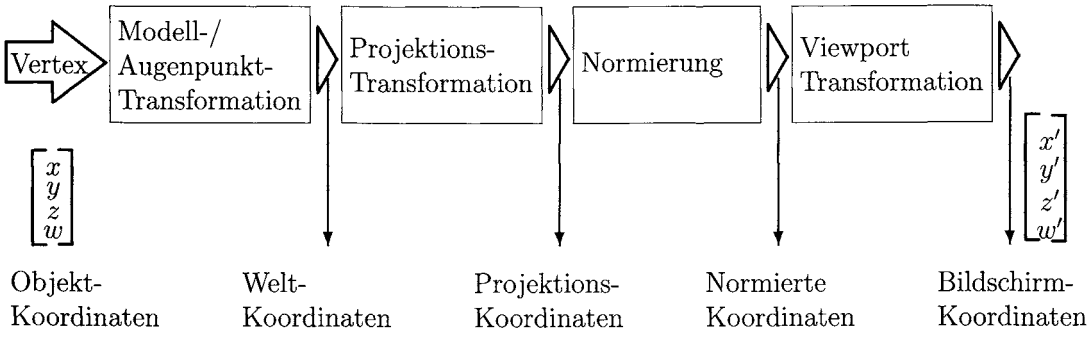


Bild 7.2: Darstellung der verschiedenen Vertex Transformationen und der zugehörigen Koordinatensysteme

Als Erstes werden die in einem lokalen Koordinatensystem definierten 3D-Objekte an eine beliebige Stelle in der Szene positioniert. Dazu wird auf die Vertices des Objekts eine *Modelltransformation* (Translation, Rotation oder Skalierung) angewendet. Als Beispiel betrachten wir die Modellierung eines einfachen Automobils: in einem lokalen Koordinatensystem, d.h. in Objektkoordinaten, wird jeweils ein Chassis und ein Rad definiert; das Rad wird 4-fach verwendet und jeweils durch eine Translation bzw. Rotation an die richtige Stelle am Chassis angebracht.

Als Zweites wird festgelegt, von welchem Augenpunkt aus die Szene „fotografiert" werden soll. Dazu wird auf alle Vertices der Szene eine *Augenpunkttransformation* angewendet, also wieder wie vorher eine Translation, Rotation oder Skalierung. Dabei ist es vollkommen egal, ob man den Augenpunkt verschiebt, oder ob man den Augenpunkt im Ursprung belässt und die gesamte Szene verschiebt, das Ergebnis ist das gleiche. Nun sind alle Vertices im Weltkoordinatensystem gegeben. Da sich die ersten beiden Transformationen in ihrer Form nicht unterscheiden, werden sie zur Modell- und Augenpunkttransformation (*Modelview-Matrix*) zusammengefasst.

Als Drittes werden die Eigenschaften der Projektion, wie z.B. die Blickwinkel, festgelegt. Dies entspricht quasi der Auswahl eines Objektivs (Weitwinkel, Tele) bzw. eines Zoomfaktors bei einer Kamera. Dazu wird auf die Weltkoordinaten die *Projektionstransformation* angewendet, die das sichtbare Volumen („*viewing volume*" oder „*viewing frustum*") definiert. Alle Vertices, die außerhalb des sichtbaren Volumens liegen, werden jetzt entfernt (*clipping*), da sie am Bildschirm sowieso nicht sichtbar wären. Nach diesem Transformationsschritt liegen die Vertex-Koordinaten als „Projektionskoordinaten" vor (im Englischen auch als „*clip coordinates*" bezeichnet, da nach dieser Transformation das *Clipping* (Abschnitt 15.2) durchgeführt wird).

Als Viertes wird eine Normierung (*perspective division*) in zwei Schritten durchgeführt. Im ersten Schritt werden die x, y, z–Koordinaten in das Intervall $[-w, +w]$ transformiert. Im zweiten Schritt werden die Koordinaten x, y, z durch den inversen Streckungsfaktor w dividiert. Auf diese Weise werden „Normierte Koordinaten" (*normalized device coordinates*) erzeugt.

Als Fünftes und Letztes wird die *Viewport-Transformation* durchgeführt. Abhängig von den in Pixel definierten Ausmaßen des Bildschirmfensters werden durch die Viewport–Transformation die Vertices in Bildschirmkoordinaten umgerechnet.

Man kann sich natürlich fragen, wieso nach der Projektion einer 3-dimensionalen Szene auf einen 2-dimensionalen Bildschirm überhaupt noch mit 3-dimensionalen Vertices weitergerechnet wird, für die Ortsbestimmung eines Bildschirm-Pixels reichen an sich ja die beiden Koordinaten x und y. Dennoch werden alle weiteren Transformationen auch für die z–Koordinate durchgeführt, denn die z–Koordinate repräsentiert die räumliche Tiefe eines Vertex bezogen auf den Bildschirm. Diese Information ist für die Verdeckungsrechnung (Kapitel 8) und die Nebelberechnung (Kapitel 11) sehr nützlich.

7.3 Mathematische Grundlagen

Um ein tieferes Verständnis der Transformationen in der 3D-Computergrafik zu bekommen, werden zunächst in knapper Form die wesentlichen mathematischen Grundlagen dazu erläutert. Für eine ausführlichere Darstellung der mathematischen Grundlagen wird auf [Fole96] verwiesen.

7.3.1 Homogene Koordinaten

Ein Punkt im 3-dimensionalen Euklidischen Raum kann durch die drei Koordinaten x, y, z beschrieben werden. Eine beliebige Transformation eines 3-komponentigen Orts–Vektors $(x, y, z)^T$ kann durch eine $3 \cdot 3$-Matrix erreicht werden:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \quad \text{mit} \quad \begin{aligned} x' &= m_{11} \cdot x + m_{12} \cdot y + m_{13} \cdot z \\ y' &= m_{21} \cdot x + m_{22} \cdot y + m_{23} \cdot z \\ z' &= m_{31} \cdot x + m_{32} \cdot y + m_{33} \cdot z \end{aligned} \tag{7.1}$$

Um auch Translationen durchführen zu können, die unabhängig von den Objektkoordinaten sind, fehlt hier aber eine unabhängige additive Komponente. Zu diesem Zweck werden homogene Koordinaten eingeführt, d.h. ein Punkt im 3-dimensionalen Euklidischen Raum wird jetzt durch die vier Komponenten x_h, y_h, z_h, w beschrieben, wobei die vierte Komponente w „inverser Streckungsfaktor" genannt wird. Der Grund für diesen Namen wird ersichtlich, wenn man einen Vertex mit homogenen Koordinaten in den 3-dimensionalen

Euklidischen Raum abbildet, denn die jeweilige euklidische Koordinate ergibt sich durch Streckung (Multiplikation) der homogenen Koordinate mit dem invertierten Faktor w^{-1}:

$$x = \frac{x_h}{w}; \quad y = \frac{y_h}{w}; \quad z = \frac{z_h}{w} \tag{7.2}$$

Standardmäßig ist der inverse Streckungsfaktor $w = 1$. Bei einem inversen Streckungsfaktor $w = 0.5$ werden die Koordinaten um einen Faktor 2 gestreckt, bei $w = 0.05$ um einen Faktor 20 und was passiert bei $w = 0.0$? Da eine Division durch 0 unendlich ergibt, wird ein Punkt mit den homogenen Koordinaten $(x, y, z, 0)$ ins Unendliche abgebildet, und zwar in Richtung des Vektors $(x, y, z)^T$. Mit einem inversen Streckungsfaktor $w = 0$ können somit Richtungsvektoren definiert werden. Dies ist z.B. bei der Beleutungsrechnung sehr nützlich, denn mit $w = 0$ kann eine unendlich ferne Lichtquelle definiert werden, die in eine bestimmte Richtung perfekt parallele Lichtstrahlen aussendet. Die Beleuchtungsrechnung vereinfacht sich dadurch erheblich, denn es muss nicht für jeden Vertex extra die Richtung des eintreffenden Lichtstrahls berechnet werden, sondern die Richtung ist für alle Vertices die gleiche.

Eine von den Objektkoordinaten unabhängige Translation eines Vertex (x, y, z, w) um einen Richtungs–Vektor $(T_x, T_y, T_z)^T$ kann jetzt im Rahmen einer $4 \cdot 4$-Matrix zusätzlich zu den in (7.1) definierten Transformationen durchgeführt werden:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & T_x \\ m_{21} & m_{22} & m_{23} & T_y \\ m_{31} & m_{32} & m_{33} & T_z \\ 0 & 0 & 0 & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{7.3}$$

wobei

$$\begin{aligned} x' &= m_{11} \cdot x + m_{12} \cdot y + m_{13} \cdot z + T_x \cdot w \\ y' &= m_{21} \cdot x + m_{22} \cdot y + m_{23} \cdot z + T_y \cdot w \\ z' &= m_{31} \cdot x + m_{32} \cdot y + m_{33} \cdot z + T_z \cdot w \\ w' &= m_{44} \cdot w \end{aligned}$$

7.3.2 Transformations–Matrizen

Die Geometrie aller 3D-Objekte wird, wie im vorigen Kapitel 6 gezeigt, durch einen Satz von Vertices definiert. Jeder Vertex wird in homogenen Koordinaten durch einen 4-komponentigen Orts–Vektor $\mathbf{v} = (x, y, z, w)^T$ gegeben. Eine allgemeine Transformation eines 4-komponentigen Vektors $\mathbf{v}$ kann durch eine $4 \cdot 4$-Matrix $\mathbf{M}$ erreicht werden:

$$\mathbf{v}' = \mathbf{M}\mathbf{v} \quad \Leftrightarrow \quad \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{7.4}$$

Alle in Bild 7.2 dargestellten Transformationen sind also nur unterschiedliche Ausprägungen einer $4 \cdot 4$-Matrix. Das ist ein wichtiger Aspekt für das Design von Grafikhardware, denn die sehr häufig benötigten und rechenintensiven Operationen mit $4 \cdot 4$-Matrizen können fest in Hardware „gegossen" und dadurch enorm beschleunigt werden. In modernen Grafikkarten mit „Transform- and Lighting"–Beschleunigung ist dies realisiert.

Alle Vertices einer Szene müssen alle Transformationsstufen durchlaufen, bevor sie gezeichnet werden können. Nun gibt es dafür aber zwei unterschiedliche Möglichkeiten: entweder werden alle Transformations–Matrizen zuerst miteinander multipliziert und die resultierende Gesamttransformations–Matrix dann mit den Vertices, oder die Vertices werden mit der Matrix jeder einzelnen Transformationsstufe der Reihe nach multipliziert. Bei der üblicherweise großen Anzahl von Vertices ist die erste Variante effektiver. Deshalb wird in OpenGL jede neu definierte Transformations–Matrix mit der „*aktuellen*" Matrix multipliziert, und das Ergebnis ist dann die neue aktuelle Gesamttransformations–Matrix.

OpenGL stellt folgende Befehle für allgemeine 4·4-Matrizen-Operationen zur Verfügung:

- `glLoadMatrixf(const GLfloat *M)`
  lädt die spezifizierte Matrix **M** als die aktuell anzuwendende Matrix. Das Argument `*M` ist ein Zeiger auf einen Vektor mit den 16 Komponenten der Matrix
  $\mathrm{M}[16] = \{m_{11}, m_{21}, m_{31}, m_{41}, m_{12}, m_{22}, m_{32}, m_{42},$
  $m_{13}, m_{23}, m_{33}, m_{43}, m_{14}, m_{24}, m_{34}, m_{44}\}$

- `glLoadIdentity(GLvoid)`
  lädt die Einheitsmatrix **I**, die für die Initialisierung des Matrizen–Speichers häufig benötigt wird

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.5}$$

- `glMultMatrixf(const GLfloat *M)`
  multipliziert die spezifizierte Matrix **M** mit der aktuell im Speicher befindlichen Matrix und schreibt das Ergebnis wieder in den Speicher. Das Argument `*M` ist genauso wie beim Befehl `glLoadMatrixf()` ein Zeiger auf einen Vektor mit 16 Komponenten
  $\mathrm{M}[16] = \{m_{11}, m_{21}, m_{31}, m_{41}, m_{12}, m_{22}, m_{32}, m_{42},$
  $m_{13}, m_{23}, m_{33}, m_{43}, m_{14}, m_{24}, m_{34}, m_{44}\}$

Es existieren auch noch die Varianten `glLoadMatrixd(const GLdouble *M)` und `glMultMatrixd(const GLdouble *M)` mit doppelter Genauigkeit.

7.4 Modell–Transformationen

3D-Objekte werden normalerweise in ihren lokalen Koordinatensystemen, den sogenannten „Objekt–Koordinaten", definiert und müssen erst durch eine Modell-Transformation an die richtige Stelle in der Szene, d.h. im Weltkoordinatensystem positioniert werden. Dazu dienen die Operationen Verschiebung (Translation), Drehung (Rotation) und Vergrößerung bzw. Verkleinerung (Skalierung). Eine andere Denkweise geht nicht von einem fixen Weltkoordinatensystem aus, in dem die Objekte verschoben werden, sondern benutzt die Vorstellung von einem lokalen Koordinatensystem, das fest an das Objekt gekoppelt ist. Verschoben wird jetzt nicht das Objekt im Koordinatensystem, sondern das Koordinatensystem mitsamt dem Objekt. Diese Denkweise, die auch im Folgenden verwendet wird, bietet den Vorteil, dass die Transformationsoperationen in der natürlichen Reihenfolge im Programm–Code erscheinen. Wichtig wird dies besonders bei kombinierten Translationen und Rotationen, denn hier ist die Reihenfolge der Operationen nicht vertauschbar.

7.4.1 Translation

Durch den OpenGL-Befehl `glTranslatef(GLfloat` T_x`, GLfloat` T_y`, GLfloat` T_z`)` wird der Ursprung des Koordinatensystems zum spezifizierten Punkt (T_x, T_y, T_z) verschoben (Bild 7.3).

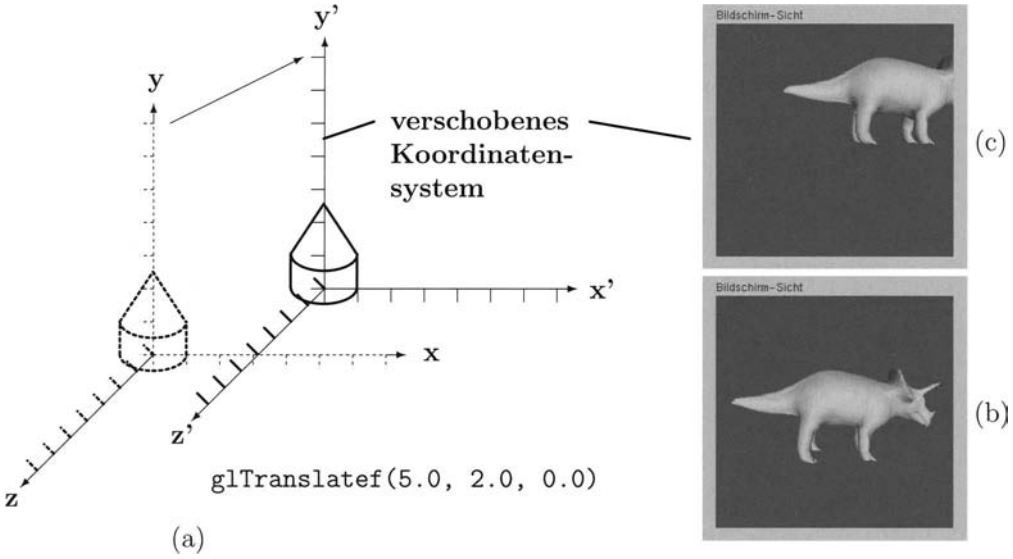


Bild 7.3: Translation des Koordinatensystems in OpenGL: (a) gestrichelt: nicht verschobenes Koordinatensystem (x, y, z), durchgezogen: verschobenes Koordinatensystem (x', y', z'). 3D-Objekt im Ursprung: Zylinder mit aufgestülptem Kegelmantel. (b) Bildschirm-Sicht eines nicht verschobenen Objekts (Triceratops). (c) Bildschirm-Sicht des mit `glTranslatef(5.0, 2.0, 0.0)` verschobenen Objekts.

In kartesischen Koordinaten wird eine Translation eines Vertex $\mathbf{v} = (x, y, z)$ um einen Richtungs–Vektor $(T_x, T_y, T_z)^T$ wie folgt geschrieben (siehe auch (7.3):

$$\begin{aligned} x' &= x + T_x \\ y' &= y + T_y \\ z' &= z + T_z \end{aligned}$$

und in homogenen Koordinaten mit der $4 \cdot 4$-Translations-Matrix $\mathbf{T}$:

$$\mathbf{v}' = \mathbf{T}\mathbf{v} \quad \Leftrightarrow \quad \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{7.6}$$

Durch den OpenGL-Befehl `glTranslatef(`T_x`,`T_y`,`T_z`)` wird also eine Translations-Matrix $\mathbf{T}$ definiert, die Matrix $\mathbf{T}$ wird mit der aktuell im Speicher befindlichen Matrix multipliziert und das Ergebnis wird wieder in den Speicher geschrieben. Dasselbe Ergebnis könnte man durch die explizite Spezifikation der 16 Komponenten der Translations–Matrix $\mathbf{T}$, gefolgt von dem Aufruf `glMultMatrixf(*T)`, erzielen. Allerdings wäre dies umständlicher in der Programmierung und langsamer in der Ausführung.

Eine alternative Form des Translations–Befehls mit doppelter Genauigkeit stellt `glTranslated(GLdouble` T_x`, GLdouble` T_y`, GLdouble` T_z`)` dar. Relevant wird die doppelt genaue Form des Befehls bei Anwendungen, mit Szenarien, die im Verhältnis zum sichtbaren Volumen sehr groß sind, wie z.B. bei Flug- oder Fahrsimulationen. Nehmen wir als Beispiel einen Schienenfahrsimulator, in dem eine Bahnstrecke von ca. 1000 *km* modelliert ist. Um einen Zug auch am entferntesten Punkt vom Ursprung des Koordinatensystems mit einer Genauigkeit von ca. 1 *cm* positionieren zu können, ist eine Genauigkeit von mindestens 8 Stellen hinter dem Komma erforderlich. Bei einem `GLfloat` mit 32 *bit* erreicht man je nach Implementierung allerdings nur eine Genauigkeit von 6 – 7 Stellen hinter dem Komma, so dass die Positionierung von Objekten im Bereich zwischen 0,1 – 1 Meter schwankt. Dies führt bei Bewegungen zu einem zufälligen Hin- und Herspringen von Objekten aufgrund der numerischen Rundungsfehler. Abhilfe kann in solchen Fällen durch die doppelt genauen Varianten der Modell–Transformationen geschaffen werden.

7.4.2 Rotation

Der OpenGL-Befehl `glRotatef(GLfloat` α`, GLfloat` R_x`, GLfloat` R_y`, GLfloat` R_z`)` dreht das Koordinatensystem um einen Winkel α bezüglich des Vektors $(R_x, R_y, R_z)^T$ (Bild 7.4). Der Winkel α wird dabei in der Einheit $[Grad]$ erwartet und der Vektor $(R_x, R_y, R_z)^T$ sollte der Einfachheit halber die Länge 1 besitzen. Positive Winkel bewirken eine Rotation im mathematisch positiven Sinn, d.h. gegen den Uhrzeigersinn. Mit Hilfe der „Rechte-Hand-Regel“ kann man sich die Drehrichtung relativ leicht klar machen: zeigt der Daumen der rechten Hand in Richtung des Vektors $(x, y, z)^T$, dann geben die restlichen vier halb

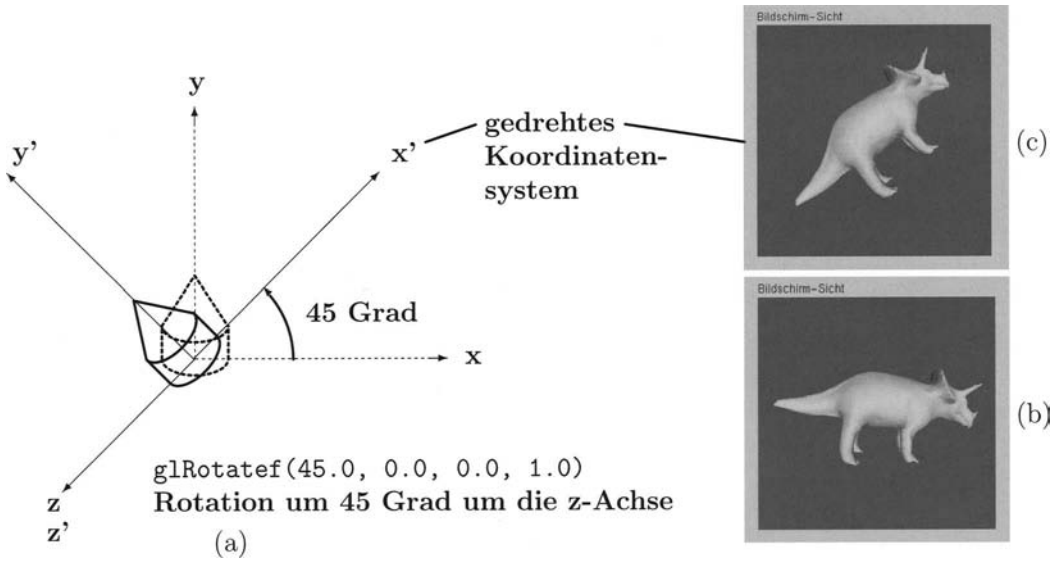


Bild 7.4: Rotation des Koordinatensystems in OpenGL: (a) gestrichelt: nicht gedrehtes Koordinatensystem (x, y, z); durchgezogen: um 45° bzgl. der z–Achse gedrehtes Koordinatensystem (x', y', z'). (b) Bildschirm-Sicht eines nicht gedrehten Objekts. (c) Bildschirm-Sicht des mit `glRotatef(45.0, 0.0, 0.0, 1.0)` gedrehten Objekts.

eingerollten Finger die Drehrichtung an. Die alternative Form des Befehls mit doppelter Genauigkeit lautet:

`glRotated(GLdouble` α`, GLdouble` R_x`, GLdouble` R_y`, GLdouble` R_z`)`.

In kartesischen Koordinaten wird eine Rotation eines Vertex $\mathbf{v} = (x, y, z)$ um einen Winkel α bzgl. eines Richtungs–Vektors $\mathbf{r} = (R_x, R_y, R_z)^T$ der Länge 1 wie folgt geschrieben:

$$\mathbf{v'} = \mathbf{Mv} \quad mit \quad \mathbf{M} = \begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{pmatrix}$$

$$\Leftrightarrow \mathbf{M} = \mathbf{r} \cdot \mathbf{r^T} + \cos\alpha \cdot (\mathbf{I} - \mathbf{r} \cdot \mathbf{r^T}) + \sin\alpha \cdot \begin{pmatrix} 0 & -R_z & R_y \\ R_z & 0 & -R_x \\ -R_y & R_x & 0 \end{pmatrix}$$

$$\Leftrightarrow \mathbf{M} = \begin{pmatrix} R_x^2 & R_xR_y & R_xR_z \\ R_xR_y & R_y^2 & R_yR_z \\ R_xR_z & R_yR_z & R_z^2 \end{pmatrix} + \cos\alpha \cdot \left[\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} R_x^2 & R_xR_y & R_xR_z \\ R_xR_y & R_y^2 & R_yR_z \\ R_xR_z & R_yR_z & R_z^2 \end{pmatrix} \right]$$

$$+ \sin\alpha \cdot \begin{pmatrix} 0 & -R_z & R_y \\ R_z & 0 & -R_x \\ -R_y & R_x & 0 \end{pmatrix} \tag{7.7}$$

und in homogenen Koordinaten mit der $4 \cdot 4$-Rotations-Matrix $\mathbf{R}$, in der die Matrix $\mathbf{M}$ aus (7.7) enthalten ist:

$$\mathbf{v}' = \mathbf{R}\mathbf{v} \quad \Leftrightarrow \quad \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{7.8}$$

In den Spezialfällen, in denen die Rotationachse eine der drei Koordinatenachsen x, y oder z ist, vereinfacht sich die jeweilige $4 \cdot 4$-Rotations-Matrix $\mathbf{R}$ erheblich:

Rotation um die x–Achse:

$$\texttt{glRotatef}(\alpha,1,0,0) \quad \Leftrightarrow \quad \mathbf{R^x} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.9}$$

Rotation um die y–Achse:

$$\texttt{glRotatef}(\alpha,0,1,0) \quad \Leftrightarrow \quad \mathbf{R^y} = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.10}$$

Rotation um die z–Achse:

$$\texttt{glRotatef}(\alpha,0,0,1) \quad \Leftrightarrow \quad \mathbf{R^z} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{7.11}$$

Durch den OpenGL-Befehl `glRotatef(`α, R_x, R_y, R_z`)` wird also eine Rotations–Matrix $\mathbf{R}$ definiert, die Matrix $\mathbf{R}$ wird mit der aktuell im Speicher befindlichen Matrix multipliziert und das Ergebnis wird wieder in den Speicher geschrieben. Dasselbe Ergebnis könnte auch hier wieder mit extrem großem Aufwand mit dem Aufruf `glMultMatrixf(*R)` erzielt werden.

7.4.3 Skalierung

Durch den OpenGL-Befehl `glScalef(GLfloat` S_x`, GLfloat` S_y`, GLfloat` S_z`)` wird das Koordinatensystem in der jeweiligen Achse gestreckt, wenn der Skalierungsfaktor betragsmäßig größer als 1 ist oder gestaucht, wenn der Faktor betragsmäßig kleiner als 1 ist (siehe folgende Tabelle und Bild 7.5). Negative Streckungsfaktoren bewirken zusätzlich eine Spiegelung an der jeweiligen Achse. Der Wert 0 ist als Streckungsfaktor unzulässig, da die Dimension verschwinden würde.

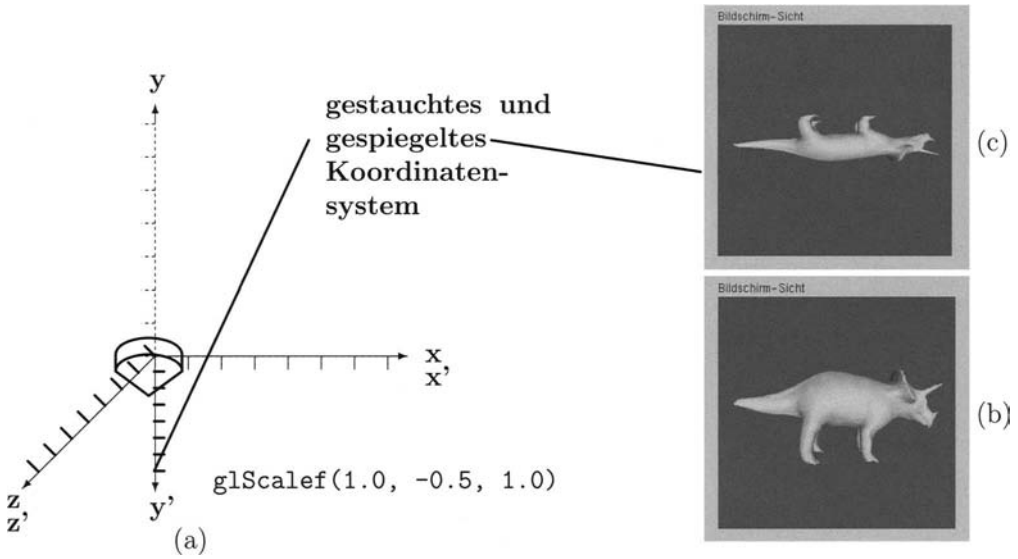


Bild 7.5: Skalierung des Koordinatensystems in OpenGL: (a) gestrichelt: original Koordinatensystem (x, y, z); durchgezogen: bzgl. der y–Achse um den Faktor $|s| = 0.5$ gestauchtes und gespiegeltes Koordinatensystem (x', y', z'). (b) Bildschirm-Sicht eines nicht skalierten Objekts (Triceratops). (c) Bildschirm-Sicht des mit `glScalef(1.0, -0.5, 1.0)` skalierten Objekts.

Skalierungsfaktor (s)	**Effekt**
$\|s\| > 1.0$	Streckung / Dimensionen vergrößern
$\|s\| = 1.0$	Dimensionen unverändert
$0.0 < \|s\| < 1.0$	Stauchung / Dimensionen verkleinern
$s = 0.0$	unzulässiger Wert

Eine alternative Form des Befehls mit doppelter Genauigkeit lautet:
`glScaled(GLdouble` S_x`, GLdouble` S_y`, GLdouble` S_z`)`.

Damit man einen Vertex $\mathbf{v} = (x, y, z)$ in jeder Achse getrennt um einen Faktor S_x, S_y oder S_z skalieren kann, muss man komponentenweise multiplizieren:

$$x' = S_x \cdot x$$
$$y' = S_y \cdot y$$
$$z' = S_z \cdot z$$

Eine allgemeine Skalierung in homogenen Koordinaten mit der $4 \cdot 4$-Skalierungs–Matrix $\mathbf{S}$ lautet:

$$\mathbf{v}' = \mathbf{S}\mathbf{v} \quad \Leftrightarrow \quad \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \tag{7.12}$$

Durch den OpenGL-Befehl `glScalef(`S_x`,`S_y`,`S_z`)` wird also eine Skalierungs–Matrix $\mathbf{S}$ definiert, die Matrix $\mathbf{S}$ wird mit der aktuell im Speicher befindlichen Matrix multipliziert und das Ergebnis wird wieder in den Speicher geschrieben. Dasselbe Ergebnis könnte auch hier wieder mit größerem Aufwand mit dem Aufruf `glMultMatrixf(*S)` erzielt werden.

7.4.4 Reihenfolge der Transformationen

Die endgültige Position und Lage eines Objektes in der Szene hängt normalerweise sehr stark von der Reihenfolge der Transformationen ab. In Bild 7.6 ist dies am Beispiel von Rotation und Translation in der Denkweise eines festen Weltkoordinatensystems gezeigt. In der Ausgangsposition befindet sich das 3D-Objekt (ein Zylinder mit übergestülptem Kegel) im Ursprung des Koordinatensystems. Da das Objekt in Bild 7.6-a zuerst eine Rotation um einen Winkel von 45 Grad bzgl. der z–Achse erfährt und anschließend eine Translation entlang der x–Achse, erscheint es letztendlich auf der x–Achse. In Bild 7.6-b dagegen wird zuerst die Translation entlang der x–Achse durchgeführt und dann die Rotation um einen Winkel von 45 Grad bzgl. der z–Achse, so dass das Objekt in diesem Fall an einer ganz anderen Stelle erscheint, und zwar auf der Winkelhalbierenden zwischen der x– und der y–Achse. Der Unterschied in der endgültigen Position des Objekts beruht darauf, dass die Transformationen nicht unabhängig voneinander sind: während in Bild 7.6-a das Objekt quasi mit einem Hebel der Länge 0 um den Ursprung gedreht wird, erfolgt in Bild 7.6-b die Drehung mit einem Hebel der Länge 5 (die Hebellänge entspricht der Translation).

An dieser Stelle bietet es sich an, noch einmal die unterschiedlichen Denkweisen bei Transformationen zu erläutern. Betrachtet man, wie in Bild 7.6 gezeigt, ein festes Weltkoordinatensystem, muss im linken Teilbild (a) zuerst die Drehung und dann die Translation erfolgen (im rechten Teilbild (b) umgekehrt). Betrachtet man dagegen, wie in Bild 7.7 gezeigt, ein lokales Koordinatensystem, das fest an das Objekt gebunden ist, muss im linken Teilbild (a) zuerst die Translation und dann die Drehung erfolgen, also genau in der entgegengesetzten Reihenfolge wie in der Denkweise des festen Weltkoordinatensystems (im rechten Teilbild (b) umgekehrt).

Im Programm-Code müssen die Transformationen in der umgekehrten Reihenfolge erscheinen wie in der Denkweise des festen Weltkoordinatensystems. Denn eine Verkettung von mehreren Transformationen wird durch eine Multiplikation der zugehörigen Matrizen realisiert, indem die neu hinzukommende Matrix von rechts auf die vorhandenen Matrizen aufmultipliziert wird. Zur Erläuterung wird ein Stück Programm-Code betrachtet, der die Situation in den linken Teilbildern von Bild 7.6 und Bild 7.7 beschreibt:

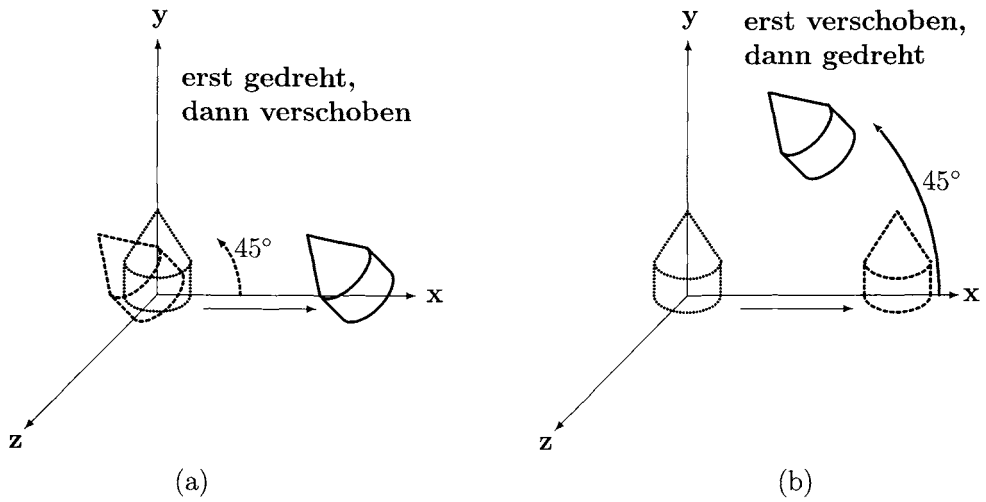


Bild 7.6: Die Reihenfolge der Transformationen in der Denkweise eines festen Weltkoordinatensystems: (a) gepunktet: original Objekt, gestrichelt: erst gedreht bzgl. der z–Achse, durchgezogen: dann verschoben bzgl. der x–Achse. (b) gepunktet: original Objekt, gestrichelt: erst verschoben bzgl. der x–Achse, durchgezogen: dann gedreht bzgl. der z–Achse.

```
glLoadIdentity();                    /* I = Einheitsmatrix */
glTranslatef();                  /* T = Translations-Matrix */
glRotatef();                       /* R = Rotations-Matrix */
glBegin(GL_TRIANGLES);
glVertex3fv(v);                     /* Vertex-Spezifikation */
...
glEnd();
```

Zur Initialisierung wird zunächst die $4 \cdot 4$-Einheitsmatrix $\mathbf{I}$ geladen. Dann wird die Translations-Matrix $\mathbf{T}$ von rechts auf die Einheitsmatrix multipliziert, d.h.: $\mathbf{I} \cdot \mathbf{T} = \mathbf{T}$. Danach wird die Rotations-Matrix $\mathbf{R}$ von rechts auf $\mathbf{T}$ multipliziert und das ergibt: $\mathbf{T} \cdot \mathbf{R}$. Letztendlich werden die Vertices $\mathbf{v}$ von rechts auf die Matrix $\mathbf{TR}$ multipliziert: $\mathbf{TRv}$. Die Vertex-Transformation lautet also: $\mathbf{T}(\mathbf{Rv})$, d.h. der Vertex wird zuerst mit der Rotations-Matrix $\mathbf{R}$ multipliziert und das Ergebnis $\mathbf{v}' = \mathbf{Rv}$ wird danach mit der Translations-Matrix $\mathbf{T}$ multipliziert: $\mathbf{v}'' = \mathbf{T} \cdot \mathbf{v}' = \mathbf{T}(\mathbf{Rv})$. Die Reihenfolge, in der die Matrizen abgearbeitet werden, ist also genau umgekehrt wie die Reihenfolge, in der sie im Programm-Code spezifiziert sind. In der Denkweise des festen Weltkoordinatensystems muss man quasi den Programm-Code rückwärts entwickeln. Deshalb ist es bequemer, im lokalen Koordinatensystem zu arbeiten, das mit dem Objekt mitbewegt wird, denn dort ist die Reihenfolge der Transformationen in der Denkweise und im Programm-Code die selbe.

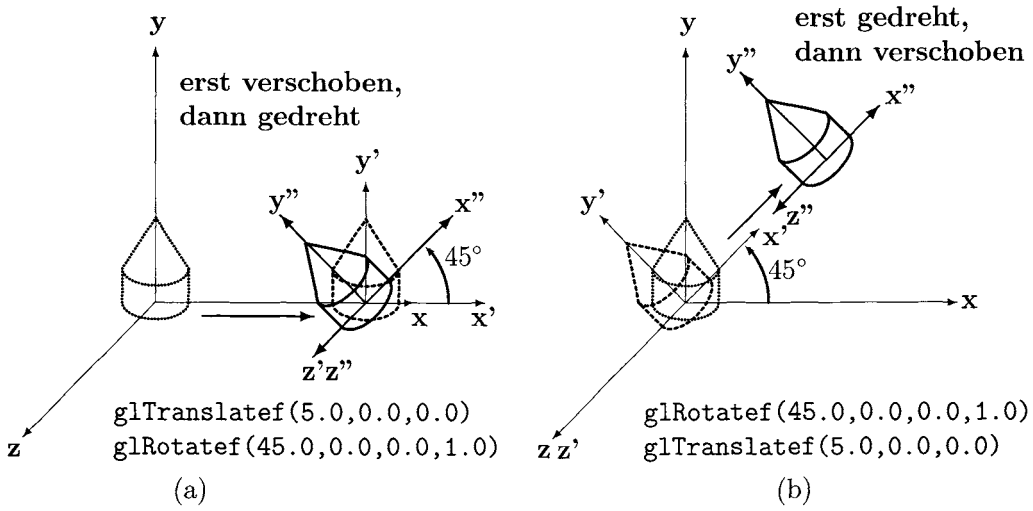


Bild 7.7: Die Reihenfolge der Transformationen in der Denkweise eines lokalen Koordinatensystems, das mit dem Objekt verschoben wird: (a) gepunktet: original Objekt, gestrichelt: erst verschoben bzgl. der x–Achse, durchgezogen: dann gedreht bzgl. der z'–Achse . (b) gepunktet: original Objekt, gestrichelt: erst gedreht bzgl. der z–Achse, durchgezogen: dann verschoben bzgl. der x'–Achse.

7.5 Augenpunkt–Transformationen

Die Augenpunkt–Transformation (*viewing transformation*) ändert die Position und die Blickrichtung des Augenpunkts, von dem aus die Szene betrachtet wird. Mit anderen Worten heißt das, man positioniert die Kamera und richtet sie auf die Objekte aus, die fotografiert werden sollen. Standardmäßig befindet sich der Augenpunkt im Koordinatenursprung (0,0,0), die Blickrichtung läuft entlang der negativen z–Achse und die y–Achse weist nach oben. Sollen aus diesem Augenpunkt heraus Objekte betrachtet werden, die sich ebenfalls um den Ursprung gruppieren, muss entweder die Kamera entlang der positiven z–Achse zurückbewegt werden, oder die Objekte müssen in die entgegengesetzte Richtung bewegt werden. Folglich ist eine Augenpunkt–Transformation, bei der der Betrachter in Richtung der positiven z–Achse bewegt wird, nicht zu unterscheiden von einer Modell–Transformation, bei der die Objekte in Richtung der negativen z–Achse bewegt werden. Letztlich sind also Modell– und Augenpunkt–Transformationen vollkommen äquivalent zueinander, weshalb sie in einem sogenannten „*Modelview–Matrix–Stack*“ zusammengefasst werden. Die Augenpunkts–Transformationen müssen immer vor allen anderen Transformationen im Programm-Code aufgerufen werden, damit alle Objekte in der Szene in gleicher Weise verschoben oder um den gleichen Bezugspunkt gedreht werden.

7.6 Projektions–Transformationen

Nach den Modell– und Augenpunkts–Transformationen sind alle Vertices an der gewünschten Position im 3-dimensionalen Raum. Um nun ein Abbild der 3-dimensionalen Szene auf einen 2-dimensionalen Bildschirm zu bekommen, wird eine Projektions–Transformation eingesetzt. In der darstellenden Geometrie wurde eine Vielzahl unterschiedlicher projektiver Abbildungen entwickelt, die in OpenGL durch die Definition eigener Transformations–Matrizen realisierbar sind (eine ausführliche Diskussion des Themas findet man in [Fole96] und [Enca96]). In der praktischen Anwendung sind jedoch zwei Projektions–Transformationen besonders relevant: die orthografische und die perspektivische. Für diese beiden Transformationen stellt OpenGL zur Verringerung des Programmieraufwands eigene Befehle zur Verfügung, auf die im Folgenden näher eingegangen wird.

Zunächst allerdings noch ein Hinweis: durch eine klassische Projektions-Transformation werden alle Punkte im 3-dimensionalen Raum auf eine 2-dimensionale Fläche abgebildet (in OpenGL standardmäßig die $x - y$–Ebene), d.h. eine Dimension würde wegfallen (in OpenGL die z–Achse). Da man aber die z–Werte später noch benötigt, modifiziert man die Projektions-Transformation so, dass die z–Werte erhalten bleiben und gleichzeitig die $x - y$–Koordinaten wie bei einer klassischen Projektion transformiert werden. Damit hat man zwei Fliegen mit einer Klappe geschlagen, denn einerseits erhält man die korrekt projizierten $x - y$–Koordinaten für die Bildschirm-Darstellung, und andererseits kann man die normierten z–Werte für die Verdeckungsrechnung (Kapitel 8) und die Nebelberechnung (Kapitel 11) nutzen.

7.6.1 Orthografische Projektion (Parallel–Projektion)

In der orthografischen Projektion werden alle Objekte durch parallele Strahlen auf die Projektionsfläche abgebildet (die deshalb auch „Parallel–Projektion“ genannt wird). Dadurch bleiben die Größen und Winkel aller Objekte erhalten, unabhängig davon wie weit sie von der Projektionsfläche entfernt sind (Bild 7.8). Aus diesem Grund wird die orthografische Projektion vor allem bei CAD–Anwendungen und technischen Zeichnungen zur Erzeugung von Vorderansicht, Seitenansicht und Draufsicht verwendet.

Durch die Orthografische Projektion wird ein Volumen in Form eines Kubus' aus der virtuellen Szene herausgeschnitten, denn nur dieser Teil der Szene ist später am Bildschirm sichtbar. Alle Vertices außerhalb des Kubus' werden nach der Projektions–Transformation weggeschnitten (*clipping*). Die sechs Begrenzungsflächen des Kubus' werden deshalb auch als „*clipping planes*“ bezeichnet.

Die Transformations–Matrix der orthografischen Projektion ist die Einheits-Matrix $\mathbf{I}$, denn alle Größen sollen erhalten bleiben. Der OpenGL-Befehl, durch den die Transformations-Matrix der orthografischen Projektion spezifiziert wird, lautet:

```
glOrtho (   GLdouble left, GLdouble right,
            GLdouble bottom, GLdouble top,
            GLdouble near, GLdouble far );
```

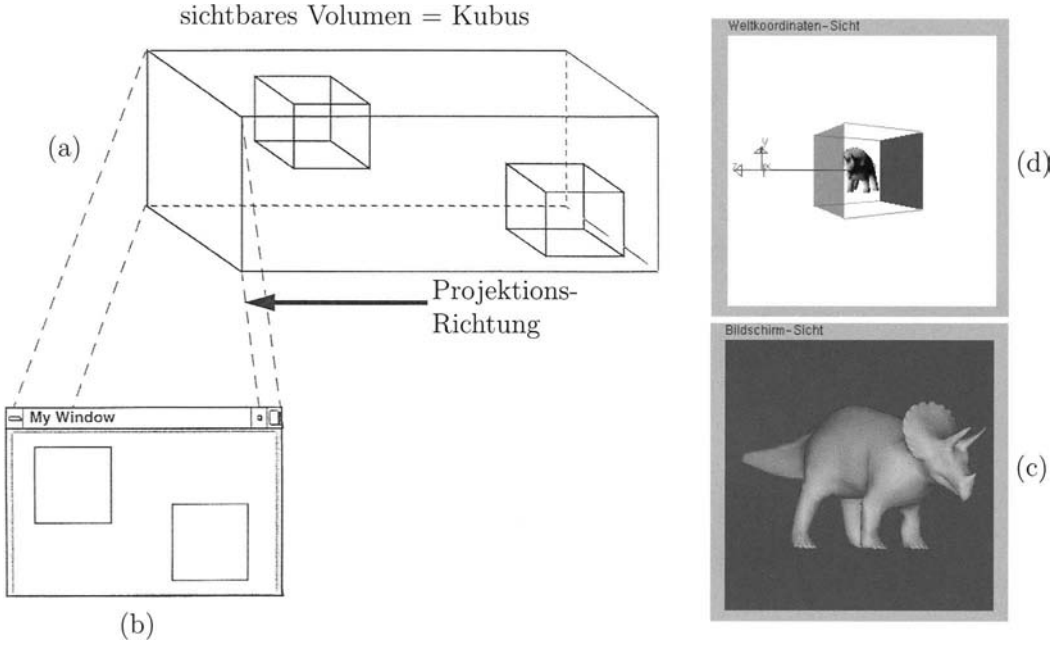


Bild 7.8: Orthografische Projektion: (a) das sichtbare Volumen (*viewing volume*) ist ein Kubus (indem hier zwei weitere Kuben als Objekte enthalten sind). (b) die Bildschirm-Sicht nach der orthografischen Projektion: die beiden kubus-förmigen Objekte erscheinen gleich groß, obwohl das Objekt rechts unten weiter vom Augenpunkt entfernt ist als das Objekt links oben. (c) Blick auf das Weltkoordinaten-System, welches das sichtbare Volumen (Kubus) enthält. (d) Bildschirm-Sicht nach der orthografischen Projektion.

Die Bedeutung der sechs Argumente des Befehls dürfte intuitiv klar sein: linke, rechte, untere, obere, vordere und hintere Begrenzung des sichtbaren Volumens. Mit dem OpenGL-Befehl `glOrtho()` wird allerdings nicht nur die Projektions–Matrix $\mathbf{I}$ definiert, sondern die Kombination aus Projektions–Matrix $\mathbf{I}$ und Normierungs–Matrix $\mathbf{N}$, d.h. $\mathbf{I} \cdot \mathbf{N} = \mathbf{N}$ (siehe auch 7.27).

7.6.2 Perspektivische Projektion

In der perspektivischen Projektion werden alle Objekte innerhalb des sichtbaren Volumens durch konvergierende Strahlen abgebildet, die im Augenpunkt zusammenlaufen (weshalb sie auch „Zentral–Projektion“ heißt). Objekte, die näher am Augenpunkt sind, erscheinen deshalb auf der Projektionsfläche größer als entferntere Objekte (Bild 7.9).

Die perspektivische Projektion entspricht unserer natürlichen Wahrnehmung, denn die Bildentstehung auf der Netzhaut unserer Augen oder auch auf einem fotografischen Film in einer Kamera wird durch diese Art der Projektion beschrieben. Erst durch die perspektivische Projektion entsteht ein realistischer räumlicher Eindruck einer Szene, und deshalb

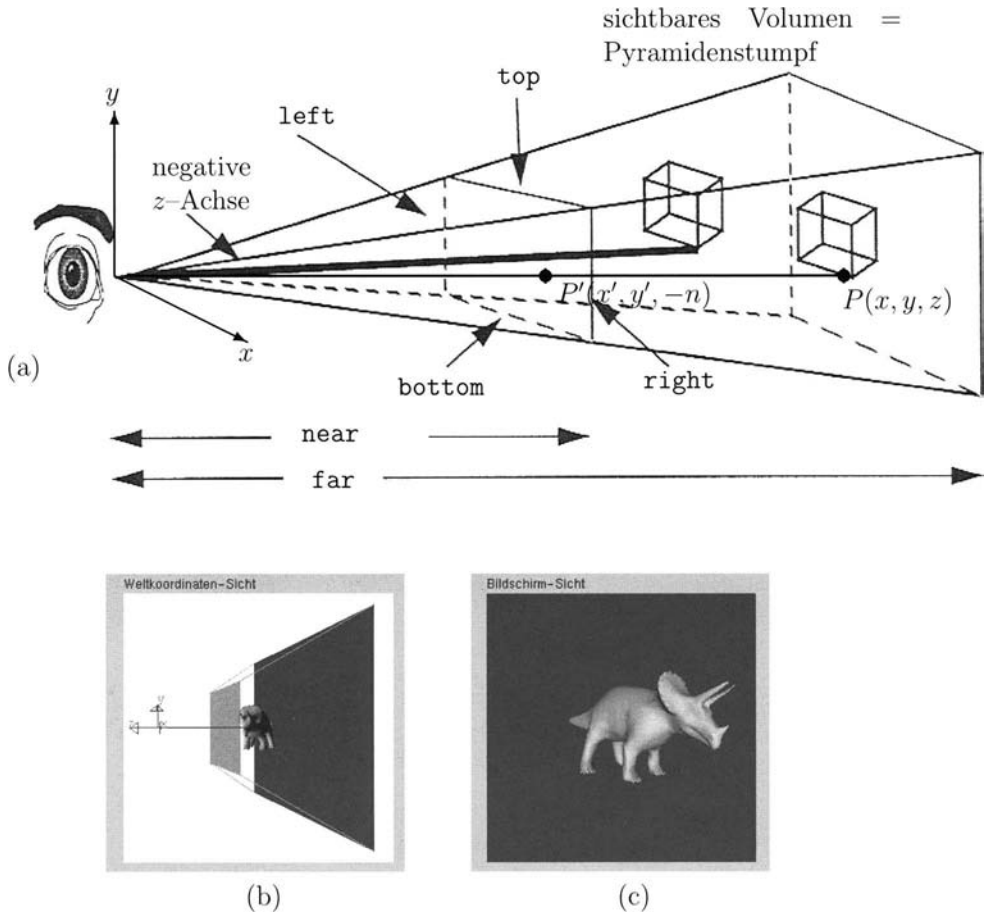


Bild 7.9: Perspektivische Projektion: (a) das sichtbare Volumen (*viewing volume*) ist ein Pyramidenstumpf, der quasi auf der Seite liegt. Alle Objekte außerhalb des sichtbaren Volumens werden weggeschnitten (*clipping*). Die sechs Begrenzungsebenen des Pyramidenstumpfs werden deshalb auch als „*clipping planes*" bezeichnet. Der Boden des Pyramidenstumpfs ist die „*far clipping plane*", in der Spitze der Pyramide sitzt der Augenpunkt, durch die „*near clipping plane*" wird die Spitze der Pyramide weggeschnitten, und die restlichen vier „*clipping planes*" werden durch die schrägen Seiten des Pyramidenstumpfs gebildet. (b) Blick auf das Weltkoordinaten-System, welches das sichtbare Volumen (Pyramidenstumpf) enthält. (c) Bildschirm-Sicht nach der perspektivischen Projektion.

wird sie auch am weitaus häufigsten in der 3D–Computergrafik angewendet.

Durch die perspektivische Projektion wird ein sichtbares Volumen (*viewing volume*) in Form eines auf der Seite liegenden Pyramidenstumpfs aus der virtuellen Szene herausgeschnitten. Alle Objekte außerhalb des Pyramidenstumpfs werden weggeschnitten (*clipping*). Die sechs Begrenzungsebenen des Pyramidenstumpfs werden deshalb auch als „*clipping planes*" bezeichnet. Der Boden des Pyramidenstumpfs ist die hintere Grenzfläche

(„*far clipping plane*"), in der Spitze der Pyramide sitzt der Augenpunkt, durch die vordere Grenzfläche („*near clipping plane*") wird die Spitze der Pyramide weggeschnitten, und die restlichen vier „*clipping planes*" werden durch die schrägen Seiten des Pyramidenstumpfs gebildet.

Der OpenGL-Befehl, durch den die Transformations–Matrix der perspektivischen Projektion (und der Normierungs–Matrix **N**, siehe (7.27)) spezifiziert wird, lautet:

```
glFrustum ( GLdouble left, GLdouble right,
            GLdouble bottom, GLdouble top,
            GLdouble near, GLdouble far );
```

Die Bedeutung der sechs Argumente des Befehls ist ähnlich wie bei `glOrtho()`: die ersten vier Argumente legen die linke, rechte, untere und obere Begrenzung der „*near clipping plane*" fest, die sich im Abstand „`near`" vom Augenpunkt entfernt befindet. Das sechste Argument „`far`" legt den Abstand der „*far clipping plane*" vom Augenpunkt fest. Die Ausmaße der „*far clipping plane*" werden indirekt durch das Verhältnis der sechs Argumente untereinander bestimmt. Die linke Begrenzung der „*far clipping plane*" z.B. berechnet sich aus `(left/near)*far`, die weiteren Begrenzungen entsprechend. Eine Veränderung des Argumentes „`near`" ändert folglich die Form des sichtbaren Volumens massiv, wie in Bild 7.10 zu sehen ist.

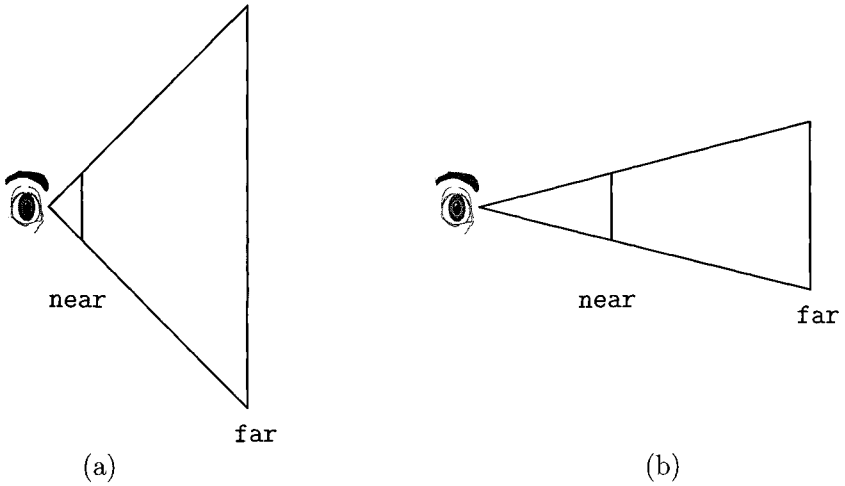


Bild 7.10: Der Einfluss des Arguments „`near`" bei der Perspektivischen Projektion mit dem OpenGL-Befehl `glFrustum()`: (a) ein kleiner Wert von „`near`" weitet das sichtbare Volumen stark auf (Weitwinkel-Effekt), (b) ein großer Wert von „`near`" engt das sichtbare Volumen stark ein (Tele-Effekt).

In der Praxis ist der Befehl `glFrustum()` manchmal etwas umständlich. Denn bei typischen Anwendungen, wie z.B. der Sicht aus einer Fahrerkabine, lauten die Anforderungen oft wie folgt: Blickwinkel vertikal $\alpha = 40°$, Blickwinkel horizontal $\beta = 60°$, Sichtweite $> 1000m$. Eine untere Grenze für die Sichtweite („*near clipping plane*") wird meist nicht

vorgegeben, sondern muss vom Software–Entwickler anwendungsspezifisch festgelegt werden. Ein typischer Wert wäre der Abstand zwischen der Sitzposition des Fahrers in der Kabine und der vorderen Begrenzung des Fahrzeugs, also z.B. $2m$. Damit liegen die letzten beiden Argumente des Befehls `glFrustum()` fest: near = 2, far = 1000. Die ersten vier Argumente müssen über trigonometrische Formeln berechnet werden:

$$\texttt{left} = \texttt{-near} \cdot \tan\frac{\beta}{2} \tag{7.13}$$

$$\texttt{right} = \texttt{near} \cdot \tan\frac{\beta}{2} \tag{7.14}$$

$$\texttt{bottom} = \texttt{-near} \cdot \tan\frac{\alpha}{2} \tag{7.15}$$

$$\texttt{top} = \texttt{near} \cdot \tan\frac{\alpha}{2} \tag{7.16}$$

Falls der Wert von „`near`“ im Nachhinein geändert werden soll, müssen die ersten vier Argumente von `glFrustum()` parallel dazu geändert werden, nachdem sie mit Hilfe der Gleichungen (7.13ff) nochmals berechnet wurden. Falls nur der Wert von „`near`“ alleine geändert wird, ändern sich auch die Blickwinkel. Weil die Handhabung des `glFrustum()`–Befehls oft nicht besonders bequem ist, wird im Rahmen der OpenGL *Utility Library* (GLU) der Befehl:

```
gluPerspective (GLdouble α, GLdouble aspect,
                GLdouble near, GLdouble far );
```

zur Verfügung gestellt. Das erste Argument „α“ ist der vertikale Blickwinkel mit einem Wertebereich von 0° bis 180°, das zweite Argument „`aspect`“ ist das Verhältnis zwischen horizontalem und vertikalem Blickwinkel (β/α). Die Bedeutung der letzten beiden Argumente ist die gleiche wie beim `glFrustum()`-Befehl. Beim obigen Anwendungsbeispiel könnten mit dem `gluPerspective()`–Befehl folglich die Spezifikationswerte ohne weitere Umrechnung direkt als Argumente eingesetzt werden, und auch eine nachträgliche Änderung der „*near clipping plane*“ hätte keine Auswirkungen auf die Sichtwinkel. Allerdings muss man bei Verwendung des `gluPerspective()`–Befehls auch Einschränkungen hinnehmen: die Blickwinkel müssen symmetrisch nach links und rechts bzw. nach oben und unten sein (wie in der obigen Anwendung angenommen). Falls asymmetrische Blickwinkel gefordert sind (z.B. 10° nach unten und 30° nach oben), ist man letztlich doch wieder auf den `glFrustum()`–Befehl angewiesen.

Die perspektivische Projektion lässt sich mit Hilfe des Strahlensatzes verstehen. Bild 7.9 zeigt die Verhältnisse: der Punkt $P(x, y, z)$ im sichtbaren Volumen wird in den Punkt $P'(x', y', -n)$ auf der Projektionsfläche (der *near clipping plane*) abgebildet. Die *near clipping plane* ist vom Augenpunkt entlang der negativen z–Achse um die Distanz n = `near` (die *far clipping plane* um die Distanz f = `far`) verschoben und liegt parallel zur $x-y$–Ebene. Die Anwendung des Strahlensatzes liefert für die x- und y-Komponenten der Punkte P und P':

$$\frac{x'}{-n} = \frac{x}{z} \quad \Leftrightarrow \quad x' = -\frac{n}{z} \cdot x \tag{7.17}$$

$$\frac{y'}{-n} = \frac{y}{z} \quad \Leftrightarrow \quad y' = -\frac{n}{z} \cdot y \tag{7.18}$$

Wie im Folgenden gezeigt, erfüllt die Transformations–Matrix der perspektivischen Projektion $\mathbf{P}$ in OpenGL die Gleichungen (7.17) und (7.18):

$$\mathbf{v}' = \mathbf{P}\mathbf{v} \quad \Leftrightarrow \quad \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1+\frac{f}{n} & f \\ 0 & 0 & -\frac{1}{n} & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x \\ y \\ \left(1+\frac{f}{n}\right) z + fw \\ -\frac{z}{n} \end{pmatrix} \tag{7.19}$$

Zur Umrechnung der homogenen Koordinaten in (7.17) in 3-dimensionale euklidische Koordinaten wird durch den inversen Streckungsfaktor w (der hier $-z/n$ ist) geteilt:

$$\begin{pmatrix} x'_E \\ y'_E \\ z'_E \end{pmatrix} = \begin{pmatrix} -\frac{n}{z} \cdot x \\ -\frac{n}{z} \cdot y \\ -\frac{n}{z} \cdot fw - (f+n) \end{pmatrix} \tag{7.20}$$

Mit (7.20) ist gezeigt, dass die Projektions-Matrix $\mathbf{P}$ die Gleichungen (7.17) und (7.18) erfüllt. Zum besseren Verständnis seien noch zwei Spezialfälle betrachtet: Erstens, Punkte, die bereits auf der *near clipping plane* liegen, d.h. einsetzen von $(z = -n, w = 1)$ in (7.20):

$$\begin{pmatrix} x'_E \\ y'_E \\ z'_E \end{pmatrix} = \begin{pmatrix} x \\ y \\ -n \end{pmatrix} \tag{7.21}$$

d.h. Punkte, die auf der Projektionsfläche liegen, bleiben unverändert. Zweitens, Punkte, die auf der *far clipping plane* liegen, d.h. einsetzen von $(z = -f, w = 1)$ in (7.20):

$$\begin{pmatrix} x'_E \\ y'_E \\ z'_E \end{pmatrix} = \begin{pmatrix} \frac{n}{f} \cdot x \\ \frac{n}{f} \cdot y \\ -f \end{pmatrix} \tag{7.22}$$

d.h. bei Punkten, die auf der hinteren Grenzfläche des sichtbaren Volumens liegen, werden die x- und y–Koordinaten um einen Faktor n/f verkleinert und die z–Koordinate wird wieder auf $z = -f$ abgebildet. Alle Punkte zwischen der *near* und der *far clipping plane* werden also wieder in das Intervall zwischen $-n$ und $-f$ abgebildet, allerdings nicht linear. Tendenziell werden die Punkte durch die perspektivische Projektion näher zur *near clipping plane* hin abgebildet. Das bewirkt, dass die Auflösung der z–Koordinate nahe am Augenpunkt größer ist als weiter entfernt. Folglich nimmt die Genauigkeit der Verdeckungsrechnung mit der Nähe zum Augenpunkt zu (Kapitel 8).

Abschließend sei noch darauf hingewiesen, dass die „*far clipping plane*“ so nah wie möglich am Augenpunkt sein sollte, die „*near clipping plane*“ dagegen so weit wie möglich vom Augenpunkt entfernt. Denn einerseits bleibt damit das sichtbare Volumen und folglich die Zahl der zu zeichnenden Objekte klein, was die Rendering–Geschwindigkeit erhöht, und andererseits wird durch einen kleineren Abstand zwischen „*near*“ und „*far*“ die Genauigkeit der Verdeckungsrechnung größer.

7.6.3 Normierung

Nach der Projektions–Transformation befinden sich alle Vertices, die sich innerhalb des sichtbaren Volumens befanden, bzgl. der x- und y-Koordinaten innerhalb der vorderen Grenzfläche dieses Volumens, d.h. auf der „*near clipping plane*". Die Ausmaße der „*near clipping plane*" können vom OpenGL-Benutzer beliebig gewählt werden, wie im vorigen Abschnitt beschrieben. Letztendlich müssen aber sowohl riesige als auch winzige „*near clipping planes*" auf eine festgelegte Anzahl von Pixeln in einem Bildschirmfenster gebracht werden. Deshalb wird vorher als Zwischenschritt noch eine Normierung aller Vertices durchgeführt. Die x–Komponente wird durch die halbe Ausdehnung der „*near clipping plane*" in x–Richtung geteilt und das Zentrum des Wertebereichs von x wird in den Ursprung verschoben. Entsprechend wird mit der y– und der $-z$–Komponente der Vertices verfahren. Mathematisch ausgedrückt heißt das:

$$x' = \frac{2}{r-l} \cdot x - \frac{r+l}{r-l} \cdot w \tag{7.23}$$

$$y' = \frac{2}{t-b} \cdot y - \frac{t+b}{t-b} \cdot w \tag{7.24}$$

$$z' = \frac{-2}{f-n} \cdot z - \frac{f+n}{f-n} \cdot w \tag{7.25}$$

$$w' = w \tag{7.26}$$

oder in Matrix–Schreibweise:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \quad \Leftrightarrow \quad \mathbf{v}' = \mathbf{N}\mathbf{v} \tag{7.27}$$

Dabei sind die Größen in der Normierungs–Matrix $\mathbf{N}$ die sechs Argumente der beiden OpenGL–Projektions–Transformationen `glOrtho()` bzw. `glFrustum()`:

$l =$ `left`; $r =$ `right`; $b =$ `bottom`; $t =$ `top`; $n =$ `near`; $f =$ `far`.

Die durch (7.27) gegebene Normierungs–Transformation überführt die Vertices in das Normierte Koordinatensystem, so dass alle Werte von x, y und z zwischen $-w$ und $+w$ (in homogenen Koordinaten) liegen. Im zweiten Schritt der Normierungstransformation werden die homogenen Koordinaten durch den inversen Streckungsfaktor w geteilt, so dass die (euklidischen) Koordinaten $(x, y, z, 1)^T$ zwischen -1 und +1 liegen, und zwar unabhängig von der Größe des sichtbaren Volumens. Durch den Aufruf der OpenGL-Befehle `glOrtho()` und `glFrustum()` wird sowohl die Projektions– als auch die Normierungs–Transformation durchgeführt.

7.7 Viewport-Transformation

Letzten Endes soll die computergenerierte Szene auf einen bestimmten Ausschnitt des Bildschirms gezeichnet werden. Dieser Bildschirmausschnitt, auch *Viewport* genannt, hat eine Ausdehnung in x– und y–Richtung, die in Pixeln festgelegt wird. In der letzten Transformationsstufe, der Viewport-Transformation, werden die normierten Vertex-Koordinaten auf die gewählte Viewport-Größe, d.h. in Window–Koordinaten umgerechnet (Bild 7.11).

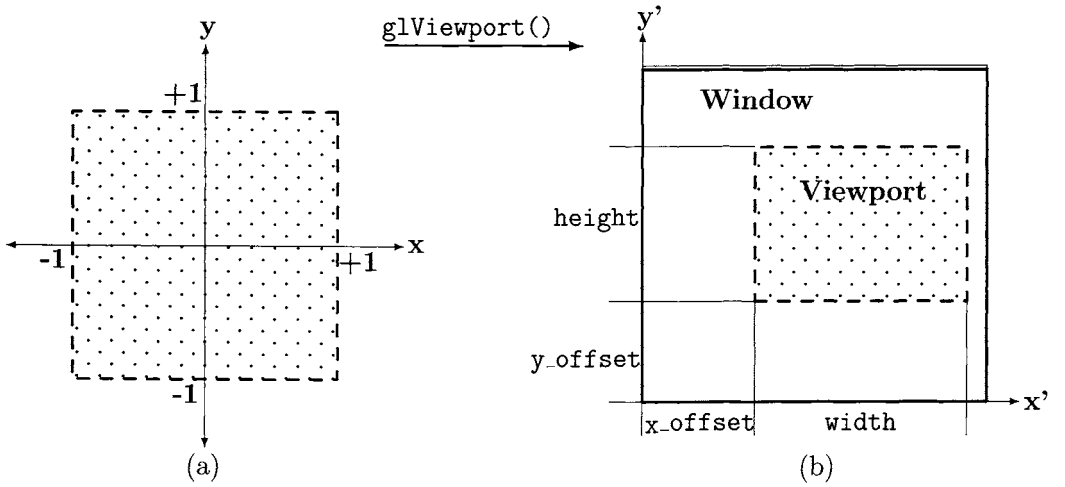


Bild 7.11: Abbildung des sichtbaren Volumens in einen *Viewport*: (a) Wertebereich der normierten Koordinaten des sichtbaren Volumens. (b) Wertebereich der Bildschirm–Koordinaten des Viewports nach der Viewport-Transformation. Der Ursprung der Bildschirm–Koordinaten liegt in der linken unteren Ecke des Windows.

Die Umrechnung läuft in folgenden Schritten ab: die x–Koordinaten werden mit der halben Windowbreite „$width/2$“ skaliert, die y–Koordinaten mit der halben Windowhöhe „$height/2$“, so dass der Wertebereich der x– und y–Vertex-Koordinaten in den Intervallen $[-width/2,+width/2]$ und $[-height/2,+height/2]$ liegt; anschließend wird der Viewport um eine halbe Windowbreite in x–Richtung sowie um eine halbe Windowhöhe in y–Richtung verschoben, und zusätzlich kann der Viewport noch um einen einstellbaren Offset „x_offset“ in x–Richtung oder „y_offset“ in y–Richtung verschoben werden, so dass der Wertebereich der x– und y–Vertex-Koordinaten in den Intervallen $[x\_offset, x\_offset + width]$ bzw. $[y\_offset, y\_offset + height]$ liegt. In Formeln ausgedrückt lautet die Viewport-Transformation (bei $w = 1$):

$$x' = \frac{width}{2} \cdot x + \left(x\_offset + \frac{width}{2} \right) \tag{7.28}$$

$$y' = \frac{height}{2} \cdot y + \left(y\_offset + \frac{height}{2} \right) \tag{7.29}$$

oder in Matrizen–Schreibweise:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \frac{width}{2} & 0 & 0 & x\_offset + \frac{width}{2} \\ 0 & \frac{height}{2} & 0 & y\_offset + \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \quad \Leftrightarrow \quad \mathbf{v}' = \mathbf{V}\mathbf{v} \qquad (7.30)$$

Der OpenGL-Befehl, durch den die Viewport–Transformations–Matrix $\mathbf{V}$ spezifiziert wird, lautet:

```
glViewport (    GLint x_offset, GLint y_offset,
                GLsizei width, GLsizei height );
```

Durch die ersten beiden Argumente `x_offset` und `y_offset` wird die linke untere Ecke des Viewports innerhalb des Bildschirmfensters festgelegt. Durch die letzten beiden Argumente `width` und `height` wird die Größe des Viewports in x– und y–Richtung spezifiziert. Alle Größen sind in Pixeln anzugeben. Falls der Viewport nicht explizit durch den Befehl `glViewport()` verändert wird, ist er deckungsgleich mit dem Window. Ein Viewport muss also nicht grundsätzlich genau so groß sein wie ein Window, sondern er kann durchaus auch kleiner sein. Anders ausgedrückt kann ein Window auch mehrere Viewports enthalten, wie in Bild 7.12-b gezeigt.

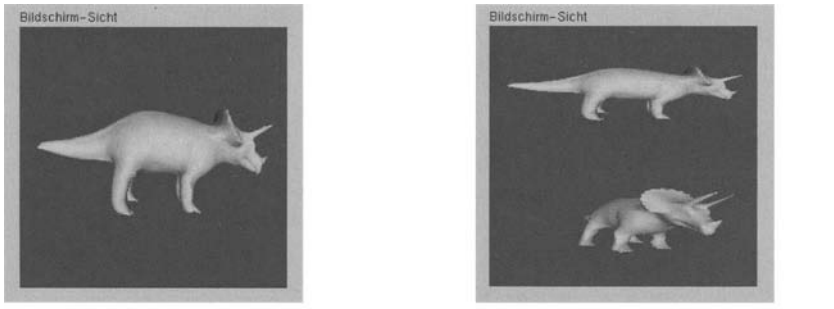


Bild 7.12: Abbildung des sichtbaren Volumens in einen *Viewport*: (a) Window und Viewport sind gleich groß. Der Viewport hat das gleiche Aspektverhältnis (x/y) wie das sichtbare Volumen. Deshalb erscheint das Objekt hier nicht verzerrt. (b) das Window enthält zwei Viewports übereinander. Die beiden Viewports haben ein anderes Aspektverhältnis als das sichtbare Volumen, weshalb hier die Objekte verzerrt werden.

Damit die Objekte nicht verzerrt am Bildschirm dargestellt werden, muss das Aspektverhältnis des Viewports das gleiche sein wie das des sichtbaren Volumens. Falls die beiden Aspektverhältnisse unterschiedlich sind, werden die dargestellten Objekte in x– oder y–Richtung gestaucht oder gedehnt, wie in Bild 7.12-b zu sehen ist. Schließlich ist noch eine wesentliche Einschränkung zu beachten: Viewports in OpenGL sind immer rechteckig.

7.8 Matrizen-Stapel

Jeder Vertex $\mathbf{v}$ einer Szene – und deren Anzahl kann in die Millionen gehen – muss alle dargestellten Transformationsstufen durchlaufen: die Modell- und Augenpunkttransformationen (Translation, Rotation, Skalierung), die Projektionstransformation, die Normierung und die Viewport–Transformation, d.h.

$$\mathbf{v}' = (\mathbf{V} \cdot \mathbf{N} \cdot \mathbf{P} \cdot (\mathbf{S} \cdot \mathbf{R} \cdot \mathbf{T})^*) \cdot \mathbf{v} \tag{7.31}$$

wobei $(\mathbf{S} \cdot \mathbf{R} \cdot \mathbf{T})^*$ für eine beliebige Kombination von Skalierungen, Rotationen und Translationen steht. Um die Effizienz der Berechnungen zu steigern, werden die einzelnen Transformations–Matrizen erst zu einer Gesamttransformations–Matrix aufmultipliziert, bevor die Vertices dann mit dieser Gesamttransformations–Matrix multipliziert werden. Denn es gilt:

$$\mathbf{v}' = (\mathbf{V}\,(\mathbf{N}\,(\mathbf{P}\,(\mathbf{S}\,(\mathbf{R}\,(\mathbf{T} \cdot \mathbf{v})))))) = (\mathbf{V} \cdot \mathbf{N} \cdot \mathbf{P} \cdot \mathbf{S} \cdot \mathbf{R} \cdot \mathbf{T}) \cdot \mathbf{v} \tag{7.32}$$

d.h. um einen Vertex zu transformieren, gibt es zwei Möglichkeiten: entweder man multipliziert den Vertex $\mathbf{v}$ zuerst mit der Matrix $\mathbf{T}$, anschließend das Ergebnis mit der Matrix $\mathbf{R}$, usw. bis zur Matrix $\mathbf{V}$, oder man multipliziert erst alle Matrizen zu einer Gesamttransformations–Matrix auf und danach den Vertex $\mathbf{v}$ mit dieser Matrix. Da in der interaktiven 3D-Computergrafik meist eine große Anzahl von Vertices mit der gleichen Gesamttransformations–Matrix multipliziert wird, ist es sehr viel effektiver, erst "ein"-mal die Matrizen miteinander zu multiplizieren und danach die "n" Vertices mit der Gesamttransformations–Matrix. Es werden also nicht die Vertices nach jeder Transformationsstufe zwischengespeichert, sondern die jeweils aktuelle Transformations–Matrix.

Beim Aufbau von komplexeren Objekten, die hierarchisch aus einfacheren Teilen zusammengesetzt sind, benötigt man aber nicht nur eine einzige Gesamttransformations–Matrix, sondern für jede Hierarchiestufe eine eigene. Ein einfaches Auto z.B. besteht aus einem Chassis, an dem vier Räder mit je 5 Schrauben befestigt sind. Anstatt nun 20 Schrauben und vier Räder an der jeweiligen Position des Chassis zu modellieren, erzeugt man nur je ein Modell einer Schraube und eines Rades in einem lokalen Koordinatensystem und verwendet das jeweilige Modell mit unterschiedlichen Transformationen entsprechend oft. Dabei ist es hilfreich, die Matrizen einzelner Transformationstufen zwischenzuspeichern. Denn um die 5 Schrauben eines Rades zu zeichnen, geht man z.B. vom Mittelpunkt des Rades eine kleine Strecke nach rechts (`glTranslatef`), zeichnet die Schraube, geht wieder zurück, dreht um 72° (`glRotatef`), geht wieder die kleine Strecke vom Mittelpunkt weg, zeichnet die nächste Schraube, geht wieder zurück, usw., bis alle 5 Schrauben gezeichnet sind. „Gehe zurück" wird jetzt nicht durch einen erneuten Aufruf von `glTranslatef` realisiert, sondern einfach dadurch, dass man die zuvor zwischengespeicherte Matrix wieder als aktuelle Matrix benützt. Aus diesem Grund wurden in OpenGL sogenannte „*Matrizen–Stapel*" (*matrix stacks*) eingeführt, ein „*Modelview–Matrizen Stapel*", der bis zu 32 verschiedene $4 \cdot 4$–Matrizen für die Modell- und Augenpunkttransformationen speichern kann, und ein „*Projektions–Matrizen Stapel*", der zwei verschiedene $4 \cdot 4$–Matrizen für die Projektionstransformationen speichern kann. Um festzulegen,

dass eine Gesamttransformations–Matrix auf den Modelview–Matrizen Stapel gelegt werden soll, wird der Befehl `glMatrixMode(GL_MODELVIEW)` ausgeführt, und um festzulegen, dass sie auf den Projektions–Matrizen Stapel gelegt werden soll, wird `glMatrixMode()` mit dem Argument „`GL_PROJECTION`“ aufgerufen. Die oberste Matrix auf dem Stapel ist die aktuelle Gesamttransformations–Matrix. Soll die oberste Matrix für spätere Zwecke abgespeichert werden, wird mit dem Befehl `glPushMatrix()` eine Kopie angefertigt und diese Kopie wird als neues Element auf den Matrizen Stapel gelegt. Der Stapel wird also durch `glPushMatrix()` um ein Element höher. Jetzt kann die oberste Matrix mit weiteren Transformations–Matrizen multipliziert werden, so dass eine neue Gesamttransformations–Matrix entsteht. Um die vorher abgespeicherte Matrix wieder verwenden zu können, wird das oberste Element des Matrizen Stapels entfernt, denn dadurch wird die vorher an zweiter Stelle des Stapels befindliche Matrix wieder zur obersten (Bild 7.13). Der zugehörige OpenGL-Befehl `glPopMatrix()` erniedrigt also den Matrizen Stapel um ein Element.

Ein Beispiel für zusammengesetzte Transformationen

Zum Abschluss dieses Kapitels wird als praktisches Beispiel für zusammengesetzte Transformationen das oben erwähnte einfache Auto modelliert, das aus einem Chassis besteht, an dem vier Räder mit je 5 Schrauben befestigt sind. Vorausgesetzt wird, dass Routinen existieren, in denen je ein Chassis, ein Rad und eine Schraube modelliert ist. Die Stufe der Einzelteile ist die erste Hierarchiestufe der Modellierung. In der zweiten Hierachiestufe werden die 5 Schrauben an das Rad geheftet. Die Routine `draw_tire_and_bolts()` bewerkstelligt dies: zunächst wird das Rad (`draw_tire()`) mit der aktuellen Transformations–Matrix gezeichnet, dann wird die aktuelle Transformations–Matrix mit dem Aufruf `glPushMatrix()` im Stapel gespeichert, anschließend werden die Modelltransformationen (Rotation um 0 Grad bzgl. des Mittelpunkts des Rads und Translation) durchgeführt, die Schraube (`draw_bolt()`) wird mit der modifizierten Transformations–Matrix gezeichnet, und abschließend wird mit dem Aufruf `glPopMatrix()` die modifizierte Transformations–Matrix vom Stapel gelöscht, so dass die vorher gespeicherte Transformations–Matrix, bei der das Koordinatensystem im Mittelpunkt des Rads ist, wieder oben auf dem Stapel liegt; die beschriebene Aktion wird beginnend bei dem Aufruf `glPushMatrix()` noch vier Mal mit den Rotationswinkeln $1, 2, 3, 4 \cdot 72°$ wiederholt.

```
draw_tire_and_bolts() {      // Routine, die 1 Rad mit 5 Schrauben zeichnet
    GLint i;

    draw_tire();             // Routine, die ein Rad zeichnet
    for(i=0;i<5;i++) {
        glPushMatrix();
            glRotatef(72.0*i, 0.0, 0.0, 1.0);
            glTranslatef(0.03, 0.0, 0.0);
            draw_bolt();     // Routine, die eine Schraube zeichnet
        glPopMatrix();
    }
}
```

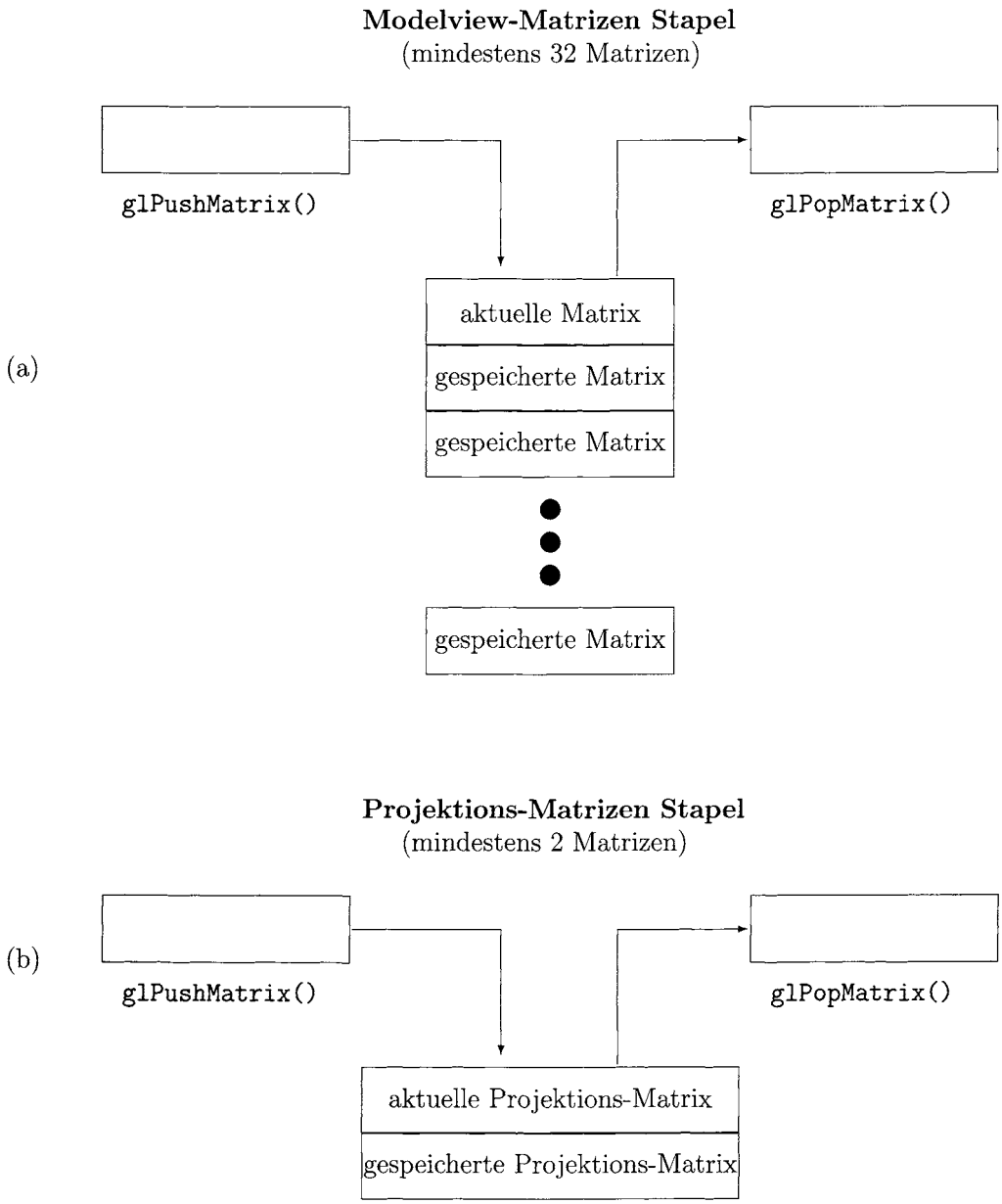


Bild 7.13: Matrizen Stapel in OpenGL: (a) Modelview–Matrizen Stapel zur Speicherung von 32 Matrizen aus der Klasse der Modell- und Augenpunkttransformationen. (b) Projektions–Matrizen Stapel zur Speicherung von zwei Matrizen aus der Klasse der Projektionstransformationen.

In der dritten Hierachiestufe werden die vier Räder an das Chassis „geschraubt". Die Routine `draw_chassis_tires_bolts()` erledigt das: zunächst wird das Chassis (`draw_chassis()`) mit der aktuellen Transformations–Matrix gezeichnet, dann wird die aktuelle Transformations–Matrix mit dem Aufruf `glPushMatrix()` im Stapel gespeichert, anschließend wird die Translation vom Mittelpunkt des Chassis zur Radaufhängung durchgeführt, das Rad inclusive der 5 Befestigungsschrauben wird mit der modifizierten Transformations–Matrix gezeichnet (`draw_tire_and_bolts()`), und abschließend wird mit dem Aufruf `glPopMatrix()` die modifizierte Transformations–Matrix vom Stapel gelöscht, so dass die vorher gespeicherte Transformations–Matrix, bei der das Koordinatensystem im Mittelpunkt des Chassis ist, wieder oben auf dem Stapel zu liegen kommt. Die beschriebene Aktion wird beginnend bei dem Aufruf `glPushMatrix()` für die anderen drei Räder mit unterschiedlichen Translationen wiederholt.

```
draw_chassis_tires_bolts() {            // Routine, die das Auto zeichnet
    GLfloat i,j;

    draw_chassis();                     // zeichne das Chassis
    for(i=-1.0;i<2.0;i+=2.0) {
        for(j=-1.0;j<2.0;i+=2.0) {
            glPushMatrix();
                glTranslatef(0.5*i, -0.2, -0.3*j);
                draw_tire_and_bolts();  // zeichne 1 Rad mit 5 Schrauben
            glPopMatrix();
        }
    }
}
```

In der höchsten Hierachiestufe wird das gesamte Fahrzeug mit der Augenpunkt– Transformation an die gewünschte Position im sichtbaren Volumen gebracht und dort gezeichnet. Alle bisher genannten Transformationen waren Modell- oder Augenpunkttransformationen und deshalb werden sie im Modelview–Matrizen Stapel abgelegt. Außerdem wird hier noch die Matrix der Projektions–Transformation (inclusive Normierung) auf den Projektions–Matrizen Stapel gelegt und die Viewport-Transformation festgelegt. In Bild 7.14 werden die Hierarchiestufen dargestellt, in denen das Fahrzeug zusammengebaut wird.

```
glViewport(0, 0, width, height);  // Viewport-Transformation

glMatrixMode(GL_PROJECTION);      // Projektions-Matrizen Stapel aktiviert
glLoadIdentity();                 // Initialisierung mit Einheitsmatrix
gluPerspective(60.0,width/height,0.5,8.0);    // Projektionstransformation

glMatrixMode(gl_MODELVIEW);       // Modelview-Matrizen Stapel aktiviert
glLoadIdentity();                 // Initialisierung mit Einheitsmatrix
glTranslatef(0.0, 0.0, -4.0);     // Augenpunkts-Transformation

draw_chassis_tires_bolts();       // zeichne das Auto (Modell-Transform.)
```

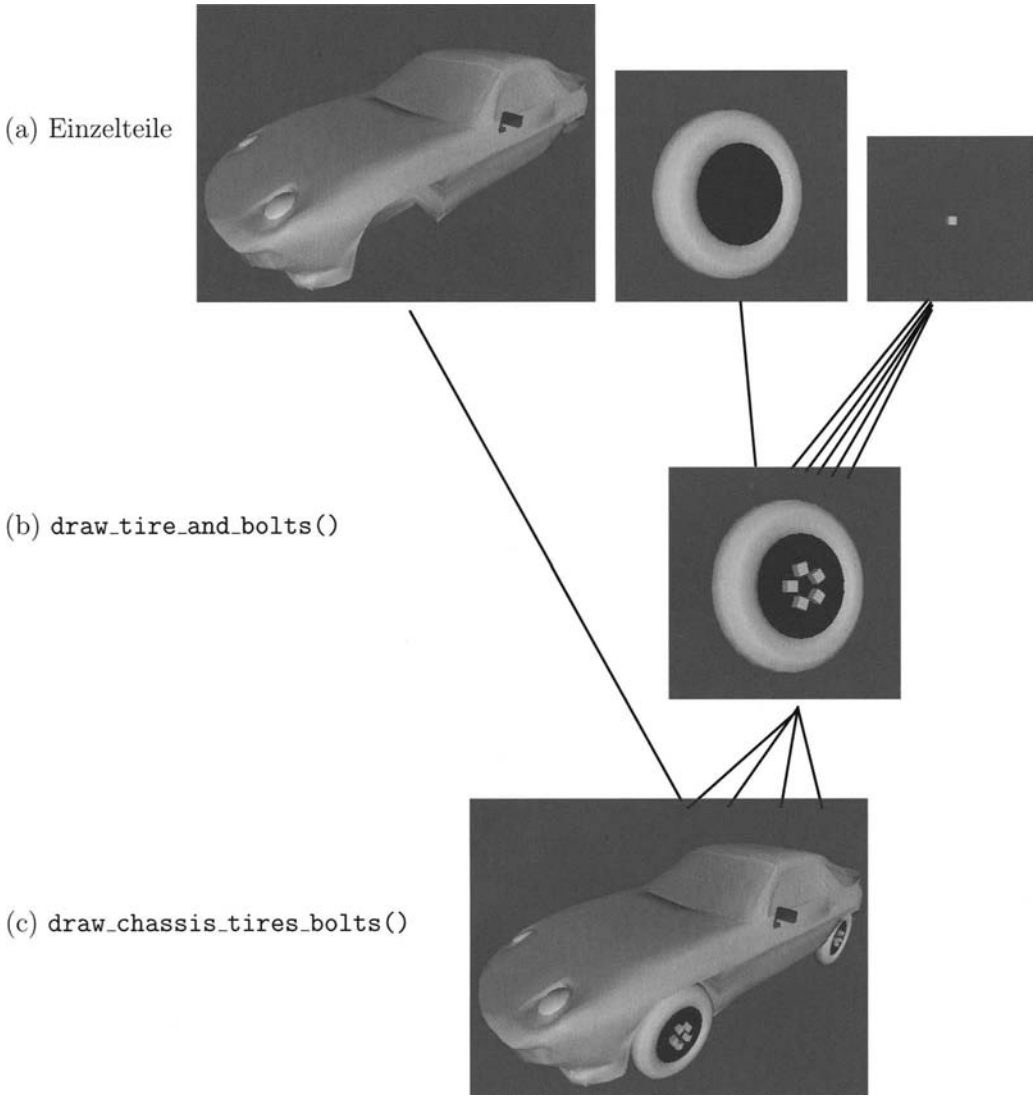


Bild 7.14: Hierarchischer Aufbau eines komplexeren Modells aus einfacheren Teilen: (a) die Einzelteile Chassis, Rad und Schraube (b) die 5 Schrauben sind an der richtigen Stelle des Rads angebracht (c) die 4 Räder inclusive Schrauben sind am Chassis befestigt

Kapitel 8

Verdeckung

Ein wichtiger Aspekt bei der räumlichen Wahrnehmung ist die Verdeckung von Objekten im Hintergrund durch (undurchsichtige) Objekte im Vordergrund. Die gegenseitige Verdeckung von Objekten gibt uns einen verlässlichen Hinweis zur Entfernung der Objekte vom Augenpunkt. Denn ein Objekt A, das vom Augenpunkt weiter entfernt ist als ein Objekt B, kann dieses niemals verdecken. Oder anders ausgedrückt, falls ein Objekt B ein Objekt A verdeckt, können wir in unserem Weltbild vollkommen sicher darauf schließen, dass Objekt B näher am Augenpunkt sein muss, als Objekt A. Sollte dieses Grundprinzip unserer Wahrnehmung in computergenerierten Bildern einer 3-dimensionalen Szene verletzt sein, wird der Beobachter verwirrt und die Bilder werden als unrealistisch verworfen.

In der Computergrafik werden die Objekte aber einfach in der Reihenfolge gezeichnet, wie sie im Programm definiert wurden (Bild 8.1). Wenn ein Pixel von zwei Objekten beschrieben wird, erhält das Pixel die Farbe des zuletzt gezeichneten Objekts, und zwar unabhängig davon, ob dieses Objekt die kürzeste Entfernung zum Augenpunkt aufweist. Für eine korrekte Darstellung 3-dimensionaler Szenen muss folglich ein Algorithmus gefunden

Bild 8.1: Probleme des Maler-Algorithmus' bei sich durchdringenden Objekten: (a) der Triceratops wird zuletzt gezeichnet und überdeckt daher eigentlich sichtbare Teile des Quaders. (b) der Quader wird zuletzt gezeichnet und überdeckt daher eigentlich sichtbare Teile des Triceratops. (c) korrekte Darstellung mit Hilfe des z-Buffer Algorithmus.

werden, der das Verdeckungsproblem löst. Eine Möglichkeit, um das Verdeckungsproblem anzugehen, ist der sogenannte „*Maler–Algorithmus*", der in den folgenden zwei Schritten abläuft:

a) Sortiere alle Objekte in Bezug auf ihren Abstand zum Augenpunkt.

b) Zeichne alle Objekte in ihrer neuen Reihenfolge, beginnend mit dem entferntesten.

Dieser z.B. von Landschaftsmalern verwendete Algorithmus funktioniert in den meisten Fällen ganz gut. Allerdings hat der Maler–Algorithmus zwei entscheidende Nachteile:

- Der Rechenaufwand bei Sortieralgorithmen steigt nichtlinear mit der Anzahl der Objekte. Da in der 3D-Computergrafik die relevanten Objekte die Polygone sind, die erst in sehr großer Zahl eine Szene realistisch nachbilden, würde der Rechenaufwand für das Sortieren ins Unerträgliche steigen. Außerdem müsste bei Objekt- oder Augenpunktsbewegungen vor jedem generierten Bild neu sortiert werden.

- Falls sich Objekte gegenseitig durchdringen, scheitert der Maler–Algorithmus vollkommen (siehe Bild 8.1). Um in solchen Fällen weiter zu kommen müssten zusätzliche komplexe Algorithmen eingesetzt werden, die zunächst einmal detektieren, welche Objekte sich durchdringen und welche Objektteile noch sichtbar sind. Als Konsequenz würde der Rechenaufwand noch einmal drastisch steigen.

8.1 Der z-Buffer Algorithmus

Aufgrund der geschilderten Nachteile des Maler–Algorithmus wird in der interaktiven 3D-Computergrafik ein anderer Algorithmus zur Verdeckungsrechnung eingesetzt: der von Cutmull [Catm74] entwickelte, sogenannte „*z-Buffer Algorithmus*" (Bild 8.2).

Die Grundidee des z-Buffer Algorithmus besteht darin, durch zusätzliche Hardware die Tiefeninformation (d.h. den z-Wert) für jedes Pixel zu speichern (alternativ auch „*Depth Buffer* Algorithmus" genannt). Falls ein Pixel durch ein Objekt beschrieben wird, muss vorher geprüft werden, ob es näher am Augenpunkt liegt (d.h. einen kleineren z-Wert hat), als das vorher gezeichnete Objekt (d.h. der abgespeicherte z-Wert). Falls ja, werden die Farbwerte und der z-Wert für das Pixel mit den neuen Werten überschrieben, andernfalls bleiben die alten Werte erhalten.

Der Pseudo–Code des z-Buffer Algorithmus ist in **A8.1** zusammengefasst. Die einzelnen Teilaspekte werden in den folgenden Abschnitten untersucht.

A8.1: Pseudo–Code des z-Buffer Algorithmus.

Voraussetzungen und Bemerkungen:

◇ zusätzlicher Speicherplatz für die z-Werte wird zur Verfügung gestellt.

Algorithmus:

(a) Initialisiere den z-Buffer auf den Maximalwert für jedes Pixel des Bildschirmfensters.

(b) Für alle Objekte (Polygone), die gezeichnet werden müssen:

(ba) Für alle Pixel eines Objekts, die gezeichnet werden müssen:

(baa) Berechne den Abstand vom Augenpunkt zu dem Objekt für das Pixel.

(bab) Vergleiche den berechneten Abstand mit dem gespeicherten Wert im z-Buffer.

(bac) falls (Abstand < gespeicherter Wert):
trage die neuen Farbwerte in den Farbspeicher (engl. color buffer) und den neuen z-Wert in den z-Buffer für dieses Pixel ein. (Das Objekt ist näher).

(bad) andernfalls:
ändere nichts. (Das Objekt ist verdeckt).

Ende des Algorithmus

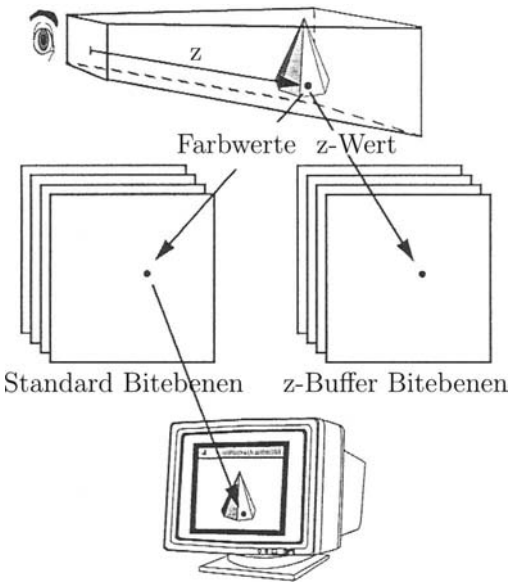


Bild 8.2: Die Lösung des Verdeckungsproblems: der z-Buffer Algorithmus. Für jedes Pixel werden die Farbwerte und die z-Werte gespeichert. Die Werte werden nur überschrieben, wenn der z-Wert des neuen Objekts kleiner ist als der gespeicherte.

Das Prinzip des z-Buffer Algorithmus wird in Bild 8.3 noch einmal grafisch verdeutlicht.

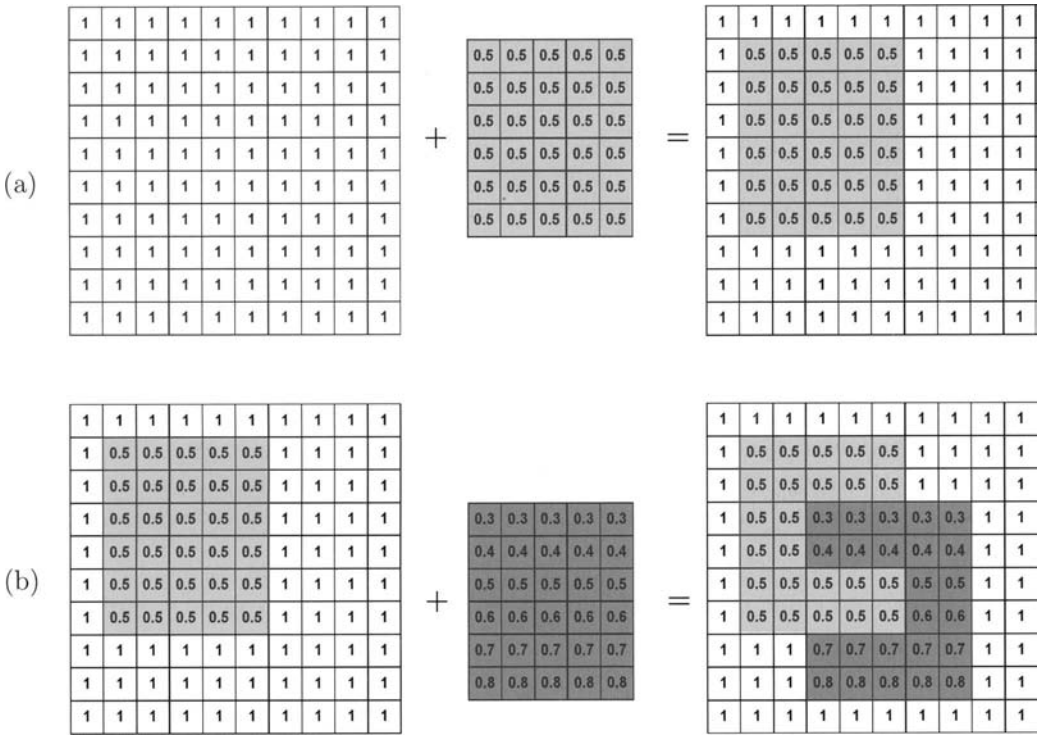


Bild 8.3: Das Prinzip des z-Buffers anhand eines 10x10 Pixel großen Bildes: die Zahl in jedem Kästchen repräsentiert die räumliche Tiefe, d.h. den z-Wert des Pixels, die Graustufe der Pixel repräsentiert die Farbwerte. (a) auf einen „sauberen" Bildspeicher (initialisiert mit maximalem z-Wert 1) wird ein Polygon mit konstantem z-Wert von 0.5 addiert (d.h. das Polygon steht senkrecht auf der Sichtlinie) (b) Addition eines weiteren Polygons, das gegenüber der Sichtlinie geneigt ist und das erste Polygon schneidet

Der z-Buffer Algorithmus bietet eine Reihe von Vorteilen:

- es ist kein aufwändiges Sortieren mehr nötig
- die Verdeckung von sich durchdringenden Objekten wird mit Pixel–Genauigkeit korrekt berechnet
- die Berechnung des z-Werts eines Pixels ist einfach und schnell. Denn die z-Werte aller Vertices sind nach den in Kapitel 7 dargestellten Transformationen schon vorhanden und die Berechnung z-Werte für jedes Pixel kann wegen der Einschränkung auf planare Polygone durch eine lineare Interpolation der Vertex-z-Werte erledigt werden.

Auf der anderen Seite gibt es beim z-Buffer Algorithmus durchaus auch einige Probleme, deren man sich bewusst sein sollte:

- z-Buffer Flimmern : aufgrund der begrenzten Auflösung des z-Buffers (meist 16 bit oder 32 bit) kommt es insbesondere bei weiter entfernten Flächen, die parallel und nah beieinander liegen, zu einer abwechselnden Darstellung. Ein typisches Beispiel dafür sind weiße Fahrbahnmarkierungen, die knapp (z.B. $1cm$) über der schwarzen Straßenfläche schweben. Aus dem in die Ferne gerichteten Blickwinkel eines bewegten Fahrzeugs erscheint, je nach numerischen Rundungsfehlern, einmal die weiße Markierung und einmal die schwarze Straße oben, so dass ein schwarz-weißes Flimmern entsteht, das erst nach Annäherung an die Fahrbahnmarkierung verschwindet. Zur Abhilfe dieses negativen Effekts bieten sich zwei unterschiedliche Methoden an: erstens die Verwendung einer Fahrbahntextur, die die Markierung enthält (diese Lösung kostet allerdings sehr viel Texturspeicherplatz), oder zweitens die Schaffung einer Datenstruktur, in der jeder Fläche ein Paritätsbit zugewiesen wird, das bestimmt, ob die Fläche unten oder oben liegt (für diese Lösung benötigt man einen Algorithmus, der das Paritätsbit auswertet und den z-Buffer Algorithmus ergänzt). Abmildern kann man das z-Buffer Flimmern mit Hilfe des Subpixel-Anti-Aliasing (Kapitel 10).

- Transparente Oberflächen werden vom z-Buffer Algorithmus nicht korrekt berücksichtigt (Die Ursache und eine Abhilfe für dieses Problem wird in Kapitel 9 dargestellt).

8.2 Die Implementierung des z-Buffer Algorithmus

Im Folgenden werden nun die konkreten OpenGL-Befehle erläutert, die den in A8.1 beschriebenen Schritten des z-Buffer Algorithmus entsprechen.

- *Voraussetzungen*: Aktivierung des OpenGL-Zustands und Konfiguration des Speicherplatzes für die z-Werte (*Depth Buffer*).
  Wie die meisten Eigenschaften in OpenGL muss auch die Verdeckungsrechnung zunächst einmal aktiviert werden mit dem Befehl `glEnable(GL_DEPTH_TEST)`. Da in manchen Anwendungen die Verdeckungsrechnung durchaus unerwünscht oder zumindest überflüssig ist, kann man sie selbstverständlich auch wieder ausschalten, und zwar mit dem Befehl `glDisable(GL_DEPTH_TEST)`.
  Der zusätzlich für die z-Werte erforderliche Platz im Bildspeicher (*frame buffer*), d.h. der z-Buffer wird mit Hilfe des Befehls `glutInitDisplayMode(GLUT_DEPTH)` aus dem OpenGL *Utility Toolkit* (GLUT) bei der Initialisierung angefordert.

- *Initialisierung des z-Buffers*: Festlegung des Initialisierungs-z-Wertes und Start der Initialisierung.
  Die zulässigen z-Werte reichen bei OpenGL von minimal 0.0 für die vordere Grenzfläche (*near clipping plane*) des sichtbaren Volumens bis maximal 1.0 für die hintere Grenzfläche (far clipping plane) des sichtbaren Volumens. Standardmäßig wird der z-Wert für die Initialisierung des z-Buffers auf 1.0 gesetzt. Mit dem Befehl

`glClearDepth(GLdouble depth)` kann ein beliebiger Initialisierungs-z-Wert zwischen 0.0 und 1.0 gewählt werden. In seltenen Fällen kann es nötig sein, den Wertebereich des z-Buffers, der normalerweise von 0.0 bis 1.0 läuft, weiter einzuschränken. Mit dem Befehl `glDepthRange(GLdouble near, GLdouble far)` wird der Wertebereich des z-Buffers auf $0.0 \leq$ `near, far` ≤ 1.0 eingeschränkt. Bevor ein neues Bild gerendert werden kann, müssen die alten z-Werte (und auch die Farbwerte) im Bildspeicher gelöscht werden. Dies geschieht am besten dadurch, dass alle z-Werte des Bildspeichers auf den mit „`glClearDepth()`" voreingestellten Wert gesetzt werden (die Farbwerte werden mit dem Befehl „`glClearColor()`" voreingestellt). Jedesmal, wenn ein neues Bild gezeichnet werden soll, werden durch den Aufruf von `glClear(GL_DEPTH_BUFFER_BIT)` die z-Werte des Bildspeichers initialisiert. Wegen der höheren Effizienz sollten neben den z-Werten gleichzeitig auch die Farbwerte des Bildspeichers mit `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` initialisiert werden.

- *Vergleich*: Festlegung des Vergleichs-Operators.
  Standardmäßig wird als Vergleichs-Operator der Parameter „`GL_LESS`" (d.h. „$<$") verwendet, wie im Algorithmus A8.1 beschrieben. Einen anderen Vergleichs-Operator kann man mit dem Befehl `glDepthFunc(GLenum operator)` festlegen. Das Argument „`operator`" kann die in der folgenden Tabelle aufgelisteten Werte annehmen:

operator	**Funktion**
GL_LESS	$<$, kleiner (Standardwert)
GL_NEVER	0, liefert immer den Wahrheitswert „FALSE"
GL_EQUAL	$=$, gleich
GL_LEQUAL	$\leq$, kleiner gleich
GL_GREATER	$>$, größer
GL_GEQUAL	$\geq$, größer gleich
GL_NOTEQUAL	$\neq$, ungleich
GL_ALWAYS	1, liefert immer den Wahrheitswert „TRUE"

  Liefert der Vergleich zwischen dem neuen z-Wert und dem gespeicherten z-Wert den Wahrheitswert „TRUE", werden die neuen Farbwerte in den Farbspeicher und der neue z-Wert in den z-Buffer für dieses Pixel eingetragen. Beim Wahrheitswert „FALSE" wird nichts geändert.

8.3 Einsatzmöglichkeiten des z-Buffer Algorithmus

Neben der üblichen Anwendung des z-Buffer Algorithmus' zur Verdeckungsrechnung gibt es eine Reihe weiterer interessanter Einsatzmöglichkeiten, die in den folgenden Abschnitten beschrieben werden.

8.3.1 Entfernen aller Vorderteile

Wird als Vergleichs-Operator „GL_GREATER“ eingesetzt, erhalten die Pixel die Farbe des am weitesten vom Augenpunkt entfernten Objekts, sprich die Rückseiten der hintersten Objekte. Dadurch werden bei einem Objekt die Vorderseiten quasi weggeschnitten, und bei einer Szene mit mehreren sich verdeckenden Objekten wird der Blick auf die Rückseiten des hintersten Objekts frei. Die Wirkung ist ähnlich, aber nicht identisch wie beim *Frontface-Culling*, denn dort können sich räumlich hintereinander angeordetete Rückseiten (*back faces*) durchaus gegenseitig verdecken (Bild 6.17-b). In Bild 8.4 wird anhand von zwei Szenen die unterschiedliche Auswirkung der Operatoren „GL_LESS“ und „GL_GREATER“ dargestellt. Im linken oberen Teilbild sind die Vorderseiten eines Balkens zu sehen (Operator „GL_LESS“). Im linken unteren Teilbild sind die Hinterseiten des Balkens (Operatoren „GL_GREATER“) von innen zu sehen; der Balken ist quasi innen leer und nur die drei hinteren Hüllflächen sind sichtbar. Im rechten oberen Teilbild ist in gewohnter Weise ein Dinosaurier von oben zu sehen, der von einem Balken durchdrungen wird. Das rechte untere Teilbild ist für unsere Wahrnehmung extrem verwirrend: denn einerseits erscheint es so, als würde man die nach außen gewölbte Vorderseite des Dinosauriers jetzt von unten sehen; andererseits sieht man die Hinterseiten des Balkens von oben! In Wirklichkeit sieht man aber auch die nach innen gewölbten Hinterseiten des Dinosaurier von oben. Wir unterliegen hier einer optischen Täuschung. Denn man sieht normalerweise nie die ausgehöhlte Rückseite eines Tieres (außer bei einem Gipsabdruck). Deshalb unterdrückt unsere Wahrnehmung diese Interpretation und gaukelt uns stattdessen die aus unserer Erfahrung sehr viel plausiblere Variante eines von unten gesehenen Dinosauriers vor.

Wird als Vergleichs-Operator für den z-Buffer Algorithmus „GL_GREATER“ eingesetzt, sollte der Initialisierungs-z-Wert mit dem Befehl `glClearDepth()` auf 0 oder einen anderen Wert < 1 gesetzt werden. Denn per Definition kann kein z-Wert größer als der Standard-Initialisierungs-z-Wert 1 sein. In Bild 8.5 ist das Prinzip des z-Buffer Algorithmus mit dem Vergleichs-Operator „GL_GREATER“ noch einmal dargestellt (im Vergleich dazu ist in Bild 8.3 das Prinzip des z-Buffer Algorithmus für den Vergleichs-Operator „GL_LESS“ dargestellt).

8.3.2 Höhenkarten generieren

Nachdem durch den Algorithmus A8.1 mit dem Vergleichs-Operator „GL_LESS“ der z-Buffer in gewohnter Weise beschrieben ist, kann er ohne weitere Veränderung z.B. für die Generierung von Höhenlinien verwendet werden (Bild 8.6-a1). Dazu wird der z-Buffer durch den Befehl `glDepthMask(GL_FALSE)` ausschließlich lesbar, aber nicht mehr beschreibbar gemacht. Danach wird eine Serie von Flächen über die Szene gelegt, die parallel zur „*far clipping plane*“ sind, aber einen abnehmenden z-Wert besitzen. Jede einzelne Fläche steht senkrecht auf der z-Achse und wird mit dem Vergleichs-Operator „GL_GREATER“ gezeichnet. Die hinterste Fläche mit einem z-Wert von z.B. 0.9 enthält all diejenigen Pixel, bei denen der z-Wert der Fläche (0.9) größer ist als der gespeicherte z-Wert (in Bild 8.6-a1 der gesamte Dinosaurier). Diese Pixel werden mit dem niedrigsten Grauwert belegt. Die nächste, etwas hellere Fläche bedeckt einen kleineren Anteil des Dinosauriers, da der hin-

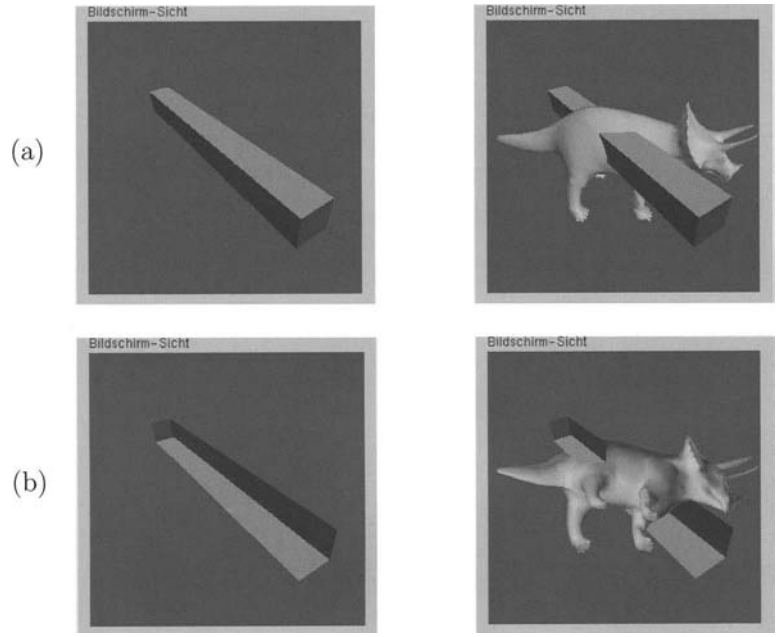


Bild 8.4: Wegschneiden aller Vorderteile einer Szene mit dem Vergleichs-Operator „`GL_GREATER`“: (a) Standard z-Buffer Algorithmus mit dem Vergleichs-Operator „`GL_LESS`“, d.h. die Vorderteile einer Szene sind sichtbar. (b) z-Buffer Algorithmus mit dem Vergleichs-Operator „`GL_GREATER`“, d.h. die Vorderteile einer Szene werden weggeschnitten.

terste Teil des Dinosaurierschwanzes einen z-Wert größer als 0.8 hat. Jede weitere und zunehmend hellere Fläche bedeckt einen immer kleineren Anteil des Dinosauriers, so dass letzten Endes der Grauwert des Pixels die räumliche Tiefe repräsentiert.

Eine andere Alternative zur Darstellung von räumlicher Tiefe durch Grauwerte besteht im direkten Auslesen aller z-Buffer-Werte aus dem Bildspeicher. Dazu definiert man zunächst ein Array „`GLfloat z[width*height]`“, das die z-Werte für jedes Pixel aufnehmen kann. Mit dem Befehl „`glReadPixels(x_offset, y_offset, width, height, GL_DEPTH_COMPONENT, GL_FLOAT, z)`“ werden die z-Werte aus dem Bildspeicher ausgelesen und in das Array „`z`“ geschrieben. Anschließend können die z-Werte bearbeitet werden, wie bei dem Beispiel in Bild 8.6-a2, bei dem durch die Operation „$z[i] = 1 - z[i]$“ ein Negativ-Bild erzeugt wird. Mit Hilfe des Befehls „`glDrawPixels(width, height, GL_LUMINANCE, GL_FLOAT, z)`“ können die bearbeiteten z-Werte dann als Grauwerte auf den Bildschirm gebracht werden. Bild 8.6-a2 ist also eine direkte Visualisierung des z-Buffer-Inhalts, bei dem ein zunehmender Grauwert einen abnehmenden z-Wert repräsentiert.

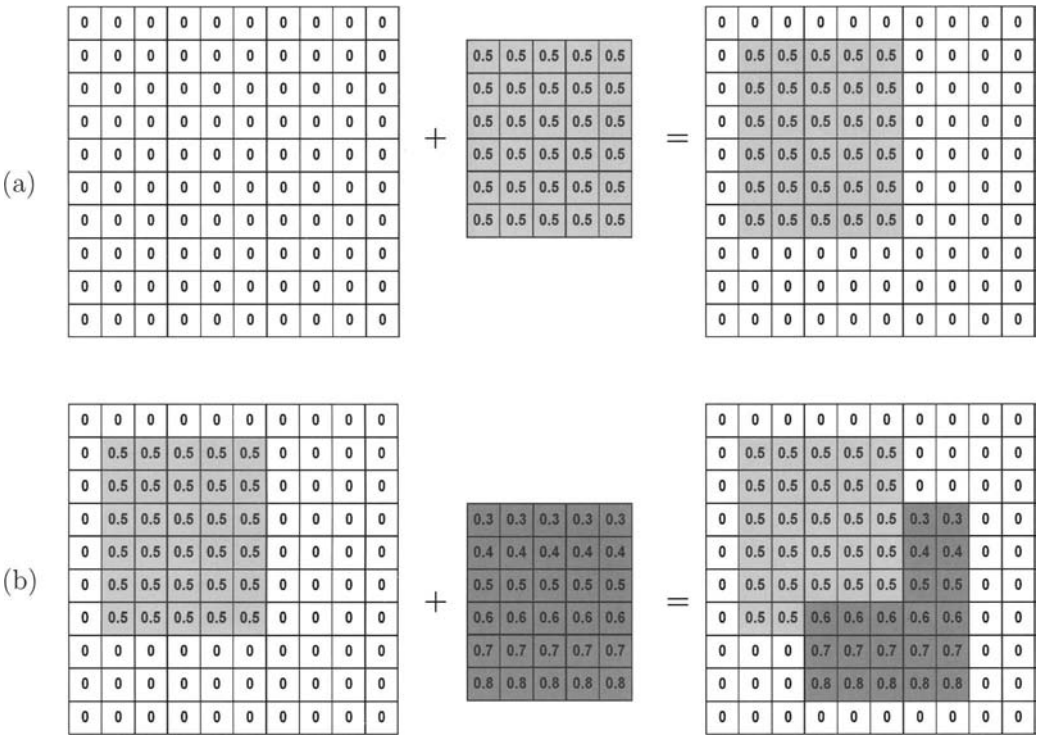


Bild 8.5: Das Prinzip des z-Buffers mit dem Vergleichs-Operator „GL_GREATER“: die Zahl in jedem Kästchen repräsentiert die räumliche Tiefe, d.h. den z-Wert des Pixels, die Graustufe der Pixel repräsentiert die Farbwerte. (a) auf einen „sauberen“ Bildspeicher (initialisiert mit minimalem z-Wert 0) wird ein Polygon mit konstantem z-Wert von 0.5 addiert (d.h. das Polygon steht senkrecht auf der Sichtlinie) (b) Addition eines weiteren Polygons, das gegenüber der Sichtlinie geneigt ist und das erste Polygon schneidet

8.3.3 Volumenmessung

Zur Volumenmessung wird das jeweilige 3D-Objekt zunächst in feine Scheiben der Dicke dz geschnitten, die senkrecht auf der z-Achse stehen (Bild 8.6-b). Das Prinzip ist ähnlich wie bei den Höhenlinien in Bild 8.6-a1, nur mit dem Unterschied, dass pro Bild nur eine einzige Scheibe herausgeschnitten wird. Der Flächeninhalt F_i der i-ten Scheibe kann mit Hilfe einfacher Bildverarbeitungsalgorithmen (Kapitel 35 „Einfache segmentbeschreibende Parameter“) bestimmt werden. Liegen die einzelnen Scheiben in z-Richtung dicht aneinander (in Bild 8.6-b sind nur 8 von 50 Scheiben dargestellt), errechnet sich das Volumen V aus:

$$V = \sum_i F_i \cdot dz \tag{8.1}$$

Für eine korrekte Volumenmessung ist die orthografische Projektion zu verwenden.

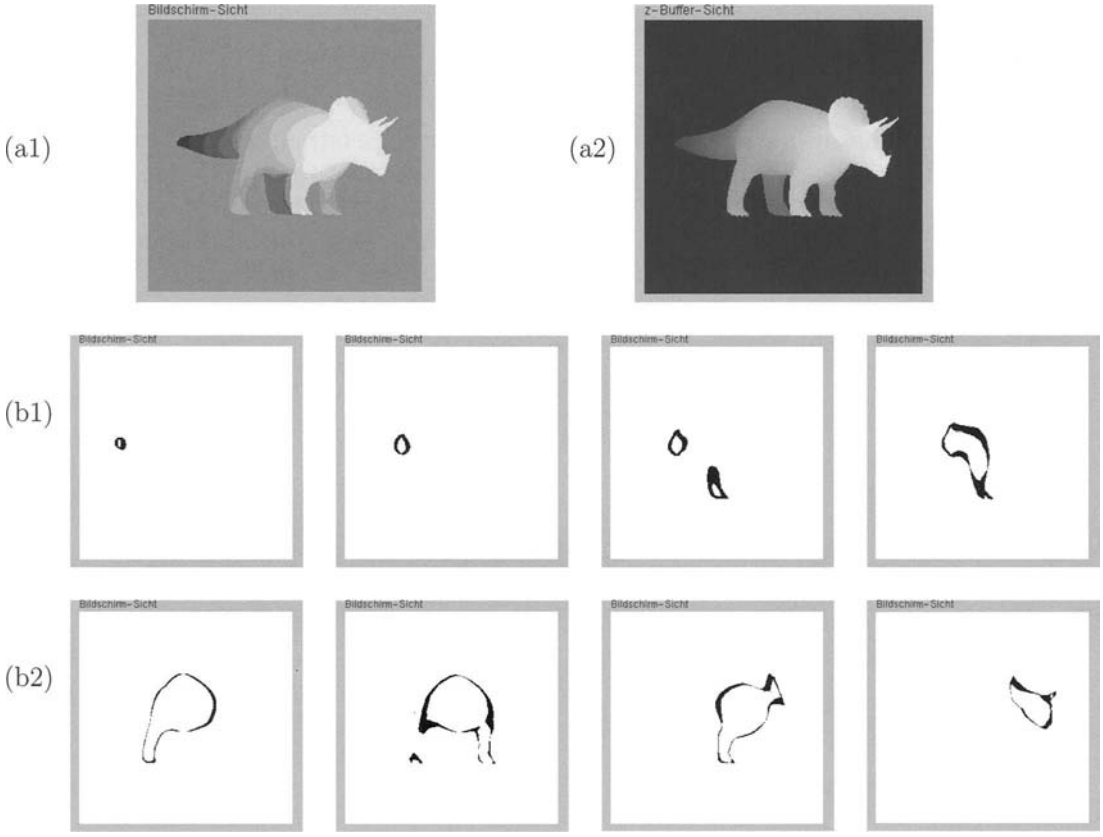


Bild 8.6: Verwendung des z-Buffers zur Herstellung von Höhenkarten (a1,a2), bei denen der Grauwert die räumliche Tiefe repräsentiert, und Schnittbildern (b1,b2), aus denen Volumina und Oberflächen der 3D-Objekte mit Hilfe von Bildverarbeitungsalgorithmen bestimmt werden können.

8.3.4 Oberflächenmessung

Die Oberflächenmessung läuft im Prinzip genau so ab wie die Volumenmessung. Der Unterschied besteht nur darin, dass aus den Schnittbildern 8.6-b mit Hilfe einfacher Bildverarbeitungsalgorithmen (Kapitel 35 „Einfache segmentbeschreibende Parameter") diesmal die Länge L_i der Objekt-Ränder der i-ten Scheibe bestimmt wird. Die Oberfläche eines Objekts ergibt sich dann aus:

$$O = \sum_i L_i \cdot dz \tag{8.2}$$

8.3.5 Entfernungsmessung

Zur Simulation eines Laser-Entfernungsmessgerätes kann der z-Buffer ebenfalls sehr gut eingesetzt werden. Zuerst werden mit Hilfe der Viewport-Transformation (7.30) die x- und y-Bildschirmkoordinaten ermittelt, an denen der Laserstrahl auf die Objektoberfläche trifft. Danach wird mit dem Befehl „`glReadPixels(x_offset, y_offset, width, height, GL_DEPTH_COMPONENT, GL_FLOAT, z)`“ die normierte z-Koordinate dieses Punktes aus dem Bildspeicher ausgelesen. Mit Hilfe der inversen Projektionstransformation $(P \cdot N)^{-1}$ (7.19, 7.27) wird die normierte z-Koordinate ins Weltkoordinatensystem umgerechnet. Im Weltkoordinatensystem kann dann der Abstand zwischen dem Ort des Lasermessgeräts und dem Punkt, an dem der Laserstrahl die Oberfläche trifft, durch die euklidische Norm berechnet werden.

Kapitel 9

Farbe, Transparenz und Farbmischung

Das eigentliche Ziel der Interaktiven 3D-Computergrafik ist die Generierung eines Farbbildes in einem Bildschirmfenster. Dieses Fenster besteht aus einer rechteckigen Anordnung von einzelnen Bildpunkten (Pixel), von denen jeder eine eigene Farbe darstellen kann.

9.1 Das Farbmodell in OpenGL

In der Computergrafik wird das RGB-Farbmodell verwendet, das neben einer Reihe anderer Farbmodelle im Abschnitt 16.5 ausführlich dargestellt wird. Allerdings wird hier die Farbe nicht wie sonst üblich durch die drei Komponenten Rot, Grün, Blau (RGB) spezifiziert, sondern meistens, wie z.B. in OpenGL, durch vier Komponenten: neben den drei Werten „RGB" wird als vierte Komponente noch der Wert „A" für die Transparenz angegeben. Dieses 4-Tupel (R,G,B,A) stellt die erweiterte Farbdefinition in der Computergrafik dar und aus diesem Grund wird „Farbe und Transparenz" in einem Atemzug genannt. Mit der Transparenz-Komponente „A" ist es möglich, verschiedenfarbige Flächen oder Pixel wie in einem Malkasten zu mischen. Weil sich als Kürzel für die Transparenz-Komponente der Buchstabe „A" eingebürgert hat, bezeichnet man die damit mögliche Farbmischung im englischen Fachjargon auch als „*Alpha Blending*".

Wie im Abschnitt 16.5.4 „Das RGB-Farbmodell" dargestellt, werden die Größen so normiert, dass der Wertebereich der Komponenten zwischen 0 und 1 liegt. Dies gilt in OpenGL nicht nur für die drei Farbkomponenten R,G,B, sondern auch für die Transparenz-Komponente A. Der Wert der jeweiligen Farbkomponente ist ein Maß für die Intensität. Durch eine Rot-Komponente von R = 1 wird die maximale Intensität, durch R = 0 die minimale Intensität für Rot spezifiziert. Dies gilt ebenso für die Grün- und die Blau-Komponente. Die Transparenz-Komponente „A" ist ein Maß für die Opazität (Undurchsichtigkeit, Intransparenz) einer Fläche oder eines Pixels. Eine „Alpha"-Komponente von 0 bedeutet 0% opak (d.h. 100% transparent), eine „Alpha"-Komponente von 1 bedeutet 100% opak (d.h. 0% transparent, also undurchsichtig).

Die OpenGL-interne Darstellung aller vier Farbkomponenten durch Gleitkommazahlen aus dem Intervall [0,1] lässt noch die Frage nach der Quantisierung (oder Auflösung) der Farbkomponenten offen (Abschnitt 16.2). Eine Farbkomponente kann z.B. durch 1 bit (2 Werte), 8 bit (256 Werte) oder 16 bit (65536 Werte) dargestellt werden. Aufgrund des Pipeline-Verarbeitungsprinzips der Computergrafik unterscheidet man drei verschiedene Quantisierungen:

- Die Eingangsquantisierung:
  Farben können in vollkommen unterschiedlichen Datenformaten eingegeben werden (z.B. als ganze Zahlen unterschiedlich feiner Quantisierung oder als Gleitkomma-Zahlen, Abschnitt 9.2.4). Durch einen Normierungsschritt werden Integer-Datenformate in Gleitkommazahlen des Intervalls [0,1] umgewandelt (die eingegebene Integer-Zahl wird durch die – von der Quantisierung abhängige – maximale Integer-Zahl geteilt). Bei Gleitkomma-Zahlen werden Werte unter null oder über eins gekappt, d.h. auf null bzw. eins gesetzt.

- Die interne Quantisierung:
  die interne Quantisierung hängt von der verwendeten Hardware ab. Einfache Grafikkarten bieten heutzutage eine Auflösung von 8 bit pro Farbkomponente, gute dagegen 32 bit pro Farbkomponente. Mit der jeweiligen internen Quantisierung werden Farbinterpolationen (z.B. Farbmischung oder Schattierung), die Beleuchtungsrechnung oder die Texturierung durchgeführt. Wenn man z.B. weiß, dass die interne Quantisierung 8 bit pro Komponente beträgt, bringt es nichts, einen Farbwert als `GLuint` mit 32 bit oder eine Textur als `GLushort` mit 16 bit pro Komponente einzugeben, da nach der Normierung die höhere Genauigkeit der Eingangswerte wieder verloren geht. Am Ende aller internen Berechnungen werden evtl. Farbwerte außerhalb des Intervalls [0,1] wieder gekappt.

- Die Ausgangsquantisierung:
  am Ende des Rendering-Prozesses steht das fertige Bild im Bildspeicher für die Ausgabe auf einem Anzeigegerät (z.B. einen Bildschirm) bereit. Die Auflösung der Farbkomponenten jedes Pixels im Bildspeicher wird als Ausgangsquantisierung bezeichnet. Sie kann – im Rahmen der von der Fensterverwaltung (z.B. GLX, WGL, GLUT) und der Hardware gebotenen Möglichkeiten – unabhängig von der internen Quantisierung gewählt werden. Normalerweise wählt man die Ausgangsquantisierung passend zum Ausgabemedium. Praktisch alle handelsüblichen Bildschirme können bestenfalls 8 bit pro Farbkanal darstellen, so dass dieser Wert meistens als Ausgangsquantisierung festgelegt wird. Die Ausgangsquantisierung wird bei der Initialisierung des Bildspeichers festgelegt.

Um einem 3D-Objekt Farben zuweisen zu können, gibt es in der Computergrafik zwei prinzipiell unterschiedliche Möglichkeiten, die den zwei Datenpfaden der Rendering-Pipeline in Bild 5.1 entsprechen:

- Zuweisung einer Farbe für jeden Vertex des 3D-Objekts. Im Rahmen der Rasterisierung werden anschließend aus den Vertex-Farben die Fragment-Farben und später die Pixel-Farben berechnet. Diese Möglichkeit, Objekte einzufärben, kann selbst wieder auf zwei Weisen erfolgen:
  - In einer einfachen und direkten Weise, indem jedem Vertex explizit Werte für die vier Komponenten R,G,B,A zugewiesen werden. Damit beschäftigt sich dieses Kapitel.
  - In einer komplexen Weise, indem aus den Eigenschaften und Anordnungen von Lichtquellen und Oberflächen über eine Beleuchtungsrechnung die Vertex-Farbe berechnet wird. Die Beleuchtungsmodelle werden in Kapitel 12 behandelt.
- Zuweisung von Farben für jeden Bildpunkt einer Textur. Nach der Rasterisierung werden die Fragment-Farben der Vertex-Daten mit den Textur-Farben gemischt oder durch sie ersetzt. Dies wird im Kapitel 13 „Texturen“ ausführlich erklärt.

Die Farbauflösung (Anzahl möglicher Farbkombinationen pro Pixel) und die Ortsauflösung (Anzahl der Pixel eines Bildes) hängen von der Größe des Bildspeichers der verwendeten Grafik-Hardware ab. Bei einer Ortsauflösung von z.B. 1600x1200 Pixel und einer Farbauflösung von 8 bit pro Farb-Komponente („true color“), d.h. 32 bit (= 4 Byte) pro Pixel, benötigt man schon ca. 7,3 MByte allein für den Farbspeicher (engl. color buffer). Da der Bildspeicher aber nicht nur aus dem „color buffer“ besteht, sondern auch noch den „z buffer“, „stencil buffer“, „accumulation buffer“ und den „double buffer“ beherbergen muss, in denen für jedes Pixel eines Bildes eine bestimmte Anzahl an bits zur Verfügung gestellt wird, kann der notwendige Speicherplatz eine beachtliche Größenordnung erreichen. Zur Reduktion der Farbauflösung und damit des Speicherplatzbedarfs hat sich deshalb schon früh die Technik der „Farb-Index–Bilder“ etabliert, bei der nur eine eingeschränkte Anzahl an Farben (z.B. die 256 häufigsten Farben eines Bildes) zur Verfügung gestellt wird. Jede Farbkombination wird in diesem Beispiel eindeutig durch einen Index zwischen 0 und 255 repräsentiert, so dass zur Speicherung eines Index nur 8 bit pro Pixel erforderlich sind. Eine ausführliche Darstellung von Verfahren zur Farbreduktion mit Hilfe von Indexbildern ist im Abschnitt 24.5 zu finden.

9.2 Modelle der Farbdarstellung

In OpenGL gibt es aus den vorher genannten Gründen zwei prinzipiell verschiedene Darstellungsmodi für die Farbe eines Pixels im Bildspeicher, die zusammen mit ihren Vor- und Nachteilen im Folgenden erläutert werden.

9.2.1 Der RGBA–Modus

Im RGBA–Modus werden für jedes einzelne Pixel des Bildspeichers vier Werte für die Komponenten R,G,B und A direkt abgespeichert.

...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	R,G,B,A	R,G,B,A	R,G,B,A	...	...
...	...	R,G,B,A	R,G,B,A	R,G,B,A	...	...
...	...	R,G,B,A	R,G,B,A	R,G,B,A	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...

Bildspeicher im RGBA–Modus

9.2.2 Der Farb-Index–Modus

Im Farb-Index–Modus wird für jedes einzelne Pixel des Bildspeichers nur ein Wert, der sogenannte „Farb-Index" (engl. color index) abgespeichert. Jeder Farb-Index ist ein Zeiger in eine Farb-Index-Tabelle (engl. color look up table (CLUT)), in der ein Satz von R,G,B–Werten definiert ist.

...	...	...	...	...	...	...
...	...	...	...	...	...	...
...	...	Index	Index	Index	...	...
...	...	Index	Index	Index	...	...
...	...	Index	Index	Index	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...

Bildspeicher im Farb-Index–Modus

Index	R	G	B
0	1	0	0
1	0.9	0	0
...	...	...	...
...	...	...	...
255	0	0	1

Farb-Index–Tabelle

Im Farb-Index–Modus können keine „Alpha (A)"-Werte definiert werden, da Farbmischungen von transparenten Oberflächen in diesem Modus äußerst problematisch wären. Denn gemischt würden in diesem Fall zwei Farb-Indizes, und ein „gemittelter" Farb-Index kann in der Farb-Tabelle durch eine völlig andere Farbe repräsentiert werden, als man normalerweise bei einer Mischung der beiden Ausgangsfarben erwarten würde. Nur unter der Voraussetzung, dass die Werte der einzelnen Farben Rot, Grün und Blau in der Farb-Index-Tabelle von Index zu Index monoton steigen oder fallen (sogenannte „Farbrampen"),

kann man eine halbwegs vernünftige Farbmischung erwarten. Die in OpenGL verfügbaren Befehle zur Farbmischung kann man allerdings nur im RGBA–Modus nutzen, im Farb-Index–Modus müssen die entsprechenden Funktionalitäten selbst programmiert werden.

9.2.3 Wahl zwischen RGBA– und Farb-Index–Modus

Aufgrund des relativ großen Bildspeichers bei modernen Grafikkarten wählt man heutzutage fast immer den RGBA–Darstellungsmodus. Denn in diesem Modus kann man auf den meisten Systemen eine sehr viel höhere Farbauflösung und damit eine deutlich bessere Bildqualität erzielen, als im Farb-Index–Modus. Außerdem sind im RGBA–Modus viele Effekte, die auf Farbmischung beruhen, wie z.B. Beleuchtung, Schattierung, Textur-Mapping, Nebel und Anti-Aliasing sehr viel einfacher und flexibler zu implementieren als im Farb-Index–Modus. In den folgenden Fällen kann der Einsatz des Farb-Index–Modus' jedoch sinnvoll sein:

- Pro Pixel steht nur eine relativ niedrige Anzahl an Farb-bits zur Verfügung und die Häufigkeitsverteilung der einzelnen RGB-Farbwerte weist deutliche Maxima auf.

- Alle Pixel mit der gleichen Farbe (bzw. dem gleichen Farbindex) sollen verändert werden. Damit kann z.B. eine sehr einfache Umschaltung von einer Sommerlandschaft in eine Winterlandschaft realisiert werden. Da eine Sommerlandschaft häufig viele satte Grüntöne enthält, die im Winter in Weiß übergehen, kann die Umschaltung durch wenige Federstriche in der Farb-Index-Tabelle erreicht werden. Es müssen nur die Zeilen, die die satten Grüntöne enthalten, auf Weiß gesetzt werden. Danach werden alle Bildpunkte, die ursprünglich zu einer grünen Wiese gehörten, einen schneebedeckten Eindruck machen. Allerdings funktioniert dieser Trick nur in bestimmten Umgebungen befriedigend. Denn andere Oberflächen, auf denen Schnee auch liegen bleiben könnte, wie z.B. rote Dächer, bleiben unverändert. Im RGBA–Modus könnte dieser Trick nicht ohne erhebliche Geschwindigkeitseinbußen implementiert werden, denn dort müsste die Farbe für jedes einzelne Pixel überprüft und gegebenenfalls ersetzt werden.

9.2.4 Farbspezifikation im RGBA–Modus

Bei Verwendung der GLUT-Library für die Window-Verwaltung wird mit dem Befehl „`glutInitDisplayMode(GLUT_RGBA)`“ bei der Initialisierung des Bildschirmfensters zunächst einmal der RGBA-Darstellungsmodus festgelegt.

Zur Festlegung der Farbe von Vertices wird der OpenGL-Befehl `glColor*()` benützt, der in verschiedenen Ausprägungen existiert, wie in der folgenden Tabelle dargestellt:

Skalar-Form	Vektor-Form	Alpha-Komponente
glColor3f(R,B,G)	glColor3fv(vec)	1.0
glColor3d(R,B,G)	glColor3dv(vec)	1.0
glColor3b(R,B,G)	glColor3bv(vec)	1.0
glColor3s(R,B,G)	glColor3sv(vec)	1.0
glColor3i(R,B,G)	glColor3iv(vec)	1.0
glColor3ub(R,B,G)	glColor3ubv(vec)	1.0
glColor3us(R,B,G)	glColor3usv(vec)	1.0
glColor3ui(R,B,G)	glColor3uiv(vec)	1.0
glColor4f(R,B,G,A)	glColor4fv(vec)	z.d.
glColor4d(R,B,G,A)	glColor4dv(vec)	z.d.
glColor4b(R,B,G,A)	glColor4bv(vec)	z.d.
glColor4s(R,B,G,A)	glColor4sv(vec)	z.d.
glColor4i(R,B,G,A)	glColor4iv(vec)	z.d.
glColor4ub(R,B,G,A)	glColor4ubv(vec)	z.d.
glColor4us(R,B,G,A)	glColor4usv(vec)	z.d.
glColor4ui(R,B,G,A)	glColor4uiv(vec)	z.d.

z.d. = zu definieren

In der Skalar–Form des Befehls müssen die Farbwerte R,G,B,A im entsprechenden Datenformat (z.B. `f = GLfloat`) direkt übergeben werden, wie im Folgenden Beispiel gezeigt:

```
glColor3f(1.0, 0.47, 0.0);
```

In der Vektor–Form des Befehls wird nur ein Zeiger auf ein Array übergeben, das die Farbwerte im entsprechenden Datenformat (z.B. `f = GLfloat`) enthält, wie im Folgenden Beispiel gezeigt:

```
GLfloat vec[3] = {1.0, 0.47, 0.0};
glColor3fv(vec);
```

Alle in der obigen Tabelle dargestellten Datenformate, in denen Farben spezifiziert werden können, werden nach der Beleuchtungsrechnung in normierte Gleitkommazahlen im Bereich zwischen 0.0 und 1.0 umgewandelt. Für die Gleitkomma-Datentypen `f (GLfloat)` und `d (GLdouble)` bedeutet dies, dass negative Zahlen auf 0.0 abgebildet werden, Zahlen zwischen 0.0 und 1.0 unverändert bleiben, und Zahlen größer als 1.0 auf 1.0 abgebildet werden. Die Datentypen `ub` (unsigned 1-byte integer), `us` (unsigned 2-byte integer), und `ui` (unsigned 4-byte integer) werden linear in den Bereich von 0.0 bis 1.0 abgebildet. Für den Datentyp `ub` z.B. heißt das konkret: $0 \rightarrow 0/255 = 0.0$, $1 \rightarrow 1/255$, ..., $255 \rightarrow 255/255 = 1.0$. Die vorzeichenbehafteten Datentypen `b, s` und `i` werden zunächst linear in den Gleitkommazahlenbereich von -1.0 bis +1.0 abgebildet, und nach der Beleuchtungsrechnung werden alle negativen Zahlen auf 0.0 abgebildet. Nach der Rasterisierung in Fragmente werden die normierten Gleitkommazahlen wieder auf den im Bildspeicher für die Farbkomponenten verfügbaren Zahlenbereich abgebildet (z.B. 8, 12, 16 oder 32 bit je Farbkomponente).

Da OpenGL ein Zustandsautomat ist, wird nach einem `glColor()`-Befehl allen folgenden Vertices die spezifizierte Farbe zugewiesen und zwar so lange, bis durch den nächsten `glColor()`-Befehl eine neue Farbe festgelegt wird. Der `glColor()`-Befehl kann nicht nur außerhalb, sondern auch innerhalb einer `glBegin()-/glEnd()`-Klammer angewendet werden. Damit ist es möglich, jedem Vertex eine eigene Farbe zuzuweisen. Falls das standardmäßig bei OpenGL eingestellte Schattierungsmodell, das „*smooth shading*" (Abschnitt 12.2.2), angewendet wird, können damit lineare Farbverläufe zwischen den unterschiedlichen Vertex-Farben erzeugt werden (Bild 9.1).

```
glColor3us(32767, 32767, 32767);
glBegin(GL_TRIANGLES);
    glVertex3fv(v0);
    glVertex3fv(v1);
    glVertex3fv(v2);
    glColor3ub(235, 235, 235);
    glVertex3fv(v3);
    glColor3f(0.0, 0.0, 0.0);
    glVertex3fv(v4);
    glVertex3fv(v5);
glEnd();
```

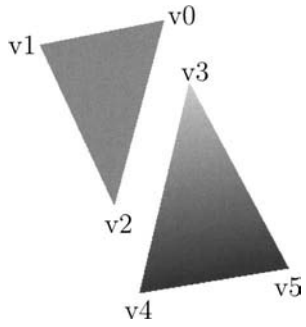


Bild 9.1: Direkte Farbzuweisung in OpenGL: das erste im Code spezifizierte Dreieck, das im Bild links oben dargestellt ist, erhält eine einheitliche Farbe (Mittelgrau), da allen Vertices die gleiche Farbe zugewiesen wurde; das zweite im Code spezifizierte Dreieck, das im Bild rechts unten zu sehen ist, weist einen linearen Farbverlauf auf (von Hellgrau bis Schwarz), da dem oberen Vertex eine andere Farbe zugwiesen wurde als den beiden unteren.

Eine Hintergrund-Farbe (*clear color*), mit der der Bildspeicher initialisiert wird, kann mit dem OpenGL-Befehl „`glClearColor(R,G,B,A)`" festgelegt werden. Dabei müssen die Argumente R = rot, G = grün, B = blau, A = alpha vom Datentyp `GLfloat` sein und im Wertebereich zwischen 0.0 und 1.0 liegen. Jedesmal, wenn ein neues Bild gezeichnet werden soll, werden durch den Aufruf von `glClear(GL_COLOR_BUFFER_BIT)` alle Farbwerte des Bildspeichers auf die mit dem Befehl „`glClearColor()`" voreingestellten Werte gesetzt. Wie in Kapitel 8 bereits erwähnt, sollten wegen der höheren Effizienz neben den Farbwerten gleichzeitig auch die z-Werte des Bildspeichers mit `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` initialisiert werden.

9.2.5 Farbspezifikation im Farb-Index–Modus

Um Farben im Farb-Index–Modus festlegen zu können, sind zunächst ein paar Vorbereitungsschritte durchzuführen, die betriebssystemspezifisch sind und deshalb nicht zum Sprachumfang von OpenGL gehören. Im Folgenden werden die entsprechenden Befehle aus der GLUT-Library angegeben:

- Laden einer Farb-Index-Tabelle mit Hilfe des Befehls „`glutSetColor(Index,R,G,B)`". Um eine Farb-Index-Tabelle mit 256 Indizes zu erstellen, die einen linearen Farbverlauf zwischen Rot und Grün enthält, kann der folgende Programmausschnitt dienen:
  `for(i=0;i<255;i++) {glutSetColor(i, 1.0-(i/255.0), i/255.0, 0.0); }`
- Initialisierung des Bildschirmfensters im Farb-Index–Modus mit dem Befehl „`glutInitDisplayMode(GLUT_INDEX)`".

Zur Festlegung der Farbe von Vertices im Farb-Index–Modus muss nur der jeweilige Index zugewiesen werden. Dazu wird der OpenGL-Befehl `glIndex*()` benützt, der in verschiedenen Ausprägungen existiert, wie in der folgenden Tabelle dargestellt:

Skalar-Form	Vektor-Form
`glIndexf(I)`	`glIndexfv(vec)`
`glIndexd(I)`	`glIndexdv(vec)`
`glIndexs(I)`	`glIndexsv(vec)`
`glIndexi(I)`	`glIndexiv(vec)`
`glIndexub(I)`	`glIndexubv(vec)`

In der Skalar–Form des Befehls muss der Index I im entsprechenden Datenformat (z.B. `f = GLfloat`) direkt übergeben werden, wie im Folgenden Beispiel gezeigt:

```
glIndexf(21.0);
```

In der Vektor–Form des Befehls wird nur ein Zeiger auf ein Array übergeben, das den Index im entsprechenden Datenformat (z.B. `f = GLfloat`) enthält, wie im Folgenden Beispiel gezeigt:

```
GLfloat vec[1] = {21.0};
glIndexfv(vec);
```

Indizes werden generell als Gleitkommazahlen gespeichert. Integer-Werte werden direkt konvertiert (z.B. `GLint 21` → `GLfloat 21.0`). Nach der Rasterisierung in Fragmente werden die Gleitkommazahlen wieder auf den im Bildspeicher für die Farbindizes verfügbaren Festkomma-Zahlenbereich abgebildet (z.B. 8 oder 16 bit).

Ebenso wie im RGBA–Modus kann auch im Farb-Index–Modus eine Hintergrund-Farbe festgelegt werden, mit der der Bildspeicher initialisiert wird. Der dafür vorgesehene OpenGL-Befehl lautet „`glClearIndex(GLfloat cI)`". Durch einen Aufruf von „`glClear(GL_COLOR_BUFFER_BIT)`" werden alle Farb-Indizes des Bildspeichers auf den Wert „`cI`" gesetzt.

9.3 Transparenz und Farbmischung

Die nur im RGBA–Modus verfügbare Transparenz- bzw. Alpha-Komponente einer Farbe und die damit mögliche Farbmischung ist die Grundlage für eine Reihe von wichtigen Techniken in der 3D-Computergrafik, wie z.B. Anti-Aliasing (Kantenglättung, Kapitel 10), Nebel (Kapitel 11), *Fading* (langsames Einblenden eines Objekts bzw. langsames Überblenden zwischen zwei Objekten, Abschnitt 15.1), Alpha-Texturen (Texturen bei denen nur einzelne Bildpunkte transparent sind und andere nicht, Abschnitt 9.3.3) und Spezialeffekte (Rauch, Feuer, Explosion usw.).

Die naheliegendste Anwendung von Alpha-Werten ist die Farbmischung zwischen halbdurchlässigen Oberflächen und dem Hintergrund. Beim Blick durch ein z.B. grünes Glas, das 70% des ankommenden Lichts transmittiert (d.h. Opazität A $= 1.0 - 0.7 = 0.3$), sieht man eine Farbmischung aus 30% Grünanteil des Glases und 70% Anteil der Hintergrundfarbe. Die resultierende Mischfarbe (R_m, G_m, B_m) berechnet sich aus der Farbe des Glases $(R_s = 0.0, G_s = 1.0, B_s = 0.0)$ und der Hintergrundfarbe (R_d, G_d, B_d) gemäß:

$$\begin{aligned} R_m &= 0.3 \cdot 0.0 + 0.7 \cdot R_d \\ G_m &= 0.3 \cdot 1.0 + 0.7 \cdot G_d \\ B_m &= 0.3 \cdot 0.0 + 0.7 \cdot B_d \end{aligned} \tag{9.1}$$

Wie generell Farbmischung in OpenGL funktioniert, wird im Folgenden beschrieben.

9.3.1 Farbmischung in OpenGL

Wie üblich muss auch die Farbmischung in OpenGL erst einmal durch folgenden Befehl aktiviert werden: „`glEnable(GL_BLEND)`". Falls die Farbmischung ausgeschaltet ist (`glDisable(GL_BLEND)`), wird die Alpha-Komponente aus dem 4-Tupel (R,G,B,A) einfach ignoriert.

Die Farbmischung wird vorgenommen, nachdem ein Polygon rasterisiert und dadurch in Fragmente konvertiert wurde, indem die Farbe eines Fragments (die „*Source Color*") mit der Farbe des im Bildspeicher vorhandenen Pixels (der „*Destination Color*") kombiniert wird. Eine Möglichkeit der Farbmischung besteht jetzt darin, den Alpha-Wert zu nutzen, um ein transparentes Fragment zu erzeugen, bei dem die Farbe des im Bildspeicher vorhandenen Pixels zu $(1 - A)\%$ „durchscheint". Dies ist eine spezielle, wenn auch die am häufigsten genutzte Art der Farbkombination. In OpenGL gibt es eine Vielzahl möglicher Arten der Farbkombination, die im Wesentlichen durch zwei OpenGL-Befehle (`glBlendFunc()` und `glBlendEquation()`) spezifiziert werden. Mit `glBlendFunc()` werden vier sogenannte „*Source Faktoren*" (S_r, S_g, S_b, S_a) und vier „*Destination Faktoren*" (D_r, D_g, D_b, D_a) festgelegt. Die vier „Source Faktoren" werden komponentenweise mit den RGBA-Farbwerten des neu hinzukommenden Fragments (den „Source Colors" (R_s, G_s, B_s, A_s)) multipliziert, und die vier „Destination Faktoren" werden komponentenweise mit den RGBA-Farbwerten des im Bildspeicher vorhandenen Pixels (den „Destination Colors" (R_d, G_d, B_d, A_d)) multipliziert. Mit `glBlendEquation()` wird festgelegt, wie diese Produkte von Source und Destination miteinander verknüpft werden:

- Addition `glBlendEquation(GL_FUNC_ADD)` (dies ist die Standard-Verknüpfungsfunktion, die auch ohne Aufruf von `glBlendEquation()` verwendet wird)

$$\begin{aligned} R_m &= S_r \cdot R_s + D_r \cdot R_d \\ G_m &= S_g \cdot G_s + D_g \cdot G_d \\ B_m &= S_b \cdot B_s + D_b \cdot B_d \\ A_m &= S_a \cdot A_s + D_a \cdot A_d \end{aligned} \tag{9.2}$$

- Subtraktion `glBlendEquation(GL_FUNC_SUBTRACT)`

$$\begin{aligned} R_m &= S_r \cdot R_s - D_r \cdot R_d \\ G_m &= S_g \cdot G_s - D_g \cdot G_d \\ B_m &= S_b \cdot B_s - D_b \cdot B_d \\ A_m &= S_a \cdot A_s - D_a \cdot A_d \end{aligned} \tag{9.3}$$

- umgekehrte Subtraktion `glBlendEquation(GL_FUNC_REVERSE_SUBTRACT)`

$$\begin{aligned} R_m &= D_r \cdot R_d - S_r \cdot R_s \\ G_m &= D_g \cdot G_d - S_g \cdot G_s \\ B_m &= D_b \cdot B_d - S_b \cdot B_s \\ A_m &= D_a \cdot A_d - S_a \cdot A_s \end{aligned} \tag{9.4}$$

- Minimalwert `glBlendEquation(GL_MIN)`

$$\begin{aligned} R_m &= min(R_s, R_d) \\ G_m &= min(G_s, G_d) \\ B_m &= min(B_s, B_d) \\ A_m &= min(A_s, A_d) \end{aligned} \tag{9.5}$$

- Maximalwert `glBlendEquation(GL_MAX)`

$$\begin{aligned} R_m &= max(R_s, R_d) \\ G_m &= max(G_s, G_d) \\ B_m &= max(B_s, B_d) \\ A_m &= max(A_s, A_d) \end{aligned} \tag{9.6}$$

Für Spezialfälle, in denen die RGB-Source- (S_r, S_g, S_b) und RGB-Destination-Faktoren (D_r, D_g, D_b) unabhängig von den Alpha-Source- (S_a) und Alpha-Destination-Faktoren (D_a) festgelegt werden sollen, existiert noch der OpenGL-Befehl `glBlendFuncSeparate(srcRGB,`

`dstRGB, srcA, dstA)` (als Alternative zu `glBlendFunc(srcRGBA, dstRBGA)`, bei dem alle 4 Werte RBG und A einheitlich festgelegt werden). Die möglichen Werte der Argumente dieser beiden Befehle (`srcRGBA, dstRBGA, srcRGB, dstRGB, srcA, dstA`) werden in der folgenden Tabelle zusammengefasst:

Symbolische Konstanten für (`srcRGBA, dstRBGA srcRGB, dstRGB, srcA, dstA`)	**relevanter Faktor**	**Misch-Faktoren** (S_r, S_g, S_b, S_a) **oder** (D_r, D_g, D_b, D_a)
GL_ZERO	Source oder Destination	$(0, 0, 0, 0)$
GL_ONE	Source oder Destination	$(1, 1, 1, 1)$
GL_DST_COLOR	Source	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	Destination	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	Source	$(1, 1, 1, 1) - (R_d, G_d, B_d, A_d)$
GL_ONE_MINUS_SRC_COLOR	Destination	$(1, 1, 1, 1) - (R_s, G_s, B_s, A_s)$
GL_SRC_ALPHA	Source oder Destination	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	Source oder Destination	$(1, 1, 1, 1) - (A_s, A_s, A_s, A_s)$
GL_DST_ALPHA	Source oder Destination	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	Source oder Destination	$(1, 1, 1, 1) - (A_d, A_d, A_d, A_d)$
GL_SRC_ALPHA_SATURATE	Source	$(f, f, f, 1); f = min(A_s, 1 - A_d)$
GL_CONSTANT_COLOR	Source oder Destination	(R_c, G_c, B_c, A_c)
GL_ONE_MINUS_CONSTANT_COLOR	Source oder Destination	$(1, 1, 1, 1) - (R_c, G_c, B_c, A_c)$
GL_CONSTANT_ALPHA	Source oder Destination	(A_c, A_c, A_c, A_c)
GL_ONE_MINUS_CONSTANT_ALPHA	Source oder Destination	$(1, 1, 1, 1) - (A_c, A_c, A_c, A_c)$

Die letzten vier symbolischen Konstanten (`GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`, `GL_CONSTANT_ALPHA`, `GL_ONE_MINUS_CONSTANT_ALPHA`) können nur genutzt werden, wenn das sogenannte „*Imaging Subset*" von der genutzten OpenGL-Implementierung unterstützt wird (ab OpenGL Version 1.3). In diesen Fällen können die Source- und Destination-Mischfaktoren direkt mit Hilfe des OpenGL-Befehls `glBlendColor(`R_c, G_c, B_c, A_c`)` eingestellt werden.

Die vier Komponenten der resultierenden Mischfarbe (R_m, G_m, B_m, A_m) werden nach der Farbmischung auf das Intervall zwischen 0.0 und 1.0 beschränkt, indem größere Werte als 1.0 auf 1.0 und kleinere Werte als 0.0 auf 0.0 gesetzt werden. Die Misch-Faktoren, bei denen A_d vorkommt, benötigen zusätzliche Hardware, um die Alpha-Werte der Destination-Pixel zu speichern (die sogenannten „*Destination Alpha Bitplanes*").

9.3.2 Beispiele für Farbmischungen

Mit den vier Befehlen `glBlendEquation()`, `glBlendFunc()`, `glBlendFuncSeparate()` und `glBlendColor()` sowie ihren Argumenten ist eine unglaubliche Vielfalt an Kombinationsmöglichkeiten für die Farbmischung gegeben. Doch nicht alle Kombinationen von Source- und Destination-Faktoren führen zu sinnvollen Ergebnissen. Damit man hier nicht die Übersicht verliert, werden im Folgenden an einigen Beispielen die am häufigsten genutzten Varianten der Farbmischung vorgeführt.

9.3.2.1 Primitive Farbmischung

Die einfachste Farbmisch-Funktion ist diejenige, bei der überhaupt keine Farbmischung auftritt:

```
glBlendEquation(GL_FUNC_ADD);       // Standard-Einstellung
glBlendFunc(GL_ONE,GL_ZERO);        // Standard-Einstellung
```

In diesem Fall sind alle Oberflächen opak (undurchsichtig) und die Berechnung der neuen Farbwerte ist trivial:

$$\begin{aligned} R_m &= R_s \\ G_m &= G_s \\ B_m &= B_s \\ A_m &= A_s \end{aligned} \tag{9.7}$$

Alpha-Blending mit dieser speziellen Mischfunktion liefert das gleiche Ergebnis, das auch mit deaktiviertem Alpha-Blending erzielt wird, es ist nur etwas langsamer.

9.3.2.2 Klassische Farbmischung

Die am häufigsten verwendete Farbmisch-Funktion verwendet ausschließlich die Alpha-Werte der neu hinzukommenden Fragmente, d.h. von Source. Bei dieser Einstellung der

Mischfunktion erhält die Alpha-Komponente der Source-Farbe die ursprüngliche Bedeutung eines Transparenz-Wertes. Damit lässt sich z.B., wie eingangs erwähnt, die Farbmischung zwischen halbtransparenten Oberflächen und dem Hintergrund sehr realistisch nachbilden. Die konkrete Realisierung eines Milchglases, das 30% des ankommenden Lichts durchlässt (d.h. Opazität A = 1.0 − 0.3 = 0.7), so dass man eine Farbmischung aus 70% Weißanteil des Glases und 30% Anteil der Hintergrundfarbe sieht, stellt sich in OpenGL folgendermaßen dar (Bild 9.2):

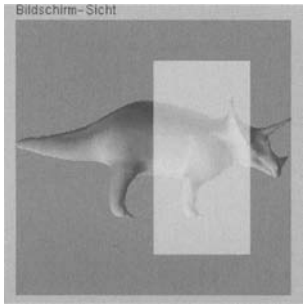


Bild 9.2: Farbmischung zur Darstellung von Transparenz in OpenGL: im linken Teil des Bildes ist nur ein opakes (undurchsichtiges) Objekt dargestellt, im rechten Teil des Bildes befindet sich davor noch ein halbtransparentes, weißes Glas (R = 1.0, G = 1.0, B = 1.0, A = 0.7).

```
glBlendEquation(GL_FUNC_ADD);       // Standard-Einstellung
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
```

Die Berechnung der neuen Farbwerte mit dem konkreten Source-Alpha-Wert $A_s = 0.7$ ergibt sich zu:

$$\begin{aligned} R_m &= A_s \cdot R_s + (1 - A_s) \cdot R_d = 0.7 \cdot R_s + 0.3 \cdot R_d \\ G_m &= A_s \cdot G_s + (1 - A_s) \cdot G_d = 0.7 \cdot G_s + 0.3 \cdot G_d \\ B_m &= A_s \cdot B_s + (1 - A_s) \cdot B_d = 0.7 \cdot B_s + 0.3 \cdot B_d \\ A_m &= A_s \cdot A_s + (1 - A_s) \cdot A_d = 0.7 \cdot A_s + 0.3 \cdot A_d \end{aligned} \tag{9.8}$$

Mit Hilfe dieser Farbmisch-Funktion kann man eine Vielzahl an optischen Phänomenen erzeugen, wie z.B. den Blick durch Fenster, Wasseroberflächen, Glasflaschen (ohne Brechung) oder auch schnell drehende Rotorblätter eines Helicopters, die Überlagerung zweier Bilder, das langsame Überblenden zwischen zwei Objekten durch einen zeitlichen Anstieg des Source-Alpha-Wertes (*Fading*) oder die Realisierung der Sprühdosen-Funktion in einem Zeichenprogramm.

9.3.2.3 Additive Farbmischung

Bei der additiven Farbmischung werden die Farben der Einzelbilder komponentenweise addiert. Die OpenGL-Befehle dafür lauten:

```
glClearColor(0.0,0.0,0.0);          // Hintergrundfarbe schwarz
glBlendEquation(GL_FUNC_ADD);       // Standard-Einstellung
glBlendFunc(GL_ONE,GL_ONE);
```

Dies entspricht der klassischen Farbmischung in Abschnitt 9.3.2.2, aber mit einem Alpha-Wert von 1.0. In mathematischen Formeln dargestellt sehen die obigen OpenGL-Befehle folgendermaßen aus:

$$\begin{aligned} R_m &= R_s + R_d \\ G_m &= G_s + G_d \\ B_m &= B_s + B_d \\ A_m &= A_s + A_d \end{aligned} \tag{9.9}$$

Mit dieser Methode ist Bild 9.3-a entstanden, in dem die additive Farbmischung mit Hilfe von überlagerten Farbkreisen erklärt wird:

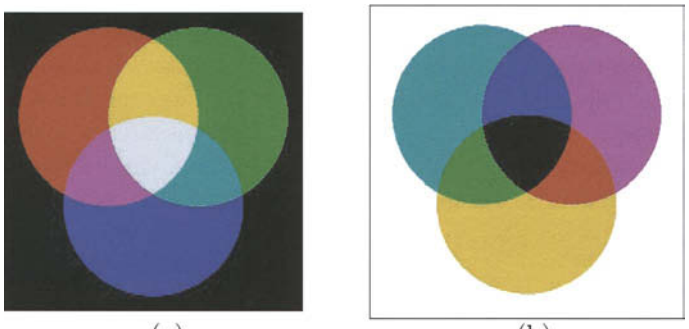
- Rot + Grün = Gelb
  (1,0,0) + (0,1,0) = (1,1,0)
- Rot + Blau = Magenta
  (1,0,0) + (0,0,1) = (1,0,1)
- Grün + Blau = Cyan
  (0,1,0) + (0,0,1) = (0,1,1)
- Rot + Grün + Blau = Weiss
  (1,0,0) + (0,1,0) + (0,0,1) = (1,1,1)

9.3.2.4 Mehrfach-Farbmischung

Es ist auch möglich, die Farben mehrerer transparenter Flächen mit bestimmten Gewichten zu mischen. Als einfaches Beispiel werden im Folgenden drei Bilder zu gleichen Anteilen überlagert:

```
glClearColor(0.0,0.0,0.0);          // Hintergrundfarbe schwarz
glBlendEquation(GL_FUNC_ADD);       // Standard-Einstellung
glBlendFunc(GL_SRC_ALPHA,GL_ONE);
```

Jedes der drei Bilder muss jetzt mit einem Alpha-Wert A = 0.33 gerendert werden. Wenn als Platzhalter für die vier Farbkomponenten des ersten Bildes $\mathbf{g}_1$ dient (entsprechend $\mathbf{g}_2$ für das zweite Bild und $\mathbf{g}_3$ für das dritte Bild), stellt sich die Farbmischung in drei Stufen dar:



(a) (b)

Bild 9.3: Farbmischung dreier transparenter Oberflächen in OpenGL: (a) additive Farbmischung, (b) subtraktive Farbmischung.

- Mischung des ersten Bildes mit der Hintergrundfarbe (schwarz):

$$\mathbf{g}_{m1} = A \cdot \mathbf{g}_1 + 1.0 \cdot \mathbf{g}_d = 0.33 \cdot \mathbf{g}_1 + 1.0 \cdot 0.0 = 0.33 \cdot \mathbf{g}_1 \tag{9.10}$$

- Mischung des zweiten Bildes mit dem ersten:

$$\mathbf{g}_{m2} = 0.33 \cdot \mathbf{g}_2 + 1.0 \cdot \mathbf{g}_{m1} = 0.33 \cdot \mathbf{g}_2 + 0.33 \cdot \mathbf{g}_1 \tag{9.11}$$

- Mischung des dritten Bildes mit den ersten beiden:

$$\mathbf{g}_{m3} = 0.33 \cdot \mathbf{g}_3 + 1.0 \cdot \mathbf{g}_{m2} = 0.33 \cdot \mathbf{g}_3 + 0.33 \cdot \mathbf{g}_2 + 0.33 \cdot \mathbf{g}_1 \tag{9.12}$$

d.h. dass am Ende die jeweilige Farbkomponente jedes einzelnen Bildes ein Drittel zur Gesamtfarbkomponente beiträgt. Dies macht sich natürlich an den Stellen, an denen die einzelnen Bilder nicht überlappen, in einer geringeren Helligkeit bemerkbar.

9.3.2.5 Subtraktive Farbmischung

Bei der substraktiven Farbmischung werden die Farben der Einzelbilder komponentenweise subtrahiert. Die OpenGL-Befehle dafür lauten:

```
glClearColor(1.0,1.0,1.0);          // Hintergrundfarbe weiß
glBlendEquation(GL_FUNC_REVERSE_SUBTRACT);
glBlendFunc(GL_ONE,GL_ONE);
```

In Formeln ausgedrückt heißt das:

$$\begin{aligned} R_m &= R_d - R_s \\ G_m &= G_d - G_s \\ B_m &= B_d - B_s \\ A_m &= A_d - A_s \end{aligned} \tag{9.13}$$

Mit dieser Methode ist Bild 9.3-b entstanden, in dem die substraktive Farbmischung mit Hilfe von überlagerten Farbkreisen erklärt wird. In diesem Fall stellt jeder Farbkreis einen idealen Farbfilter dar, der aus dem weißen Licht des Hintergrunds eine Spektralfarbe herausfiltert und die anderen beiden Spektralfarben durchlässt. Ein ideales Rot-Filter vor einem weißen Hintergrund erscheint deshalb in der Farbe Cyan, ein ideales Grünfilter erscheint in der Farbe Magenta und ein ideales Blaufilter in Farbe Gelb. Das heißt:

- Weiss - Rot = Cyan
  (1,1,1) - (1,0,0) = (0,1,1)
- Weiss - Grün = Magenta
  (1,1,1) - (0,1,0) = (1,0,1)
- Weiss - Blau = Gelb
  (1,1,1) - (0,0,1) = (1,1,0)

Die Farben Gelb, Magenta und Cyan sind die drei Grundfarben für die subtraktive Farbmischung. Die Überlagerung eines idealen Rot- und Grünfilters vor weißem Hintergrund (dies entspricht der subtraktiven Farbmischung von Cyan und Magenta) lässt nur blaues Licht passieren. Die Überlagerung eines idealen Rot- und Blaufilters vor weißem Hintergrund (dies entspricht der subtraktiven Farbmischung von Cyan und Gelb) lässt nur grünes Licht passieren. Die Überlagerung eines idealen Grün- und Blaufilters vor weißem Hintergrund (dies entspricht der subtraktiven Farbmischung von Magenta und Gelb) lässt nur rotes Licht passieren.

- Weiss - Rot - Grün = Blau = Cyan & Magenta
  (1,1,1)- (1,0,0)- (0,1,0) = (0,0,1)
- Weiss - Rot - Blau = Grün = Cyan & Gelb
  ((1,1,1) - (1,0,0) - (0,0,1) = (0,1,0)
- Weiss - Grün - Blau = Rot = Magenta & Gelb
  (1,1,1) - (0,1,0) - (0,0,1) = (1,0,0)
- Weiss - Rot - Grün - Blau = Schwarz = Cyan & Magenta & Gelb
  (1,1,1) - (1,0,0) - (0,1,0) - (0,0,1) = (0,0,0)

9.3.2.6 Farbfilter

Eine alternative Möglichkeit zur Realisierung der subtraktiven Farbmischung und somit von Farbfiltern sieht in OpenGL folgendermaßen aus:

```
glClearColor(1.0,1.0,1.0);          // Hintergrundfarbe weiß
glBlendEquation(GL_FUNC_ADD);
glBlendFunc(GL_ZERO,GL_SRC_COLOR);
```

In Formeln ausgedrückt heißt das z.B. für die Überlagerung eines magentafarbenen Pixels im Bildspeicher (Destination) mit einem neu hinzukommenden gelben Fragment:

$$\begin{aligned} R_m &= R_s \cdot R_d = 1.0 \cdot 1.0 = 1.0 \\ G_m &= G_s \cdot G_d = 1.0 \cdot 0.0 = 0.0 \\ B_m &= B_s \cdot B_d = 0.0 \cdot 1.0 = 0.0 \\ A_m &= A_s \cdot A_d \end{aligned} \tag{9.14}$$

d.h. das Ergebnis der subtraktiven Farbmischung zwischen Magenta und Gelb ist wieder Rot, genau wie bei den geschilderten OpenGL-Befehlen im vorigen Absatz. Der Unterschied zwischen den beiden Varianten besteht zum Einen darin, dass bei der ersten Variante für Source die Komplementärfarben anzugeben sind und sich außerdem die Operation der Subtraktion der Source-Komplementärfarbe von der Operation der Multiplikation mit der Source-Farbe in bestimmten Fällen unterscheidet (Beispiel: $R_s = 0.9, R_d = 0.1$: Variante 1: $R_m = R_d - (1.0 - R_s) = 0.1 - 0.1 = 0.0$, Variante 2: $R_m = R_d \cdot R_s = 0.1 \cdot 0.9 = 0.09$).

9.3.2.7 Direkte Festlegung der Farbmisch-Faktoren

Bei Verwendung der symbolischen Konstanten (`GL_CONSTANT_COLOR`, `GL_ONE_MINUS_CONSTANT_COLOR`) für den Befehl „`glBlendFunc()`“ können die Source- und Destination-Faktoren für die Farbmischung direkt mit Hilfe des OpenGL-Befehls „`glBlendColor(`R_c, G_c, B_c, A_c`)`“ festgelegt werden. Mit der Befehlssequenz

```
glBlendEquation(GL_FUNC_ADD);       // Standard-Einstellung
glBlendFunc(GL_CONSTANT_COLOR,GL_ONE_MINUS_CONSTANT_COLOR);
glBlendColor(1.0, 0.5, 0.0, 0.7);
```

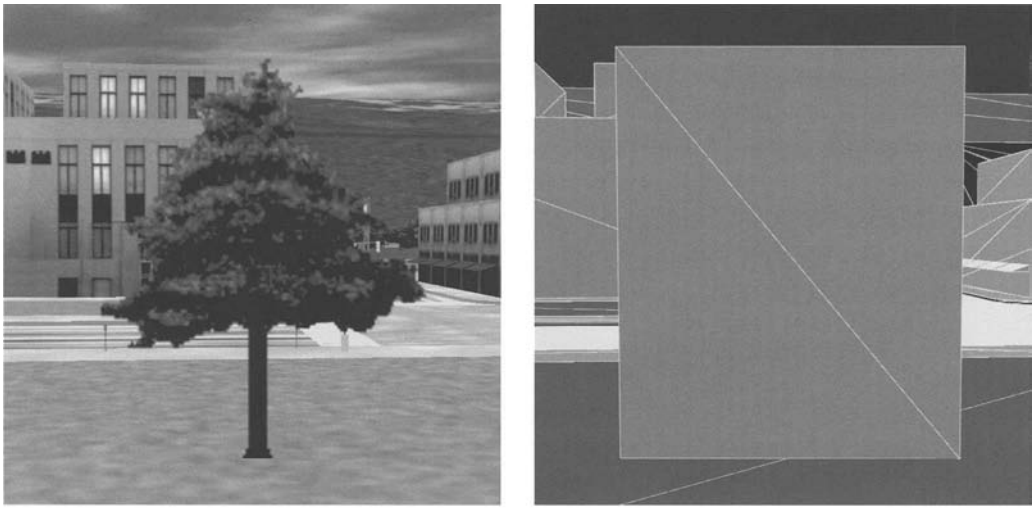
erhält man z.B. eine Farbmischung aus den Source- und Destination-Farben, bei der die Rot-Komponente von Source stammt, die Grün-Komponente zu 50% von Source und Destination, die Blau-Komponente von Destination und die Alpha-Komponente zu 70% von Source und zu 30% Destination. In Formeln:

$$\begin{aligned} R_m &= R_c \cdot R_s + (1.0 - R_c) \cdot R_d = R_s \\ G_m &= G_c \cdot G_s + (1.0 - G_c) \cdot G_d = 0.5 \cdot (G_s + G_d) \\ B_m &= B_c \cdot B_s + (1.0 - B_c) \cdot B_d = B_d \\ A_m &= A_c \cdot A_s + (1.0 - A_c) \cdot A_d = 0.7 \cdot A_s + 0.3 \cdot A_d \end{aligned} \tag{9.15}$$

Dadurch ist es auch möglich, ein Gesamtfarbbild aus drei Einzelfarbbildern zu komponieren, bei dem die Rot-Komponente aus dem ersten Bild stammt, die Grün-Komponente aus dem zweiten Bild und die Blau-Komponente aus dem dritten Bild.

9.3.3 Transparente Texturen

Im Vorgriff auf das *Texture Mapping* (Kapitel 13) werden hier Texturen mit Transparenz-Komponente beschrieben. 2-dimensionale Texturen stellen „Fotos" dar, die auf Polygone quasi „aufgeklebt" werden. Enthalten die Fotos an jedem Bildpunkt nicht nur die drei Farbwerte R,G,B, sondern auch noch einen Alpha-Wert A, kann man auf sehr einfache Weise äußert komplex berandete Objekte, wie z.B. Bäume, Zäune, Personen usw. mit einem einzigen Polygon darstellen. Um diesen Effekt zu erreichen, müssen die Alpha-Werte der Bildpunkte des eigentlichen Objekts in der Textur auf 1.0 (d.h. opak) gesetzt werden und die Alpha-Werte der Bildpunkte des Hintergrunds auf 0.0 (vollkommen durchsichtig). Mit der klassischen Farbmisch-Funktion (`glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA)`) sieht man von dem Polygon, auf das eine so präparierte Alpha-Textur aufgebracht wird, nichts mehr. Im Bild erscheint nur das eigentliche Objekt, so als wäre es quasi mit einer sehr kleinen Schere pixelgenau aus der Textur ausgeschnitten worden (Bild 9.4). Eine Erweiterung dieser Technik durch eine automatische Drehung des Polygons zum Augenpunkt hin, das sogenannte „*Billboarding*", wird im Abschnitt 15.1 beschrieben. Dadurch kann man auch um einen Baum „herumgehen", ohne dass die 2-dimensionale Natur des *Billboards* allzusehr auffällt.



(a) (b)

Bild 9.4: Transparente Texturen in OpenGL ermöglichen die Darstellung komplex berandeter Objekte mit einem einzigen Polygon. (a) Ein Baum als transparente Textur auf einem Viereck. Die Teile des Vierecks, die nicht zum Baum gehören, sind transparent (Alpha = 0) und daher nicht sichtbar. (b) Die selbe Szene wie links, jetzt allerdings ohne *Texture Mapping*. Das Viereck, auf das die transparente Baumtextur gemappt wird, ist jetzt sichtbar und verdeckt den Hintergrund. Die Bilder stammen aus der „town"-Demodatenbasis, die bei dem Echtzeit-Renderingtool „OpenGL Performer" von der Firma SGI mitgeliefert wird.

Um die Funktionalität eines *Billboards* zu erhalten, ist an sich nur eine binäre Entscheidung (ist das Texel transparent oder nicht?) zu treffen. Eine echte Mischung verschiedener Farben, die eine relativ aufwändige Rechenoperation darstellt, ist in diesem Fall gar nicht nötig. Aus diesem Grund stellt OpenGL mit dem „*Alpha-Test*" eine schnelle binäre Entscheidungsfunktion zur Verfügung. Zur Aktivierung des Alpha-Tests wird der OpenGL-Befehl `glEnable(GL_ALPHA_TEST)` aufgerufen. Anschließend wird mit dem Befehl `glAlphaFunc(GL_GREATER, 0.5)` die Vergleichsfunktion (`GL_GREATER`) und der Vergleichswert (hier z.B. `0.5`) festgelegt. In dieser Konstellation wird der Alpha-Wert des neuen Fragments mit dem Vergleichswert (hier 0.5) verglichen. Falls das Vergleichsergebnis positiv ist (d.h. A > 0.5), wird das Fragment weiterbearbeitet, andernfalls eliminiert. Somit erscheinen alle Texel mit einem Alpha-Wert < 0.5 unsichtbar und alle anderen sichtbar. Der Alpha-Test ist dem z-Buffer-Test (Abschnitt 8.2) sehr ähnlich, die zulässigen Vergleichsfunktionen und Vergleichswerte sind identisch und deshalb hier nicht nochmals aufgelistet.

9.3.4 3D-Probleme bei der Farbmischung

Bei der Farbmischung im 3-dimensionalen Raum treten zwei grundsätzliche Probleme auf, die im Folgenden geschildert werden.

9.3.4.1 Zeichenreihenfolge

Die Reihenfolge, in der transparente Polygone mit der klassichen Farbmischfunktion (Abschnitt 9.3.2.2) gezeichnet werden, hat einen entscheidenden Einfluss auf die Mischfarbe. Man betrachte z.B. zwei halbtransparente Flächen mit einem Alpha-Wert von 0.7, d.h. 30% der Hintergrundfarbe scheint durch und 70% der Farbe stammt vom Polygon: das eine Polygon sei grün ((R,G,B,A) = (0.0,1.0,0.0,0.7)) und das andere rot ((R,G,B,A) = (1.0,0.0,0.0,0.7)). Wird im ersten Schritt das grüne Polygon vor weißem Hintergrund gezeichnet, ergibt sich als Mischfarbe ((R,G,B,A) = (0.3,1.0,0.3,0.79)), also ein um 30% aufgehelltes Grün. Im zweiten Schritt wird das rote Polygon vor dem grünen gezeichnet, so dass sich mit Gleichung (9.8) folgende Mischfarbe ergibt: ((R,G,B,A) = (0.79,0.30,0.09,0.73)), d.h. ein starker rot-Ton mit leichtem grün-Stich (Bild 9.5-a). Werden die beiden Polygone in der umgekehrten Reihenfolge gezeichnet, d.h. zuerst das rote und dann das grüne, ergibt sich eine vollkommen andere Mischfarbe als vorher: nach dem ersten Schritt ist ((R,G,B,A) = (1.0,0.3,0.3,0.79)), also ein um 30% aufgehelltes Rot, und nach dem zweiten Schritt (0.30,0.79,0.09,0.73)), d.h. ein starker grün-Ton mit leichtem rot-Stich (Bild 9.5-b). In diesem Beispiel muss der z-Buffer Algorithmus ausgeschaltet sein (`glDisable(GL_DEPTH_TEST)`), wie im nächsten Absatz gleich klar wird.

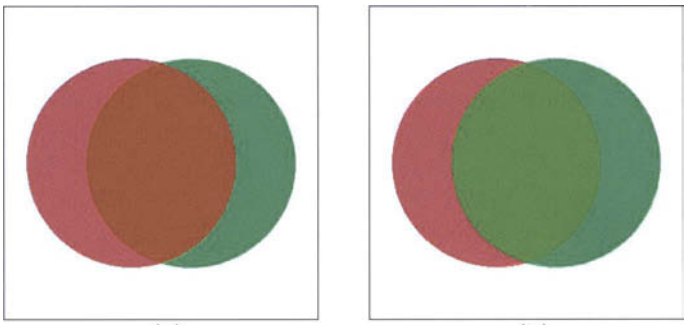


Bild 9.5: Das erste Problem der Farbmischung: das Ergebnis hängt von der Reihenfolge des Zeichnens ab. (a) zuerst wird das grüne Polygon gezeichnet, dann das rote, d.h. in der Mischfarbe überwiegt rot. (b) zuerst wird das rote Polygon gezeichnet, dann das grüne, d.h. in der Mischfarbe überwiegt grün.

9.3.4.2 z-Buffer Algorithmus

Der z-Buffer Algorithmus **A8.1** berücksichtigt keine Alpha-Werte. Wird z.B. ein neues Objekt räumlich hinter einem schon im Bildspeicher befindlichen transparenten Objekt gezeichnet, verhindert der z-Buffer Algorithmus die Farbmischung, da er einfach abbricht, wenn ein neu hinzukommendes Fragment einen größeren z-Wert aufweist als das gespeicherte Pixel (Bild 9.6-b). Will man mehrere opake und transparente Objekte in einer Szene zeichnen, benötigt man aber den z-Buffer Algorithmus, damit all diejenigen Objekte, die hinter einem opaken Objekt liegen, korrekt verdeckt werden. Andererseits sollen Objekte, die hinter einem transparenten Objekt liegen, nicht verdeckt werden.

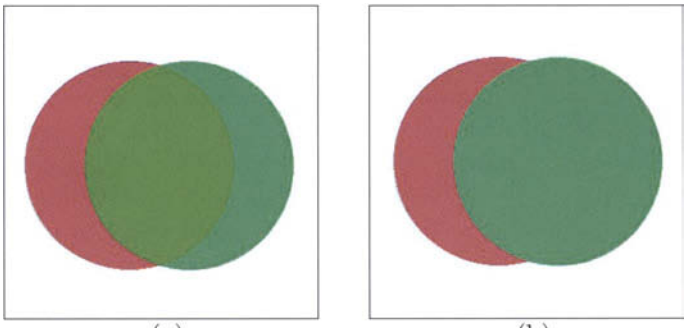


Bild 9.6: Das zweite Problem der Farbmischung: Der z-Buffer Algorithmus berücksichtigt keine Alpha-Werte. (a) zuerst wird das hintere, rote Polygon gezeichnet, dann das grüne, was eine korrekte Farbmischung ergibt, (b) zuerst wird das vordere, grüne Polygon gezeichnet, dann das rote, was dazu führt, dass der z-Buffer Algorithmus die Farbmischung verhindert.

Die Lösung dieser Probleme ist im Algorithmus **A9.1** zusammengefasst. Die Grundidee besteht darin, den z-Buffer Algorithmus für die opaken Objekte einzuschalten und den z-Buffer für das Zeichnen der transparenten Objekte lesbar, aber nicht mehr beschreibbar zu machen (d.h. *read-only*).

A9.1: Pseudo–Code für korrekte Farbmischung bei opaken und transparenten Objekten.

Voraussetzungen und Bemerkungen:

◇ opake Objekte werden von transparenten Objekten separiert.

◇ transparente Objekte werden von hinten nach vorne sortiert, so dass das hinterste transparente Objekt zuerst gezeichnet wird (dieser „Maler-Algorithmus“ ist für eine korrekte Farbmischung wichtig, wenn mehrere transparente Objekte sich überlagern, siehe Abschnitt 9.3.4.1).

Algorithmus:

(a) Einschalten des z-Buffer Algorithmus (`glEnable(GL_DEPTH_TEST)`).

(b) Zeichne alle opaken Objekte (`drawOpaqueObjects()`).

(c) Sorge dafür, dass der z-Buffer nicht mehr beschreibbar ist, aber noch gelesen werden kann, d.h. „*read-only*“ (`glDepthMask(GL_FALSE)`).

(d) Schalte die Farbmischung in OpenGL ein und spezifiziere die Mischfunktion:
```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

(e) Zeichne die von hinten nach vorne sortierten transparenten Objekte (`drawTransparentObjects()`).

(f) Schalte die Farbmischung wieder aus (`glDisable(GL_BLEND)`).

(g) Mache den z-Buffer wieder beschreibbar (`glDepthMask(GL_TRUE)`).

Ende des Algorithmus

Das eher seltene Problem der korrekten Farbmischung von mehreren sich durchdringenden transparenten Objekten wird aber auch durch diesen Algorithmus nicht gelöst, da der „Maler-Algorithmus“ (siehe 2.Voraussetzung) in diesem Fall versagt.

Kapitel 10

Anti-Aliasing

10.1 Aliasing-Effekte

Bei genauerer Betrachtung fast aller bisherigen Bilder dieses Buches fällt auf, dass schräge Linien und Kanten nicht glatt sind, sondern einen sogenannten „Treppenstufen-Effekt" aufweisen (Bild 10.1). Die Ursache dafür liegt in der Natur der digitalen Bildgenerierung

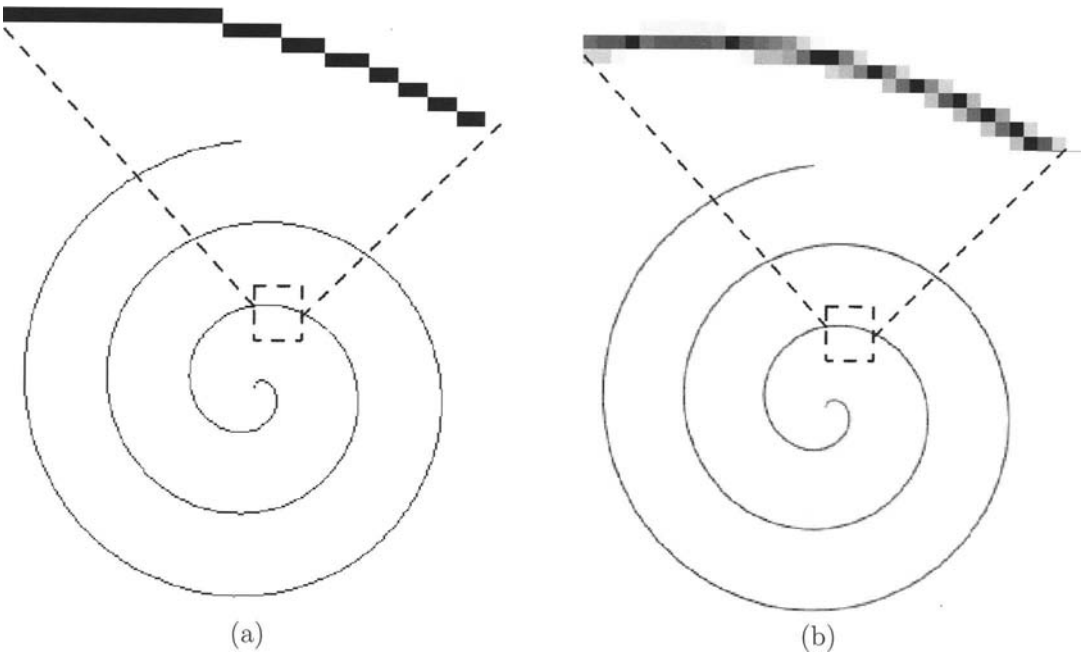


Bild 10.1: Treppenstufen-Effekt und Kompensation durch Anti-Aliasing: (a) Linienzug ohne Anti-Aliasing und darüber eine Ausschnittsvergrößerung (b) Linienzug mit Anti-Aliasing und darüber eine Ausschnittsvergrößerung

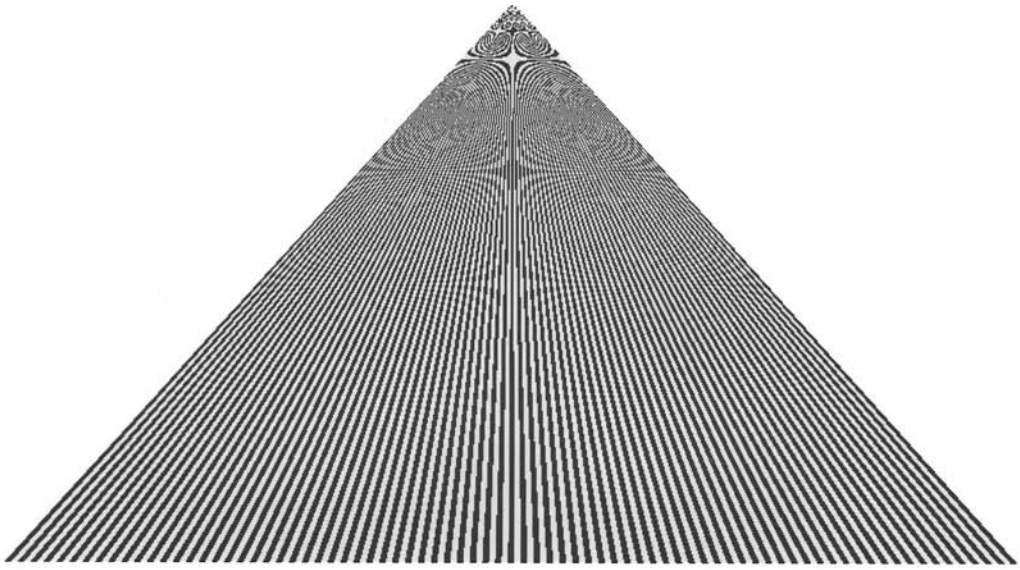
und -verarbeitung selbst: denn ein Bild besteht hier aus einer endlichen Anzahl von Pixeln, die auf einem rechtwinkligen Gitter liegen. Eine ideale schräge Linie kann in diesem Umfeld nur durch Pixel approximiert werden, die auf diesem Raster liegen, so dass die Linie, je nach Steigung mehr oder weniger häufig, von einer Pixelzeile zur nächsten springt. Wird die Linie auch noch langsam bewegt, folgt aus der örtlichen Diskretisierung auch noch eine zeitliche Diskretisierung, d.h. die Linie bewegt sich nicht kontinuierlich über das Pixel-Raster, sondern springt zu bestimmten Zeitpunkten um eine Rasterposition weiter. Zusätzlich richtet unser Wahrnehmungssystem seine Aufmerksamkeit insbesondere auf örtliche und zeitliche Sprünge in Signalen, so dass dieser sogenannte „*Aliasing*"-Effekt sehr störend wirkt.

Mit einem weitereren Aliasing-Effekt hat man sowohl in der Computergrafik beim Texture Mapping (Kapitel 13) als auch in der Bildverarbeitung bei der Modifikation der Ortskoordinaten (Kapitel 22) zu kämpfen: bei der Verkleinerung, Vergrößerung oder (perspektivischen) Verzerrung von Bildern. Wird z.B. eine Bildzeile aus abwechselnd weißen und schwarzen Pixeln perspektivisch verzerrt, d.h. kontinuierlich verkleinert bzw. vergrößert, passen die neuen Rasterpositionen nicht mehr auf das ursprüngliche Raster. Dadurch treten sogenannte „*Moiré*"-Muster auf, dies sind spezielle Aliasing-Effekte bei periodischen Bildsignalen (Bild 10.2).

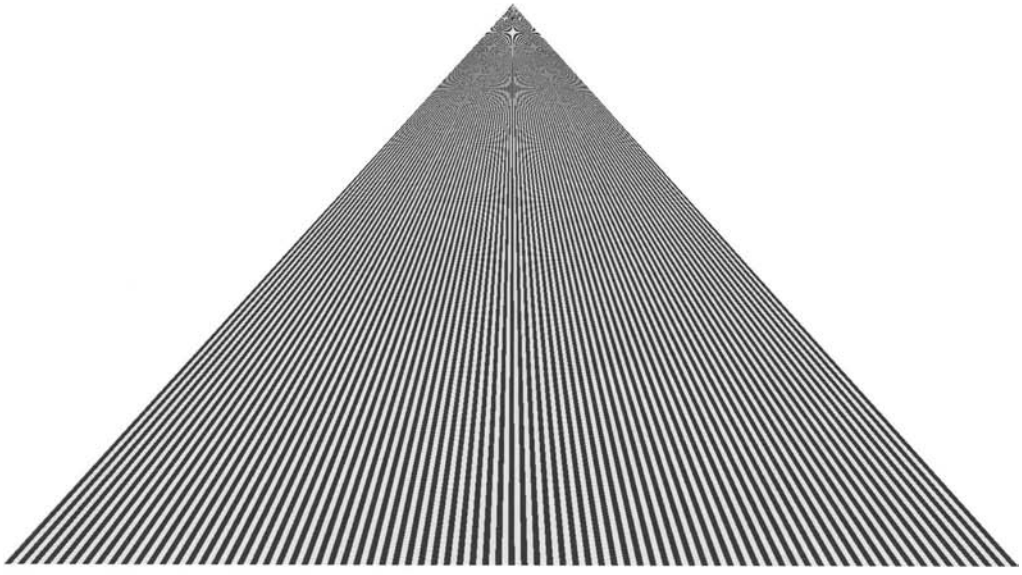
Für ein tieferes Verständnis der Aliasing-Effekte und der Gegenmaßnahmen, d.h. dem „*Anti-Aliasing*", benötigt man die Signaltheorie und hier speziell das Abtasttheorem (eine ausführlichere Darstellung dieser Thematik ist den Kapiteln 16.2, 21 und 28 bzw. in [Fole96] zu finden). Denn die Rasterisierung eines Bildes ist nichts Anderes als die Abtastung eines an sich kontinuierlichen zweidimensionalen Bildsignals an den diskreten Punkten eines rechteckigen Gitters. Das Abtasttheorem besagt nun, dass der Abstand zweier Abtastpunkte höchstens halb so groß sein darf wie die minimale Wellenlänge, damit das kontinuierliche Bildsignal ohne Informationsverlust rekonstruiert werden kann (oder anders ausgedrückt, die Abtastfrequenz muss mindestens doppelt so groß sein wie die höchste Ortsfrequenz, die im Bild vorkommt). Die Bildzeile aus abwechselnd weißen und schwarzen Pixeln in Bild 10.2 ist aber schon an der Grenze des Machbaren, denn ein periodisches Signal mit einer kleineren Wellenlänge als $\lambda_{min} = 2$ Pixel (d.h. $f_{max} = 1/\lambda_{min} = 1/2$ Wellen pro Pixel) ist in einem Digitalbild schlicht nicht darstellbar. Wird dieses Muster dennoch verkleinert, erhöht sich die Ortsfrequenz des Signals f_{sig} über den maximal zulässigen Wert von f_{max} und es kommt unweigerlich zu Abtast-Artefakten (Bild 10.3). Die Ortsfrequenz der Abtast-Artefakte f_{alias} ist gegeben durch:

$$f_{alias} = (f_{sig} - f_{max}) \bmod 1 \tag{10.1}$$

Bei einem Verkleinerungsfaktor von 5/6 (wie in Bild 10.3-a gezeigt), d.h. einer Ortsfrequenz des Signals $f_{sig} = 6/5 f_{max}$, beträgt gemäß 10.1 die Ortsfrequenz des Alias-Effekts $f_{alias} = 1/5 f_{max}$. Bei einem Verkleinerungsfaktor von 2/3 (d.h. $f_{sig} = 3/2 f_{max}$), ergibt sich eine Alias-Ortsfrequenz $f_{alias} = 1/2 f_{max}$ (Bild 10.3-b). Nimmt der Verkleinerungsfaktor linear von 1 bis 1/2 ab, wie z.B. bei der perspektivischen Projektion in Bild 10.2, werden alle Alias-Ortsfrequenzen von 0 beginnend bis f_{max} durchlaufen. Bei einer weiteren Abnahme des Verkleinerungsfaktors von 1/2 bis 1/3, d.h. Signal-Ortsfrequenzen von $2f_{max}$ bis $3f_{max}$,



(a)



(b)

Bild 10.2: Aliasing-Effekt bei Texturen mit abwechselnd weißen und schwarzen Pixeln: (a) 1 Pixel breite parallele Linien, die sich bei perspektivischer Darstellung im Fluchtpunkt treffen. (b) Das gleiche Bild, aber 2-fach höher abgetastet. Die Aliasing-Effekte treten erst weiter hinten auf.

wiederholen sich gemäß der Modulo-Funktion in 10.1 die Alias-Ortsfrequenzen vom vorigen Bereich und damit auch das Moiré-Muster in verkleinerter Ausgabe (Bild 10.3-c). Dies geht immer so weiter, bis das Moiré-Muster letztlich nur noch ein Pixel groß ist. Auch bei einer Vergrößerung der Schwarz/Weiß–Bildzeile ergeben sich trotz Einhaltung des Abtast-Theorems noch Aliasing-Effekte (Bild 10.3-d), die aber bei zunehmender Vergrößerung nicht mehr auffallen.

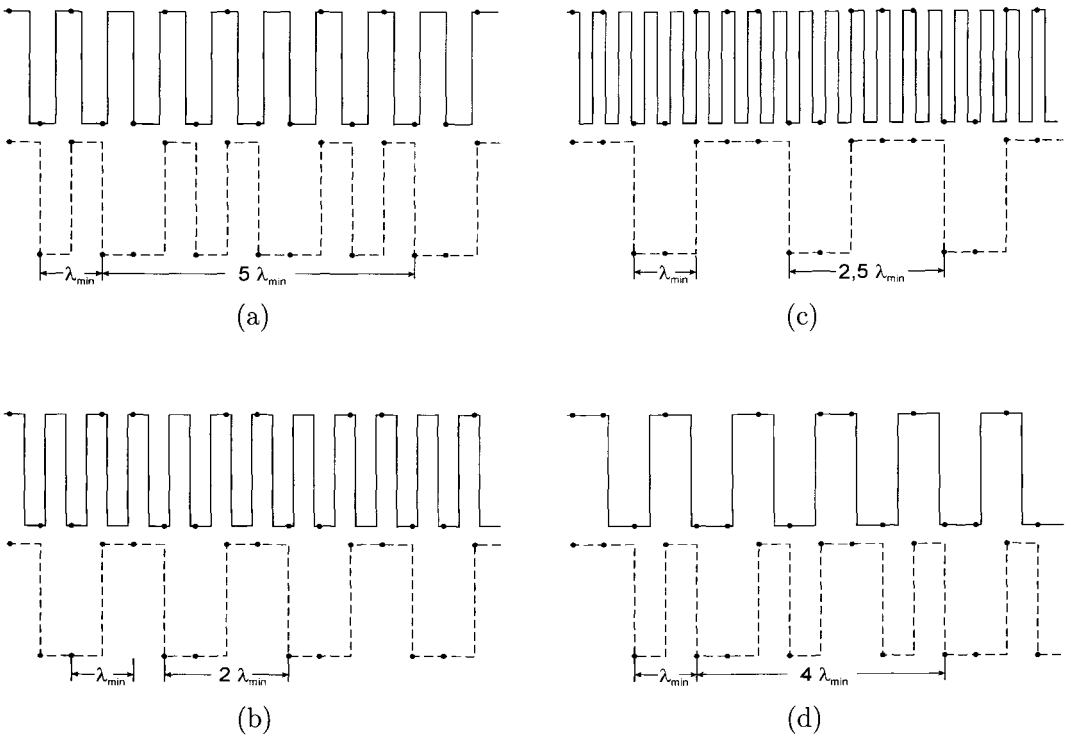


Bild 10.3: Erklärung der Aliasing-Effekte bei digitalen Bildsignalen: in den 4 Grafiken ist nach oben der Grauwert und nach rechts der Ort aufgetragen; im oberen Teil jeder Grafik ist das Eingangssignal durchgezogen dargestellt, im unteren Teil das abgetastete Signal gestrichelt; die Abtastung wird durch dicke Punkte markiert. (a) Signal-Ortsfrequenz $f_{sig} = 6/5 f_{max}$ (bei der gegebenen Abtastrate maximal mögliche Ortsfrequenz). Daraus folgt eine Alias-Ortsfrequenz $f_{alias} = 1/5 f_{max}$ bzw. eine Alias-Wellenlänge von $\lambda_{alias} = 5\lambda_{min}$. (b) Signal-Ortsfrequenz $f_{sig} = 3/2 f_{max}$, daraus folgt eine Alias-Ortsfrequenz $f_{alias} = 1/2 f_{max}$, d.h. $\lambda_{alias} = 2\lambda_{min}$. (c) Signal-Ortsfrequenz $f_{sig} = 12/5 f_{max}$, daraus folgt eine Alias-Ortsfrequenz $f_{alias} = 2/5 f_{max}$, d.h. $\lambda_{alias} = 5/2\lambda_{min}$. (d) Signal-Ortsfrequenz $f_{sig} = 3/4 f_{max}$, daraus folgt eine Alias-Ortsfrequenz $f_{alias} = 1/4 f_{max}$, d.h. $\lambda_{alias} = 4\lambda_{min}$.

10.2 Gegenmaßnahmen – Anti-Aliasing

Der Schlüssel zur Verringerung der Aliasing-Effekte ist die Tiefpass-Filterung (Kapitel 18 und Abschnitt 21.5). Denn die Störeffekte treten ja nur dann bei der Abtastung auf, wenn die höchsten im Bildsignal enthaltenen Ortsfrequenzen in die Nähe der Abtastrate kommen. Die unterschiedlichen Anti-Aliasing-Verfahren unterscheiden sich nur durch den Zeitpunkt der Anwendung des Tiefpass-Filters. Eine Möglichkeit ist der Einsatz eines Tiefpass-Filters vor der Abtastung des Signals (eine „Pre-Filterungs-Methode"), denn dadurch werden die höchsten Ortsfrequenzen eliminiert, so dass die störenden Aliasing-Effekte nicht mehr auftreten. Die zweite Möglichkeit besteht darin, das Bildsignal mit einer höheren Rate abzutasten, so dass die höchsten im Bild vorkommenden Ortsfrequenzen wieder deutlich unter der Abtastrate liegen. Eine höhere Abtastrate bedeutet aber nichts anderes als eine höhere Auflösung des Bildes. Um wieder auf die ursprüngliche Auflösung des Bildes zurück zu kommen, muss das Bild ohne Verletzung des Abtasttheorems verkleinert werden, d.h. tiefpassgefiltert und anschließend unterabgetastet (Kapitel 28). Da in diesem Fall die Tiefpass-Filterung nach der Abtastung erfolgt, spricht man hier von einer „Post-Filterungs-Methode".

10.2.1 Pre-Filterungs-Methode: Flächenabtastung

Das Standard-Anti-Aliasing in OpenGL ist eine Pre-Filterungs-Methode: Punkte und Linien werden standardmäßig als 1 Pixel breit bzw. hoch angenommen und besitzen somit einen gewissen Flächeninhalt (Polygone sowieso). Schräge Linien z.B. bedecken daher bestimmte Pixel zu einem größeren Teil und andere Pixel nur zu einem kleineren Teil (Bild 10.4). Falls der Zustand „Anti-Aliasing" mit einem der Befehle

OpenGL-Befehl	Grafik-Primitiv
`glEnable(GL_POINT_SMOOTH)`	Punkte
`glEnable(GL_LINE_SMOOTH)`	Linien
`glEnable(GL_POLYGON_SMOOTH)`	Polygone

für den jeweiligen Grafik-Primitiv-Typ aktiviert ist, berechnet OpenGL einen Wert für die Bedeckung (*coverage*) eines jeden Pixels durch ein Objekt.

Im RGBA-Modus wird nun der Alpha-Wert eines jeden Pixels mit dem Bedeckungswert multipliziert. Der resultierende Alpha-Wert wird dann benutzt, um die Farbwerte zwischen dem Fragment des neu zu zeichnenden Objektes und dem entsprechenden Pixel, das sich bereits im Bildspeicher befindet, zu mischen. Voraussetzung für die Benutzung der Alpha-Werte zur Farbmischung ist, wie in Kapitel 9 geschildert, die Aktivierung des Alpha-Blendings durch den Befehl „`glEnable(GL_BLEND)`" und die Festlegung einer Farbmischung-Funktion. Am häufigsten wird für das Anti-Aliasing die klassische Farbmisch-Funktion „`glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`" verwendet, denn sie realisiert die Grundidee der Flächenabtastung: bei einem opaken Objekt (d.h. Alpha = 1), dessen Rand ein Pixel z.B. zu 30% bedeckt (d.h. Bedeckungswert = 0.3

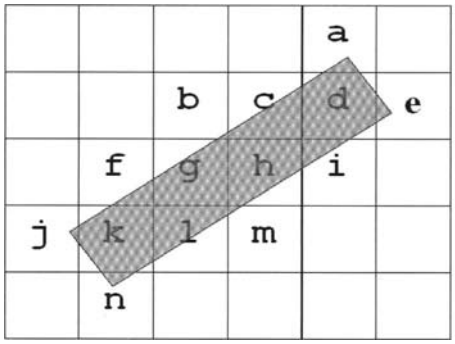
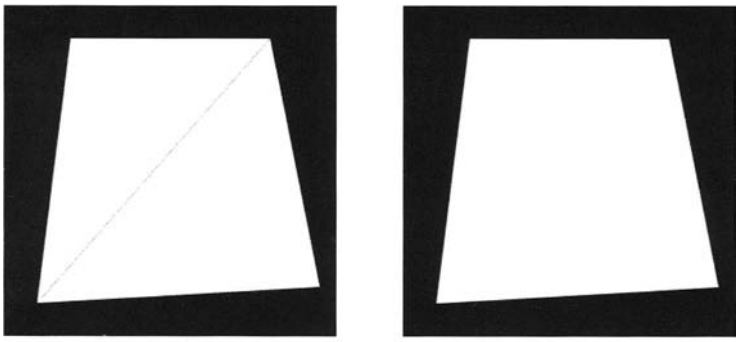


Bild 10.4: Anti-Aliasing in OpenGL: Berechnung eines Bedeckungswertes (*coverage*) für jedes Pixel. Links ist die schräge Linie vor dem Pixelraster zu sehen und rechts die Bedeckungswerte der betroffenen Pixel.

= neuer Alpha-Wert) trägt die Objektfarbe entsprechend dem Flächenanteil des Objekts an dem Pixel 30% (Alpha $\cdot C_s$) und die Hintergrundfarbe 70% ((1-Alpha) $\cdot C_d$) zur Mischfarbe bei. Diese Einstellungen wurden für das Anti-Aliasing der Spirallinie in Bild 10.1-b benutzt.

Die klassische Farbmisch-Funktion führt allerdings beim Anti-Aliasing von Polygonen zu einem gewissen Problem. Denn jedes Polygon mit 4 oder mehr Vertices wird intern in einen Satz von verbundenen Dreiecken zerlegt, die der Reihe nach gerendert werden (dies gilt ebenso für *Quads* oder *Quad_Strips*). Angenommen es wird ein weißes Viereck vor schwarzem Hintergrund gezeichnet. Dann erhalten die Randpixel des ersten Dreiecks im Mittel einen Grauwert von (R = 0.5, G = 0.5, B = 0.5). Die Randpixel des zweiten Dreiecks, die auf der gemeinsamen Kante mit dem ersten Dreieck liegen, treffen jetzt auf den vorher berechneten mittleren Grauwert als Destination-Farbe. Da die Randpixel des zweiten Dreiecks ebenfalls im Mittel einen Bedeckungswert von 0.5 aufweisen, berechnet sich die Mischfarbe zu $C_m = A_s \cdot C_s + (1 - A_s) \cdot C_d = 0.5 \cdot 1.0 + (1.0 - 0.5) \cdot 0.5 = 0.75$, d.h. ein hellerer Grauton. Aus diesem Grund sieht man bei dem weißen Viereck in Bild 10.5-a eine hellgraue Diagonallinie. Abhilfe kann in diesem Fall durch die Verwendung der Farbmisch-Funktion „`glBlendFunc(GL_SRC_ALPHA, GL_ONE)`“ geschaffen werden, bei der die Destination-Farbe zu 100% in die Farbmischung eingeht. Da die Bedeckungwerte für gemeinsame Randpixel vom ersten und zweiten Dreieck komplementär sind, ergänzt sich auch die Mischfarbe wieder zur ursprünglichen Polygonfarbe, und deshalb tauchen hier keine störenden Diagonallinien auf (Bild 10.5-b). Allerdings funktioniert diese Methode nur, wenn sich nicht mehrere Flächen überdecken. Für diesen Fall gibt es als Alternative die Farbmisch-Funktion „`glBlendFunc(GL_SRC_ALPHA_SATURATE, GL_ONE)`“, bei der allerdings die Objekte von vorne nach hinten sortiert werden müssen [Shre05].

Die geschilderten Aliasing-Effekte können sowohl beim Texture-Mapping (Bild 10.2-a) als auch in der Bildverarbeitung bei der Modifikation der Ortskoordinaten mit einem Tiefpass-Filter vor der Abtastung (also eine Pre-Filterungs-Methode) vermindert werden.



(a) (b)

Bild 10.5: Anti-Aliasing bei Polygonen mit 4 oder mehr Vertices: (a) bei der klassischen Farbmischfunktion „`glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`“ entstehen störende Trennlinien innerhalb des Polygons. (b) mit der modifizierten Farbmischfunktion „`glBlendFunc(GL_SRC_ALPHA, GL_ONE)`“ kann man dies vermeiden.

Dazu können unterschiedlich aufwändige Verkleinerungs- und Vergrößerungs-Filter eingesetzt werden, bei denen die Farbwerte eines Pixels als gewichteter Mittelwert aus den Farben der umgebenden Textur-Fragmente (bzw. der umgebenden Originalbild-Pixel) berechnet werden. Eine ausführliche Darstellung der verschiedenen Filter wird in allgemeiner Form in Kapitel 22 gegeben, und die speziellen Textur-Filter, die in OpenGL realisierbar sind, werden in Kapitel 13 beschrieben.

10.2.2 Post-Filterungs-Methoden

10.2.2.1 Nutzung des Accumulation Buffers

Die Grundidee des „*Accumulation Buffer*“-Verfahrens besteht darin, die selbe Szene mehrfach aus minimal unterschiedlichen Blickwinkeln zu rendern, so dass die einzelnen Objekte um den Bruchteil eines Pixels gegenüber dem vorhergehenden Bild verschoben sind. Das endgültige Bild wird durch gewichtete Summation (Akkumulation) der Einzelbilder berechnet. Damit lassen sich beliebige Tiefpass-Filter-Kerne realisieren. Die Implementierung dieser Methode in OpenGL mit Hilfe des „*Accumulation Buffers*“ wird im Folgenden an dem einfachen Fall eines bewegten Mittelwerts als Tiefpass-Filter besprochen.

Der „*Accumulation Buffer*“ ist vom Speicherplatzangebot her eine Kopie des „*Color Buffers*“. Er dient zum Aufsammeln von gerenderten Bildern. Im folgenden Beispiel werden vier Bilder akkumuliert. Nachdem das aus dem ersten Augenpunkt heraus gerenderte Bild im „*Color Buffer*“ steht, werden die Farbwerte der Pixel mit einem Faktor 1/4 gewichtet und in den „*Accumulation Buffer*“ geschrieben. Im nächsten Schritt werden die Pixel des aus dem zweiten Augenpunkt heraus gerenderten Bildes ebenfalls mit dem Faktor 1/4 multipliziert und auf die im „*Accumulation Buffer*“ vorhandenen Werte addiert. Mit den Bildern vom dritten und vierten Augenpunkt wird ebenso verfahren. Am Ende steht im „*Accumulation Buffer*“ ein Bild, dessen Pixel einen Mittelwert aus den vier

leicht verschobenen Einzelbildern darstellen (Bild 10.6). Dieses Verfahren simuliert also eine höhere Abtastrate (hier die doppelte Auflösung), eine anschließende Tiefpass-Filterung mit dem gleitenden Mittelwert und zuletzt eine Auflösungsreduktion (Unterabtastung) um den Faktor 2. Das Verfahren ist mathematisch äquivalent zum „Hardware Anti-Aliasing", das weiter unten besprochen wird. Ein Unterschied besteht darin, das bei der „*Accumulation Buffer*"-Methode vier Einzelbilder sequentiell gerendert werden, was die Bildgenerierrate um einen Faktor vier verlangsamt und im Gegensatz dazu beim „Hardware Anti-Aliasing" ein einziges Bild mit der doppelten Auflösung parallel gerastert wird, so dass bei einer ausreichenden Rasterisierungskapazität die Bildgenerierrate nahezu konstant bleibt. Das „Hardware Anti-Aliasing" ist zwar deutlich schneller, dafür aber weniger flexibel als die „*Accumulation Buffer*"-Methode, da hier sowohl die Anzahl als auch die Position der Abtastwerte vom Software-Entwickler festgelegt werden kann. Da in beiden Fällen die Abtastung vor der Tiefpass-Filterung erfolgt, spricht man von „Post-Filterungs-Methoden".

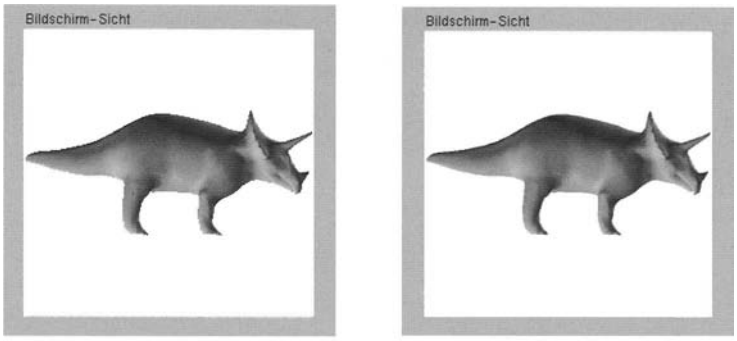


Bild 10.6: Anti-Aliasing mit Hilfe des „Accumulation Buffers": (a) das Objekt ohne Anti-Aliasing. (b) Anti-Aliasing des Objekts mit der „Accumulation Buffer"-Methode.

Der „*Accumulation Buffer*", der genauso ein Teil des Bildspeichers ist wie der „*Color Buffer*" oder der „z-Buffer", wird bei der Initialisierung angelegt. Bei Verwendung der GLUT-Bibliothek lautet der entsprechende Befehl:

```
glutInitDisplayMode(GLUT_ACCUM | GL_RGBA | GLUT_DEPTH).
```

Operationen auf dem „Accumulation Buffer" werden durch den OpenGL-Befehl:

```
glAccum(operation, f)
```

ausgeführt, wobei das Argument „`f`" eine Gleitkommazahl ist, mit der die Farb-Komponenten multipliziert werden und das Argument „`operation`" die in der folgenden Tabelle aufgelisteten Werte annehmen kann:

operation	Formel	Bedeutung
GL_ACCUM	$C'_{acc} = f \cdot C_{col} + C_{acc}$	Addition der mit „f“ skalierten „Color Buffer“-Werte und der „Accumulation Buffer“-Werte
GL_LOAD	$C'_{acc} = f \cdot C_{col}$	Laden der mit „f“ skalierten „Color Buffer“-Werte in den „Accumulation Buffer“, d.h. überschreiben der alten Werte
GL_ADD	$C'_{acc} = f + C_{acc}$	Addition von „f“ auf die RGBA-Werte des „Accumulation Buffer“
GL_MULT	$C'_{acc} = f \cdot C_{acc}$	Multiplikation von „f“ mit den RGBA-Werten des „Accumulation Buffer“
GL_RETURN	$C_{col} = f \cdot C_{acc}$	Kopieren der mit „f“ skalierten „Accumulation Buffer“-Werte in den aktuellen „Color Buffer“

Ein typischer Programm-Ausschnitt, in der die „Accumulation Buffer“-Methode zum Anti-Aliasing angewendet wird, würde folgendermaßen aussehen:

```
int i;
GLdouble left, right, bottom, top, near, far, dx, dy;
GLdouble world_w, world_h, viewport_w, viewport_h;
GLdouble jitter[4][2] = {
                        { 0.25, 0.25 },
                        { 0.25, 0.75 },
                        { 0.75, 0.75 },
                        { 0.75, 0.25 } };

world_w = right - left;
world_h = top - bottom;

for(i=0; i<4; i++) {
    dx = jitter[i][0] * world_w / viewport_w;
    dy = jitter[i][1] * world_h / viewport_h;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(left + dx, right + dx, bottom + dy, top + dy, near, far);
    drawObjects();
    if (i==0)
        glAccum(GL_LOAD, 0.25);
    else
        glAccum(GL_ACCUM, 0.25);
}
glAccum(GL_RETURN, 1.0);
```

10.2.2.2 Hardware Anti-Aliasing

Das Hardware Anti-Aliasing funktioniert im Prinzip genauso wie die „*Accumulation Buffer*"-Methode: die Rasterisierung der Grafik-Primitive und der Texturen wird mit einer höheren örtlichen Auflösung durchgeführt als für den Viewport gefordert wäre. Jedes Pixel wird z.B. in 2 x 2 Subpixel unterteilt und alle Berechnungen für die Farb- und z-Werte werden auf diesem feineren Abtastraster durchgeführt (Bild 10.2-a). Deshalb wird das Hardware Anti-Aliasing auch häufig als „Multi-Sample Anti-Aliasing" oder als „Subpixel Anti-Aliasing" bezeichnet. Die höhere Auflösung erfordert einen zusätzlichen Bildspeicher in der 4-fachen Größe des normalerweise nötigen Bildspeichers und außerdem die 4-fache Rasterisierungsleistung, falls die Bildgenerierrate gleich bleiben soll. Weil die Rasterisierung aber eine sehr gut parallelisierbare Aufgabe ist, kann diese Rechenleistung einfach durch zusätzliche Hardware nahezu ohne Einbußen bei der Renderinggeschwindigkeit erbracht werden. Der endgültige Farbwert eines Pixels wird dann aus dem Mittelwert der 4 Subpixel-Farben berechnet und in den Teil des Bildspeichers geschrieben, der für die Darstellung am Bildschirm bestimmt ist. Da bei dieser Methode die Tiefpass-Filterung, d.h. die Mittelwertbildung, nach der Abtastung, sprich nach der Rasterisierung, erfolgt, zählt das Hardware Anti-Aliasing zu den Post-Filterungs-Methoden.

Eine weitere Qualitätsverbesserung beim Anti-Aliasing kann erreicht werden, wenn die Subpixel-Positionen von Bild zu Bild, d.h. zeitlich zufällig variiert werden. In diesem Fall spricht man von „stochastischem Multi-Sample Anti-Aliasing". Hier betrachtet man z.B. 8 x 8 = 64 Subpixel-Positionen. Pro Bild werden zufällig z.B. 4 Subpixel-Positionen ausgewählt, für die die Farb- und z-Werte berechnet und gemittelt werden. Im zeitlichen Mittel werden dadurch alle 64 Subpixel-Positionen innerhalb von 16 Bildern einmal berechnet. Bei hohen Bildgenerierraten von z.B. 60 Hz und mehr wird jede Subpixel-Position mehrfach pro Sekunde gerendert. Dadurch entsteht der Eindruck eines (örtlichen) Subpixel Anti-Aliasing mit 64-facher Genauigkeit, das allerdings auf Kosten der zeitlichen Auflösung geht.

Falls harte Echtzeit-Anforderungen gelten, bietet das Hardware Anti-Aliasing die beste Qualität, da ein großer Anteil der Farbberechnungen, wie z.B. Transparenz und Nebel, sowie der z-Buffer Algorithmus mit der erhöhten Subpixel-Genauigkeit berechnet werden.

10.2.2.3 Zeitliches Anti-Aliasing

Die bisher betrachteten Aliasing-Effekte sind rein örtliche Phänomene, die durch die Abtastung des Bildsignals in den zwei Raumdimensionen des Bildschirms auftreten. Bei Bewegtbildsequenzen wird allerdings nicht nur örtlich, sondern auch zeitlich abgetastet, da mit der Bildgenerierrate eine Sequenz von „Schnappschüssen" erzeugt wird. Die zeitlichen Aliasing-Effekte können bei rotierenden Objekten erkannt werden, deren Winkelgeschwindigkeit zunimmt. Ein typisches Beispiel sind die Rotorblätter eines startenden Helicopters, die sich zunehmend schneller drehen. Zu Beginn kann man die Zunahme der Winkelgeschwindigkeit noch einwandfrei erkennen. Wenn jedoch die Winkelgeschwindigkeit so groß ist, dass innerhalb zweier aufeinander folgender Bilder das zweite Rotorblatt an die Stelle des ersten getreten ist, scheinen die Rotorblätter zum Stillstand gekommen zu sein,

denn unsere Wahrnehmung kann bei einem rotationssymmetrischen Objekt die einzelnen Drehlagen nicht voneinander unterscheiden (jedes Rotorblatt sieht gleich aus). Bei weiter steigender Winkelgeschwindigkeit wiederholt sich das Spiel wieder von vorne, genau wie bei der örtlichen Abtastung, was durch die Modulo-Funktion in 10.1 ausgedrückt wird. Die geeignete Gegenmaßnahme ist natürlich auch hier wieder die Tiefpass-Filterung, diesmal allerdings in der zeitlichen Dimension. Als Werkzeug bietet sich ebenfalls der „Accumulation Buffer" an, der in diesem Fall nicht örtlich, sondern zeitlich „verschmierte" Bilder aufsammelt. Dadurch wird quasi eine Filmkamera mit einer längeren Blendenöffnungszeit simuliert, so dass schnell bewegte Objekte eine Art „Kondensstreifen" auf den Einzelbildern nach sich ziehen (Bild 10.7). Für diese zeitliche Verschmierung schnell bewegter Objekte hat sich auch bei uns der englische Fachbegriff „*Motion Blur*" eingebürgert.

Bild 10.7: „*Motion Blur*", d.h. zeitliches Verschmieren von bewegten Objekten mit Hilfe des „Accumulation Buffers". Dies entspricht einer Tiefpass-Filterung bzw. einem Anti-Aliasing in der Zeit.

Kapitel 11

Nebel und atmosphärische Effekte

11.1 Anwendungen

Computergenerierte Bilder erscheinen oft deswegen unrealistisch, weil sie viel zu „sauber" sind. In unserer natürlichen Umgebung gibt es dagegen immer eine gewisse Luftverschmutzung durch kleine Staubpartikel oder Wassertröpfchen. Reflektiertes oder abgestrahltes Licht von Oberflächen wird daher auf seinem Weg durch die Luft an den kleinen Verunreinigungen gestreut oder ganz absorbiert. Diese Dämpfung der Lichtintensität führt zu einer Verblassung von weiter entfernten Objekten, die für die menschliche Wahrnehmung von räumlicher Tiefe ein wichtiges Hilfsmittel ist. Abhängig von einer als räumlich konstant angenommenen Partikelkonzentration in der Luft wird pro Längenstück dz, den das Licht zurücklegt, ein bestimmter Prozentsatz A der eingestrahlten Lichtintenstität I absorbiert. Folglich ist die Änderung der Lichtintensität dI/dz proportional zur Partikelkonzentration bzw. zum Absorptionskoeffizienten A und zur eingestrahlten Lichtintensität I, d.h.

$$\frac{dI}{dz} = -A \cdot I \tag{11.1}$$

Die Lösung dieser Differentialgleichung ergibt das Absorptionsgesetz:

$$I(z) = I_0 \cdot e^{-A \cdot z} \tag{11.2}$$

(11.2) besagt, dass das von einer Oberfläche ausgestrahlte Licht exponentiell mit dem Abstand z zum Augenpunkt gedämpft wird. Damit lassen sich eine Reihe von atmosphärischen Effekten, wie z.B. Verblassung entfernter Objekte, Dunst, Nebel, Rauch, Luftverschmutzung usw. beschreiben (Bild 11.1-a).

In natürlichen Umgebungen kommt es aber auch häufiger vor, dass man z.B. auf eine Nebelbank zufährt, d.h. dass man aus einer Position mit geringerer Nebeldichte in eine Position mit größerer Nebeldichte fährt. Eine solche Situation lässt sich durch eine linear ansteigende Nebeldichte recht gut beschreiben. In diesem Fall ist (11.1) also durch einen linearen Faktor z zu ergänzen, d.h.

$$\frac{dI}{dz} = -A \cdot I \cdot z \tag{11.3}$$

Als Lösung dieser Differentialgleichung ergibt sich:

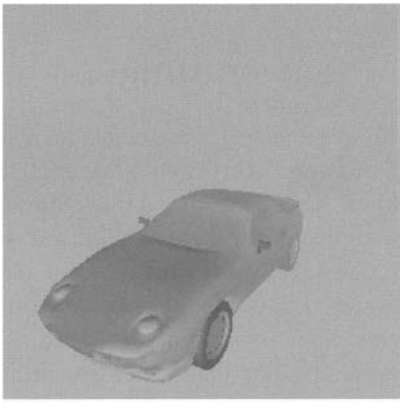
$$I(z) = I_0 \cdot e^{-\frac{A}{2} \cdot z^2} \tag{11.4}$$

(11.4) beschreibt also einen exponentiell quadratischen Abfall der Lichtintensität mit dem Abstand z (Bild 11.1-b).

Weniger physikalisch motiviert, aber in praktischen Anwendungen dennoch relevant ist die lineare Abnahme der Sichtbarkeit einer Oberfläche mit dem Abstand z zum Augenpunkt (Bild 11.1-d):

$$I(z) = I_0(B - C \cdot z) \tag{11.5}$$

(a) GL_EXP



(b) GL_EXP2

(c) ohne Nebel

(d) GL_LINEAR

Bild 11.1: (a) Exponentielle Nebelfunktion. (b) Exponentiell quadratische Nebelfunktion. (c) Ohne Nebel wird das hintere Auto von der *far clipping plane* durchgeschnitten. (d) Mit linearer Nebelfunktion verschwindet das hintere Auto langsam, die *far clipping plane* wird kaum bemerkt.

Damit kann z.B. das schlagartige Verschwinden oder Erscheinen von Objekten beim Durchgang durch die *far clipping plane* (Bild 11.1-c) kaschiert werden, indem beispielsweise ab einer bestimmten Entfernung ein linear ansteigender Nebel beginnt, der bis zur *far clipping plane* auf 100% ansteigt. Dadurch werden die Objekte zum Ende des sichtbaren Bereichs hin langsam im Nebel verschwinden oder langsam aus dem Nebel auftauchen, so dass die Begrenzung der Sichtweite durch die *far clipping plane* kaum mehr auffällt (Bild 11.1-d). Die durch (11.5) beschriebene lineare Abnahme der Sichtbarkeit entspricht einem hyperbolisch zunehmenden Nebeldichteverlauf.

Eine weitere wichtige Eigenschaft des Nebels ist seine Farbe. Während atmosphärischer Nebel je nach Tageszeit und Bewölkung durch unterschiedlich helle Grautöne dargestellt werden kann, lassen sich mit ungewöhnlichen Nebelfarben verschiedene Blendungseffekte simulieren. Ein hellblauer Nebel kann z.B. dazu dienen, den Blendungseffekt bei der Ausfahrt aus einem längeren dunklen Tunnel zu erzeugen (Bild 11.2), und ein gelber Nebel kann genutzt werden, um Sonnenblendung darzustellen.

Bild 11.2: Simulation des Blendungseffekts bei der Ausfahrt aus einem längeren dunklen Tunnel durch hellblauen Nebel

11.2 Nebel in OpenGL

In OpenGL werden all die genannten atmosphärischen Effekte unter dem Begriff „Nebel" (*fog*) zusammengefasst. Die Realisierung von Nebel in OpenGL ist an sich recht einfach, sie baut aber auf zwei wesentlichen Voraussetzungen auf:

- Dem z-Buffer (Kapitel 8), der eigentlich für die Verdeckungsrechnung eingeführt wurde und der die Entfernungen (d.h. die z-Werte) zwischen dem Augenpunkt und den Oberflächen für jedes Pixel enthält. Die gespeicherten z-Werte können jetzt für die Nebelberechnung nach (11.2),(11.4) bzw. (11.5) nochmals genutzt werden.

- Der Farbmischung (Kapitel 9), denn der Nebeleffekt wird dadurch realisiert, dass die Objektfarben mit einer festzulegenden Nebelfarbe in Abhängigkeit von der Entfernung gemischt werden.

Zur Spezifikation der Parameter des Nebels, wie Farbe, Dichte, Koordinaten oder Dämpfungsgleichung dienen die in der folgenden Tabelle dargestellten Varianten des `glFog*`-Befehls:

Skalar-Form	Vektor-Form
`glFogf(GLenum name, GLfloat param)`	`glFogfv(GLenum name, GLfloat *param)`
`glFogi(GLenum name, GLint param)`	`glFogiv(GLenum name, GLint *param)`

Der erste Parameter, `name`, nimmt einen der sechs möglichen Werte aus der folgenden Tabelle an, und der zweite Parameter, `param`, beschreibt den zu übergebenden Wert:

name	param	Bedeutung
GL_FOG_MODE	GL_EXP	exponentiell abnehmende Sichtbarkeit, d.h. konstante Nebeldichte (11.2).
	GL_EXP2	exponentiell quadratisch abnehmende Sichtbarkeit, d.h. zunehmende Nebeldichte (11.4).
	GL_LINEAR	linear abnehmende Sichtbarkeit, d.h. hyperbolisch zunehmende Nebeldichte (11.5).
GL_FOG_COLOR	$\mathbf{g_n} = (R_n, G_n, B_n, A_n)^T$	Zeiger auf eine Nebelfarbe (nur in der Vektor-Form des Befehls möglich)
GL_FOG_DENSITY	a	Nebeldichte bei exp- und exp2-Funktion
GL_FOG_START	z_{start}	Entfernung, bei der ein linear ansteigender Nebel beginnt
GL_FOG_END	z_{end}	Entfernung, ab der ausschließlich die Farbe des linear ansteigenden Nebels zu sehen ist
GL_FOG_COORDINATE_SOURCE	GL_FRAGMENT_DEPTH	Entfernung (z-Wert) des Fragments vom Augenpunkt, wie sie im z-Buffer steht.
	GL_FOG_COORDINATE	z-Wert des Fragments, linear interpoliert aus den mit `glFogCoordfv()` an den Vertices festgelegten z-Werten.

Der Nebeleffekt wird dadurch realisiert, dass die Farbe $\mathbf{g_i} = (R_i, G_i, B_i, A_i)^T$ eines Pixels im Bildspeicher mit einer festzulegenden Nebelfarbe $\mathbf{g_n} = (R_n, G_n, B_n, A_n)^T$ mit Hilfe eines Nebelfaktors f gemäß der folgenden Formel gemischt wird:

$$\mathbf{g_m} = f \cdot \mathbf{g_i} + (1-f) \cdot \mathbf{g_n} \Longleftrightarrow \begin{pmatrix} R_m \\ G_m \\ B_m \\ A_m \end{pmatrix} = f \cdot \begin{pmatrix} R_i \\ G_i \\ B_i \\ A_i \end{pmatrix} + (1-f) \cdot \begin{pmatrix} R_n \\ G_n \\ B_n \\ A_n \end{pmatrix} \tag{11.6}$$

Der Nebelfaktor f wird standardmäßig für jedes Pixel neu berechnet, und zwar nach einer der folgenden drei Gleichungen, die durch den dahinter stehenden OpenGL-Befehl ausgewählt werden:

$$f = e^{-a \cdot z} \qquad \text{bei } \texttt{glFogi(GL\_FOG\_MODE, GL\_EXP);} \tag{11.7}$$

$$f = e^{-(a \cdot z)^2} \qquad \text{bei } \texttt{glFogi(GL\_FOG\_MODE, GL\_EXP2);} \tag{11.8}$$

$$f = \frac{z_{end} - z}{z_{end} - z_{start}} \qquad \text{bei } \texttt{glFogi(GL\_FOG\_MODE, GL\_LINEAR);} \tag{11.9}$$

Durch den Nebelfaktor f in (11.7) wird eine exponentiell abnehmende Sichtbarkeit realisiert, also genau die in (11.1) und (11:2) dargestellte Situation einer räumlich konstanten Nebeldichte (der Nebeldichtewert a in (11.7) ist identisch mit dem Absorptionskoeffizienten A in (11.2). Der Nebelfaktor f in (11.8) realisiert eine exponentiell quadratisch abnehmende Sichtbarkeit, d.h. die in (11.3) und (11.4) dargestellte Situation einer linear zunehmenden Nebeldichte (es ist zu beachten, dass der Nebeldichtewert a in (11.8) nicht identisch ist mit dem Absorptionskoeffizienten A in (11.4), denn es gilt: $a = \sqrt{A/2}$). Durch (11.9) wird eine linear abnehmende Sichtbarkeit realisiert, d.h. die in (11.5) dargestellte Situation einer hyperbolisch zunehmenden Nebeldichte (die Konstanten B und C in (11.5) ergeben sich aus dem Nebelstart- und -endwert zu: $B = z_{end}/(z_{end} - z_{start})$ bzw. $C = 1/(z_{end} - z_{start})$). In Bild 11.3 ist der typische Verlauf des Nebelfaktors f in Abhängigkeit vom z-Wert dargestellt und darunter jeweils die entsprechende Nebeldichte. Generell gilt für (11.7), (11.8) und (11.9), dass der Betrag des z-Wertes eingeht, so dass die Exponentialfunktionen immer abklingend sind und dass der Nebelfaktor f nach seiner Berechnung auf das Intervall [0,1] eingeschränkt wird.

Wie in OpenGL üblich, muss der Zustand „Nebel" eingeschaltet werden, damit die entsprechenden Berechnungen aktiviert werden. Dafür dient der Befehl:

```
glEnable(GL_FOG);
```

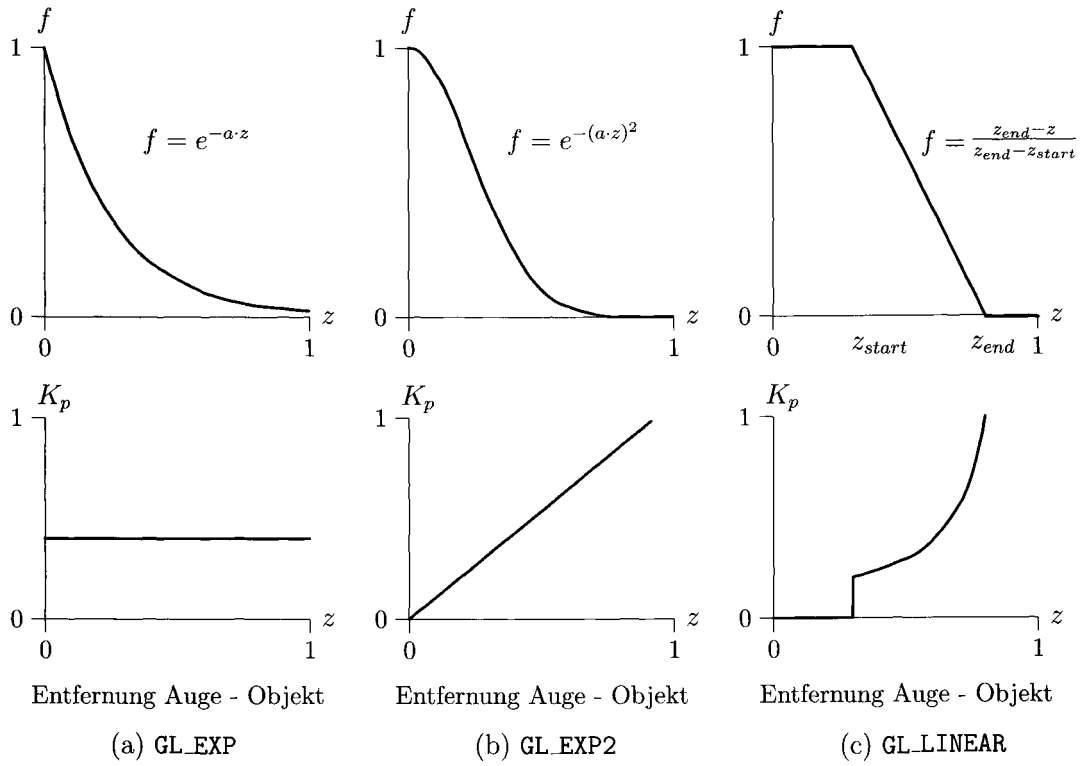


Bild 11.3: Verlauf des Nebelfaktors f in Abhängigkeit vom z-Wert und darunter jeweils die entsprechende Konzentration der Nebelpartikel K_p (a) exponentielle Nebelfunktion – konstante Nebeldichte (b) exponentiell quadratische Nebelfunktion – linear ansteigende Nebeldichte (c) lineare Nebelfunktion – hyperbolisch ansteigende Nebeldichte

Der relevante Ausschnitt eines Programm-Codes zur Aktivierung der exponentiellen Nebelfunktion mit hellgrauer Farbe könnte folgendermaßen lauten (in Bild 11.1-a ist der entsprechende Nebel-Effekt zu sehen):

```
glEnable(GL_FOG);

GLfloat color[4] = {0.7, 0.7, 0.7, 1.0};

glFogi(GL_FOG_MODE, GL_EXP);
glFogfv(GL_FOG_COLOR, color);
glFogf(GL_FOG_DENSITY, 0.4);

glClearColor(color[0], color[1],color[2], color[3]);
                                    // Nebel-Farbe
```

Generell sollte die *clear color*, d.h. die Farbe mit der der Bildspeicher zu Beginn eines neuen Bildes initialisiert wird, identisch sein mit der Nebelfarbe, denn sonst würde ein entferntes Objekt z.B. von einem hellgrauen Nebel eingehüllt, während z.B. ein himmelblauer Hintergrund dazu in ungewöhnlichem Kontrast stünde. Der entsprechende Programm-Code für eine lineare Nebelfunktion mit gelber Farbe könnte folgendermaßen lauten:

```
glEnable(GL_FOG);

GLfloat fogColor[4] = {0.9, 0.9, 0.2, 1.0};

glFogi(GL_FOG_MODE, GL_LINEAR);
glFogfv(GL_FOG_COLOR, fogColor);
glFogf(GL_FOG_START, 2.5);
glFogf(GL_FOG_END, 10.5);

glClearColor(color[0], color[1],color[2], color[3]);
                                    // Nebel-Farbe
```

Ein gewisses Problem bei der Nebelberechnung stellen transparente Oberflächen dar. Denn für eine korrekte Farbmischung bei transparenten Oberflächen wurde als Lösung der Algorithmus **A9.1** vorgeschlagen, bei dem der z-Buffer für transparente Objekte nicht mehr beschrieben, sondern nur noch gelesen werden kann. Folglich stehen im z-Buffer nicht die korrekten z-Werte für die transparenten Objekte, und die Nebelberechnung, in die der z-Wert des jeweiligen Pixels als wesentliche Größe eingeht, kann nicht mehr richtig funktionieren. Eine Lösung dieses Problems ist seit der Einführung von OpenGL Version 1.4 im Jahre 2002 in Form von sogenannten *fog coordinates* (Nebelkoordinaten) möglich. Jedem Vertex kann programmgesteuert eine Nebelkoordinate zugewiesen werden. Dazu dienen die in der folgenden Tabelle dargestellten Varianten des `glFogCoord`-Befehls:

Skalar-Form
`glFogCoordf(GLfloat x,y,z,w)`
`glFogCoordd(GLdouble x,y,z,w)`
Vektor-Form
`glFogCoordfv(GLfloat *vec)`
`glFogCoorddv(GLdouble *vec)`

Damit anstatt der z-Werte im Bildspeicher die zugewiesenen Nebelkoordinaten für die Berechnung des Nebelfaktors f verwendet werden, muss die Quelle für die Nebelkoordinaten mit dem Befehl `glFogi(GL_FOG_COORDINATE_SOURCE, GL_FOG_COORDINATE)` umgesetzt werden. Der Nebelfaktor f, in den die Nebelkoordinaten (d.h. der z-Wert) einfließen, wird für jedes Pixel neu berechnet. Die Nebelkoordinaten sind aber nur an jedem Vertex definiert. Deshalb werden aus den zum jeweiligen Grafik-Primitiv gehörenden vertexbezogenen Nebelkoordinaten durch lineare Interpolation pixelbezogene Nebelkoordinaten berechnet. Erst danach werden die Nebelfaktoren berechnet.

Außerdem sind Nebelkoordinaten sehr hilfreich, um komplexere Nebelmodelle berechnen zu können. Ein typisches Phänomen bei realem Nebel ist die Abhängigkeit der Nebeldichte von der Höhe über der Erdoberfläche, wie z.B. bei Bodennebel (*ground fog*, Bild 11.4) oder höheren Nebelschichten (*layered fog*). Diese Abhängigkeit der Nebeldichte von der Höhe über Grund kann in einem Algorithmus einfließen, mit dem die entsprechenden Nebelkoordinaten berechnet und dann explizit gesetzt werden.

Bild 11.4: Bodennebel: eine Anwendung der OpenGL-*Extension „fog coordinate"*. Quelle: Thomas Bredl.

Kapitel 12

Beleuchtung und Schattierung

Beleuchtung und Schattierung sind wesentliche Elemente, damit computergenerierte Bilder auf einem 2-dimensionalen Bildschirm einen 3-dimensionalen Eindruck beim menschlichen Beobachter hervorrufen. Denn erst durch die Beleuchtung eines Objekts mit einer Lichtquelle und die damit verbundene Abschattung der lichtabgewandten Seiten wird die 3-dimensionale Form des Objekts im Gehirn des Beobachters rekonstruiert. Im Fachjargon nennt man diesen Vorgang „Formwahrnehmung aus Schattierung" (*shape from shading*, [Rama88]). In Bild 12.1-a ist das schon häufig benutzte 3-dimensionale Modell eines Triceratops mit direkter Farbzuweisung (wie in Kapitel 9 erläutert) dargestellt.

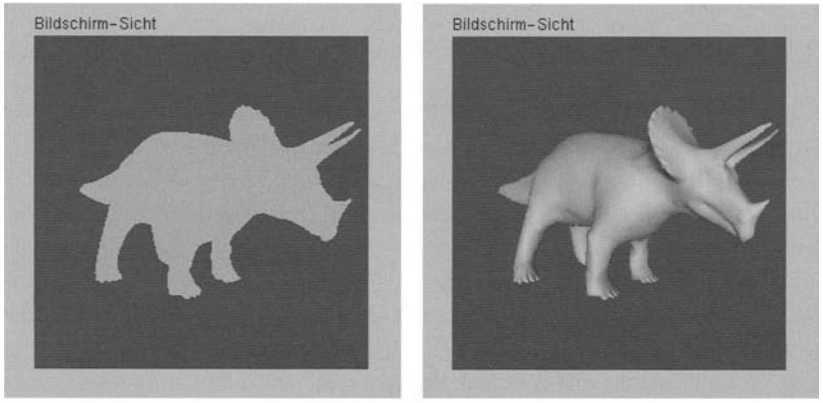


Bild 12.1: Formwahrnehmung aus Schattierung: (a) Direkte Farbzuweisung mit einer einheitlichen Farbe (OpenGL-Befehl `glColor*`): das Modell erscheint flach (b) Berechnung der Farben mit Beleuchtung und Schattierung: der Triceratops erscheint 3-dimensional

Da allen Polygonen des Modells die gleiche Farbe zugewiesen wurde, existieren keine Abschattungen, so dass das Modell vollkommen flach, wie bei einem Scherenschnitt erscheint. In Bild 12.1-b ist das geometrisch gleiche Modell mit Beleuchtung und Schat-

tierung gerendert, so dass ein realistischer Eindruck von der 3-dimensionalen Form des Objekts entsteht.

Die Berechnung der Farbwerte für jedes Pixel läuft in zwei Stufen ab: in der ersten Stufe wird aus den Eigenschaften und Anordnungen von Lichtquellen und Oberflächen mit Hilfe einer Beleuchtungsformel für jeden Vertex eines Polygonnetzes ein Farbwert berechnet; in der zweiten Stufe werden aus den vertex-bezogenen Farbwerten mit Hilfe eines Schattierungsverfahrens die Farbwerte für jedes Fragment bzw. jedes Pixel interpoliert. Nun könnte man fragen, wieso man nicht gleich für jedes Pixel die Beleuchtungsformel auswertet. Der Grund liegt im Rechenaufwand: die Beleuchtungsformel ist relativ komplex, die Standard-Schattierungsverfahren (*Flat-* und *Gouraud-shading*) sind sehr einfach. Also führt man die aufwändige Beleuchtungsformel nur an den relativ wenigen Vertices (1.000 - 100.000) durch und benützt für die Berechnung der großen Anzahl an Pixelfarbwerten (~1.000.000) die schnellen Schattierungsverfahren. In den nächsten beiden Abschnitten werden die verschiedenen Beleuchtungsmodelle und Schattierungsverfahren dargestellt. Das letzte vorgestellte Schattierungsverfahren, das sogenannte *Phong-Shading*, weicht von den Standard-Schattierungsverfahren ab, denn in diesem Verfahren werden nur die Normalenvektoren zwischen den Vertices linear interpoliert, die Beleuchtungsformel wird für jedes Pixel berechnet. Mit der neuesten Hard- und Software, auf die im letzten Abschnitt dieses Kapitels eingegangen wird, ist es möglich, *Phong-Shading* in Echtzeit zu realisieren.

12.1 Beleuchtungsmodelle

12.1.1 Physikalische Optik und Näherungen der Computergrafik

Durch Maxwells Theorie des Elektromagnetismus wurde klar, dass Licht nichts anderes ist als elektromagnetische Wellen. Auf der von Planck und Einstein erkannten Tatsache, dass Licht offenbar auch Teilcheneigenschaften besitzt, beruht die Quantentheorie. Es dauerte eine Weile, bis man die Doppelnatur des Lichts, den sogenannten *Welle-Teilchen-Dualismus*, akzeptiert hatte. Um die Ausbreitung von Licht zu beschreiben, ist das Wellenbild zu benutzen, und um die Wechselwirkung von Licht mit Materie zu beschreiben, ist das Teilchenbild adäquat (Bild 12.2-a). Mit der Quantentheorie des Elektromagnetismus (der sogenannten *Quantenelektrodynamik*, QED) lassen sich alle bekannten Aspekte elektromagnetischer Wellen sowohl auf mikroskospischer als auch auf makroskopischer Ebene mit extrem hoher Präzision vorhersagen ([Nach86]). Allerdings sind viele Effekte im Makroskopischen irrelevant oder sehr subtil, so dass in der Interaktiven 3D-Computergrafik weitreichende Näherungen möglich und zur Reduktion des Rechenaufwandes auch nötig sind.

Die folgenden Näherungen werden in nahezu allen Anwendungen der Computergrafik (incl. Ray Tracing und Radiosity-Verfahren) angewendet. Sie entsprechen, bis auf den ersten Punkt, der geometrischen Optik, die eine makroskopische Näherung darstellt. Das bedeutet, dass die geometrischen Optik nur gilt, wenn die betrachteten Gegenstände sehr groß im Verhältnis zur Wellenlänge des Lichts sind (Bild 12.2-b).

- In homogenen Medien, in denen der Brechungsindex n konstant ist, breiten sich elektromagnetische Wellen geradlinig aus. Man rechnet daher nicht mit Wellenfunktionen, die die elektrische und magnetische Feldstärke der Welle zu jeder Zeit und an jedem Ort festlegen, sondern einfach mit geraden Lichtstrahlen einer bestimmten Intensität.

- Das kontinuierliche Spektrum von elektromagnetischen Wellenlängen wird an drei Stellen abgetastet: Rot, Grün und Blau. Bei allen Berechnungen wird so getan, als bestünde sichtbares Licht nur aus diesen drei monochromatischen Komponenten. Lichtquellen senden quasi nur bestimmte Anteile dieser drei diskreten Wellenlängen aus, und Oberflächeneigenschaften werden durch je drei Komponenten für die Reflexion bzw. Absorption sowie die Transmission dieser Wellenlängen beschrieben. Die menschliche Wahrnehmung merkt von dieser Vereinfachung fast nichts, da im Auge ebenfalls drei verschiedene Rezeptortypen mit Empfindlichkeitsmaxima im Roten, Grünen und Blauen vorhanden sind (Bild 16.11).

- Typische Wellen-Phänomene auf mikroskopischer Ebene, d.h. wenn die betrachteten Gegenstände in der Größenordnung der Wellenlänge oder darunter liegen, sind Interferenz, Beugung und Polarisation. Diese Effekte werden in der geometrischen Optik und in der Computergrafik generell vernachlässigt. Aus diesem Grund können z.B. die schillernden Farben dünner Schichten, wie z.B. bei einem Ölfilm auf Wasser oder bei Schildpatt, nicht realistisch simuliert werden. Polarisationseffekte, die man sich häufig bei Stereoprojektionen zunutze macht, indem man das Bild für das linke Auge z.B. waagrecht linear polarisiert und das Bild für das rechte Auge senkrecht linear polarisiert, so dass die beiden überlagerten Bilder durch eine Brille mit entsprechenden Polarisationsfolien wieder getrennt werden können, werden ebenfalls vernachlässigt.

- Die Wechselwirkung von Licht mit Materie wird stark vereinfacht. Die im Mikroskopischen zum Teil sehr komplexen Vorgänge der Absorption, Streuung und Emission von Licht durch Materie werden im Makroskopischen durch eine geringe Anzahl an Materialkonstanten und Gesetzen beschrieben. Bei Grenzflächen zwischen transparenten und opaken (undurchsichtigen) Medien benötigt man nur das Reflexionsgesetz und die materialabhängigen Anteile an reflektiertem und absorbiertem Licht. An der Grenzfläche zweier transparenter Medien tritt sowohl Reflexion als auch Brechung des einfallenden Lichts auf. Der vom Einfallswinkel abhängige Anteil an reflektiertem bzw. gebrochenem Licht wird durch die Fresnel'schen Gesetze wiedergegeben. Die Abhängigkeit des Brechungsindex eines Materials von der Wellenlänge des Lichts bezeichnet man als *Dispersion*.

Betrachtet man die Wechselwirkung von Licht mit Materie etwas detaillierter, wird die Situation auch in der geometrischen Optik sehr schnell wieder kompliziert. Im Mikroskopischen regen Photonen, die Lichtteilchen, Atome bzw. deren elektrisch geladene Bestandteile, Elektronen und Protonen, zu Schwingungen an. Liegt die Anregungsfrequenz, d.h. die Lichtwellenlänge, an einer Resonanzfrequenz des Materials, wird das Photon *absorbiert* und

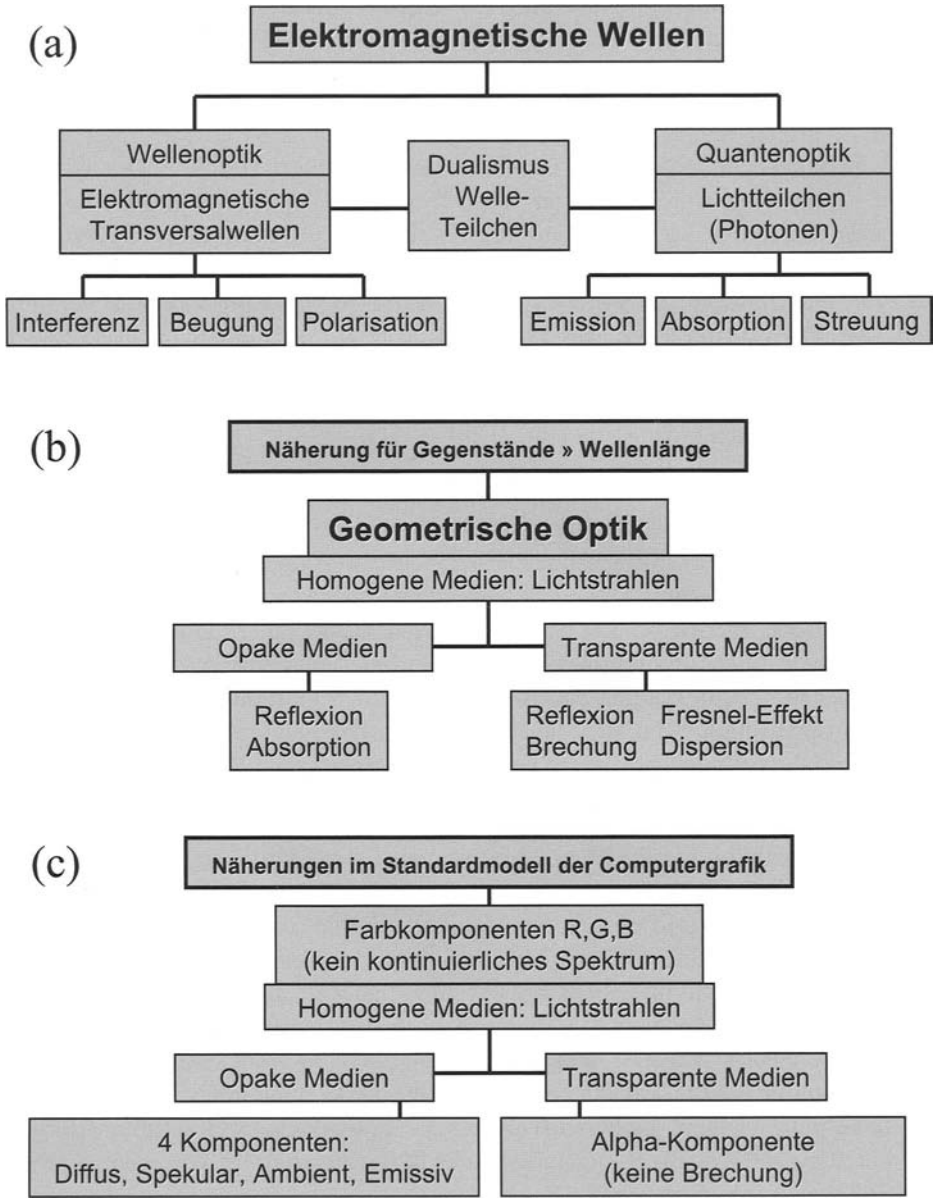


Bild 12.2: Physikalische Optik und Näherungen: (a) Theorie elektromagnetischer Wellen. (b) Geometrische Optik: Makroskopische Näherung. (c) Licht und Beleuchtung im Standardmodell der Computergrafik.

das Atom wechselt in einen energetisch höheren Zustand. Bei anderen Lichtwellenlängen wird das Photon an den Atomen *gestreut*, d.h. es ändert nur seine Flugrichtung. Jedes Atom an der Oberfläche eines Materials wirkt daher wie ein punktförmiges Streuzentrum. Dies entspricht genau dem Huygen'schen Prinzip der Wellenausbreitung, nach dem in jedem Punkt einer Wellenfront ein Streuzentrum sitzt, von dem aus elementare Kugelwellen ausgehen. Die Überlagerung aller elementaren Kugelwellen ergibt eine neue Wellenfront zu einem späteren Zeitpunkt. Damit lässt sich das Reflexions- und das Brechungsgesetz für ebene, ideal glatte Grenzflächen herleiten ([Lang96]). Für beide Gesetze gilt, dass alle relevanten Vektoren in einer Ebene liegen, der Vektor des einfallenden **l**, des reflektierten **r** und des transmittierten Lichts **t**, sowie der auf der Grenzfläche senkrecht stehende Normalenvektor **n**. Der Winkel zwischen dem Normalenvektor **n** und dem Lichtvektor **l** wird Einfallswinkel α genannt, der Winkel zwischen dem Normalenvektor **n** und dem Vektor des reflektierten Strahls **r** wird Reflexionswinkel β genannt, der Winkel zwischen dem negativen Normalenvektor **n** und dem Vektor des transmittierten Strahls **t** wird Brechungswinkel γ genannt. Das *Reflexionsgesetz* (12.1) besagt, dass der Einfallswinkel α gleich dem Reflexionswinkel β ist (Bild 12.3-a).

$$\alpha = \beta \tag{12.1}$$

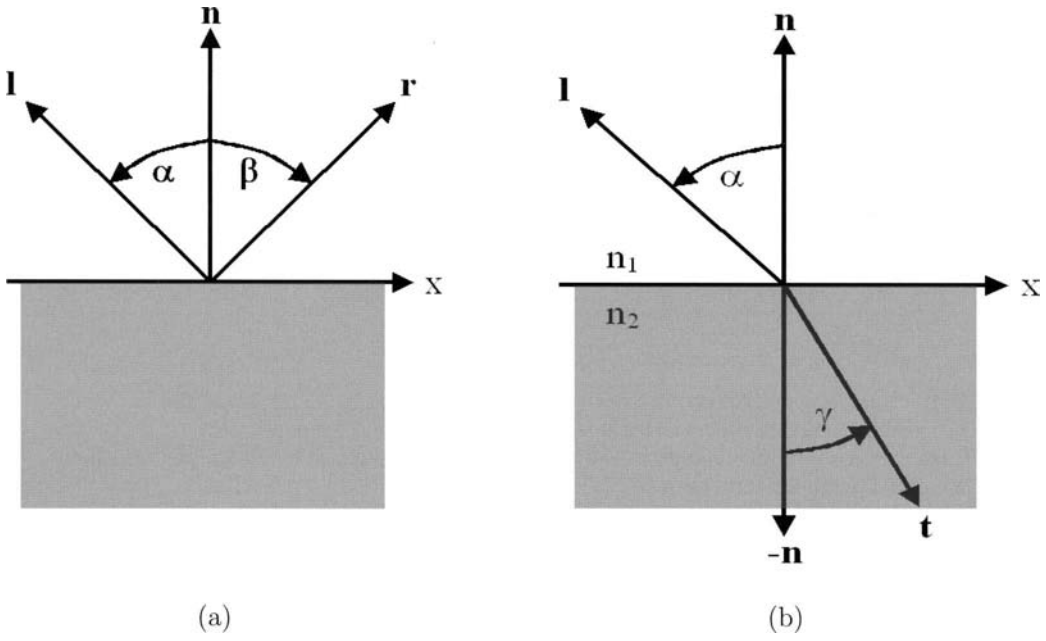


Bild 12.3: Ideal glatte Oberflächen: (a) Reflexion. (b) Brechung.

Das *Brechungsgesetz* (12.2) besagt, dass das Verhältnis zwischen dem Sinus des Einfallswinkels α und dem Sinus des Brechungswinkels γ gleich dem Verhältnis der Brechungsindizes n_1 bzw. n_2 der beiden Medien ist (Bild 12.3-b).

$$\frac{\sin\alpha}{\sin\gamma} = \frac{n_1}{n_2} \tag{12.2}$$

Bei der Brechung von Licht wird allerdings auch im Fall ideal glatter Grenzflächen immer ein gewisser Anteil des Lichts reflektiert (Bild 12.4-a). Je größer der Einfallswinkel α ist, desto geringer wird der Anteil des gebrochenen Lichts und umso größer wird der Anteil des reflektierten Lichts (Bild 12.4-b).

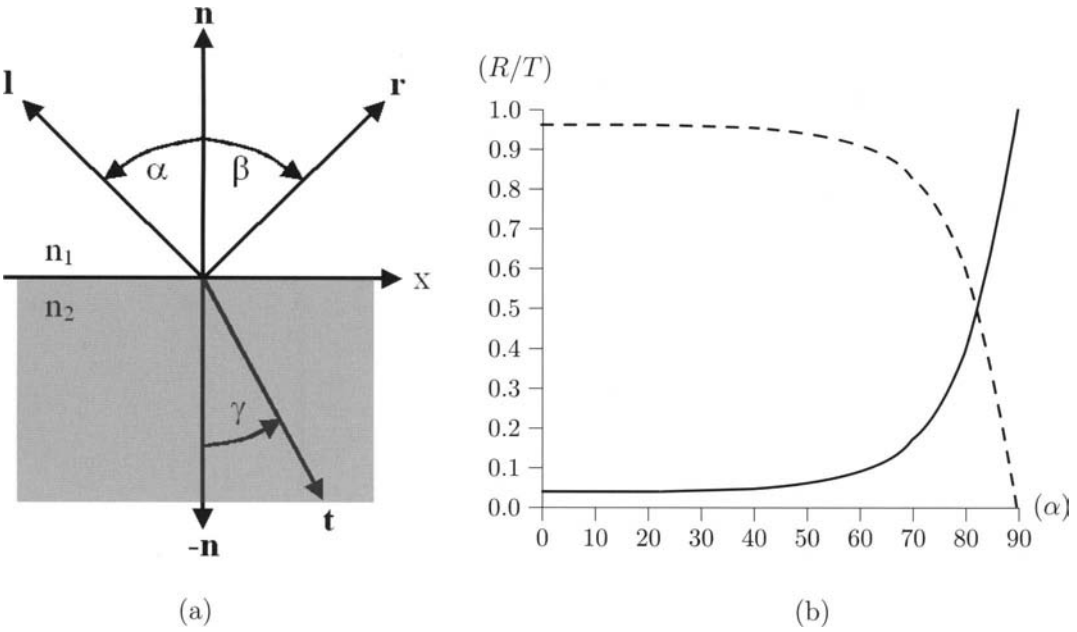


Bild 12.4: Der Fresnel-Effekt: (a) Reflexion und Brechung an einer ideal glatten Oberfläche. (b) Der Anteil des reflektierten (R, durchgezogen) und transmittierten (T, gestrichelt) Lichts in Abhängigkeit vom Einfallswinkel α bei unpolarisiertem Licht und einem Verhältnis der Brechungsindizes von 1,5.

Dieser sogenannte *Fresnel-Effekt* führt dazu, dass sich selbst hochtransparente Flächen, von der Seite gesehen, wie Spiegel verhalten. Für eine genaue Betrachtung des Fresnel-Effekts benötigt man das Wellenbild, denn das Verhältnis zwischen gebrochenem und reflektiertem Licht hängt nicht nur vom Einfallswinkel α ab, sondern auch noch von der Polarisationsrichtung des einfallenden Lichts. Als Näherung geht man in der geometrischen Optik von einem Lichtstrahl mit gleichverteilten Polarisationsrichtungen aus, so dass sich die beiden *Fresnel'schen Gleichungen* für das Reflexionsvermögen R dielektrischer Mate-

rialien zu einer Gleichung (12.3) zusammen fassen lassen:

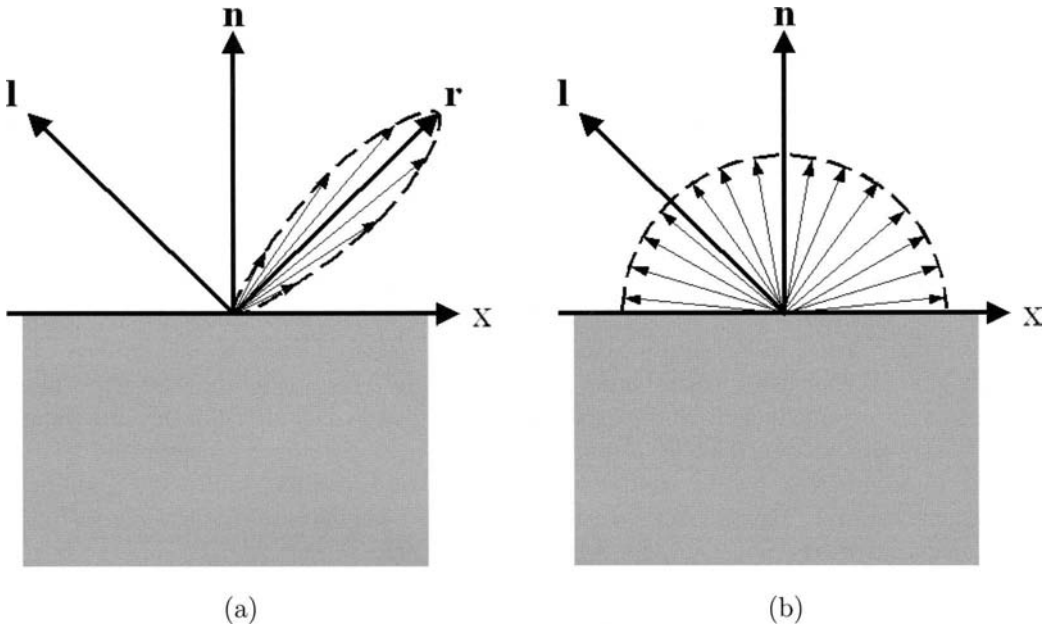
$$R = \frac{1}{2}\left(\frac{\sin^2(\gamma-\alpha)}{\sin^2(\gamma+\alpha)} + \frac{\tan^2(\gamma-\alpha)}{\tan^2(\gamma+\alpha)}\right) \tag{12.3}$$

Bei der Brechung von Licht kommt ein weiterer Effekt hinzu, der die Sache noch komplizierter macht: die Dispersion, d.h. die Abhängigkeit des Brechungsindex n von der Wellenlänge λ. Gemäß (12.2) hängt damit auch der Brechungswinkel γ von der Wellenlänge ab. Dieser Effekt ist wohlbekannt und bei der Auffächerung eines weißen Lichtstrahls durch ein Prisma in seine Spektralfarben eindrucksvoll zu beobachten. Für die Computergrafik bedeutet dieser schöne Effekt, will man ihn denn simulieren, den dreifachen Rechenaufwand. Denn fast die gesamte Beleuchtungsrechnung muss für jeden Vertex bzw. für jedes Pixel mit den drei leicht unterschiedlichen Brechungsindizes für Rot, Grün und Blau neu durchgeführt werden. Mit sehr anspruchsvollen *Environment Mapping* Verfahren (Abschnitt 13.4) sowie programmierbaren Pixel-Shadern und der neuesten Grafik-Hardware kann die Kombination aus Dispersion und Fresnel-Effekt bei transparenten Oberflächen mittlerweile in Echtzeit gerendert werden ([Fern03]).

Wie eingangs erwähnt, besitzen die Atome in einem Material mehrere Resonanzfrequenzen, bei denen Photonen mit der entsprechenden Wellenlänge absorbiert werden. Die Lage der Resonanzfrequenzen im Spektrum, also die atomaren Eigenschaften des Materials, bestimmen die wellenlängenabhängigen Koeffizienten für die Absorption $A(\lambda)$ und die Reflexion $R(\lambda)$ von Licht durch Materie. Aus dem Energieerhaltungssatz folgt, dass die Absorptions- und Reflexionskoeffizienten komplementär sein müssen, d.h. $A(\lambda)+R(\lambda) = 1$. Für die Charakterisierung einer Oberflächenfarbe reicht daher die Angabe eines Koeffizienten, z.B. der Reflexionsfunktion $R(\lambda)$, aus. Aufgrund der spektralen Abtastung an den Stellen Rot, Grün und Blau vereinfacht sich die Reflexionsfunktion zu einem 3-komponentigen Vektor $(R_r, R_g, R_b)^T$. Eine Oberfläche erscheint bei Bestrahlung mit weißem Licht, in dem alle Wellenlängen mit gleicher Intensität vertreten sind, z.B. deshalb als Rot, weil die Reflexionskoeffizienten für Grün und Blau (R_g, R_b) klein sind und nur der Reflexionskoeffizient für Rot (R_r) groß ist.

Die bisher beschriebenen Effekte der geometrischen Optik gelten für ideal glatte Oberflächen. Bei realen Oberflächen, die immer eine gewisse Rauigkeit aufweisen, verkompliziert sich die Sache noch einmal erheblich. Denn durch die Unebenheiten einer realen Oberfläche wird ein einfallender Lichtstrahl aufgespaltet und in verschiedene Richtungen reflektiert. Trägt man die Anteile des Lichts, die in die verschiedenen Raumrichtungen reflektiert werden, in einer Grafik auf, entsteht ein sogenannter *Leuchtkörper* um die ausgezeichnete ideale Reflexionsrichtung (Bild 12.5-a). Bei einer real spiegelnden Oberfläche ist der Leuchtkörper zigarrenförmig, da der größte Teil des reflektierten Lichts in einen engen Raumwinkelbereich um die ideale Reflexionsrichtung gestreut wird. Je glatter die Oberfläche, desto länger und dünner wird der zigarrenförmige Leuchtkörper, d.h. desto näher kommt die reale Spiegelung der idealen. Dieser real spiegelnde Beleuchtungsanteil wird auch als *spekularer* Anteil bezeichnet.

Auf der anderen Seite der Rauigkeitsskala steht das physikalische Modell der *ideal diffusen Reflexion*. Dieses nach ihrem Erfinder genannte *Lambert'sche Beleuchtungsmodell*,



(a) (b)

Bild 12.5: (a) Real spiegelnde Reflexion: ein gewisser Anteil der reflektierten Strahlen weicht von der idealen Reflexionsrichtung ab. Die Einhüllende aller reflektierten Strahlen stellt einen Schnitt durch den zigarrenförmigen Leuchtkörper dar. (b) Ideal diffuse Reflexion nach Lambert: einfallende Lichtstrahlen werden gleichmäßig in alle Raumrichtungen des oberen Halbraums gestreut. Die Einhüllende aller reflektierten Strahlen stellt einen Schnitt durch den halbkugelförmigen Leuchtkörper dar.

beschreibt die gleichmäßige Streuung eines einfallenden Lichtstrahls in alle Richtungen mit gleicher Intensität. Der zugehörige Leuchtkörper ist in diesem Fall eine Halbkugel (Bild 12.5-b). Real diffuse Oberflächen, die dem Lambert'schen Ideal sehr nahe kommen, sind z.B. Löschblätter oder Puderschichten.

Zigarren- und halbkugelförmige Leuchtkörper, die im Standard-Beleuchtungsmodell von OpenGL verwendet werden, sind aber selbst wieder nur idealisierte Spezialfälle. Reale Materialien können beliebig geformte Leuchtkörper aufweisen, die auch noch mit dem Einfallswinkel α variieren. Manche Materialien sind auch noch anisotrop, d.h. sie besitzen eine Vorzugsrichtung, wie z.B. gewalztes Stahlblech oder eine gefräste Oberfläche, die Riefen in einer bestimmten Richtung enthält. In diesen Fällen hängt die Form des Leuchtkörpers auch noch vom Azimutwinkel des einfallenden Lichtstrahls θ ab. Im Allgemeinen ist also der Anteil an Lichtintensität, der in eine bestimmte Raumrichtung reflektiert wird, eine Funktion der Richtung und der Wellenlänge des einfallenden Lichtstrahls. Die Reflexionsfunktion R (*Bidirectional Reflectance Distribution Function*, BRDF) hängt also von fünf Variablen ab: den Elevations- und Azimutwinkeln α und θ des einfallenden Strahls, den Elevations- und Azimutwinkeln β und ϕ des reflektierten Strahls, sowie der Wellenlänge λ des Lichts (Bild 12.6).

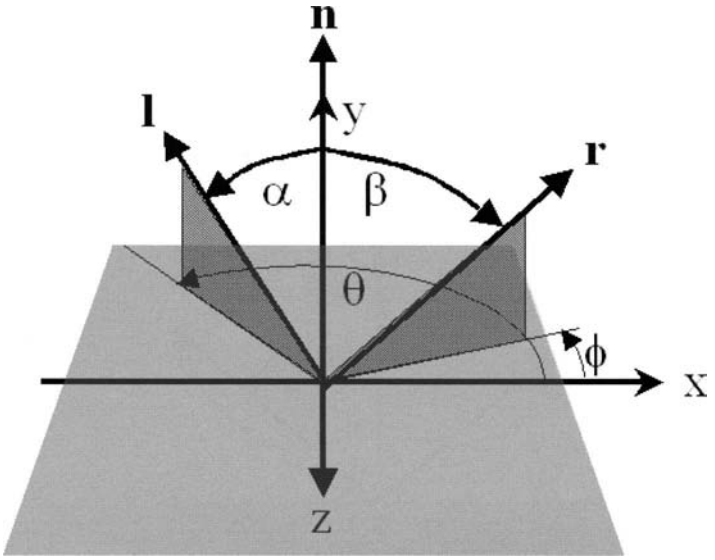


Bild 12.6: Geometrie bei der 3-dimensionalen Beschreibung der Reflexion in der BRDF-Theorie.

Mit einem Leuchtkörper wird die BRDF für eine bestimmte Richtung und Wellenlänge des einfallenden Lichtstrahls visualisiert. Ist die BRDF für ein Material bekannt (z.B. durch Vermessung oder theoretische Herleitung), kann für einen unter bestimmten Winkeln (α, θ) einfallenden Lichtstrahl mit der Intensität I_e die Intensität der in alle Raumrichtungen reflektierten Strahlen I_r berechnet werden:

$$I_r(\beta, \phi, \lambda) = R(\alpha, \beta, \theta, \phi, \lambda) \cdot I_e(\alpha, \theta, \lambda) \cdot \cos(\alpha) \tag{12.4}$$

(12.4) muss wegen der spektralen Abtastung für die Farbkanäle Rot, Grün und Blau drei Mal berechnet werden. Ein einfallender Lichtstrahl aus einer Richtung kann nur von einer Punktlichtquelle stammen. Reale Lichtquellen haben aber immer eine gewisse Ausdehnung, so dass in einem Punkt einer Oberfläche Lichtstrahlen aus allen Raumrichtungen der entsprechenden Hemisphäre eintreffen können. Zur Berechnung der Intensität eines in eine bestimmte Richtung reflektierten Lichtstrahls müssen die Beiträge der einfallenden Lichtstrahlen, die ja aus allen Raumrichtungen der Hemisphäre Ω oberhalb der Fläche stammen, aufintegriert werden. Für ausgedehnte Lichtquellen wird aus (12.4) das folgende Flächenintegral:

$$I_r(\beta, \phi, \lambda) = \int\int_{\Omega} R(\alpha, \beta, \theta, \phi, \lambda) \cdot I_e(\alpha, \theta, \lambda) \cdot \cos(\alpha) \cdot dA(\beta, \phi) \tag{12.5}$$

Vollkommen analog läuft auch die Beschreibung der Transmission von realen, d.h. rauen Oberflächen. Auch in diesem Fall entsteht ein in der Regel zigarrenförmiger Leuchtkörper

um die ausgezeichnete ideale Brechungsrichtung (Bild 12.7). Die Transmission beliebiger Materialien wird durch eine Transmissionsfunktion $T(\alpha, \beta, \theta, \phi, \lambda)$, die sogenannte BTRF (*Bidirectional Transmission Distribution Function*) beschrieben. Eine ausführlichere Darstellung der BRDF-/BTDF-Theorie und ihrer Anwendung in der Computergrafik ist in ([Rusi97]) und ([Aken02]) zu finden.

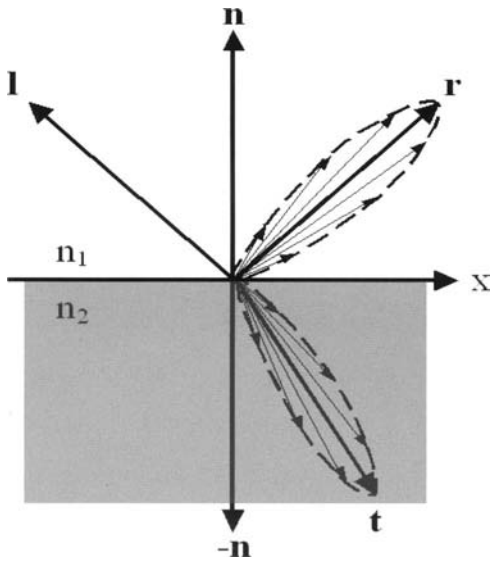


Bild 12.7: Reale Transmission und reale spiegelnde Reflexion: ein gewisser Anteil der transmittierten bzw. reflektierten Strahlen weicht von der idealen Transmissions- bzw. Reflexionsrichtung ab. Die Einhüllende aller transmittierten bzw. reflektierten Strahlen stellt einen Schnitt durch einen zigarrenförmigen Leuchtkörper dar.

Die Emission von Licht durch Materie wird ebenfalls vereinfacht modelliert. Nach dem Plank'schen Strahlungsgesetz emittiert ein idealer schwarzer Körper abhängig von seiner Temperatur ein kontinuierliches Spektrum elektromagnetischer Wellen. Reale Materialien besitzen ein charakteristisches Emissionsspektrum mit verschiedenen Maxima. Die Emissionsmaxima befinden sich bei den Resonanzfrequenzen des Materials, d.h. genau an den Stellen, an denen auch Absorptionsmaxima auftreten. Außerdem existiert bei realen Materialen auch eine richtungsabhängige Abstrahlcharakteristik. Ein Extrembeispiel ist ein Laser, der nur Licht einer Wellenlänge in einen sehr engen Raumwinkelbereich abstrahlt. Wie bereits erwähnt, nähert man in der Computergrafik die kontinuierlichen Emissionsspektren durch drei Koeffizienten für Rot, Grün und Blau an. Die richtungsabhängige Abstrahlcharakteristik wird durch das Modell des Lambert'schen Strahlers ersetzt, der in alle Richtungen des entsprechenden Halbraums die gleiche Lichtintensität ausstrahlt.

12.1.2 Lokale und globale Beleuchtungsmodelle

Trotz aller Vereinfachungen ist die Theorie der geometrischen Optik in der Anwendung auf reale Situationen immer noch extrem komplex: ein konstanter Fluss von Lichtstrahlen aus einer Quelle wird an den Oberflächen einer Szene teilweise absorbiert, in verschiedenste Richtungen reflektiert oder gebrochen. Das heißt, es senden natürlich nicht nur Lichtquellen Licht aus, sondern alle Oberflächen einer Szene. In diesem Punkt unterscheiden sich die lokalen von den globalen Beleuchtungsmodellen. Die lokalen Beleuchtungsmodelle berücksichtigen ausschließlich das Licht von *punktförmigen* Lichtquellen, d.h. sie berechnen die Farbe an jedem Punkt einer Oberfläche mit einer Gleichung vom Typ (12.4). Globale Beleuchtungsmodelle berechnen das Licht, das vom Punkt einer Oberfläche abgestrahlt wird, indem sie das einfallende Licht aus allen Raumrichtungen miteinbeziehen, also sowohl das Licht *ausgedehnter* Lichtquellen, als auch das von anderen Oberflächen reflektierte oder transmittierte Licht. D.h. sie berechnen die Farbe an jedem Punkt einer Oberfläche mit einer Integralformel vom Typ (12.5). Da die Integralformel letztlich numerisch gelöst werden muss, verursachen globale Beleuchtungsmodelle einen um Größenordnungen höheren Rechenaufwand als lokale. Aus diesem Grund wird in der interaktiven 3D-Computergrafik bisher ausschließlich ein lokales Beleuchtungsmodell verwendet, das im nächsten Abschnitt vorgestellt wird.

An dieser Stelle werden nur die Grundideen der beiden Klassen von globalen Beleuchtungsmodellen, den *Ray Tracing*- und den *Radiosity*-Verfahren, erläutert. Eine ausführliche Darstellung aller Varianten und Kombinationen dieser Verfahren findet man in [Watt02]. Aus Aufwandsgründen berücksichtigen diese beiden globalen Beleuchtungsverfahren aber auch wieder nur einen Teil der globalen Wechselwirkung: Ray Tracing beschränkt sich auf ideal glatte Oberflächen, Radiosity auf ideal diffuse Oberflächen.

Der Ray Tracing Algorithmus verfolgt für jedes zu berechnende Pixel den Lichtstrahl rückwärts vom Augenpunkt aus durch dieses Pixel in die Szene zurück, bis es auf ein Objekt trifft. Dies ist der sogenannte *Primärstrahl* (Bild 12.8). Nun tritt der hybride Charakter des Ray Tracing zu Tage: zunächst wird für diesen Punkt ein lokales Beleuchtungsmodell berechnet, in dem nur Punktlichtquellen I_{10} und die spekularen und diffusen Anteile der Oberfläche berücksichtigt werden; anschließend wird das globale Beleuchtungsmodell für ideal glatte Oberflächen berechnet, d.h. nach (12.1) und (12.2) wird die Richtung eines reflektierten $\mathbf{r}_{10}$ und eines gebrochenen Strahls $\mathbf{t}_{10}$ berechnet, um herauszufinden, ob die Lichtstrahlen anderer Oberflächen einen Beitrag (nach dem lokalen Modell) zur Beleuchtung liefern. Dies sind die *Sekundärstrahlen.* Nun kann auch an den sekundären Oberflächenpunkten wieder das globale Beleuchtungsmodell angewendet werden, so dass ein rekursives Strahlverfolgungsverfahren entsteht, dessen Rechenaufwand explodiert. Als Abbruchkriterien verwendet werden daher die Rekursionstiefe, ein vorher festgelegter Mindestwert an Strahlintensität, oder wenn der Strahl die Szene verlässt bzw. auf eine ideal diffuse Oberfläche trifft. Prinzipbedingt liefert das Ray Tracing zwei wesentliche Vorteile gegenüber einem rein lokalen Beleuchtungsmodell: man erhält korrekte Objektspiegelungen und Schatten, da Lichtquellen durch dazwischen liegende Objekte verdeckt werden können. Andererseits sind durch die Beschränkung auf ideal glatte Oberflächen auch Nachteile ver-

bunden: Ray Tracing Bilder sind immer als solche erkennbar, denn Beleuchtungsübergänge erscheinen unnatürlich hart.

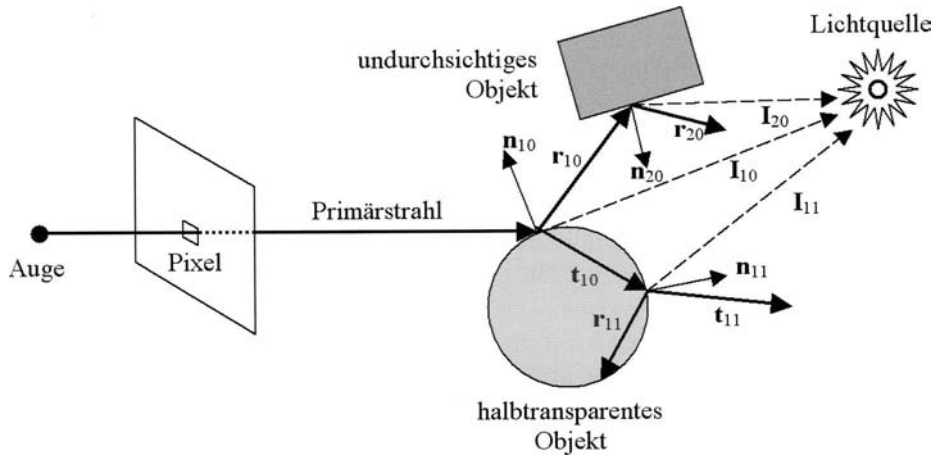


Bild 12.8: Das Prinzip des Ray Tracing

Bei der Radiosity-Methode werden nicht einzelne Strahlen verfolgt, sondern es wird der Strahlungsaustausch zwischen ideal diffusen Oberflächenstücken (Patches oder Polygone) berechnet. Von jedem Flächenelement i in einer Szene geht ein konstanter Lichtstrom L_i (*radiosity*) aus, der sich zusammensetzt aus einem emittierten Lichtstrom E_i (falls das Flächenelement eine Lichtquelle ist) und einem reflektierten Lichtstrom L_r. Der reflektierte Lichtstrom berechnet sich aus der Summe aller einfallenden Lichtströme von den anderen Flächenelementen gewichtet mit dem Reflexionskoeffizienten der Oberfläche R_i. Der Beitrag eines anderen Flächenelements j zum einfallenden Lichtstrom ist einfach dessen Lichtstrom L_j gewichtet mit einem Formfaktor F_{ji}, der sich aus der geometrischen Anordnung der beiden Flächenelemente zueinander ergibt. Zusammengefasst heißt das:

$$L_i = E_i + R_i \cdot \sum_{j=1}^{n} F_{ji} \cdot L_j \tag{12.6}$$

Bei n Flächenelementen muss man also ein System aus n gekoppelten Gleichungen vom Typ (12.6) lösen. Diese ziemlich rechenaufwändige Lösung ist blickwinkelunabhängig. Um eine Ansicht aus einem bestimmten Blickwinkel zu bekommen, benötigt man in einem zweiten Schritt noch ein konventionelles Rendering-Verfahren, in dem eine bestimmte Projektion der Radiosity-Lösung dargestellt wird. Man kann sich daher durch eine statische Szene bewegen, ohne den aufwändigen ersten Schritt, die Lösung des Radiosity-Gleichungssystems, jedes Mal neu berechnen zu müssen. Radiosity-Verfahren sind ideal geeignet, um Innenraumszenen in gedämpftem Licht realistisch darzustellen, denn hier überwiegt der diffus

reflektierende Charakter von Oberflächen, so dass sanfte Beleuchtungsübergänge entstehen. Wegen der Beschränkung auf ideal diffuse Reflexionen können mit der Radiosity-Methode keine spiegelnden Oberflächeneigenschaften dargestellt werden. Da die Vor- und Nachteile des Ray Tracings und der Radiosity-Verfahren komplementär sind, werden sie im Rahmen neuerer Mehr-Wege-Methoden kombiniert.

Durch Anwendung moderner Textur-Mapping-Techniken (Kapitel 13), wie dem *Cube Map Environment Mapping*, dem *Shadow Mapping* oder *projektiven Lichttexturen*, können die Effekte von globalen Beleuchtungsmodellen, wie Spiegelung der Umgebung, Schattenwurf oder auch weiche Beleuchtungsübergänge, sehr häufig in akzeptabler Näherung erzeugt werden.

12.1.3 Das Standard-Beleuchtungsmodell in OpenGL

Die Näherungen der geometrischen Optik gegenüber der mikroskopischen physikalischen Theorie elektromagnetischer Wellen sind für die interaktive 3D-Computergrafik immer noch viel zu rechenaufwändig. Um trotzdem attraktive Bilder in der interaktiven 3D-Computergrafik erzeugen zu können, kommen die folgenden weiteren drastischen Vereinfachungen gegenüber der geometrischen Optik zur Anwendung (Bild 12.2-c):

- Es wird ein lokales Beleuchtungsmodell angewendet, bei dem ausschließlich die Beiträge von punktförmigen Lichtquellen in die Berechnung einfließen. Globale Beleuchtungseffekte, d.h. indirektes Licht, das von anderen Oberflächen reflektiert oder transmittiert wird, werden vernachlässigt. Das Streulicht anderer Oberflächen wird nur durch einen primitiven *ambienten* Term berücksichtigt (Abschnitt 12.1.3.1).

- Das lokale Beleuchtungsmodell berücksichtigt keine Verdeckung von Lichtquellen durch andere Objekte. D.h. es gibt keinen Schattenwurf.

- Das kontinuierliche Spektrum von elektromagnetischen Wellenlängen wird an drei Stellen abgetastet: Rot, Grün und Blau. Diese Vereinfachung gegenüber der geometrischen Optik, die auch schon bei den komplexen globalen Beleuchtungsmodellen zur Anwendung kommt, wird in der interaktiven 3D-Computergrafik unverändert übernommen.

- Atmosphärische Effekte wie die Verblassung und Verblauung von entfernten Objekten, Luftspiegelungen oder der emissive Strahlungsanteil der Luft bleiben in der Beleuchtungsrechnung unberücksichtigt. Ein Teil dieser Effekte wird allenfalls durch Nebel angenähert (Kapitel 11).

- Tranzparenz wird bei der Beleuchtungsrechnung vollkommen vernachlässigt. Erst im Rahmen der in Kapitel 9 beschriebenen Farbmischung (dem *Alpha-Blending*) kann durch Zuordnung eines skalaren Alpha-Werts Transparenz simuliert werden. Das Alpha-Blending kann erst nach der Beleuchtung und Schattierung aller Oberflächen durchgeführt werden. Brechungseffekte und somit auch Dispersion und Fresnel-Effekt

bleiben dabei allerdings unberücksichtigt. Erst durch moderne *Cube Map Environment Mapping* Techniken lassen sich diese Effekte mit Einschränkungen in Echtzeit realisieren (Abschnitt 13.4.2).

- Reale Reflexion und Absorption an undurchsichtigen Materialien, die man in der geometrischen Optik durch 5-dimensionale Reflexionsfunktionen $R(\alpha, \beta, \theta, \phi, \lambda)$ (BRDF-Theorie) annähert, werden ersetzt durch eine gewichtete Kombination aus ideal diffuser Reflexion nach Lambert und eingeschränkter real spiegelnder (spekularer) Komponente nach Phong [Phon75].
  Vorsicht: das Phongsche Beleuchtungsmodell darf nicht verwechselt werden mit dem ebenfalls von Phong stammenden Schattierungsverfahren (Abschnitt 12.2.3).

- Die Emission von Licht wird durch zwei strikt getrennte Methoden realisiert:
  - durch punktförmige Lichtquellen, die ausschließlich andere Oberflächen beleuchten, aber selbst nicht sichtbar sind
  - oder durch Oberflächen, die als Lambertsche Strahler selbst leuchten und daher auch ohne Lichtquelle sichtbar sind, die aber keine anderen Oberflächen beleuchten.

Nun, da der Leser sich bewusst sein sollte über die Einschränkungen und Näherungen des Standard-Beleuchtungsmodells, werden im Folgenden dessen Komponenten und die Programmierung in OpenGL erläutert.

12.1.3.1 Die Beleuchtungskomponenten: emissiv, ambient, diffus und spekular

Das Standard-Beleuchtungsmodell in OpenGL enthält vier Komponenten, die unabhängig von einander berechnet werden:

- die *emissive* Komponente, die den selbstleuchtenden Farbanteil einer Oberfläche darstellt,

- die *ambiente* Komponente, die den Anteil des ungerichteten Streulichts und somit den Ersatz für die globalen Beleuchtungskomponenten darstellt,

- die *diffuse* Komponente, die den Anteil des gerichteten Lichts von einer Punktlichtquelle darstellt, das von einer ideal diffusen Oberfläche in alle Richtungen gleichmäßig gestreut wird (Lambert'sches Beleuchtungsmodell),

- die *spekulare* Komponente, die den Anteil des gerichteten Lichts von einer Punktlichtquelle darstellt, das von einer real spiegelnden Oberfläche hauptsächlich in Richtung des idealen Reflexionswinkels gestreut wird (Phongsches Beleuchtungsmodell).

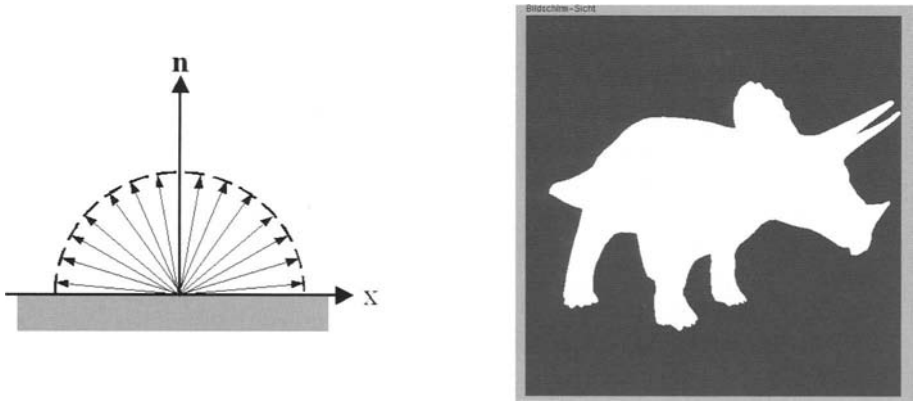
Die Summe dieser vier Beleuchtungskomponenten ergibt die Vertexfarbe:

$$\text{Vertexfarbe} = \text{emissiv} + \text{ambient} + \text{diffus} + \text{spekular} \tag{12.7}$$

Die Berechnung des *emissiven*, d.h. selbstleuchtenden Lichtanteils ist extrem einfach. Da keine Lichtquelle nötig ist, sondern die Oberfläche selbst nach dem Lambert'schen Modell Lichtstrahlen gleichverteilt in alle Richtungen des Halbraums oberhalb der Fläche aussendet (Bild 12.9), wird der emissive Lichtanteil allein durch die emissiven Eigenschaften des Materials festgelegt:

$$\text{emissiv} = \mathbf{e}_{mat} = \begin{pmatrix} R_{\mathbf{e}_{\mathbf{mat}}} \\ G_{\mathbf{e}_{\mathbf{mat}}} \\ B_{\mathbf{e}_{\mathbf{mat}}} \\ A_{\mathbf{e}_{\mathbf{mat}}} \end{pmatrix} \tag{12.8}$$

Die vierte Komponente, d.h. die Alpha-Komponente, spielt in der Beleuchtung keine Rolle, sie wird nur aus Konsistenzgründen mitgeschleift. Beispiele für den Einsatz des emissi-



(a) (b)

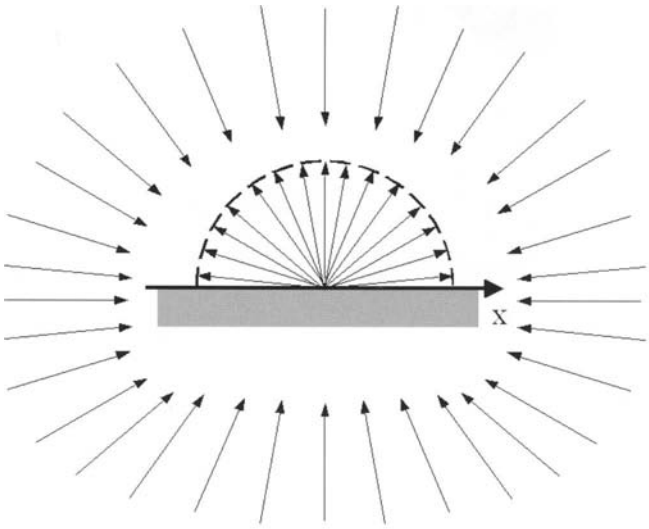
Bild 12.9: Der emissive Lichtanteil: (a) die selbstleuchtende Oberfläche sendet Licht gleichmäßig in alle Richtungen aus. (b) das 3-dimensionale Modell Triceratops erscheint flach, da es ausschließlich mit einem emissiven Lichtanteil gerendert wurde und daher keine Abschattungen entstehen.

ven Beleuchtungsanteils in der Interaktiven 3D-Computergrafik sind Verkehrsampeln oder Leuchtreklame, deren Lichtzeichen unabhängig von der Tageszeit (d.h. einer Lichtquelle) immer sichtbar sind, sowie nachts leuchtende Fensterscheiben, die eine Innenraumbeleuchtung simulieren. Es ist zu beachten, dass emissive Oberflächen in diesem Modell keine anderen Oberflächen beleuchten.

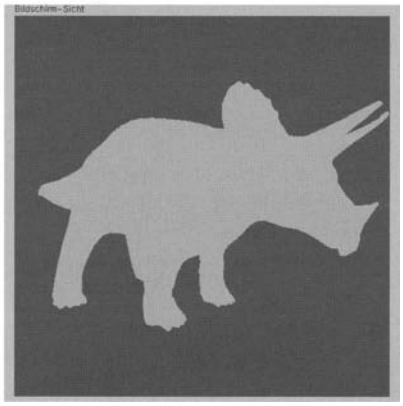
Die Berechnung des *ambienten* Lichtanteils ist ebenfalls sehr einfach. Der ambiente Lichtanteil ist ein extrem vereinfachter Ersatz für die globalen Beleuchtungsanteile, also für das von Lichtquellen stammende Licht, das nicht direkt auf eine Oberfläche fällt, sondern zuerst von anderen Oberflächen gestreut wird. Die Idealisierung besteht jetzt darin,

dass man den gesamten Raum, der die beleuchtete Szene umgibt, als ideal diffus reflektierend annimmt. In diesem Fall trifft aus allen Raumrichtungen die gleiche Intensität an Streulicht ein, d.h. die Position der Lichtquelle ist irrelevant. Die Farbe der (gedachten) Oberflächen, die das Streulicht reflektieren, wird durch eine ambiente Lichtquellenfarbe ersetzt, die unabhängig von der diffusen und spekularen Lichtquellenfarbe festgelegt werden kann. Im Gegensatz zum emissiven Lichtanteil ist also beim ambienten Anteil eine Lichtquelle nötig, damit überhaupt Streulicht vorhanden ist. Dieses ungerichtete Streulicht wird an der betrachteten Oberfläche wieder nach dem Lambert'schen Modell gleichmäßig in alle Richtungen ausgesendet (Bild 12.10-a). Der ambiente Lichtanteil wird folglich durch die ambienten Eigenschaften der Lichtquelle $\mathbf{a}_{light}$ und des Materials $\mathbf{a}_{mat}$ festgelegt (die Operation $*$ bedeutet hier komponentenweise Multiplikation der vier Farbanteile):

$$\text{ambient} = \mathbf{a}_{light} * \mathbf{a}_{mat} = \begin{pmatrix} R_{\mathbf{a}_{\mathbf{light}}} \\ G_{\mathbf{a}_{\mathbf{light}}} \\ B_{\mathbf{a}_{\mathbf{light}}} \\ A_{\mathbf{a}_{\mathbf{light}}} \end{pmatrix} * \begin{pmatrix} R_{\mathbf{a}_{\mathbf{mat}}} \\ G_{\mathbf{a}_{\mathbf{mat}}} \\ B_{\mathbf{a}_{\mathbf{mat}}} \\ A_{\mathbf{a}_{\mathbf{mat}}} \end{pmatrix} = \begin{pmatrix} R_{\mathbf{a}_{\mathbf{light}}} \cdot R_{\mathbf{a}_{\mathbf{mat}}} \\ G_{\mathbf{a}_{\mathbf{light}}} \cdot G_{\mathbf{a}_{\mathbf{mat}}} \\ B_{\mathbf{a}_{\mathbf{light}}} \cdot B_{\mathbf{a}_{\mathbf{mat}}} \\ A_{\mathbf{a}_{\mathbf{light}}} \cdot A_{\mathbf{a}_{\mathbf{mat}}} \end{pmatrix} \tag{12.9}$$



(a)



(b)

Bild 12.10: Der ambiente Lichtanteil: (a) das aus allen Richtungen eintreffende Licht wird an der Oberfläche gleichmäßig in alle Richtungen gestreut. (b) das 3-dimensionale Modell Triceratops erscheint flach, da es ausschließlich mit einem ambienten Lichtanteil gerendert wurde und daher keine Abschattungen entstehen.

Ein geringer Anteil der ambienten Beleuchtung wird in nahezu jeder Situation eingesetzt, um den Anteil des ungerichteten Streulichts zu simulieren. Ohne diesen Anteil

wären abgeschattete Oberflächen vollkommen schwarz, d.h. überhaupt nicht sichtbar, was sehr unnatürlich wirkt. Ein 3-dimensionales Modell, das ausschließlich mit einem ambienten Lichtanteil gerendert wird, erscheint, ebenso wie beim emissiven Lichtanteil, vollkommen flach, da es keine Abschattungen und daher auch keine Anhaltspunkte für die 3D-Formwahrnehmung gibt (Bild 12.10-b).

Die Berechnung des *diffusen* Lichtanteils ist schon deutlich komplizierter. Der diffuse Anteil ist das gerichtete Licht, das von einer Punktlichtquelle aus einer einzigen Richtung kommt und ideal diffus reflektiert wird. Falls die Lichtquelle sehr weit entfernt ist, wie z.B. die Sonne, kommen die Lichtstrahlen praktisch parallel an. Der Vektor vom Vertex zur Lichtquelle heißt Lichtvektor $\mathbf{l}$. Die von einer Lichtquelle auf ein Oberflächenstück A einfallende Lichtintensität hängt von der Orientierung der Oberfläche zur Lichtquelle ab. Bei senkrecht einfallendem Licht wird die maximale Lichtintensität auf die Oberfläche eingestrahlt, je flacher der Einfallswinkel ist, desto niedriger wird die eingestrahlte Lichtintensität. Nur der aus Sicht der Lichtquelle effektive Flächeninhalt des Oberflächenstücks $A_\perp$, d.h. die Projektion des Oberflächenstücks auf eine Fläche senkrecht zum Lichtvektor $\mathbf{l}$, wird mit voller Intensität beleuchtet (Bild 12.11-a). Falls der auf der Oberfläche senkrecht stehende Normalenvektor $\mathbf{n}$ und der Lichtvektor $\mathbf{l}$ Einheitsvektoren sind, dann gilt für die projizierte Fläche $A_\perp = A \cdot (\mathbf{l} \cdot \mathbf{n}) = A \cdot |\mathbf{l}| \cdot |\mathbf{n}| \cdot \cos(\alpha) = A \cdot \cos(\alpha)$, wobei der Einfallswinkel α der Winkel zwischen dem Normalenvektor $\mathbf{n}$ und dem Lichtvektor $\mathbf{l}$ ist. Das einfallende Licht wird nach dem Lambert'schen Beleuchtungsmodell ideal diffus reflektiert, d.h. gleichverteilt in alle Richtungen (Bild 12.11-b). Damit ist die reflektierte diffuse Lichtintensität nur abhängig von der Position und der Farbe der Lichtquelle und der Orientierung sowie den Reflexionseigenschaften der Oberfläche, aber unabhängig von der Position des Beobachters:

$$\text{diffus} = \max(\mathbf{l} \cdot \mathbf{n}, \mathbf{0}) \cdot \mathbf{d_{light}} * \mathbf{d_{mat}} = \max(\mathbf{l} \cdot \mathbf{n}, \mathbf{0}) \begin{pmatrix} R_{\mathbf{d_{light}}} \cdot R_{\mathbf{d_{mat}}} \\ G_{\mathbf{d_{light}}} \cdot G_{\mathbf{d_{mat}}} \\ B_{\mathbf{d_{light}}} \cdot B_{\mathbf{d_{mat}}} \\ A_{\mathbf{d_{light}}} \cdot A_{\mathbf{d_{mat}}} \end{pmatrix} \tag{12.10}$$

Nur die Oberflächen, die der Lichtquelle zugewandt sind, können überhaupt beleuchtet werden. Das heißt nur Lichtstrahlen mit einem Einfallswinkel aus dem Intervall $[+90°, -90°]$ dürfen einen Beitrag zur Beleuchtung liefern. Um zu verhindern, dass Lichtstrahlen, die auf die Rückseite einer Oberfläche treffen, einen Beleuchtungsbeitrag liefern, müssen negative Werte des Skalarprodukts $\mathbf{l} \cdot \mathbf{n}$ unterdrückt werden. Dies wird mit der Maximum-Funktion $\max(\mathbf{l} \cdot \mathbf{n}, \mathbf{0})$ erreicht.

Praktisch jede Oberfläche reflektiert einen mehr oder weniger großen Teil des aus einer bestimmten Richtung kommenden Lichts diffus nach dem Lambert'schen Beleuchtungsmodell. Der diffuse Beleuchtungsanteil ist in den meisten Fällen der dominante Anteil und bestimmt somit das Aussehen der Objekte. Erst durch die diffuse Beleuchtung werden Flächen, die nicht senkrecht zur Lichtquelle ausgerichtet sind, dunkler, so dass 3D-Formwahrnehmung möglich wird (Bild 12.11-c).

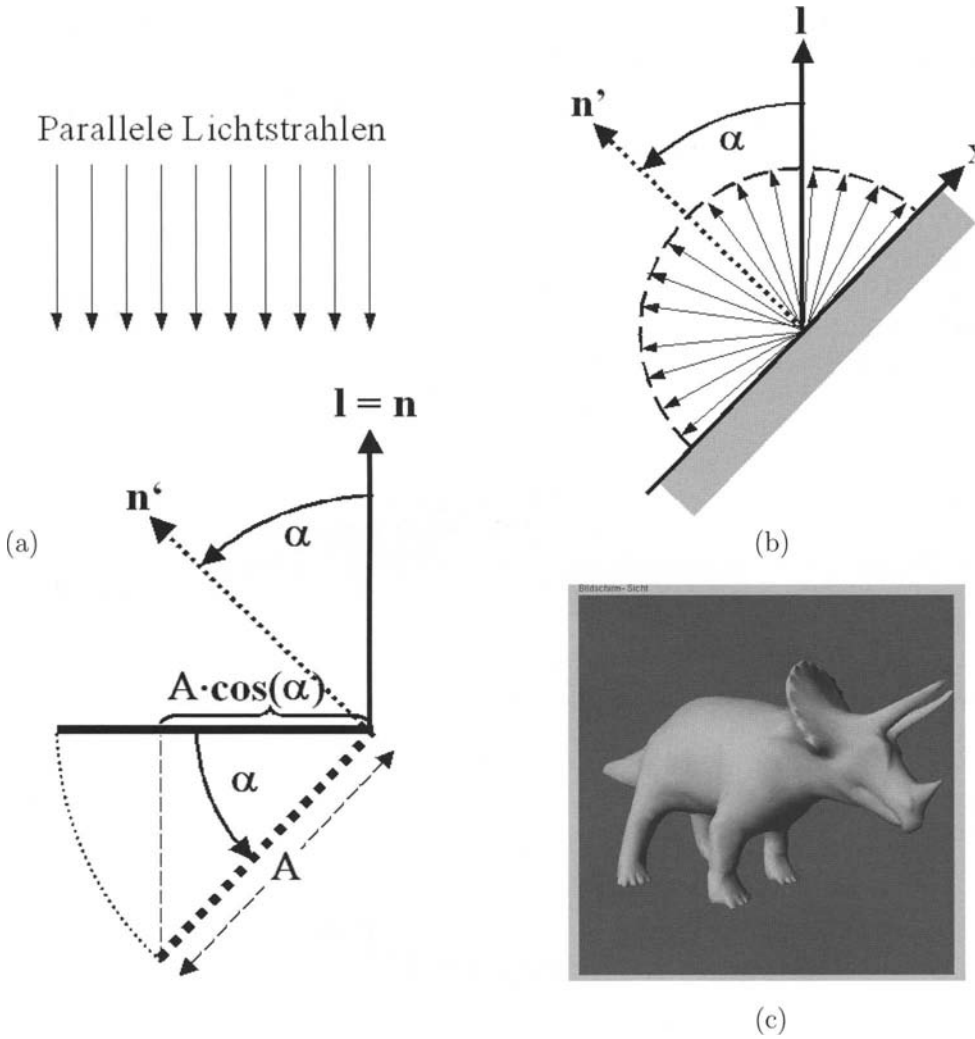


Bild 12.11: Der diffuse Lichtanteil: (a) Der Anteil des einfallenden Lichts hängt von der effektiven Oberfläche aus Sicht der Lichtquelle ab. (b) Das aus einer einzigen Richtung eintreffende Licht wird an der Oberfläche gleichmäßig in alle Richtungen gestreut. (c) Die diffuse Lichtkomponente erzeugt einen 3-dimensionalen Eindruck des Modells, da Flächen, deren Normalenvektoren nicht zur Lichtquelle ausgerichtet sind, dunkler erscheinen.

Die Berechnung des *spekularen* Lichtanteils ist von allen Vieren der aufwändigste. Der spekulare (spiegelnde) Anteil ist das gerichtete Licht, das von einer Punktlichtquelle aus einer einzigen Richtung kommt und überwiegend in eine Vorzugsrichtung reflektiert wird (Bild 12.12-a). Bei diesem nach Phong [Phon75] benannten Beleuchtungsterm wird die größte Lichtintensität in der idealen Reflexionsrichtung **r** abgestrahlt. Je größer der Winkel

Ψ zwischen dem ideal reflektierten Lichtstrahl $\mathbf{r}$ und dem Augenpunktsvektor $\mathbf{a}$ wird, desto kleiner ist die in Richtung Augenpunkt gespiegelte Lichtintensität (Bild 12.12-b). Der Abfall der reflektierten Intensität mit der Abweichung Ψ der Betrachtungsrichtung von der idealen Reflexionsrichtung wird durch den Term $(\cos(\Psi))^S$ modelliert (Bild 12.12-c). Der Exponent S wird *Spiegelungsexponent* bzw. *Shininess*-Faktor genannt. Große Werte von S (z.B. 50-100) lassen eine Oberfläche sehr glatt erscheinen, denn Licht wird fast nur in der idealen Reflexionsrichtung $\mathbf{r}$ abgestrahlt. Die Zigarrenform des Leuchtkörpers wird zur Bleistiftform und das Abbild der Lichtquelle auf der spiegelnden Oberfläche, das sogenannte *Glanzlicht* (*specular highlight*), wird punktförmig klein. Kleine Werte von S (z.B. 5-10) lassen eine Oberfläche relativ matt erscheinen, denn ein erheblicher Anteil des Lichts wird in stark von $\mathbf{r}$ abweichende Richtungen abgestrahlt. Die Zigarrenform des Leuchtkörpers wird zur Eiform und die Glanzlichter werden immer ausgedehnter (Bild 12.12-d). Da die Berechnung des Reflexionsvektors $\mathbf{r}$ aus dem Lichtvektor $\mathbf{l}$ und dem Normalenvektor $\mathbf{n}$ relativ aufwändig ist ($\mathbf{r} = \mathbf{2}(\mathbf{l} \cdot \mathbf{n}) \cdot \mathbf{n} - \mathbf{l}$), wird stattdessen der sogenannte *Halfway*-Vektor $\mathbf{h} = (\mathbf{l} + \mathbf{a})/\|\mathbf{l} + \mathbf{a}\|$ eingeführt. Statt dem Kosinus des Winkels Ψ zwischen den Vektoren $\mathbf{r}$ und $\mathbf{a}$ wird der Kosinus des Winkels Δ zwischen den Vektoren $\mathbf{h}$ und $\mathbf{n}$ benutzt (Bild 12.12-b). Für kleine Winkel ist $\Delta \approx \Psi$, so dass die Näherung gerechtfertig ist, für große Winkel kann $(\cos(\Delta))^S$ positiv sein, obwohl der Lichtvektor $\mathbf{l}$ bereits die Rückseite der Oberfläche bestrahlt. Daher sollte ein möglicher Beitrag des spekularen Lichtanteils vom Vorzeichen des in (12.10) bereits berechneten Skalarprodukts $(\mathbf{l} \cdot \mathbf{n})$ abhängig gemacht werden. Da die Auswirkungen der Approximation durch den Halfway-Vektor im Großen und Ganzen aber kaum erkennbar sind, wird sie in OpenGL aus Geschwindigkeitsgründen verwendet. Insgesamt ist die reflektierte spekulare Lichtintensität abhängig von der Position und der Farbe der Lichtquelle, der Orientierung sowie den Reflexionseigenschaften der Oberfläche und der Position des Beobachters:

$$\text{spekular} = (\max(\mathbf{h} \cdot \mathbf{n}, \mathbf{0}))^{\mathbf{s}} \cdot \mathbf{s_{light}} * \mathbf{s_{mat}} = (\max(\mathbf{h} \cdot \mathbf{n}, \mathbf{0}))^{\mathbf{s}} \begin{pmatrix} R_{\mathbf{s_{light}}} \cdot R_{\mathbf{s_{mat}}} \\ G_{\mathbf{s_{light}}} \cdot G_{\mathbf{s_{mat}}} \\ B_{\mathbf{s_{light}}} \cdot B_{\mathbf{s_{mat}}} \\ A_{\mathbf{s_{light}}} \cdot A_{\mathbf{s_{mat}}} \end{pmatrix} \quad (12.11)$$

Eine berechtigte Frage an dieser Stelle wäre, wieso man die spekularen Materialeigenschaften ($\mathbf{s}_{mat}$) unabhängig von den diffusen Materialeigenschaften ($\mathbf{d}_{mat}$) wählen kann. Die reflektierende Oberfläche ist doch in beiden Fällen die gleiche. Der Grund liegt in den für verschiedene Materialien unterschiedlichen mikroskopischen Prozessen der Absorption und Reflexion von Licht. Poliertes Plastik besteht z.B. aus einem weißen Substrat, das farbige Pigmentpartikel enthält. Der spekulare Lichtanteil bleibt in diesem Fall weiß, da das Licht direkt an der Oberfläche gespiegelt wird. Der diffuse Anteil stammt von Licht, das in die Oberfläche eingedrungen ist, von den farbigen Pigmentpartikeln diffus reflekiert wurde und dann wieder durch die Oberfläche in alle Richtungen austritt. Metalle dagegen lassen aufgrund ihrer guten elektrischen Leitfähigkeit elektromagnetische Wellen kaum in ihre Oberfläche eindringen, so dass der diffuse Lichtanteil relativ gering ist. Da bestimmte Wellenlängenbereiche, je nach Energiebänderstruktur, stark absorbiert werden, erscheint auch der spekulare Lichtanteil in der Metallfarbe (z.B. kupferrot) eingefärbt.

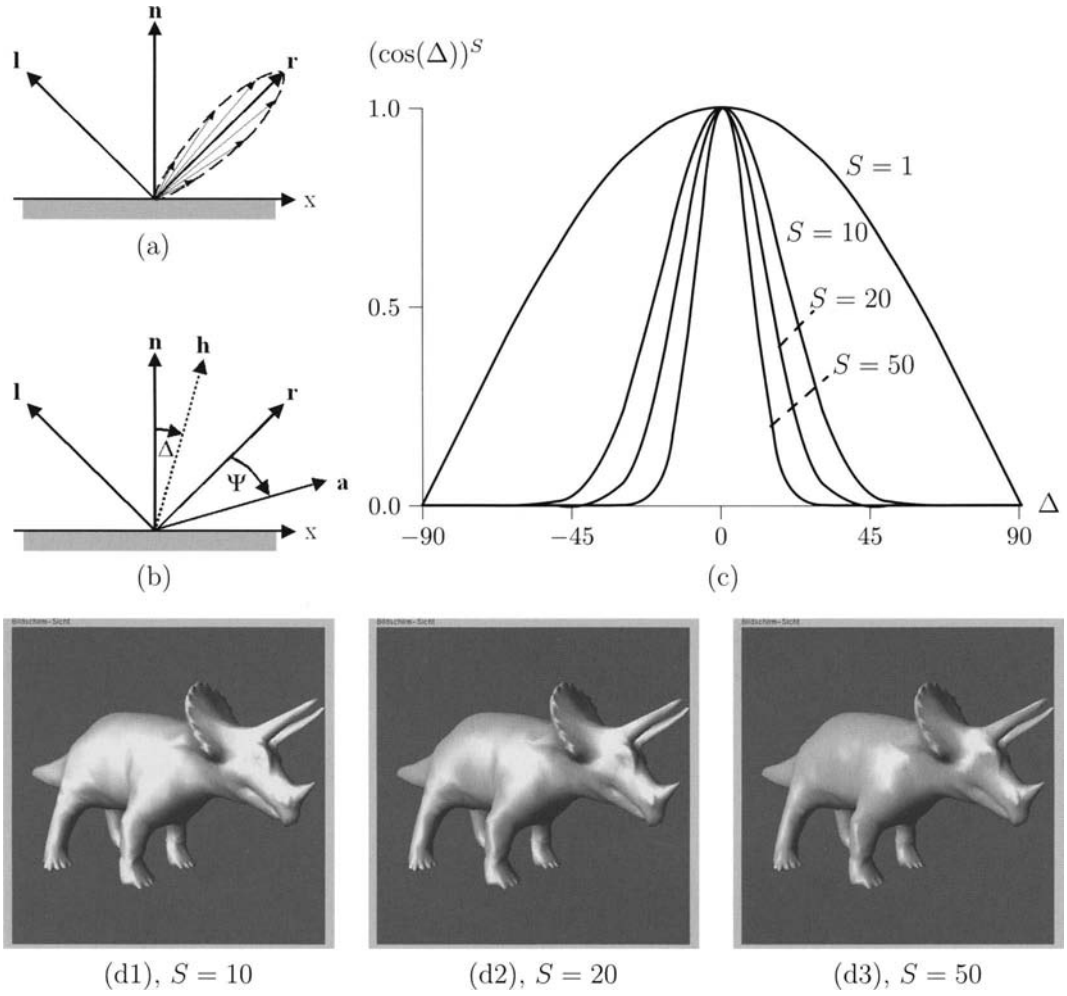


Bild 12.12: Der spekulare Lichtanteil: (a) Das aus einer einzigen Richtung eintreffende Licht wird an der Oberfläche überwiegend in eine Vorzugsrichtung reflektiert. (b) Geometrie beim spekularen Lichtanteil: genau, aber langsam ist die Berechnung des Winkels Ψ zwischen dem ideal reflektierten Lichtstrahl **r** und dem Augenpunktsvektor **a**; ungenauer bei größeren Winkeln, dafür aber schneller ist die Berechnung des Winkels Δ zwischen dem Normalenvektor **n** und dem Halfway-Vektor **h**. (c) Der Abfall der reflektierten Intensität $(\cos(\Delta))^S$ in Abhängigkeit von der Abweichung Δ des *Halfway*-Vektors **h** vom Normalenvektor **n** für verschiedene Werte des Spiegelungsexponenten S. (d) Die Oberfläche des Triceratops erscheint poliert, da er mit einem diffusen und einem spekularen Lichtanteil gerendert wurde und daher an bestimmten Stellen Glanzlichter erscheinen. Von links nach rechts nimmt der Spiegelungsexponent S zu, so dass die Glanzlichter immer kleiner werden und die Oberfläche glatter wirkt.

Die Zusammenfassung aller vier Lichtanteile liefert die Beleuchtungsformel (12.12) im RGBA-Modus bei einer Lichtquelle (ohne Spotlight) für einen Vertex. Die Formel ist bewusst in Matrixform dargestellt, die Zeilen enthalten die vier Lichtanteile (emissiv, ambient, diffus und spekular), die Spalten zeigen an, welche Faktoren den jeweiligen Lichtanteil beeinflussen (das beleuchtete Material, die Lichtquelle bzw. die 3D-Anordnung von Oberfläche und Lichtquelle):

Vertexfarbe	3D-Anordnung		Lichtquelle		Material		Komponente
$\mathbf{g}_{Vertex} =$					$\mathbf{e}_{mat}$	$+$	emissiv
			$\mathbf{a}_{light}$	$*$	$\mathbf{a}_{mat}$	$+$	ambient
	$\max(\mathbf{l}\cdot\mathbf{n},\mathbf{0})$	$\cdot$	$\mathbf{d}_{light}$	$*$	$\mathbf{d}_{mat}$	$+$	diffus
	$(\max(\mathbf{h}\cdot\mathbf{n},\mathbf{0}))^{\mathbf{s}}$	$\cdot$	$\mathbf{s}_{light}$	$*$	$\mathbf{s}_{mat}$		spekular

(12.12)

In den folgenden Abschnitten wird gezeigt, wie all die vielen Parameter der Beleuchtungsformel (12.12) in OpenGL spezifiziert werden und welche Erweiterungen zu dieser Basis noch existieren.

12.1.3.2 Lichtquelleneigenschaften

Um Eigenschaften einer einfachen Standard-Lichtquelle, wie sie in (12.12) verwendet wird, zu verändern, können vier Komponenten eingestellt werden: die RGBA-Intensitäten für die ambiente Emission, die RGBA-Intensitäten für die diffuse Emission, die RGBA-Intensitäten für die spekulare Emission und die Position bzw. Richtung der Lichtquelle. Dazu dienen die in der folgenden Tabelle dargestellten Varianten des `glLight`-Befehls:

Skalar-Form
`glLightf(GLenum light, GLenum name, GLfloat param)`
`glLighti(GLenum light, GLenum name, GLdouble param)`
Vektor-Form
`glLightfv(GLenum light, GLenum name, GLfloat *param)`
`glLightiv(GLenum light, GLenum name, GLdouble *param)`

Der erste Parameter des `glLight`-Befehls, `light`, legt fest, welche Lichtquelle in ihren Eigenschaften verändert werden soll. Er kann die Werte `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7` annehmen, d.h. es können bis zu acht verschiedene Lichtquellen gleichzeitig aktiviert werden (in (12.12) ist nur eine Standard-Lichtquelle berücksichtigt, (12.17) zeigt den allgemeinen Fall). Der zweite Parameter, `name`, legt fest, welche Eigenschaft der jeweiligen Lichtquelle eingestellt werden soll und der dritte Parameter, `param`, legt fest, auf welche Werte die ausgewählte Eigenschaft gesetzt werden soll. In der folgenden Tabelle sind die vier Standard-Eigenschaften von Lichtquellen mit ihren Standard-Werten und ihrer Bedeutung aufgelistet:

Eigenschaft name	Standard-Werte von param	Bedeutung
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	RGBA-Intensitäten für den ambienten Anteil der Lichtquelle
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0) oder (0.0, 0.0, 0.0, 1.0)	RGBA-Intensitäten für den diffusen Anteil der Lichtquelle. Der Standardwert für die Lichtquelle 0 ist weiß, für die anderen Lichtquellen schwarz
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0) oder (0.0, 0.0, 0.0, 1.0)	RGBA-Intensitäten für den spekularen Anteil der Lichtquelle. Der Standardwert für die Lichtquelle 0 ist weiß, für die anderen Lichtquellen schwarz
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x,y,z,w)-Position der Lichtquelle. w = 0 bedeutet, dass die Lichtquelle im Unendlichen sitzt

Zur Positionierung einer Lichtquelle sind die folgenden beiden Punkte zu beachten:

- Die Position einer Lichtquelle wird in homogenen Koordinaten (Kapitel 7) angegeben, wobei die vierte Koordinate den inversen Streckungsfaktor w darstellt. Bei einer *Richtungslichtquelle* gilt $w = 0$, was bedeutet, dass die Lichtquelle im Unendlichen sitzt und die Lichtstrahlen parallel eintreffen. In diesem Fall muss nur ein einziger Lichtvektor $\mathbf{l}$ berechnet werden, der für alle Vertices einer Szene verwendet werden kann. Bei einer *lokalen Lichtquelle* gilt $w > 0$ und die Lichtstrahlen treffen nicht mehr parallel ein, sondern für jeden Vertex muss ein eigener Lichtvektor $\mathbf{l}_i$ berechnet werden (Bild 12.13). Dies ist erheblich aufwändiger und daher echtzeitschädlich. Allerdings bieten nur lokale Lichtquellen die Möglichkeit, sogenannte *Abschwächungsfaktoren* zu verwenden, die eine Verminderung der Lichtintensität mit der Entfernung zur Lichtquelle bewirken (Abschnitt 12.1.3.5).

- Die Position einer Lichtquelle wird den gleichen Modell- und Augenpunktstransformationen unterworfen wie Objekte und anschließend in Weltkoordinaten gespeichert. Soll eine Lichtquelle unabhängig von den Objekten bewegt werden, sind die entsprechenden Modelltransformationen durchzuführen und anschließend ist die Lichtquelle zu positionieren. Diese Operationen werden zwischen einer `glPushMatrix()`-`glPopMatrix()`-Klammer ausgeführt, so dass für die Lichtquellenposition eine eigene Matrix auf dem *Modelview*-Matrizenstapel abgelegt wird.

Aktivierung von Beleuchtungsrechnung und Lichtquellen

Um eine Lichtquelle zur Wirkung zu bringen, muss der OpenGL-Zustandsautomat noch in die richtige Stellung gebracht werden. Damit die Beleuchtungsrechnung nach (12.12) bzw. (12.17) durchgeführt wird, muss der Zustand durch den Befehl:

```
glEnable(GL_LIGHTING);
```

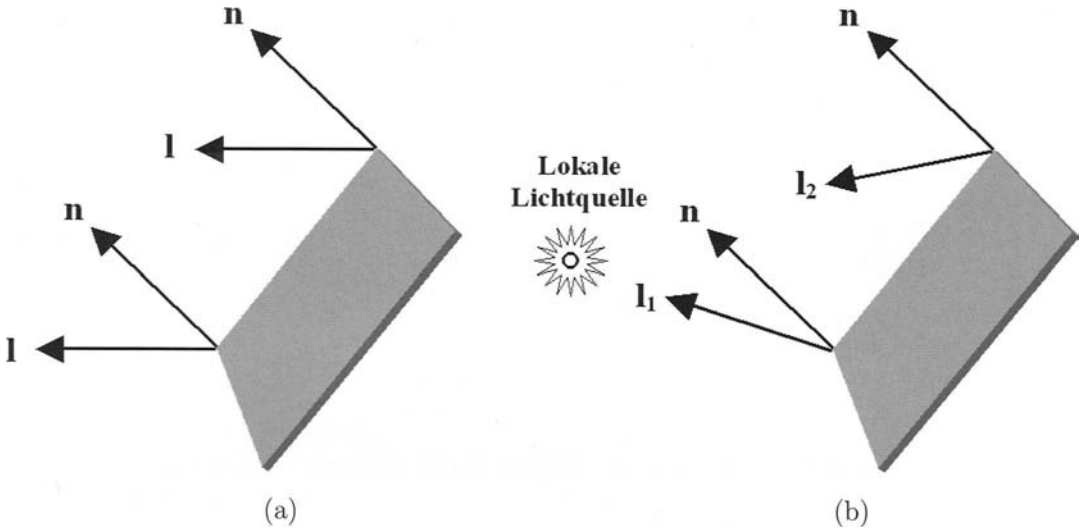


Bild 12.13: Richtungslichtquellen versus lokale Lichtquellen: (a) Richtungslichtquelle ($w = 0$): alle Lichtstrahlen verlaufen parallel, es genügt ein einziger Lichtvektor $\mathbf{l}$. (b) Lokale Lichtquelle ($w = 1$): die Lichtstrahlen verlaufen nicht parallel, für jeden Vertex muss ein eigener Lichtvektor $\mathbf{l}_i$ berechnet werden.

eingeschaltet werden. In diesem Fall sind die mit dem `glColor*()`-Befehl festgelegten Farbwerte wirkungslos. Standardmäßig ist die Beleuchtungsrechnung ausgeschaltet (`glDisable(GL_LIGHTING)`), so dass die Vertices direkt die mit dem `glColor*()`-Befehl festgelegten Farbwerte erhalten. Außerdem besteht die Möglichkeit, jede Lichtquelle einzeln ein- und auszuschalten. Dazu dient der Befehl:

```
glEnable(GL_LIGHT0);
```

In diesem Beispiel wird die Lichtquelle Nr. `0` eingeschaltet. Standardmäßig sind alle Lichtquellen ausgeschaltet, so dass der obige Befehl immer nötig ist, um eine Lichtquelle zur Wirkung zu bringen.

12.1.3.3 Oberflächeneigenschaften

Für eine sinnvolle Beleuchtung müssen neben der Lichtquelle auch noch die Eigenschaften der Oberflächen spezifiziert werden. Gemäß (12.12) sind dies die Materialeigenschaften $(\mathbf{e}_{mat}, \mathbf{a}_{mat}, \mathbf{d}_{mat}, \mathbf{s}_{mat}, S)$ und die Normalenvektoren $\mathbf{n}$.

Materialeigenschaften

Um die Materialeigenschaften einer Oberfläche zu verändern, können vier Komponenten eingestellt werden: die RGBA-Werte für die ambiente Reflexion, die diffuse Reflexion, die spekulare Reflexion und für die Emission, sowie den Spiegelungsexponenten. Dazu dienen die in der folgenden Tabelle dargestellten Varianten des `glMaterial`-Befehls:

Skalar-Form
`glMaterialf(GLenum face, GLenum name, GLfloat param)`
`glMateriali(GLenum face, GLenum name, GLdouble param)`
Vektor-Form
`glMaterialfv(GLenum face, GLenum name, GLfloat *param)`
`glMaterialiv(GLenum face, GLenum name, GLdouble *param)`

Der erste Parameter des `glMaterial`-Befehls, `face`, legt fest, welche Seite der Oberfläche in ihren Eigenschaften verändert werden soll. Er kann die Werte `GL_FRONT`, `GL_BACK`, oder `GL_FRONT_AND_BACK` annehmen, d.h. es kann entweder die Vorderseite, die Rückseite, oder beide Seiten in gleicher Weise verändert werden. Der zweite Parameter, `name`, legt fest, welche Materialeigenschaft eingestellt werden soll, und der dritte Parameter, `param`, legt fest, auf welche Werte die ausgewählte Eigenschaft gesetzt werden soll. In der folgenden Tabelle sind die Materialeigenschaften von Oberflächen mit ihren Standard-Werten und ihrer Bedeutung aufgelistet:

Eigenschaft `name`	**Standard-Werte von** `param`	**Bedeutung**
`GL_AMBIENT`	$(0.2, 0.2, 0.2, 1.0)$	RGBA-Werte für die ambiente Reflexion. Standardwert dunkelgrau
`GL_DIFFUSE`	$(0.8, 0.8, 0.8, 1.0)$	RGBA-Werte für die diffuse Reflexion. Standardwert hellgrau
`GL_AMBIENT_AND_DIFFUSE`		Die ambienten und diffusen Reflexionskoeffizienten werden auf die gleichen RGBA-Werte gesetzt
`GL_SPECULAR`	$(0.0, 0.0, 0.0, 1.0)$	RGBA-Werte für die spekulare Reflexion. Standardwert schwarz
`GL_SHININESS`	0.0	Spiegelungsexponent S
`GL_EMISSION`	$(0.0, 0.0, 0.0, 1.0)$	Emissive Farbe des Materials. Standardwert schwarz

In der folgenden Tabelle findet man für reale Materialien sinnvolle Vorschläge für die entsprechenden Reflexionskoeffizienten und die Spiegelungsexponenten.

Material	R_a, G_a, B_a, A_a	R_d, G_d, B_d, A_d	R_s, G_s, B_s, A_s	**S**
Schwarzes Plastik	.00, .00, .00, 1.0	.01, .01, .01, 1.0	.50, .50, .50, 1.0	32.0
Schwarzer Gummi	.02, .02, .02, 1.0	.01, .01, .01, 1.0	.40, .40, .40, 1.0	10.0
Messing	.33, .22, .03, 1.0	.78, .57, .11, 1.0	.99, .94, .81, 1.0	27.9
Bronze	.21, .13, .05, 1.0	.71, .43, .18, 1.0	.39, .27, .17, 1.0	25.6
Poliertes Bronze	.25, .15, .06, 1.0	.40, .24, .10, 1.0	.77, .46, .20, 1.0	76.8
Chrom	.25, .25, .25, 1.0	.40, .40, .40, 1.0	.77, .77, .77, 1.0	76.8
Kupfer	.19, .07, .02, 1.0	.70, .27, .08, 1.0	.26, .14, .09, 1.0	12.8
Poliertes Kupfer	.23, .09, .03, 1.0	.55, .21, .07, 1.0	.58, .22, .07, 1.0	51.2
Gold	.25, .20, .07, 1.0	.75, .61, .23, 1.0	.63, .56, .37, 1.0	51.2
Poliertes Gold	.25, .22, .06, 1.0	.35, .31, .09, 1.0	.80, .72, .21, 1.0	83.2
Zinn	.11, .06, .11, 1.0	.43, .47, .54, 1.0	.33, .33, .52, 1.0	9.8
Silber	.19, .19, .19, 1.0	.51, .51, .51, 1.0	.51, .51, .51, 1.0	51.2
Poliertes Silber	.23, .23, .23, 1.0	.28, .28, .28, 1.0	.77, .77, .77, 1.0	89.6
Smaragdgrün	.02, .17, .02, 0.5	.08, .61, .08, 0.5	.63, .73, .63, 0.5	76.8
Jade	.14, .22, .16, 0.9	.54, .89, .63, 0.9	.32, .32, .32, 0.9	12.8
Obsidian	.05, .05, .07, 0.8	.18, .17, .23, 0.8	.33, .33, .35, 0.8	38.4
Perle	.25, .21, .21, 0.9	.99, .83, .83, 0.9	.30, .30, .30, 0.9	11.3
Rubin	.17, .01, .01, 0.5	.61, .04, .04, 0.5	.73, .63, .63, 0.5	76.8
Türkis	.10, .19, .17, 0.8	.40, .74, .69, 0.8	.30, .31, .31, 0.8	12.8

In OpenGL gibt es noch eine zweite Möglichkeit, um die Materialeigenschaften zu ändern: den sogenannten *Color Material Mode*. In diesem Fall wird OpenGL angewiesen, die bei eingeschalteter Beleuchtung eigentlich sinnlosen `glColor*()`-Befehle als Materialspezifikation zu nutzen. Jeder `glColor*()`-Aufruf ändert effizienter als der `glMaterial*()`-Aufruf eine Materialeigenschaft. Nur der Spiegelungsexponent S kann im *Color Material Mode* nicht verändert werden. Zur Aktivierung des *Color Material Mode* muss zunächst der entsprechende OpenGL-Zustand durch den Befehl:

```
glEnable(GL_COLOR_MATERIAL);
```

eingeschaltet werden. Anschließend wird durch den Befehl:

```
glColorMaterial(GLenum face, GLenum mode);
```

festgelegt, welche Seite der Oberfläche (`face = GL_FRONT, GL_BACK, GL_FRONT_AND_BACK`) bezüglich welcher Materialeigenschaft (`mode = GL_AMBIENT, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE, GL_SPECULAR, GL_EMISSION`) durch die `glColor*()`-Aufrufe geändert wird.

Der folgende Ausschnitt aus einem Programm-Code zeigt die Anwendung des *Color Material Mode* exemplarisch:

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
// glColor ändert die ambiente und diffuse Reflexion

glColor3f(0.61, 0.04, 0.04);
draw_objects1(); // rubinrot
glColor3f(0.28, 0.28, 0.28);
draw_objects2(); // silbergrau
glColorMaterial(GL_FRONT, GL_SPECULAR);
// glColor ändert nur mehr die spekulare Reflexion

glColor3f(0.99, 0.99, 0.63);
draw_objects3(); // hellgelbes Glanzlicht
glDisable(GL_COLOR_MATERIAL);
```

Normalenvektoren

Der Normalenvektor $\mathbf{n}$ einer ebenen Oberfläche wird für den diffusen und den spekularen Beleuchtungsanteil in (12.12) benötigt. Er kann mathematisch aus dem Kreuzprodukt zweier Vektoren gewonnen werden, die die Ebene aufspannen. Denn das Kreuzprodukt zweier Vektoren steht senkrecht auf den beiden Vektoren und daher auch senkrecht auf der Ebene. Am sichersten funktioniert dies bei Dreiecken, da drei Vertices im Raum, die nicht auf einer Linie liegen, immer eine Ebene definieren. Bei Polygonen muss sichergestellt sein, dass sich alle Vertices in einer Ebene befinden, dann können drei beliebige Vertices $\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}$ ausgesucht werden, die nicht auf einer Linie liegen, um den Normalenvektor $\mathbf{n}$ zu berechnen:

$$\begin{aligned}\mathbf{n} &= (\mathbf{v}_1 - \mathbf{v}_2) \times (\mathbf{v}_2 - \mathbf{v}_3) = \begin{pmatrix} \mathbf{v_{1x}} - \mathbf{v_{2x}} \\ \mathbf{v_{1y}} - \mathbf{v_{2y}} \\ \mathbf{v_{1z}} - \mathbf{v_{2z}} \end{pmatrix} \times \begin{pmatrix} \mathbf{v_{2x}} - \mathbf{v_{3x}} \\ \mathbf{v_{2y}} - \mathbf{v_{3y}} \\ \mathbf{v_{2z}} - \mathbf{v_{3z}} \end{pmatrix} \\ &= \begin{pmatrix} (\mathbf{v_{1y}} - \mathbf{v_{2y}}) \cdot (\mathbf{v_{2z}} - \mathbf{v_{3z}}) - (\mathbf{v_{1z}} - \mathbf{v_{2z}}) \cdot (\mathbf{v_{2y}} - \mathbf{v_{3y}}) \\ (\mathbf{v_{1z}} - \mathbf{v_{2z}}) \cdot (\mathbf{v_{2x}} - \mathbf{v_{3x}}) - (\mathbf{v_{1x}} - \mathbf{v_{2x}}) \cdot (\mathbf{v_{2z}} - \mathbf{v_{3z}}) \\ (\mathbf{v_{1x}} - \mathbf{v_{2x}}) \cdot (\mathbf{v_{2y}} - \mathbf{v_{3y}}) - (\mathbf{v_{1y}} - \mathbf{v_{2y}}) \cdot (\mathbf{v_{2x}} - \mathbf{v_{3x}}) \end{pmatrix} \end{aligned} \tag{12.13}$$

Damit die Beleuchtungsrechnung nach (12.12) korrekt funktioniert, müssen die Normalenvektoren (und auch die anderen beteiligten Vektoren) Einheitsvektoren sein, d.h. sie müssen die Länge 1 haben. Dazu müssen die Normalenvektoren noch normiert werden:

$$\mathbf{n_e} = \frac{\mathbf{n}}{\|\mathbf{n}\|} = \frac{\mathbf{n}}{\sqrt{\mathbf{n} \cdot \mathbf{n}}} = \frac{1}{\sqrt{\mathbf{n_x^2} + \mathbf{n_y^2} + \mathbf{n_z^2}}} \cdot \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \tag{12.14}$$

In OpenGL werden die Normalenvektoren aber nicht automatisch aus den Vertices berechnet, sondern diese Aufgabe bleibt aus Gründen der Flexibilität, die in Abschnitt (12.2.2) erläutert werden, dem Programmierer überlassen. Er muss die Normalenvektoren selber berechnen und am besten auch gleich normieren. Danach kann er sie mit dem

Befehl `glNormal3f(GLfloat coords)` einer Fläche bzw. einem Vertex zuordnen. Selbstverständlich existieren auch für den `glNormal3*()`-Befehl verschiedene Varianten, die in der folgenden Tabelle aufgelistet sind:

Skalar-Form	Vektor-Form
`glNormal3f(GLfloat coords)`	`glNormal3fv(GLfloat *coords)`
`glNormal3f(GLdouble coords)`	`glNormal3fv(GLdouble *coords)`
`glNormal3b(GLbyte coords)`	`glNormal3bv(GLbyte *coords)`
`glNormal3s(GLshort coords)`	`glNormal3sv(GLshort *coords)`
`glNormal3i(GLint coords)`	`glNormal3iv(GLint *coords)`

Man kann die Normierung der Normalenvektoren auch OpenGL überlassen, indem man den Befehl:

```
glEnable(GL_NORMALIZE);
```

aufruft. Dies ist jedoch mit Geschwindigkeitseinbußen verbunden. Falls es möglich ist, sollte man immer vorab die Normierung in einem externen Programm durchführen. Allerdings besteht auch in diesem Fall immer noch ein Problem mit der Länge der Normalenvektoren, denn üblicherweise sind sie im lokalen Modellkoordinatensystem berechnet bzw. normiert und durch die Modell- und Augenpunktstransformationen (Kapitel 7), die auch eine Skalierung enthalten kann, werden die Maßstäbe verzerrt, so dass die Normierung verloren gehen kann. Als Abhilfe bieten sich zwei Möglichkeiten an: entweder man stellt sicher, dass die Modell- und Augenpunktstransformationen keine Skalierung enthalten, oder man ruft den OpenGL-Befehl:

```
glEnable(GL_RESCALE_NORMALS);
```

auf, der aus der Modelview-Matrix den inversen Skalierungsfaktor berechnet und damit die Normalenvektoren reskaliert, so dass sie wieder die Länge 1 besitzen. Die Reskalierung ist mathematisch einfacher und daher in OpenGL schneller als die Normierung.

12.1.3.4 Lighting Model

In OpenGL versteht man unter einem *Lighting Model* die folgenden speziellen Einstellungen des Standard-Beleuchtungsmodells (und somit etwas Anderes, als unter einem Beleuchtungsmodell in der Computergrafik):

- Einen *globalen ambienten Lichtanteil*, der unabhängig von den ambienten Anteilen der Lichtquellen festgelegt werden kann. Der voreingestellte Wert $(0.2, 0.2, 0.2, 1.0)$, ein schwaches weißes Licht, führt dazu, dass die Objekte in einer Szene auch ohne die Aktivierung einer Lichtquelle sichtbar sind.

- Die Definition eines *lokalen Augenpunkts*. Für den spekularen Lichtanteil wird der Augenpunktsvektor **a**, d.h. der Vektor vom Vertex zum Augenpunkt, benötigt. Ähnlich wie bei Lichtquellen wird standardmäßig auch der Augenpunkt als unendlich entfernt (mit den Koordinaten $(0.0, 0.0, 1.0, 0.0)$) angenommen, so dass für alle Vertices in einer Szene nur ein einziger Augenpunktsvektor **a** berechnet werden muss. Wird ein lokaler Augenpunkt eingestellt, der immer an der Stelle $(0.0, 0.0, 0.0, 1.0)$ sitzt, muss für jeden Vertex ein eigener Augenpunktsvektor $\mathbf{a_i}$ berechnet werden, was zwar realitätsnäher, aber dafür natürlich wieder echtzeitschädlicher ist (Bild 12.13).

- Die Aktivierung der *zweiseitigen Beleuchtung*. Standardmäßig werden nur die Vorderseiten von Polygonen korrekt beleuchtet. Damit die Rückseiten, die nur innerhalb geschlossener Objekte oder bei aufgeschnittenen Objekten sichtbar sind, ebenfalls korrekt beleuchtet werden, müssen die Normalenvektoren invertiert werden. Diese Invertierung führt OpenGL nach der Aktivierung der zweiseitigen Beleuchtung automatisch durch. Außerdem werden im Zuge der zweiseitigen Beleuchtung auch unterschiedliche Materialeigenschaften von Vorder- und Rückseiten berücksichtigt.

- Die *separate Berechnung des spekularen Lichtanteils* und die Addition dieses Anteils erst nach dem Textur Mapping. Eine ausführlichere Darstellung dieses Punkts wird im Abschnitt 12.1.3.6 gegeben.

Zur Spezifikation des *Lighting Model* von OpenGL dienen die in der folgenden Tabelle dargestellten Varianten des `glLightModel`-Befehls:

Skalar-Form
`glLightModelf(GLenum name, GLfloat param)`
`glLightModeli(GLenum name, GLdouble param)`
Vektor-Form
`glLightModelfv(GLenum name, GLfloat *param)`
`glLightModeliv(GLenum name, GLdouble *param)`

Der erste Parameter, `name`, legt fest, welche Eigenschaft der jeweiligen Lichtquelle eingestellt werden soll. Der zweite Parameter, `param`, legt fest, auf welche Werte die ausgewählte Eigenschaft gesetzt werden soll. In der folgenden Tabelle sind die vier Eigenschaften des *Lighting Model* von OpenGL mit ihren Standard-Werten und ihrer Bedeutung aufgelistet:

Eigenschaft name	Standard-Werte von param	Bedeutung
GL_LIGHT_MODEL _AMBIENT	(0.2, 0.2, 0.2, 1.0)	RGBA-Intensitäten für den globalen ambienten Lichtanteil ($\mathbf{a}_{global}$)
GL_LIGHT_MODEL _LOCAL_VIEWER	0.0 oder GL_FALSE	unendlich entfernter oder lokaler Augenpunkt, relevant für die Berechnung des spekularen Lichtanteils
GL_LIGHT_MODEL _TWO_SIDE	0.0 oder GL_FALSE	Beleuchtungsrechnung nur bezogen auf die Vorderseiten (einseitig), oder getrennt für Vorder- und Rückseite (zweiseitig)
GL_LIGHT_MODEL _COLOR_CONTROL	GL_SINGLE_COLOR	Alle Beleuchtungskomponenten in einer Farbe oder separate Berechnung des spekularen Farbanteils

12.1.3.5 Spotlights und Abschwächungsfaktoren

Für lokale Lichtquellen kann noch ein ganze Reihe weiterer Parameter festgelegt werden, die die Realitätsnähe steigern, aber die Beleuchtungsrechnung komplizieren und somit die Renderinggeschwindigkeit senken. Mit der ersten Gruppe von Parametern können sogenannte *Spotlights* (Scheinwerfer) definiert werden, die nur in einen begrenzten Raumwinkelbereich Licht ausstrahlen. Der Raumwinkelbereich wird als radialsymmetrischer Kegel angenommen, so dass er durch eine Richtung $\mathbf{k}$ und den halben Kegelöffnungswinkel ($Spot_{Cutoff}$) charakterisiert werden kann (Bild 12.14). Die zulässigen Werte des halben Kegelöffnungswinkels ($Spot_{Cutoff}$) liegen im Intervall $[0°, 90°]$ für echte *Spotlights*. Außerdem ist noch der Wert 180° für normale Punktlichtquellen zulässig, die in alle Raumrichtungen ausstrahlen. Die Idee, *Spotlights* dadurch zu erzeugen, dass man eine lokale Lichtquelle mit einigen opaken Polygonen umgibt, scheitert natürlich, denn wie eingangs erwähnt, berücksichtigt das Standard-Beleuchtungsmodell von OpenGL die Verdeckung von Lichtquellen durch andere Polygone nicht (deshalb gibt es echten Schattenwurf in OpenGL auch nicht kostenlos). Reale Lichtquellen weisen noch zwei weitere Phänomene auf, die in OpenGL durch eine zweite Gruppe von Paramtern nachgebildet werden können: die Abschwächung der Lichtintensität eines Scheinwerferkegels vom Zentrum zum Rand hin (also senkrecht zur Strahlrichtung), sowie die Abschwächung der Lichtintensität mit der Entfernung zur Lichtquelle (also in Strahlrichtung).

Die Abschwächung senkrecht zur Strahlrichtung wird durch einen Faktor $f_\perp$ modelliert, der sich aus dem Kosinus des Winkels zwischen der zentralen Strahlrichtung ($\mathbf{k}$) und der Richtung von der Lichtquelle zum beleuchteten Vertex ($\mathbf{v}$) potenziert mit einem Wert $Spot_{Exp}$ ergibt, d.h.:

$$f_\perp = (\max(\mathbf{v} \cdot \mathbf{k}, \mathbf{0}))^{\mathbf{Spot_{Exp}}} \tag{12.15}$$

Dabei wird wieder vorausgesetzt, dass die beiden Vektoren $\mathbf{v}$ und $\mathbf{k}$ normiert sind und dass $\cos(Spot_{Cutoff}) < \max(\mathbf{v} \cdot \mathbf{k}, \mathbf{0})$, d.h. dass der Vertex innerhalb des Lichtkegels liegt,

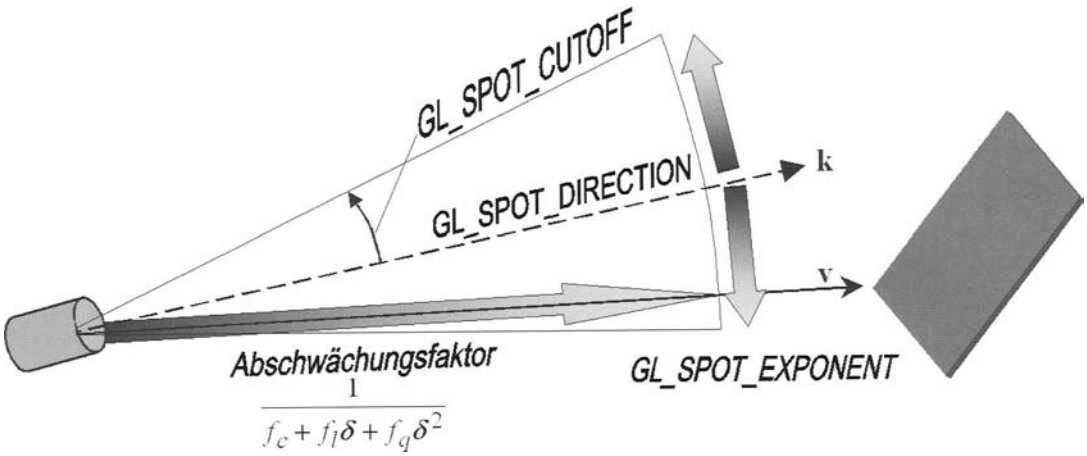


Bild 12.14: *Spotlights* und Abschwächungsfaktoren: Richtung Lichtquelle–Vertex (**v**), Kegelrichtung (**k**), halber Kegelöffnungswinkel ($Spot_{Cutoff}$), Abschwächungsfaktor senkrecht zur Strahlrichtung ($Spot_{Exp}$), Abschwächungsfaktoren in Strahlrichtung (konstant: f_c, linear: f_l, quadratisch: f_q).

ansonsten wird der Faktor $f_\perp = 0$. Aus rein physikalischer Sicht müsste die Intensität einer Punktlichtquelle quadratisch mit der Entfernung abfallen, denn bei gleichbleibender Leistung der Lichtquelle verteilt sich diese auf eine quadratisch mit dem Radius zunehmende Kugeloberfläche ($4\pi \cdot r^2$). In OpenGL wird dies durch einen quadratischen Abschwächungsfaktor f_q (*quadratic attenuation*) berücksichtigt. In praktischen Anwendungen stellt sich ein rein quadratischer Abschwächungsfaktor aber häufig als zu extrem heraus, so dass in OpenGL auch noch ein linearer f_l und ein konstanter f_c Abschwächungsfaktor eingeführt wurde. Damit kann für jede Lichtquelle getrennt ein Abschwächungsfaktor in Strahlrichtung $f_\parallel$ als

$$f_\parallel = \frac{1}{f_c + f_l\delta + f_q\delta^2} \tag{12.16}$$

angegeben werden, wobei δ die Entfernung zwischen der lokalen Lichtquelle und dem Vertex darstellt.

Zur Spezifikation der Eigenschaften von *Spotlights* und Abschwächungsfaktoren dient der bereits in Abschnitt (12.1.3.2) vorgestellte `glLight*()`-Befehl. In der folgenden Tabelle sind die sechs entsprechenden Parameter mit ihren Standard-Werten und ihrer Bedeutung aufgelistet:

Eigenschaft name	Standard-Werte von param	Bedeutung
GL_SPOT _DIRECTION	$(0.0, 0.0, -1.0)$	$(\mathbf{k} = (k_x, k_y, k_z)^T)$-Richtung des Scheinwerferkegels
GL_SPOT _CUTOFF	180.0	halber Öffnungswinkel des Scheinwerferkegels ($Spot_{Cutoff}$)
GL_SPOT _EXPONENT	0.0	Abschwächungsfaktor senkrecht zur Strahlrichtung ($Spot_{Exp}$)
GL_CONSTANT _ATTENUATION	1.0	konstanter Abschwächungsfaktor f_c
GL_LINEAR _ATTENUATION	0.0	linearer Abschwächungsfaktor f_l
GL_QUADRATIC _ATTENUATION	0.0	quadratischer Abschwächungsfaktor f_q

Insgesamt lassen sich die Abschwächungsfaktoren und die *Spotlight*-Effekte für jede Lichtquelle zu einem Vorfaktor zusammenfassen, der auf die ambienten, diffusen und spekularen Anteile dieser Lichtquelle dämpfend wirkt. Zusammen mit dem globalen ambieten Licht aus dem OpenGL *Lighting Model* ($\mathbf{a}_{global}$) folgt daraus in Verallgemeinerung von (12.12) die vollständige Beleuchtungsformel im RGBA-Modus **mit i *Spotlights* und Abschwächungsfaktoren** für einen Vertex:

$$
\begin{array}{lrcll}
\text{Vertexfarbe} & \text{3D-Anordnung} & \text{Lichtquelle} & \text{Material} & & \text{Komponente} \\
\hline
\mathbf{g}_{Vertex} = & & & \mathbf{e}_{mat} & + & \text{emissiv} \\
 & & \mathbf{a}_{global} \ * & \mathbf{a}_{mat} & + & \text{global ambient} \\
 & \sum_{i=0}^{n-1} \left(\frac{1}{f_c + f_l\delta + f_q\delta^2}\right)_i (\max(\mathbf{v}\cdot\mathbf{k}, 0))_{\mathbf{i}}^{\mathbf{Spot_{Exp}}} & & & \cdot & \text{Faktor Nr. i} \\
 & \Big\{ & \mathbf{a}_{light} \ * & \mathbf{a}_{mat} & + & \text{ambient Nr. i} \\
 & \max(\mathbf{l}\cdot\mathbf{n}, \mathbf{0}) \ \cdot & \mathbf{d}_{light} \ * & \mathbf{d}_{mat} & + & \text{diffus Nr. i} \\
 & (\max(\mathbf{h}\cdot\mathbf{n}, \mathbf{0}))^{\mathbf{s}} \ \cdot & \mathbf{s}_{light} \ * & \mathbf{s}_{mat} & \Big\}_i & \text{spekular Nr. i}
\end{array}
\tag{12.17}
$$

In (12.17) ist zu beachten, dass der Faktor Nr. i nur dann einen Beitrag liefert, falls: $\cos(Spot_{Cutoff}) < \max(\mathbf{v}\cdot\mathbf{k}, \mathbf{0}))$, ansonsten ist der Faktor null.

Wenn man bedenkt, dass in realistischen Szenen zwischen zehn- und hunderttausend Vertices im sichtbaren Volumen enthalten sind, also die doch schon ziemlich komplexe Beleuchtungsformel (12.17) ebenso oft berechnet werden muss und dafür obendrein nur

ca. 16 Millisekunden (bei 60 Hz Bildgenerierrate) zur Verfügung stehen, bekommt man ein gewisses Gefühl für die immense Rechenleistung, die für interaktive 3D-Computergrafik erforderlich ist.

12.1.3.6 Separate spekulare Farbe

In der normalen Beleuchtungsformel (12.17) werden die vier Komponenten emissiv, ambient, diffus und spekular einfach zu einer Vertexfarbe addiert. Da das Textur-Mapping in der Rendering-Pipeline (Bild 5.1) nach der Beleuchtungsrechnung erfolgt, werden die spekularen Glanzlichter häufig durch die Texturfarben unterdrückt. Um dieses Problem zu umgehen, kann man OpenGL durch den Befehl:
`glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SEPARATE_SPECULAR_COLOR)`
anweisen, die spekulare Farbkomponente separat zu berechnen. Die Beleuchtungsrechnung liefert dann zwei Farben pro Vertex: eine primäre Farbe ($\mathbf{g}_{primär}$), die alle nicht-spekularen Beleuchtungsanteile enthält und eine sekundäre Farbe ($\mathbf{g}_{sekundär}$), die alle spekularen Anteile enthält. Während des Textur-Mappings (Kapitel 13) wird nur die primäre Farbe ($\mathbf{g}_{primär}$) mit der Texturfarbe kombiniert. Danach wird auf die kombinierte Farbe die sekundäre Farbe ($\mathbf{g}_{sekundär}$) addiert und das Ergebnis auf den Wertebereich [0, 1] beschränkt. Die Beleuchtungsformeln zur Berechnung der primären und sekundären Farbe für einen Vertex lauten:

Vertexfarbe	3D-Anordnung	Lichtquelle		Material		Komponente
$\mathbf{g}_{primär} =$				$\mathbf{e}_{mat}$	$+$	emissiv
		$\mathbf{a}_{global}$	$*$	$\mathbf{a}_{mat}$	$+$	global ambient
	$\sum_{i=0}^{n-1} \left(\frac{1}{f_c+f_l\delta+f_q\delta^2}\right)_i (\max(\mathbf{v}\cdot\mathbf{k},\mathbf{0}))_i^{\mathbf{Spot_{Exp}}}$				$\cdot$	Faktor Nr. i
	$\Big\{$	$\mathbf{a}_{light}$	$*$	$\mathbf{a}_{mat}$	$+$	ambient Nr. i
	$\max(\mathbf{l}\cdot\mathbf{n},\mathbf{0})$ $\cdot$	$\mathbf{d}_{light}$	$*$	$\mathbf{d}_{mat}$	$\Big\}_i$	diffus Nr. i

(12.18)

Vertexfarbe	3D-Anordnung	Lichtquelle		Material		Komponente
$\mathbf{g}_{sekundär} =$	$\sum_{i=0}^{n-1} \left(\frac{1}{f_c+f_l\delta+f_q\delta^2}\right)_i (\max(\mathbf{v}\cdot\mathbf{k},\mathbf{0}))_i^{\mathbf{Spot_{Exp}}}$				$\cdot$	Faktor Nr. i
	$\Big\{(\max(\mathbf{h}\cdot\mathbf{n},\mathbf{0}))^{\mathbf{S}}$ $\cdot$	$\mathbf{s}_{light}$	$*$	$\mathbf{s}_{mat}$	$\Big\}_i$	spekular Nr. i

(12.19)

Die Addition der separaten spekularen Farbe nach dem Textur-Mapping führt zu deutlich besser sichtbaren Glanzlichteffekten.

Um zum normalen Beleuchtungsmodus zurück zu kehren, wird der Befehl `glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL, GL_SINGLE_COLOR)` aufgerufen. Falls ein Objekt keine Texturen enthält, sollte man beim normalen Beleuchtungsmodus bleiben, da er einfacher und somit schneller ist.

12.2 Schattierungsverfahren

Mit den im vorigen Abschnitt (12.1) beschriebenen Beleuchtungsmodellen ist es möglich, die Farbe eines beliebigen Punktes auf der Oberfläche eines Objektes zu berechnen. Aufgrund der enormen Komplexität der Beleuchtungsmodelle, die selbst in der stark vereinfachten Form des Standard-Beleuchtungsmodells ((12.17)) noch sehr hoch ist, wird die Beleuchtungsformel zur Bestimmung der Farbe von Objekten nur an den Eckpunkten der Polygone, d.h. an den Vertices ausgewertet. Dies passt auch mit der polygonalen Approximation von real gekrümmten Objekten zusammen. Nachdem die Farben an den relativ wenigen Vertices (1.000 - 100.000) berechnet sind, werden die Farbwerte für die große Anzahl an Fragmenten bzw. Pixeln (~1.000.000) eines Bildes mit Hilfe einfacher und daher schneller Schattierungsverfahren (*shading*) interpoliert. Als Standard-Verfahren haben sich hier das primitive *Flat-Shading* und das zwischen den Vertexfarben linear interpolierende *Smooth-* oder *Gouraud-shading* etabliert (Bild 12.15-a).

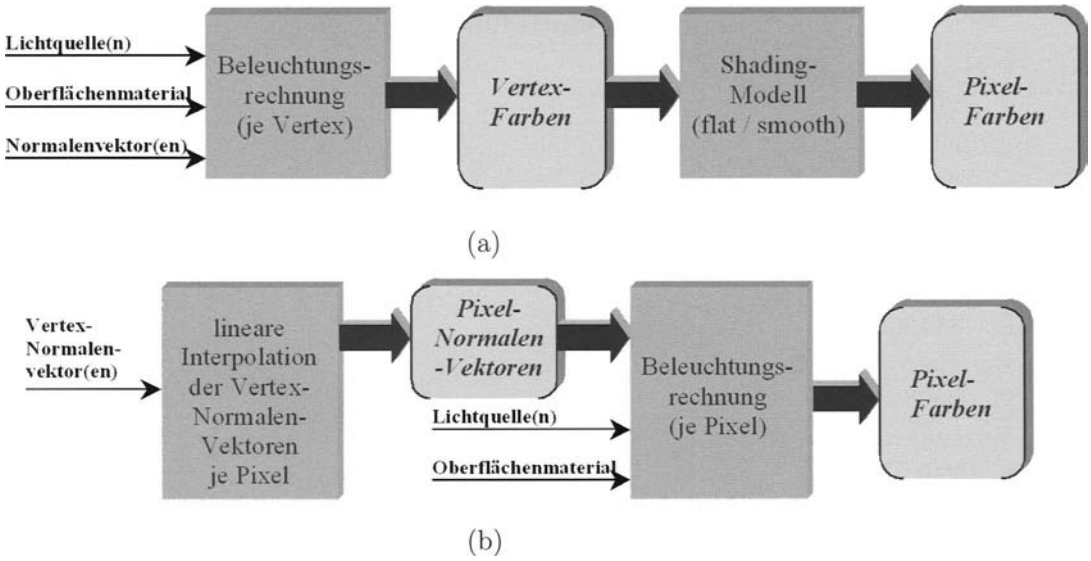


Bild 12.15: *Rendering-Pipeline* für die Beleuchtung und Schattierung: (a) Standardverfahren für die Beleuchtung und Schattierung. (b) *Phong-Shading*.

Das letzte vorgestellte Schattierungsverfahren, das sogenannte *Phong-Shading*, weicht von den beiden Standard-Schattierungsverfahren ab. Denn bei diesem Verfahren werden in der Rasterisierungsstufe der *Rendering-Pipeline* zunächst die Normalenvektoren zwischen den Vertices linear interpoliert und anschließend wird die Beleuchtungsformel für jedes Pixel berechnet (Bild 12.15-b). Der Rechenaufwand für das Phong-Shading liegt in der Regel um mehrere Größenordnungen über den Standard-Schattierungsverfahren. Mit der

neuesten Hard- und Software, auf die im letzten Abschnitt dieses Kapitels eingegangen wird, ist es möglich, *Phong-Shading* in Echtzeit zu realisieren.

Vorsichtshalber sei hier noch einmal darauf hingewiesen, dass Schattierung in diesem Zusammenhang nicht heißt, dass Objekte einen Schatten werfen, sondern nur, dass die pixel-bezogenen Farben für die Objekte einer Szene berechnet werden.

12.2.1 Flat-Shading

Das *Flat-Shading* ist das einfachste Schattierungsverfahren, das man sich vorstellen kann. Das Beleuchtungsmodell wird nur an einem einzigen Punkt der ganzen Facette (ein Dreieck, Viereck oder Polygon) ausgewertet und der berechnete Farbwert wird auf alle Pixel dieser Facette kopiert. Der ausgewählte Punkt ist in der Regel der erste oder der letzte Vertex der Facette, da diese Punkte bereits vorliegen. In der folgenden Tabelle ist aufgelistet, welcher Vertex die Farbe des jeweiligen Grafik-Primitivs bestimmt:

Grafik-Primitiv	**Vertex, der die Farbe der i-ten Facette des Grafik-Primitivs bestimmt**
Polygon `GL_POLYGON`	1
Einzelne Dreiecke `GL_TRIANGLES`	$3i$
Verbundene Dreiecke `GL_TRIANGLE_STRIP`	$i+2$
Dreiecks-Fächer `GL_TRIANGLE_FAN`	$i+2$
Einzelne Vierecke `GL_QUADS`	$4i$
Verbundene Vierecke `GL_QUAD_STRIP`	$2i+2$

Da beim *Flat-Shading* sowieso nur an einem Vertex die Beleuchtungsrechnung durchgeführt wird, ist es auch vollkommen ausreichend, einen einzigen Normalenvektor gemäß (12.13) und (12.14) zu berechnen und dem relevanten Vertex zuzuordnen. In diesem Fall spricht man von *Flächen-Normalenvektoren*, da nur ein Normalenvektor pro Fläche definiert bzw. verwendet wird und dieser auch, wie man im mathematischen Sinne erwarten würde, senkrecht auf der Fläche steht. Sollte allen Vertices ein eigener Normalenvektor zugewiesen werden, wie bei dem im nächsten Abschnitt dargestellten Gouraud-Shading, werden die überflüssigen Normalenvektoren einfach ignoriert.

In OpenGL wird der Rendering-Zustand *Flat-Shading* durch folgenden Befehl aktiviert:

```
glShadeModel(GL_FLAT);
```

Die Zuordnung eines Normalenvektors zu einem Vertex geschieht innerhalb einer `glBegin()`-`glEnd()`-Klammer durch einen vorausgestellten `glNormal3*()`-Befehl:

```
glBegin(GL_POLYGON);
    glNormal3f(0.0,0.0,1.0);
    glVertex3f(1.0,1.0,0.0);
    glVertex3f(0.2,0.5,0.0);
    glVertex3f(0.8,0.3,0.0);
glEnd();
```

Das hervorstechende Merkmal von Bildern, die mit *Flat-Shading* gerendert wurden, ist das facettenartige Aussehen der Objekte (Bild 12.16-$a_{1,2}$). Man sieht jede einzelne Facette, aus der ein Objekt aufgebaut ist, besonders deutlich, da jede Facette ja eine einheitliche Farbe erhält. Beim Übergang von einer Facette zur nächsten tritt meist ein Sprung im Farb- bzw. Helligkeitsverlauf auf, den die menschliche Wahrnehmung durch eine differenzierende Vorverarbeitung verstärkt wahrnimmt (der sogenannte „Mach-Band-Effekt"). Dadurch werden auch kleine Farb- oder Helligkeitssprünge noch sehr deutlich wahrgenommen, wie in Bild 12.16-a_2 zu sehen ist. Erst wenn die Tesselierung der Objekte so fein ist, dass jedes Polygon nur noch ein oder wenige Pixel groß ist, verschwindet das facettierte Aussehen (Bild 12.16-a_3). In diesem Grenzfall unterscheidet sich das Flat-Shading praktisch nicht mehr vom Gouraud- bzw. Phong-Shading, da für jedes Pixel ($\approx$ Facette) ein Normalenvektor vorhanden ist und die Beleuchtungsrechnung ausgeführt wird.

Das *Flat-Shading* ist das einfachste und daher schnellste der drei vorgestellten Schattierungsverfahren. Deshalb wird es nach wie vor in Einsatzbereichen verwendet, in denen keine Hardwarebeschleunigung für die Beleuchtung und Schattierung zur Verfügung steht, oder z.B. in Entwurfsansichten von CAD-Programmen, um den polygonalen Aufbau der Konstruktion gut zu erkennen.

12.2.2 Smooth-/Gouraud-Shading

Der Ausgangspunkt der Überlegungen beim *Smooth-Shading*, das alternativ auch nach dem Namen des Erfinders *Gouraud-Shading* [Gour71] genannt wird, ist die Beseitigung der Sprünge im Farb- bzw. Helligkeitsverlauf, die so störend wirken. Die einfachste Möglichkeit, einen stetigen Farbverlauf zu erzeugen, ist die lineare Interpolation der Farbwerte zwischen den Facetten. Aber an welcher Stelle der jeweiligen Facette soll man den Farbwert bestimmen, damit man überall einen glatten Farbverlauf (*Smooth-Shading*) zwischen den Facetten erreicht? Hier setzte die Idee Gouraud's ein: ausgehend von den sowieso vorhandenen Vertex-Koordinaten der Polygone werden zunächst wieder mit Hilfe der Gleichungen (12.13) und (12.14) die Flächen-Normalenvektoren berechnet; um jetzt den Sprung im Farbverlauf an den Facettenrändern zu vermeiden, wird durch Mittelung aller im betreffenden Eckpunkt angrenzenden Flächen-Normalenvektoren ein sogenannter

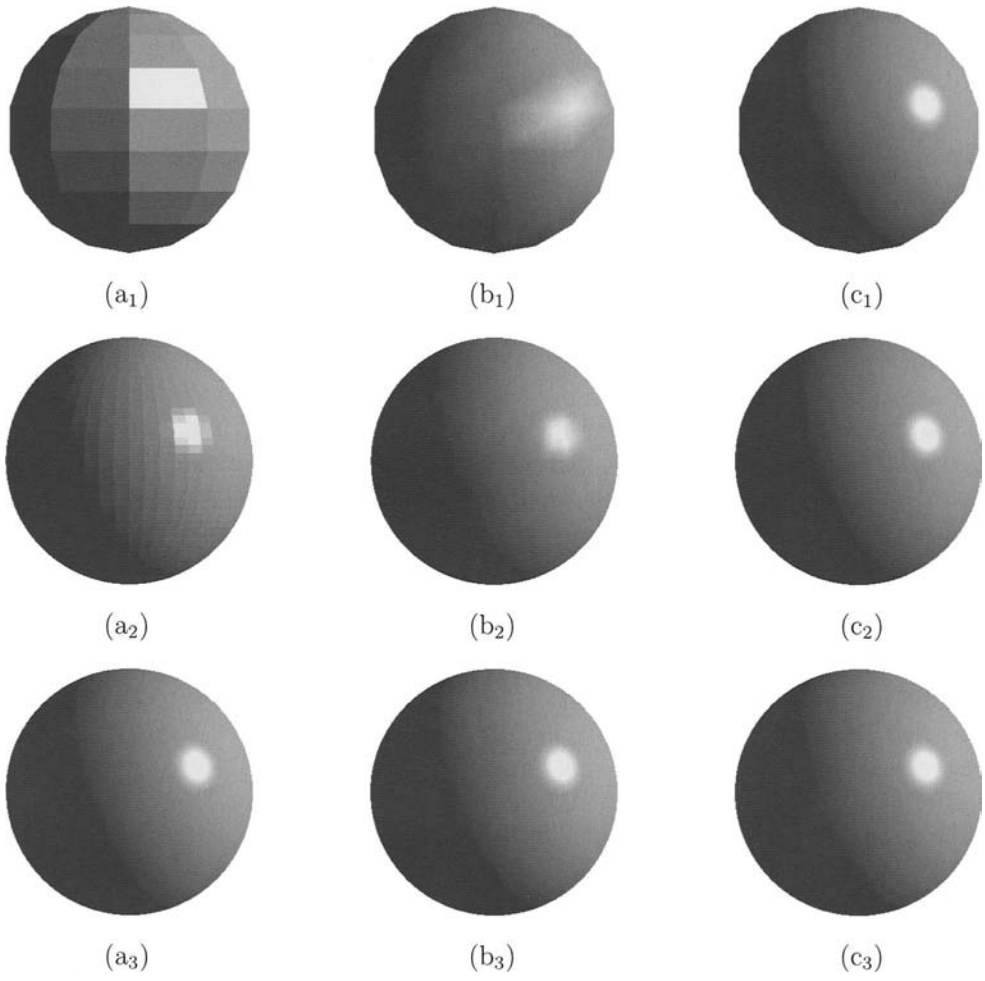


Bild 12.16: Schattierungsverfahren versus Tesselierung: (a) *Flat-Shading.* (b) *Smooth-/Gouraud-Shading.* (c) *Phong-Shading.* (obere Reihe, Index 1) grobe Tesselierung 81 Polygone. (mittlere Reihe, Index 2) mittlere Tesselierung 2500 Polygone. (untere Reihe, Index 3) feine Tesselierung 250000 Polygone.

Vertex-Normalenvektor bestimmt, der entgegen der üblichen mathematischen Definition eines „Normalenvektors" auf keiner der angrenzenden Facetten senkrecht[1] steht (Bild 12.18). Falls neben der polygonalen Approximation auch die analytische Beschreibung einer ge-

[1] bis auf den uninteressanten Fall, dass alle benachbarten Facetten in einer Ebene liegen

krümmten Oberfläche bekannt ist, können die „korrekten“ Normalenvektoren direkt analytisch berechnet werden. Im nächsten Schritt wird die Beleuchtungsrechnung für jeden Vertex mit den „korrekten“ *Vertex-Normalenvektoren* durchgeführt, so dass als Ergebnis ein Farbwert pro Vertex heraus kommt (*Per-Vertex-Lighting*). Abschließend werden die Farbwerte der einzelnen Pixel $\mathbf{g}_p$ durch *lineare Interpolation der Farbwerte* zwischen den Vertices entlang der Facettenkanten bestimmt und danach zwischen den Kanten entlang der sogenannten *Scan Lines* (Bild 12.17):

$$\begin{aligned} \mathbf{g}_a &= \mathbf{g}_1 - (\mathbf{g}_1 - \mathbf{g}_2) \cdot \frac{y_1 - y_s}{y_1 - y_2} \\ \mathbf{g}_b &= \mathbf{g}_1 - (\mathbf{g}_1 - \mathbf{g}_3) \cdot \frac{y_1 - y_s}{y_1 - y_3} \\ \mathbf{g}_p &= \mathbf{g}_b - (\mathbf{g}_b - \mathbf{g}_a) \cdot \frac{x_b - x_p}{x_b - x_a} \end{aligned} \tag{12.20}$$

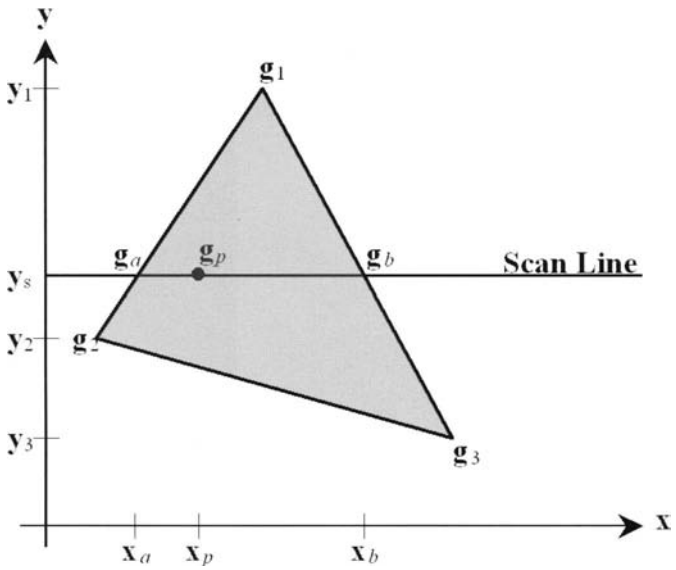


Bild 12.17: Der *Scan Line*-Algorithmus zur Berechnung der Pixel-Farbwerte: lineare Interpolation der Vertex-Farben entlang der Polygonkanten und *Scan Lines*.

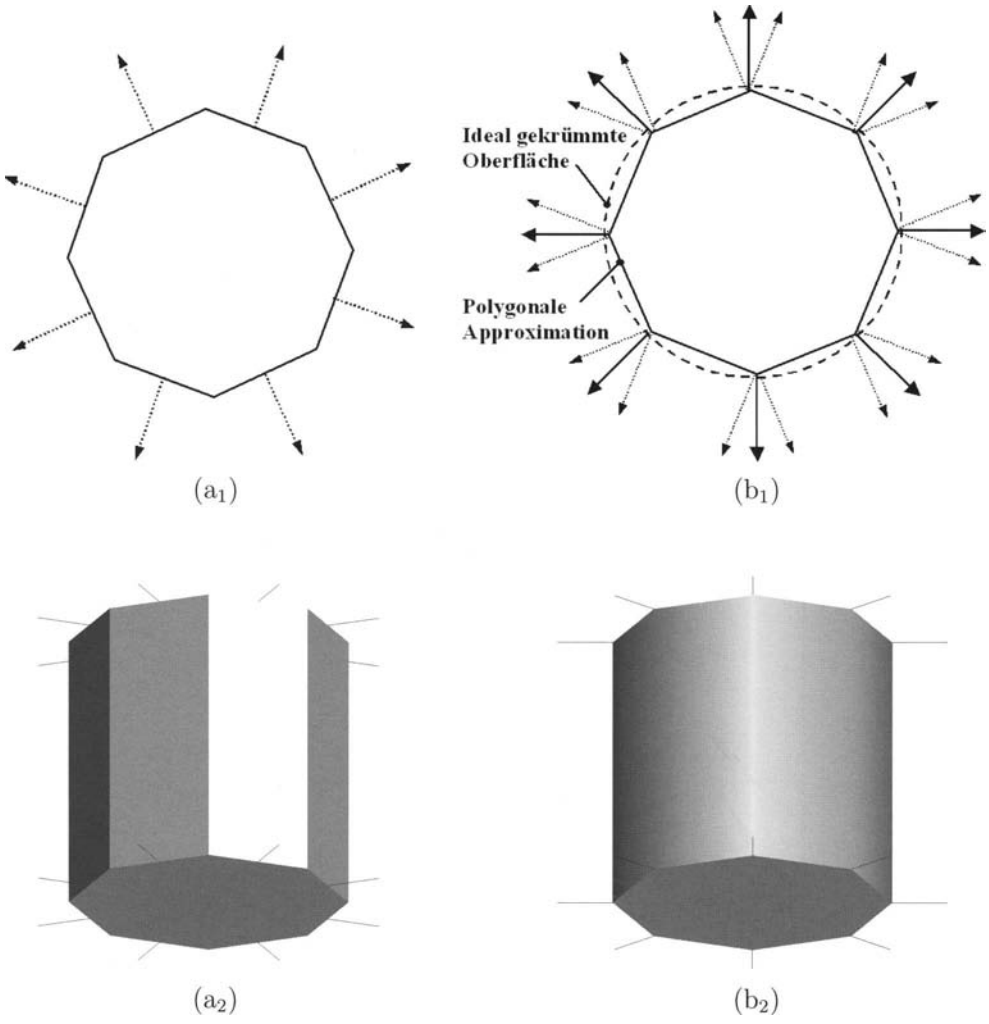


Bild 12.18: Flächen-Normalen versus Vertex-Normalen am Beispiel eines grob tesselierten Zylinders: (a_1) Flächen-Normalenvektoren (gepunktet), die senkrecht auf den Polygonen (durchgezogen) stehen. (a_2) 3D-Ansicht des Zylinders mit Flächen-Normalenvektoren: *Gouraud-Shading* liefert ebenso wie *Flat-Shading* Farb- bzw. Helligkeitssprünge an den Polygonkanten. (b_1) Vertex-Normalenvektoren (durchgezogen), die durch Mittelung aus den Flächen-Normalenvektoren (gepunktet) des Zylindermantels gewonnen wurden. Die Vertex-Normalenvektoren stehen senkrecht auf der idealen Zylinderoberfläche (gestrichelt), aber auf keiner Facette der polygonalen Approximation (durchgezogen). (b_2) 3D-Ansicht des Zylinders mit Vertex-Normalenvektoren: *Gouraud-Shading* läßt den Zylindermantel rund erscheinen, da benachbarte Polygonkanten die gleichen Normalenvektoren besitzen und zusammen mit der linearen Interpolation der Farbwerte zwischen den Vertices Farb- bzw. Helligkeitssprünge vermieden werden.

In OpenGL wird der Rendering-Zustand *Smooth-Shading* durch folgenden Befehl aktiviert:

```
glShadeModel(GL_SMOOTH);
```

Im Gegensatz zum *Flat-Shading* wird beim *Smooth- oder Gouraud-Shading* jedem Vertex ein eigener *Vertex-Normalenvektor* zugeordnet:

```
glBegin(GL_POLYGON);
    glNormal3f(0.5,0.5,0.707);
    glVertex3f(1.0,1.0,0.0);
    glNormal3f(-0.5,0.5,0.707);
    glVertex3f(0.2,0.5,0.0);
    glNormal3f(0.5,-0.5,0.707);
    glVertex3f(0.8,0.3,0.0);
glEnd();
```

Auf den ersten Blick sind durch das *Gouraud-Shading* die Probleme des *Flat-Shadings* behoben: die Farb- bzw. Helligkeitssprünge sind eliminiert, so dass selbst polygonal approximierte Oberflächen rund erscheinen (Bild 12.18-b_2). Beim *Gouraud-Shading* reicht in der Regel bereits eine mittlere Tesselierung aus, um eine akzeptable Bildqualität zu erreichen (Bild 12.16-b_2). Insbesondere bei zu grober Tesselierung der Objekte treten jedoch auch beim *Gouraud-Shading* eine Reihe von Problemen auf:

- Glanzlichter werden nicht korrekt dargestellt. Diese entstehen ja, wenn der Augenpunktsvektor sehr nahe bei der idealen Reflexionsrichtung liegt. Da die Beleuchtungsformel beim *Gouraud-Shading* aber nur in den Eckpunkten der Polygone berechnet wird und die Farben dazwischen linear interpoliert werden, sind die Glanzlichter sehr häufig gar nicht zu sehen. Aber selbst wenn man gerade ein Glanzlicht erwischt hat (Bild 12.16-b_1), erscheint der Farbverlauf nicht glatt, sondern sternförmig, da unsere visuelle Wahrnehmung auch Sprünge in der ersten Ableitung des Farb- bzw. Helligkeitsverlaufs verstärkt. Noch problematischer wird es, wenn Bewegung ins Spiel kommt, denn dann blitzt das Glanzlicht abwechselnd auf und ab. Abhilfe kann hier entweder durch eine höhere Tesselierung wie in Bild 12.16-$b_2, 3$ geschaffen werden (falls die Rechenkapazität dies zulässt), oder durch den Wegfall der spekularen Lichtkomponente.

- Die Silhouette der Objekte bleibt immer so eckig, wie das polygonale Modell (Bild 12.16-b_1 und -b_2).

- Bei Spotlights und lokalen Lichtquellen ist der Rand des Lichtkegels oft sehr „ausgefranst“, da hier die polygonale Struktur des Modells deutlich sichtbar wird (Bild 12.19).

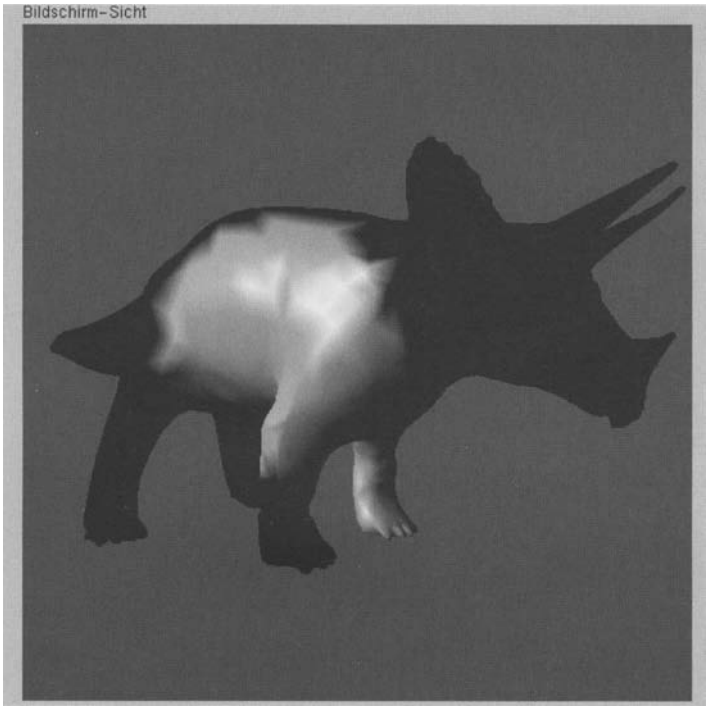


Bild 12.19: Probleme beim *Gouraud-Shading* mit *Spotlights*: der Rand des Lichtkegels ist durch zu große Polygone oft sehr „ausgefranst".

Vorsicht ist auch bei der Berechnung von Vertex-Normalenvektoren aus Flächen- Normalenvektoren geboten. Ab einem bestimmten Winkel zwischen den Flächen-Normalenvektoren macht es keinen Sinn mehr zu mitteln, und manchmal möchte man Kanten auch sehen können, wie z.B. die Kante zwischen Mantel und Deckel bei dem Zylinder in Bild 12.18-b_2. Man sollte also bei einem Algorithmus zur Berechnung von Vertex-Normalenvektoren einen Grenzwinkel vorsehen, ab dem Flächen-Normalenvektoren nicht mehr zur Mittelung herangezogen werden.

Im praktischen Einsatz ist das *Smooth-* oder *Gouraud-Shading* bis heute dominierend, da es bei realen Anwendungen mit komplexeren Szenen von der Bildqualität her meistens ausreichend ist und dennoch aufgrund seiner geringer Komplexität die Echtzeitanforderungen erfüllt.

12.2.3 Phong-Shading

Beim *Phong-Shading* [Phon75] werden in der Rasterisierungsstufe der *Rendering-Pipeline* (Bild 5.1), in der die Umrechnung von vertex-bezogenen in pixel-bezogene Daten stattfindet, zunächst die Pixel-Normalenvektoren durch *lineare Interpolation zwischen den Vertex-Normalenvektoren* gewonnen. Dabei geht man von den gleichen Vertex-Normalenvektoren aus, die auch beim *Gouraud-Shading* Verwendung finden. Der ebenfalls vom *Gouraud-Shading* bereits bekannte *Scan Line*-Algorithmus (12.20) wird hier auf die Normalenvektoren angewendet, allerdings mit einer Ergänzung: Sowohl nach der linearen Interpolation entlang der Facettenkanten, als auch nach der linearen Interpolation entlang der *Scan Lines* (Bild 12.17) müssen die Normalenvektoren gemäß (12.14) normiert werden. Das Ergebnis dieses ersten Schritts sind Normalenvektoren der Länge 1 für jedes Pixel, die *Pixel-Normalen* (Bild 12.20). Im zweiten Schritt wird die Beleuchtungsrechnung für jedes Pixel mit diesen *Pixel-Normalen* durchgeführt, so dass als Endergebnis wieder ein Farbwert pro Pixel bzw. pro Fragment heraus kommt (*Per-Pixel-* bzw. *Per-Fragment-Lighting*).

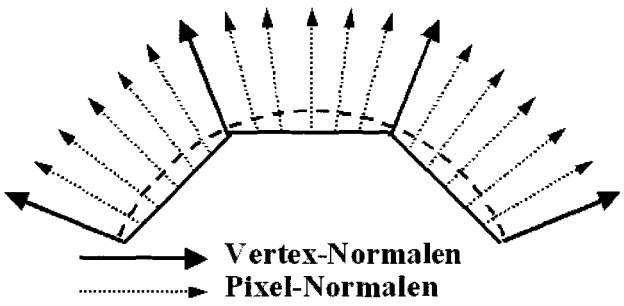


Bild 12.20: Grundprinzip des *Phong-Shadings*: Berechnung der Pixel-Normalenvektoren durch lineare Interpolation zwischen den Vertex-Normalenvektoren und Auswertung der Beleuchtungsformel an jedem Pixel.

Durch das *Phong-Shading* werden die gravierendsten Probleme des *Gouraud-Shading* beseitigt. Aufgrund der pixel-bezogenen Beleuchtungsrechnung werden selbst bei grober Tesselierung Glanzlichter nicht mehr verpasst und die Ränder eines *Spotlight*-Kegels glatt dargestellt. Nur die eckige Silhouette lässt noch auf die geringe Zahl an verwendeten Polygonen schließen (Bild 12.16-c_1). Außerdem ist das *Phong-Shading* mit seiner pixel-bezogenen Berechnung von Normalenvektoren und Beleuchtungsmodell die Basis für fortgeschrittene Texture-Mapping-Verfahren wie z.B. Bump-Mapping (Kapitel 13).

An dieser Stelle lohnt sich allerdings ein ganz allgemeiner Blick auf den Rechenaufwand und die erzielbare Bildqualität bei den verschiedenen Schattierungsverfahren. Wie in Bild 12.16 zu sehen ist, kann man durch eine ausreichend feine Tesselierung der Objekte mit dem einfachen *Flat-Shading* die gleiche hohe Bildqualität erreichen, wie bei beim *Phong-Shading* mit niedriger Tesselierung. Bei Anwendung des Gouraud-Shadings reicht ein mittlerer Tesselierungsgrad aus, um eine vergleichbare Bildqualität zu erreichen. Schattierungsverfahren

und Tesselierung sind also austauschbar im Hinblick auf die Bildqualität: je einfacher das Schattierungsverfahren, desto feiner muss die Tesselierung sein (bei bekannter Auflösung) oder umgekehrt: je aufwändiger das Schattierungsverfahren, desto geringer kann die Anzahl der verwendeten Polygone sein. Es bleibt also die Frage des Rechenaufwands bei gleicher Bildqualität, die hier mit einer groben Abschätzung beantwortet wird. Für *Flat-Shading* bedeutet dies, dass praktisch pro Pixel ein Polygon (Dreieck) vorhanden sein muss. Das heißt aber andererseits auch, dass nicht nur das Beleuchtungsmodell ähnlich häufig wie beim *Phong-Shading* ausgewertet werden muss, sondern Modell- und Augenpunktstransformationen sowie die Projektions- und Viewporttransformationen müssen für sehr viel mehr Vertices als beim Phong-Shading durchgeführt werden. Unter dem Strich ist also *Flat-Shading* bei gleich hoher Bildqualität rechenaufwändiger als *Phong-Shading*. Aufgrund der linearen Farbinterpolation reicht es beim *Gouraud-Shading* aus, wenn die Tesselierung ca. 10 - 100 Mal geringer ist, als beim *Flat-Shading*. Das aufwändige Beleuchtungsmodell muss beim *Gouraud-Shading* pro Dreieck drei Mal ausgewertet werden und beim *Flat-Shading* ein Mal. Da die lineare Farbinterpolation kaum Rechenzeit kostet, ist das *Gouraud-Shading* letztlich ca. 3 - 30 Mal effizienter als das *Flat-Shading*. *Phong-Shading* erlaubt zwar eine weitere Reduktion der Tesselierung gegenüber *Gouraud-Shading*, die aber wegen der immer sichtbaren Silhouette in der Regel auf einen Faktor < 10 begrenzt bleibt. Die Einsparung an Geometrietransformationen beim *Phong-Shading* wird aber durch die Auswertung des Beleuchtungsmodells pro Pixel wieder weit mehr als aufgewogen, so dass sich am Ende das *Gouraud-Shading* als das effizienteste Verfahren erweist. Aus diesem Grund ist eine Kombination aus *Gouraud-Shading* und dem Beleuchtungsmodell gemäß (12.17) der Standard bei den meisten Grafiksystemen. Derzeit findet allerdings ein gewisser Umbruch statt, da mittlerweile die Grafikhardware so enorm an Leistungsfähigkeit zugelegt hat, dass *Phong-Shading* in Echtzeit möglich und als Basis für anspruchsvolle Texture-Mapping-Verfahren wie z.B. Bump-Mapping (Kapitel 13) auch nötig ist.

12.3 Programmierbare Shader

Die Kombination aus Standard-Beleuchtungsmodell (12.17) und *Gouraud-Shading* wird heutzutage bei den meisten Grafikkarten in Hardware gegossen und damit soweit beschleunigt, dass die Bildgenerierung echtzeitfähig wird. Diese „feste Verdrahtung" von Beleuchtung und Schattierung wird als *fixed function graphics pipeline* bezeichnet, weil der Grafik-Programmierer nur noch ein paar Parameter des Modells einstellen kann, aber keinen Einfluss auf die Programmierung des Beleuchtungsmodells und der *Shading*-Algorithmen hat. Mit der vierten Generation von *GPUs* (*Graphics Processing Units*) wie nVIDIA's GeForce FX Familie oder ATI's Radeon 9700, sowie den *Shading*-Programmiersprachen *Cg* (*C for graphics*) von nVIDIA, *HLSL* (*High Level Shading Language*) von Microsoft und OpenGL SL (*Shading Language*, OpenGL Version 1.5), die seit Ende 2002 bzw. 2003 verfügbar sind, wurde die Flexiblität durch sogenannte *Programmierbare Shader* wieder erhöht. Ein *Programmierbarer Shader* ist nicht nur ein Schattierungsverfahren, sondern ein ganz allgemeines kleines Programm, das durch den Download in die Grafikkarte bestimmte Teile

der in Hardware gegossenen *fixed function graphics pipeline* ersetzt, ohne dass dabei die Hardware-Beschleunigung verloren geht.

Man unterteilt die *Programmierbaren Shader* gemäß den Stufen der *Rendering Pipeline* (Bild 12.21) in *Vertex Shader* und *Pixel Shader* (auch *Fragment Shader* genannt). Die Vertex-Operationen umfassen der Reihe nach die Modell- und Augenpunkt- Transformationen, die Beleuchtungsrechnung (im Fall des *Gouraud-Shadings*) sowie die Projektionstransformation, das Clipping, die Normierung und die Viewport-Transformation. Mit einem *Vertex Shader* kann man folglich die Standard-Bleuchtungsformel (12.17) durch ein anderes Beleuchtungsmodell ersetzen und die Geometrietransformationen kann man so erweitern, dass komplexe Animationen durch Hardware-Beschleunigung echtzeitfähig werden (Kapitel 14). Zu den Pixel- oder Fragment-Operationen gehört unter anderem die Texturierung und die Nebelberechnung. Deshalb sind mit einem *Pixel Shader* extrem aufwändige

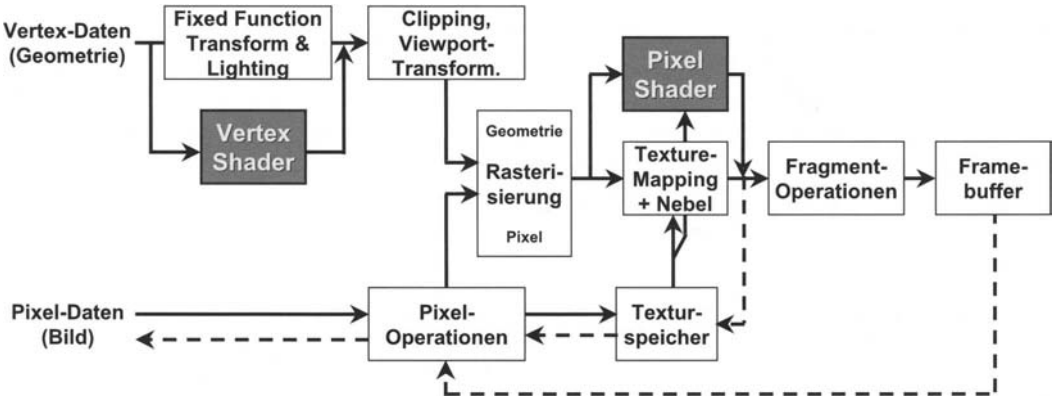


Bild 12.21: Die *Rendering Pipeline* mit *Vertex* und *Pixel Shadern* (grau dargestellt), die verschiedene Teile der *fixed function graphics pipeline* durch programmierbare Shader ersetzen. Ein *Vertex Shader* kann den *Fixed Function Transform & Lighting*-Teil ersetzen, ein *Pixel Shader* das Texture Mapping und die Nebelberechnung.

12.3.1 Shading Programmiersprachen

Derzeit existieren drei auf *GPUs* zugeschnittene *Shading*-Programmiersprachen, die einander sehr ähnlich sind: *Cg* (*C for graphics*) von nVIDIA, *HLSL* (*High Level Shading Language*) von Microsoft und OpenGL SL (*Shading Language*, OpenGL Version 1.5) von den Firmen, die sich im OpenGL *Architecture Review Board* (Kapitel 5) zusammengeschlossen haben. nVIDIA und Microsoft bildeten ein Konsortium zur Entwicklung von *Cg* und *HLSL*. Im Wesentlichen sind *Cg* und *HLSL* eine einzige *Shading*-Programmiersprache, die sich nur in den Namenskonventionen unterscheiden. *HLSL* ist Teil von Microsoft's DirectX Multimedia-Schnittstelle seit der Version 9 und als solcher nur unter Direct3D auf Windows-Betriebssystemen lauffähig. Das im Jahr 2002 vorgestellte *Cg* ist ein offener Standard, der unabhängig von der Grafik-Programmierschnittstelle, d.h. sowohl unter OpenGL als auch unter Direct3D, eingebunden werden kann. *Cg* ist damit unabhängig vom Betriebssystem und von der verwendeten Grafik-Hardware. Die Funktionalität und Wirkung von Cg-Shadern kann in den weit verbreiteten Datenbasisgenerier-Werkzeugen (*Digital Content Creation Tools*) „3d studio max" von discreet, „MAYA" von Alias|Wavefront und „Softimage|XSI" von AVID Technology bereits benutzt und getestet werden. Bei OpenGL SL wurde die Spezifikation erst Ende 2003 zur Verfügung gestellt und es existiert derzeit nur eine Entwicklungsumgebung mit Treibern für ATI-Grafikkarten. Ein Vergleich der OpenGL SL Spezifikation mit *Cg* zeigt aber auch hier eine sehr große Übereinstimmung in der Semantik. Das Beispiel der Realisierung eines Phong-Shaders im nächsten Abschnitt wird anhand von *Cg* dargestellt, da dies im Augenblick der einzig gangbare Weg vor dem OpenGL-Hintergrund des Buches ist. Eine Umsetzung in OpenGL SL oder HLSL sollte aufgrund der großen Ähnlichkeit der *Shading*-Programmiersprachen keine Schwierigkeiten bereiten. Welche *Shading*-Programmiersprache sich am Ende durchsetzt, muss die Zukunft erweisen.

Design Prinzipien von Cg:

Cg-Programme, d.h. *Vertex* oder *Pixel Shader*, sind keine normalen Programme, mit denen man es in Hochsprachen wie z.B. C/C++ oder Java üblicherweise zu tun hat. Ein normales Programm wird vom Programmierer aufgerufen, wenn es ausgeführt werden soll, und mögliche Eingabedaten werden ebenfalls vom Programmierer zur Verfügung gestellt oder von ihm gesteuert. Ein *Vertex* oder *Pixel Shader* dagegen ist ein Block der *Rendering-Pipeline*, durch welche die Daten zur Bildgenerierung geschleust werden. Als Eingabe werden in einen *Vertex Shader* ständig z.B. Vertices, Farbwerte, Normalenvektoren und Nebel- oder Texturkoordinaten vom OpenGL-Programm eingespeist, das Vertex-Programm führt die entsprechenden Berechnungen aus (z.B. Koordinatentransformationen oder Beleuchtungsrechnungen) und liefert die Ergebnisse an die nächste Stufe der *Rendering-Pipeline*. Ein *Pixel Shader* wird für jedes Pixel von der Rasterisierungsstufe aufgerufen und erhält von dort seine Eingaben (z.B. Farbwerte, Texturkoordinaten), führt die entsprechenden Berechnungen aus (z.B. *Texture Mapping* oder Nebelberechnungen) und liefert die Ergebnisse (Farbwerte) an die vorletzte Stufe der *Rendering-Pipeline*. Cg-Programme sind

also eher vergleichbar mit Unterprogrammen, die von anderen Programmteilen mit entsprechenden Eingabeparametern aufgerufen werden und am Ende die Ergebnisse wieder zurück liefern. Bei der Parameterübergabe an ein Cg-Programm werden zwei Typen unterschieden: „*uniform*" und „*varying*". Eingaben vom Typ *varying* sind Werte, die von Vertex zu Vertex (in Vertex Shadern) oder von Pixel zu Pixel (in Pixel Shadern) variieren. Reservierte Cg-Codewörter, die in den Programm-Listings im nächsten Abschnitt immer fett gedruckt sind, kennzeichnen die Bedeutung der jeweiligen Eingabewerte: **POSITION** steht z.B. für Ortskoordinaten, **NORMAL** steht für Normalenvektoren, **TEXCOORD0** steht für Texturkoordinatensatz 0, **TEXCOORD1** steht für Texturkoordinatensatz 1 usw., **COLOR** steht für Farben. Eingaben vom Typ *uniform* dienen zur Übergabe von langsam veränderlichen Parametern aus dem OpenGL-Programm, wie z.B. der ModelView-Matrix, Eigenschaften einer Lichtquelle, Materialeigenschaften oder beliebigen anderen Parametern zur Steuerung von *Vertex* und *Pixel Shadern* aus dem OpenGL-Programm heraus. Bei einem *uniform*-Parameter muss das Cg-Codewort **uniform** vor der Vereinbarung stehen, ein Codewort zur Kennzeichnung der Bedeutung des Eingabewerts entfällt dafür. Bei einem *varying*-Parameter ist es genau umgekehrt: hier entfällt ein *varying*-Codewort, dafür benötigt man am Ende der Vereinbarung ein Codewort, das die Bedeutung des Eingabewerts festlegt. Die Ausgabewerte eines Cg-Programms werden durch das voran gestellte Codewort **out** gekennzeichnet, am Ende der Vereinbarung muss eines der Codewörter stehen, die auch schon bei *varying*-Parametern die Bedeutung der jeweiligen Ausgabewerte festlegen. Allerdings sind für die Ausgabe nicht alle Codewörter zulässig: **NORMAL** ist bei Vertex-Programmen nicht zulässig, da die Beleuchtungsrechnung bei den normalen OpenGL-Shadingverfahren (*flat* und *smooth*) im Rahmen der Vertex-Operationen durchgeführt wird und somit Normalenvektoren in den späteren Stufen der Rendering-Pipeline nicht mehr benötigt werden; bei Fragment-Programmen ist ausschließlich das Codewort **COLOR** für die Ausgabe zulässig, da alle Fragment-Operationen nur noch mit Farbwerten für jedes Fragment bzw. Pixel rechnen.

Cg basiert, wie der Name schon sagt (*C for graphics*) auf der Programmiersprache C. Das heißt, das Grundkonzept von Syntax und Semantik ist C-ähnlich und besitzt deshalb die üblichen Vorteile einer sehr weit verbreiteten höheren Programmiersprache. Zusätzlich gibt es einige Erweiterungen, die über den Sprachumfang von C hinausgehen:

- Datentypen für Grafik:
  - 2-, 3-, 4-dimensionale Vektortypen **float2**, **float3**, **float4**
  - Typen für 2x2-, 3x3-, 4x4-Matrizen **float2x2**, **float3x3**, **float4x4**
  - Datentypen für Zugriffe auf 1-, 2-, 3-dimensionale Texturen **sampler1D**, **sampler2D**, **sampler3D**, sowie auf *Cube Map*-Texturen **samplerCUBE** und *Non-power-of-two*-Texturen **samplerRECT**
- spezielle mathematische Operationen:
  - Vektoroperationen: Normierung von Vektoren (**normalize**), Skalarprodukt (**dot**)

und Vektorprodukt (**cross**), Berechnung von Reflexions- und Brechungsvektoren (**reflect, refract**)

- Matrizen-Operationen: Multiplikation (**mul**), Transponierung (**transpose**), Determinante (**determinant**)
- Sonstige mathematische Operationen, wie z.B. lineare Interpolation (**lerp**), Maximum- und Minimum-Funktionen (**max, min**), Potenzierung (**pow**), trigonometrische Funktionen, Texture-Mapping-Funktionen

- Überladung von Funktionen, d.h. mit einer einzigen Funktion können unterschiedliche Datentypen verarbeitet werden. Das Skalarprodukt (**dot**) kann z.B. für 2-, 3- und 4-dimensionale Vektoren benutzt werden, Cg ruft jedesmal die geeignete Version der (**dot**)-Funktion auf.

Die hier vorgestellten Sprachkonstrukte von Cg reichen für das Verständnis der in diesem Buch vorgestellten Cg-Programme aus. Für eine vollständige Darstellung des gesamten Sprachumfangs wird auf das Cg-Tutorial [Fern03] verwiesen.

Nicht alle Grafikkarten können den gesamten Sprachumfang von Cg verarbeiten. Manche lassen nur eingeschränkte *Vertex Shader* zu, aber keine *Pixel Shader*, andere lassen zwar beide *Shader*-Typen zu, implementieren aber nur einen Teil der mathematischen Funktionen. Um diese Vielfalt abzudecken, definiert man in Cg sogenannte „*Profile*" für die verschiedenen Hardware-Betriebssystem-API-Kombinationen, die den zur Verfügung stehenden Sprachumfang festlegen. Welche Kombination für ein Cg-Programm gerade zur Verfügung steht, kann durch eine Abfrage festgestellt werden, und abhängig davon können unterschiedliche Cg-Programmvarianten aufgerufen werden.

12.3.2 Realisierung eines Phong-Shaders in Cg

Zur Realisierung des *Phong-Shadings* in *Cg* (in Kombination mit OpenGL) muss man sich zunächst ein paar Gedanken zur *Rendering-Pipeline* (Bild 12.21) machen. Beim Phong-Shading findet die Beleuchtungsrechnung pro Pixel, d.h. im Fragment-Shader statt. Dazu benötigt man pixel-bezogene Normalenvektoren und Positionen, die normalerweise nach der Rasterisierung gar nicht mehr vorliegen, da beim Gouraud-Shading die vertex-bezogene Beleuchtungsrechnung vor der Projektionstransformation im Weltkoordinatensystem stattfindet. Deshalb wird ein Trick angewendet, um die Vertex-Normalenvektoren und Eckpunkte linear zu interpolieren und in das Fragment-Programm zu schaufeln: man tarnt sie als 3-dimensionale Texturkoordinaten, so dass die Rasterisierungsstufe mit Hilfe des *Scan Line*-Algorithmus (12.20) die Arbeit erledigt. Denn die Rasterisierungsstufe führt die lineare Interpolation nicht nur für Vertex-Farben durch, sondern auch für vertex-bezogene Texturkoordinaten. Danach liegen alle benötigten Daten im Fragment-Programm vor, um die Beleuchtungsrechnung pro Pixel durchzuführen und den resultierenden Farbwert auszugeben.

Für die Implementierung von „echtem" *Phong-Shading* müssen mindestens drei verschiedene Dateien editiert werden:

- Ein normales OpenGL-Programm muss so erweitert werden, dass die Cg-Programmteile (Runtime, Vertex-Shader, Pixel/Fragment-Shader) geladen und aktiviert werden. Außerdem muss hier die Parameterübergabe zwischen OpenGL und Cg organisiert werden.

- Ein *Vertex-Shader*-Programm, hier „vsPhong.cg" genannt, das die Berechnung der nötigen Transformationen durchführt. Darin werden die vom OpenGL-Programm stammenden Vertices mit Hilfe der aufeinander multiplizierten ModelView- und Projektions-Matrizen vom Objekt-Koordinatensystem ins Projektions-Koordinatensystem transformiert. Das Ergebnis wird der normalen OpenGL-Rendering-Pipeline für die weiteren Transformationsschritte übergeben (Clipping, Normierung, Viewport-Transformation). Zusätzlich werden die Vertices mit Hilfe der ModelView-Matrix vom Objekt-Koordinatensystem ins Welt-Koordinatensystem (in dem später die Beleuchtungsrechnung stattfindet) transformiert und anschließend als Texturkoordinate Nr. 1 ausgegeben. Die Vertex-Normalenvektoren werden mit Hilfe der invers transponierten ModelView-Matrix vom Objekt-Koordinatensystem ins Welt-Koordinatensystem transformiert und als Texturkoordinate Nr. 2 ausgegeben (Normalenvektoren verhalten sich unter Transformationen nicht wie gewöhnliche Vertices, sondern wie die Flächen, auf denen sie senkrecht stehen).

- Ein *Fragment-Shader*-Programm, hier „fsPhong.cg" genannt, das das eigentliche *Phong-Shading* durchführt. Als Input von der Rasterisierungsstufe stehen die pixelbezogenen Positionen und Normalenvektoren sowie die aus dem OpenGL-Programm stammenden Parameter für die Beleuchtungsrechnung zur Verfügung. Nach der Durchführung dieser Rechnung wird als Ergebnis der Farbwert pro Pixel bzw. Fragment an die normale OpenGL-Rendering-Pipeline für die weiteren Fragment-Operationen übergeben (z.B. z-Buffer-Test, Alpha-Blending, Accumulation Buffer).

Voraussetzung für das *Phong-Shading* ist eine Grafikhardware der vierten Generation, die ein Profil besitzt, das programmierbare Fragment-Shader unterstützt, wie z.B. die GeForce FX-Karten von nVIDIA mit dem Profil `CG_PROFILE_FP30`.

Ergänzungen in einem normalen OpenGL-Programm:

Im folgenden Programm-Listing sind die benötigten Ergänzungen in einem normalen OpenGL-Programm dargestellt, die eine Benutzung der Cg-Funktionalitäten erlauben. Die Ergänzungen sind in Abschnitte unterteilt, die typischerweise auch in normalen OpenGL-Programmen vorkommen (Deklarationen, Initialisierung, Display-Funktion, Resourcen-Freigabe). Die Bedeutung einzelner Befehle wird in den davor stehenden Kommentarzeilen erläutert.

```
// *** DEKLARATIONEN ***
// Einbindung der allgemeinen Cg-Funktionalitäten
#include <Cg/cg.h>
// Einbindung der OpenGL-spezifischen Cg-Funktionalitäten
#include <Cg/cgGL.h>

// Definition globaler OpenGL-Variablen
GLfloat Am[] = {0.2, 0.2, 0.2, 1.0};
GLfloat Dm[] = {0.7, 0.7, 0.7, 1.0};
GLfloat Sm[] = {1.0, 1.0, 1.0, 1.0};
GLfloat Shin = 50.0;
GLfloat Posl[] = {10.0, 0.0, 10.0, 1.0};
GLfloat Al[] = {1.0, 1.0, 1.0, 1.0};
GLfloat Dl[] = {1.0, 1.0, 1.0, 1.0};
GLfloat Sl[] = {1.0, 1.0, 1.0, 1.0};

// Definition globaler Cg-Variablen
CGcontext Context = NULL;
CGprogram vProgram = NULL;
CGprofile vProfile = CG_PROFILE_VP30;
CGprogram fProgram = NULL;
CGprofile fProfile = CG_PROFILE_FP30;

// OpenGL-Namen der an den Vertex-Shader zu übergebenden uniform-Parameter
CGparameter vModelView, vModelViewInvT, vModelViewProj;
// OpenGL-Namen der an den Fragment-Shader zu übergebenden uniform-Param.
CGparameter fEmission_mat, fAmbient_mat, fDiffuse_mat;
CGparameter fSpecular_mat, fShininess;
CGparameter fPosition_light;
CGparameter fAmbient_light, fDiffuse_light, fSpecular_light;
// *** ENDE DER DEKLARATIONEN ***


// *** INITIALISIERUNG ***
// lineare Interpolation bei der Rasterisierung
glShadeModel(GL_SMOOTH);

// Cg-Kontext initialisieren
Context = cgCreateContext();

// Bestimmung des Vertex Profils
vProfile = cgGLGetLatestProfile(CG_GL_VERTEX);
// setze optimization compiler Optionen
```

```
cgGLSetOptimalOptions(vProfile);
// Bestimmung des Fragment Profils
fProfile = cgGLGetLatestProfile(CG_GL_FRAGMENT);
// setze optimization compiler Optionen
cgGLSetOptimalOptions(fProfile);

// lade den Quellcode des Vertex-Shaders aus der Datei vsPhong.cg
vProgram = cgCreateProgramFromFile(Context, CG_SOURCE, vsPhong.cg,
           vProfile, NULL, NULL);
if(!vProgram) {cgDestroyContext(Context); exit(0);}
// lade den Quellcode des Fragment-Shaders aus der Datei fsPhong.cg
fProgram = cgCreateProgramFromFile(Context, CG_SOURCE, fsPhong.cg,
           fProfile, NULL, NULL);
if(!fProgram) {cgDestroyContext(Context); exit(0);}

// lade den Vertex- und Fragment-Shader in die Laufzeitumgebung
cgGLLoadProgram(vProgram);
cgGLLoadProgram(fProgram);

// hole die Zeiger für die uniform-Parameter des Vertex-Shaders
vModelView = cgGetNamedParameter(vProgram,"ModelView");
vModelViewInvT = cgGetNamedParameter(vProgram,"ModelViewInvT");
vModelViewProj = cgGetNamedParameter(vProgram,"ModelViewProj");
// hole die Zeiger für die uniform-Parameter des Fragment-Shaders
fEmission_mat = cgGetNamedParameter(fProgram,"e_mat");
fAmbient_mat = cgGetNamedParameter(fProgram,"a_mat");
fDiffuse_mat = cgGetNamedParameter(fProgram,"d_mat");
fSpecular_mat = cgGetNamedParameter(fProgram,"s_mat");
fShininess = cgGetNamedParameter(fProgram,"S\);
fPosition_light = cgGetNamedParameter(fProgram,"pos_light");
fAmbient_light = cgGetNamedParameter(fProgram,"a_light");
fDiffuse_light = cgGetNamedParameter(fProgram,"d_light");
fSpecular_light = cgGetNamedParameter(fProgram,"s_light");
// *** ENDE DER INITIALISIERUNG ***


// *** DISPLAY-FUNKTION ***
// aktiviere das Profil des Vertex-Shaders
cgGLEnableProfile(vProfile);
// lade den Vertex-Shader in die GPU der Grafikkarte
cgGLBindProgram(vProgram);
// aktiviere das Profil des Fragment-Shaders
cgGLEnableProfile(fProfile);
```

```
// lade den Fragment-Shader in die GPU der Grafikkarte
cgGLBindProgram(fProgram);

// Übergabe der ModelView-Matrix an den Vertex-Shader
cgGLSetStateMatrixParameter(vModelView,
CG_GL_MODELVIEW_MATRIX, CG_GL_MATRIX_IDENTITY);
// Übergabe der invertierten und transponierten ModelView-Matrix
cgGLSetStateMatrixParameter(vModelViewInvT,
CG_GL_MODELVIEW_MATRIX, CG_GL_MATRIX_INVERSE_TRANSPOSE);
// Übergabe des Produkts aus ModelView- und Projektions-Matrix
cgGLSetStateMatrixParameter(vModelViewProj,
CG_GL_MODELVIEW_PROJECTION_MATRIX, CG_GL_MATRIX_IDENTITY);

// Übergabe der Material- und Lichtquellendaten an den Fragment-Shader
cgGLSetParameter4f(fEmission_mat, 0.0, 0.0, 0.0, 1.0);
cgGLSetParameter4fv(fAmbient_mat, Am);
cgGLSetParameter4fv(fDiffuse_mat, Dm);
cgGLSetParameter4fv(fSpecular_mat,Sm);
cgGLSetParameter1fv(fShininess, Shin);
cgGLSetParameter4fv(fPosition_light, Pos1);
cgGLSetParameter4fv(fAmbient_light, A1);
cgGLSetParameter4fv(fDiffuse_light, D1);
cgGLSetParameter4fv(fSpecular_light,S1);

// zeichne alle Objekte in der Szene
draw_scene();

// Umschaltung auf fixed function graphics pipeline
cgGLDisableProfile(vProfile);
cgGLDisableProfile(fProfile);
// *** ENDE DER DISPLAY-FUNKTION ***


// *** RESOURCEN-FREIGABE ***
// Freigabe des Vertex-Programms
cgDestroyProgram(vProgram);
// Freigabe des Fragment-Programms
cgDestroyProgram(fProgram);
// Freigabe aller Cg-Programme
cgDestroyContext(Context);
// *** ENDE DER RESOURCEN-FREIGABE ***
```

Das Vertex-Programm vsPhong.cg:

Im Vertex-Programm finden beim *Phong-Shading* ausschließlich Transformationen statt, da die Beleuchtungsrechnung ja im Fragment-Programm per Pixel durchgeführt wird. Als Input vom OpenGL-Programm, der sich bei jedem Aufruf des Vertex-Programms ändert (varying-Parameter), wird die Position und der Normalenvektor des betrachteten Vertex übergeben. Als weiterer Input vom OpenGL-Programm, der sich selten ändert (uniform-Parameter), wird die ModelView-Matrix, die invers transponierte der ModelView-Matrix, sowie die Kombination aus ModelView- und Projektions-Matrix übergeben. Zunächst einmal müssen die Koordinaten des übergebenen Vertex mit Hilfe der kombinierten ModelViewProjektions-Matrix vom Objekt-Koordinatensystem in das Projektions-Koordinatensystem transformiert werden. Das Ergebnis ist der „`Position`"-Output des Vertex-Programms, der für die weiteren Berechnungen auf Vertex-Ebene (Clipping, Normierung, Viewport-Transformation) und für den *Scan Line*-Algorithmus (12.20) herangezogen wird. Für die Beleuchtungsrechnung im Fragment-Programm benötigt man aber die 3-dimensionalen Positionen und Normalenvektoren pro Pixel im Welt-Koordinatensystem. Deshalb transformiert man die Koordinaten des übergebenen Vertex mit Hilfe der ModelView-Matrix vom Objekt-Koordinatensystem in das Welt-Koordinatensystem. Normalenvektoren verhalten sich bezüglich Koordinatentransformationen nicht wie Vertices, sondern wie die Flächen, auf denen sie senkrecht stehen. Aus diesem Grund müssen die Vertex-Normalenvektoren mit der invers transponierten der ModelView-Matrix multipliziert werden, um vom Objekt-Koordinatensystem in das Welt-Koordinatensystem transformiert zu werden [Shre05]. Nun bleiben noch zwei Aufgaben zu erledigen: aus den vertexbezogenen Eckpunkten und Normalenvektoren im Welt-Koordinatensystem müssen pixelbezogene Positionen und Normalenvektoren berechnet werden und diese Daten müssen vom Vertex-Shader zum Fragment-Shader transferiert werden. Dazu bedient man sich eines Tricks, mit dem man beiden Aufgaben in einem Arbeitsgang erledigen kann: man deklariert die ins Welt-Koordinatensystem transformierten Eckpunkte und Normalenvektoren als 3-dimensionale Texturkoordinaten, und die Rasterisierungsstufe der *Rendering-Pipeline* erledigt die beiden Aufgaben. Denn Texturkoordinaten werden in OpenGL ebenfalls den Vertices zugeordnet (Kapitel 13) und der *Scan Line*-Algorithmus (12.20), der im Rasterizer auch für Texturkoordinaten ausgeführt wird, berechnet die Texturkoordinaten für jedes Pixel. Da es für den Rasterizer keine Rolle spielt, ob er „echte" Texturkoordinaten oder als solche deklarierte Eckpunkte und Normalenvektoren erhält, führt er die gewünschte lineare Interpolation aus und liefert pixel-bezogene Positionen und Normalenvektoren an das Fragment-Programm.

Das Vertex-Programm vsPhong.cg:

```
void main(          float4 position : POSITION,
                    float3 normal : NORMAL,

                out float4 oPosition : POSITION,
                out float3 objectPos : TEXCOORD0,
                out float3 oNormal : TEXCOORD1,

            uniform float4x4 ModelView,
            uniform float4x4 ModelViewInvT,
            uniform float4x4 ModelViewProj)
{
    // Vertex aus Objekt- in Projektionskoordinaten
    oPosition = mul(ModelViewProj, position);
    // Vertex aus Objekt- in Weltkoordinaten
    objectPos = mul(ModelView, position).xyz;
    // Vertex-Normale aus Objekt- in Weltkoordinaten
    oNormal = normalize(mul((float3x3)ModelViewInvT, normal).xyz);
}
```

Das Fragment-Programm fsPhong.cg:

Im Fragment-Programm finden die wesentlichen Berechnungen des Beleuchtungsmodells statt. Als Input vom Rasterizer kommen die für jedes Pixel linear interpolierten varying-Parameter Position und Normalenvektor (jeweils in Weltkoordinaten). Als Input vom OpenGL-Programm kommen die uniform-Parameter für die Material- und Lichtquelleneigenschaften. Zunächst müssen die durch den Rasterizer für jedes Pixel linear interpolierten Normalenvektoren normiert werden. Anschließend werden die beiden einfachsten Lichtanteile, der emissive und der ambiente Anteil, bestimmt. Danach wird im Falle einer lokalen Lichtquelle der Lichtvektor von jedem Pixel zur Lichtquelle berechnet. Das Skalarprodukt von normiertem Lichtvektor und normiertem Pixel-Normalenvektor ergibt den Cosinus des Winkels zwischen diesen beiden Vektoren, der für das Lambert'sche Modell einer ideal diffusen Oberfläche relevant ist. Die Maximum-Funktion sorgt dafür, dass Oberflächen nicht von hinten beleuchtet werden. Der diffuse Beleuchtungsanteil ergibt sich aus dem Produkt von diffusen Reflexionskoeffizienten des Materials mit dem ausgestrahlten Licht und dem Lambert'schen Faktor. Für die Berechnung des spekularen Lichtanteils benötigt man noch den *Halfway-Vektor*, der sich als Winkelhalbierender zwischen dem Lichtvektor und dem Augenpunktsvektor ergibt (standardmäßig wird in OpenGL für die Beleuchtungsrechnung ein infiniter Augenpunkt angenommen, so dass der in die positive z-Richtung zeigende Augenpunktsvektor fix ist). Das durch die Maximum-Funktion positiv definite Skalarprodukt von normiertem *Halfway-Vektor* und normiertem Pixel-Normalenvektor ergibt den Cosinus

des Winkels zwischen diesen beiden Vektoren, der, potenziert mit dem Shininess-Faktor, für das Phongsche Modell einer real spiegelnden Oberfläche relevant ist. Da der spekulare Lichtanteil verschwinden muss, wenn der Lambert'sche Faktor null ist (denn in diesem Fall wird ja die Oberfläche von hinten beleuchtet), wird in diesem Fall der spekulare Faktor ebenfalls null gesetzt. Der spekulare Beleuchtungsanteil ergibt sich aus dem Produkt von spekularen Reflexionskoeffizienten des Materials mit dem ausgestrahlten Licht und dem spekularen Faktor. Die endgültige Farbe des Pixels ist die Summe aus emissivem, ambientem, diffusem und spekularem Lichtanteil. Die vierte Farbkomponente des Pixels, der Alpha-Wert, wird auf 1 (d.h. opak) gesetzt. Der einzige Output des Fragment-Programms ist der Farbwert des Pixels, der dann im Rahmen der weiteren Fragment-Operationen (wie z.B. z-Buffer-Test, Alpha-Blending, Accumulation Buffer) noch verändert werden kann.

Das Fragment-Programm fsPhong.cg:

```
void main(          float3 position : TEXCOORD0,
                    float3 normal : TEXCOORD1,
                out float4 color : COLOR,
            uniform float4 e_mat,
            uniform float4 a_mat,
            uniform float4 d_mat,
            uniform float4 s_mat,
            uniform float S,
            uniform float4 pos_light,
            uniform float4 a_light,
            uniform float4 d_light,
            uniform float4 s_light)
{
    float3 N = normalize(normal);

    // Berechne den emissiven Term
    float4 emissive = e_mat;

    // Berechne den ambienten Term
    float4 ambient = a_light * a_mat;

    // Berechne den diffusen Term
    float3 L;
    if(pos_light.w == 0) {
        L = pos_light.xyz;
    } else {
        L = normalize(pos_light.xyz - position);
    }
```

```
    float diffuseLight = max(dot(L, N), 0);
    float4 diffuse = diffuseLight * d_light * d_mat;

    // Berechne den spekularen Term
    // Der Vektor zum Augenpunkt zeigt immer in z-Richtung
    float4 A = float4(0.0, 0.0, 1.0, 0.0);
    // Berechne den Halfway-Vektor
    float3 H = normalize(L + A.xyz);
    float specularLight = pow(max(dot(H, N), 0), S);
    if (diffuseLight <= 0) specularLight = 0;
    float4 specular = specularLight * s_light * s_mat;

    color.xyz = emissive + ambient + diffuse + specular;
    color.w = 1;
}
```

Bei diesem *Fragment-Shader* wird ein infiniter Augenpunkt angenommen, was der standardmäßigen OpenGL-Einstellung des *Light Models* entspricht: `glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_FALSE);` (Abschnitt 12.1.3.4). Falls die Umschaltung auf einen lokalen Augenpunkt berücksichtigt werden soll, müsste, analog wie bei der Umschaltung zwischen lokaler und infiniter Lichtquelle, eine Verzweigung ins Programm eingebaut werden, bei der im zweiten Ast der Vektor zum Augenpunkt für jedes Pixel individuell im *Fragment Shader* berechnet wird:

```
float4 pos_eye = float4(0.0, 0.0, 0.0, 1.0);
float3 A = normalize(pos_eye.xyz - position);
```

An dieser Stelle sind noch einige Effizienzbetrachtungen insbesondere zum Fragment-Programm fsPhong.cg angebracht. Man sollte sich bewusst machen, dass das Fragment-Programm bei einem Bild mit einer Auflösung von 1600 x 1200 Pixel 1,92 Millionen Mal ausgeführt werden muss. Daher sollte jede unnötige Operation eingespart werden.

Das Fragment-Programm fsPhong.cg setzt z.B. voraus, dass bei der Festlegung der Position der Lichtquelle (`pos_light`) der inverse Streckungsfaktor `w` entweder 0 oder 1 ist. Falls andere Werte zugelassen werden, müsste die Position der Lichtquelle noch durch den inversen Streckungsfaktor geteilt werden (die Eingabe eines anderen Wertes wird durch den *Cg*-Compiler nicht verhindert, aber das Ergebnis wäre falsch).

Die Effizienz des Fragment-Programms kann z.B. dadurch gesteigert werden, dass die Berechnung des ambienten Lichtanteils, die ja für jedes Pixel gleich ist, in das OpenGL-Programm verlagert wird. An das Fragment-Programm wird dann anstatt der `uniform`-Parameter `a_mat` und `a_light` einfach ein `uniform`-Parameter übergeben, der das Produkt der beiden enthält.

Eine andere Möglichkeit, die Effizienz des Fragment-Programms fsPhong.cg zu steigern, besteht darin, die Abfrage, ob es sich um eine infinite Lichtquelle handelt oder um

eine lokale (`if(pos_light.w == 0)`), ins OpenGL-Programm zu verlagern. Dazu müssen zwei verschiedene Versionen des Fragment-Programms bereitgestellt werden. Eines für die infinite Lichtquelle, bei der der Lichtvektor für alle Pixel der gleiche ist und somit nichts berechnet werden muss, sowie ein zweites Fragment-Programm für die lokale Lichtquelle, bei der für jedes Pixel ein eigener Lichtvektor aus der normierten Differenz zwischen der Lichtquellenposition und der Pixelposition berechnet werden muss. Im OpenGL-Programm muss dann, abhängig vom Wert des inversen Streckungsfaktors für die Lichtquellenposition, das jeweilige Fragment-Programm geladen werden.

Eine weitere Möglichkeit der Effizienzsteigerung besteht unter Umständen darin, die Beleuchtungsrechnung nicht in Welt-Koordinaten, sondern in Objekt-Koordinaten durchzuführen (dieser Ansatz wird im *Cg-Tutorial* [Fern03] verfolgt). Falls der Augenpunkt und die Lichtquelle lokal sind, bleibt das Fragment-Programm in beiden Koordinatensystemen das gleiche. Das Vertex-Programm vereinfacht sich jedoch, da die Vertex-Positionen und die Vertex-Normalenvektoren nicht vom Objekt–Koordinatensystem in das Welt–Koordinatensystem transformiert werden müssen (mit $\mathbf{M} \cdot \mathbf{v}$ und $\mathbf{M^{-1^T}} \cdot \mathbf{n}$). Im Gegenzug müssen im OpenGL-Programm aber die Positionen von Lichtquelle und Augenpunkt in das für jedes Objekt evtl. unterschiedliche Objekt-Koordinatensystem mit Hilfe der inversen ModelView-Matrix ($\mathbf{M^{-1}}$) transformiert werden (darüber stolpert man leicht beim Lesen des *Cg-Tutorials*). Da es in der Regel deutlich weniger Objekt–Koordinatensysteme als Vertices und Normalenvektoren gibt, kann in diesem Fall die Rendering-Geschwindigkeit leicht erhöht werden (der Engpass der *Rendering-Pipeline* liegt beim *Phong-Shading* praktisch immer am Fragment-Programm). In der folgenden Tabelle sind die notwendigen Rechenschritte gegenüber gestellt:

Größe	**Objektkoordinaten**	**Weltkoordinaten**
Vertex-Position	–	$\mathbf{M} \cdot \mathbf{v}$
Vertex-Normalenvektor	–	$\mathbf{M^{-1^T}} \cdot \mathbf{n}$
Lichtquellen-Position	$\mathbf{M^{-1}} \cdot \mathbf{l_{pos}}$	–
Augenpunkt-Position	$\mathbf{M^{-1}} \cdot \mathbf{a_{pos}}$	–

Falls der Augenpunkt und die Lichtquelle im Unendlichen sitzen, ist das Fragment-Programm im Welt-Koordinatensystem einfacher, da der Lichtvektor und der Vektor zum

Augenpunkt nicht für jedes Pixel berechnet werden müssen. Es reicht, wenn diese Vektoren nur einmal für die ganze Szene im OpenGL-Programm berechnet werden und als `uniform`-Parameter an das Fragment-Programm übergeben werden. Diese Einsparung im Fragment-Programm wirkt sich wegen des um Größenordnungen häufigeren Aufrufs natürlich sehr viel stärker aus als Einsparungen beim Vertex-Programm. Deshalb ist in diesem Fall das Welt-Koordinatensystem das effizientere.

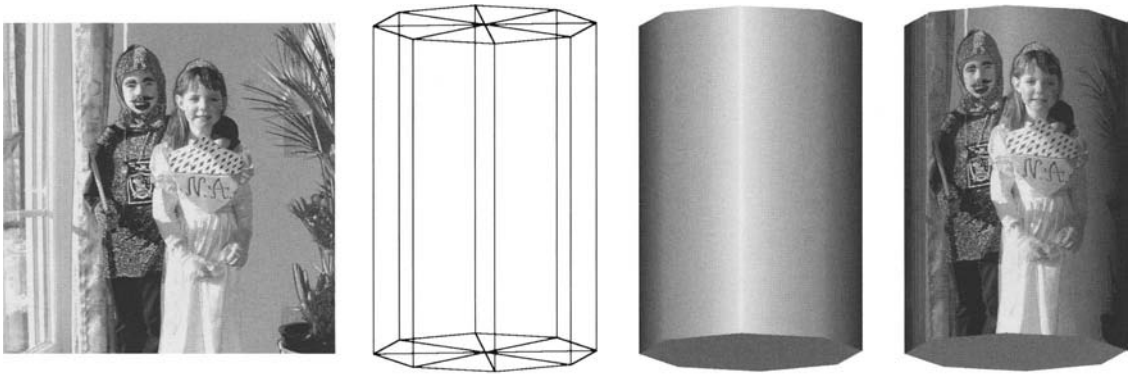
Das vollständige Beleuchtungsmodell von OpenGL (12.17) mit globaler ambienter Lichtquelle und bis zu acht Spotlichtquellen mit Abschwächungsfaktoren kann mit den hier vorgestellten Mitteln implementiert werden. Dies würde allerdings weitere Seiten an *Cg*-Codelistings bedeuten und bleibt daher aus Platzgründen dem Leser überlassen.

Kapitel 13

Texturen

Bisher wurden alle Farben, die man einem Objekt zuordnen kann, entweder direkt oder über ein Beleuchtungsmodell den Vertices zugewiesen. Die Farben der am Bildschirm sichtbaren Pixel eines Objekts wurden durch unterschiedliche *Shading*-Verfahren (*Flat, Gouraud, Phong*) aus den Vertexfarben und Vertex-Normalenvektoren interpoliert. Durch die Zuweisung von wenigen Vertexfarben sind also indirekt auch schon die Farben aller Pixel eines Objekts festgelegt. Objekte die mit diesen Verfahren gerendert werden, besitzen monotone Farbverläufe und erscheinen deshalb plastikartig und künstlich. Reale Oberflächen, wie z.B. Hauswände, Rasenflächen, Plakatwände, Fell oder Stoff besitzen fast immer eine gewisse regelmäßige oder unregelmäßige Struktur, die man als „*Textur*" bezeichnet. Während man in der Bildverarbeitung aus Texturen in erster Linie Merkmale für die Segmentierung gewinnt (Kapitel 27), werden Texturen in der Computergrafik eingesetzt, um die Oberfläche von Objekten mit einer Struktur zu überziehen. Ein mittlerweile klassisches Beispiel für eine Textur ist ein ganz normales Foto, das auf ein Polygonnetz gemappt wird (Bild 13.1). Mit dieser Technik gelang im Laufe der 1980iger Jahre der Durchbruch zu einer neuen Qualitätsstufe, der sogenannten „Fotorealistischen Computergrafik".

Möchte man die Feinstruktur, die mit Foto-Texturen möglich ist, ohne Texture-Mapping erzeugen, ist eine gigantische Erhöhung der Tesselierung notwendig. Man müsste die Zahl der Polygone soweit erhöhen, bis pro Pixel mindestens ein Vertex vorhanden ist, dem man dann den entsprechenden Farbwert des Fotos zuweisen kann. Da eine Szene mit unterschiedlichen Auflösungen gerendert werden kann, müsste die Tesselierung jedesmal darauf angepasst werden. Die Effizienz eines solchen Vorgehens wäre natürlich extrem niedrig, denn es müsste eine riesige Anzahl an Vertices durch die gesamte *Rendering-Pipeline* geschickt werden (Bilder 5.1 und 12.21). Dagegen genügt beim Texture-Mapping eine relativ geringe Anzahl an Vertices, um realistische Bilder zu generieren. Die Textur-Farbwerte werden erst nach der Rasterisierungsstufe der *Rendering-Pipeline* mit den aus der Beleuchtung und Schattierung hervorgegangenen Fragment-Farbwerten kombiniert. Dieser Vorgang des Texture-Mappings wird heutzutage in nahezu allen gängigen Grafiksystemen hardwarebeschleunigt und kostet daher verhältnismäßig wenig Rechenzeit. Diese am häufigsten genutzte Technik des Foto-Texture-Mappings (*Image Texturing*) wird im folgenden Abschnitt ausführlich dargestellt.



(a) (b) (c) (d)

Bild 13.1: Fotorealistische Computergrafik: (a) Ein Foto. (b) Ein Drahtgittermodell eines Zylinders. (c) Der Zylinder mit Beleuchtung und *Gouraud*-Schattierung. (d) Der Zylinder mit gemappter Foto-Textur.

Mittlerweile wurden jedoch viele weitere Anwendungen für Texturen entwickelt, um die Struktur von Oberflächen realistischer darstellen zu können. Manche Oberflächen sind an bestimmten Stellen glatt und an anderen rau. Diese Oberflächeneigenschaft kann durch eine Glanz-Textur (*Gloss Map*) simuliert werden, die den Beitrag der spekularen Lichtkomponente in der Beleuchtungsformel pro Pixel festlegt. Voraussetzung für diesen Effekt ist allerdings, wie bei vielen fortgeschrittenen Texture-Mapping-Techniken, das *Phong-Shading*, bei dem das Beleuchtungsmodell pro Pixel berechnet wird. Viele Oberflächen sind nicht nur rau, sondern besitzen kleine Erhöhungen oder Vertiefungen, wie bei einem Relief. Nachdem die menschliche Wahrnehmung aus der Schattierung auf die räumliche Form schließt, kann der Eindruck eines Reliefs durch eine lokale (d.h. pixel-bezogene) Veränderung der Normalenvektoren erreicht werden. Eine Relief-Textur (*Bump Map*) enthält folglich Werte, die angeben, wie die Normalenvektoren modifiziert werden. Andere Texture-Mapping-Verfahren dienen der Approximation globaler Beleuchtungsverfahren, wie z.B. Umgebungs-Texturen (*Environment Maps*) und Schatten-Texturen (*Shadow Maps*), die die Effekte von *Ray-Tracing*-Verfahren simulieren, oder Beleuchtungs-Texturen (*Light Maps, Light Fields*), die die Effekte von *Radiosity*-Vefahren imitieren. Diese in den späteren Abschnitten vorgestellten Verfahren gewinnen zunehmend an Bedeutung, nachdem mit der vierten Generation von *GPUs* (*Graphics Processing Units*) wie nVIDIA's GeForce FX Familie oder ATI's Radeon 9700, programmierbare Vertex- und Pixel Shader zur Verfügung stehen, die eine hardware-beschleunigte und damit echtzeitfähige Implementierung dieser fortschrittlichen Texture-Mapping-Techniken erlauben.

13.1 Foto-Texturen (Image Texturing)

Das klassische und auch heute noch am häufigsten eingesetzte Mapping-Verfahren verwendet gewöhnliche Fotos als Texturen ([Catm74], [Heck86]). Diese 2-dimensionalen und in der Regel rechteckigen Fotos können dabei z.B. Grauwert-Bilder, Farb-Bilder oder auch Farb-Bilder mit einem Alpha-Kanal (der Transparenz-Komponente, Kapitel 9) sein. Ein Stapel von Fotos, die z.B. Schnittbilder eines komplexen Objekts oder eine zeitliche Abfolge von Bildern sind, kann als 3-dimensionale Textur aufgefasst werden. In bestimmten Fällen, wie z.B. bei der Darstellung eines Regenbogens, reicht auch schon eine Zeile eines Bildes, d.h. eine 1-dimensionale Textur aus, um einen realistischen Effekt zu erzeugen. Die Texturen können künstlich erzeugt werden, z.B. durch 3D-Computergrafik oder mit einem Algorithmus (prozedurale Texturen, Fraktale [Peit86]), oder mit Hilfe einer Kamera aus der natürlichen Umwelt inclusive der komplexesten physikalischen Beleuchtungsphänomene aufgenommen werden.

Wie wird nun eine Textur auf eine Oberfläche aufgebracht? Man kann sich diesen Vorgang vereinfacht etwa so vorstellen: man nehme ein Foto, das nicht auf Papier, sondern auf einer beliebig dehnbaren Gummihaut entwickelt wurde; dieses dehnbare Foto wird über die gewünschte Oberfläche gestülpt und mit Hilfe von Stecknadeln an den Eckpunkten der Polygone fixiert. Anders ausgedrückt heißt das, den Vertices der Oberfläche werden bestimmte Punkte der Textur (spezifiziert in Texturkoordinaten) zugeordnet. Zwischen den Vertices werden die Punkte der Textur gleichmäßig verteilt, d.h. die Texturkoordinaten werden linear interpoliert. Dafür bietet sich natürlich wieder der in der Rasterisierungsstufe ablaufende Scan-Line-Algorithmus (12.20) an, der schon aus Kapitel 12 vom *Gouraud-Shading*, also der linearen Interpolation von Vertexfarben, bekannt ist.

Mathematisch gesehen laufen beim Texture Mapping zwei Transformationen ab: im ersten Schritt werden Punkte aus dem 2-dimensionalen Texturkoordinatensystem auf Vertices im 3-dimensionalen Objektkoordinatensystem abgebildet; im zweiten Schritt werden die Objekte durch die Projektions- und Viewporttransformation wieder auf 2-dimensionale Bildschirmkoordinaten abgebildet. Die lineare Interpolation der Texturkoordinaten läuft also im Bildschirmkoordinatensystem ab, in dem perspektivische Verzerrungen durch die Projektionstransformation auftreten. Solange die Tesselierung der Oberfläche fein genug ist, stören diese Verzerrungen kaum. Andernfalls kann durch den OpenGL-Befehl `glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST)` die lineare Interpolation der Texturkoordinaten so modifiziert werden (d.h. die Texturkoordinaten werden durch den inversen Streckungsfaktor w geteilt), dass keine perspektivischen Verzerrungen auftreten.

Nach der Rasterisierung liegen also für jedes Pixel entsprechend interpolierte Texturkoordinaten vor. Nun muss der Farbwert aus der Textur geholt werden. Die einfachste Variante besteht darin, den Bildpunkt aus der Textur auszulesen, dessen Koordinaten den interpolierten Werten am nächsten liegen. Allerdings führt dieses Verfahren zu starken Aliasing-Effekten, wie in Kapitel 10 beschrieben. Denn eine Textur passt von der Größe her praktisch nie 1 : 1 auf ein durch die perspektivische Abbildung verzerrtes Polygon, so dass sehr häufig viele Bildpunkte der Textur (die sogenannten „*Texel*“ bzw. *texture elements*) auf ein Pixel (*picture element*) am Bildschirm treffen, oder umgekehrt. Zur Eindämmung

dieser Störungen benötigt man entsprechende Textur-Filter, die im Folgenden beschrieben werden.

Letztlich stellt sich noch die Frage, wie die eventuell gefilterten Farbwerte aus der Foto-Textur mit dem Farbwert aus der Beleuchtungs- und Schattierungsrechnung kombiniert werden. Dies wird im sogenannten *Texture Environment* festgelegt.

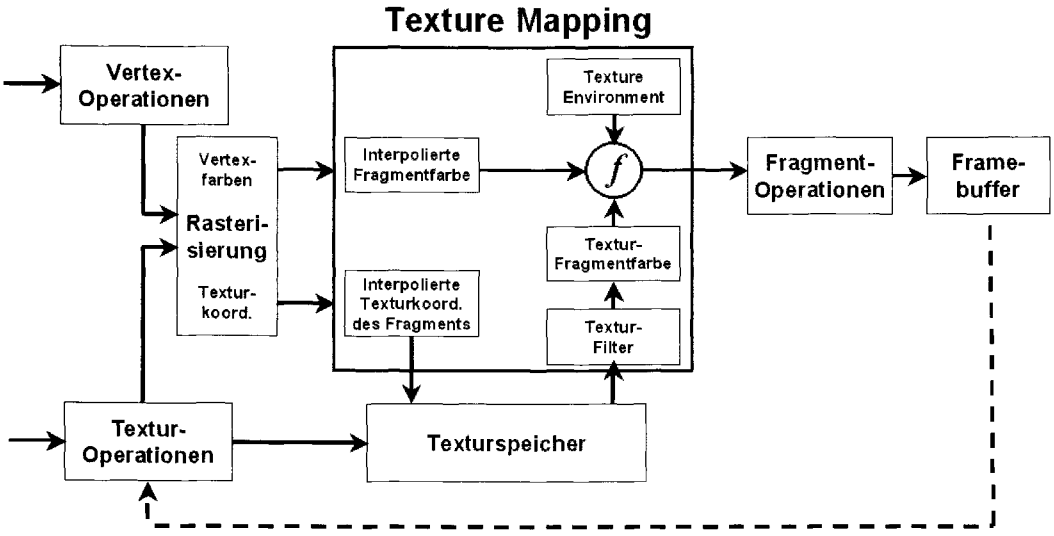


Bild 13.2: Die Rendering Pipeline für Texturen. *Eingaben*: Texturkoordinaten pro Vertex, Textur-Filter, Texture Environment und laden der Texturdaten in den Texturspeicher. *Verarbeitung*: Interpolation der Texturkoordinaten, Abtasten der Textur und interpolieren der Texturfarben gemäß Textur-Filter, Mischung von Textur- und Beleuchtungsfarbe gemäß Texture Environment. *Ausgabe*: Mischfarbe des Fragments (Pixel).

Zusammengefasst sind also beim Texture Mapping folgende Schritte durchzuführen (Bild 13.2):

- Spezifikation der Textur
- Festlegung, wie die Textur auf jedes Pixel aufgetragen wird, d.h.:
  - Spezifikation der Textur-Filter
  - Spezifikation der Mischung von Textur- und Beleuchtungsfarbe
- Zuordnung von Texturkoordinaten an Vertices
- Einschalten des Texture Mappings

13.1.1 Spezifikation der Textur

Bevor eine Textur auf ein Polygonnetz gemappt werden kann, muss sie erst einmal definiert und geladen werden. Im Folgenden werden 2-dimensionale Texturen behandelt. Die gesamte Diskussion und die zugehörigen OpenGL-Befehle gelten jedoch analog auch für 1-dimensionale und 3-dimensionale Texturen. Eine 2-dimensionale Textur wird formal als eine Bildmatrix **T** beschrieben, die von den zwei Variablen (s, t) eines orthogonalen Texturkoordinatensystems abhängt. Es ist sinnvoll, unabhängig von der Auflösung der Textur, ein normiertes Texturkoordinatensystem zu verwenden, d.h. die Variablen (s, t) werden auf den Parameterbereich $[0, 1]$ beschränkt (Bild 13.3).

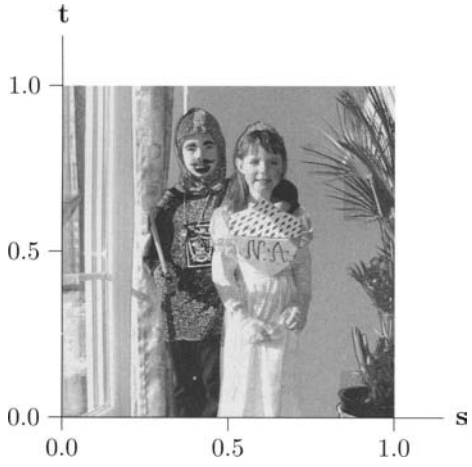


Bild 13.3: Eine 2-dimensionale Foto-Textur im Texturkoordinatensystem mit den orthogonalen Achsen s und t.

Die einzelnen Texel einer Textur können einfache Grauwerte sein, wie in Bild 13.3, Grauwerte mit Alpha-Kanal (der Transparenz-Komponente, Kapitel 9), 3-komponentige Farbwerte (R,G,B), oder Farbwerte mit Alpha-Kanal. Die Quantisierung jedes einzelnen Kanals kann ebenfalls variieren, je nach Kanal sind 1 bit, 2 bit, 3 bit, 4 bit, 8 bit, 12 bit oder 16 bit pro Kanal möglich. OpenGL-intern werden die Werte aller Kanäle durch Gleitkommazahlen aus dem Intervall $[0, 1]$ dargestellt.

Zur Spezifikation einer 2-dimensionalen Textur stellt OpenGL vier unterschiedliche Befehle zur Verfügung:

- `glTexImage2D()`
  Spezifikation einer Textur mit Daten aus dem Hauptspeicher

- `glTexSubImage2D()`
  Ersetzen eines Texturausschnitts mit Daten aus dem Hauptspeicher

- `glCopyTexImage2D()`
  Spezifikation einer Textur mit Daten vom Bildspeicher (*Framebuffer*)

- `glCopyTexSubImage2D()`
  Ersetzen eines Texturausschnitts mit Daten aus dem Bildspeicher

Zur Spezifikation von 1-dimensionalen oder 3-dimensionalen Texturen werden die obigen OpenGL-Befehle leicht modifiziert: anstatt `*2D()` wird `*1D()` bzw. `*3D()` benutzt. Die Parameter und die zulässigen Konstanten unterscheiden sich teilweise von der 2-dimensionalen Variante der Befehle. Aus Platzgründen wird hier auf die OpenGL-Spezifikation [Sega04] bzw. den OpenGL Programming Guide [Shre05] verwiesen.

Am häufigsten wird eine Textur mit Daten aus dem Hauptspeicher spezifiziert. Dazu dient der folgende OpenGL-Befehl:

```
glTexImage2D ( GLenum target, GLint level, GLint internalFormat,
               GLsizei width, GLsizei height, GLint border,
               GLenum format, GLenum type, const GLvoid *image )
```

Zur großen Zahl an Parameter dieses Befehls kommt eine noch größere Anzahl möglicher Werte dieser Parameter hinzu, die in den folgenden Tabellen zusammen mit ihrer Bedeutung aufgelistet sind.

Parameter	Werte	Bedeutung
target	GL_TEXTURE_2D	Normale 2-dimensionale Textur.
	GL_PROXY_TEXTURE_2D	Abfrage, ob der Texturspeicher für die spezifizierte Texturgröße ausreicht.
	GL_TEXTURE_CUBE_MAP_* POSITIVE_X / _Y / _Z NEGATIVE_X / _Y / _Z	Eine von sechs möglichen Cube-Map-Texturen. Diese Werte sind für 1- bzw. 3-dimensionale Texturen nicht möglich. Cube-Map-Texturen werden im Abschnitt 13.4 erläutert.
level	$0, 1, 2, ...$	Auflösungsstufe der Textur (MipMap-Level, siehe Abschnitt 13.1.3). 0 = höchste Auflösung, 1 = zweithöchste Auflösung ...
internalFormat	siehe nächste Tabelle	Bestimmt die Anzahl und evtl. die Quantisierung der Farbkomponenten.
width	$2^n + 2 \cdot$ *border* $n = 0, 1, 2, ...$	Breite der Textur in Texel. Im Rahmen von OpenGL-Extensions kann die Breite auch von einer Zweier-Potenz abweichen.
height	$2^m + 2 \cdot$ *border* $m = 0, 1, 2, ...$	Höhe der Textur in Texel. Im Rahmen von OpenGL-Extensions kann die Höhe auch von einer Zweier-Potenz abweichen.
border	0 oder 1	Breite des Rands. 0 = kein Rand, 1 = Rand der Breite 1 Texel
format	GL_RGB, GL_RGBA GL_BGR, GL_BGRA GL_RED, GL_GREEN GL_BLUE, GL_ALPHA GL_LUMINANCE GL_LUMINANCE_ALPHA	Format der Texturdaten, weist den Daten eine Bedeutung zu.
type		Datentyp der Texturdaten:
	GL_BYTE	8 bit Integer mit Vorzeichen
	GL_UNSIGNED_BYTE	8 bit Integer ohne Vorzeichen
	GL_BITMAP	U_BYTE mit signifikantem unterstem Bit
	GL_SHORT	16 bit Integer mit Vorzeichen
	GL_UNSIGNED_SHORT	16 bit Integer ohne Vorzeichen
	GL_INT	32 bit Integer mit Vorzeichen
	GL_UNSIGNED_INT	32 bit Integer ohne Vorzeichen
	GL_FLOAT	32 bit Fließkomma
**image*	–	Zeiger auf die Texturdaten im Hauptspeicher

Für den Parameter *internalFormat* sind die folgenden 38 Werte zugelassen, die die Zahl und die empfohlene Quantisierung der Komponenten pro Texel vorgeben:

internalFormat Wert	Komponenten	R bits	G bits	B bits	A bits	L bits	I bits
GL_LUMINANCE	1						
GL_LUMINANCE4	1					4	
GL_LUMINANCE8	1					8	
GL_LUMINANCE12	1					12	
GL_LUMINANCE16	1					16	
GL_LUMINANCE_ALPHA	2						
GL_LUMINANCE4_ALPHA4	2				4	4	
GL_LUMINANCE6_ALPHA2	2				2	6	
GL_LUMINANCE8_ALPHA8	2				8	8	
GL_LUMINANCE12_ALPHA4	2				4	12	
GL_LUMINANCE12_ALPHA12	2				12	12	
GL_LUMINANCE16_ALPHA16	2				16	16	
GL_ALPHA	1						
GL_ALPHA4	1				4		
GL_ALPHA8	1				8		
GL_ALPHA12	1				12		
GL_ALPHA16	1				16		
GL_INTENSITY	1						
GL_INTENSITY4	1						4
GL_INTENSITY8	1						8
GL_INTENSITY12	1						12
GL_INTENSITY16	1						16
GL_RGB	3						
GL_R3_G3_B2	3	3	3	2			
GL_RGB4	3	4	4	4			
GL_RGB5	3	5	5	5			
GL_RGB8	3	8	8	8			
GL_RGB10	3	10	10	10			
GL_RGB12	3	12	12	12			
GL_RGB16	3	16	16	16			
GL_RGBA	4						
GL_RGBA2	4	2	2	2	2		
GL_RGBA4	4	4	4	4	4		
GL_RGB5_A1	4	5	5	5	1		
GL_RGBA8	4	8	8	8	8		
GL_RGB10_A2	4	10	10	10	2		
GL_RGBA12	4	12	12	12	12		
GL_RGBA16	4	16	16	16	16		

Der Parameter *internalFormat* kann neben den oben aufgelisteten 38 symbolischen Konstanten zusätzlich noch die Zahlen $1, 2, 3, 4$ annehmen, die einfach nur die Anzahl der Komponenten, aber nicht die Quantisierung pro Texel vorgeben, und außerdem noch sechs symbolische Konstanten, die im Rahmen von OpenGL-Extensions eine Datenkompression von Texturen ermöglichen [Leng03].

Falls Texturen mit einer einzigen Auflösungsstufe benutzt werden, sollte der Parameter *level* $= 0$ gesetzt werden.

Der Parameter *type* kann neben den in der vorletzten Tabelle aufgelisteten Werten auch noch gepackte Texel-Datentypen enthalten. Denn häufig kann die Hardware sehr viel effizienter auf die Texturdaten zugreifen, wenn die Daten an 2-, 4- oder 8-Byte-Grenzen im Hauptspeicher abschließen. Die gepackten Datentypen für den Parameter *type* lauten:

type Wert	OpenGL-Datentyp	Komponenten	Texel-Format
GL_UNSIGNED_BYTE_3_3_2	GLubyte 8 bit	3	RGB
GL_UNSIGNED_BYTE_2_3_3_REV	GLubyte 8 bit	3	RGB
GL_UNSIGNED_SHORT_5_6_5	GLushort 16 bit	3	RGB
GL_UNSIGNED_SHORT_5_6_5_REV	GLushort 16 bit	3	RGB
GL_UNSIGNED_SHORT_4_4_4_4	GLushort 16 bit	4	RGBA, BGRA
GL_UNSIGNED_SHORT_4_4_4_4_REV	GLushort 16 bit	4	RGBA, BGRA
GL_UNSIGNED_SHORT_5_5_5_1	GLushort 16 bit	4	RGBA, BGRA
GL_UNSIGNED_SHORT_1_5_5_5_REV	GLushort 16 bit	4	RGBA, BGRA
GL_UNSIGNED_INT_8_8_8_8	GLuint 32 bit	4	RGBA, BGRA
GL_UNSIGNED_INT_8_8_8_8_REV	GLuint 32 bit	4	RGBA, BGRA
GL_UNSIGNED_INT_10_10_10_2	GLuint 32 bit	4	RGBA, BGRA
GL_UNSIGNED_INT_2_10_10_10_REV	GLuint 32 bit	4	RGBA, BGRA

Diese Datentypen werden nach folgendem Schema interpretiert: die Zahlen am Ende des jeweiligen *type* geben die Anzahl an bits für die jeweilige Komponente an. Der Wert GL_UNSIGNED_BYTE_3_3_2 bedeutet z.B., dass von bit 7 bis 5 die erste Farbkomponente gespeichert wird, von bit 4 bis 2 die zweite Farbkomponente und von bit 1 bis 0 die dritte Farbkomponente. Der umgekehrte Typ GL_UNSIGNED_BYTE_2_3_3_REV bedeutet, dass von bit 7 bis 6 die dritte Farbkomponente gespeichert wird, von bit 5 bis 3 die zweite Farbkomponente und von bit 2 bis 0 die erste Farbkomponente. Die anderen Werte werden entsprechend interpretiert. Neben der verkürzten Lese- und Schreibgeschwindigkeit ist auch die erhebliche Speicherplatzersparnis ein wichtiger Grund für den Einsatz von gepackten Datentypen. Eine normale RGBA-Textur mit den üblichen 8 bit pro Farbkomponente benötigt z.B. 32 bit pro Texel. In den meisten Anwendungen, insbesondere wenn Bewegung mit im Spiel ist, genügt aber schon der Datentyp GL_UNSIGNED_SHORT_5_5_5_1, der nur 16 bit pro Texel benötigt. Damit kann die Hälfte des meist knappen Texturspeicherplatzes gespart werden. Um sicherzustellen, dass die Texturdaten im Hauptspeicher auch direkt nacheinander abgespeichert werden, sollte der folgende OpenGL-Befehl aufgerufen werden:

```
glPixelStorei(GL_PACK_ALIGNMENT, 1);
```

Nach der Vielzahl von Tabellen und Möglichkeiten zur Spezifikation von Texturen, hier nun ein Beispiel in OpenGL für ein Codefragment zur Definition einer Textur der Größe 512 x 256 Texel mit 4 Farbkomponenten (R,G,B,A); die Daten sollen im Hauptspeicher ungepackt und direkt hintereinander stehen:

```
GLint width = 512, height = 256;
static GLubyte image[width][height][4];

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
glTexImage2D ( GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
               GL_RGBA, GL_UNSIGNED_BYTE, image);
```

Da es unzählige Datenformate für Bilder gibt und OpenGL von der Design-Philosophie unabhängig von der Plattform oder irgendwelchen Datenformaten sein soll, gibt es keine Lader (Leseroutinen) für Texturen. Entweder man erzeugt sich eine Textur über einen Algorithmus selbst, oder man programmiert bzw. sucht sich einen Lader für das vorliegende Grafikformat. Auf der Webseite zu diesem Buch werden Lader für Bitmap- (*.bmp) und SGI-Texturen (*.sgi) bereitgestellt.

In den meisten Anwendungsfällen muss die Höhe und die Breite einer Textur eine Zweierpotenz sein, wie z.B. 128 x 1024 Texel. Die heutzutage häufig von digitalen Kameras stammenden Bilder erfüllen diese Bedingungen in der Regel nicht. Die Bilder müssen also entweder beschnitten oder skaliert werden, damit sie dieser Einschränkung von OpenGL genügen. Dazu kann man Standard-Bildbearbeitungswerkzeuge oder auch den folgenden Befehl aus der OpenGL-Utility-Library verwenden:

```
gluScaleImage ( GLenum format, GLsizei width_in, GLsizei height_in,
                GLenum type_in, const GLvoid *image_in, GLsizei width_out,
                GLsizei height_out, GLenum type_out, GLvoid *image_out )
```

Durch diesen Befehl wird ein Eingabebild **image_in* im Verhältnis *width_in* / *width_out* bzw. *height_in* / *height_out* mit Hilfe von linearer Interpolation oder einem bewegten Mittelwert (Abschnitt 18.3) skaliert und als Ausgabebild **image_out* in einen vorher reservierten Bereich des Hauptspeichers zurückgeschrieben. Die Bedeutung der Parameter und die möglichen Werte sind die gleichen wie beim `glTexImage2D()`-Befehl.

Selbst heutzutage ist der Texturspeicher eine sehr begrenzte Resource. Möchte man beispielsweise eine Textur mit dem internen Format `GL_RGBA16`, d.h. 4 x 16 = 64 bit und der Größe 4096 x 4096 Texel benutzen, benötigt man immerhin schon 128 MByte Texturspeicher. Um heraus zu finden, ob der zur Verfügung stehende Texturspeicher ausreicht, kann man den `glTexImage2D()`-Befehl mit einem speziellen Platzhalter, einem sogenannten „*Texture Proxy*“ (*target* = `GL_PROXY_TEXTURE_2D`), ausführen. Der Zeiger auf die

Texturdaten muss in diesem Fall NULL sein. Durch Abfrage des Texturzustands mit dem Befehl `glGetTexLevelParameteriv()` kann man feststellen, ob der Texturspeicher ausreicht oder nicht. Die Textur-Zustandsvariablen für Höhe und Breite ergeben den Wert 0, falls der Texturspeicher zu klein ist. Das folgende Codefragment zeigt die Anwendung des *Texture Proxy*:

```
GLint width = 512, level = 0;

glTexImage2D ( GL_PROXY_TEXTURE_2D, 0, GL_RGBA16, 4096, 4096, 0,
               GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glGetTexLevelParameteriv ( GL_PROXY_TEXTURE_2D, level,
                           GL_TEXTURE_WIDTH, &width);
if(width == 0) fprintf(stderr, ''Not enough texture memory \n'');
```

Es ist zu beachten, dass Texturen ein Bestandteil des OpenGL-Zustands sind, d.h. eine Texturdefinition gilt solange, bis sie durch eine andere abgelöst wird. Zu einem bestimmten Zeitpunkt kann also immer nur eine einzige Textur definiert sein. Dies führt dazu, dass für ein etwas komplexeres Objekt, auf das verschiedene Texturen gemappt werden, zwischendurch der OpenGL-Zustand geändert werden muss. Jeder Zustandswechsel ist mit zum Teil erheblichen Geschwindigkeitsnachteilen verbunden. Deshalb sollte man beim Design von Objekten darauf achten, dass man möglichst nur eine einzige große Textur verwendet, die die kleineren Sub-Texturen enthält. Die Auswahl der gewünschten Sub-Textur kann durch eine entsprechende Wahl der Texturkoordinaten vorgenommen werden. Außerdem sollte man alle Objekte, die mit der gleichen Textur gerendert werden können, hintereinander (d.h. sortiert nach der verwendeten Textur) in die Grafik-Hardware schicken, damit die Zahl der OpenGL-Zustandswechsel minimiert wird. Weitere Möglichkeiten, den Aufwand von Zustandswechseln beim Texturaustausch zu verringern, werden im nächsten Abschnitt, sowie im Abschnitt 13.1.8 „Textur-Objekte“ beschrieben.

13.1.1.1 Ersetzen eines Texturausschnitts

In der Regel ist es deutlich rechenzeitintensiver, eine Textur neu zu erzeugen, als eine bereits bestehende zu modifizieren. Deshalb bietet OpenGL mit dem `glTexSubImage2D()`-Befehl die Möglichkeit, Texturausschnitte oder auch die ganze Textur mit Daten aus dem Hauptspeicher zu ersetzen. Im Folgenden sind einige Anwendungen erläutert, die sich dieser Möglichkeiten bedienen:

- Echtzeit-Simulationen mit geospezifischen Texturen: in diesen Fällen hat man es meist mit relativ großen Datenbasen zu tun, wobei als Texturen Originalfotos vom jeweiligen Ort verwendet werden müssen. Die Menge an Texturdaten ist deshalb immens und passt in keinen Texturspeicher. Deshalb müssen die Texturen während der Echtzeit-Simulation laufend vom Hauptspeicher in den Texturspeicher nachgeladen werden, und zwar ohne dass dadurch die Bildgenerierrate kurzzeitig abfällt (ein sogenannter *frame drop*). Um dies zu vermeiden ist einerseits eine ausreichend große

Busbandbreite zur Grafikhardware notwendig und andererseits das Zerstückeln einer großen Textur in beliebig kleine Teile. Diese kleinen Texturstücke können dann mit dem `glTexSubImage2D()`-Befehl Stück für Stück über mehrere Bilder hinweg in den Texturspeicher befördert und dort wieder zusammengesetzt werden, ohne dass die Bildgenerierrate Schaden nimmt.

- Echtzeit-Video: in diesem Fall wird nur eine einzige Textur erzeugt und jedes nachfolgende Bild der Sequenz ersetzt mit Hilfe des `glTexSubImage2D()`-Befehls das vorherige [Shre05].

- Benutzung von Texturausmaßen, die keiner Zweierpotenz entsprechen: für den `glTexSubImage2D()`-Befehl gilt diese Einschränkung nicht. Daher ist es möglich, sogenannte „*non-power-of-two*"-Texturen zu benutzen, indem man mit dem normalen `glTexImage2D()`-Befehl zuerst eine Dummy-Textur lädt, deren Ausmaße in Höhe und Breite bis zur jeweils nächstliegenden Zweierpotenz vergrößert werden. Anschließend kann dann eine Textur mit beliebiger Ausdehnung gemappt werden, solange sie kleiner als die Dummy-Textur ist.

Zum Ersetzen eines Texturausschnitts mit Daten aus dem Hauptspeicher dient der folgende OpenGL-Befehl:

```
glTexSubImage2D (   GLenum target, GLint level, GLint xoffset,
                    GLint yoffset, GLsizei width, GLsizei height,
                    GLenum format, GLenum type, const GLvoid *image )
```

Die Parameter *xoffset* und *yoffset* definieren eine Verschiebung des zu ersetzenden Texturausschnitts vom Texturkoordinatenursprung links unten aus gesehen um *xoffset* Texel nach rechts und um *yoffset* nach oben. Alle anderen Parameter sind die gleichen wie bei dem `glTexImage2D()`-Befehl.

13.1.1.2 Der Bildspeicher als Quelle für Texturen (Multipass Rendering)

Durch die Nutzung des Bildspeichers als Quelle für Texturen entsteht ein mächtiges Werkzeug für die 3D-Computergrafik: die Möglichkeit der rekursiven Bildgenerierung (*multipass rendering*). In einem ersten Schritt kann eine Szene z.B. aus einem bestimmten Blickwinkel oder mit speziellen Eigenschaften gerendert werden. Im zweiten Schritt kann das Bild aus dem ersten Durchlauf als Textur wiederverwendet und erneut auf die Polygone der Szene gemappt werden. Auf diesem Prinzip beruht eine ganze Reihe von Methoden, die versuchen, die Effekte der indirekten Lichtanteile von globalen Beleuchtungsverfahren zu approximieren. Denn sowohl das Ray Tracing als auch das Radiosity-Rendering-Verfahren beruhen auf rekursiven Lösungsansätzen (Abschnitt 12.1.2). Typische Anwendungen, die mit einem Rekursionsschritt auskommen, sind:

- Spiegelungen an planaren Flächen: hier wird im ersten Rendering-Schritt die Szene aus der Position des virtuellen Augenpunkts hinter dem Spiegel gerendert. Dieses

Spiegelbild wird im zweiten Rendering-Schritt als Textur auf das „spiegelnde" Polygon gemappt, so dass wie beim rekursiven Ray Tracing eine echte und interaktive Spiegelung entsteht. Für Mehrfach-Spiegelungen benötigt man natürlich entsprechend mehr Rekursionsschritte.

- Spiegelungen an gekrümmten Flächen: hier wird im ersten Rendering-Schritt die Szene aus dem Zentrum des spiegelnden Objekts mit einer extremen Weitwinkelperspektive (Fischauge) gerendert und anschließend als Umgebungs-Textur (Abschnitt 13.4) auf die gekrümmte Oberfläche gemappt. Falls man das Objekt aus allen Richtungen mit nahezu perfekten Spiegelungen rendern will ist eine sogenannte *Cube Map* als Umgebungs-Textur nötig. In diesem Fall benötigt man schon sechs Rendering-Schritte zur Erzeugung der Texturen für die sechs Flächen eines Kubus, bevor man die eigentliche Spiegelung berechnen kann.

- Geometrische Entzerrungen: bei anspruchsvollen Flug- oder Fahrsimulatoren verwendet man als Projektionsfläche für die Außensicht kugelförmige Oberflächen, um einen möglichst großen horizontalen und vertikalen Sichtwinkel zu erzeugen (sogenannte Dom-Projektionen). Die Projektionstransformation in OpenGL (Kapitel 7) geht aber immer von einer ebenen Projektionsfläche aus, so dass auf einer gekrümmten Kugeloberfläche nach außen hin zunehmende Verzerrungen entstehen. Dies kann durch eine Vorentzerrung des Bildes im Rahmen eines *dualpass rendering* kompensiert werden. Im ersten Durchlauf wird die Szene ganz normal gerendert, d.h. auf eine ebene Fläche projiziert. Im zweiten Durchlauf wird dieses Bild als Textur auf ein invers gekrümmtes Polygonnetz gemappt. Bei der realen Projektion dieses Bildes mit einem Beamer auf die kugelförmige Projektionsfläche kompensieren sich die beiden Oberflächenkrümmungen, und es entsteht ein verzerrungsfreies Bild. Mit diesem Verfahren können übrigens beliebige geometrische Entzerrungen hardware-beschleunigt und somit extrem schnell vorgenommen werden, indem auf entsprechend geformte Polygonnetze gemappt wird (Kapitel 22).

- Schattenwurf: in diesem Fall rendert man die Szene im ersten Durchlauf aus der Position der Lichtquelle und speichert den z-Buffer-Inhalt als sogenannte *shadow map*. Die *shadow map* enthält die Entfernungen von der Lichtquelle zu allen beleuchteten Oberflächenpunkten. Im zweiten Durchlauf rendert man die Szene aus der Position des Augenpunkts und benutzt die *shadow map*, um festzustellen, ob das gerade bearbeitete Pixel im Schatten der Lichtquelle ist (Entfernung des Punktes zur Lichtquelle ist größer als der z-Wert, der in der *shadow map* gespeichert ist) oder nicht. Falls das Pixel im Schatten ist, wird nur der emissive und der ambiente Lichtanteil in die Beleuchtung einbezogen, der diffuse und der spekulare Lichtanteil entfällt (Abschnitt 13.6).

So schön die Effekte auch sind, die mit dem *multipass rendering* erzielbar sind, ist doch ein entscheidender Nachteil unvermeidlich: die Bildgenerierrate fällt sehr stark ab. Bei zwei Durchläufen durch die *Rendering Pipeline* halbiert sich die Bildgenerierrate, bei

drei Durchläufen fällt sie auf ein Drittel des ursprünglich erreichbaren Wertes ab, so dass die Interaktivität einer solchen Anwendung sehr schnell inakzeptabel wird. Bei einigen Grafikkarten und bestimmten Anwendungen fällt der Zeitverlust deutlich geringer aus, da nicht die gesamte *Rendering Pipeline* neu durchlaufen werden muss, sondern nur der Teil nach der Rasterisierung. Dennoch sind in erster Linie solche Anwendungen interessant, die bereits mit zwei Durchläufen einen attraktiven Effekt erzielen.

Zur Spezifikation einer Textur mit Daten vom Bildspeicher (*Frame Buffer*) dient der folgende OpenGL-Befehl:

```
glCopyTexImage2D ( GLenum target, GLint level, GLint internalFormat,
                   GLint x, GLint y, GLsizei width, GLsizei height,
                   GLint border )
```

Die Textur wird von einem Ausschnitt des Bildspeichers kopiert, dessen linke untere Ecke die Bildschirmkoordinaten (*x*, *y*) besitzt und dessen Höhe und Breite durch die Parameter *width* und *height* spezifiziert ist. Alle anderen Parameter sind die gleichen wie bei dem `glTexImage2D()`-Befehl. Ob vom *Front* oder *Back Buffer* gelesen wird (falls man im *Double Buffer Modus* rendert, Abschnitt 14.1), legt man durch den Befehl `glReadBuffer(GL_FRONT)` bzw. `glReadBuffer(GL_BACK)` fest.

Außerdem existiert in OpenGL auch noch ein Befehl zum Ersetzen eines Texturausschnitts mit Daten aus dem Bildspeicher (*Frame Buffer*). Dies ist das Gegenstück zum bereits beschriebenen `glTexSubImage2D()`-Befehl, mit der gleichen Funktionalität, nur dass diesmal die Texturdaten aus dem Bildspeicher kopiert werden. Der dazu gehörige OpenGL-Befehl lautet:

```
glCopyTexSubImage2D ( GLenum target, GLint level,
                      GLint xoffset, GLint yoffset, GLint x, GLint y,
                      GLsizei width, GLsizei height )
```

Die Textur wird von einem Ausschnitt des Bildspeichers kopiert, dessen linke untere Ecke die Bildschirmkoordinaten (*x*, *y*) besitzt und dessen Höhe und Breite durch die Parameter *width* und *height* spezifiziert ist. Die Parameter *xoffset* und *yoffset* definieren eine Verschiebung des zu ersetzenden Texturausschnitts vom Texturkoordinatenursprung links unten aus gesehen um *xoffset* Texel nach rechts und um *yoffset* nach oben. Alle anderen Parameter und Einstellungen sind die gleichen wie bei dem `glCopyTexImage2D()`-Befehl.

13.1.2 Textur-Filter

Nach der Spezifikation liegt eine Textur als rechteckiges Array von Farbwerten vor. Diese Textur wird jetzt auf ein Polygon bzw. auf eine Oberfläche gemappt, indem jedem Vertex eine Texturkoordinate zugeordnet wird (Abschnitt 13.1.6). Dabei findet eine Abbildung

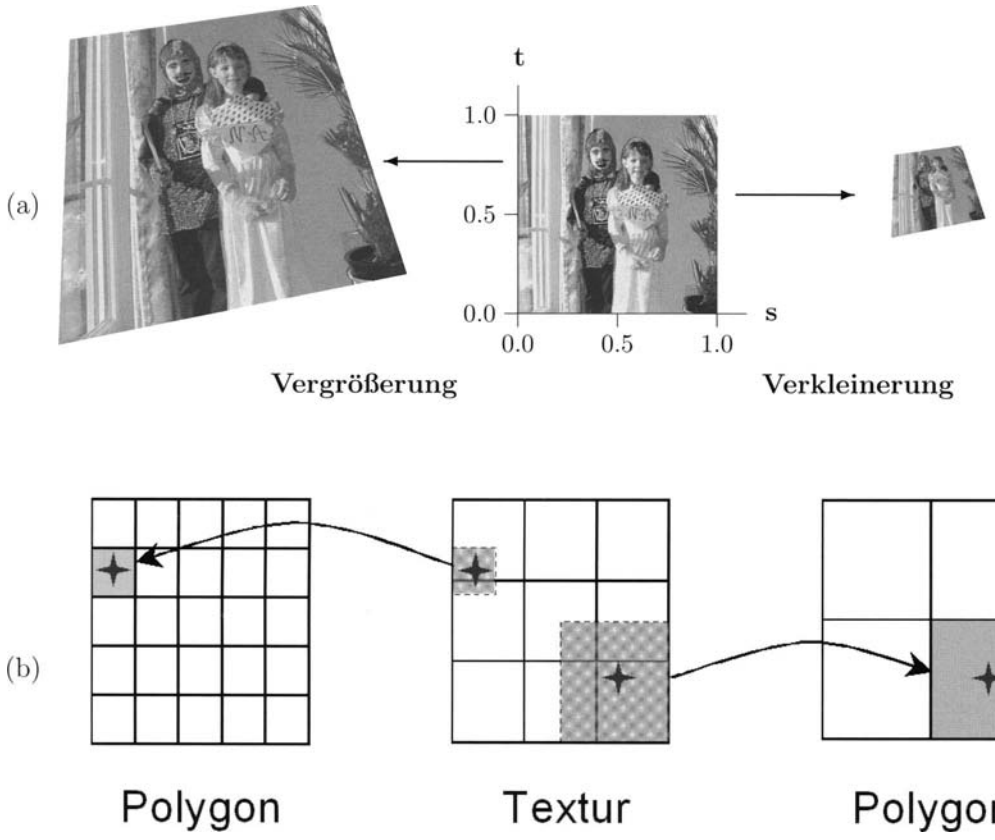


Bild 13.4: Vergrößerung (links) und Verkleinerung (rechts) von Texturen beim Mapping auf ein Polygon: (a) Ein Bildbeispiel. (b) Die zu einem Pixel korrespondierende Fläche in der Textur.

vom 2-dimensionalen Texturkoordinatensystem in das 3-dimensionale Objektkoordinatensystem statt, die häufig mit einer Verzerrung verbunden ist (Bild 13.11). Anschließend finden die Modell- und Augenpunktstransformationen statt, die im Falle der Verwendung der Skalierungsfunktion (`glScalef`) eine weitere Quelle für Verzerrungen sind. Im letzten Schritt werden die Polygone durch die Projektions- und Viewporttransformation wieder auf 2-dimensionale Bildschirmkoordinaten abgebildet und somit perspektivisch verzerrt. Folglich stimmt die Größe eines Texels praktisch nie mit der Größe eines Pixels am Bildschirm überein. Je nachdem welche Transformationen durchgeführt werden und wie die Texturkoordinatenzuordnung abläuft, muss die Textur entweder vergrößert (*Magnification*) oder verkleinert (*Minification*) werden (Bild 13.4-a). Umgekehrt ausgedrückt entspricht jedem Pixel auf dem Bildschirm entweder ein kleiner Teil eines einzigen Texels (*Magnification*) oder gleich mehrere Texel (*Minification*), wie in Bild 13.4-b dargestellt.

Es stellt sich in jedem Fall die Frage, welche Texel-Farbwerte benutzt werden sollen. Man hat es also beim Texture Mapping mit der klassischen Abtastproblematik zu tun, die ebenso in der Bildverarbeitung bei der Modifikation der Ortskoordinaten auftritt (Kapitel 22). Bei der Abtastung einer Textur entstehen immer dann Aliasing-Effekte (Bild 10.2), wenn die höchste in der Textur vorkommende Ortsfrequenz in die Nähe der Abtastrate kommt oder diese überschreitet, wie in Kapitel 10 ausführlich erläutert wird. Zur Verminderung der störenden Aliasing-Effekte werden deshalb Tiefpass-Filter (Kapitel 18 und Abschnitt 21.5) eingesetzt, die die hohen Ortsfrequenzen in der Textur unterdrücken und somit auch die Aliasing-Effekte. Aus Rechenzeitgründen ist die Auswahl an Tiefpass-Filtern in OpenGL nicht besonders groß: im einfachsten Fall wird überhaupt kein Filter eingesetzt, sondern es wird nur der Bildpunkt in der Textur ausgewählt, dessen Koordinaten dem Pixelzentrum am nächsten liegen; als echter Tiefpass-Filter wird der bewegte Mittelwert (Abschnitt 18.3) mit einem 2 x 2 Texel großen Filterkern angeboten. Weitere Filter werden im nächsten Abschnitt 13.1.3 besprochen.

Um OpenGL mitzuteilen, welche Textur-Filter für die Vergrößerung (*Magnification*) und für die Verkleinerung (*Minification*) verwendet werden sollen, dienen die in der folgenden Tabelle dargestellten Varianten des `glTexParameter`-Befehls:

Skalar-Form
`glTexParameterf(GLenum` *target*`, GLenum` *name*`, GLfloat` *param*`)`
`glTexParameteri(GLenum` *target*`, GLenum` *name*`, GLint` *param*`)`
Vektor-Form
`glTexParameterfv(GLenum` *target*`, GLenum` *name*`, GLfloat` **param*`)`
`glTexParameteriv(GLenum` *target*`, GLenum` *name*`, GLint` **param*`)`

Der erste Parameter des `glTexParameter`-Befehls, *target*, legt fest, ob es sich um eine 1-, 2- oder 3-dimensionale Textur handelt. Die möglichen Werte sind daher `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_3D`. Der zweite Parameter, `name`, legt in diesem Zusammenhang fest, ob der Vergrößerungsfilter (`GL_TEXTURE_MAG_FILTER`) oder der Verkleinerungsfilter (`GL_TEXTURE_MIN_FILTER`) spezifiziert werden soll (der `glTexParameter`-Befehl dient auch noch dazu, andere Eigenschaften des Texture Mappings zu spezifizieren). Der dritte Parameter, `param`, legt fest, welche der beiden Standard-Filter-Methoden verwendet werden soll:

- `GL_NEAREST`:
  Die Farbwerte des Texels, welches dem Pixel-Zentrum am nächsten ist, werden benutzt. Dies ist die schnellste Variante, führt aber zu Aliasing-Effekten (Bild 13.5-a).

- `GL_LINEAR`:
  Es wird ein linearer Mittelwert aus den Farbwerten des 2 x 2 Texel-Arrays gebildet, welches dem Pixel-Zentrum am nächsten liegt (bei 3-dimensionalen Texturen ist das Array ein Kubus aus 2 x 2 x 2 Texeln, bei 1-dimensionalen Texturen sind es nur 2

Texel). Dieser Textur-Filter ist zwar etwas langsamer, führt aber zu glatteren Bildern mit geringeren Aliasing-Störungen (Bild 13.5-b).

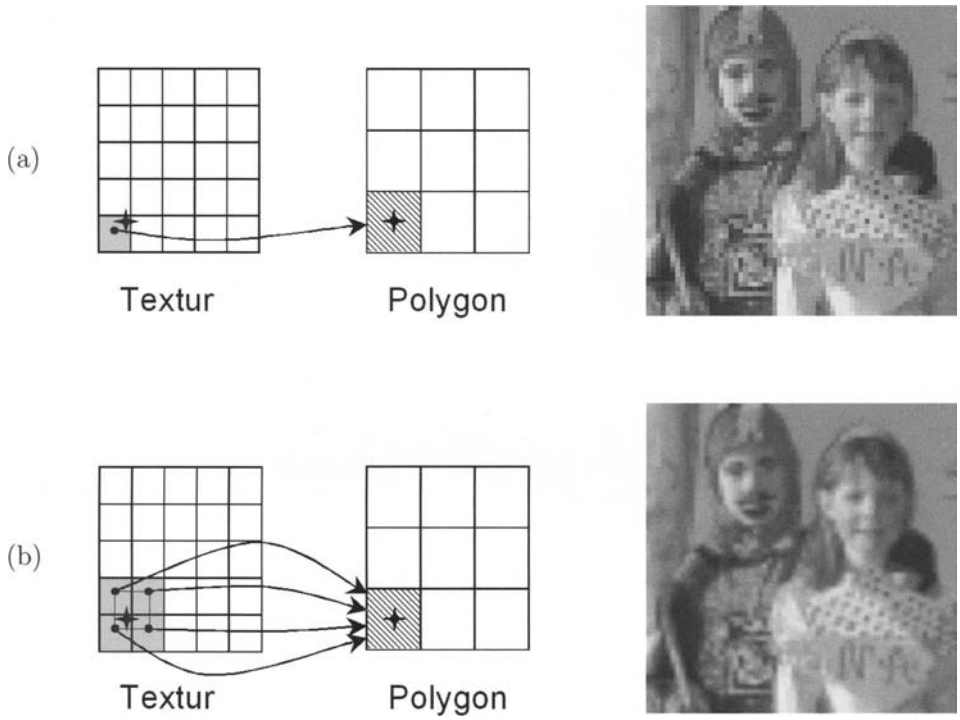


Bild 13.5: Die Standard-Textur-Filter in OpenGL: links die Prinzipien, rechts die Bildbeispiele. (a) `GL_NEAREST`: Die Farbwerte des Texels, welches dem Pixel-Zentrum am nächsten ist, werden benutzt. Dies ist die schnellste Variante, die aber zu Aliasing-Effekten führt. (b) `GL_LINEAR`: Es wird ein linearer Mittelwert aus den Farbwerten des 2 x 2 Texel-Arrays gebildet, welches dem Pixel-Zentrum am nächsten liegt. Dieser Textur-Filter ist zwar etwas langsamer, führt aber zu glatteren Bildern mit geringeren Aliasing-Störungen.

13.1.3 Gauß-Pyramiden-Texturen (MipMaps)

Die Textur-Filterung mit dem bewegten Mittelwert (`GL_LINEAR`) funktioniert befriedigend, solange der Vergrößerungsfaktor kleiner als 2 und der Verkleinerungsfaktor größer als $\frac{1}{2}$ ist. Bei Vergrößerungsfaktoren über 2 wird ein Texel auf mehr als vier Pixel verschmiert, so dass als Ergebnis ein zunehmend unschärferes Bild am Bildschirm entsteht. Dagegen kann man fast nichts tun (außer den Augenpunkt nicht zu nah an texturierte Polygone heranzuführen), denn eine Textur kann eben nicht detaillierter abgebildet werden, als sie definiert wurde. Verkleinerungsfaktoren kleiner als $\frac{1}{2}$ treten in der 3D-Computergrafik

Bild 13.6: Gauß-Pyramiden-Textur (MipMap): Die Original-Textur mit der höchsten Auflösung ist ganz links zu sehen (MipMap-Level 0). Die verkleinerten Varianten (MipMap-Levels $1, 2, 3, ...$) nach rechts hin wurden zuerst tiefpass-gefiltert und dann in jeder Dimension um den Faktor 2 unterabgetastet. Stapelt man die immer kleineren MipMap-Levels übereinander, entsteht eine Art Pyramide (daher der Name).

fast immer auf, denn texturierte Objekte werden durch die perspektivische Projektion mit zunehmender Entfernung vom Augenpunkt immer stärker verkleinert. Nach dem Abtasttheorem (Kapitel 10) darf eine Verkleinerung einer Textur, d.h. eine Unterabtastung, erst erfolgen, nachdem die hohen Ortsfrequenzen in der Textur durch ein Tiefpass-Filter eliminiert wurden. Bei einer Verkleinerung um den Faktor $\frac{1}{2}$ muss vor der Unterabtastung die maximal in der Textur enthaltene Ortsfrequenz halbiert werden, bei einem Verkleinerungsfaktor $\frac{1}{4}$ muss die maximale Ortsfrequenz geviertelt werden und so weiter. Die Größe des mindestens nötigen Tiefpass-Filterkerns beträgt je nach Verkleinerungsfaktor 2 x 2, 4 x 4 usw., d.h. die Zahl der Operationen zur Berechnung des tiefpassgefilterten und anschließend verkleinerten Bildes steigt quadratisch mit dem inversen Verkleinerungsfaktor an. Die Standardmethode zur Verringerung dieses Rechenaufwands besteht darin, stufenweise vorzugehen: im ersten Schritt tiefpassfiltern und dann die Textur in jeder Dimension um den Faktor $\frac{1}{2}$ verkleinern, im zweiten Schritt, ausgehend von der verkleinerten Textur, wieder tiefpassfiltern und um den Faktor $\frac{1}{2}$ verkleinern und so weiter, bis die Textur auf die Minimalgröße von 1 x 1 Texel geschrumpft ist. Eine solche Folge von sukzessive um den Faktor $\frac{1}{2}$ verkleinerter Bilder bezeichnet man als Gauß-Pyramide (Kapitel 28) und in der Computergrafik auch als „*MipMap*“ („*Mip*“ ist eine Abkürzung für den lateinischen Begriff „multum in parvo“, d.h. „viele in einem“ und der englische Begriff „*Map*“ bedeutet in diesem Zusammenhang Karte oder Textur). Beispiele sind in Bild 13.6 und Bild 28.3 zu sehen. Die unterste Stufe der Gauß-Pyramide $\mathbf{G}_0$, also die Original-Textur wird als MipMap-Level

0 bezeichnet, die n-te Verkleinerungsstufe $\mathbf{G}_n$ als MipMap-Level n. Der Parameter *level* bei der Textur-Definition mit dem `glTexImage2D()`-Befehl ist der MipMap-Level.

Zu „einer" Textur gehört in OpenGL die ganze Gauß-Pyramide, d.h. alle n Texturebenen (MipMap-Levels), die die jeweiligen Auflösungsstufen enthalten. Falls die Original-Textur nicht quadratisch, sondern rechteckig ist, degenerieren die Texturebenen ab einem bestimmten MipMap-Level zu einer 1-dimensionalen Textur. Um noch mehr Rechenzeit einzusparen, werden bei interaktiven Anwendungen alle Auflösungsstufen einer MipMap vorab berechnet, d.h. entweder offline oder während der Initialisierung. Nachteil dieser Lösung ist, dass der benötigte Texturspeicherbedarf um knapp $\frac{1}{3}$ zunimmt, denn die Fläche sinkt pro Texturebene um den Faktor $\frac{1}{4}$, so dass die Summe aller MipMap-Levels > 0 sehr nahe an den Grenzwert der geometrischen Reihe $\frac{1}{3} = \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + ...$ heran kommt.

13.1.3.1 Erzeugung von Gauß-Pyramiden (MipMaps)

Es gibt zwei prinzipiell verschiedene Arten, Gauß-Pyramiden (MipMaps) zu erzeugen:

- Manuell: indem man jeden notwendigen MipMap-Level in einem (externen) Programm vorab berechnet, dann einliest und anschließend durch mehrfachen Aufruf des OpenGL-Befehls `glTexImage2D()` definiert. Diese Variante hat den Vorteil, dass man sehr viel bessere Tiefpass-Filter einsetzen kann als den in OpenGL üblichen bewegten Mittelwert mit einem 2 x 2 Texel großen Filterkern, bei dem noch ein gewisser Anteil an Abtast-Artefakten übrig bleibt. Mit einem rechenaufwändigeren Gauß-Tiefpass-Filter, wie er in Kapitel 28 detailliert beschrieben ist, lassen sich fast alle Abtast-Artefakte eliminieren. Dies ist auch der Ursprung der Bezeichnung Gauß-Pyramide.

- Automatisch: indem man die OpenGL Utility Routine
  `gluBuild2DMipmaps( GLenum` *`target`*`, GLint` *`internalFormat`*`, GLsizei` *`width`*`, GLsizei` *`height`*`, GLenum` *`format`*`, GLenum` *`type`*`, const GLvoid *`*`image` `)`
  benutzt, die aus einer eingegebenen Textur (*`*image`*) alle MipMap-Levels mit Hilfe des bewegten Mittelwerts berechnet und automatisch den Befehl `glTexImage2D()` zur Definition aller Levels aufruft (die Parameter sind die gleichen wie beim `glTexImage2D()`-Befehl).

Falls der `glTexParameter()`-Befehl mit den Parametern `GL_TEXTURE_2D`, `GL_GENERATE_MIPMAP` und `GL_TRUE` aufgerufen wird, führt eine Änderung der Original-Textur (MipMap-Level 0) automatisch, d.h. auch während einer interaktiven Anwendung, zu einer Neuberechnung aller weiteren MipMap-Levels gemäß des `gluBuild2DMipmaps()`-Befehls. Für spezielle Anwendungen besteht seit OpenGL Version 1.2 auch noch die Möglichkeit, nur einen bestimmten Teil der MipMap-Levels mit Hilfe des Befehls `gluBuild2DMipmapLevels()` zu erzeugen. In diesem Zusammenhang macht es auch Sinn, die zur Textur-Filterung verwendbaren MipMap-Levels auf das vorher erzeugte Subset einzuschränken. Für eine detaillierte Darstellung dieser Funktionalitäten wird aus Platzgründen auf den OpenGL Programming Guide [Shre05] verwiesen.

13.1.3.2 Auswahl der MipMap-Levels

Für die Auswahl der adäquaten MipMap-Levels berechnet OpenGL automatisch zwei Verkleinerungsfaktoren ρ_x und ρ_y zwischen der Texturgröße (in Texel) und der Größe des texturierten Polygons (in Pixel) für jede Dimension x und y. Aus dem Maximum der beiden inversen Verkleinerungsfaktoren $(\max(\frac{1}{\rho_x}, \frac{1}{\rho_y})$ berechnet OpenGL noch einen weiteren Skalierungsfaktor λ durch Logarithmierung zur Basis 2, d.h.

$$\lambda = \log_2(\max(\frac{1}{\rho_x}, \frac{1}{\rho_y})) \tag{13.1}$$

Falls z.B. der maximale Verkleinerungsfaktor $\rho = \frac{1}{4}$ ist, d.h. dass in einer Dimension 4 Texel auf ein Pixel abgebildet werden, ist $\lambda = 2$, bei einem maximalen Verkleinerungsfaktor von $\rho = \frac{1}{8}$ ist $\lambda = 3$. Wird eine Textur 2-mal um den Faktor 2 verkleinert (dies entspricht MipMap-Level 2), wird aus 4 Texel in einer Dimension ein Texel. Wird eine Textur 3-mal um den Faktor 2 verkleinert (dies entspricht MipMap-Level 3), wird aus 8 Texel in einer Dimension ein Texel. Folglich gibt der durch OpenGL berechnete Skalierungsfaktor λ also den zur Größe des Polygons am Bildschirm passenden MipMap-Level an.

13.1.3.3 MipMap-Verkleinerungsfilter

Nachdem die benötigten MipMap-Levels definiert sind, kann der in Abschnitt 13.1.2 bereits vorgestellte OpenGL-Befehl
`glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GLint param)` benutzt werden, um einen MipMap-Minification-Filter festzulegen. Der dritte noch freie Parameter, `param`, legt fest, welcher der vier MipMap-Minification-Filter verwendet werden soll:

- `GL_NEAREST_MIPMAP_NEAREST`:
  Durch Auf- oder Abrunden des Skalierungsfaktors λ wird der zur Polygongröße passendste MipMap-Level ausgewählt. Die Farbwerte des Texels, welches dem Pixel-Zentrum am nächsten ist, werden benutzt. Dies ist die schnellste Variante, führt aber zu Aliasing-Effekten. Dennoch ist dieser Filter immer noch sehr viel besser als der `GL_NEAREST`-Filter, denn er wählt ein Texel aus dem passenden MipMap-Level aus anstatt ein Texel aus der evtl. viel zu großen Original-Textur.

- `GL_LINEAR_MIPMAP_NEAREST`:
  Wie vorher wird durch Auf- oder Abrunden des Skalierungsfaktors λ der zur Polygongröße passendste MipMap-Level ausgewählt. Es wird ein linearer Mittelwert aus den Farbwerten des 2 x 2 Texel-Arrays gebildet, welches dem Pixel-Zentrum am nächsten liegt.

- `GL_NEAREST_MIPMAP_LINEAR`:
  Durch Auf- und Abrunden des Skalierungsfaktors λ wird sowohl der bzgl. der Polygongröße nächstkleinere MipMap-Level ausgewählt, als auch der nächstgrößere. Danach wird ein linearer Mittelwert gebildet zwischen den beiden Texel-Farbwerten, die dem Pixel-Zentrum im jeweiligen MipMap-Level am nächsten liegen.

- `GL_LINEAR_MIPMAP_LINEAR`:
  Wie vorher wird durch Auf- und Abrunden des Skalierungsfaktors λ sowohl der bzgl. der Polygongröße nächstkleinere MipMap-Level ausgewählt, als auch der nächstgrößere. Anschließend wird für jeden der beiden ausgewählten MipMap-Level ein linearer Mittelwert aus den Farbwerten des 2 x 2 Texel-Arrays gebildet, welches dem Pixel-Zentrum am nächsten liegt. Aus diesen beiden Mittelwerten wird durch eine weitere Mittelung der endgültige Farbwert des Fragments berechnet. Da bei diesem aufwändigen Textur-Filter dreimal hintereinander ein linearer Mittelwert berechnet wird, bezeichnet man diese Filterung auch als „*tri-linear mipmap*".

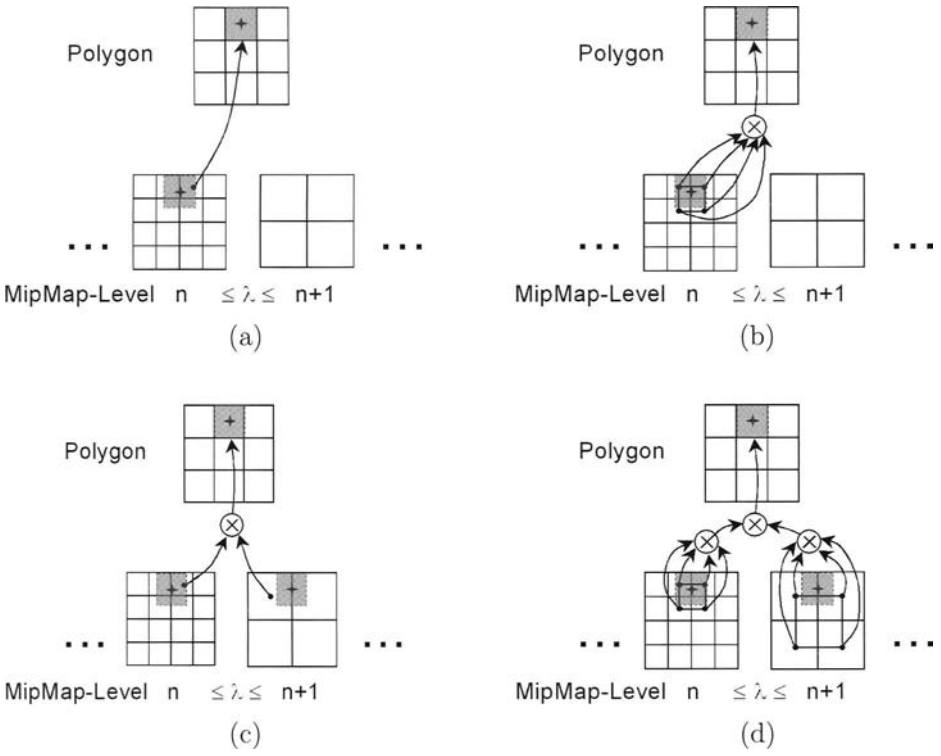


Bild 13.7: MipMap Verkleinerungsfilter: (a) `GL_NEAREST_MIPMAP_NEAREST`. (b) `GL_LINEAR_MIPMAP_NEAREST`. (c) `GL_NEAREST_MIPMAP_LINEAR`. (d) `GL_LINEAR_MIPMAP_LINEAR`.

In Bild 13.7 ist die Funktionsweise der vier MipMap-Minification-Filter grafisch erläutert. Der Rechenaufwand der verschiedenen Filtertypen steigt mit der Anzahl der Mittelungen und somit nach unten an, ebenso wie die erzielbare Bildqualität. Es gilt also, wie immer in der interaktiven 3D-Computergrafik, abzuwägen zwischen Rendering-Geschwindigkeit und Bildqualität.

13.1.3.4 Anisotrope Filter

Ein gewisses Problem taucht bei MipMap-Verkleinerungsfiltern immer dann auf, wenn eine Textur auf ein Polygon gemappt wird, das unter einem sehr flachen Blickwinkel betrachtet wird. Denn in diesem Fall wird eine Textur in s-Richtung evtl. nur wenig verkleinert, während sie in t-Richtung sehr stark verkleinert wird. Die nach 13.1 ausgewählten MipMap-Levels beziehen sich aber auf den stärksten Verkleinerungsfaktor und sind daher für die s-Richtung viel zu klein. Das Bild wird also horizontal unscharf. Abhilfe kann hier mit einer OpenGL-Extension für anisotrope Filter geschaffen werden, die im OpenGL Extensions Guide [Leng03] ausführlicher beschrieben ist. Die Grundidee dabei ist, den MipMap-Level auszuwählen, der für die geringere Verkleinerung in s-Richtung adäquat ist und gleichzeitig Aliasing-Effekte in t-Richtung zu vermeiden, indem man eine entsprechend größere Anzahl an Texel in t-Richtung zur Mittelung heranzieht. Dies entspricht einem asymmetrischen Tiefpass-Filterkern, der in t-Richtung verlängert ist. Dadurch wird in t-Richtung stärker tiefpass-gefiltert als in s-Richtung. Zur Aktivierung der anisotropen Filterung wird der Befehl `glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, GLint param)` aufgerufen, wobei *param* die maximale Anisotropie a_{max} festlegt.

Für die Auswahl der adäquaten MipMap-Levels berechnet OpenGL zunächst die Anzahl N der Abtastpunkte, die in t-Richtung (im allgemeinen in die Richtung, in die der stärkere Verkleinerungsfaktor auftritt) genommen werden. Mit $f_{max} = \max(\frac{1}{\rho_x}, \frac{1}{\rho_y})$ und $f_{min} = \min(\frac{1}{\rho_x}, \frac{1}{\rho_y})$ ergibt sich

$$N = \min(\frac{f_{max}}{f_{min}}, a_{max}) \tag{13.2}$$

Anschließend berechnet OpenGL den Skalierungsfaktor λ mit

$$\lambda = \log_2(\frac{f_{max}}{N}) \tag{13.3}$$

Falls die berechnete Anisotropie $\frac{f_{max}}{f_{min}}$ kleiner ist als der durch den `glTexParameterf()`-Befehl vorgegebene Wert a_{max}, berechnet sich der Skalierungsfaktor einfach zu $\lambda = \log_2(f_{min})$, d.h. es wird der MipMap-Level ausgewählt, der für die geringere Verkleinerung in s-Richtung adäquat ist. Damit bleibt das Bild horizontal scharf, und wegen des in t-Richtung ausgedehnten Tiefpass-Filters treten in dieser Richtung auch keine Aliasing-Effekte auf. Der Preis für die höhere Bildqualität ist natürlich wieder in Form einer geringeren Bildgenerierrate zu bezahlen.

13.1.4 Textur-Fortsetzungsmodus (Texture Wraps)

In vielen Fällen weisen Texturen regelmäßige Strukturen auf. Beispiele dafür sind künstlich erzeugte Oberflächen wie Ziegeldächer, gepflasterte oder geflieste Böden, tapezierte Wände, Textilien, homogene Fensterfronten bei Häusern etc. oder auch natürliche Oberflächen wie Sand, Granit, Rasen, Kornfelder oder ein Wald aus etwas größerer Entfernung. In all diesen

Fällen ist es sehr viel effektiver, einen kleinen charakteristischen Ausschnitt der gesamten Oberfläche, z.B. einen Dachziegel, als Textur zu definieren und mit diesem die gesamte Fläche zu „bepflastern“ (Bild 13.8-a,b).

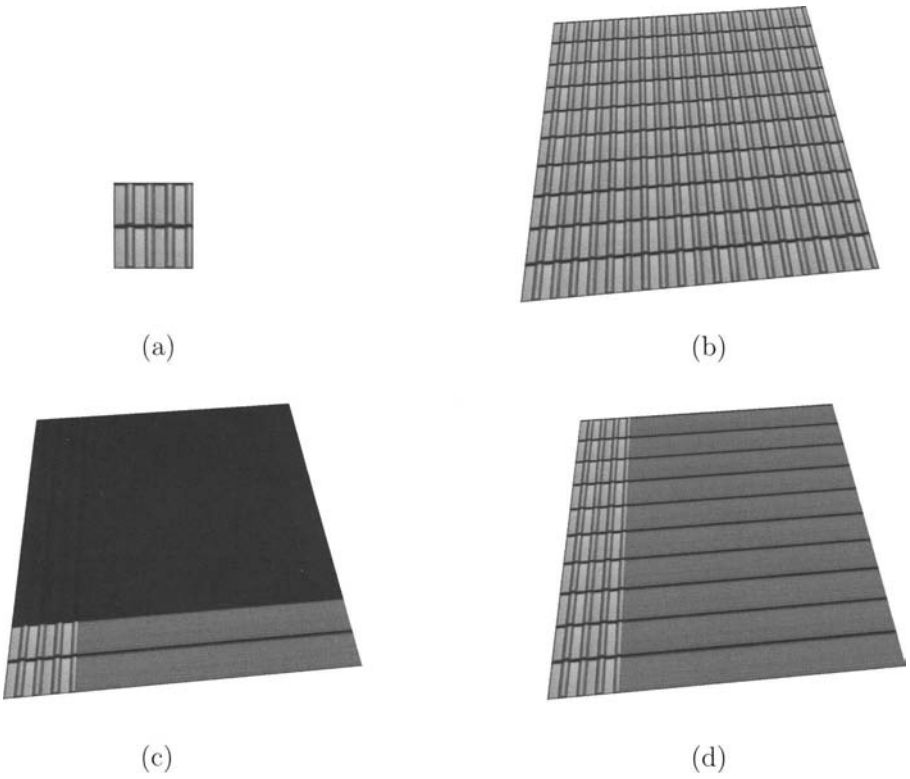


Bild 13.8: Textur-Fortsetzungsmodus (Texture Wraps): (a) Eine Textur mit einem kleinen Ausschnitt eines Ziegeldaches. (b) Durch wiederholtes Mapping der Textur wird ein größeres Polygon bedeckt (Textur-Fortsetzungsmodus `GL_REPEAT`). (c) Die letzte Zeile bzw. Spalte der Textur werden wiederholt, um den Rest der Textur auszufüllen (Textur-Fortsetzungsmodus `GL_CLAMP`). (d) In s-Richtung wird die letzte Spalte der Textur wiederholt (`GL_CLAMP`) und in t-Richtung die gesamte Textur (`GL_REPEAT`).

OpenGL ermöglicht diese Art der Texturwiederholung, indem es Texturkoordinaten außerhalb des Bereichs $[0, 1]$ zulässt. Wenn man einem Rechteck Texturkoordinaten zuordnet, die in jeder Richtung von 0.0 bis 5.0 gehen, wird z.B. die Dachziegel-Textur fünf Mal in jeder Richtung wiederholt, d.h. sie wird ingesamt 25 Mal auf das Rechteck gemappt, falls der Textur-Fortsetzungsmodus (Texture Wrap) auf den Wert “`GL_REPEAT`“ eingestellt ist. Bei diesem Modus wird einfach der ganzzahlige Anteil der Texturkoordinaten ignoriert, so dass die Textur periodisch auf der Fläche wiederholt wird. Damit die einzelnen Textur-„Fliesen“ nicht erkennbar sind, müssen die Farben und die Gradienten der Farbverläufe am

linken und am rechten Rand, sowie am unteren und oberen Rand möglichst ähnlich sein. Solche Texturen sind in der Praxis gar nicht so einfach zu erzeugen. Deshalb gibt es seit der OpenGL Version 1.4 als weiteren Textur-Fortsetzungsmodus "GL_MIRRORED_REPEAT", bei dem jede zweite Textur (d.h. falls der ganzzahlige Anteil der Texturkoordinaten ungerade ist) in einer (horizontal oder vertikal) gespiegelten Version wiederholt wird.

Eine andere Möglichkeit des Textur-Fortsetzungsmodus ist es, Texturkoordinaten außerhalb des Bereichs $[0, 1]$ zu kappen (GL_CLAMP), d.h. Werte größer als 1 auf 1 zu setzen und Werte kleiner als 0 auf 0 zu setzen. Dadurch wird die erste oder letzte Zeile bzw. Spalte der Textur immer wieder kopiert, falls als Textur-Filter GL_NEAREST eingestellt wurde (Bild 13.8-c). Falls als Textur-Filter GL_LINEAR eingestellt wurde, ist die Situation etwas komplizierter. In diesem Fall wird ein linearer Mittelwert aus den Farbwerten des 2 x 2 Texel-Arrays gebildet, welches dem Pixel-Zentrum am nächsten liegt. Wenn nun bei einer Textur ein Rand (*border*) definiert wurde (Abschnitt 13.1.1), gehen die Rand-Texel in die Berechnung des linearen Mittelwerts ein. Ist dies nicht gewünscht, kann durch Verwendung des Textur-Fortsetzungsmodus "GL_CLAMP_TO_EDGE" der Rand bei der Berechnung des linearen Mittelwerts ignoriert werden. Falls ausschließlich die Rand-Texel in die Berechnung des linearen Mittelwerts einfließen sollen, wird der Textur-Fortsetzungsmodus "GL_CLAMP_TO_BORDER" ausgewählt.

Der Textur-Fortsetzungsmodus kann separat für die s- und t-Richtung ausgewählt werden, so dass es z.B. auch möglich ist, in t-Richtung den Modus GL_REPEAT festzulegen und in s-Richtung den Modus GL_CLAMP (Bild 13.8-d). Der Textur-Fortsetzungsmodus wird mit dem in Abschnitt 13.1.2 bereits vorgestellten OpenGL-Befehl

`glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GLint` *`param`*`)` bzw.
`glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GLint` *`param`*`)`

festgelegt. Der dritte noch freie Parameter, `param`, bestimmt, welcher der fünf Textur-Fortsetzungsmodi verwendet werden soll:

- GL_REPEAT: Wiederholung der Textur.
- GL_MIRRORED_REPEAT: Wiederholung der Textur, wobei jede zweite Textur gespiegelt wird.
- GL_CLAMP: Wiederholung der ersten oder letzten Zeile bzw. Spalte der Textur, falls als Textur-Filter GL_NEAREST eingestellt wurde. Wiederholung der gemittelten Farbwerte der ersten oder letzten *beiden* Zeilen bzw. Spalten der Textur, falls als Textur-Filter GL_LINEAR eingestellt wurde. Wenn ein Textur-Rand definiert wurde, geht dieser als letzte Zeile bzw. Spalte in die Mittelung ein.
- GL_CLAMP_TO_EDGE: wie GL_CLAMP, allerdings wird der Textur-Rand ignoriert.
- GL_CLAMP_TO_BORDER: wie GL_CLAMP, wobei ausschließlich der Textur-Rand in die Wiederholung eingeht.

Zusätzlich besteht noch die Möglichkeit, explizit die Farbe des Texturrandes festzulegen. Dazu dient wieder der `glTexParameter*()`-Befehl mit den Parametern:

`glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, GLfloat *bordercolor)`

13.1.5 Mischung von Textur- und Beleuchtungsfarbe (Texture Environment)

Als Ergebnis liefert die Textur-Filterung einen Farbwert für das in Bearbeitung stehende Fragment (so wird ein Pixel genannt, solange es noch nicht endgültig im Bildspeicher steht). Diesen Farbwert bezeichnet man als Textur-Fragmentfarbe $\mathbf{g_t}$. Außerdem steht von der Beleuchtungsrechnung und der anschließenden Rasterisierung noch die gemäß Gouraud linear interpolierte Fragmentfarbe $\mathbf{g_f}$ zur Verfügung. Wie diese Farben miteinander (oder evtl. mit einer weiteren Hintergrundfarbe) gemischt werden, legt man durch Auswahl einer sogenannten Textur-Funktion (*Texture Environment*) fest (Bild 13.2). Dazu dienen die in der folgenden Tabelle dargestellten Varianten des `glTexEnv`-Befehls:

Skalar-Form
`glTexEnvf(GLenum target, GLenum name, GLfloat param)`
`glTexEnvi(GLenum target, GLenum name, GLint param)`
Vektor-Form
`glTexEnvfv(GLenum target, GLenum name, GLfloat *param)`
`glTexEnviv(GLenum target, GLenum name, GLint *param)`

Der erste Parameter des `glTexEnv`-Befehls, *target*, muss in diesem Zusammenhang immer (`GL_TEXTURE_ENV`) sein. Falls der zweite Parameter, `name`, den Wert (`GL_TEXTURE_ENV_COLOR`) annimmt, wird eine Textur-Hintergrundfarbe $\mathbf{g_h}$ definiert, die bei einer speziellen Textur-Funktion (`GL_BLEND`) einfließt. In diesem Fall werden über den dritten Parameter `*param` die vier Farbkomponenten R,G,B,A in einem Array übergeben. Wenn aber der zweite Parameter, `name`, den Wert (`GL_TEXTURE_ENV`) annimmt, wird eine der folgenden Textur-Funktionen mit dem dritten Parameter, `param`, festgelegt:

- `GL_REPLACE`:
  Die Textur-Fragmentfarbe wird zu 100% übernommen, d.h. die aus der Beleuchtungsrechnung interpolierte Fragmentfarbe wird ersetzt durch die Textur-Fragmentfarbe. Nach dem Texture Mapping ist die resultierende Fragmentfarbe $\mathbf{g_r}$ einfach die Textur-Fragmentfarbe:

$$\mathbf{g_r} = \mathbf{g_t} \tag{13.4}$$

- GL_MODULATE:
  Die Textur-Fragmentfarbe wird mit der interpolierten Fragmentfarbe aus der Beleuchtungsrechnung moduliert, d.h. komponentenweise multipliziert:

$$\mathbf{g_r} = \mathbf{g_t} * \mathbf{g_f} \Longleftrightarrow \begin{pmatrix} R_r \\ G_r \\ B_r \\ A_r \end{pmatrix} = \begin{pmatrix} R_t \cdot R_f \\ G_t \cdot G_f \\ B_t \cdot B_f \\ A_t \cdot A_f \end{pmatrix} \tag{13.5}$$

- GL_DECAL:
  Abhängig vom Wert der Alpha-Komponente des Texels wird entweder die Textur-Fragmentfarbe stärker gewichtet, oder die interpolierte Fragmentfarbe aus der Beleuchtungsrechnung. Bei einem Alpha-Wert von 1 wird ausschließlich die Textur-Fragmentfarbe verwendet, bei einem Alpha-Wert von 0 ausschließlich die interpolierte Fragmentfarbe aus der Beleuchtungsrechnung:

$$\mathbf{g_r} = \begin{pmatrix} R_r \\ G_r \\ B_r \\ A_r \end{pmatrix} = \begin{pmatrix} (1 - A_t) \cdot R_f + A_t \cdot R_t \\ (1 - A_t) \cdot G_f + A_t \cdot G_t \\ (1 - A_t) \cdot B_f + A_t \cdot B_t \\ A_f \end{pmatrix} \tag{13.6}$$

- GL_BLEND:
  Abhängig vom Wert der Farbkomponente des Texels wird entweder die Textur-Hintergrundfarbe stärker gewichtet, oder die interpolierte Fragmentfarbe aus der Beleuchtungsrechnung. Bei einem Wert der Textur-Farbkomponente von 1 wird ausschließlich die Textur-Hintergrundfarbe verwendet, bei einem Wert von 0 auschließlich die interpolierte Fragmentfarbe aus der Beleuchtungsrechnung:

$$\mathbf{g_r} = \begin{pmatrix} R_r \\ G_r \\ B_r \\ A_r \end{pmatrix} = \begin{pmatrix} (1 - R_t) \cdot R_f + R_t \cdot R_h \\ (1 - G_t) \cdot G_f + G_t \cdot G_h \\ (1 - B_t) \cdot B_f + B_t \cdot B_h \\ A_t \cdot A_f \end{pmatrix} \tag{13.7}$$

- GL_ADD:
  In diesem Fall werden die Textur-Fragmentfarbe und die interpolierte Fragmentfarbe aus der Beleuchtungsrechnung addiert:

$$\mathbf{g_r} = \begin{pmatrix} R_r \\ G_r \\ B_r \\ A_r \end{pmatrix} = \begin{pmatrix} R_t + R_f \\ G_t + G_f \\ B_t + B_f \\ A_t \cdot A_f \end{pmatrix} \tag{13.8}$$

- GL_COMBINE:
  Diese Textur-Funktion wird ausschließlich für Mehrfach-Texturen (Multitexturing) benötigt (Abschnitt 13.2). Aufgrund der vielfältigen Kombinationsmöglichkeiten wird hier aus Platzgründen auf die OpenGL Spezifikation [Sega04] und den OpenGL Programming Guide [Shre05] verwiesen.

Die oben aufgelisteten Textur-Funktionen gelten für eine Textur mit den vier Komponenten R,G,B,A. Wie in Abschnitt 13.1.1 beschrieben, gibt es noch fünf weitere grundlegende interne Texturformate: GL_RGB (drei Komponenten), GL_LUMINANCE_ALPHA (zwei Komponenten), GL_LUMINANCE, GL_INTENSITY und GL_ALPHA (je eine Komponente). Die Formeln (13.4 – 13.8) gelten (bis auf zwei Ausnahmen) auch für diese internen Texturformate, wenn die in der folgenden Tabelle aufgelisteten Ersetzungen für die Textur-Fragmentfarbe $\mathbf{g_t} = (R_t, G_t, B_t, A_t)^T$ vorgenommen werden:

Internes Texturformat	R_t	G_t	B_t	A_t
GL_RGB	R_t	G_t	B_t	1
GL_ALPHA	1	1	1	A_t
GL_LUMINANCE	L_t	L_t	L_t	1
GL_LUMINANCE_ALPHA	L_t	L_t	L_t	A_t
GL_INTENSITY	I_t	I_t	I_t	I_t

Die beiden Ausnahmen betreffen das interne Texturformat GL_INTENSITY für die Textur-Funktionen GL_BLEND und GL_ADD. In diesen Fällen berechnet sich die Alpha-Komponente des resultierenden Fragments genauso wie die anderen Farbkomponenten. Für GL_BLEND gilt:

$$A_r = (1 - A_t) \cdot A_f + A_t \cdot A_h = (1 - I_t) \cdot A_f + I_t \cdot A_h \tag{13.9}$$

Für GL_ADD gilt:

$$A_r = A_t + A_f = I_t + A_f \tag{13.10}$$

Am häufigsten benutzt werden die Textur-Funktionen GL_REPLACE, GL_MODULATE und GL_DECAL. Die Textur-Funktion GL_REPLACE wird in Situationen verwendet, in denen die Beleuchtungsrechnung deaktiviert ist, oder keine Lichtquelle vorhanden bzw. erwünscht ist, wie z.B. bei einer Leuchtreklame in der Nacht. Die Textur-Funktion GL_MODULATE vereint die Vorteile der Beleuchtungsrechnung (Abschattung von Flächen, die von der Lichtquelle abgewandt sind und somit 3D-Formwahrnehmung) und des Texture Mappings (Fotorealistische Darstellung von Oberflächendetails). Zu diesem Zweck werden in der Regel Polygone mit weißen ambienten, diffusen und spekularen Materialeigenschaften verwendet, so dass die Helligkeit durch die Beleuchtungsrechnung bestimmt wird und die Farbe durch die Textur. Die Anwendung der Textur-Funktion GL_DECAL ist nur in Verbindung mit dem

internen Format `GL_RGBA` sinnvoll, denn hier hängt die Farbmischung zwischen der Textur-Fragmentfarbe und der interpolierten Fragmentfarbe aus der Beleuchtungsrechnung linear vom Wert der Alpha-Komponente der Textur ab. Damit ist es möglich, z.B. einen Schriftzug auf ein farbiges Polygon aufzubringen, oder Fahrbahnmarkierungen auf Straßenpolygone. An den Stellen der Textur, an denen sich der Schriftzug bzw. die Fahrbahnmarkierung befindet, muss der Alpha-Wert auf 1 gesetzt werden und an alle anderen auf 0. Bei einer solchen Alpha-Textur wird somit nur der Schriftzug bzw. die Fahrbahnmarkierung aufgemappt, an allen anderen Stellen scheint die ursprüngliche Polygonfarbe durch. Falls man Mehrfach-Texturierung zur Verfügung hat, darf das Polygon auch vor dem Mappen der Decal-Textur schon anderweitig texturiert sein.

Addition der spekularen Farbe nach der Texturierung

Durch das Texture Mapping gehen ausgeprägte Glanzlichter, d.h. die spekulare Lichtkomponente, meistens wieder verloren, da eine Textur nur per Zufall genau an den Stellen sehr hell ist, an denen auch die Glanzlichter auftreten. In der Realität übertünchen aber die Glanzlichter die Texturfarben. Deshalb bietet OpenGL, wie in Abschnitt 12.1.3.6 beschrieben, die Möglichkeit, zwei Farben pro Vertex zu berechnen: eine primäre Farbe, die die diffusen, ambienten und emissiven Beleuchtungsanteile enthält, und eine sekundäre Farbe, die alle spekularen Anteile enthält. Während des Textur-Mappings wird nur die primäre Farbe mit der Texturfarbe gemäß der in diesem Abschnitt beschriebenen Textur-Funktionen kombiniert. Danach wird auf die kombinierte Farbe die sekundäre Farbe addiert und das Ergebnis auf den Wertebereich [0, 1] beschränkt. Dieses Verfahren führt zu deutlich besser sichtbaren Glanzlichteffekten beim Texture Mapping.

13.1.6 Zuordnung von Texturkoordinaten

Wenn man eine Objektoberfläche mit einer Textur überziehen will, muss jedem Punkt der Oberfläche ein Punkt der Textur zugeordnet werden. Da man es in OpenGL mit polygonalen Modellen zu tun hat, kann man sich dabei auf die Zuordnung von Texturkoordinaten zu den Vertices der Objektoberfläche beschränken. Die Rasterisierungsstufe der *Rendering Pipeline* führt eine lineare Interpolation der Texturkoordinaten zwischen den Vertices mit Hilfe des Scan-Line-Algorithmus (12.20) durch, so dass danach für jedes Pixel die nötigen Texturkoordinaten vorliegen. Die Zuordnung von Texturkoordinaten zu Vertices bezeichnet man als *Textur-Abbildung*. OpenGL stellt dafür zwei verschiedene Methoden zur Verfügung, nämlich die explizite Spezifikation von Texturkoordinaten pro Vertex und eine implizite Variante, bei der die Texturkoordinaten automatisch, d.h. mit Hilfe von Gleichungen aus den Vertexkoordinaten berechnet werden.

13.1.6.1 Explizite Zuordnung von Texturkoordinaten

Texturkoordinaten können ein, zwei, drei oder vier Komponenten besitzen. Sie werden mit (s, t, r, q) bezeichnet, um sie von normalen Objektkoordinaten (x, y, z, w) unterscheiden zu

können. Für 1-dimensionale Texturen benötigt man nur die s-Koordinate, t und r werden 0 gesetzt und $q = 1$. 2-dimensionale Texturen werden durch die zwei Koordinaten s und t beschrieben, wobei $r = 0$ und $q = 1$ gesetzt wird. Für 3-dimensionale Texturen werden drei Koordinaten angegeben s, t, r, wobei $q = 1$ gesetzt wird. Die vierte Koordinate q ist ebenso wie bei Objektkoordinaten ein inverser Streckungsfaktor, der dazu dient, homogene Texturkoordinaten zu erzeugen. Normalerweise wird $q = 1$ gesetzt, für projektive Textur-Abbildungen (Abschnitt 13.3) kann $q \neq 1$ werden.

Zur Spezifikation von Texturkoordinaten wird der Befehl `glTexCoord*(TYPE coords)` benützt, der in folgenden Ausprägungen existiert:

Skalar-Form	Vektor-Form	t	r	q
`glTexCoord1f(s)`	`glTexCoord1fv(vec)`	0.0	0.0	1.0
`glTexCoord1d(s)`	`glTexCoord1dv(vec)`	0.0	0.0	1.0
`glTexCoord1s(s)`	`glTexCoord1sv(vec)`	0.0	0.0	1.0
`glTexCoord1i(s)`	`glTexCoord1iv(vec)`	0.0	0.0	1.0
`glTexCoord2f(s,t)`	`glTexCoord2fv(vec)`		0.0	1.0
`glTexCoord2d(s,t)`	`glTexCoord2dv(vec)`		0.0	1.0
`glTexCoord2s(s,t)`	`glTexCoord2sv(vec)`		0.0	1.0
`glTexCoord2i(s,t)`	`glTexCoord2iv(vec)`		0.0	1.0
`glTexCoord3f(s,t,r)`	`glTexCoord3fv(vec)`			1.0
`glTexCoord3d(s,t,r)`	`glTexCoord3dv(vec)`			1.0
`glTexCoord3s(s,t,r)`	`glTexCoord3sv(vec)`			1.0
`glTexCoord3i(s,t,r)`	`glTexCoord3iv(vec)`			1.0
`glTexCoord4f(s,t,r,q)`	`glTexCoord4fv(vec)`			
`glTexCoord4d(s,t,r,q)`	`glTexCoord4dv(vec)`			
`glTexCoord4s(s,t,r,q)`	`glTexCoord4sv(vec)`			
`glTexCoord4i(s,t,r,q)`	`glTexCoord4iv(vec)`			

In der Skalar-Form des Befehls müssen die Texturkoordinaten im entsprechenden Datenformat (z.B. f = GLfloat) direkt übergeben werden, in der Vektor-Form des Befehls wird nur ein Zeiger (`vec`) auf ein Array übergeben, das die Texturkoordinaten im entsprechenden Datenformat enthält. Intern werden Texturkoordinaten als Gleitkommazahlen gespeichert. Andere Datenformate werden entsprechend konvertiert.

Die explizite Zuordnung der Texturkoordinaten geschieht innerhalb einer `glBegin-glEnd`-Klammer, indem der `glTexCoord*`-Befehl vor dem `glVertex*`-Befehl aufgerufen wird, wie in dem folgenden einfachen Beispiel gezeigt (Bild 13.9):

```
glBegin(GL_QUADS);
    glTexCoord2fv( t0 ); glVertex3fv( v0 );
    glTexCoord2fv( t1 ); glVertex3fv( v1 );
    glTexCoord2fv( t2 ); glVertex3fv( v2 );
    glTexCoord2fv( t3 ); glVertex3fv( v3 );
glEnd();
```

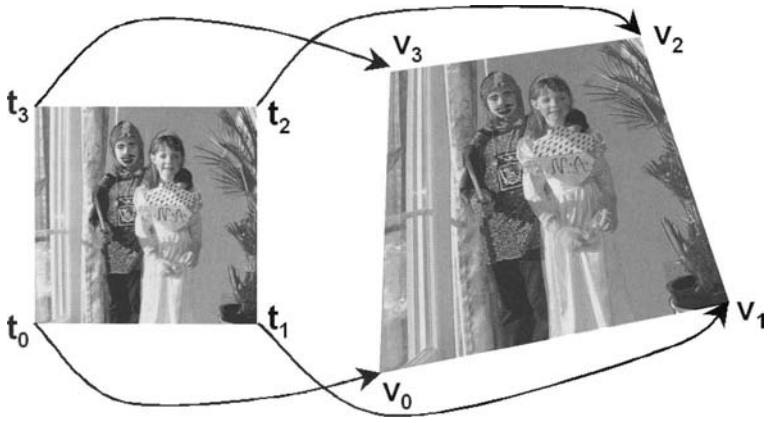


Bild 13.9: Explizite Zuordnung von Texturkoordinaten an jeden Vertex.

Texturkoordinaten sind damit Teil der Datenstruktur, die jedem Vertex zugeordnet werden kann, genau wie Normalenvektoren und Farben.

13.1.6.2 Textur-Abbildungen

Solange man eine quadratische oder rechteckige Textur auf ein planares Rechteck abbilden will, ist die Sache noch recht einfach. Man ordnet den vier Vertices des Rechtecks (gegen den Uhrzeigersinn) die Texturkoordinaten $(0,0)$, $(1,0)$, $(1,1)$ und $(0,1)$ zu. Für eine verzerrungsfreie Abbildung muss man das Höhen-Breiten-Verhältnis des Rechtecks und der Textur berücksichtigen. Angenommen, die Textur ist um den Faktor a_t höher als breit (es gilt $a_t = 2^n, n \in 0, \pm 1, \pm 2, \ldots$) und das rechteckige Polygon um den Faktor a_p höher als breit, dann gibt es zwei Möglichkeiten: entweder wird nur ein Ausschnitt der Textur auf das Rechteck gemappt (z.B. Texturkoordinaten $(0,0)$, $(\frac{a_t}{a_p},0)$, $(\frac{a_t}{a_p},1)$ und $(0,1)$ für $a_t < a_p$ bzw. $(0,0)$, $(1,0)$, $(1,\frac{a_p}{a_t})$ und $(0,\frac{a_p}{a_t})$ für $a_t > a_p$) oder die Textur wird vollständig aufgebracht und je nach Texturfortsetzungsmodus z.B. wiederholt (Texturkoordinaten $(0,0)$, $(1,0)$, $(1,\frac{a_p}{a_t})$ und $(0,\frac{a_p}{a_t})$ für $a_t < a_p$ bzw. $(0,0)$, $(\frac{a_t}{a_p},0)$, $(\frac{a_t}{a_p},1)$ und $(0,1)$ für $a_t > a_p$). In Bild 13.10 sind konkrete Beispiele dargestellt.

Die Manteloberfläche eines Zylinders ist wegen ihrer ebenen Abwicklung als Rechteck genauso wie gerade beschrieben zu behandeln. Man legt den Zylinder mit der Höhe h und dem Radius r in die z-Achse eines Zylinderkoordinatensystems (z, r, φ), wobei $\varphi \in [0, 2\pi]$. Der Mantelboden liegt in der xy-Ebene und wird durch die kartesischen Koordinaten $(x = r \cdot \cos(\varphi), y = r \cdot \sin(\varphi), z = 0)$ beschrieben. Der obere Deckel des Zylindermantels liegt parallel zum Mantelboden in der Höhe $z = h$. Angenommen, das Höhen-Umfangs-Verhältnis des Zylindermantels sei gleich dem Höhen-Breiten-Verhältnis der Textur, dann kann der Programmcode zur Approximation des texturierten Zylindermantels z.B. durch 16 Rechtecke folgendermaßen aussehen:

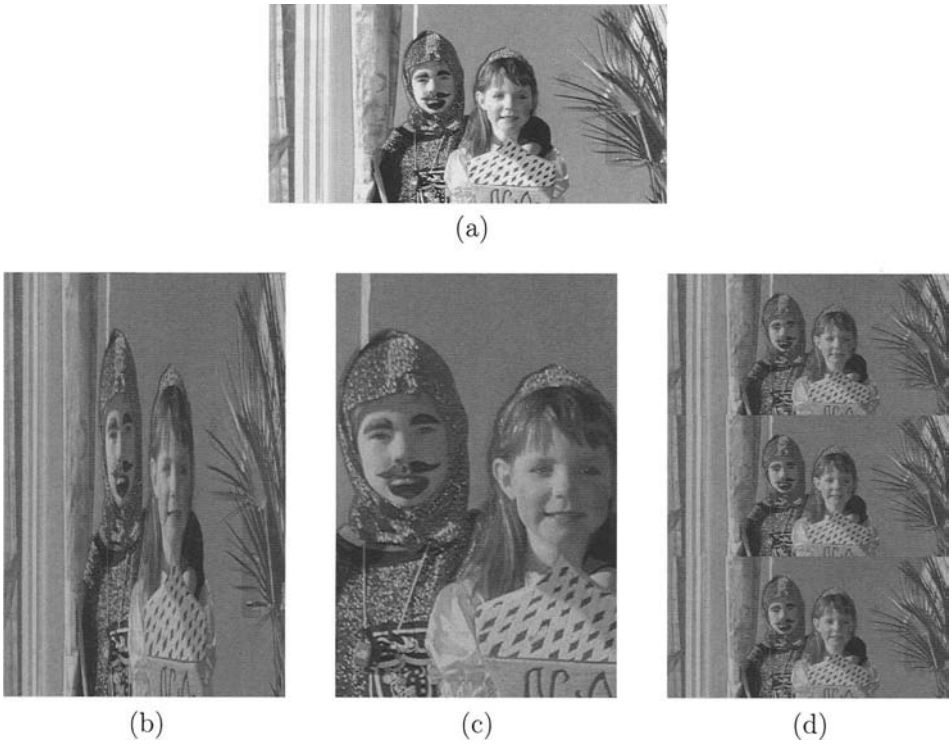


Bild 13.10: Texture Mapping mit und ohne Verzerrung: (a) Textur mit einem Höhen-Breiten-Verhältnis von $a_t = \frac{1}{2}$. (b) Mapping dieser Textur auf ein rechteckiges Polygon mit einem Höhen-Breiten-Verhältnis von $a_p = \frac{3}{2}$ führt zu Verzerrungen. (c) Verzerrungsfreies Mapping eines Texturausschnitts. (d) Verzerrungsfreies Mapping der gesamten Textur mit dem Fortsetzungsmodus `GL_REPEAT`.

```
glBegin(GL_QUAD_STRIP);
    for(phi = 0; phi < 2 * 3.1416; phi += 3.1415/8)
    {
        x = cos(phi);
        y = sin(phi);
        s = phi/(2*3.1415);
        glNormal3f(x,y,0); glTexCoord2f(s,0); glVertex3f(r*x,r*y,0);
        glNormal3f(x,y,0); glTexCoord2f(s,1); glVertex3f(r*x,r*y,h);
    }
glEnd();
```

Das Abbruchkriterium in der for-Schleife ist bewusst an der letzten Stelle um eins erhöht (3.1416 anstatt 3.1415), damit der Mantel geschlossen wird. Aufgrund der numerischen Ungenauigkeiten wäre es durchaus sinnvoll, die letzten beiden Vertices des Quadstrips

identisch zu den ersten beiden Vertices zu setzen. Ein Beispiel für einen texturierten Zylindermantel ist in Bild 13.1 zu sehen. Leider gibt es nur sehr wenige gekrümmte Oberflächen, die eine Abwicklung in eine ebene Fläche besitzen, wie z.B. Zylinder oder Kegel. Bei allen anderen Oberflächenformen tritt immer eine gewisse Verzerrung der Textur auf. Je höher die Krümmung der Oberfläche, desto stärker die Verzerrung.

Dies wird beim Texture Mapping auf eine Kugeloberfläche deutlich. Das Zentrum der Kugel mit Radius r befinde sich im Ursprung eines Kugelkoordinatensystems (r, φ, ϑ), wobei $\varphi \in [0, 2\pi]$ und $\vartheta \in [0, \pi]$. Ein Punkt auf der Kugeloberfläche wird durch die kartesischen Koordinaten $(x = r \cdot \cos(\varphi) \cdot \cos(\vartheta), y = r \cdot \sin(\varphi) \cdot \cos(\vartheta), z = r \cdot \sin(\vartheta))$ beschrieben. Der Programmcode zur Approximation einer texturierten Kugeloberfläche durch z.B. $2048 (= 32 \cdot 64)$ Rechtecke kann (ohne Optimierungen) folgendermaßen aussehen:

```
for(theta = 3.1415/2; theta > -3.1416/2; theta -= 3.1415/32)
{
    thetaN = theta - 3.1415/32;
    glBegin(GL_QUAD_STRIP);
        for(phi = 0; phi < 2 * 3.1416; phi += 3.1415/32)
        {
            x = cos(phi) * cos(theta);
            y = sin(phi) * cos(theta);
            z = sin(theta);
            s = phi/(2 * 3.1415);
            t = theta/3.1415 + 0.5;
            glNormal3f(x,y,z); glTexCoord2f(s,t); glVertex3f(r*x,r*y,r*z);

            x = cos(phi) * cos(thetaN);
            y = sin(phi) * cos(thetaN);
            z = sin(thetaN);
            t = thetaN/3.1415 + 0.5;
            glNormal3f(x,y,z); glTexCoord2f(s,t); glVertex3f(r*x,r*y,r*z);
        }
    glEnd();
}
```

In diesem Programmbeispiel wird die Textur genau einmal auf die Kugeloberfläche gemappt, was zu immer stärkeren Verzerrungen führt, je näher man den beiden Polen kommt (Bild 13.11-a). Die gesamte oberste Zeile der Textur wird auf einen Punkt, den Nordpol, gemappt, und die unterste Zeile der Textur auf den Südpol. Wird die Textur dagegen nur auf einen Teil der Kugeloberfläche gemappt, indem die Texturkoordinaten entsprechend skaliert werden, fällt die Verzerrung sehr viel geringer aus (Bild 13.11-b). Denn in diesem Fall ist die Krümmung der Oberfläche im Verhältnis zur gesamten Textur sehr viel geringer als beim Mapping der Textur auf die gesamte Kugeloberfläche.

(a) (b)

Bild 13.11: Texture Mapping auf eine Kugeloberfläche: (a) Die Textur wird genau einmal vollständig auf die gesamte Kugeloberfläche gemappt. Dies führt zu immer stärkeren Verzerrungen, je näher man den beiden Polen kommt. (b) Die Textur wird nur auf einen Ausschnitt der Kugeloberfläche gemappt. Die Verzerrungen sind verhältnismäßig gering.

Für die bisherigen geometrischen Objekte, Ebene, Zylinder und Kugel ist es relativ leicht eine geeignete Textur-Abbildung zu finden, da die Oberflächen analytisch beschreibbar sind. Wie ordnet man aber den Vertices die Texturkoordinaten zu, wenn man ein völlig frei geformtes Objekt texturieren will? Eine arbeitsintensive Methode, die auch künstlerisches Geschick erfordert, ist es, das Objekt so zu zerlegen, dass die Teile einigermaßen planar sind. Auf diese Teile kann man dann ohne große Verzerrungen entsprechende Texturausschnitte mappen. Außerdem gibt es Werkzeuge, die es erlauben, das Polygonnetz der Oberfläche durch eine Rückprojektion in die Texturebene zu transformieren, in der dann per Hand die Vertices an die Textur angepasst werden können. Eine andere Methode, die oft ausreicht und für den Designer weniger Aufwand bedeutet, ist die zweistufige Textur-Abbildung. In der ersten Stufe („S-Mapping“ genannt) wählt man eine analytisch beschreibbare Zwischenoberfläche, wie z.B. eine Ebene, einen Würfel, einen Zylinder oder eine Kugel, das dem eigentlichen Objekt ähnlich ist und es umhüllt. Auf diese Zwischenoberfläche werden nach den oben beschriebenen Verfahren die Texturkoordinaten abgebildet. In einer zweiten Stufe („O-Mapping“ genannt) werden die Texturkoordinaten der Zwischenoberfläche auf das eigentliche Objekt abgebildet. Für das „O-Mapping“ werden in der Regel vier Möglichkeiten angeboten, um die Texturkoordinaten auf der Zwischenoberfläche zu bestimmen:

- Ausgehend vom Objektschwerpunkt wird ein Strahl durch einen Vertex der Objektoberfläche geschickt und bis zum Schnittpunkt mit der umhüllenden Zwischenoberfläche verlängert (Bild 13.12-a).

- Der Normalenvektor am Vertex der Objektoberfläche wird benutzt, um den Schnittpunkt mit der umhüllenden Zwischenoberfläche zu bestimmen (Bild 13.12-b).
- Der Ausgangspunkt des Normalenvektors der Zwischenoberfläche, welcher den Vertex der Objektoberfläche trifft, wird benutzt. Ist die Zwischenoberfläche eine Ebene, entspricht dies einer projektiven Textur (Abschnitt 13.3), da die Textur quasi wie mit einem Diaprojektor auf das Objekt gemappt wird (Bild 13.12-c).
- Ein Strahl, der vom Augenpunkt ausgeht und am Vertex der Objektoberfläche ideal reflektiert wird, dient zur Bestimmung des Schnittpunkts mit der Zwischenoberfläche. Dieses Verfahren ist prinzipiell das gleiche wie bei den Umgebungs-Texturen (Abschnitt 13.4). Der Unterschied besteht nur darin, dass der Inhalt einer Umgebungs-Textur, wie der Name schon sagt, aus der Umgebung besteht, die man aus der Objektposition sieht, während eine normale Textur die Detailstruktur der Objektoberfläche enthält (Bild 13.12-d).

Eine ausführliche Darstellung der mathematischen Methoden dieser zweistufigen Textur-Abbildungen findet man in [Watt02].

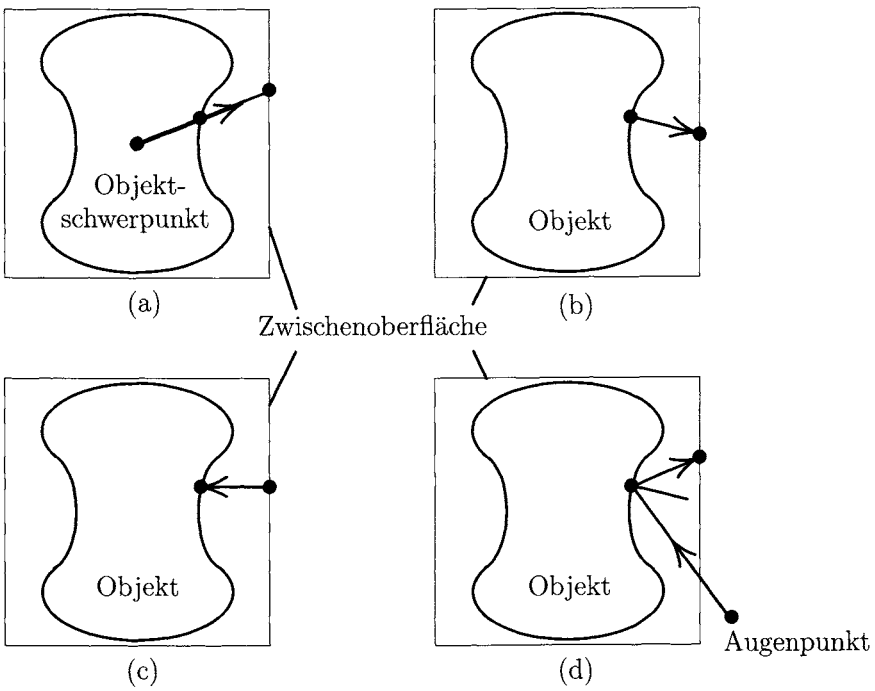


Bild 13.12: Die vier Varianten des „O-Mappings“, mit denen man die Texturkoordinaten von der Zwischenoberfläche (hier ein Kubus im Schnitt) auf die Objektoberfläche abbildet: (a) Objektschwerpunkt. (b) Normalenvektor des Objekts. (c) Normalenvektor der Zwischenoberfläche. (d) Von der Objektoberfläche reflektierter Augenpunktsvektor.

13.1.6.3 Automatische Generierung von Texturkoordinaten

Die zweite Möglichkeit, Texturkoordinaten zu definieren, ist die automatische Generierung mit Hilfe von Gleichungen, die mit dem OpenGL-Befehl `glTexGen*()` festgelegt werden. Für jede Texturdimension (s, t, r, q) kann eine Gleichung festgelegt werden, in die die Vertexkoordinaten (x, y, z, w) einfließen und die das Ergebnis eben diesem Vertex als Texturkoordinate zuweist. Der `glTexGen*()`-Befehl existiert in den Varianten, die in der folgenden Tabelle dargestellt sind:

Skalar-Form
`glTexGenf(GLenum` *coord*`, GLenum` *name*`, GLfloat` *param*`)`
`glTexGend(GLenum` *coord*`, GLenum` *name*`, GLdouble` *param*`)`
`glTexGeni(GLenum` *coord*`, GLenum` *name*`, GLint` *param*`)`
Vektor-Form
`glTexGenfv(GLenum` *coord*`, GLenum` *name*`, GLfloat *`*param*`)`
`glTexGendv(GLenum` *coord*`, GLenum` *name*`, GLdouble *`*param*`)`
`glTexGeniv(GLenum` *coord*`, GLenum` *name*`, GLint *`*param*`)`

Der erste Parameter des `glTexGen`-Befehls, *coord*, legt die Texturdimension (s, t, r, q) fest, um die es geht. Die zulässigen Parameter für *coord* sind daher `GL_S`, `GL_T`, `GL_R` und `GL_Q`. Falls der zweite Parameter, `name`, die Werte (`GL_TEXTURE_OBJECT_PLANE`) oder (`GL_TEXTURE_EYE_PLANE`) annimmt, kann man die Parameter (a, b, c, d) einer Ebenengleichung festlegen. Dazu übergibt man mit *param* einen Vektor, dessen vier Komponenten die Parameter enthalten. Wenn aber der zweite Parameter, `name`, den Wert (`GL_TEXTURE_GEN_MODE`) annimmt, wird der Generierungsmodus mit dem dritten Parameter, `param`, festgelegt. Derzeit bietet OpenGL die folgenden Generierungsmodi an:

- `GL_OBJECT_LINEAR`:
  bewirkt die Erzeugung der Texturkoordinate mit Hilfe der folgenden Ebenengleichung, in die die Vertices in Objektkoordinaten (x_o, y_o, z_o, w_o) einfließen:

$$s(x_o, y_o, z_o, w_o) = a \cdot x_o + b \cdot y_o + c \cdot z_o + d \cdot w_o \tag{13.11}$$

  Davon unabhängig davon können für die anderen drei Texturkoordinaten (t, r, q) entsprechende Ebenengleichungen festgelegt werden. Dadurch dass die Texturkoordinaten im ursprünglichen Objektkoordinatensystem definiert wurden, bleibt die Textur unverrückbar mit dem Objekt verbunden, auch wenn das Objekt später durch Modell- und Augenpunktstransformationen beliebig bewegt wird.

- `GL_EYE_LINEAR`:
  bewirkt die Erzeugung der Texturkoordinate mit Hilfe der folgenden Ebenengleichung

chung, in die die Vertices in Weltkoordinaten (auch Augenpunktskoordinaten genannt) (x_e, y_e, z_e, w_e) einfließen:

$$s(x_e, y_e, z_e, w_e) = a' \cdot x_e + b' \cdot y_e + c' \cdot z_e + d' \cdot w_e \quad \text{wobei} \quad (a', b', c', d') = (a, b, c, d) \cdot \mathbf{M^{-1}} \tag{13.12}$$

Die übergebenen Parameter der Ebenengleichung (a, b, c, d) werden vorher mit der inversen Modell- und Augenpunkts-Matrix $\mathbf{M^{-1}}$ multipliziert, damit die projizierte Textur fest mit dem Augenpunkt verbunden bleibt. In diesem Fall „schwimmen" die Objekte quasi unter der Textur hindurch, wie bei einem Diaprojektor, vor dessen Linse sich Objekte bewegen. Entsprechende Gleichungen können für die anderen drei Texturkoordinaten (t, r, q) festgelegt werden.

- `GL_SPHERE_MAP`:
  bewirkt eine sphärische Projektion bezüglich der beiden Texturkoordinaten (s, t) einer 2-dimensionalen Textur. Diese Projektion dient zur Darstellung von Spiegelungen mit Hilfe von Umgebungstexturen und wird im Abschnitt 13.4 besprochen.

- `GL_REFLECTION_MAP`:
  benutzt die gleichen Berechnungen wie `GL_SPHERE_MAP` bezüglich der drei Texturkoordinaten (s, t, r) einer Cube Map Textur (Abschnitt 13.4) und liefert bessere Ergebnisse bei Spiegelungen auf Kosten der Rechenzeit.

- `GL_NORMAL_MAP`:
  in diesem Fall werden die Normalenvektoren mit Hilfe der invers transponierten Modell- und Augenpunkts-Matrix $\mathbf{M^{-1^T}}$ vom Modellkoordinatensystem ins Weltkoordinatensystem transformiert. Die resultierenden Komponenten der Normalenvektoren (n_x, n_y, n_z) werden als Texturkoordinaten (s, t, r) benutzt, um eine Cube Map Textur abzutasten (man vergleiche dies auch mit Bild 13.12).

Im Folgenden werden die beiden Modi `GL_OBJECT_LINEAR` und `GL_EYE_LINEAR` anhand eines konkreten Beispiels erläutert. Angenommen, die vier Parameter der Ebenengleichung zur automatischen Berechnung der s-Texturkoordinate sind $(0.1, 0.0, 0.0, 0.5)$, dann gilt (mit $w_o = 1$):

$$s(x_o, y_o, z_o, w_o) = 0.1 \cdot x_o + 0.5 \tag{13.13}$$

Die Parameter für die Ebenengleichung der t-Texturkoordinate seien $(0.0, 0.1, 0.0, 0.5)$, dann gilt (mit $w_o = 1$):

$$t(x_o, y_o, z_o, w_o) = 0.1 \cdot y_o + 0.5 \tag{13.14}$$

Der entsprechende Programmcode zur Festlegung der Ebenengleichungen, der üblicherweise in einer Initialisierungsroutine steht, lautet:

```
GLfloat s_plane[] = (0.1, 0.0, 0.0, 0.5);
GLfloat t_plane[] = (0.0, 0.1, 0.0, 0.5);

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glTexGenfv(GL_S, GL_TEXTURE_OBJECT_PLANE, s_plane);
glTexGenfv(GL_T, GL_TEXTURE_OBJECT_PLANE, t_plane);

glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

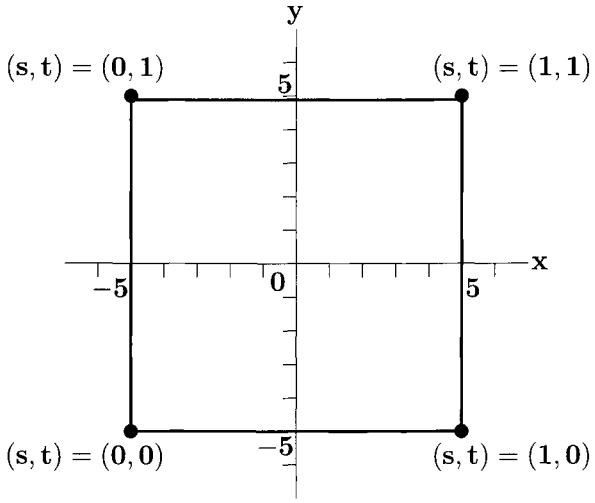


Bild 13.13: Das Prinzip der orthografischen Projektion von Texturkoordinaten.

Trägt man die Texturkoordinaten gemäß (13.13) und (13.14) im x, y-Koordinatensystem eines Objekts auf (Bild 13.13), so wird offensichtlich, dass die Textur durch eine orthografische Projektion (Kapitel 7) auf die x, y-Ebene eines Objekts gemappt wird. Bei den gegebenen Parametern wird die Textur dabei im lokalen Koordinatensystem des Objekts auf einen rechteckigen Bereich von $x \in [-5, +5], y \in [-5, +5]$ projiziert. Die Parameter der Ebenengleichungen wirken dabei wie inverse Skalierungsfaktoren. Angenommen, die Parameter für die Ebenengleichung der s-Texturkoordinate sind $(0.07, 0.0, 0.07, 0.5)$, so dass gilt $s(x_o, y_o, z_o, w_o) = 0.07 \cdot x_o + 0.07 \cdot z_o + 0.5$, erhält man eine orthografische Projektion der Textur auf eine um 45° bzgl. der y-Achse gedrehte Ebene. Durch eine geeignete Vorgabe der Parameter für die Ebenengleichungen kann man orthografische Projektionen von Texturen über beliebig positionierte, gedrehte, skalierte und sogar gescherte Ebenen im Raum definieren. Da die Texturkoordinaten im Generierungsmodus `GL_OBJECT_LINEAR` im *lokalen Koordinatensystem des Objekts* definiert werden, haftet die Textur auch bei Bewegungen des Objekts immer fest an der Oberfläche (Bild 13.14-a bis -f). Je nach Form der Objekte und Perspektive des Augenpunkts wird die Textur mehr oder weniger verzerrt auf das Objekt abgebildet.

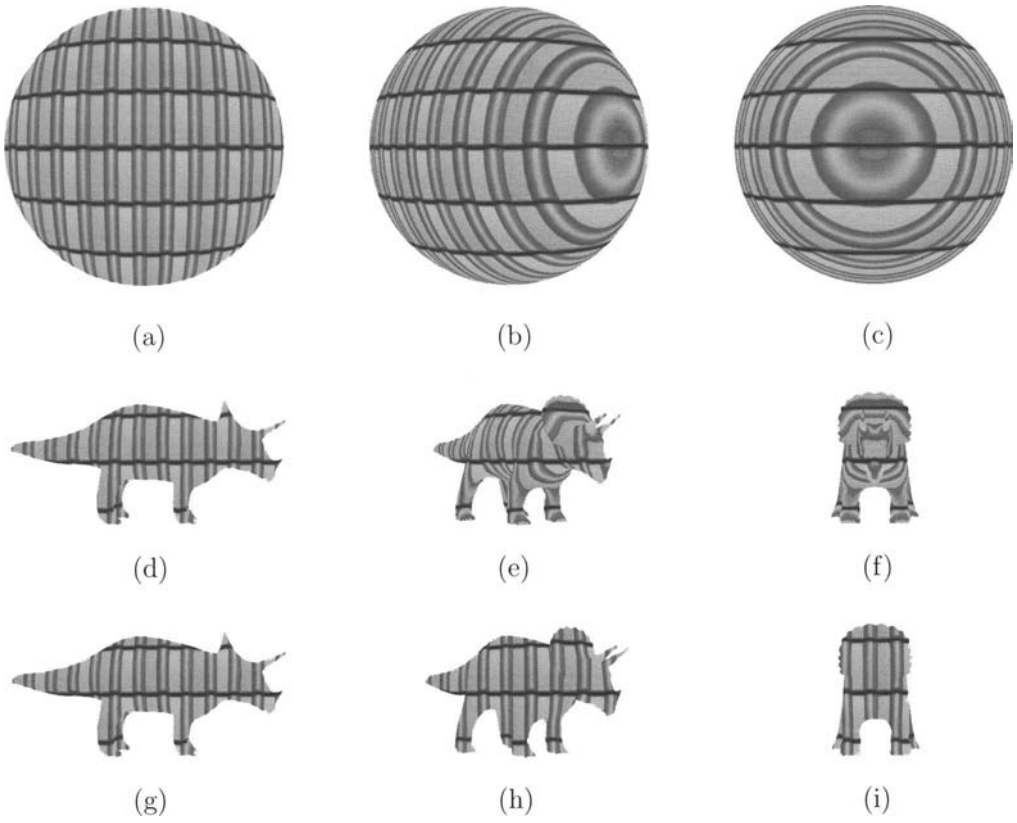


Bild 13.14: Automatische Generierung von Texturkoordinaten: Das Objekt ist von links nach rechts jeweils um 45° bzgl. der y-Achse gedreht. Obere Reihe (a, b, c) Generierungsmodus `GL_OBJECT_LINEAR` beim Objekt Kugel. Die Textur „klebt" fest auf der Kugel. Mittlere Reihe (d, e, f) Generierungsmodus `GL_OBJECT_LINEAR` beim Objekt Triceratops. Die Textur „klebt" fest auf dem Triceratops. Untere Reihe (g, h, i) Generierungsmodus `GL_EYE_LINEAR` beim Objekt Triceratops. Der Triceratops „schwimmt" unter der fest stehenden Textur hinweg.

Mit dem folgenden Programmcode definiert man entsprechende Ebenengleichungen im Generierungsmodus `GL_EYE_LINEAR`:

```
GLfloat s_plane[] = (0.1, 0.0, 0.0, 0.5);
GLfloat t_plane[] = (0.0, 0.1, 0.0, 0.5);

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
glTexGenfv(GL_S, GL_TEXTURE_EYE_PLANE, s_plane);
glTexGenfv(GL_T, GL_TEXTURE_EYE_PLANE, t_plane);
```

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

In diesem Fall werden die Texturkoordinaten im *Koordinatensystem des Augenpunkts* definiert. Daher bleibt die Textur starr an den Augenpunkt gebunden (Bild 13.14-g bis -i). Bewegte Objekte „schwimmen" quasi unter der Textur hindurch, wie bei einem Diaprojektor, vor dessen Linse sich Objekte bewegen.

13.1.6.4 Textur Matrizen Stapel

Texturkoordinaten werden vor dem Texture Mapping mit einer 4 x 4-Matrix multipliziert. Dies geschieht nach den gleichen Prinzipien wie die Transformation von Vertexkoordinaten durch Modell- und Projektionstransformationen (Kapitel 7). Ohne einen expliziten Aufruf ist die Textur-Matrix die Einheitsmatrix. Um die Textur-Matrix zu verändern, wird der Matrizenmodus auf `GL_TEXTURE` gesetzt:

```
glMatrixMode(GL_TEXTURE);
glTranslatef(...);
glRotatef(...);
glScalef(...);
glMatrixMode(GL_MODELVIEW);
```

Durch Veränderung der Textur-Matrix für jedes neue Bild kann man interessante Animationen erzeugen, wie z.B. eine Textur über eine Oberfläche gleiten lassen, sie drehen oder skalieren. Außerdem ist auch möglich, perspektivische Projektionstransformationen mit den homogenen Texturkoordinaten durchzuführen (Abschnitt 13.3). Genauso wie alle anderen Matrizen können auch Textur-Matrizen in einem "Textur-Matrizen-Stapel" gespeichert und bei Bedarf wieder hervorgeholt werden. Jede OpenGL-Implementierung muss einen Textur-Matrizen-Stapel zur Verfügung stellen, der mindestens zwei Textur-Matrizen aufnehmen kann. Operationen auf dem Textur-Matrizen-Stapel können mit den üblichen Matrizen-Operatoren wie z.B. `glPushMatrix()`, `glPopMatrix()`, `glMultMatrix()` vorgenommen werden.

13.1.7 Einschalten des Texture Mappings

Damit die Algorithmen für das Texture Mapping überhaupt aufgerufen werden, muss OpenGL in den entsprechenden Zustand gesetzt werden. Dazu dient der Befehl:

```
glEnable(GL_TEXTURE_2D);
```

Ein Aufruf dieses Befehls schließt die Aktivierung von 1-dimensionalen Texturen mit ein. Wird die Berechnung von 3-dimensionalen Texturen aktiviert (`glEnable(GL_TEXTURE_3D)`), schließt dies 2- und 1-dimensionale Texturen mit ein. Wird die Berechnung von Cube Map Texturen aktiviert (`glEnable(GL_TEXTURE_CUBE_MAP)`, Abschnitt 13.4), schließt dies 3-, 2-

und 1-dimensionale Texturen mit ein. Deaktivierung des jeweiligen Texture Mappings läuft wie bei allen anderen Zuständen auch über den `glDisable()`-Befehl.

Die automatische Generierung von Texturkoordinaten (Abschnitt 13.1.6) muss für jede Texturdimension separat aktiviert werden. Dazu dienen die Befehle:

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

Für 3-dimensionale Texturen gibt es noch den Parameter `GL_TEXTURE_GEN_R` und für projektive Texturen kann auch noch die vierte Texturkoordinate `GL_TEXTURE_GEN_Q` automatisch generiert werden.

13.1.8 Textur-Objekte

Beim Rendern komplexerer Szenen benötigt man immer eine Vielzahl verschiedener Texturen. Da in einem OpenGL-Zustand zu einem bestimmten Zeitpunkt aber immer nur eine einzige Textur definiert sein kann, sind zeitaufwändige Zustandswechsel in solchen Fällen unvermeidlich. OpenGL bietet jedoch neben dem `glTexSubImage2D()`-Befehl zum Ersetzen eines Texturausschnitts (Abschnitt 13.1.1.1) auch noch die Möglichkeit, sogenannte „*Textur-Objekte*“ zu definieren, die einen ganzen Satz an Texturen enthalten können. Dadurch muss beim Wechsel einer Textur diese nicht jedesmal neu spezifiziert und in den Texturspeicher geladen werden, wie beim `glTexImage2D()`-Befehl. Falls der Texturspeicher ausreicht, kann der ganze Satz an Texturen mitsamt aller Attribute dort gehalten werden, was den Aufwand beim Zustandswechsel zwischen zwei Texturen gering hält. Für große visuelle Simulationen oder aufwändige Spiele reicht der verfügbare Texturspeicher jedoch nie aus, um alle Texturen gleichzeitig im Texturspeicher zu halten. Einige OpenGL-Implementierungen unterstützen deshalb Textur-Nachladestrategien, bei denen häufig genutzte oder in Kürze benötigte Texturen eine hohe Priorität zugewiesen bekommen können, so dass sie mit großer Wahrscheinlichkeit im Texturspeicher sind, wenn sie gebraucht werden.

Um Textur-Objekte zu nutzen, sind folgende Schritte durchzuführen:

- Generierung von Texturnamen mit dem Befehl:
  `glGenTexture(GLsizei n, GLuint *texName)`. Dadurch werden n Texturnamen im Array `*texName` für eine entsprechende Anzahl an Texturen reserviert.

- Erzeugung eines Textur-Objekts während der Initialisierung durch Anbindung von Textureigenschaften und Texturbilddaten. Dazu dient der Befehl: `glBindTexture(GL_TEXTURE_2D, texName[n])`, sowie eine Folge von `glTexParameteri`-Befehlen zur Festlegung der Texturattribute (Textur-Filter, Textur-Fortsetzungsmodus etc.) und der `glTexImage2D`-Befehl zur Spezifikation und zum Laden der Textur.

- Festlegung von Prioritäten für die Textur-Objekte, falls dies die OpenGL-Implementierung unterstützt. Dies wird mit dem Befehl:

`glPrioritizeTextures(GLsizei n, GLuint *texName, GLfloat *priorities)` festgelegt. Für jedes der *n* Textur-Objekte wird in dem Array **priorities* ein Prioritätswert aus dem Wertebereich $[0, 1]$ festgelegt und dem jeweiligen Textur-Objekt aus dem Array **texName* zugeordnet.

- Wiederholtes Anbinden verschiedener Textur-Objekte während der Laufzeit durch erneuten Aufruf des `glBindTexture`-Befehls.
- Löschen nicht mehr benötigter Textur-Objekte mit Hilfe des Befehls: `glDeleteTextures(GLsizei n, GLuint *texName)`. Dadurch werden *n* Textur-Objekte gelöscht, deren Namen in dem Array **texName* übergeben werden.

Zur Veranschaulichung dieser Schritte werden im Folgenden die Programmergänzungen dargestellt, die für die Nutzung von Textur-Objekten notwendig sind:

```
GLint width = 512, height = 256;
static GLubyte firstImage[width][height][4];
static GLubyte secondImage[width][height][4];
static GLuint texName[2];
GLfloat priorities[] = (0.5, 0.9);

/*** INITIALISIERUNGS-FUNKTION ***/
glGenTexture(2, texName);
glBindTexture(GL_TEXTURE_2D, texName[0]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage2D(   GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0
                GL_RGBA, GL_UNSIGNED_BYTE, firstImage);

glBindTexture(GL_TEXTURE_2D, texName[1]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexImage2D(   GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0
                GL_RGBA, GL_UNSIGNED_BYTE, secondImage);

/*** DISPLAY-FUNKTION ***/
glPrioritizeTextures(2, texName, priorities);
glBindTexture(GL_TEXTURE_2D, texName[0]);
draw_first_objects();
glBindTexture(GL_TEXTURE_2D, texName[1]);
draw_second_objects();
```

13.2 Mehrfach-Texturierung (Multitexturing)

Bisher wurde eine Fläche immer nur mit einer einzigen Textur überzogen. Sehr viele interessante neue Möglichkeiten ergeben sich, wenn man in einem einzigen Durchlauf durch die Rendering Pipeline (*single pass rendering*) mehrere Texturen auf einer Fläche überlagern kann. So lassen sich z.B. komplexe Beleuchtungsverhältnisse in Räumen mit aufwändigen *Radiosity*-Vefahren berechnen und in einer Textur speichern, deren Texel eine bestimmte Intensität des Lichteinfalls repräsentieren (eine sogenannte „*light map*").

(a) (b) (c)

Bild 13.15: Mehrfach-Texturierung (Multitexturing): (a) Die erste Textur ist eine sogenannte *light map*, die die Intensitätsverteilung in einem Lichtkegel darstellt. (b) Die zweite Textur ist eine gewöhnliche RGB-Textur, das Foto einer Ziegelmauer. (c) Die Kombination der beiden Texturen erweckt den Eindruck, dass jemand mit einer Taschenlampe die Ziegelmauer beleuchtet. Man beachte, dass die Geometrie der Szene nur aus vier Vertices besteht, die zu einem Rechteck verbunden sind.

In Bild 13.15-a ist ein einfaches Beispiel für eine *light map* dargestellt, die z.B. den Lichtkegel einer Taschenlampe simulieren kann. Dieser Lichtkegel kann nun auf eine bereits texturierte Oberfläche treffen und dort die Helligkeit modulieren (Bild 13.15-b,c). Mathematisch gesehen müssen die beiden Textur-Fragmentfarben nur miteinander kombiniert (d.h. multipliziert, addiert, interpoliert etc.) werden, um den gewünschten Effekt zu erzielen. Mit diesem Verfahren kann man trotz grober Tesselierung von Objekten und normalem Gouraud-Shading ähnlich gute Ergebnisse erzielen wie mit echtem Phong-Shading. Viele fortgeschrittene Texture Mapping Verfahren beruhen auf Mehrfach-Texturierung.

OpenGL bietet seit der Version 1.2 Mehrfach-Texturierung an, und bei vielen Grafikkarten ist dies heutzutage Standard. Allerdings benötigt man für die Programmierung den Mechanismus der OpenGL-Extensions (hier `GL_ARB_multitexture`), der doch um Einiges aufwändiger ist, als die Benutzung normaler OpenGL-Befehle. Aus Platzgründen wird hier nur das Prinzip dargestellt, die Programmierdetails sind auf der Webseite zu diesem Buch zu finden.

Das Verfahren der Mehrfach-Texturierung läuft wieder nach dem Pipeline-Prinzip ab (Bild 13.16): es gibt eine Textur-Kaskade, die aus mehreren Textureinheiten besteht. Jede Textureinheit führt die in Abschnitt 13.1 dargestellten Texturoperationen aus und liefert ihr Ergebnis (eine Texturfarbe) an die nächste Stufe in der Textur-Kaskade, bis die oberste Stufe erreicht ist und der endgültige Farbwert des Fragments zur weiteren Bearbeitung (Nebelberechnung, Alpha-Blending etc.) ausgegeben wird.

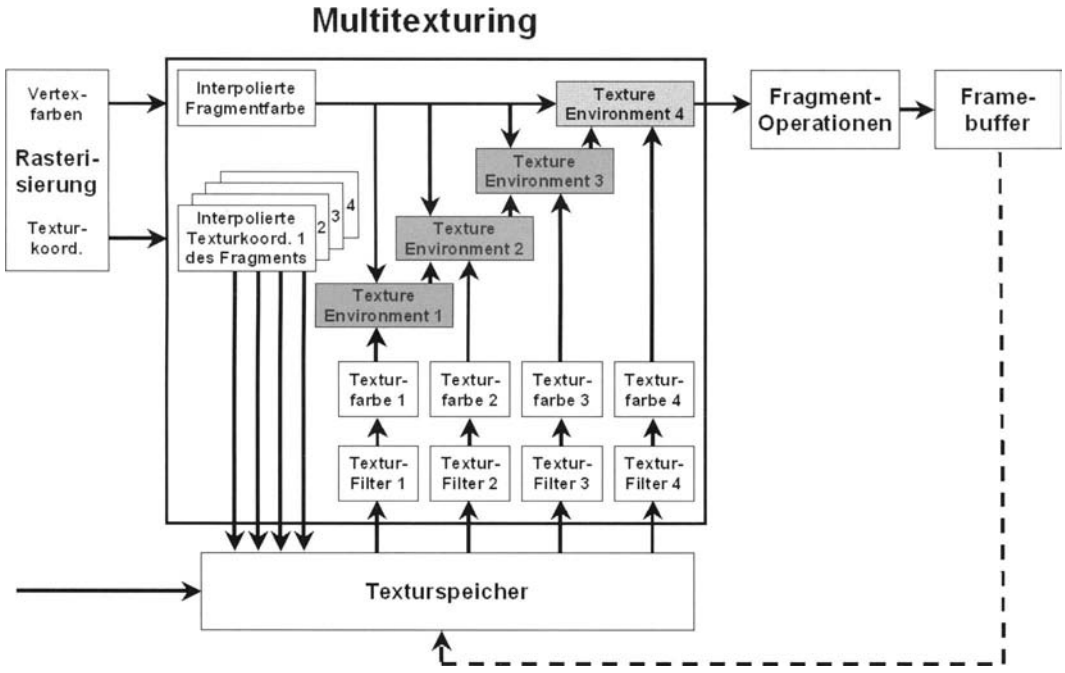


Bild 13.16: Die Rendering Pipeline für die Mehrfach-Texturierung (Multitexturing): in diesem Bild sind vier unabhängige Textureinheiten dargestellt, die sequentiell arbeiten. Man vergleiche auch mit den Bildern (5.1), (13.2) und (12.21).

Zur Nutzung der Mehrfach-Texturierung in OpenGL sind folgende Schritte durchzuführen:

- Abfrage, ob die verwendete OpenGL-Implementierung und die Grafikkarte die Extension `GL_ARB_multitexture` unterstützen (Befehl `glGetString(GL_EXTENSIONS)`) und wie viele Texturen in einem *Rendering Pass* aufgebracht werden können (Befehl `glGetIntegerv(GL_MAX_TEXTURE_UNITS, &maxTexUnits)`).

- Aktivierung der OpenGL-Befehle für Mehrfach-Texturierung: `glActiveTexture()`, `glMultiTexCoord*()`.

- Spezifikation mehrerer Texturen incl. statischer Texturattribute (z.B. Textur-Filter, Textur-Fortsetzungsmodus), wie in Abschnitt 13.1.8 dargestellt.
- Festlegung der aktiven Texturen, die in Kombination auf die Polygone gemappt werden sollen (Befehl `glActiveTexture()`).
- Festlegung aller dynamischen Attribute einer Textureinheit, wie z.B.:
  - Festlegung, wie die Texturfarben der verschiedenen Textureinheiten und die Beleuchtungsfarbe miteinander gemischt werden sollen (Texture Environment). Dazu dient der bereits vorgestellte `glTexEnv*()`-Befehl mit den vielfältigen Möglichkeiten der Texture Combiner Funktionen, die hier aus Platzgründen nicht dargestellt werden, sondern in der OpenGL Spezifikation [Sega04] und im OpenGL Programming Guide [Shre05] zu finden sind.
  - Festlegung der Textur-Matrix (Abschnitt 13.1.6.4).
- Festlegung der Texturkoordinaten für jede einzelne Textur. Dafür gibt es wieder wie üblich die beiden Möglichkeiten, die Texturkoordinaten explizit den Vertices zuzuordnen (Befehl `glMultiTexCoord*()`) oder sie automatisch mit Hilfe von Gleichungen erzeugen zu lassen (Abschnitt 13.1.6.3, Befehl `glTexGen*()`).

Zur Veranschaulichung der letzten drei Schritte werden im Folgenden die Programmergänzungen dargestellt, die für die Nutzung der Mehrfach-Texturierung notwendig sind:

```
/*** DISPLAY-FUNKTION ***/
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texName[0]);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_MODULATE);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glTranslatef(...);
glRotatef(...);
glScalef(...);
glFrustum(...);
glMatrixMode(GL_MODELVIEW);

glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, texName[1]);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf (GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_ADD_SIGNED_EXT);
```

```
glBegin(GL_QUADS);
    glNormal3fv(n0, n1, n2 );
    glMultiTexCoord2fvARB(GL_TEXTURE1_ARB, t0 ); glVertex3fv( v0 );
    glMultiTexCoord2fvARB(GL_TEXTURE1_ARB, t1 ); glVertex3fv( v1 );
    glMultiTexCoord2fvARB(GL_TEXTURE1_ARB, t2 ); glVertex3fv( v2 );
    glMultiTexCoord2fvARB(GL_TEXTURE1_ARB, t3 ); glVertex3fv( v3 );
glEnd();
```

Im ersten Abschnitt dieses Programmierbeispiels wird die Textureinheit Nr. 0 aktiviert und das Textur-Objekt `texName[0]` angebunden. Anschließend wird das Texture Environment festgelegt, bei dem hier die Texturfarbe mit der interpolierten Farbe aus der Beleuchtungsrechnung multipliziert wird. Die Texturkoordinaten für Textureinheit Nr. 0 werden automatisch generiert und durch eine Textur-Matrix modifiziert. Im zweiten Abschnitt wird die Textureinheit Nr. 1 aktiviert und das Textur-Objekt `texName[1]` angebunden. Anschließend wird im Texture Environment festgelegt, dass die Texturfarbe Nr. 1 und die Texturfarbe Nr. 0 komponentenweise addiert werden und davon jeweils der mittlere Grauwert 0.5 subtrahiert wird. Im dritten Abschnitt werden die Texturkoordinaten für die Textureinheit Nr. 1 explizit mit dem Befehl `glMultiTexCoord2fvARB(GL_TEXTURE1_ARB, t* )` den Vertices zugewiesen. Das Ergebnis ist in Bild 13.15-c zu sehen.

Mit der Mehrfach-Texturierung lassen sich eine ganze Reihe interessanter Bildverarbeitungsverfahren in Echtzeit auf einer Grafikkarte realisieren. Typische Anwendungsbeispiele dafür sind die Summen-, Differenz- und Ratiobildung bei mehrkanaligen Bildsequenzen (Kapitel 25). Durch die programmierbaren Pixel-Shader auf den neuesten Grafikkarten werden diese Möglichkeiten noch einmal erheblich erweitert. Damit ist es möglich, bis zu 16 Texturen mit einer großen Vielfalt an mathematischen Operationen miteinander zu verknüpfen und in einem einzigen Durchlauf durch die Rendering Pipeline zu berechnen. Die enorm hohe Rechenleistung heutiger Grafikkarten bei gleichzeitiger Programmierbarkeit lässt erwarten, dass so manche Investition in enorm teure Spezialhardware für Echtzeit-Bildverarbeitung überdacht wird.

13.3 Projektive Texturen (Projective Texture)

In Abschnitt 13.1.6.3 wurde gezeigt, dass bei der automatischen Generierung von Texturkoordinaten im Modus `GL_EYE_LINEAR` eine Textur nicht unbedingt fest mit einem Objekt verbunden sein muss, sondern dass sie auch an den Augenpunkt gebunden sein kann. Die Objekte „schwimmen" in diesem Fall unter der Textur durch (Bild 13.14-g,h,i), wie bei einem fest aufgestellten Diaprojektor, der im Augenpunkt sitzt. Geht man noch einen Schritt weiter, so kann man den Diaprojektor – und damit die projizierte Textur – beliebig im Raum drehen und bewegen, ohne den Augenpunkt zu verändern. Dadurch wird die Textur über eine beliebig im 3-dimensionalen Raum liegende Ebene auf ein oder mehrere Polygone projiziert. Auf diese Weise lassen sich qualitativ hochwertige Effekte erzielen, wie z.B. die Bewegung eines Scheinwerferkegels über die Oberflächen einer Szene (Bild 13.17).

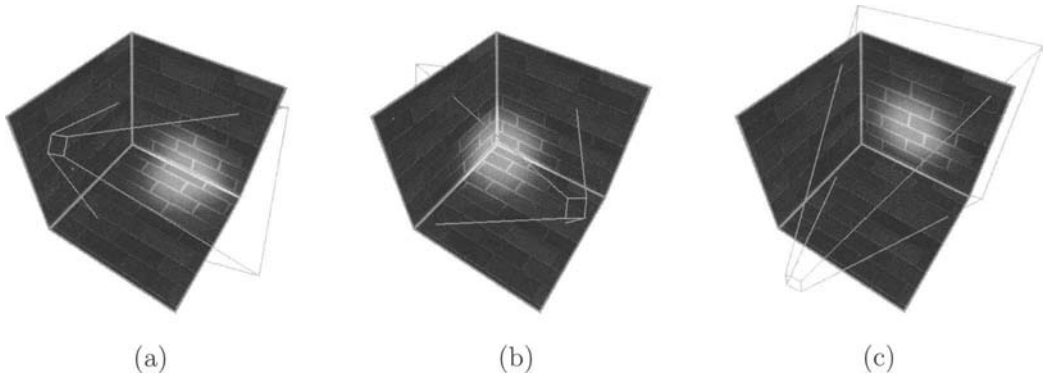


Bild 13.17: Projektive Texturen (Projective Texture) in Verbindung mit Mehrfach-Texturierung: ein Lichtkegel wird als projektive *light map* über Polygongrenzen hinweg bewegt und simuliert damit eine lokale gerichtete Lichtquelle (*spot light*, Abschnitt 12.1.3.5).

Für eine unabhängige Bewegung der projizierten Textur werden die entsprechenden Transformationen in einer eigenen Matrix gesondert gespeichert. Zur automatischen Generierung der Texturkoordinaten wird diese Matrix invertiert, genau wie im `GL_EYE_LINEAR`-Modus. Die beliebig im 3-dimensionalen Raum liegende Projektionsebene muss nun noch auf die vordere Begrenzungsebene (*near clipping plane*) des sichtbaren Volumens abgebildet sowie verschoben und skaliert werden. Die resultierende Matrix wird als Textur-Matrix im Textur-Matrizen-Stapel abgelegt und bei der automatischen Generierung der Texturkoordinaten angewendet. Zur Realisierung einer projektiven Textur lässt sich das im vorigen Abschnitt aufgelistete Programm zur Mehrfach-Texturierung gut nutzen. Die Textur-Matrix im ersten Absatz des Programms muss dazu folgendermaßen modifiziert werden:

```
GLfloat M[4][4];
GLfloat Minv[4][4];
```

```
InvertMatrix((GLfloat *) Minv, M);

...

glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glTranslatef(0.5, 0.5, 0.5);
glScalef(0.5, 0.5, 0.5);
glFrustum(xmin, xmax, ymin, ymax, znear, zfar);
glTranslatef(0.0, 0.0, -1.0);
glMultMatrixf((GLfloat *) Minv);
glMatrixMode(GL_MODELVIEW);
```

Die Matrix `M` enthält die Verschiebungen und Drehungen der Projektionsebene. Die Funktion `InvertMatrix((GLfloat *) Minv, M)` invertiert die Matrix `M` und schreibt das Ergebnis in die Matrix `Minv`.

Als projektive Texturen lassen sich nicht nur Lichtkegel verwenden, sondern beliebige Texturen. Ein Beispiel dafür sind projektive Schatten-Texturen. Unter der Annahme einer weit entfernten Lichtquelle wirft ein Objekt einen Schatten, dessen Größe und Form unabhängig von der Position des Objekts in der Szene ist. Der Schatten ergibt sich einfach durch eine orthografische Projektion des Objekts entlang des Lichtvektors auf eine Ebene. Die Schatten-Textur enthält innerhalb des Schattens einen Grauwert kleiner als eins und außerhalb den Grauwert eins. Bei einer multiplikativen Verknüpfung von Schatten-Textur und Objekt-Textur wird die Objekt-Textur im Bereich des Schattens um einen bestimmten Prozentsatz dunkler. Mit dieser Technik lassen sich auch weiche Schattenübergänge erzeugen. Die Position und Lage der Schatten-Textur ergibt sich einfach aus der Verbindungsgeraden zwischen Lichtquelle und Objekt. Diese Technik des Schattenwurfs ist allerdings mit einigen gravierenden Nachteilen verbunden:

- Jedes Objekt innerhalb einer Szene benötigt eine eigene projektive Schatten-Textur, was sehr schnell zu Problemen mit dem Texturspeicher führt.

- Falls es sich nicht um rotationssymmetrische Objekte handelt, müsste sich die projektive Schatten-Textur mit allen Drehungen des Objekts ändern, deren Drehachse nicht dem Lichtvektor entspricht.

- Falls die Lichtquelle nicht weit entfernt ist, ändert sich die Größe des Schattens abhängig vom Abstand zwischen dem Objekt und der Oberfläche, auf die der Schatten fällt. Eine Größenkorrektur des Schattens zieht aufwändige Abstandsberechnungen nach sich.

Aufgrund dieser Nachteile werden projektive Schatten-Texturen in der interaktiven 3D-Computergrafik eher selten verwendet. Eine bessere Methode der Schattenerzeugung wird in Abschnitt 13.6 dargestellt.

13.4 Umgebungs-Texturen (Environment Maps)

Bis vor wenigen Jahren war ein klares Erkennungszeichen von interaktiver 3D-Computergrafik das Fehlen von realistischen Spiegelungen. Dies war den sehr viel aufwändigeren Ray-Tracing-Verfahren (Abschnitt 12.1.2) vorbehalten. Durch die Einführung von Umgebungs-Texturen (*Environment Maps*, [Blin76]) können Objekt-Spiegelungen jedoch schon ziemlich gut auf aktuellen Grafikkarten in Echtzeit dargestellt werden. Das Grundprinzip ist einfach und gleicht in den ersten Schritten dem Ray-Tracing-Verfahren. Man schickt einen Strahl vom Augenpunkt auf einen Punkt der reflektierenden Oberfläche und berechnet mit Hilfe des Normalenvektors den Reflexionsvektor. Anstatt den reflektierten Strahl bis zur nächsten Objektoberfläche zu verfolgen, um dort ein lokales Beleuchtungsmodell auszuwerten, wie dies beim Ray-Tracing der Fall ist, wird jetzt die Richtung des Reflexionsvektors benutzt, um die Texturkoordinaten in einer Umgebungs-Textur zu bestimmen (Bild 13.18).

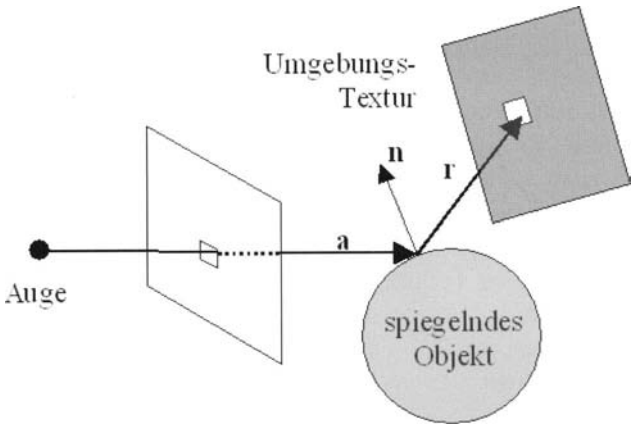


Bild 13.18: Das Prinzip des Environment Mappings: ein Beobachter blickt auf ein spiegelndes Objekt und sieht darin die Umgebung in der Richtung des Reflexionsvektors. Die Umgebung des Objekts wird in Form einer Umgebungs-Textur gespeichert.

Die Umgebungs-Textur enthält das Bild der Umgebung aus dem Blickwinkel des Objekts. Solange die Umgebung relativ weit entfernt von dem Objekt ist und sich die Umgebung nicht verändert, muss man sie nicht, wie beim Ray-Tracing, jedesmal neu berechnen, sondern kann sie in einer Textur abspeichern. Auch bei Objektbewegungen, die im Verhältnis zur Entfernung zwischen Objekt und Umgebung klein sind, kann die Umgebungs-Textur wieder verwendet werden. Im Umkehrschluss sind damit auch die Einschränkungen von *Environment Mapping* Techniken offensichtlich. Spiegelungen in Szenen, bei denen mehrere Objekte relativ nah beieinander sind, können nur für eine Objektkonstellation und einen Blickwinkel korrekt dargestellt werden. Falls sich Blickwinkel oder Objektpositionen ändern, müsste für jedes Bild und evtl. sogar für jedes Objekt vorab eine Umgebungs-

Textur erzeugt werden. Dies ist natürlich sehr rechenaufwändig und benötigt mehrere Durchläufe durch die Rendering Pipeline (*Multipass Rendering*, Abschnitt 13.1.1.2).

Die Berechnung der Texturkoordinaten für eine Umgebungs-Textur kann auf unterschiedlichen Genauigkeitsstufen ablaufen:

- per Vertex, dann müssen die pixel-bezogenen Texturkoordinaten wie üblich durch lineare Interpolation gewonnen werden.
- per Pixel, was zu besseren Ergebnissen führt, aber deutlich rechenaufändiger ist, da der Reflexionsvektor für jedes Pixel zu berechnen ist. Diese Variante setzt einen programmierbaren Pixel-Shader auf der Grafikkarte voraus.

In der interaktiven 3D-Computergrafik werden vor allem die zwei *Environment Mapping* Techniken eingesetzt, die auch in OpenGL verfügbar sind: Sphärische Texturierung (*Sphere Mapping*) und Kubische Texturierung (*Cube Mapping*).

13.4.1 Sphärische Texturierung (Sphere Mapping)

Die Grundidee der Sphärischen Texturierung (*Sphere Mapping*) besteht darin, eine ideal verspiegelte Kugel aus großer Entfernung mit einem starken Teleobjektiv zu betrachten, was im Grenzfall einer orthografischen Projektion entspricht [Mill84]. In der Kugel spiegelt sich deren gesamte Umgebung mit zunehmender Verzerrung zum Rand hin. Ein Foto dieser Kugel mit der gespiegelten Umgebung wird als Sphärische Textur (*Sphere Map*) bezeichnet. Die Kugel deckt innerhalb einer solchen Textur einen kreisförmigen Bereich mit Mittelpunkt $(0.5, 0.5)$ und Radius 0.5 ab (Bild 13.19-a). Die verspiegelte Kugel bildet die Richtung eines Reflexionsvektors auf einen Punkt der sphärischen Textur ab. Man benötigt also für die Berechnung der Texturkoordinaten diese Abbildung und den Reflexionsvektor.

Für jeden Reflexionsvektor $\mathbf{r}$ muss man Texturkoordinaten (s, t) bestimmen, die einem Punkt innerhalb des kreisförmigen Bereichs der sphärischen Textur entsprechen. Nachdem die sphärische Textur durch eine orthografische Projektion der Einheitskugel erzeugt wurde, ergibt sich der Zusammenhang zwischen den Texturkoordinaten (s, t) und einem Punkt (x, y, z) auf der Einheitskugel durch Skalierung um den Faktor 2 und Verschiebung um eine negative Einheit für die x- und die y-Koordinate, sowie aus der Kreisgleichung $r = 1 = \sqrt{x^2 + y^2 + z^2}$ für die z-Koordinate (Bild 13.19-a), d.h.:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2s - 1 \\ 2t - 1 \\ \sqrt{1 - x^2 - y^2} \end{pmatrix} \tag{13.15}$$

Für die Einheitskugel im Ursprung gilt aber, dass die Einheitsnormalenvektoren $\mathbf{n^e}$ durch die Koordinaten (x, y, z) des jeweiligen Punkts auf der Oberfläche gegeben sind (Bild 13.19-b), d.h.:

$$\mathbf{n^e} = \begin{pmatrix} n_x^e \\ n_y^e \\ n_z^e \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2s - 1 \\ 2t - 1 \\ \sqrt{1 - x^2 - y^2} \end{pmatrix} \tag{13.16}$$

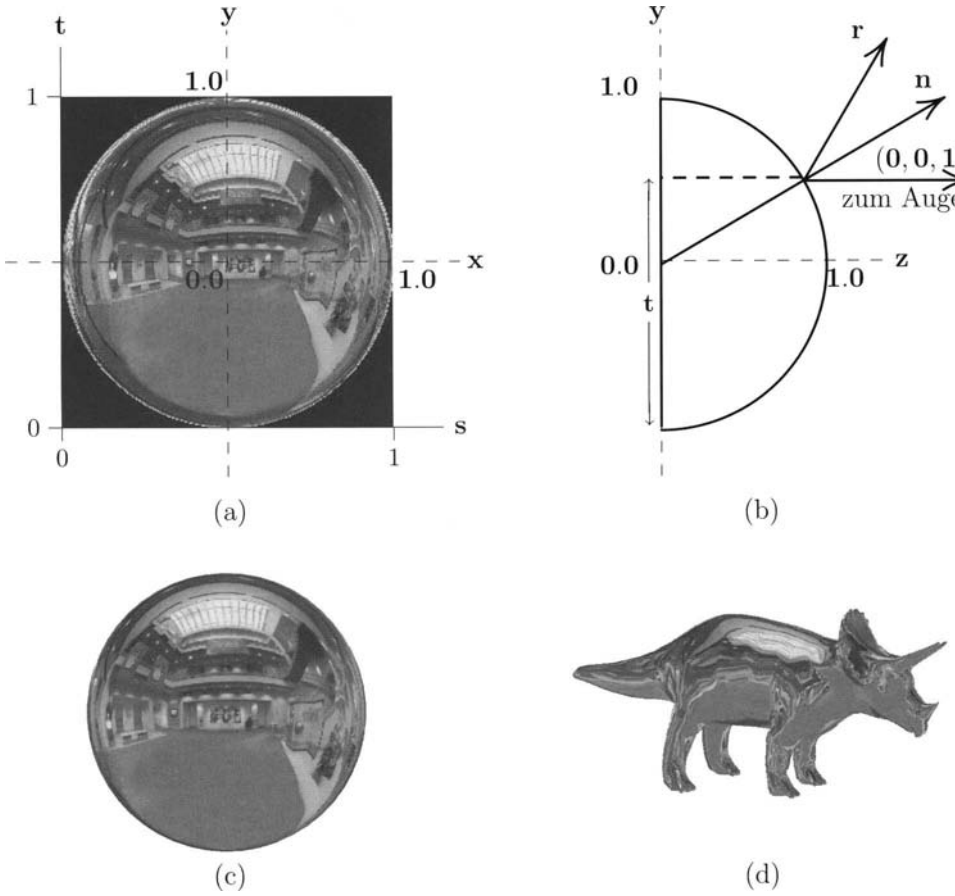


Bild 13.19: Sphärische Texturierung (*Sphere Mapping*): (a) Eine sphärische Textur ist das Foto einer spiegelnden Kugel. Die Kugel deckt in Texturkoordinaten einen kreisförmigen Bereich mit Mittelpunkt $(0.5, 0.5)$ und Radius 0.5 ab. (b) Die Seitenansicht der vorderen Halbkugel: der Normalenvektor $\mathbf{n}$ ist die Winkelhalbierende zwischen Augenpunktsvektor $(0, 0, 1)^T$ und Reflexionsvektor $\mathbf{r}$. Für die Einheitskugel im Ursprung gilt, dass die Koordinaten eines Punktes auf der Oberfläche gleich dem Einheitsnormalenvektor sind $((x, y, z)^T = (n_x^e, n_y^e, n_z^e)^T)$. (c) Das Mapping einer sphärischen Textur auf eine Kugel. (d) Das Mapping einer sphärischen Textur auf das Triceratops-Modell.

Als Nächstes bestimmt man den Zusammenhang zwischen dem Reflexionsvektor $\mathbf{r}$ und dem Normalenvektor $\mathbf{n}$. In Weltkoordinanten berechnet sich der Reflexionsvektor $\mathbf{r}$ aus dem Vektor $\mathbf{a}$ vom Augenpunkt zum Oberflächenpunkt und dem Normalenvektor $\mathbf{n}$ mit Hilfe des Reflexionsgesetzes (12.2) zu:

$$\mathbf{r} = \mathbf{a} - 2(\mathbf{a} \cdot \mathbf{n}) \cdot \mathbf{n} \tag{13.17}$$

Der Vektor $\mathbf{a}$ vom Augenpunkt zum Oberflächenpunkt ist in Weltkoordinaten durch die

negative z-Achse gegeben: $\mathbf{a} = (0, 0, -1)^T$. Setzt man dies in (13.17) ein, erhält man:

$$\begin{pmatrix} r_x \\ r_y \\ r_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} - 2\left(\begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \cdot \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}\right) \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} + 2n_z \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} \quad (13.18)$$

Aufgelöst nach dem Normalenvektor ergibt:

$$\mathbf{n} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} = \frac{1}{2n_z} \begin{pmatrix} r_x \\ r_y \\ r_z + 1 \end{pmatrix} \quad (13.19)$$

Durch Normierung $|\mathbf{n}| = \sqrt{n_x^2 + n_y^2 + n_z^2} = \frac{1}{2n_z}\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$ entsteht der Einheitsnormalenvektor $\mathbf{n^e}$:

$$\mathbf{n^e} = \begin{pmatrix} n_x^e \\ n_y^e \\ n_z^e \end{pmatrix} = \frac{1}{\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}} \begin{pmatrix} r_x \\ r_y \\ r_z + 1 \end{pmatrix} \quad (13.20)$$

Dieses Ergebnis ist sehr gut verständlich, da der Normalenvektor nichts Anderes ist als der winkelhalbierende Vektor zwischen dem Augenpunktsvektor $(0, 0, 1)^T$ und dem reflektierten Strahl $(r_x, r_y, r_z)^T$. Der Einheitsnormalenvektor ergibt sich daraus einfach durch Addition $(r_x, r_y, r_z + 1)^T$ und Normierung (Bild 13.19-b). Durch Gleichsetzen von (13.16) und (13.20)

$$\begin{pmatrix} 2s - 1 \\ 2t - 1 \\ \sqrt{1 - x^2 - y^2} \end{pmatrix} = \begin{pmatrix} n_x^e \\ n_y^e \\ n_z^e \end{pmatrix} = \frac{1}{\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}} \begin{pmatrix} r_x \\ r_y \\ r_z + 1 \end{pmatrix} \quad (13.21)$$

kann man die gesuchten Texturkoordinaten s, t in Abhänigkeit vom Reflexionsvektor $\mathbf{r}$ angeben:

$$s = \frac{r_x}{2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}} + \frac{1}{2} \quad (13.22)$$

$$t = \frac{r_y}{2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}} + \frac{1}{2} \quad (13.23)$$

Die Berechnung des Reflexionsvektors mit (13.17) und der Texturkoordinaten mit (13.22) und (13.23) wird von OpenGL vorgenommen, wenn die automatische Texturkoordinatengenerierung auf den Modus `GL_SPHERE_MAP` eingestellt wurde:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

Weitere Parameter, wie bei den anderen Generiermodi (`GL_OBJECT_LINEAR`, `GL_EYE_LINEAR`), gibt es in diesem Fall nicht. Die Ergebnisse, die man mit der sphärischen Projektion der Texturkoordinaten in Verbindung mit einer sphärischen Textur für konvexe spiegelnde Oberflächen erzielen kann, sind durchaus beeindruckend (Bild 13.19-c,d).

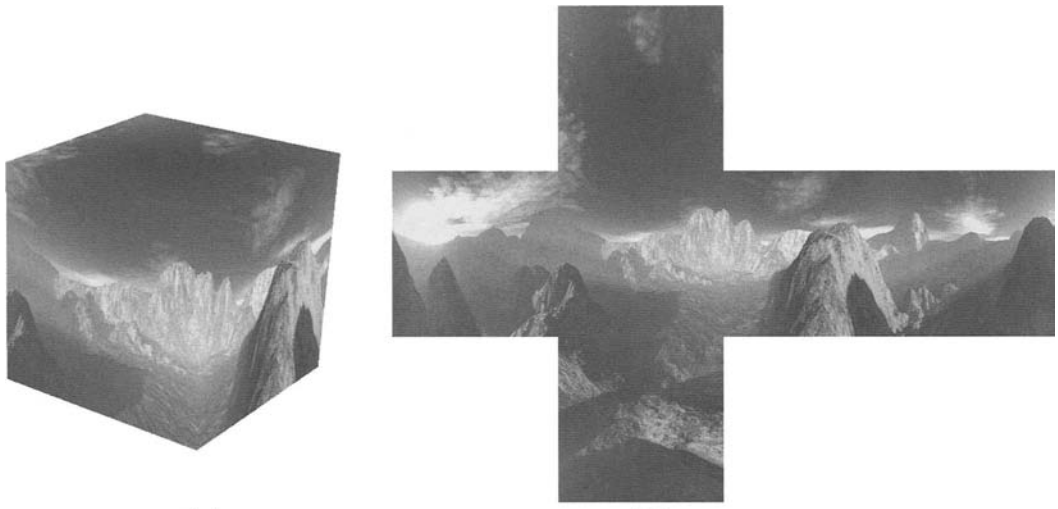
Sphärische Texturierung ist auf nahezu allen heutigen Grafikkarten verfügbar, da man nur eine normale Textur benötigt und die Texturkoordinatengenerierung im Rahmen einer Textur-Matrix implementiert werden kann. Sinnvolle Ergebnisse kann man jedoch nur erzielen, wenn die Textur sphärisch ist, d.h. das Bild einer spiegelnden Kugel enthält. Solch eine Textur kann aber relativ einfach erzeugt werden. Entweder durch Fotografieren der realen Umgebung mit einem extremen Weitwinkelobjektiv (Fischauge) bzw. einer versilberten Christbaumkugel mit einem starken Teleobjektiv oder durch 3D-Computergrafik z.B. mit Hilfe von Ray-Tracing-Verfahren bzw. Verzerrung planarer Texturen.

Sphärische Texturierung funktioniert bei konkaven Oberflächen nur unzureichend, da keine Eigenspiegelungen möglich sind. Ein weiterer Nachteil der sphärischen Texturierung ist, dass sie streng genommen nur für einen Blickwinkel gilt. Bewegt sich der Beobachter beispielsweise um das spiegelnde Objekt herum, sieht er nicht die andere Seite der Umgebung, sondern immer nur die gleiche. Daher entsteht der Eindruck, dass sich nicht der Beobachter um das Objekt bewegt, sondern dass sich das Objekt dreht. Diese Schwäche kann erst durch die im nächsten Abschnitt dargestellte kubische Texturierung beseitigt werden.

13.4.2 Kubische Texturierung (Cube Mapping)

Bei der kubischen Texturierung (*Cube Mapping*, [Gree86]) verwendet man nicht nur eine Umgebungs-Textur, wie bei der sphärischen Texturierung, sondern sechs 2-dimensionale Umgebungs-Texturen, die die Flächen eines Kubus' bilden. Im Zentrum des Kubus befindet sich das zu texturierende Objekt. Die sechs Einzeltexturen, die zusammen die kubische Textur bilden, werden einfach dadurch gewonnen, dass man die Umgebung aus der Position des Objektmittelpunkts sechs mal mit einem Öffnungswinkel von 90° fotografiert oder rendert, und zwar so, dass die sechs Würfelflächen genau abgedeckt werden. Die sechs Einzeltexturen müssen also an den jeweiligen Rändern übergangslos zusammen passen. In Bild 13.20 ist ein Beispiel für eine kubische Umgebungs-Textur dargestellt.

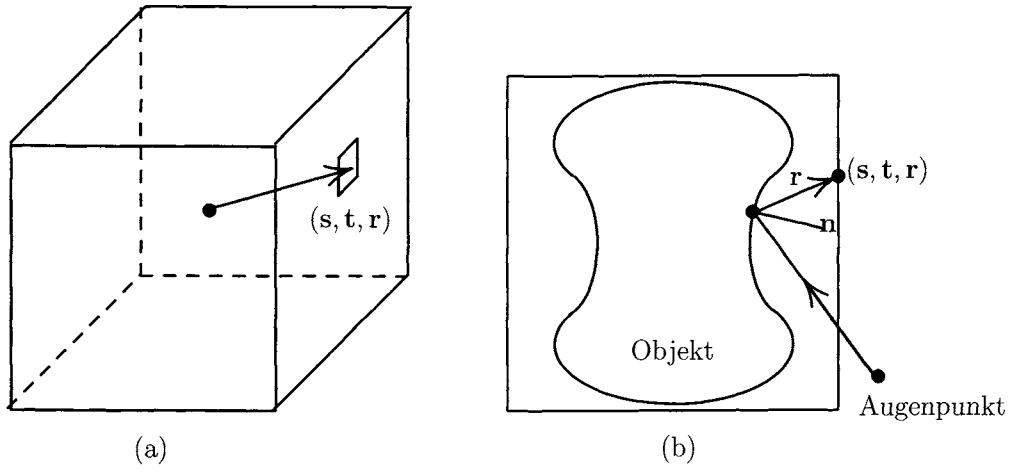
Eine kubische Textur wird formal als eine Bildmatrix $\mathbf{T}$ beschrieben, die von den drei Variablen (s, t, r) eines orthogonalen Texturkoordinatensystems abhängt, genau wie bei einer 3-dimensionalen Textur. Allerdings werden die Texturkoordinaten als Richtungsvektor betrachtet, der angibt, welches Texel man sieht, wenn man vom Zentrum in Richtung des Vektors geht (Bild 13.21). Möchte man wieder die Spiegelung der Umgebung in einem Objekt bestimmen, wird der Reflexionsvektor $\mathbf{r}$ als Richtungsvektor benutzt. Die Vektoren müssen in diesem Fall nicht normiert werden. Die Koordinate des Reflexionsvektors mit dem größten Absolutwert bestimmt, welche der sechs Texturen ausgewählt wird (z.B. wird durch den Vektor $(1.3, -4.2, 3.5)$ die $-Y$ Fläche ausgewählt). Die verbleibenden zwei Koordinaten werden durch den Absolutwert der größten Koordinate geteilt, so dass sie im Intervall $[-1, +1]$ liegen. Anschließend werden sie mit dem Faktor $\frac{1}{2}$ skaliert und um



(a) (b)

Bild 13.20: Kubische Texturierung (*Cube Mapping*): (a) Eine kubische Textur bedeckt die sechs Seiten eines Würfels, in dessen Zentrum sich die zu rendernden Objekte befinden. (b) Die sechs Flächen der kubischen Textur aufgeklappt. Quelle: Thomas Bredl

den Wert 0.5 verschoben, so dass sie im gewünschten Intervall $[0, 1]$ liegen (im Beispiel $(1.3/4.2/2 + 0.5, 3.5/4.2/2 + 0.5) = (0.65, 0.92)$). Die auf diese Weise berechneten Texturkoordinaten (s', t') dienen zum Abgreifen der Texel in der ausgewählten Textur.



(a) (b)

Bild 13.21: Das Prinzip der kubischen Texturierung: (a) Die Texturkoordinaten (s, t, r) werden als Richtungsvektor interpretiert. (b) Zur Berechnung von Spiegelungen wird der Reflexionsvektor $\mathbf{r}$ benutzt, um die Texturkoordinaten zu bestimmen.

Seit der Version 1.3 sind kubische Texturen Bestandteil von OpenGL. Zur Spezifikation kubischer Texturen wird der OpenGL-Befehl `glTexImage2D()` sechs Mal aufgerufen (Abschnitt 13.1.1), wobei der erste Parameter ***target*** angibt, welche der sechs Seiten des Kubus $(+X, -X, +Y, -Y, +Z, -Z)$ definiert wird. Alle Einzeltexturen einer kubischen Textur müssen quadratisch sein und die gleiche Größe besitzen. Im Folgenden Programmausschnitt zur Definition einer kubischen Textur wird angenommen, dass die Bilddaten bereits in das 4-dimensionale Array `image` eingelesen wurden:

```
GLint size = 256;
static GLubyte image[6][size][size][4];

glTexImage2D ( GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA,
               size, size, 0, GL_RGBA, GL_UNSIGNED_BYTE, image[0]);
glTexImage2D ( GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGBA,
               size, size, 0, GL_RGBA, GL_UNSIGNED_BYTE, image[1]);
glTexImage2D ( GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGBA,
               size, size, 0, GL_RGBA, GL_UNSIGNED_BYTE, image[2]);
glTexImage2D ( GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGBA,
               size, size, 0, GL_RGBA, GL_UNSIGNED_BYTE, image[3]);
glTexImage2D ( GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGBA,
               size, size, 0, GL_RGBA, GL_UNSIGNED_BYTE, image[4]);
glTexImage2D ( GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGBA,
               size, size, 0, GL_RGBA, GL_UNSIGNED_BYTE, image[5]);
```

Texturfortsetzungsmodi und Textur-Filter werden immer für die gesamte kubische Textur festgelegt, wie im Folgenden Beispiel gezeigt:

```
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_REPEAT);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Grundsätzlich kann man die Texturkoordinaten einer kubischen Textur auch explizit den Vertices zuordnen (Abschnitt 13.1.6.1). Zur Erzielung eines Spiegelungseffekts benutzt man jedoch sinnvollerweise die automatische Texturkoordinatengenerierung (Abschnitt 13.1.6.3) in dem Modus `GL_REFLECTION_MAP`. Dadurch werden die Reflexionsvektoren und die Texturkoordinaten (s, t, r) für die *Cube Map* Textur automatisch erzeugt. Abschließend muss man nur noch die entsprechenden OpenGL-Zustände aktivieren, wie in den folgenden Zeilen dargestellt:

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP);
```

```
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glEnable(GL_TEXTURE_CUBE_MAP);
```

Kubische Texturen bieten gegenüber sphärischen Texturen einige Vorteile:

- Die Spiegelungen sind blickwinkelunabhängig, d.h., bewegt sich der Beobachter um das Objekt herum, spiegelt sich in dem Objekt die richtige Umgebung wieder, da die entsprechende Einzeltextur ausgewählt wird.
- Die einzelnen Texturen der kubischen Textur sind einfacher zu generieren, da man nur eine ebene und keine sphärische Projektion benötigt. Kubische Texturen können daher sogar in Echtzeit berechnet werden, wie z.B. in dem Xbox-Spiel „*Project Gotham Racing*“ [Aken02].
- Es gibt geringere Verzerrungen, da jede der sechs Einzeltexturen nur einen Teil des Raumwinkels abdecken muss (Bild 13.11).
- Mit kubischen Texturen ist es auch möglich, andere optische Phänomene wie z.B. Brechung, Dispersion und den Fresnel-Effekt zu simulieren, wie im nächsten Absatz dargestellt.
- Kubische Texturen können nicht nur Farbwerte enthalten, die die Umgebung repräsentieren, sondern auch beliebige andere Inhalte transportieren. Ein Beispiel dafür ist der Einsatz von kubischen Texturen zur schnellen Normierung von Vektoren, wie bei der Relief-Texturierung (Abschnitt 13.5).

Die einzigen Nachteile von kubischen Texturen sind der um den Faktor 6 größere Texturspeicherbedarf und die nicht durchgängige Verfügbarkeit auf heutigen Grafikkarten. Die Relevanz dieser Nachteile nimmt jedoch mit dem rasanten Fortschritt der Grafikhardware immer mehr ab, so dass kubische Texturierung wahrscheinlich der Standard für das *Environment Mapping* und andere Texture Mapping Verfahren wird.

Brechungseffekte, Dispersion und Fresnel-Effekt

Die Berechnung von Brechungseffekten läuft ganz analog wie bei den Spiegelungen. Der einzige Unterschied ist, das jetzt anstatt dem Reflexionsvektor der Brechungsvektor berechnet werden muss. Der Vektor des gebrochenen Strahls kann mit Hilfe von (12.2) aus dem Augenpunktsvektor, dem Normalenvektor und dem Brechungsindex berechnet werden. Unter Vernachlässigung der Brechung des Strahls beim Austreten aus dem transparenten Objekt kann jetzt anstatt des Reflexionsvektors der Brechungsvektor benutzt werden, um die Texturkoordinaten der Umgebungs-Textur zu bestimmen. Um die Dispersion des Lichts, d.h. die zunehmende Brechung von kurzwelligeren elektromagnetischen Wellen, zu

simulieren, berechnet man für jede der drei Farbkomponenten R,G,B einen eigenen Brechungsvektor auf der Basis der jeweiligen Brechungsindizes. Für jeden der drei Brechungsvektoren wird ein Satz Texturkoordinaten bestimmt. Die Texturkoordinaten des „roten" Brechungsvektors dienen zum Abgreifen der roten Farbkomponente des Fragments aus der Umgebungs-Textur, die anderen beiden Sätze an Texturkoordinaten für die grüne und die blaue Farbkomponente. Jede Farbkomponente wird also an einer leicht verschobenen Position der Umgebungs-Textur abgegriffen. Dadurch entsteht eine Art Prismeneffekt, d.h. die örtliche Aufspaltung verschiedener Spektralanteile von weißem Licht. Es entsteht natürlich kein kontinuierliches Farbspektrum, wie in der Realität, sondern nur ein diskretes Spektrum aus den drei R,G,B-Linien. Dennoch sind die Ergebnisse sehr ansprechend, und darauf kommt es letztlich in der Computergrafik an. Schließlich kann man auch noch den Fresnel-Effekt simulieren, bei dem mit zunehmendem Einfallswinkel des Lichts auf die Oberfläche ein immer größerer Prozentsatz des Lichtstrahls reflektiert und der komplementäre Anteil gebrochen wird. Neben den drei Texturkoordinaten-Sätzen für die drei Farbkomponenten des gebrochenen Lichtstrahl kommt einfach noch ein vierter Texturkoordinaten-Satz für die Reflexion dazu. Man hat es in diesem Fall also mit einer vierfachen Abtastung der Umgebungs-Textur zu tun (Abschnitt 13.2). Gemäß dem Fresnel'schen Gesetz (12.3) werden je nach Einfallswinkel des Lichtstrahls zwei Faktoren berechnet, mit denen die Farben des gebrochenen und des reflektierten Lichts gewichtet aufsummiert werden. Solche anspruchsvollen *Environment Mapping* Verfahren lassen sich mittlerweile in Echtzeit auf den neuesten Grafikkarten realisieren. Für die programmiertechnischen Details wird aus Platzgründen auf das *Cg Tutorial* [Fern03] verwiesen.

13.5 Relief-Texturierung (Bump Mapping)

Durch das Fotografieren einer rauen oder reliefartigen Oberfläche und das anschließende Mapping dieser Fototextur auf Polygone kann man die komplexesten Beleuchtungseffekte wiedergeben. Allerdings gilt das nur für statische Szenen, in denen die Position der Lichtquelle relativ zu den Oberflächen genau der fotografischen Aufnahmesituation entspricht. Um dynamische Beleuchtungssituationen in Verbindung mit Oberflächenunebenheiten korrekt darstellen zu können, benötigt man daher ein neues Verfahren, die sogenannte „Relief-Texturierung" (*Bump Mapping*).

Da die menschliche Wahrnehmung aus der Schattierung auf die räumliche Form schließt, kann der Eindruck eines Reliefs durch eine lokale (d.h. pixel-bezogene) Veränderung der Normalenvektoren in Verbindung mit einer pixel-bezogenen Beleuchtungsrechnung (Phong-Shading, Abschnitt 12.2.3) erreicht werden. Eine normale Relief-Textur (*Bump Map*) enthält folglich Werte, die angeben, wie die Normalenvektoren einer glatten Oberfläche modifiziert werden müssen. Eine andere Relief-Textur-Variante, die sogenannte Normalen-Textur (*Normal Map*), enthält direkt die modifizierten Normalenvektoren für jedes Pixel. Da sich die zuletzt genannte Variante am Markt durchgesetzt hat, werden im Folgenden nur noch Normalen-Texturen behandelt.

Die Situation bei der Relief-Texturierung ist sehr gut mit dem Gouraud-Shading (Ab-

schnitt 12.2.2) vergleichbar: um einen glatten Farb- bzw. Helligkeitsverlauf an den Polygonkanten zu erhalten, werden dort die Flächen-Normalenvektoren angrenzender Polygone gemittelt und der resultierende Normalenvektor dem gemeinsamen Vertex zugewiesen. Der Vertex-Normalenvektor steht auf keinem der angrenzenden Polygone senkrecht, sondern nur auf der virtuellen Oberfläche, die die Polygone approximieren. Dadurch erscheinen Polyeder rund, ohne dass die zugrunde liegende Geometrie geändert werden müsste. Das Grundprinzip bei der Relief-Texturierung ist das gleiche, nur dass es jetzt auf die Pixel-Ebene bezogen ist und nicht auf die Vertex-Ebene. In diesem Fall werden die Normalenvektoren pro Pixel modifiziert, um virtuelle Oberflächenkrümmungen mit sehr hohen Ortsfrequenzen zu simulieren. Dadurch erscheint die Oberfläche eines Polygons durch reliefartige Vertiefungen oder Erhöhungen verbeult, obwohl die zugrunde liegende Geometrie glatt und somit primitiv ist (Bild 13.22).

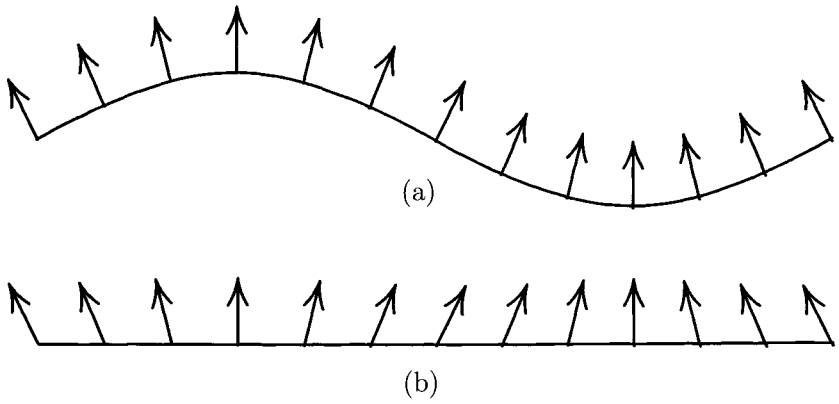


Bild 13.22: Das Prinzip der Relief-Texturierung: (a) Eine wellige Oberfläche mit den zugehörigen Normalenvektoren. (b) Die selben Normalenvektoren wie oben simulieren die wellige Oberfläche durch Schattierung, obwohl die tatsächliche Oberfläche eben ist.

Zur Darstellung des Prinzips wird ein einfaches Beispiel betrachtet: ein ebenes Rechteck, das als erhobenes Relief das Logo der Fachhochschule München (fhm) enthalten soll (Bild 13.23-a). Dazu werden die Grauwerte des Originalbildes als Höhenwerte aufgefasst, so dass eine Art „Grauwertgebirge" entsteht (Abschnitt 18 und Bild 18.1-c,d). Dunklere Bereiche, d.h. niedrigere Grauwerte, werden als niedrig eingestuft, und hellere Bereiche als hoch. Zu diesem Grauwertgebirge konstruiert man nun die zugehörigen Normalenvektoren für jedes Texel durch folgenden Algorithmus **A13.1**:

A13.1: Algorithmus zur Erzeugung einer *Normal Map* Textur.

Voraussetzungen und Bemerkungen:

◇ Falls die Originaltextur ein RGB-Farbbild ist, wird durch eine geeignete Opera-

tion (z.B. Mittelung der drei Farbkomponenten), ein einkanaliges Grauwertbild daraus erzeugt.

◇ Die Grauwerte der Originaltextur können sinnvoll als Höhenwerte interpretiert werden.

Algorithmus:

(a) Berechne die Gradienten des Grauwertgebirges in Richtung der s- und der t-Texturkoordinaten für jedes Texel. Dies kann z.B. durch Differenzenoperatoren erreicht werden (Abschnitt 18.4): der Gradientenvektor in s-Richtung ist $(1, 0, g_{s+1,t} - g_{s,t})^T$ und der Gradientenvektor in t-Richtung ist $(0, 1, g_{s,t+1} - g_{s,t})^T$, wobei $g_{s,t}$ der Grauwert des betrachteten Texels ist, $g_{s+1,t}$ der Grauwert des rechten Nachbar-Texels und $g_{s,t+1}$ der Grauwert des oberen Nachbar-Texels.

(b) Berechne das Kreuzprodukt der beiden Gradientenvektoren, um einen Normalenvektor zu erhalten, der senkrecht auf der Oberfläche des Grauwertgebirges steht:

$$\mathbf{n} = \begin{pmatrix} 1 \\ 0 \\ g_{s+1,t} - g_{s,t} \end{pmatrix} \times \begin{pmatrix} 0 \\ 1 \\ g_{s,t+1} - g_{s,t} \end{pmatrix} = \begin{pmatrix} g_{s,t} - g_{s+1,t} \\ g_{s,t} - g_{s,t+1} \\ 1 \end{pmatrix} \tag{13.24}$$

(c) Normiere den Normalenvektor, um als Endergebnis einen Einheitsnormalenvektor zu erhalten:

$$\mathbf{n}^{\mathrm{e}} = \frac{1}{\sqrt{(g_{s,t} - g_{s+1,t})^2 + (g_{s,t} - g_{s,t+1})^2 + 1}} \begin{pmatrix} g_{s,t} - g_{s+1,t} \\ g_{s,t} - g_{s,t+1} \\ 1 \end{pmatrix} \tag{13.25}$$

(d) Transformiere den Einheitsnormalenvektor aus dem Wertebereich $[-1, +1]$ in den Bereich $[0, 1]$, damit die drei Komponenten des Vektors als RGB-Werte einer Textur $\mathbf{t}$ gespeichert werden können:

$$\mathbf{t} = \frac{1}{2} \cdot (\mathbf{n}^{\mathrm{e}} + \mathbf{1}) \tag{13.26}$$

Ende des Algorithmus

Anstatt der relativ störanfälligen Differenzenoperatoren können als Alternative z.B. auch Sobel-Operatoren zur Berechnung der Gradienten eingesetzt werden (Abschnitt 18.4). Die mit dem Algorithmus **A13.1** aus dem Eingabebild 13.23-a berechnete Normalen-Textur ist in den Bildern 13.23-b,c,d in ihren RGB-Farbauszügen zu sehen. Der Rotauszug (b) entspricht der n_x-Komponente und enthält die Ableitung des Bildes in horizontaler Richtung. Aufgrund der Transformation in den Wertebereich $[0, 1]$ erscheinen Kanten mit einem Hell-Dunkel-Übergang weiß, Kanten mit einem Dunkel-Hell-Übergang schwarz und Bereiche des

Bildes mit einer einheitlichen Helligkeit, also Ableitung gleich null, erscheinen mittelgrau (0.5). Der Grünauszug (c) entspricht der n_y-Komponente und enthält die Ableitung des Bildes in vertikaler Richtung. Der Blauauszug (d) entspricht der n_z-Komponente und enthält in allen Bereichen des Bildes mit einer einheitlichen Helligkeit den maximalen Grauwert, d.h. weiß, da in diesem Fall der Normalenvektor genau in z-Richtung zeigt $\mathbf{n}^e = (0, 0, 1)^T$. Nur an den Kanten, egal ob horizontal oder vertikal, weicht der Normalenvektor von der z-Richtung ab, so dass die z-Komponente kleiner als 1 wird und somit auch der Grauwert des Blauauszugs. Kanten jeglicher Orientierung zeichnen sich also im Blauauszug dunkler ab.

Mit dieser Normalen-Textur wird jetzt die Relief-Texturierung eines einfachen Rechtecks durchgeführt. Alles, was man dafür sonst noch benötigt, ist eine Beleuchtungsrechnung pro Pixel, d.h. Phong-Shading. Der wesentliche Unterschied zum dem in Abschnitt 12.3.2 ausführlich vorgestellten Phong-Shader besteht darin, dass die Normalenvektoren für jedes Pixel zur Berechnung der diffusen und spekularen Lichtanteile nicht durch lineare Interpolation der Vertex-Normalen gewonnen werden, sondern dass sie einfach aus der Normalen-Textur stammen. Alle anderen Programmteile können fast unverändert übernommen werden. Deshalb wird im Folgenden nur noch das Vertex- und Fragment-Programm für die Relief-Texturierung vorgestellt. Die Ergebnisse der Relief-Texturierung sind in den Bildern 13.23-e,f für verschiedene Positionen der Lichtquelle dargestellt.

Das Vertex-Programm vsBump.cg:

Das Vertex-Programm vsBump.cg weist gegenüber dem Phong-Shading Vertex-Programm vsPhong.cg (Abschnitt 12.3.2) nur eine geringfügige Änderung auf: anstatt eines Vertex-Normalenvektors wird dem Vertex-Programm ein Satz Texturkoordinaten für den Vertex übergeben. Denn bei der Relief-Texturierung stammt der Pixel-Normalenvektor ja aus der Normalen-Textur (*Normal Map*), die vom OpenGL-Hauptprogramm dem Fragment-Programm bereitgestellt werden muss, und nicht, wie beim Phong-Shading, aus der linearen Interpolation des Vertex-Normalenvektors. Die Texturkoordinaten des Vertex' werden nur zum Rasterizer durchgereicht, der durch lineare Interpolation die richtigen Texturkoordinaten für jedes Fragment berechnet.

```
void main(          float4 position : POSITION,
                    float2 texCoord : TEXCOORD0,

                out float4 oPosition : POSITION,
                out float3 objectPos : TEXCOORD0,
                out float2 oTexCoord : TEXCOORD1,

           uniform float4x4 ModelView,
           uniform float4x4 ModelViewProj)
{
    // Vertex aus Objekt- in Projektionskoordinaten
```

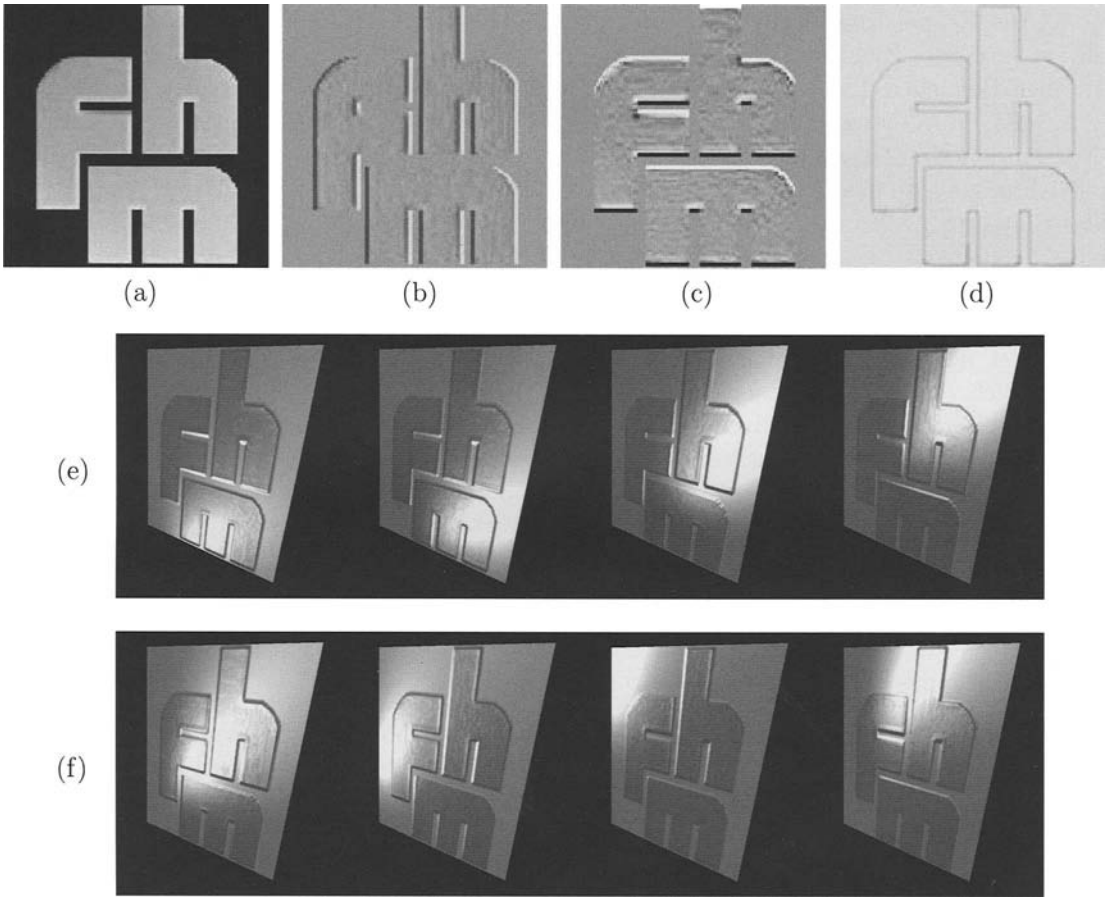


Bild 13.23: Relief-Texturierung (*Bump Mapping*): (a) Die Originaltextur: das Logo der Fachhochschule München (FHM). (b,c,d) Die aus der Originaltextur abgeleitete Normalen-Textur: der Rotauszug (b) entspricht der n_x-Komponente, der Grünauszug (c) entspricht der n_y-Komponente, der Blauauszug (d) entspricht der n_z-Komponente. (e,f) Das FHM-Logo mit Specular Bump Mapping und unterschiedlichen Positionen der lokalen Lichtquelle. Quelle: Gerhard Lorenz

```
    oPosition = mul(ModelViewProj, position);
    // Vertex aus Objekt- in Weltkoordinaten
    objectPos = mul(ModelView, position).xyz;
    // Weitergabe der Texturkoordinate
    oTexCoord = texCoord;
}
```

Das Fragment-Programm fsBump.cg:

Im Wesentlichen gibt es nur die beiden folgenden Änderungen im Fragment-Programm fsBump.cg gegenüber dem Phong-Shading Fragment-Programm fsPhong.cg. Für ausführlichere Erläuterungen der unveränderten Programmteile wird deshalb auf Abschnitt 12.3.2 verwiesen.

- Aufgrund der höheren Geschwindigkeit benutzt man zur Normierung von Vektoren nicht die `normalize`-Funktion, sondern eine „*Normalization Cube Map*". Auf diese spezielle kubische Textur kann man mit einem beliebig langen Vektor zugreifen und erhält als Ergebnis die normierte Variante des Eingabe-Vektors zurück. Die Komponenten dieses normierten Vektors liegen aber, wie für Texturen üblich, im Wertebereich $[0, 1]$. Deshalb werden die Komponenten in den für Vektoren gewünschten Wertebereich $[-1, +1]$ mit Hilfe der vorab definierten `expand`-Funktion transformiert.

- Der Normalenvektor des Fragments stammt aus der Relief-Textur (`normalMap`). Mit Hilfe der vorab definierten `expand`-Funktion werden die Normalenvektoren aus dem Wertebereich $[0, 1]$ nach $[-1, +1]$ transformiert. Da die Normalenvektoren aus der Relief-Textur in Objektkoordinaten definiert sind, müssen sie durch die invers transponierte ModelView-Matrix ins Weltkoordinatensystem transformiert werden. Der resultierende Normalenvektor wird abschließend noch auf die im vorigen Punkt beschriebene Weise normiert.

```
float3 expand(float3 v) = { return (v - 0.5) * 2; }

void main(          float3 position : TEXCOORD0,
                    float3 normalMapTexCoord : TEXCOORD1,
                out float4 color : COLOR,
          uniform float4 e_mat,
          uniform float4 a_mat,
          uniform float4 d_mat,
          uniform float4 s_mat,
          uniform float S,
          uniform float4 pos_light,
          uniform float4 a_light,
          uniform float4 d_light,
          uniform float4 s_light,
          uniform sampler2D normalMap,
          uniform samplerCUBE normalizeCube,
          uniform float4x4 ModelViewInvT)
```

```
{
    // Hole den Normalenvektor aus der Normal Map
    float3 Nobjtex = tex2D(normalMap, normalMapTexCoord).xyz;
    // Transformiere die Werte von [0,1] nach [-1,+1]
    float3 Nobj = expand(Nobjtex);
    // Pixel-Normale aus Objekt- in Weltkoordinaten
    float3 Nworld = mul(ModelViewInvT, Nobj).xyz;
    // Normierung
    float3 Nworldtex = texCUBE(normalizeCube, Nworld).xyz;
    float3 N = expand(Nworldtex);

    // Berechne den emissiven Term
    float4 emissive = e_mat;

    // Berechne den ambienten Term
    float4 ambient = a_light * a_mat;

    // Berechne den diffusen Term
    if(pos_light.w == 0) {
        float3 L = pos_light.xyz;
    } else {
        float3 Ltex = texCUBE(normalizeCube, pos_light.xyz - position).xyz;
        float3 L = expand(Ltex);
    }
    float diffuseLight = max(dot(L, N), 0);
    float4 diffuse = diffuseLight * d_light * d_mat;

    // Berechne den spekularen Term
    // Der Vektor zum Augenpunkt zeigt immer in z-Richtung
    float4 A = float4(0.0, 0.0, 1.0, 0.0);
    // Berechne den Halfway-Vektor
    float3 Htex = texCUBE(normalizeCube, L + A.xyz).xyz;
    float3 H = expand(Htex);
    float specularLight = pow(max(dot(H, N), 0), S);
    if (diffuseLight <= 0) specularLight = 0;
    float4 specular = specularLight * s_light * s_mat;

    color.xyz = emissive + ambient + diffuse + specular;
    color.w = 1;
}
```

Die schönen Ergebnisse der Relief-Texturierung waren in dem vorgestellten Beispiel relativ einfach zu erzielen. So einfach ist die Sache aber nur deshalb, weil die Normalen-Textur auf eine ebene Oberfläche gemappt wird, bei der die Normalenvektoren einheitlich den Wert $(0, 0, 1)^T$ besitzen. Bei einer gekrümmten Oberfläche wird die Angelegenheit aber sofort sehr viel komplizierter. Hier sind zusätzliche Transformationen des Lichtvektors und des Halfway-Vektors in das Texturkoordinatensystem notwendig. Weitere Schwierigkeiten treten auf, wenn beim Mapping der Normalen-Textur auf die Geometrie Verzerrungen berücksichtigt werden müssen, oder wenn Normalen-Texturen im Sinne von Gauß-Pyramiden tiefpassgefiltert und stark verkleinert werden sollen. Für eine ausführliche Diskussion dieser Probleme und der entsprechenden Lösungsansätze wird auf ([Aken02]) und ([Fern03]) verwiesen.

13.6 Schatten-Texturierung (Shadow Mapping)

In der interaktiven 3D-Computergrafik wird bei der Beleuchtungsrechnung kein Schattenwurf berücksichtigt (Abschnitt 12.1.3). Der Grund dafür ist, dass der Test, ob ein Lichtstrahl von einer Lichtquelle zu einem Oberflächenpunkt durch ein anderes Objekt unterbrochen wird, der Punkt also im Schatten eines anderen Objekts liegt, sehr rechenaufwändig ist. Mit der enormen Leistungsfähigkeit heutiger Grafikhardware ist es jedoch möglich, einfache Varianten des Schattenwurfs mit Hilfe von Texture Mapping in Echtzeit zu realisieren. Eine Technik ist die Anwendung projektiver Schatten-Texturen (Abschnitt 13.3), die jedoch mit vielen Einschränkungen verbunden ist.

Deshalb ist heutzutage die Standardmethode zur Realisierung von realistischen Schatten bei interaktiven Anwendungen das *Shadow Mapping*, auch *Shadow-Buffer*-Verfahren genannt. Die Idee dabei ist, die Szene zunächst aus dem Blickwinkel der Lichtquelle zu rendern, denn nur was aus der Position der Lichtquelle sichtbar ist, wird auch beleuchtet, alles Andere liegt im Schatten. Der z-Buffer enthält somit die Entfernungen von der Lichtquelle zum jeweils nächstliegenden Oberflächenpunkt der Szene (Bild 13.24-a). Der Inhalt des z-Buffers wird mit dem Befehl `glCopyTexImage2D()` als Schatten-Textur (*Shadow Map*) gespeichert. Nun wird die Szene ein zweites Mal gerendert, und zwar aus der Sicht des Beobachters. In die Beleuchtungsrechnung fließt diesmal die Schatten-Textur ein. Dazu wird der Abstand z_p des Oberflächenpunkts p von der Lichtquelle mit dem entsprechenden z-Wert der Schatten-Textur z_s verglichen (vorher muss der z-Wert noch vom Koordinatensystem der Lichtquelle ins Koordinatensystem des Beobachters projiziert werden). Ist $z_s < z_p$, so liegt der Punkt p bezüglich der Lichtquelle im Schatten (Bild 13.24-b). In die Beleuchtungsrechnung fließen nur die indirekten Anteile (emissiv und ambient) ein, direkte Anteile (diffus und spekular) werden unterdrückt. Ist $z_s = z_p$, so wird der Punkt p von der Lichtquelle direkt beleuchtet und die normale Beleuchtungsrechnung wird durchgeführt (Bild 13.24-c).

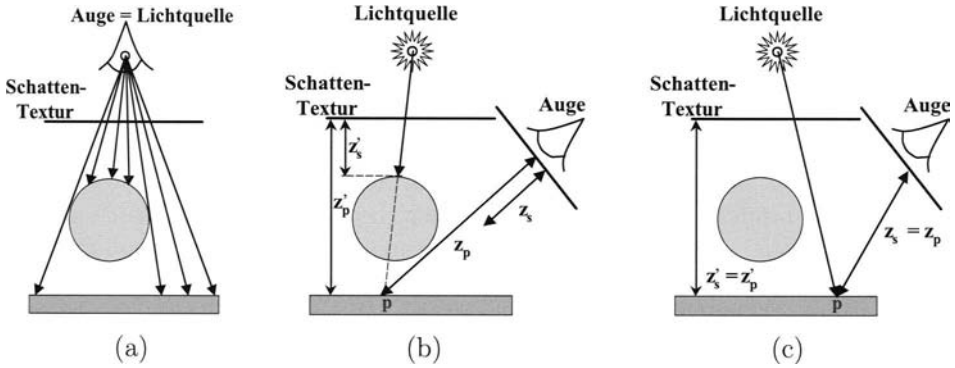


Bild 13.24: Schatten-Texturierung (*Shadow Mapping*): (a) Der z-Buffer enthält nach dem ersten Durchlauf durch die Rendering Pipeline die Entfernungen von der Lichtquelle zum jeweils nächstliegenden Oberflächenpunkt der Szene. Der z-Buffer-Inhalt wird als Schatten-Textur gespeichert. (b) Der Oberflächenpunkt p liegt im Schatten, da $z_s < z_p$. (c) Der Oberflächenpunkt p wird direkt beleuchtet, da $z_s = z_p$.

Für die Realisierung dieses Verfahrens in OpenGL wird aus Platzgründen auf den OpenGL Programming Guide [Shre05] verwiesen. Vorteile dieses Verfahrens sind:

- Es ist auf aktueller Grafikhardware einsetzbar.
- Die Berechnung der Schatten-Textur ist unabhängig von der Augenposition. Sie kann somit bei Animationen, in denen nur der Augenpunkt durch die Szene bewegt wird, wiederverwendet werden.
- Man benötigt nur eine einzige Schatten-Textur für die gesamte Szene und zwar unabhängig davon, wie viele Objekte in der Szene vorhanden sind bzw. wie die Objekte geformt und angeordnet sind.
- Der Aufwand zur Erzeugung der Schatten-Textur ist proportional zur Szenenkomplexität, bleibt aber deutlich niedriger als ein normaler Durchlauf durch die *Rendering Pipeline*, da nur die z-Werte gerendert werden müssen, nicht aber die sehr viel aufwändigeren Farbwerte.

Die Qualität der Schatten hängt jedoch von der Auflösung und Quantisierung der Schatten-Textur sowie von der z-Buffer-Auflösung ab. Der Trend geht im Augenblick dahin, spezielle Hardware für die Schatten-Texturierung zur Verfügung zu stellen.

Kapitel 14

Animationen

14.1 Animation und Double Buffering

In den bisherigen Kapiteln zur 3D-Computergrafik fehlte ein wesentliches Element: Bewegung. Um den Bildern das „Laufen“ beizubringen, d.h. um einen kontinuierlichen Bewegungseindruck zu erzielen, sind mindestens 24 Bilder/Sekunde nötig. Ab dieser Bildgenerierrate nimmt der Mensch keine Einzelbilder mehr wahr, in denen die Objekte oder die ganze Szene stückweise verschoben sind, sondern es entsteht der Eindruck einer flüssigen Bewegung. Wie in Kapitel 3 „Interaktive 3D-Computergrafik“ dargestellt, erfordern manche Anwendungen mindestens 60 Bilder/Sekunde, damit eine akzeptable Bildqualität erreicht wird. Da zwei aufeinander folgende Bilder sich bei diesen Bildgenerierraten in der Regel nur wenig unterscheiden, könnte man auf die Idee kommen, nicht jedes Bild vollkommen neu zu zeichnen, sondern nur die Teile, die sich verändert haben. In sehr einfachen Szenen, wie in Bild 14.1, bei denen sich nur wenige Objekte bewegen, wäre ein solches Vorgehen durchaus möglich. Dabei müssten die Objekte in ihrer neuen Position gezeichnet werden, und die alte Position des Objektes müsste mit dem korrekten Hintergrund übermalt werden. In natürlichen Szenen mit Objektbewegungen, einer Bewegung des Augenpunkts durch die Szene und evtl. noch bewegten Lichtquellen müssen aber in der Regel mehr als 50% des Bildes neu gezeichnet werden. In diesem Fall wäre der Aufwand, die alten Objektpositionen mit dem korrekten Hintergrund zu übermalen, unverhältnismäßig groß.

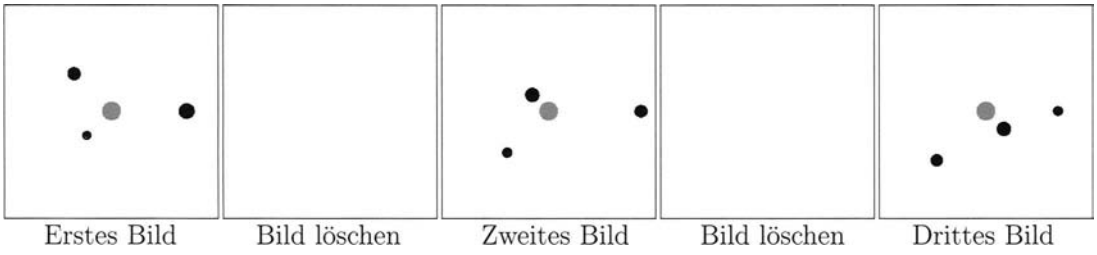


Bild 14.1: Animation und das Flicker-Problem, das durch das Löschen des Bildes verursacht wird.

Es ist sehr viel einfacher, die gesamte Szene jedesmal von Anfang an neu zu zeichnen. Dazu wird zunächst ein „sauberes Blatt" benötigt, d.h. der Bildspeicher wird durch den Befehl `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` mit der eingestellten *clear color* bzw. *clear depth* gelöscht. Anschließend wird die Szene mit den neuen Positionen gezeichnet. Das Problem mit dieser Technik ist, dass das Auge die abwechselnden Lösch- und Zeichenvorgänge bemerkt. Dies führt zum sogenannten *Flicker-Effekt*, der äußert störend wirkt. Die Lösung des Flicker-Problems liegt wieder in einem Stück zusätzlicher Hardware: dem *Double Buffer*. Der Anteil des Bildspeichers (*frame buffer*), der die endgültigen Farbwerte für jedes Pixel enthält, der sogenannte *color buffer*, wird einfach verdoppelt (Bild 14.2).

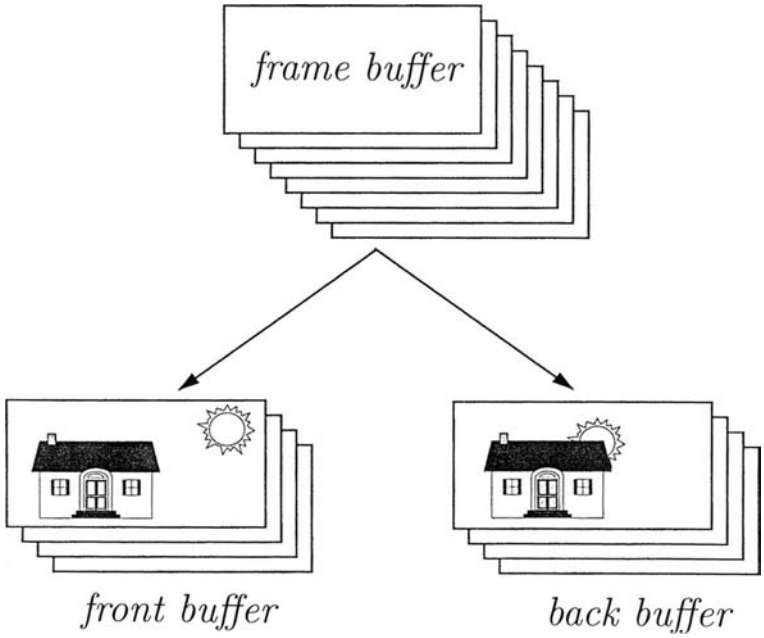


Bild 14.2: *Double Buffering*: Aufteilung des Bildspeichers (*frame buffer*) in einen *front buffer*, dessen Inhalt auf dem Bildschirm dargestellt wird, und in einen *back buffer*, in dem alle Zeichenvorgänge unbemerkt vorgenommen werden.

Die Lösch- und Zeichenvorgänge werden im „hinteren" Teilspeicher (*back buffer*) vorgenommen, der gerade nicht am Bildschirm dargestellt wird. Währenddessen wird der „vordere" Teilspeicher (*front buffer*) ausgelesen, das digitale Bildsignal wird über einen Digital-Analog-Konverter in ein analoges Videosignal umgewandelt und z.B. mit einem Bildschirm zur Ansicht gebracht. Wenn das neue Bild im *back buffer* fertig gezeichnet ist, wird die Rolle der Speicherbereiche einfach vertauscht: der bisherige *back buffer* wird zum neuen *front buffer* und umgekehrt. Dabei werden keine Bilddaten hin- und her kopiert,

sondern es werden nur die entsprechenden Zeiger auf die jeweiligen Bildspeicherbereiche vertauscht. Danach beginnt der Ablauf wieder von vorne (Bild 14.3).

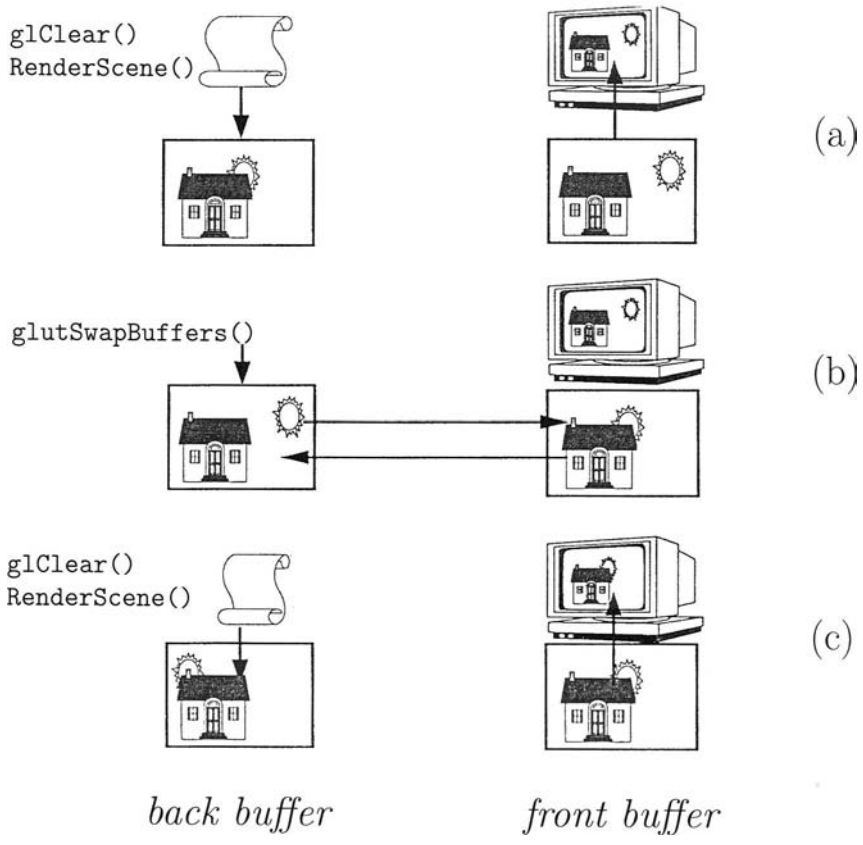


Bild 14.3: Das Zusammenspiel von *front buffer* und *back buffer* zur flickerfreien Darstellung von Animationen. (a) während im front buffer noch ein früher gerendertes Bild steht, bei dem die Sonne noch rechts neben dem Haus ist, wird der back buffer gelöscht und anschließend neu beschrieben, wobei zwischenzeitlich die Sonne ein Stückchen nach links hinter die rechte Seite des Hauses gewandert ist. (b) durch den Befehl `glutSwapBuffers()` werden die Inhalte von front buffer und back buffer vertauscht, so dass das soeben gerenderte Bild jetzt im front buffer steht. (c) das Spiel beginnt wieder von vorne, d.h. während am Bildschirm noch das vorher gerendertes Bild dargestellt wird, bei dem sich die Sonne noch hinter der rechten Seite des Hauses befindet, wird der back buffer gelöscht und danach neu beschrieben, wobei die Sonne jetzt bereits hinter die linke Seite des Hauses gewandert ist.

Das *Double Buffering* zur Lösung des Flicker-Problems ist im Algorithmus **A14.1** zusammengefasst.

A14.1: Pseudo–Code für das *Double Buffering*.

Voraussetzungen und Bemerkungen:

◇ zusätzlicher Speicherplatz für die Verdoppelung des color buffer wird bei der Initialisierung zur Verfügung gestellt. Zugehöriger Befehl aus der GLUT-Library: `glutInitDisplayMode(GLUT_DOUBLE)`

Algorithmus:

(a) Durchlaufe die Hauptprogramm-Schleife immer wieder, bis das Programm beendet wird. (`glutMainLoop()`).

(aa) Initialisiere den *back buffer* mit der eingestellten *clear color* bzw. *clear depth*. (`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`).

(ab) Zeichne die gesamte Szene mit den neuen Positionen (`RenderScene()`).

(ac) Vertausche *front buffer* und *back buffer* (`glutSwapBuffers()`).

Ende des Algorithmus

Die Lösung des Flicker-Problems durch das *Double Buffering* ist zwar sehr elegant, bringt jedoch den Nachteil mit sich, dass eine zusätzliche Transport-Verzögerung von einem Zeittakt entsteht. Denn das neu in den *back buffer* gerenderte Bild erscheint nicht unmittelbar am Bildschirm, sondern es dauert einen Zeittakt, bis *front buffer* und *back buffer* vertauscht werden und somit das neue Bild am Bildschirm dargestellt wird. In sehr zeitkritischen Anwendungen, wie z.B. Trainingssimulatoren (Kapitel 3), ist dies ein unangenehmer, aber unvermeidlicher Nachteil. Um diesen Nachteil zumindest einzudämmen, sollte die Bildgenerierrate möglichst hoch gewählt werden, so dass ein Zeittakt, der ja reziprok zur Bildgenerierrate ist, verhältnismäßig klein bleibt.

Durch das *Double Buffering* alleine entsteht aber noch keine Bewegung in einer Szene. Es verhindert nur den Flicker-Effekt am Bildschirm, wenn sich Objekte in einer Szene bewegen. Um Objekte zu bewegen, d.h. um sie zu animieren, muss die Funktion `RenderScene()`, die der Positionierung aller Vertices dient, regelmäßig mit neuen Werten aufgerufen werden. Die Häufigkeit des Aufrufs kann z.B. mit der GLUT-Funktion `glutTimerFunc(msecs, TimerFunc, value)` spezifiziert werden. Der Parameter `msecs` vom Typ `GLuint` legt fest, nach wie vielen Millisekunden jeweils die Funktion `TimerFunc` aufgerufen wird (der `value`-Parameter dient der Auswahl unterschiedlicher Timer-Funktionen). Die Funktion `TimerFunc` kann z.B. folgendermaßen aussehen:

```
static GLfloat t = 0.0;

void TimerFunc(int value) {
       t += 1.0;
       glutPostRedisplay();
       glutTimerFunc(20, TimerFunc, 1);
}
```

Zunächst wird eine globale Variable `t` definiert, die in der Funktion `RenderScene()` zur Positionsänderung benützt wird. Bei jedem Aufruf der Funktion `TimerFunc` wird die Variable `t` um eins erhöht, durch den GLUT-Befehl `glutPostRedisplay()` wird ein Flag gesetzt, das in der Hauptprogramm-Schleife den Aufruf der Funktion `RenderScene()` auslöst, und zum Schluss wird nach *20 Millisekunden* die rekursive Funktion `TimerFunc` erneut aufgerufen. Falls die Rechenleistung für die definierte Szene ausreichend ist, wird dadurch eine Bildgenerierrate von *50 Hz* erzeugt. Falls nicht, wird die Bildgenerierrate entsprechend niedriger ausfallen. Solange die Rechenleistung ausreicht, lassen sich mit dieser Konstruktion beliebige Animationen erzeugen, deren Ablaufgeschwindigkeit unabhängig von der eingesetzten Hardware ist.

14.2 Animationstechniken

Unter Animation versteht man in der Computergrafik jegliche Veränderungen einer Szene mit der Zeit, und zwar unabhängig davon, wodurch die Veränderung hervorgerufen wurde. Der häufigste Fall ist, dass sich Objekte, der Augenpunkt oder die Lichtquellen bewegen. Aber auch die Veränderung der Gestalt, der Materialeigenschaften und der Texturen von Oberflächen, sowie Änderungen in den Lichtquelleneigenschaften wie Öffnungswinkel oder ausgestrahltes Farbspektrum zählen im weitesten Sinne zu den Animationen.

Zur Bewegung von Objekten benötigt man eine Bahnkurve im 3-dimensionalen Raum, die entweder durch kontinierliche Funktionen beschrieben sein kann, oder durch diskrete Abtastpunkte, zwischen denen interpoliert wird. Außerdem werden bei den Animationstechniken mehrere Hierarchieebenen unterschieden: Bewegungen von starren oder zumindest fest verbundenen Objekten entlang einer räumlichen Bahn werden als *Pfadanimation* bezeichnet. Besitzt ein Objekt innere Freiheitsgrade, wie z.B. Gelenke, spricht man bei deren Bewegung von einer *Artikulation.* Ist ein Objekt auch noch elastisch oder plastisch verformbar, bezeichnet man dies als *Morphing.* Gehorcht eine ganze Gruppe von Objekten ähnlichen Bewegungsgleichungen, die sich nur durch eine Zufallskomponente unterscheiden, hat man es mit Partikelsystemen oder Schwärmen zu tun. Die verschiedenen Hierarchieebenen der Animation können sich selbstverständlich auch noch überlagern, wie z.B. in einer Szene, in der eine Gruppe von Personen in ein Schwimmbecken springt.

Müsste man zur Beschreibung des Bewegungsablaufs einer solch komplexen Szene für jeden Freiheitsgrad und für jeden Zeitschritt eine komplizierte *kontinuierliche* Funktion berechnen, entstünde ein gigantischer Rechenaufwand. Deshalb hat sich hier schon sehr früh, d.h. seit den Anfangszeiten des Trickfilms, die *diskrete* Abtastung und Speicherung der Sze-

ne zu ausgewählten Zeitpunkten eingebürgert. Diese sogenannten *Key Frames* (Schlüsselszenen) wurden bei der Produktion von Trickfilmen immer zuerst gezeichnet, und anschließend wurden die Zwischenbilder durch Interpolation aufgefüllt.

Je nach Anwendung wird in der 3D-Computergrafik sowohl die die kontinuierliche als auch die diskrete Art der Animationsbeschreibung in den verschiedenen Hierachiestufen eingesetzt, wie im Folgenden dargestellt.

14.2.1 Bewegung eines starren Objektes – Pfadanimation

Für die Bewegung eines Gesamtobjekts benötigt man eine Parameterdarstellung der Bahnkurve $\mathbf{K}(t)$ im 3-dimensionalen Raum:

$$\mathbf{K}(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix} \tag{14.1}$$

So lässt sich z.B. eine geradlinige Bewegung in x-Richtung mit konstanter Geschwindigkeit v darstellen als:

$$\mathbf{K}(t) = \begin{pmatrix} x(t) = v \cdot t \\ y(t) = 0 \\ z(t) = 0 \end{pmatrix} \tag{14.2}$$

Eine Kreisbewegung in der $x - z$-Ebene mit dem Radius R und konstanter Winkelgeschwindigkeit ω, wie z.B. bei Einem der drei Satelliten in Bild 14.1, lässt sich darstellen als:

$$\mathbf{K}(t) = \begin{pmatrix} x(t) = R \cdot \cos(\omega \cdot t) \\ y(t) = 0 \\ z(t) = R \cdot \sin(\omega \cdot t) \end{pmatrix} \tag{14.3}$$

Durch das aneinander Setzen von Geradenstücken und Kreisbögen kann man praktisch beliebige Bahnkurven approximieren. Die Echtzeit-Computergrafiksoftware „*OpenGL Performer*“ der Firma SGI bietet z.B. genau diese beiden grundlegenden Kurvenformen zur Pfadanimation von Objekten an [Ecke04]. In der Computergrafik, in der nur die Bilder von Objekten bewegt werden, reicht diese rein kinematische Betrachtung von *kontinuierlichen* Bahnkurven aus. In der physikalischen Realität, in der Objekte massebehaftet sind und folglich zur Bewegungsänderung Kräfte aufzubringen sind, wird die Bahnkurve aus einem Kräftegleichgewicht hergeleitet. Will man physikalisch vernüftige Bewegungen simulieren, darf man also nicht beliebige Bahnen programmieren, sondern nur solche, die sich aus der Newton'schen Mechanik (d.h. der Dynamik) ableiten lassen. Eine Lokomotive, die auf einem Schienennetz fahren würde, das ausschließlich aus Geradenstücken und Kreisbögen bestünde, könnte leicht ins Wanken geraten, denn beim Übergang von einer Geraden zu einem Kreisbogen würde schlagartig eine seitliche Kraft einsetzen, die sogenannte

Zentripetalkraft. Um ein sanftes Ansteigen der seitlichen Kraft von Null auf den Maximalwert, der durch den Kreisradius gegeben ist, sicherzustellen, werden bei der Konstruktion von Schienennetzen Übergangsstücke zwischen Geraden und Kreisbögen eingebaut, deren Krümmung linear mit der Wegstrecke ansteigt. Da die Krümmung direkt proportional zur Zentripetalkraft ist, steigt diese ebenfalls linear mit der Wegstrecke an. Mathematisch formuliert ergibt die Forderung nach einer linear mit der Wegstrecke ansteigenden Krümmung als Bahnkurve eine sogenannte *Klothoide*:

$$\mathbf{K}(t) = \begin{pmatrix} x(t) = a\sqrt{\pi}\int_0^t \cos(\frac{\pi u^2}{2})du \\ y(t) = a\sqrt{\pi}\int_0^t \sin(\frac{\pi u^2}{2})du \\ z(t) = 0 \end{pmatrix} \tag{14.4}$$

Bei der Klothoide steckt die Variable t in der Obergrenze des Integrals, da es keine analytische Darstellung der Kurve gibt. Die Funktionswerte $x(t)$ und $y(t)$ können somit nur numerisch berechnet werden. Weil die numerische Lösung der Integrale in (14.4) für Echtzeit-Anwendungen aber zu lange dauern würde, tabelliert man die Kurve im erforderlichen Wertebereich, d.h. man berechnet die Funktionswerte für äquidistante Abstände der Variablen t und schreibt sie in eine Wertetabelle. Dadurch ist die eigentlich kontinuierliche Kurve der Klothoide aber nur noch an *diskreten* Abtastpunkten gegeben. Werden Funktionswerte zwischen den gegebenen Abtastpunkten benötigt, wird eine Interpolation der Tabellenwerte durchgeführt. Am häufigsten verwendet wird dabei die lineare Interpolation, sowie die quadratische und die Spline-Interpolation.

Da komplexe Raumkurven, wie z.B. Klothoiden, sowieso nur an diskreten Abtastpunkten gegeben sind, kann man die Darstellungweise auch gleich ganz umkehren und die gesamte Raumkurve, unabhängig vom Kurventyp, durch n diskrete Abtastpunkte definieren:

$$\mathbf{K} = \begin{pmatrix} \mathbf{x} = (x_0, x_1, \ldots, x_n) \\ \mathbf{y} = (y_0, y_1, \ldots, y_n) \\ \mathbf{z} = (z_0, z_1, \ldots, z_n) \end{pmatrix} \tag{14.5}$$

Zwischenwerte werden wieder durch Interpolation gewonnen. Damit ist man bei der eingangs erwähnten *Key Frame Technik* angelangt, in der die animierten Objekte zu diskreten Zeitpunkten abgetastet und gespeichert werden.

Bisher wurden nur die drei translatorischen Freiheitsgrade bei der Bewegung von Objekten im 3-dimensionalen Raum betrachtet. Im Allgemeinen besitzt ein Objekt aber auch noch drei rotatorische Freiheitsgrade der Bewegung, d.h. die Drehwinkel um die drei Raumachsen x, y, z, die als Nickwinkel p (*pitch*), Gierwinkel h (*heading* oder *yaw*) und Rollwinkel r (*roll*) bezeichnet werden.

Ein Animationspfad für das Beispiel der Bewegung auf einem Schienennetz kann in der kontinuierlichen Darstellung durch eine Sequenz von Kurventypen (Geraden, Kreisbögen, Klothoiden etc.) sowie der zugehörigen Parameter (Start- und Endpunkt bzw. -winkel,

Krümmung etc.) beschrieben werden. In der diskreten Darstellung ist der Pfad durch eine evtl. längere Tabelle der Form

$$\mathbf{K} = \begin{pmatrix} \mathbf{x} = (x_0, x_1, \ldots, x_n) \\ \mathbf{y} = (y_0, y_1, \ldots, y_n) \\ \mathbf{z} = (z_0, z_1, \ldots, z_n) \\ \mathbf{h} = (h_0, h_1, \ldots, h_n) \\ \mathbf{p} = (p_0, p_1, \ldots, p_n) \\ \mathbf{r} = (r_0, r_1, \ldots, r_n) \end{pmatrix} \tag{14.6}$$

gegeben. Der Vorteil der kontinuierlichen Darstellung liegt meist im kleineren Speicherplatzbedarf, der Nachteil im größeren Rechenaufwand, der für die Bestimmung einer beliebigen Position und Lage auf dem Pfad benötigt wird. Deshalb wird in der interaktiven 3D-Computergrafik häufig die diskrete Variante der Pfadanimation benützt.

14.2.2 Bewegung von Gelenken eines Objektes – Artikulation

Die nächste Komplexitätsstufe der Computeranimation ist erreicht, wenn nicht nur starre Objekte in einer Szene bewegt werden, sondern Objekte mit inneren Freiheitsgraden, wie z.B. Personen- oder Tiermodelle mit Gelenken. Die mit Gelenken verbundenen Teile des Objekts sind aber starr. Das einfache Personenmodell in Bild 14.4 besitzt zehn Gelenke, zwei Hüftgelenke, zwei Kniegelenke, zwei Fußgelenke, zwei Schultergelenke und zwei Ellenbogengelenke. Die Gelenkwinkel werden durch einen Satz von fünf Kurven gesteuert, wobei jede einzelne Kurve einen Gelenkwinkel über der Zeit darstellt. Die Gelenkwinkel der rechten Extremitäten werden alle zum gleichen Zeitpunkt bestimmt, die der linken Extremitäten werden um eine halbe Periodendauer versetzt abgelesen. Dadurch kann man bei achsensymmetrischen Modellen die Hälfte der Kurven einsparen. Durch das Laden unterschiedlicher Kurvengruppen lässt sich zwischen verschiedenen Bewegungstypen, wie z.B. Gehen oder Laufen, umschalten. Außerdem hat man damit die nächsthöhere Abstraktionsstufe der Animationsbeschreibung erreicht, denn statt jeden einzelnen Gelenkwinkel zu jedem Zeitpunkt vorzugeben, braucht nur noch ein einziger Begriff angegeben zu werden, um das entsprechende Bewegungsverhalten zu erzielen.

Anstatt kontinuierlicher Kurven, wie in Bild 14.4 dargestellt, kann natürlich auch bei der Artikulation die diskrete *Key Frame*–Technik verwendet werden. Dazu werden „Schnappschüsse" des Modells zu charakteristischen Zeitpunkten geschossen, wie z.B. bei den größten Auslenkungen und den Nulldurchgängen der Gelenke. Bei einer geringen Zahl von Abtastpunkten werden nichtlineare Interpolationsverfahren verwendet, um zu Zwischenwerten zu gelangen, bei einer größeren Zahl an Abtastpunkten genügt die lineare Interpolation.

Realistischere Personenmodelle, wie sie z.B. in der Filmproduktion eingesetzt werden, besitzen zwei- bis dreihundert Gelenke. Hier ist es nahezu unmöglich, im *try-and-error*-Verfahren hunderte aufeinander abgestimmter Kurven für die Gelenkbewegungen vorzugeben, so dass ein glaubwürdiger Bewegungsablauf entsteht. Deshalb haben sich für sol-

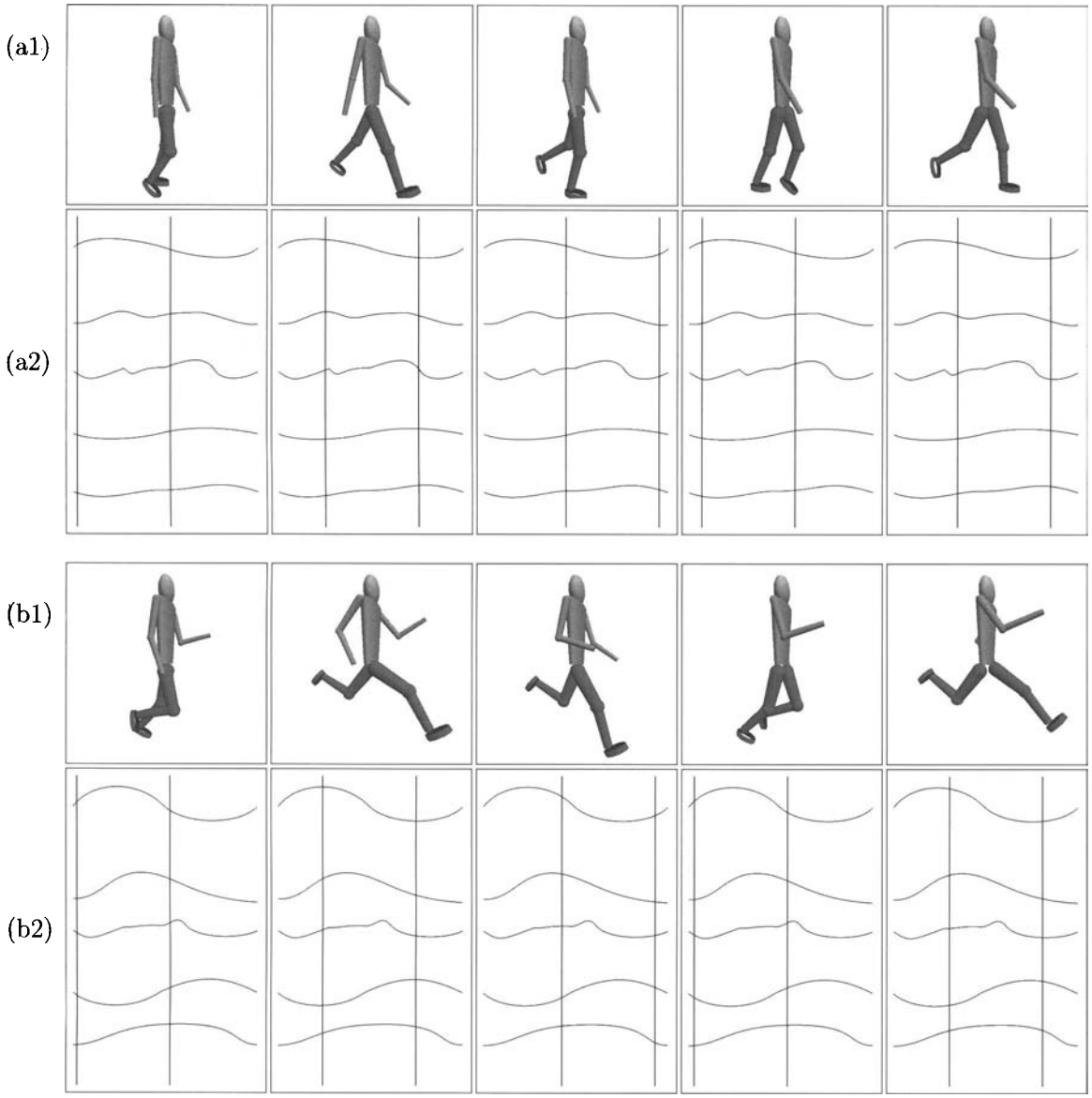


Bild 14.4: *Artikulation*: Bewegung von Gelenken eines Objekts, dessen Teile starr sind. Das Personenmodell besitzt zehn Gelenke (Hüften, Knie, Füße, Schultern, Ellenbogen). Die Gelenkwinkel werden durch fünf Kurven gesteuert, wobei jede Kurve einen Gelenkwinkel über der Zeit darstellt. Der erste senkrechte Strich markiert die Gelenkwinkel der rechten Extremitäten und der zweite, um 180 Grad versetzte Strich, die der linken Extremitäten. Durch das Laden unterschiedlicher Kurvengruppen lässt sich zwischen verschiedenen Bewegungstypen, wie z.B. Gehen oder Laufen, umschalten. (a1) Bewegungssequenz „Gehen". (a2) Zugehörige Kurven, aus denen die Gelenkwinkel bestimmt werden. (b1) Bewegungssequenz „Laufen". (b2) Zugehörige Kurven. Das Programm stammt von Philip Wilson.

che Aufgabenstellungen *motion-capturing*–Verfahren [Gins83] durchgesetzt: einem lebenden Modell werden an den relevanten Gelenken Leuchtdioden angebracht und der Bewegungsablauf wird mit Hilfe mehrerer, räumlich verteilter Kameras unter kontrollierten Beleuchtungsverhältnissen aufgezeichnet. Aus den vorab vermessenen Kamerapositionen und der korrekten Zuordnung korrespondierender Leuchtdioden im jeweiligen Kamerabild kann durch Triangulation die Position jeder Leuchtdiode berechnet werden. Aus der zeitlichen Sequenz aller Leuchtdiodenpositionen kann schließlich der Verlauf aller Gelenkwinkel bestimmt werden. Für jeden Bewegungstyp (z.B. gehen, laufen, kriechen usw.) wird ein Satz an Gelenkwinkelkurven abgeleitet. Ein sanfter Übergang zwischen den verschiedenen Bewegungstypen ist wieder durch Interpolation realisierbar: jeder Gelenkwinkel wird zweimal berechnet, einmal mit dem Kurvensatz für den ersten Bewegungstyp und einmal mit dem Kurvensatz für den zweiten Bewegungstyp, daraus wird der endgültige Gelenkwinkel interpoliert. Mit dieser Technik kann ein künstliches Modell extrem realitätsgetreu animiert werden.

Für gezielte Bewegungsabläufe komplexer Modelle sind die bisher beschriebenen Methoden aber nicht gut geeignet. Mit diesen Methoden lässt sich sehr genau vorhersagen, welche Position und welche Gelenkwinkel ein Modell zu einem bestimmten Zeitpunkt besitzt. Für eine gezielte Bewegung, wie z.B. beim Ergreifen eines bestimmten Gegenstandes in einer vorgegebenen Position, ist aber genau die umgekehrte Aufgabe zu lösen, nämlich welcher Bewegungsablauf zur Ergreifung des Gegenstandes führt. Und da es bei einem komplexen Modell mit vielen Gelenken sehr viele Freiheitsgrade gibt, kann das Ziel auf unterschiedlichsten Wegen erreicht werden. Durch Vorgabe zusätzlicher Kriterien, wie z.B. minimaler Energieverbrauch oder minimale Dauer bis zur Zielerreichung, wird daraus ein hochdimensionales Optimierungsproblem, das mit den entsprechenden mathematischen Methoden numerisch gelöst werden kann ([Gira85], [Bend05]). Die Lösung der Aufgabe, für ein vorgegebenes Ziel die optimalen Bewegungsparameter zu finden, wird als *Inverse Kinematik* bezeichnet. Werden auch noch die verursachenden Kräfte, die beteiligten Massen und evtl. die Reibung berücksichtigt, ist man bei der noch komplexeren *Inversen Dynamik* angelangt [Wilh87]. Mit diesen in der Robotik entwickelten Methoden, die in der Computergrafik übernommen wurden, ist man auf einer noch höheren Abstraktionsebene der Beschreibung von Bewegungen angekommen. Denn es ist deutlich einfacher, ein Ziel und ein Optimierungskriterium anzugeben, als alle Gelenkwinkelkurven so aufeinander abzustimmen, dass das gewünschte Verhalten auch zum vorgegebenen Ziel führt.

14.2.3 Verformung von Oberflächen – Morphing

Bei den bisher beschriebenen Animationen waren entweder die ganzen Objekte, oder zumindest alle Teile starr. Wenn sich reale Personen bewegen, drehen sich aber nicht nur die Knochen in ihren Gelenken, sondern Muskeln ziehen sich zusammen und werden dadurch kürzer und dicker, Haut wird gedehnt oder faltet sich zusammen, Haare wehen im Wind usw., d.h. die äußere Hülle der Person verformt sich. Es gibt zahlreiche weitere Beispiele für elastische oder plastische Verformungen (Bild 14.5) bis hin zu Metamorphosen, bei denen z.B. aus einer Raupe ein Schmetterling wird, oder wie in Gruselfilmen, bei denen sich

Menschen langsam in Untiere verwandeln. All diese Verformungen von Oberflächen werden in der Computergrafik unter dem Begriff *Morphing* zusammengefasst.

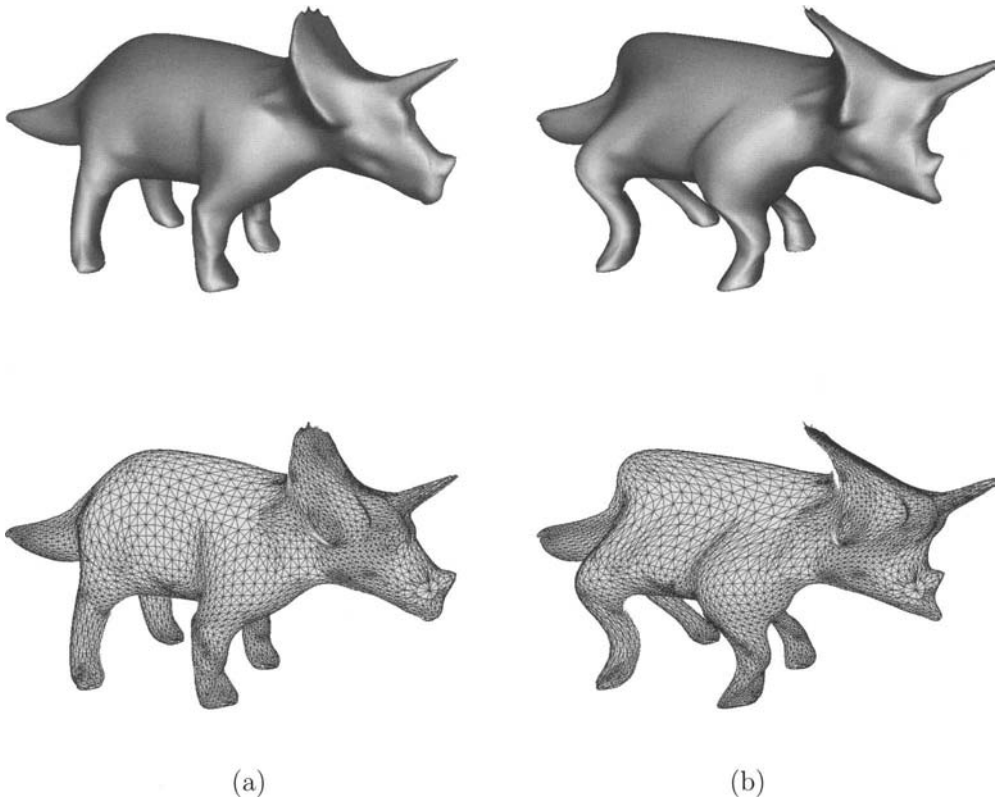


Bild 14.5: *Morphing*: Verformung von Oberflächen am Beispiel des Triceratops-Modells. (a) Das Original-Modell, oben gefüllt, unten als Drahtgitter. (b) Das verformte Modell, oben gefüllt, unten als Drahtgitter.

Eine andere Anwendung, bei der Morphing zum Einsatz kommt, ist das sogenannte level-of-detail (LOD)–Morphing. Bei diesem in Kapitel 15.1 ausführlicher beschriebenen Verfahren wird bei Annäherung an ein Objekt nicht einfach bei einem fixen Abstand schlagartig von einer gröberen zu einer detaillierteren Repräsentation eines Objekts umgeschaltet, sondern es wird innerhalb eines bestimmten Abstandsbereichs ein weicher Übergang von der gröberen zur feineren Tessellierung durch Morphing erreicht. Dadurch kann der störende *pop-up*–Effekt beim Umschalten zwischen verschiedenen LOD-Stufen verringert werden. Alternativ kann bei vergleichbarer Bildqualität der Umschaltabstand verkleinert werden, so dass in Summe weniger Polygone in einer Szene vorhanden sind und die Grafiklast sinkt.

Es ist allerdings zu beachten, dass Morphing um Größenordnungen rechenintensiver ist als die bisher besprochenen Animationstechniken. Dies hat mehrere Gründe: nicht nur wenige Objektpositionen und Gelenkwinkel müssen pro Bild berechnet oder interpoliert werden, sondern zig-tausende Vertexpositionen. Ein weiteres Problem, das erst beim Morphing auftaucht, ist die Thematik zeitveränderlicher Normalenvektoren. Während bei starren Objekten oder Objektteilen die für die Beleuchtungsrechnung (Kapitel 12) erforderlichen Normalenvektoren vorab und für alle Fälle geltend durch Kreuzproduktbildung, Mittelung und Normierung ermittelt werden, müssen bei verformbaren Oberflächen die Normalenvektoren für jeden Vertex und für jedes Bild zur Laufzeit des Programms neu berechnet werden. Selbst wenn die Vertices und Normalenvektoren zwischen zwei *Key Frames* nur linear interpoliert werden, entsteht ein enormer zusätzlicher Rechenaufwand. Mit der neuesten Generation an Grafikhardware, die über programmierbare *Vertex- und Pixel-Shader* verfügt, kann die riesige Menge an Interpolations- und Normalenvektorberechnungen auf die *GPU* (*Graphics Processing Unit* = Grafikkarte) verlagert und somit hardwarebeschleunigt werden, so dass Morphing in Echtzeit realisiert werden kann [Fern03]. Falls auch noch die Bewegungen eines halbwegs realistischen Personenmodells aus Knochen, Fleisch und Blut mit Hilfe von Finite-Elemente-Methoden und Fluid-Dynamikmodellen berechnet werden, ist an eine interaktive Anwendung selbst bei heutiger Rechner- und Grafik-Hardware nicht zu denken.

14.2.4 Bewegung von Objektgruppen: Schwärme und Partikelsysteme

In diesem Abschnitt wird die Animation einer ganzen Gruppe von ähnlichen Objekten betrachtet. Bei einer überschaubaren Anzahl von Objekten, die noch komplexes individuelles Verhalten zeigen, wie z.B. bei einer Menge von Autos im Straßenverkehr, oder einer Gruppe von Zugvögeln, die in wärmere Regionen fliegen, spricht man von *Schwärmen.* Steigt die Zahl der Objekte in einer Gruppe soweit an, dass es unmöglich wird, ein individuelles Verhalten jedes einzelnen zu berücksichtigen, wie z.B. bei Molekülen in einem Gas, handelt es sich um *Partikelsysteme.* Die Grundidee zur Beschreibung von Schwärmen und Partikelsystemen stammt aus der statistischen Physik: bestimmte Eigenschaften der einzelnen Objekte, wie z.B. ihre Geschwindigkeit, werden durch einen Zufallsterm bestimmt, und der Mittelwert über alle Objekte ergibt eine globale Größe wie die Temperatur oder den Druck eines Gases. Damit kann man mit einigen wenigen globalen Einflussparametern das Verhalten der gesamten Gruppe festlegen und muss sich nicht mehr um jedes einzelne Objekt kümmern.

Als Beispiel für einen Schwarm wird die Modellierung des interaktiven Verkehrs in einem Straßenfahrsimulator genauer beschrieben [Bodn98]. Der interaktive Verkehr besteht aus einer Zahl von z.B. 10 bis 100 anderen Verkehrsteilnehmern, die sich in der Sichtweite des Augenpunkts bewegen. Das Verhalten des Fremdverkehrs soll nicht deterministisch sein, um Gewöhnungseffekte bei wiederholtem Training zu vermeiden. Deshalb enthält die Simulation des Fremdverkehrs eine zufällige Komponente, so dass immer wieder

neue Situationen entstehen. Dennoch bewegt sich der Fremdverkehr keineswegs chaotisch, sondern er muss vorgegebene Regeln beachten, wie z.B. Geschwindigkeitsbeschränkungen, Vorfahrtsregeln, Ampelschaltungen und Überholverbote. Jeder autonome Verkehrsteilnehmer berücksichtigt sowohl die anderen Verkehrsteilnehmer als auch das simulierte Eigenfahrzeug. Dadurch wird das Verhalten des autonomen Verkehrs auch durch die Fahrweise des Fahrschülers beeinflusst. Gleichzeitig bekommt der Fahrschüler das Gefühl vermittelt, mit den anderen Verkehrsteilnehmern zu interagieren. Grundlage für die Simulation der autonomen Verkehrsteilnehmer ist eine Datenbasis, in der alle relevanten Informationen enthalten sind: Abmessungen aller verfügbaren Fahrbahnen, Position und Art von Verkehrzeichen, Eigenschaften der Verkehrsteilnehmer wie z.B. Masse, Außenmaße, maximale Beschleunigung bzw. Geschwindigkeit und minimaler Wendekreis. Auf der Basis dieser statischen Informationen, den bekannten Positionen und Geschwindigkeiten aller anderen Verkehrsteilnehmer, sowie den globalen Vorgaben zur Steuerung des autonomen Verkehrs, berechnet ein Dynamikmodul Geschwindigkeit und Position des betrachteten Fahrzeugs. Die wenigen globalen Vorgaben, mit denen der autonome Verkehr gesteuert wird, sind Verkehrsdichte, Aggressivität der autonomen Verkehrsteilnehmer, sowie das gezielte und reproduzierbare Herbeiführen von Gefahrensituationen für den Fahrschüler. Verlässt ein anderer Verkehrsteilnehmer dauerhaft das sichtbare Volumen, wird er gelöscht. Dafür entstehen außerhalb des sichtbaren Volumens auch laufend wieder neue Verkehrsteilnehmer, die später in das Sichtfeld des Fahrschülers kommen, so dass die gewählte Verkehrsdichte in etwa konstant bleibt. Die autonomen Verkehrsteilnehmer verteilen sich also immer in einer Art „Blase“ um den Fahrschüler herum. Die interaktive Simulation eines solch komplexen Schwarms von autonomen Verkehrsteilnehmern erfordert einen eigenen leistungsfähigen Simulationsrechner.

Partikelsysteme werden zur realitätsnahen Beschreibung von Spezialeffekten, wie z.B. Feuer, Explosionen [Reev83], Nebel und Schneefall, oder auch für Grasflächen und Ansammlungen von Bäumen [Reev85] eingesetzt. Die Grundprinzipien sind die gleichen, wie bei der geschilderten Modellierung des interaktiven Verkehrs:

- Neue Partikel können erzeugt und alte Partikel gelöscht werden.
- Alle Partikel besitzen ähnliche Eigenschaften und folgen ähnlichen Verhaltensregeln.
- Neben einem deterministischen regelbasierten Verhaltensanteil existiert auch noch der Einfluss einer Zufallskomponente.
- Ein Dynamikmodul berechnet die Bewegung und ggf. auch noch veränderliche Eigenschaften der Partikel auf der Basis von deterministischen und stochastischen Anteilen der Systemgleichungen sowie der globalen Steuerparameter.
- Die Steuerung des Partikelsystems erfolgt durch Angabe einiger weniger globaler Parameter wie Partikeldichte, Partikelgröße, Mittelwert und Varianz von Geschwindigkeit und Vorzugsrichtung usw.

Ein einfaches Beispiel für ein Partikelsystem ist in Bild 14.6 dargestellt.

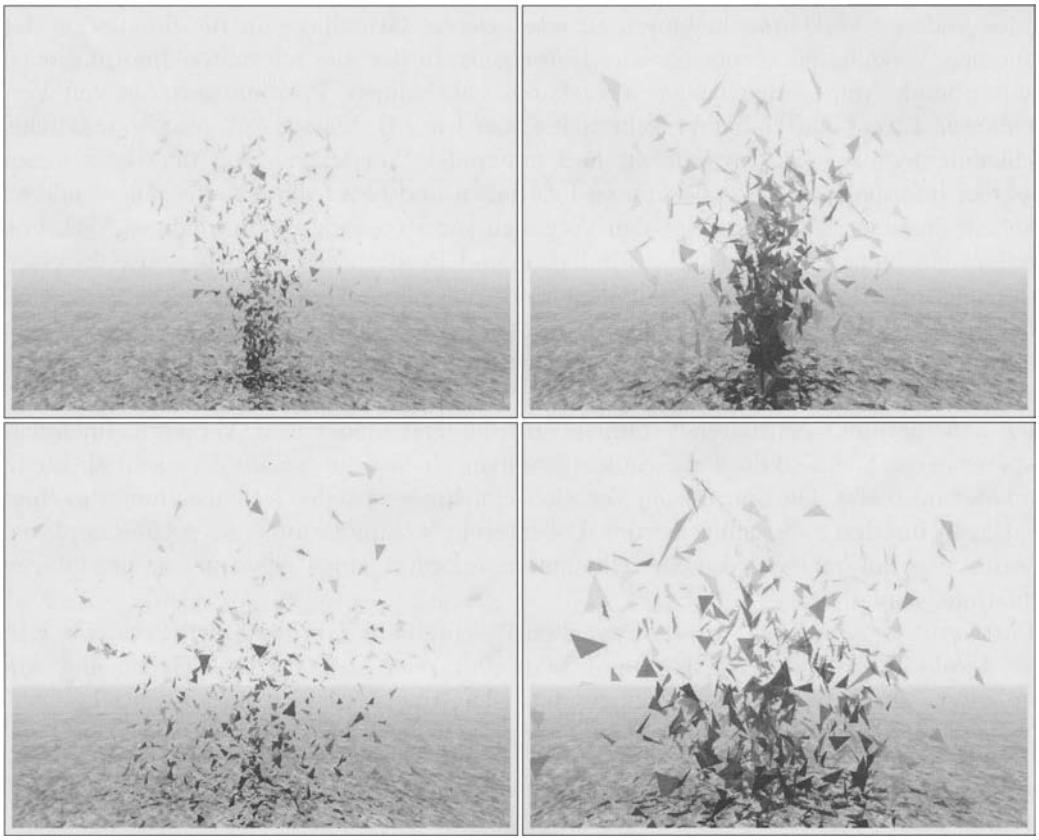


Bild 14.6: *Partikelsysteme*: eine große Zahl ähnlicher Partikel wird durch wenige globale Parameter gesteuert. Vom linken zum rechten Bild erhöht sich jeweils die Größe der Partikel. Vom oberen zum unteren Bild erhöht sich jeweils der Emissionsradius der Partikel. Das Programm stammt von David Bucciarelli.

Kapitel 15

Beschleunigungsverfahren für Echtzeit 3D-Computergrafik

Alle bisher gezeigten Beispiele für interaktive 3D-Computergrafik haben Eines gemeinsam: die Szene besteht aus einem einzigen oder allenfalls wenigen Objekten, die in der Regel vollständig im sichtbaren Volumen (*Viewing Frustum*) enthalten sind. Diese einfachen Szenen sind ideal, um die grundlegenden Prinzipien der Computergrafik anhand von OpenGL darzustellen. Reale Anwendungen, wie sie in Kapitel 4 vorgestellt werden, besitzen im Gegensatz zu den Lehrbeispielen eine sehr viel größere Szenenkomplexität. Ein LKW-Simulator z.B. benötigt als Geländedatenbasis schon mal eine Großstadt und einen Landstrich mit Autobahnen, Landstraßen und Dorfstraßen. Ein Bahn-Simulator benötigt evtl. ein Schienennetz von mindestens 1000 *km* Strecke und ein Flug-Simulator evtl. sogar ein Modell der gesamten Erdoberfläche. Hinzu kommen meist noch eine Vielzahl weiterer Verkehrsteilnehmer, wie Personen, Tiere, Fahrzeuge und Flugzeuge, die selbst wieder eine große Komplexität aufweisen. Für ein normales OpenGL-Programm ergeben sich daraus zwei grundlegende Probleme:

- Es entsteht eine Unmenge an Programmcode, um all die Millionen von Vertices des Geländemodells und der animierbaren Objekte zu spezifizieren.
- Selbst mit der enormen Leistungsfähigkeit heutiger Grafikhardware ist es in der Regel unmöglich, das gesamte Gelände und alle animierbaren Objekte gleichzeitig zu rendern.

Für jedes der beiden Probleme gibt es eine Kategorie von Werkzeugen, die Lösungen anbieten. Zur Erzeugung komplexer 3D-Modelle gibt es zahlreiche Modellierwerkzeuge (*Digital Content Creation Tools (DCC-Tools)*), wie in Abschnitt 6.4 dargestellt. Diese Modellierwerkzeuge gestatten es, über eine grafische Benutzeroberfläche 3D-Szenarien interaktiv zu generieren und abzuspeichern. Zum Darstellen von Szenen mit sehr hoher Komplexität und gleichzeitig hoher Bildgenerierrate gibt es eine Reihe von Echtzeit-Rendering-Werkzeugen, die für diesen Zweck einige grundlegende Funktionalitäten zur Verfügung stellen. Dazu zählen:

- Lader für 3D-Datenformate und Texturen.
  Damit können 3D-Szenarien, die vorher mit einem Modellierwerkzeug generiert und abgespeichert wurden, in die Datenstruktur (den Szenen Graphen) des Echtzeit-Rendering-Werkzeugs übersetzt und in den Hauptspeicher geladen werden.

- Szenen Graph.
  Dies ist eine Datenstruktur, in der die Objekte der virtuellen Welt räumlich gruppiert in einer hierarchischen Baumstruktur angeordnet werden. Diese Datenstruktur beschleunigt andere Algorithmen (Culling, Kollisionserkennung etc.) enorm.

- Culling-Algorithmen.
  Damit wird der sichtbare Teil aus dem gesamten 3D-Szenario ausgeschnitten, so dass nur noch die Objekte gerendert werden müssen, die später auch am Bildschirm erscheinen.

- Level-of-Detail.
  Jedes Objekt wird in unterschiedlichen Detailstufen gespeichert, und mit zunehmender Entfernung zum Augenpunkt wird eine immer niedrigere Detailstufe dargestellt.

- Billboard-Objekte.
  Sehr komplexe Objekte (wie z.B. ein Baum oder eine Person) werden durch eine Foto-Textur ersetzt, die sich immer zum Augenpunkt ausrichtet.

- Unterstützung für Multiprozessorsysteme.
  Die Aufteilung von Rendering-Aufgaben auf mehrere Prozessoren ermöglicht eine Beschleunigung der Bildgenerierung.

- Statistik-Werkzeuge:
  Statistik-Werkzeuge dienen zur Erfassung und Auswertung der Rechenzeiten einzelner Teile der *Rendering Pipeline* und liefern somit die Informationen, an welchen Stellen der *Rendering Pipeline* optimiert werden sollte.

- Routinen für häufig benötigte Aufgaben:
  Dazu zählen z.B. verschiedene Bewegungsmodi (fahren, fliegen, etc.), Kollisionserkennung, Himmel-Modell mit ziehenden Wolken (*Sky Box*), Nachladen von Geländestücken von der Festplatte in den Hauptspeicher während der laufenden Simulation (für sehr große Geländedatenbasen), Spezialeffekte (Feuer, Rauch, kalligrafische Lichtpunkte (extrem hell), Oberflächeneffekte, Partikelsysteme etc.), verteilte Simulationen (DIS/HLA).

Im Folgenden sind einige OpenGL-basierte Echtzeit-Rendering-Werkzeuge aufgelistet, die die wichtigsten der oben genannten Funktionalitäten besitzen:

- „OpenGL Performer“ von SGI.
  Eines der leistungsfähigsten Werkzeuge in diesem Bereich, für die Betriebssysteme IRIX (SGI), Linux und Windows (Microsoft) verfügbar und für private Zwecke kostenlos [Ecke04].

- „Vega Prime“ von Multigen-Paradigm Inc.
  Die ersten Versionen von Vega basierten auf „OpenGL Performer“, Vega Prime setzt direkt auf OpenGL auf und ist für die Betriebssysteme Windows (Microsoft), Linux und Solaris (SUN) verfügbar. Ein sehr leistungsfähiges Werkzeug, das zahlreiche Zusatzmodule bereitstellt (z.B. für Infrarot- und Radarbildgenerierung, Spezialeffekte, verteilte Simulationen).

- „Open Inventor“ von TGS.
  Ursprünglich von SGI entwickelt, Vorläufer von „OpenGL Performer“, Basis für das Internet-3D-Grafikformat VRML97 (Virtual Reality Markup Language) und den Grafikanteil des Codierstandards MPEG-4. Open Inventor enthält im Wesentlichen eine Untermenge der Funktionalitäten von „OpenGL Performer“.

- „VTree/Mantis“ von CG^2/Quantum3D und „GIZMO3D“ von Saab Training Systems.
  Konkurrenzprodukte zu „Vega Prime“ mit ähnlicher Funktionalität.

- „Open Scene Graph“ und „OpenSG“.
  Zwei konkurrierende offene Standards für Echtzeit-Rendering.

OpenGL stellt die Basisfunktionalitäten für interaktive 3D-Computergrafik zur Verfügung und ist somit das Bindeglied zwischen der 3D-Grafikanwendung und der Hardware bzw. dem Betriebssystem. OpenGL stellt somit in einem Schichtenmodell die unterste Grafik-Software-Schicht dar, auf der höherintegrierte Werkzeuge aufsetzen (Bild 15.1).

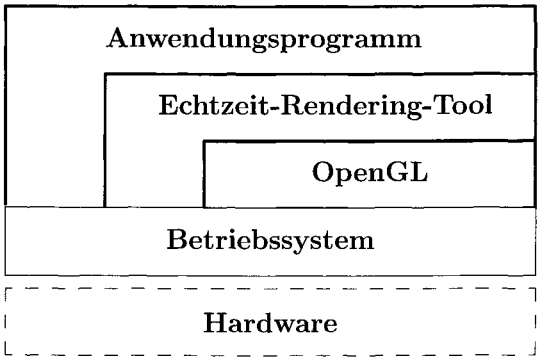


Bild 15.1: Die Software-Schichten einer Echtzeit-3D-Computergrafik-Anwendung.

In den folgenden Abschnitten werden die wesentlichen Beschleunigungsverfahren dargestellt, die in der Echtzeit-3D-Computergrafik Anwendung finden.

15.1 Szenen Graphen

Eine Szene in der 3D-Computergrafik besteht aus der Geometrie aller Objekte, sowie den Zuständen (Materialeigenschaften, Texturen, Transformationen, Nebel, Anti-Aliasing usw.), in denen die Geometrie gerendert werden soll. Für einen schnellen Zugriff auf die Objekte wird die Szene in Form einer hierarchischen Datenstruktur, dem Szenen Graph, organisiert. Ein Szenen Graph besteht aus miteinander verbundenen Knoten (*nodes*), die in einer Baumstruktur angeordnet werden (Bild 15.2). Es gibt einen Wurzelknoten, der der Ursprung des gesamten Baums ist, interne Knoten (quasi die Äste des Baums) und Endknoten (Blätter). Die Endknoten enthalten in der Regel die Geometrie der Objekte, sowie bestimmte Attribute (z.B. Texturen und Materialeigenschaften). Interne Knoten dienen verschiedenen Zwecken, und für jeden Zweck gibt es in der Regel einen eigenen Knotentyp:

- Gruppen-Knoten sind dazu da, eine Ansammlung von Unterknoten zu einer Gruppe zusammenzufassen (z.B. mehrere Häuser zu einem Dorf).
- Switch-Knoten ermöglichen die gezielte Auswahl eines ganz bestimmten Unterknotens (z.B. Bahnschranke offen oder geschlossen).
- Sequence-Knoten durchlaufen in einem bestimmten Zyklus die Unterknoten (z.B. automatische Ampelschaltung oder kurze Film-Sequenz).
- LOD-Knoten schalten je nach Entfernung zum Augenpunkt auf einen der verschiedenen Unterknoten, die das Objekt in den entsprechenden Detailstufen enthalten.
- Transformations-Knoten enthalten eine Transformations-Matrix **M** (Translation, Rotation, Skalierung).

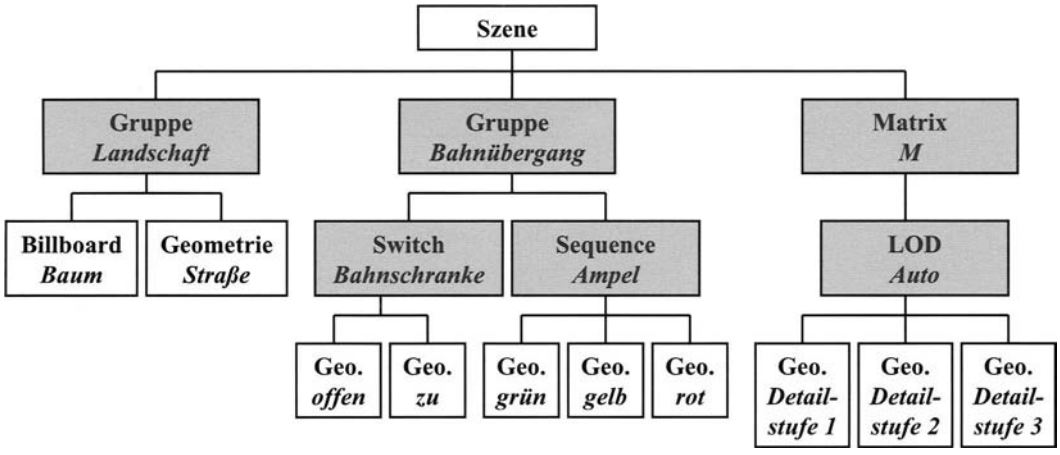


Bild 15.2: Ein Szenen Graph mit verschiedenen internen Knotentypen (grau). Der oberste Knoten ist der Wurzelknoten, die untersten Knoten ohne Nachfolger sind die Endknoten, die die Geometrie der Objekte enthalten.

Einer der wichtigsten Gründe, warum man für komplexe Szenen einen hierarchisch organisierten Szenen Graphen aufbaut, ist, dass damit Culling Algorithmen, die den sichtbaren Teil aus der gesamten visuellen Datenbasis ausschneiden, sehr viel schneller ablaufen als bei einer schlecht organisierten Datenbasis. Dazu wird jedem Knoten eines Szenen Graphen eine Hülle (*bounding volume*) zugewiesen, die die Objekte unterhalb des Knotens möglichst genau umschließt. Die Hülle hat eine sehr viel einfachere geometrische Form als die umschlossenen Objekte, so dass die Frage, ob ein Objekt außerhalb des sichtbaren Volumens ist, sehr schnell entschieden werden kann. Typische geometrische Formen für solche Hüllen sind z.B. Kugeloberflächen (*bounding sphere*), axial ausgerichtete Quader (*axis-aligned bounding box*) oder beliebig orientierte Quader (*oriented bounding box*). Eine wesentliche Voraussetzung für einen effizienten Cull-Algorithmus ist, dass die Hierarchie des Szenen Graphen räumlich organisiert ist. Dazu fasst man die Objekte einer Szene über mehrere Hierarchieebenen räumlich zu jeweils größeren Gruppen zusammen (Bild 15.3). Die Hülle eines Endknotens umschließt nur das darin enthaltene Objekt, die Hülle eines internen Knotens umschließt die darin enthaltene Gruppe von Objekten, und die Hülle des Wurzelknotens umschließt die gesamte Szene. Dadurch entsteht eine Hierarchie von Hüllen (*bounding volume hierarchy*), die die nachfolgenden Algorithmen (in erster Linie Culling und Kollisionserkennung) stark vereinfacht. Typischerweise reduziert sich die Komplexität der Algorithmen durch eine räumliche Hierarchie von $O(n)$ zu $O(\log n)$.

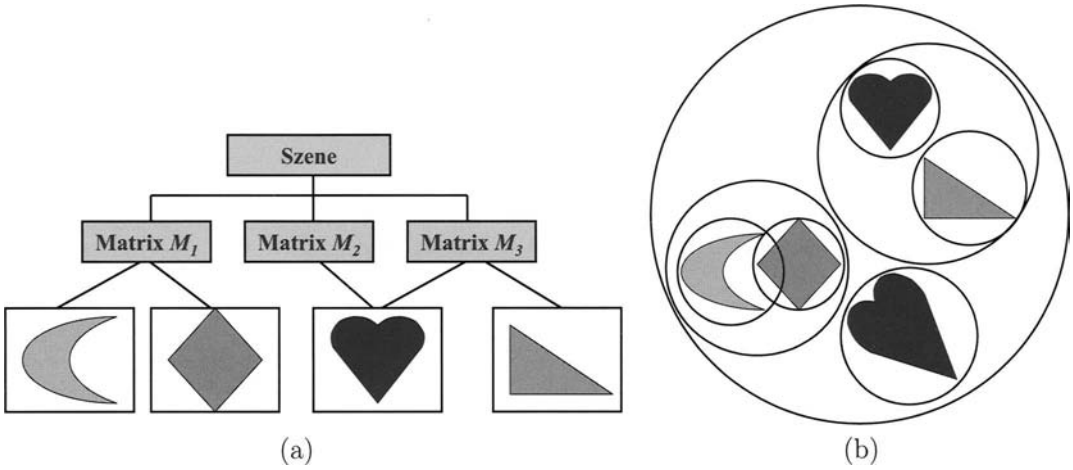


Bild 15.3: (a) Ein räumlich organisierter Szenen Graph. Jedem Knoten ist eine Hülle (hier eine Kugeloberfläche) zugeordnet. (b) Die Darstellung der Szene. Jedes Objekt ist mit seiner zugehörigen Hülle dargestellt. Die Objekte sind teilweise zu Gruppen zusammengefasst, die wieder von größeren Hüllen umgeben sind. Die gesamte Szene, d.h. der Wurzelknoten besitzt eine Hülle, die alle Objekte umfasst.

Mit Hilfe von Transformations-Knoten lassen sich in einem hierarchisch organisierten Szenen Graphen komplexe Modelle aus einfachen Grundbausteinen sehr elegant aufbau-

en. Jeder Transformations-Knoten i speichert eine Transformations-Matrix $\mathbf{M}_i$, die eine Kombination von Modell-Transformationen enthält (Translation * Rotation * Skalierung). Da mehrere interne Knoten einen einzigen Nachfolge-Knoten referenzieren können, ist es möglich, Kopien (Instanzen) eines Objekts anzufertigen, ohne die Geometrie zu vervielfältigen. Der Aufbau eines Szenen Graphen mit mehreren Schichten von Transformations-Knoten entspricht genau dem Konzept der Matrizen-Stapel in OpenGL (Abschnitt 7.8). Jede Schicht von Transformations-Knoten repräsentiert eine Ebene des Matrizen-Stapels, wobei die Zuordnung reziprok ist. Die unterste Schicht von Transformations-Knoten im Szenen Graph entspricht der obersten Matrix auf dem Stapel und umgekehrt. Das in Bild 7.14 gezeigte Beispiel eines Autos aus einem Chassis, vier Rädern und jeweils fünf Befestigungsschrauben ist als Szenen Graph in Bild 15.4 dargestellt.

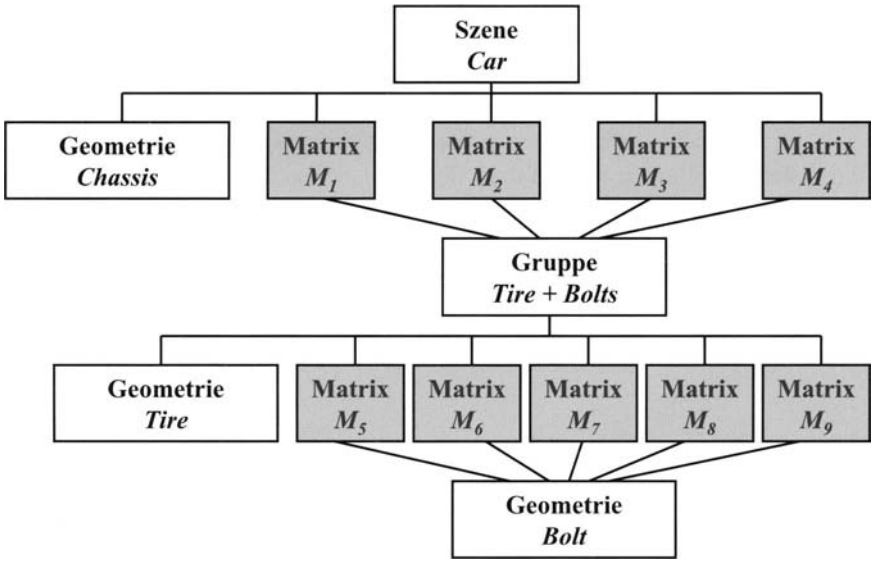


Bild 15.4: Der Szenen Graph für ein hierarchisch aufgebautes Modell eines Autos, das aus einem Chassis, vier Räder und jeweils fünf Befestigungsschrauben besteht. Die Realisierung dieses Szenen Graphen in OpenGL mit Hilfe eines Matrizen-Stapels ist in Bild 7.14 zu sehen.

Eine Hierarchie von Transformations-Knoten bietet eine elegante Möglichkeit zur Animation von Objekten. Wie in Abschnitt 14.2 beschrieben, unterscheidet man verschiedene Hierarchieebenen der Animation: Die Bewegung ganzer Objektgruppen, eines einzelnen Objekts oder von Objektteilen. Dies kann durch eine laufende Anpassung der Transformations-Matrix auf der jeweiligen Hierarchieebene des Szenen Graphen erreicht werden. Wird z.B. die Transformations-Matrix (M_5) auf der unteren Ebene der Transformations-Knoten in Bild 15.4 mit jedem neu zu zeichnenden Bild angepasst, kann das Aufdrehen einer Be-

festigungsschraube simuliert werden. Passt man dagegen die Matrix (M_1) auf der oberen Ebene der Transformations-Knoten laufend an, kann z.B. die Drehung des ganzen Rades incl. Befestigungsschrauben dargestellt werden.

Bei der Darstellung einer kontinuierlichen Bewegung wird der Szenen Graph also laufend angepasst. Dabei wird der Szenen Graph von oben nach unten traversiert (d.h. durchlaufen), so dass sich Änderungen an einem Transformations-Knoten weiter oben auf alle Nachfolge-Knoten weiter unten auswirken. Mit jeder Änderung eines Transformations-Knotens müssen auch die zugehörigen Hüllen neu berechnet werden, damit sie die richtige Größe und Position besitzen.

Der Szenen Graph wird vor jedem neu zu generierenden Bild mehrmals traversiert (*database traversal*). Die soeben angesprochenen Änderungen von Transformations-Matrizen zur Erzeugung von Bewegungen werden während des sogenannten „*application traversal*" vorgenommen. Bei dem im nächsten Abschnitt genauer beschriebenen „*cull traversal*" wird der Szenen Graph ein zweites Mal von oben nach unten durchlaufen, um die außerhalb des sichtbaren Volumens liegenden Anteile des Szenen Graphen zu eliminieren. In Anwendungen, bei denen eine Kollisionserkennung erforderlich ist, wird der Szenen Graph noch einmal im Rahmen eines „*intersection traversal*" durchlaufen. Bei bestimmten Anwendungen sind auch noch weitere Traversierungen sinnvoll [Ecke04].

15.2 Cull Algorithmen

Die Grundidee aller *Cull*[1] Algorithmen ist, nur die Teile einer Szene zu rendern, die man später am Bildschirm auch sieht. In anderen Worten heißt das, die später nicht sichtbaren Teile vom Szenen Graphen abzutrennen, bevor sie in die *Rendering Pipeline* geschickt werden. Dies wird im Rahmen des *cull traversals* von der CPU erledigt, bevor der sichtbare Teil des Szenen Graph in die Grafikhardware zum Rendern geschickt wird. Zum besseren Verständnis wird folgendes Beispiel betrachtet: Ein Bahn-Simulator mit einer Sichtweite von 2 *km* (*far clipping plane*) und einer visuellen Datenbasis mit einer Länge von 1000 *km*. Ohne *Cull* Algorithmus müsste die gesamte, 1000 *km* lange Strecke mit einer Unmenge an Geometriedaten für jedes neu zu berechnende Bild (z.B. 60 mal pro Sekunde) durch die Grafik Pipeline geschickt werden, mit *Cull* Algorithmus nur ca. 0,2% davon. Der *Cull* Algorithmus beschleunigt also die Bildgenerierung um einen Faktor 500 (falls der Flaschenhals der Anwendung die Geometrieeinheit ist).

Der Einsatz eines *Cull* Algorithmus' befreit zwar die Grafikhardware von einer unter Umständen riesigen Last, dafür belastet er die CPU je nach Größe und Organisation der visuellen Datenbasis entsprechend stark. Letztlich wird also die Rechenlast von der Seite der Grafikhardware auf die Host-Seite (CPU) verlagert. Da der Wirkungsgrad eines *Cull* Algorithmus' bei guter räumlicher Organisation des Szenen Graphen jedoch sehr hoch ist, lohnt sich diese Verlagerung der Rechenlast in jedem Fall. Falls die verfügbare CPU allerdings

[1]Der englische Begriff „to cull" bedeutet wörtlich übersetzt „trennen von der Herde" oder „auslesen aus einer größeren Menge".

durch andere Aufgaben schon stark belastet ist (z.B. durch Kollisions- oder Dynamikberechnungen), wird eine zusätzliche CPU unvermeidlich, wenn man die Bildgenerierrate halten möchte (Abschnitt 15.5).

Das schon häufiger erwähnte *Clipping*[2] (Abschnitt 7.2) darf nicht mit *Culling* verwechselt werden. Das *Clipping* wird in der OpenGL *Rendering Pipeline* immer durchgeführt, auch ohne dass ein *Cull* Algorithmus aktiv war. Dabei wird nach der Projektionstransformation, d.h. nachdem die Vertices in Projektionskoordinaten (*clip coordinates*) vorliegen, überprüft, welche Grafik-Primitive vollständig innerhalb des sichtbaren Bereichs liegen, welche den Rand des sichtbaren Bereichs schneiden und welche vollkommen außerhalb des sichtbaren Bereiches liegen. Letztere werden eliminiert, genau wie bei einem *Cull* Algorithmus. Allerdings müssen alle Vertices (also auch die letztlich nicht sichtbaren) vorher mit der ModelView-Matrix transformiert, beleuchtet und dann noch einmal mit der Projektions-Matrix transformiert werden. Genau diesen Aufwand spart man sich, wenn vor dem *Clipping* ein (*Viewing Frustum*) *Culling* durchgeführt wird. Der eigentliche Zweck des Clippings besteht darin, die Grafik-Primitive, die teilweise innerhalb und teilweise außerhalb des sichtbaren Bereichs liegen, so zuzuschneiden, dass sie exakt mit dem Rand des Sichtbereichs abschließen (Bild 15.5). *Clipping* ist ein Bestandteil von OpenGL und wird in der Geometrie-Stufe der Grafikhardware immer durchgeführt. *Cull* Algorithmen sind, bis auf das *Backface Culling*, nicht Bestandteil von OpenGL, sie werden auf der CPU durchgeführt bevor die Daten an die Grafikhardware übergeben werden.

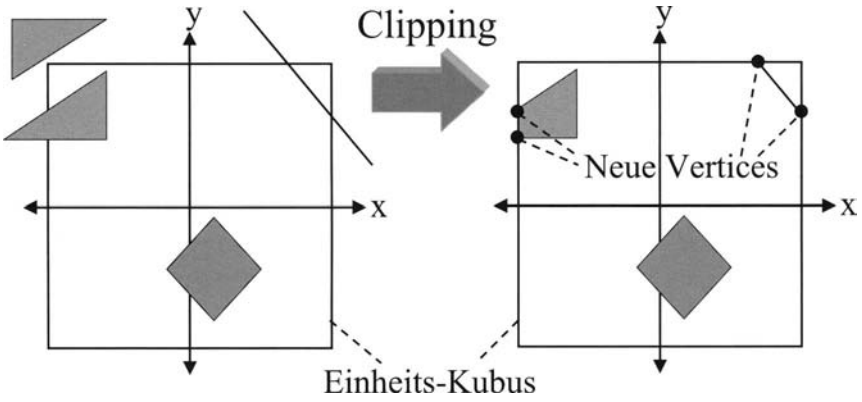


Bild 15.5: *Clipping*: Nach der Projektionstransformation werden die Grafik-Primitive, die außerhalb des sichtbaren Bereichs liegen (dies ist nach der Normierungstransformation der Einheits-Kubus) eliminiert. Die Grafik-Primitive, die teilweise innerhalb und teilweise außerhalb liegen, werden so beschnitten („geclippt"), dass sie mit dem Sichtbereich abschließen. Dadurch entstehen zum Teil neue Vertices.

Neben der Eliminierung nicht sichtbarer Teile des Szenen Graphen werden im Rahmen des *cull travesals* in der Regel noch weitere Aufgaben erledigt. Dazu zählen z.B.

[2]Der englische Begriff „to clip" bedeutet wörtlich übersetzt „beschneiden" oder „ausschneiden".

die Auswahl der passenden Detailstufe eines LOD-Knotens (15.3), die Ausrichtung eines Billboards zum Augenpunkt hin (15.4), und nicht zuletzt die Sortierung des nach dem Culling verbliebenen Szenen Graphen im Hinblick auf gemeinsame Rendering-Zustände. Die Sortierung kann zwar bei komplexeren Szenen zeitaufwändig sein und somit die CPU belasten, aber andererseits beschleunigt es die Bildgenerierung meistens relativ stark, da häufige Zustandswechsel in OpenGL mit erheblichem Overhead verbunden sind. Der Cull Algorithmus produziert letztendlich eine sogenannte „Darstellungsliste" (*display list*), in der die sichtbaren Teile der Szene in Form von sortierten OpenGL-Befehlen stehen. Diese „Darstellungsliste" wird anschließend im Rahmen eines „*draw traversals*" in die *Rendering Pipeline* geschoben (im Falle eines Mehrprozessorsystems (Abschnitt 15.5) wird dies von einem eigenen "*Draw*"-Prozess durchgeführt).

Im Folgenden werden die verschiedenen Arten von *Cull* Algorithmen, sowie deren Vor- und Nachteile ausführlicher dargestellt.

15.2.1 Viewing Frustum Culling

Die Grundidee des *Viewing Frustum Culling* besteht darin, nur das zu rendern, was sich innerhalb des sichtbaren Volumens (*Viewing Frustum*) befindet, denn nur diese Teile der Szene können am Bildschirm sichtbar sein. Nachdem man einen gut präparierten Szenen Graph erzeugt hat, der in einer räumlichen Hierarchie aufgebaut ist und bei dem jeder Knoten eine einfache Hülle besitzt, wie in Abschnitt 15.1 beschrieben, kann das *Viewing Frustum Culling* effizient ablaufen. Der Algorithmus vergleicht die Hülle des jeweiligen Knotens mit dem sichtbaren Volumen. Ist die Hülle vollständig außerhalb des sichtbaren Volumens (in Bild 15.6 der Knoten $\mathbf{M}_4$), wird der Knoten und alle seine Nachfolger, d.h. der gesamte Ast, eliminiert („gecullt"). Ist die Hülle vollständig innerhalb des sichtbaren Volumens (in Bild 15.6 die Knoten $\mathbf{M}_1$ und $\mathbf{M}_2$), geschieht nichts, denn der gesamte Ast ist potentiell sichtbar. In diesem Fall durchläuft der Algorithmus die Nachfolge-Knoten zwar noch wegen seiner anderen Aufgaben (LOD-Auswahl, Billboard-Drehung etc.), aber das Testen der Hüllen gegen das sichtbare Volumen kann entfallen. Schneidet die Hülle das sichtbare Volumen (in Bild 15.6 der Knoten $\mathbf{M}_3$), startet die Rekursion, und der Algorithmus beginnt eine Ebene unterhalb des aktuellen Knotens wieder von vorne. Falls die Hülle eines Endknotens das sichtbare Volumen schneidet, bleibt der Knoten erhalten, denn es könnte ein Teil des Objekts innerhalb des sichtbaren Volumens sein (das *Clipping* innerhalb der Rendering Pipeline schneidet solche Objekte an den Begrenzungsflächen des sichtbaren Volumens ab). Mit diesem rekursiven *Cull* Algorithmus wird, beginnend mit dem Wurzelknoten, der gesamte Szenen Graph traversiert. In Bild 15.6 ist die Wirkung des *Viewing Frustum Culling* an einem einfachen Szenen Graphen dargestellt.

Als abschreckende Beispiele sollen noch einige Varianten von schlecht organisierten Szenen Graphen betrachtet werden. Angenommen, der Szenen Graph einer 1000 *km* langen Bahnstrecke wird nicht räumlich, sondern nach Objekttypen organisiert. Alle Bäume werden in einem Knoten zusammengefasst, in einem Nachfolge-Knoten werden alle Laubbäume erfasst, in einem anderen alle Nadelbäume usw., dann folgen die Häuser, Straßen und so fort. Da alle Objekttypen überall entlang der Bahnstrecke vorkommen, werden die Hüllen

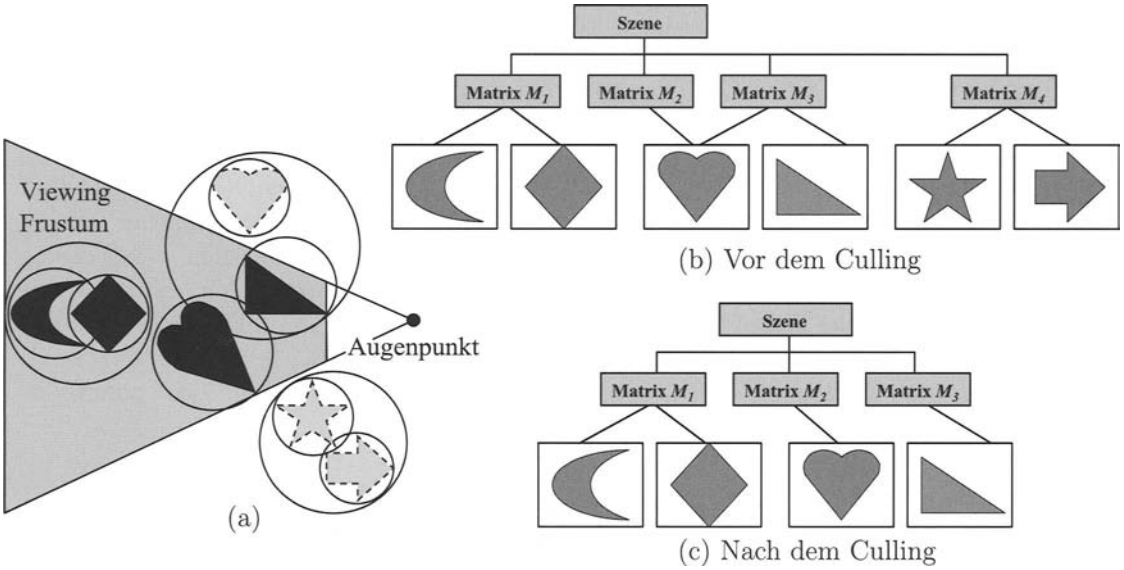


Bild 15.6: *Viewing Frustum Culling*: (a) Die Szene aus der Vogelperspektive, mit sichtbarem Volumen (grau) und Hüllen der Objekte (Kreise). Die gestrichelt umrandeten Objekte (Pfeil, Stern und Herz) liegen außerhalb des sichtbaren Volumens und werden „gecullt". (b) Der räumlich organisierte Graph der Szene vor dem Culling. (c) Der Szenen Graph nach dem Culling: Man beachte, dass nicht nur der Knoten $\mathbf{M}_4$ mitsamt seinen Folgeknoten eliminiert wurde, sondern auch die Verbindung zwischen dem Knoten $\mathbf{M}_3$ und dem Endknoten mit dem Herz.

der Knoten auf allen Ebenen (evtl. bis auf die unterste Ebene mit den Endknoten) immer die gesamte riesige Bahnstrecke umschließen. Der *Cull* Algorithmus kann folglich auf keiner höheren Knotenebene eine größere Menge an Objekten auf einen Schlag eliminieren, sondern er muss bis hinunter zur untersten Knotenebene, um dort jedes Objekt einzeln gegen das sichtbare Volumen zu testen. In solchen Fällen ist zu erwarten, dass der *Cull* Algorithmus zum Flaschenhals der Bildgenerierung wird. Eine ähnliche Problematik tritt auf, wenn der Szenen Graph zwar räumlich organisiert ist, aber mit einer sehr flachen Hierarchie. Eine heuristische Regel für die Aufteilung einer Szene ist, dass die Zahl der Hierarchie-Stufen etwa gleich der Zahl der Nachfolge-Knoten sein sollte. Außerdem muss vor der Erstellung einer visuellen Datenbasis, abhängig von der Anwendung, entschieden werden, wie groß bzw. klein die Objekte auf der untersten Knoten-Ebene gewählt werden sollen. Soll bei einem Bahn-Simulator die kleinste Einheit z.B. ein Zug, ein Waggon, ein Radgestell oder evtl. sogar eine Schraube sein? Abhängig von der Granularität, mit der eine visuelle Datenbasis durch einen Szenen Graph repräsentiert wird, kann ein *Cull* Algorithmus die Szene mehr oder weniger genau auf die sichtbaren Teile zuschneiden. Allerdings muss man sich vorher fragen, ob die evtl. geringe Einsparung an Polygonen den zusätzlichen Aufwand beim *Cull* Algorithmus und bei der Erstellung der visuellen Datenbasis rechtfertigt.

Das *Viewing Frustum Culling* läuft auf der CPU ab und entlastet den Bus (z.B. AGP) zur Grafikkarte sowie die Geometrie-Stufe der Grafikhardware (*Geometry Engine*), da eine sehr viel kleinere Anzahl an Polygonen bzw. Vertices in die Rendering Pipeline geschoben werden muss, als ohne *Cull* Algorithmus. Je größer die visuelle Datenbasis im Verhältnis zum sichtbaren Volumen ist, desto höher ist die Einsparung durch das *Viewing Frustum Culling*. Die Last der Rasterisierungs-Stufe (*Raster Manager*) bleibt aber unverändert, da Polygone außerhalb des sichtbaren Volumens in jedem Fall (d.h. auch ohne *Culling*) vorher schon durch das in der Geometrie-Stufe ablaufende *Clipping* eliminiert worden wären. Falls man mit einem mittelmäßigen *Cull* Algorithmus das Gros der nicht sichtbaren Polygone bereits eliminiert hat und der Flaschenhals der Anwendung in der Rasterisierungs-Stufe steckt, wäre es deshalb sinnlos, zusätzlichen Aufwand in einen genaueren *Viewing Frustum Cull* Algorithmus zu stecken. In solch einem Fall sollte man den Aufwand bei einem anderen *Cull* Algorithmus (z.B. *Occlusion Culling*, Abschnitt 15.2.2) investieren, der auch die Rasterisierungs-Stufe entlastet. Weitere Methoden zur gleichmäßigen Verteilung der Last auf alle Stufen einer Grafikanwendung werden in Abschnitt 15.6 besprochen.

Bei den meisten Echtzeit-Anwendungen ist es üblich, dass der Augenpunkt oder andere Objekte mit einer begrenzten Geschwindigkeit durch die visuelle Datenbasis bewegt werden. Bei einer hohen Bildgenerierrate (z.B. 60 Hz) wird der Augenpunkt bzw. die anderen Objekte deshalb nur um ein kleines Stückchen bewegt. Das *Viewing Frustum Culling*, das für jedes Bild von Neuem durchgeführt wird, liefert aufgrund dieser zeitlichen Kohärenz häufig identische Ergebnisse, d.h. die gecullten Szenen Graphen zweier aufeinander folgender Bilder stimmen meistens zu mehr als 99% überein. Fortgeschrittene *Viewing Frustum Cull* Algorithmen [Assa00] machen sich diese zeitliche Kohärenz zu Nutze, um so den Rechenaufwand für den *Cull* Algorithmus zu senken, ohne dessen positive Wirkung auf die nachfolgenden Stufen der *Rendering Pipeline* zu schmälern.

15.2.2 Occlusion Culling

Das *Viewing Frustum Culling* eliminiert zwar alle Objekte außerhalb des sichtbaren Volumens und reduziert damit die Grafiklast bei großen visuellen Datenbasen drastisch. Bei bestimmten Anwendungen aber, in denen sich innerhalb des sichtbaren Volumens noch sehr viele Objekte befinden und gegenseitig verdecken, bleibt die Grafiklast immer noch viel zu hoch für akzeptable Bildgenerierraten. Solche Anwendungen sind z.B. Simulationen in großen Städten oder Gebäuden, in denen die Wände einen Großteil aller dahinter liegenden Objekte verdecken, oder ein CAD-Werkzeug, das ein Auto mit seinem extrem komplexen Motor darstellen soll, der aber gar nicht sichtbar ist, solange die Motorhaube geschlossen bleibt.

In diesen Fällen benötigt man einen *Occlusion Cull* Algorithmus, der die verdeckten Objekte aus dem Szenen Graph eliminiert (Bild 15.7). Die Aufgabe, die korrekte gegenseitige Verdeckung von Objekten zu berechnen, ist eine der Standardaufgaben der 3D-Computergrafik. Zur Lösung dieser Aufgabe wurde in Kapitel 8 der „z-Buffer Algorithmus“ vorgestellt. Allerdings hat dieser Algorithmus einen erheblichen Nachteil: Bevor die korrekte Verdeckung aller Objekte feststeht, müssen alle Polygone der Szene die *Rendering*

Pipeline durchlaufen haben. Beim *Occlusion Culling* versucht man durch einfache Tests vorab zu vermeiden, dass die verdeckten Polygone durch den Großteil der *Rendering Pipeline* geschickt werden, bevor sie der z-Buffer Algorithmus am Ende dann doch pixelweise eliminiert.

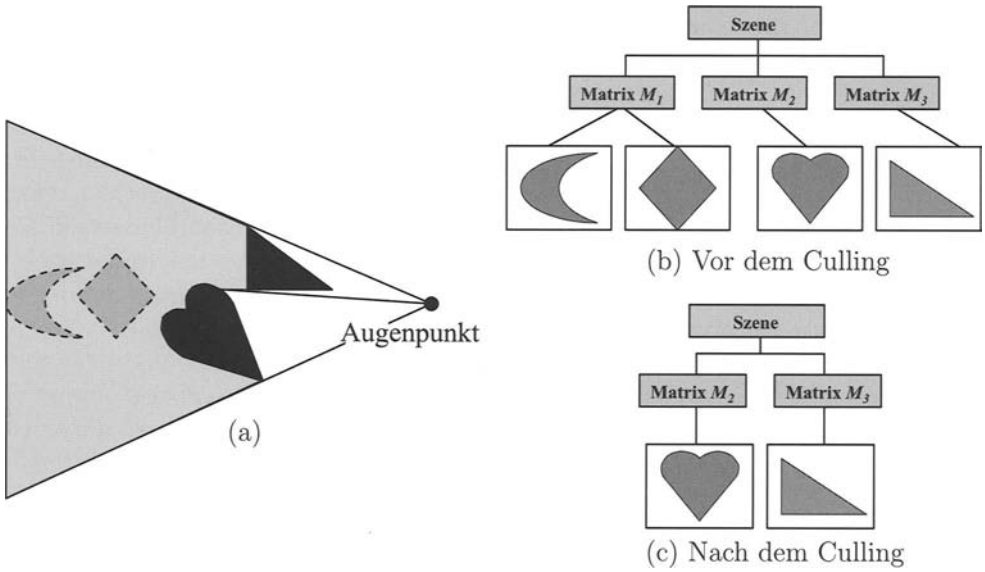


Bild 15.7: *Occlusion Culling*: (a) Die Szene aus der Vogelperspektive, mit sichtbarem Volumen und verdecktem Bereich (grau). Die gestrichelt umrandeten Objekte (Halbmond und Raute) werden durch die schwarzen Objekte (Herz und Dreieck) verdeckt und deshalb „gecullt". (b) Der räumlich organisierte Graph der Szene vor dem *Occlusion Culling* (dass was nach dem *Viewing Frustum Culling* noch übrig ist). (c) Der Szenen Graph nach dem *Occlusion Culling*: Der Knoten $\mathbf{M}_1$ mitsamt seinen Folgeknoten wurde eliminiert.

Das Problem beim *Occlusion Culling* ist, dass es keine wirklich „einfachen" Verdeckungstests gibt. Es existieren zwar zahlreiche Konzepte für das *Occlusion Culling* (z.B. *Occlusion Horizons* [Down01], *Shaft Occlusion Culling* [Scha00], *Hierarchical z-Buffering* [Gree93] und *Hierarchical Occlusion Maps* [Zhan97]), aber alle Ansätze sind enorm rechenaufwändig. Deshalb macht der Einsatz von *Occlusion Cull* Algorithmen erst ab einem bestimmten Verdeckungsgrad (*depth complexity*) Sinn, bei dem der Aufwand des Cullings geringer ist, als das Rendering der überflüssigen Objekte. Die erwähnten Ansätze sind (bisher) nicht durch Grafikhardware beschleunigt, sondern müssen auf der CPU ausgeführt werden.

Seit der im Jahre 2003 erschienen Version 1.5 von OpenGL werden sogenannte *Occlusion Queries* im Rahmen der OpenGL *Extension* (`GL_ARB_OCCLUSION_QUERY`) durch Hardware-Beschleunigung unterstützt (`glGenQueries()`, `glBeginQuery()`, `glEndQuery()`, [Sega04]). Ein *Occlusion Query* ist eine Anfrage an die Grafikhardware, ob ein Objekt bzw. ein Satz an Polygonen durch den bisherigen Inhalt des z-Buffers verdeckt ist oder nicht.

Um unter dem Strich Rechenzeit zu sparen, schickt man nicht die komplexen Objekte in die Grafikhardware, sondern nur deren sehr viel einfachere Hüllen (Kugeloberflächen oder Quader). Die Hüllen durchlaufen in einem vereinfachten Verfahren die *Rendering Pipeline* (d.h. sie werden transformiert und rasterisiert, aber nicht beleuchtet, texturiert usw.) und deren z-Werte werden am Ende mit dem Inhalt des z-Buffers verglichen. Dabei wird mitgezählt, wie viele Pixel den z-Buffer-Test bestehen, d.h. wie viele Pixel sichtbar sind. Diese Zahl n wird als Ergebnis des *Occlusion Queries* an die Anwendung zurückgeliefert. Falls die Zahl $n = 0$ ist, muss die Hülle und somit das gesamte Objekt verdeckt sein und kann „gecullt" werden. Falls die Zahl $n > 0$ ist, sind n Pixel der Hülle sichtbar. Wenn die Zahl n unter einem bestimmten Schwellwert bleibt, kann man das Objekt trotzdem noch eliminieren, denn es könnte ja sein, dass zwar die Hülle sichtbar ist, aber nicht die darin enthaltenen Objekte. Auf diese Weise kann man die Bildqualität gegen die Rendering-Geschwindigkeit abwägen [Aken02]. Ein anderer Algorithmus kann sich die Zahl n zu Nutze machen, um zu entscheiden, ob (und evtl. welche) Nachfolge-Knoten für eine erneute *Occlusion Query* benutzt werden. Eine weitere Möglichkeit, die Zahl n zu nutzen, besteht darin, sie zur Auswahl des Level-Of-Detail (LOD, Abschnitt 15.3) eines Objekts heranzuziehen. Denn solange nur wenige Pixel des Objekts sichtbar sind, reicht evtl. eine geringere Detailstufe des Objekts für eine befriedigende Darstellung aus [Aken02]. Eine weitere Effizienzsteigerung solcher *Occlusion Cull* Algorithmen ist durch eine Vorsortierung der Knoten des Szenen Graphen nach der Entfernung vom Augenpunkt zu erreichen [Meis99]. Der vorderste Knoten wird ohne Verdeckungstest gerendert. Die Hüllen der nächsten Knoten werden sukzessive in den Verdeckungstest geschickt. Falls eine Hülle sichtbar ist, wird der Knoten gerendert oder die Hüllen der Nachfolge-Knoten werden rekursiv getestet. Falls eine Hülle verdeckt ist, wird der Knoten und alle seine Nachfolger „gecullt".

Ein weiteres Konzept für das *Occlusion Culling* besteht in der Verbindung von Ideen des *Shadow Mapping* (Abschnitt 13.6) und des *Hierarchical z-Buffering* [Gree93]. Dabei wird die Szene (entweder die Objekte selbst oder nur die Rückseiten der Hüllen) zunächst in einem vereinfachten Verfahren gerendert (d.h. sie wird transformiert und rasterisiert, aber nicht beleuchtet, texturiert usw.), um den Inhalt des z-Buffers und damit die korrekten Verdeckungen zu erzeugen (ähnlich wie beim *Shadow Mapping*, nur dass hier nicht aus der Position der Lichtquelle, sondern aus der Position der Augenpunkts gerendert wird). Der Inhalt des z-Buffers wird nun mit dem Befehl `glReadPixels()` als „z-Textur" in den Hauptspeicher kopiert. Um die späteren Verdeckungstests zu beschleunigen, werden ähnlich wie bei den Gauß-Pyramiden-Texturen (Abschnitt 13.1.3) verkleinerte Varianten der z-Textur erzeugt, so dass eine „z-Pyramide" entsteht. Im Unterschied zu den Gauß-Pyramiden wird die z-Pyramide aber dadurch erzeugt, dass in einem $2 \cdot 2$ Pixel großen Fenster der maximale z-Wert bestimmt wird und dieser z-Wert auf der um den Faktor 2 in jeder Dimension verkleinerten Variante der z-Textur eingesetzt wird. Mit diesem modifizierten REDUCE-Operator (Abschnitt 28.4) wird die gesamte z-Pyramide erzeugt, die an der Spitze nur noch eine $1 \cdot 1$ Pixel große Textur mit dem maximalen z-Wert der gesamten Szene enthält. Weiterhin wird vorausgesetzt, dass die Hüllen der Knoten des Szenen Graphen aus axial ausgerichteten Quadern (*axis-aligned bounding boxes*) im Weltkoordinatensystem bestehen. In diesem Fall benötigt man nur drei Vertices, um die Vorderseite des Quaders

zu beschreiben, die dem Augenpunkt am nächsten liegt. Außerdem besitzen alle Punkte der Vorderseite eines axial ausgerichteten Quaders die gleichen z-Werte, so dass die Rasterisierung des Quaders nicht mehr erforderlich ist. Diese drei Vertices der Vorderseite des Quaders werden nun mit Hilfe der Projektionsmatrix vom Weltkoordinatensystem in das normierte Projektionskoordinatensystem transformiert, so dass die z-Werte der Vertices mit den Werten der z-Pyramide verglichen werden können. Abhängig von der Ausdehnung des Quaders in x- und y-Richtung wird nun eine von der Größe her adäquate z-Textur aus der z-Pyramide ausgewählt, so dass für den Vergleich des z-Werts der Vorderseite des Quaders nur verhältnismäßig wenig Werte der z-Textur herangezogen werden müssen. Ist der z-Wert der Vorderseite des Quaders größer als alle Werte der z-Textur im Bereich des Quaders, so ist der Quader vollständig verdeckt und kann „gecullt" werden. Andernfalls wird der Test rekursiv mit den Nachfolge-Knoten durchgeführt, oder, falls es sich um einen Endknoten handelt, wird dieser gerendert.

Im Gegensatz zum *Viewing Frustum Culling*, das in nahezu allen Echtzeit-Rendering-Werkzeugen standardmäßig implementiert ist, gibt es für das *Occlusion Culling* praktisch nur Speziallösungen, die der Anwender selbst programmieren muss. Die großen Vorteile des *Occlusion Cullings* sind:

- Es ist komplementär zum *Viewing Frustum Culling*. Deshalb bietet es sich an, im ersten Schritt alle Objekte außerhalb des *Viewing Frustum* zu eliminieren, so dass das aufwändigere *Occlusion Culling* auf der Basis einer sehr viel geringeren Zahl an Objekten im zweiten Schritt alle verdeckten Objekte eliminieren kann.

- Das *Occlusion Culling* entlastet den Bus zur Grafikkarte, die Geometrie-Stufe und – im Gegensatz zum *Viewing Frustum Culling* – auch die Rasterisierungs-Stufe der Grafikhardware, da die Anzahl an Polygonen bzw. Vertices verringert wird und zusätzlich jedes Pixel des Bildes meist nur einmal die Fragment-Operationen (Rasterisierung, Texturierung, Anti-Aliasing usw.) durchlaufen muss.

- Das *Occlusion Culling* ist praktisch die einzige Möglichkeit um Szenen mit sehr hohem Verdeckungsgrad (depth complexity) interaktiv zu rendern.

- Die Last wird von der Grafikhardware auf die CPU-Seite vorverlagert.

Die Nachteile des *Occlusion Cullings* sind die relativ hohe Rechenzeit und – wegen der schlechten Verfügbarkeit in Echtzeit-Rendering-Werkzeugen – der hohe Programmieraufwand. *Occlusion Culling* ist ein nach wie vor aktives Gebiet der Forschung. Aufgrund seiner Bedeutung für viele Anwendungen und der rasanten Entwicklung der Grafikhardware ist zu erwarten, dass in Zukunft auch das *Occlusion Culling* standardmäßig unterstützt wird.

15.2.3 Backface Culling

Bei undurchsichtigen Objekten sind die Rückseiten normalerweise nicht sichtbar[3]. Deshalb braucht man sie auch nicht zu rendern, denn sie tragen nichts zum Bild bei. Das ist die Grundidee des *Backface Cullings* (Bild 15.8). Auf diese Weise lässt sich die Zahl der zu rendernden Polygone auch nach der Durchführung von *Viewing Frustum* und *Occlusion Culling* noch weiter reduzieren.

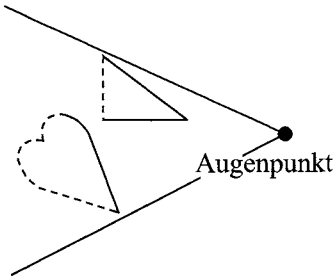


Bild 15.8: *Backface Culling*: Die Polygone, bei denen man nur die Rückseite sieht (im Bild gestrichelt dargestellt), werden eliminiert.

Voraussetzung für das *Backface Culling* ist eine konsistente Polygon-Orientierung (z.B. entgegen dem Uhrzeigersinn), wie in Abschnitt 6.2.3.4 dargestellt. Erscheinen die Vertices eines Polygons z.B. im Uhrzeigersinn auf dem Bildschirm, würde man die Rückseite des Polygons sehen, und kann es deshalb weglassen. Die Entscheidung, welche Seite eines Polygons man am Bildschirm sieht, kann man mit Hilfe der Berechnung des Normalenvektors $\mathbf{n}$ in Bildschirm-Koordinaten treffen:

$$\begin{aligned} \mathbf{n} &= (\mathbf{v}_{01}) \times (\mathbf{v}_{02}) = (\mathbf{v}_1 - \mathbf{v}_0) \times (\mathbf{v}_2 - \mathbf{v}_0) = \begin{pmatrix} x_1 - x_0 \\ y_1 - y_0 \\ 0 \end{pmatrix} \times \begin{pmatrix} x_2 - x_0 \\ y_2 - y_0 \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 0 \\ (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0) \end{pmatrix} \end{aligned} \tag{15.1}$$

wobei ($\mathbf{v}_0, \mathbf{v}_1$ und $\mathbf{v}_2$) die Vertices des Polygons in Bildschirm-Koordinaten sind. Der Normalenvektor des auf den Bildschirm projizierten Polygons kann also entweder nur in Richtung des Augenpunkts zeigen ($n_z > 0$), was bedeutet, dass man die Vorderseite des Polygons sieht, oder in die entgegengesetzte Richtung ($n_z < 0$), was heißt, dass man die Rückseite des Polygons sieht. Das *Backface Culling* kann also nach der Projektionstransformation der Vertices noch innerhalb der Geometrie-Stufe der *Rendering Pipeline* durchgeführt werden.

[3]Ausnahmen sind transparente Vorderseiten, durch die die Rückseiten durchschimmern und teilweise aufgeschnittene Hohlkörper

Backface Culling ist Bestandteil der OpenGL *Rendering Pipeline* und kann mit den in Abschnitt 6.2.3.4 beschriebenen Befehlen aktiviert oder deaktiviert werden. *Backface Culling* reduziert die Last der Rasterisierungs-Stufe, da die Rückseiten der Polygone nicht rasterisiert und texturiert werden müssen. Da der Algorithmus in der Geometrie-Stufe der Grafikhardware (in der alle vertex-bezogenen Operationen stattfinden) durchgeführt wird, erhöht er dort die Last. Mit dem *Backface Culling* wird also der Rechenaufwand von der Rasterisierungs-Stufe zur Geometrie-Stufe der Grafikhardware vorverlagert. Eine Anwendung, deren Flaschenhals in der Rasterisierungs-Stufe steckt, kann somit durch die Aktivierung des *Backface Culling* besser ausbalanciert werden. Im umgekehrten Fall, wenn die Geometrie-Stufe den Engpass darstellt, kann durch die Deaktivierung des *Backface Culling* evtl. ein Ausgleich hergestellt werden. Allerdings ist dies eher selten zu empfehlen, da der Wirkungsgrad des *Backface Culling* größer als 1 ist, d.h. der in der Geometrie-Stufe betriebene Rechenaufwand wird in der Rasterisierungs-Stufe um ein Mehrfaches eingespart.

15.2.4 Portal Culling

Bei Anwendungen wie Architekturvisualisierung oder vielen Computerspielen kann man die speziellen räumlichen Eigenschaften der Szene sehr gut für das *Culling* benutzen. Die Szenen sind in diesen Fällen häufig dadurch gekennzeichnet, dass sich der Beobachter immer in irgend einem Raum befindet, dessen Wände die Sicht auf den Großteil der restlichen visuellen Datenbasis verdecken. Einzige Ausnahme bilden die Fenster und Türen der Räume, die sogenannten „*Portale*“ (Bild 15.9). Der erste Schritt ist nun, die räumliche Organisation des Szenen Graphen mit den Räumen und Portalen der virtuellen Welt in Einklang zu bringen. Geschickterweise ordnet man jedem Knoten des Szenen Graphen einen bestimmten Raum zu. Die Objekte eines jeden Raumes und die Portale werden in Nachfolge-Knoten räumlich geordnet organisiert. Die „Portal-Knoten“ enthalten außerdem einen Verweis auf den oder die Nachbar-Räume, die man durch das Portal sehen kann. In einer Art von positivem *Culling* wird zunächst der Raum ausgewählt, in dem sich der Beobachter befindet. Mit Hilfe des *Viewing Frustum Cullings* wählt man die Nachfolge-Knoten des Raums aus, die sich innerhalb des sichtbaren Volumens befinden. Falls sich innerhalb des sichtbaren Volumens ein Portal befindet, schneidet die Portalöffnung ein verkleinertes Sichtvolumen aus dem Original aus. Mit dem verkleinerten sichtbaren Volumen beginnt das *Viewing Frustum Culling* für den Nachbar-Raum von Neuem. Dieser rekursive Algorithmus endet, sobald keine neuen Portale mehr innerhalb der sukzessive verkleinerten Sichtvolumina enthalten sind.

Die visuellen Datenbasen vieler Computerspiele sind so aufgebaut, dass aus jeder Position innerhalb eines Raumes immer nur ein einziger Nachbar-Raum durch Portale gesehen werden kann. Auf diese Weise kann eine sehr komplexe visuelle Datenbasis, deren Räume während des Spiels auch alle zugänglich sind, für das Rendering auf lediglich zwei Räume vereinfacht werden. Mit Hilfe des *Portal Cullings* lassen sich auch Spiegelungen sehr effektiv behandeln. Jeder Spiegel ist ein Portal mit einem Verweis auf den eigenen Raum. Allerdings wird in diesem Fall das verkleinerte sichtbare Volumen dadurch gebildet, dass zuerst der Augenpunkt an der Wand gespiegelt wird und dann der Beobachter von hinten

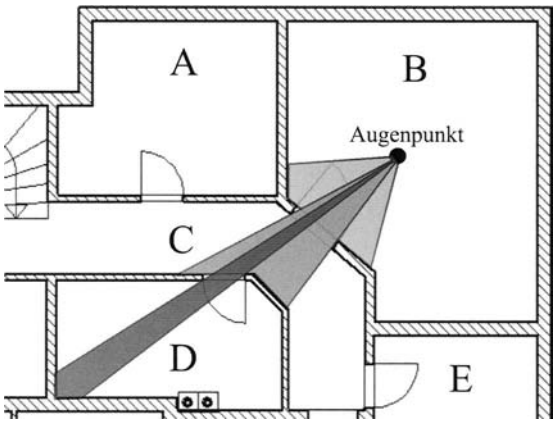


Bild 15.9: *Portal Culling*: Die Räume sind durch Buchstaben von **A** bis **E** gekennzeichnet und entsprechen jeweils einem Knoten des Szenen Graphen. Der Beobachter befindet sich in Raum **B**, so dass der Knoten ausgewählt und die im sichtbaren Volumen (hellgrau) befindliche Geometrie gerendert wird. Das sichtbare Volumen enthält ein Portal zu Raum **C**. Mit einem verkleinerten sichtbaren Volumen (mittelgrau) wird jetzt der Knoten **C** gerendert. Der Raum **C** enthält innerhalb des verkleinerten Sichtvolumens ein weiteres Portal zu Raum **D**. Das Sichtvolumen (dunkelgrau) wird durch das Portal noch einmal verkleinert und die darin befindlichen Polygone von Raum **D** werden gerendert. Weitere Portale innerhalb der Sichtvolumina existieren nicht, so dass die Rekursion hier endet.

durch den Spiegel in den Raum blickt.

Das *Portal Culling* ist eine geschickte Mischung aus *Viewing Frustum Culling* und des *Occlusion Culling* und bietet daher auch die Vorteile beider Verfahren. Es entlastet alle Stufen der Grafikhardware und auch den Bus zur Grafik. Auf der anderen Seite ist der zusätzliche Rechenaufwand auf der CPU-Seite häufig sogar geringer als bei normalem *Viewing Frustum Culling*, da im ersten Schritt nur der Raum des Beobachters gesucht werden muss und die Nachbar-Räume durch die Datenstruktur vorgegeben sind. Der Algorithmus wird also gezielt durch die Datenstruktur geleitet und muss deshalb nur wenige überflüssige Abfragen durchführen.

15.2.5 Detail Culling

Der Grundgedanke des *Detail Culling* besteht darin, Objekte die am Bildschirm sehr klein erscheinen, einfach wegzulassen. Im Gegensatz zu den bisher dargestellten *Cull* Algorithmen, die alle konservativ in dem Sinne waren, dass die Bildqualität durch das *Culling* nicht gesenkt wurde, reduziert man beim *Detail Culling* die Bildqualität zugunsten der Rendering-Geschwindigkeit. Für die Implementierung des *Detail Cullings* wird die Hülle des Objekts mit Hilfe der Projektionsmatrix vom Weltkoordinatensystem in das normierte Projektionskoordinatensystem transformiert und falls die Größe der Hülle am Bildschirm

unter einen definierten Schwellwert fällt, wird das Objekt „gecullt". Die visuelle Akzeptanz dieses Verfahrens ist dann am höchsten, wenn sich der Beobachter durch die Szene bewegt. Denn in diesem Fall verändert sich das Bild auf der Netzhaut meist relativ stark, so dass man das schlagartige Verschwinden eines wenige Pixel großen Objekts am Bildschirm leicht übersieht. Falls der Beobachter ruht und sich nur das Objekt entfernt, fällt das schlagartige Verschwinden des Objekts sehr viel eher auf. Es ist deshalb sinnvoll, das *Detail Culling* auszuschalten, sobald der Beobachter stehen bleibt. Den selben Effekt wie mit dem *Detail Culling* kann man auch mit Hilfe der im nächsten Abschnitt vorgestellten „*Level Of Detail*"-Knoten erreichen, indem die niedrigste Detailstufe des LOD-Knotens, die ab einer bestimmten Entfernung zum Augenpunkt ausgewählt wird, einfach leer ist. Der Vorteil des LOD-Verfahrens ist jedoch, dass man im Rahmen des „*Fade LOD*" (Abschnitt 15.3.2) ein verschwindendes Objekt langsam ausblenden kann, so es selbst ein ruhender Beobachter kaum bemerkt.

15.3 Level Of Detail (LOD)

Durch die im vorigen Abschnitt vorgestellten *Cull* Algorithmen wird die Zahl der zu rendernden Polygone bereits drastisch reduziert. Dennoch erfordert eine qualitativ ausreichende 3-dimensionale Modellierung unserer Umgebung immer noch eine riesige Anzahl an Polygonen. Und mit der steigenden Leistungsfähigkeit der Grafikhardware nehmen auch die Qualitätsansprüche zu. Man kann sich jedoch die Eigenschaft der perspektivischen Projektion zu Nutze machen, dass die Objekte am Bildschirm mit zunehmendem Abstand r zum Augenpunkt quadratisch kleiner werden ($1/r^2$). Deshalb genügt es, entferntere Objekte, die auf dem Bildschirm nur durch eine geringe Zahl an Pixeln dargestellt werden, mit weniger Polygonen zu modellieren. Der Beobachter kann bei interaktiven Anwendungen durch die visuelle Datenbasis bewegt werden, so dass der Abstand zu einem Objekt zwischen der *near clipping plane* und der *far clipping plane* variieren kann. Die Größe eines jeden Objekts kann daher zwischen bildschirmfüllend und winzig schwanken. Zur Lösung dieser Problematik speichert man jedes Objekt in unterschiedlichen Detailstufen (*Level Of Detail* (LOD), Bild 15.10). Abhängig von der Entfernung zwischen Objekt und Augenpunkt wird dann die adäquate LOD-Stufe ausgewählt (es gibt auch andere Kriterien für die LOD-Auswahl, wie z.B. die auf den Bildschirm projizierte Größe der Objekthülle, die Wichtigkeit des Objekts oder die Relativgeschwindigkeit des Objekts, aber die Objektentfernung wird am häufigsten verwendet). Mit der *Level Of Detail* Technik ist es möglich, bei realistischen visuellen Datenbasen die Zahl der zu rendernden Polygone um ein bis zwei Größenordnungen (d.h. Faktoren zwischen 10 und 100) zu senken. LOD ist daher ein unverzichtbarer Bestandteil von Echtzeit-3D-Computergrafik-Anwendungen.

Die *Level Of Detail* Technik ist ideal zugeschnitten auf die Architektur moderner Grafikhardware [Aken02]. Denn die einzelnen Detailstufen eines LOD-Knotens können in einer OpenGL Display Liste (Abschnitt 5.2.1) in compilierter Form im Hauptspeicher abgelegt werden und dort von der Grafikhardware über einen „*Direct Memory Access*" (DMA) abgerufen werden. Der DMA-Zugriff beschleunigt das Rendering, denn die Daten müssen

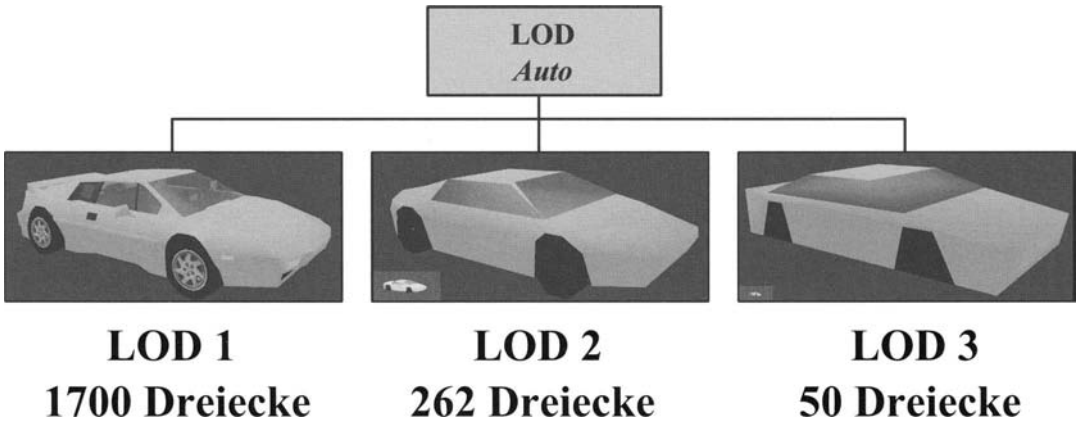


Bild 15.10: *Level Of Detail (LOD)*: Jedes Objekt wird in unterschiedlichen Detailstufen, d.h. mit einer unterschiedlichen Zahl an Polygonen und Texturen, gespeichert. Mit zunehmender Entfernung zum Augenpunkt wird eine niedrigere Detailstufe dargestellt. Die LOD-Stufen 2 und 3 des Autos sind zur Verdeutlichung sowohl in ihrer Originalgröße als auch in der perspektivisch korrekten Verkleinerung (jeweils unten links im Bild) zu sehen, ab der sie eingesetzt werden. Das Auto-Modell `esprit.flt` stammt aus der Demodatenbasis, die bei dem Echtzeit-Renderingtool „OpenGL Performer" von der Firma SGI mitgeliefert wird.

nicht über die CPU an die Grafikhardware übergeben werden, sondern können eben direkt aus dem Hauptspeicher geladen werden.

Eine äußerst umfangreiche Thematik, zu der es eigene Bücher gibt (siehe z.B. [Gall00] oder [Warr02]), ist die Generierung der Geometrie für die verschiedenen LOD-Stufen eines Objekts. Sinnvollerweise geht man dabei so vor, dass man zuerst die detaillierteste LOD-Stufe modelliert und die niedrigeren LOD-Stufen per Algorithmus automatisch erzeugt. Die modernen Modellierwerkzeuge (Abschnitt 6.4) bieten für diesen Zweck mittlerweile leistungsfähige Routinen an, die ein Polygonnetz nach einstellbaren Optimierungskriterien ausdünnen. Ein wichtiges Kriterium ist dabei, dass die Silhouette des Objekts in allen LOD-Stufen soweit als möglich erhalten bleibt, denn eine Änderung der Kontur des Objekts beim Übergang in andere LOD-Stufe fällt am ehesten störend auf.

Der Wechsel von einer LOD-Stufe zur nächsten wird im einfachsten Fall durch einen schlagartigen Austausch der Modelle realisiert. Dieses schlagartige Umschalten von einer Modellrepräsentation zur nächsten, das im folgenden Abschnitt detaillierter dargestellt wird, zieht jedoch häufig, aber ungewollt die Aufmerksamkeit des Betrachters auf sich und wird als störender „*pop-up*"-Effekt empfunden. Aus diesem Grund wurde eine Reihe von Verfahren entwickelt, die einen weicheren Übergang zwischen verschiedenen LOD-Stufen ermöglichen. Die beiden wichtigsten Vertreter, das *Fade LOD* und das *Morph LOD*, werden im Anschluss an die Standardmethode – das *Switch LOD* – hier vorgestellt.

15.3.1 Switch LOD

In seiner einfachsten Form, dem *Switch LOD*, enthält ein LOD-Knoten eines Szenen Graphen eine bestimmte Anzahl von Nachfolge-Knoten, die unterschiedliche Detailstufen eines Objekts enthalten (Bild 15.10), sowie einen Geltungsbereich $]r_{i-1}, r_i]$ für jeden Nachfolge-Knoten. Für ein *Switch LOD* mit drei Stufen gilt die LOD-Stufe 1 im Bereich $]r_0, r_1]$, die LOD-Stufe 2 im Bereich $]r_1, r_2]$ und die LOD-Stufe 3 im Bereich $]r_2, r_3]$ (Bild 15.11). Während des *Cull Traversals* (Abschnitt 15.1) wird abhängig vom Abstand r zwischen dem LOD-Knoten und dem Augenpunkt die LOD-Stufe ausgewählt, für die gilt: $(r_{i-1} < r < r_i)$. Alle anderen Nachfolge-Knoten des LOD-Knotens werden eliminiert. Somit wird für ein Bild immer nur genau eine LOD-Stufe zu 100% dargestellt und alle anderen zu 0%. Falls der Abstand r die Geltungsbereiche aller LOD-Stufen übersteigt (in Bild 15.11 $r > r_3$), wird der ganze LOD-Knoten eliminiert und somit gar kein Objekt dargestellt. Damit wird die gleiche Wirkung erzielt, wie beim *Detail Culling* (Abschnitt 15.2.5).

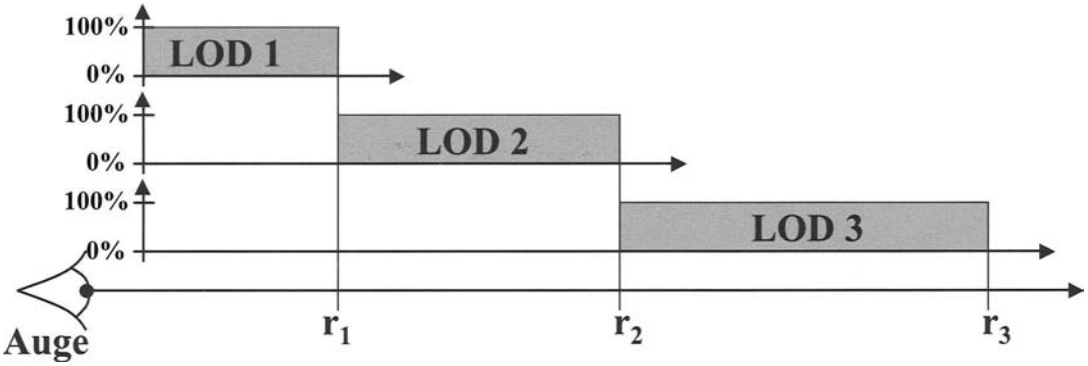


Bild 15.11: *Switch LOD*: Ist der Abstand r zwischen dem Augenpunkt und dem darzustellenden Objekt klein $(r_0 = 0 < r < r_1)$, erscheint das Objekt am Bildschirm relativ groß und wird daher in seiner höchsten Detailstufe (LOD 1) gerendert. Übersteigt der Abstand die Bereichsgrenze von LOD 1 $(r_1 < r < r_2)$ wird schlagartig auf die mittlere Detailstufe (LOD 2) umgeschaltet Bei weiter zunehmendem Abstand $(r_2 < r < r_3)$ wird die niedrigste Detailstufe (LOD 3) zu 100% dargestellt. Falls der Abstand r die Geltungsbereiche aller LOD-Stufen übersteigt $(r > r_3)$, wird das Objekt gar nicht mehr dargestellt.

Überschreitet aufgrund einer Bewegung des Augenpunkts oder des Objekts der Abstand r eine der LOD-Bereichsgrenzen r_i, wird von einem Bild zum nächsten die dargestellte LOD-Stufe schlagartig gewechselt. Die dadurch verursachten *pop-up*-Effekte fallen umso geringer aus, je weiter die Bereichsgrenzen in Richtung großer Abstände verschoben werden. Ebenso fallen aber auch die Einspareffekte entsprechend niedriger aus. Level Of Detail ist also ein klassisches Beispiel für die Austauschbarkeit von Bildqualität und Rendering-Geschwindigkeit. Diese Eigenschaft nutzt man bei der Auslegung einer Regelung für eine konstante Bildgenerierrate aus (Abschnitt 15.6).

15.3.2 Fade LOD

Der naheliegendste Weg die *PopUp*-Effekte beim Umschalten zwischen verschiedenen LOD-Stufen zu vermeiden, ist das langsame Überblenden (fading) von einer Repräsentation zur nächsten, ähnlich wie bei einem Diavortrag mit Überblendtechnik. Dabei wird dem LOD-Knoten neben den Bereichsgrenzen r_i für jeden Übergang noch ein „*Fading*"-Wert $fade_i$ zugewiesen, der die Breite des Übergangsbereichs festlegt (Bild 15.12). Für den Umschaltvorgang zwischen den LOD-Stufen wird die Farbmischung (d.h. die vierte Farbkomponente A, Kapitel 9) benutzt. Beginnend bei einer Entfernung $(r_i - fade_i/2)$ wird der LOD-Stufe i eine linear abnehmende Alpha-Komponente zugewiesen, die bei einer Entfernung von $(r_i + fade_i/2)$ auf den Wert 0 abgesunken ist. Die LOD-Stufe i wird also zwischen $(r_i - fade_i/2)$ und $(r_i + fade_i/2)$ linear ausgeblendet. Gleichzeitig wird der Alpha-Wert der LOD-Stufe $i+1$ bei $(r_i - fade_i/2)$ vom Wert 0 linear erhöht, bis er bei einer Entfernung $(r_i + fade_i/2)$ den Wert 1 erreicht hat. Die LOD-Stufe $i+1$ wird also komplementär zur LOD-Stufe i eingeblendet.

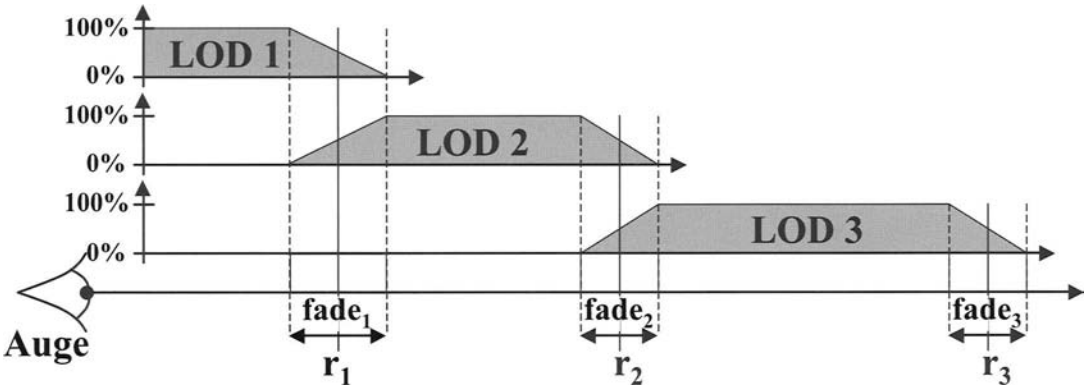


Bild 15.12: *Fade LOD*: Um *pop-up*-Effekte beim Umschalten zwischen verschiedenen LOD-Stufen zu vermeiden, wird mit Hilfe des Alpha-Blendings innerhalb eines Übergangsbereichs $fade_i$ eine LOD-Stufe langsam eingeblendet, während gleichzeitig die vorhergehende LOD-Stufe ausgeblendet wird.

Bei Verwendung des *Fade* LOD ist wegen des vermiedenen *PopUp*-Effekts eine aggressivere, d.h. frühere Umschaltung auf höhere LOD-Stufen möglich. Das entlastet die Geometrie-Stufe der Grafikhardware. Allerdings sind während des Überblendens pro Objekt zwei LOD-Stufen gleichzeitig zu rendern, so dass die Zahl der zu rendernden Polygone genau in dem Moment ansteigt, an dem man sie eigentlich gerade verringern will. Diese beiden Effekte kompensieren sich mehr oder weniger, je nachdem wie stark die Umschaltentfernungen verkürzt werden und wie stark die Polygonzahl zwischen den LOD-Stufen gesenkt wird. Bei der letzten LOD-Stufe sieht die Situation aber besser aus, denn hier müssen nicht zwei LOD-Stufen gleichzeitig dargestellt werden. Nur die höchste LOD-Stufe

wird langsam ausgeblendet, bis das Objekt vollständig verschwunden ist. In der Praxis wird *Fade* LOD deshalb meist nur für das Ein- oder Ausblenden der höchsten LOD-Stufe verwendet.

15.3.3 Morph LOD

Stand der Technik zur Vermeidung der *pop-up*-Effekte beim Umschalten zwischen verschiedenen LOD-Stufen ist derzeit das „*Morphing*", d.h. die kontinuierliche Formveränderung zwischen zwei verschiedenen Objektrepräsentationen. Zur Darstellung des Prinzips wird ein extrem einfaches Beispiel gewählt (Bild 15.13): Ein Viereck, das in der höheren LOD-Stufe 3 aus zwei Dreiecken besteht und in der niedrigeren LOD-Stufe 2 aus vier Dreiecken, deren Spitzen sich im Mittelpunkt des Vierecks treffen. Zum Zeitpunkt des Umschaltens von LOD-Stufe 3 auf Stufe 2 ist die Form der beiden Repräsentationen identisch, d.h. die vier Dreiecksspitzen, die sich im Mittelpunkt des Vierecks treffen, liegen in der Ebene des Vierecks. Wenn die vier Dreiecke der LOD-Stufe 2 die gleiche Farbe oder Textur wie die zwei Dreiecke der LOD-Stufe 3 besitzen, ist kein Unterschied im Bild der beiden Repräsentationen vorhanden. Im Übergangsbereich der Breite *morph* werden die vier Dreiecksspitzen linear angehoben, von der Entfernung r_2 bis zu $(r_2 - morph)$. Dadurch ändert sich im Übergangsbereich die Form des Objekts kontinuierlich von einer Repräsentation zur nächsten.

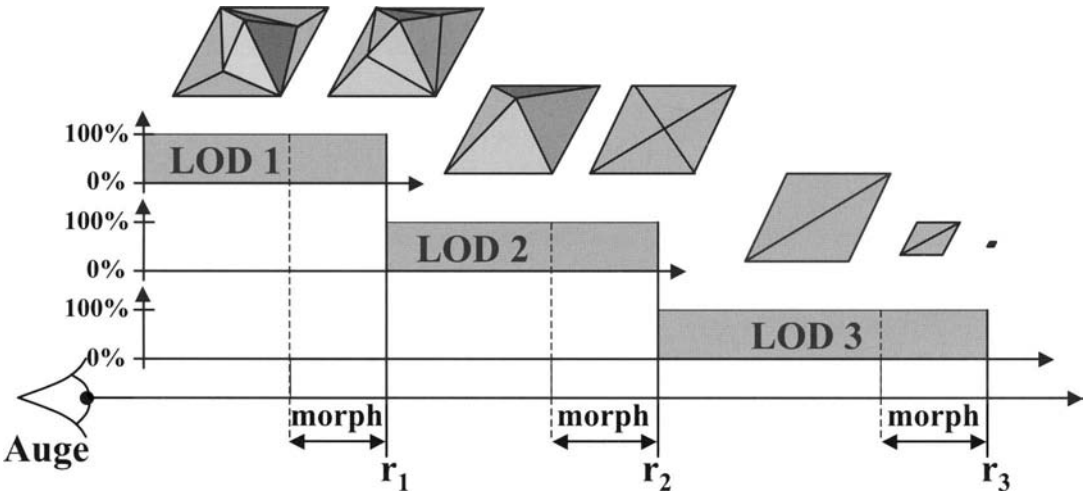


Bild 15.13: *Morph LOD*: Um *PopUp*-Effekte beim Umschalten zwischen verschiedenen LOD-Stufen zu vermeiden, wird innerhalb eines Übergangsbereichs die Form zwischen den beiden LOD-Stufen linear interpoliert. Bei den Umschaltentfernungen r_i besitzen die unterschiedlich fein tessellierten Objektrepräsentationen die gleiche geometrische Form, so dass der Umschaltvorgang nicht bemerkt werden kann.

Das LOD *Morphing* verursacht bei komplexeren Modellen im Übergangsbereich einen erheblichen Rechenaufwand zur Interpolation der Vertexpositionen, der innerhalb des *Cull Traversals*, d.h. auf der CPU-Seite anfällt. Bei modernen Grafikkarten, die über programmierbare Vertex-Shader verfügen (Abschnitt 12.3), können die *Morphing*-Berechnungen jedoch durch die Hardware beschleunigt werden, so dass sie nicht mehr so stark ins Gewicht fallen.

Eine Erweiterung des LOD *Morphing* stellt die Technik des *Continuous Level Of Detail* (CLOD) dar. Dabei wird nicht eine diskrete Anzahl verschieden detaillierter Objektrepräsentationen vorab erzeugt und in einem LOD-Knoten gespeichert, sondern das Polygonnetz des Originalmodells wird während der Laufzeit abhängig von der Entfernung zum Augenpunkt ausgedünnt. Dadurch entsteht für jede Entfernung ein eigenes Modell mit einer adäquaten Anzahl an Polygonen. Diese Technik besitzt jedoch neben dem Rechenaufwand zur Erzeugung der ausgedünnten Modelle noch einen Nachteil: nicht alle Varianten der kontinuierlich ausgedünnten Modelle sehen gut aus und sie können auch nicht im Vorhinein inspiziert werden, wie die diskreten Stufen eines LOD-Knotens.

15.4 Billboards

Eine weitere wichtige Methode zur Reduktion der Polygonzahlen ist der Einsatz von Foto-Texturen, wie in Kapitel 13 ausführlich geschildert. Allerdings lassen sich mit normalen Foto-Texturen auf ortsfesten Polygonen nicht alle Situationen befriedigend beherrschen. Dazu zählt z.B. die Darstellung von Pflanzen (Bäume, Sträucher, Farne usw.), die eine fraktale Geometrie aufweisen (Abschnitt 29.2). Mappt man z.B. das Foto eines Baumes als Farb-Textur mit Transparenzkomponente (RGBA) auf ein feststehendes Viereck, so sieht der Baum aus einem Blickwinkel korrekt aus, der senkrecht auf der Viereckfläche steht. Bewegt man den Augenpunkt in einem Kreisbogen um 90° um das Viereck herum, wobei der Blickwinkel immer in Richtung des Vierecks gerichtet bleibt, so degeneriert das Viereck zu einer Linie.

Eine Möglichkeit zur Lösung dieses Problems besteht in der Verwendung eines sogenannten „Billboard[4]"-Knotens[5]. Dies ist ein Endknoten, der ein Viereck mit einer Alpha-Textur enthält, das sich immer zum Augenpunkt hin ausrichtet (Bild 15.14). Egal, aus welcher Richtung man das Objekt betrachtet, es sieht immer korrekt aus. Als Billboard eignen sich alle einigermaßen rotationssymmetrische Objekte, insbesondere Bäume und Sträucher, aber auch Personen, die am Geh- oder Bahnsteig stehen, Straßenlaternen oder z.B. ein Brunnen. Dies sind alles Beispiele für zylindersymmetrische Objekte, die eine ausgezeichnete Drehachse besitzen. Zur Darstellung solcher Objekte wählt man Billboards, die sich nur um eine Achse drehen können (deshalb spricht man von „axialen" Billboards).

In der Praxis von geringerer Relevanz sind Billboards, die sich beliebig um einen Punkt drehen können, so dass der Normalenvektor der Fläche immer in Richtung des Augenpunkts

[4]Billboard heißt wörtlich aus dem Englischen übersetzt „Reklamefläche" oder „Anzeigetafel".

[5]Alternativ zu Billboards werden in der Praxis insbesondere für Bäume manchmal auch zwei oder drei statische Vierecke eingesetzt, die sich in der Mitte durchdringen.

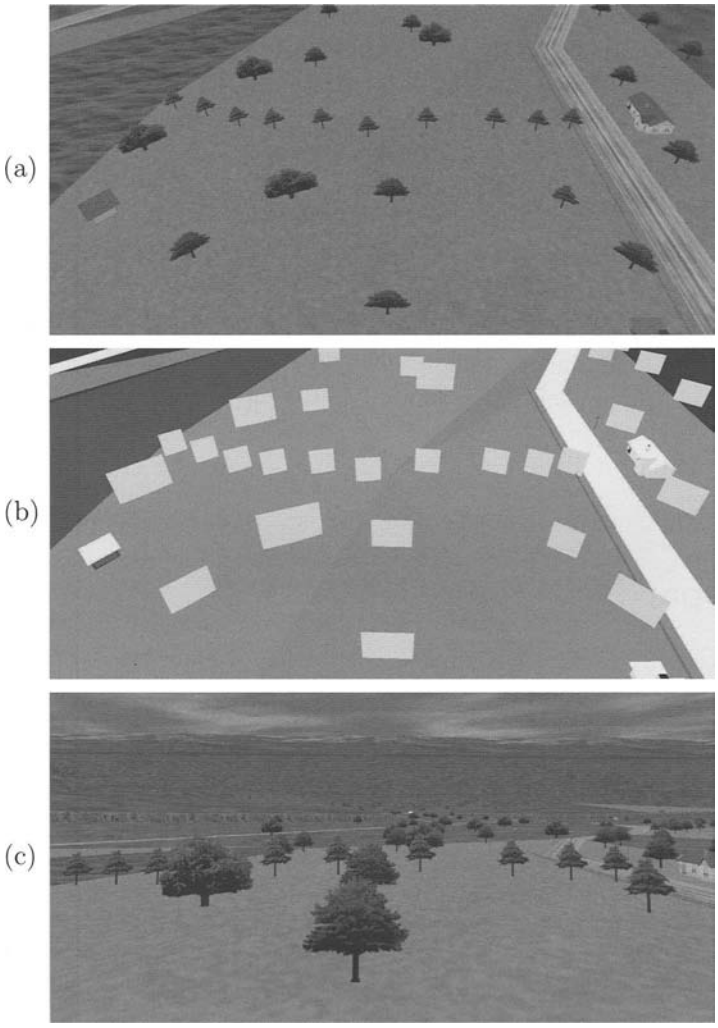


Bild 15.14: *Billboards* sind Vierecke mit einer Alpha-Textur, die sich immer zum Augenpunkt hin ausrichten. (a) Eine Landschaft mit vielen Billboard-Bäumen aus der Vogelperspektive. Die Billboards ordnen sich in konzentrischen Kreisen um die Projektion des Augenpunkts auf das Gelände an. (b) Die selbe Szene, wie in dem oberen Bild, aber diesmal ohne Texturierung. Hier fällt die konzentrische Anordnung der Billboards noch stärker ins Auge. Außerdem erkennt man, dass ein Billboard nur ein Viereck ist, das erst durch den Einsatz der Alpha-Komponente bei der Textur zu einem Baum mit einer komplexen Silhouette wird. (c) Der selbe Landschaftsausschnitt wie in den oberen Bildern, aber diesmal aus einer sinnvollen Perspektive nahe am Boden. Aus dieser Sicht sehen die Billboard-Bäume sehr realistisch aus. Diese Bilder stammen aus der „*Town*"-Demodatenbasis, die bei dem Echtzeit-Renderingtool „OpenGL Performer" von der Firma SGI mitgeliefert wird.

zeigt. Solche Billboards eignen sich zur Darstellung kugelsymmetrischer Objekte, wie z.B. einer Rauchwolke oder von Leuchtkugeln eines Feuerwerks.

Die Berechnung der Koordinaten der vier Vertices eines Billboards in Abhängigkeit von der Augenposition erfolgt innerhalb des *Cull Traversals* (Abschnitt 15.1) beim Durchlaufen des Szenen Graphen.

Für einen erfolgreichen Einsatz von Billboards sind die folgenden Einschränkungen und Probleme zu beachten:

- Man darf dem Billboard nicht zu nahe kommen, denn erstens reicht dann häufig die Texturauflösung nicht mehr aus, so dass das Objekt unscharf wird, und zweitens bemerkt man bei zu nahem Vorbeigehen, dass sich das Billboard gegenüber dem Untergrund dreht. Beide Effekte tragen dazu bei, dass die Illusion eines komplexen 3-dimensionalen Objekts zusammenbricht.

- Normale Billboard-Bäume oder -Personen bewegen sich nicht, da sie ja nur aus einer Textur bestehen. Dies wirkt bei etwas längerer Betrachtung unrealistisch, da sich die Blätter eines Baumes immer etwas bewegen und auch Personen nie ganz still stehen.

- Werden z.B. Bäume mit einem Billboard modelliert, das um einen Punkt rotieren kann, neigen sich die Bäume aus der Vogelperspektive auf den Boden. Es sieht so aus, als wären alle Bäume gefällt worden. Für solche Objekte muss man daher axiale Billboards verwenden.

- Axiale Billboards sind nur für Beobachterpositionen geeignet, bei denen der Vektor vom Billboard zum Augenpunkt einigermaßen senkrecht auf der Drehachse des Billboards steht, d.h. der Beobachter muss sich in Bodennähe aufhalten, wie z.B. bei Fahrsimulationen (Bild 15.14-c). Bei einer Flugsimulation, bei der der Beobachter steil nach unten blickt, wirken axiale Billboards unrealistisch, da sie sich in konzentrischen Kreisen um die Projektion des Augenpunkts auf das Gelände anordnen und nur noch wie ein „Strich in der Landschaft" aussehen.

- Axiale Billboards sind blickwinkelabhängig, d.h. nicht perfekt rotationssymmetrische Objekte, wie z.B. Personen, sehen nur aus der Richtung korrekt aus, aus der sie fotografiert wurden. Versucht man um eine „Billboard-Person" herumzugehen, stellt man fest, dass die Person sich mitdreht, so dass man die andere Seite der Person nie sehen kann.

Einige der angesprochenen Probleme bei normalen Billboards lassen sich durch (zum Teil aufwändige) Erweiterungen beheben:

- Blickwinkelabhängige Billboards (in „OpenGL Performer" IBR[6]-Knoten genannt): Zunächst fährt man mit einer Kamera im Kreis um das Objekt herum und macht

[6]IBR = Image Based Rendering. Darunter versteht man eigentlich eine ganze Klasse bildbasierter Rendering-Verfahren, die im Abschnitt 2.4 etwas genauer erklärt werden.

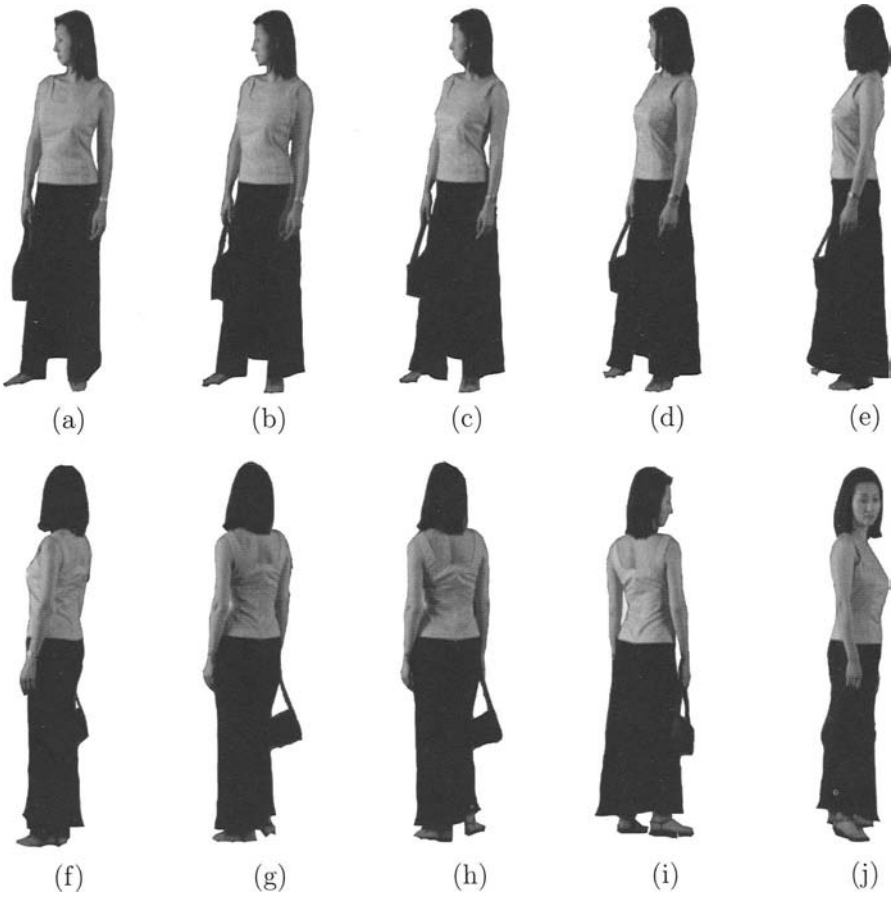


Bild 15.15: Blickwinkelabhängiges *Billboard*: Die Bilder des Modells `florence.rpc` stammen von der Firma ArchVision (Copyright 2001 ArchVision, Inc.) und werden bei der Demodatenbasis des Echtzeit-Renderingtools „OpenGL Performer" von der Firma SGI mitgeliefert.

dabei z.B. aus 64 verschiedenen Richtungen Fotografien. Diese werden als Texturen mit Richtungsinformation in einem normalen (axialen) Billboard gespeichert. Zur Darstellung des Billboards werden jeweils die beiden Texturen linear überlagert, die dem Richtungsvektor zum Augenpunkt am nächsten liegen. Dadurch erscheint auch ein nicht zylindersymmetrisches Objekt aus jedem (horizontalen) Blickwinkel korrekt. Man kann also um ein beliebiges Objekt herumgehen und erhält immer den richtigen Eindruck (Bild 15.15). Diese Technik lässt sich natürlich auf beliebige Raumwinkel erweitern. Der große Nachteil dieser Technik ist der immense Textur-Speicherplatzbedarf.

- Animierte Texturen auf Billboards:
  Durch den Ablauf von kurzen Bildsequenzen (1–5 *sec*) auf einem Billboard kann der Realitätsgrad gesteigert werden. So kann z.B. eine Person am Bahnsteig einen abgestellten Koffer in die Hand nehmen, eine Fahne flattert bzw. die Blätter eines Baums tanzen ein wenig im Wind, oder die Flammen eines kleinen Feuers züngeln. Auch hier ist der große Nachteil der stark zunehmende Textur-Speicherplatzbedarf.

- Impostors oder Sprites[7]:
  Darunter versteht man die Billboard-Generierung während der laufenden Simulation. Dabei wird ein geometrisch sehr aufwändiges Objekt aus der aktuellen Augenposition direkt in eine Foto-Textur gerendert, die dann auf das Billboard gemappt wird. Ein solcher Impostor kann dann anstelle des komplexen 3D-Modells für eine kurze Zeit genutzt werden, in der sich der Beobachter nicht allzu weit von der Aufnahmeposition entfernt hat. Unter dem Strich entsteht also in einem Bild ein gewisser Zusatzaufwand zur Erzeugung des Impostors, der aber durch die mehrfache Verwendung in den Folgebildern insgesamt zu einer Einsparung führt. Ein gewisses Problem bei diesem Verfahren ist es, automatisch evaluierbare Kriterien festzulegen, nach deren Erfüllung die Impostoren neu erzeugt werden müssen [Aken02].

15.5 Multiprozessorsysteme

Bisher wurde eine Reihe von Beschleunigungs-Algorithmen besprochen, die durch zusätzliche Rechenleistungen auf der CPU-Seite die Aufwände in den später folgenden Stufen der Grafikhardware-Seite meist drastisch reduzieren. Viele Aufgaben in der erweiterten *Rendering Pipeline* einer Echtzeit-Anwendung (Bild 15.16) müssen daher sequentiell, d.h. nacheinander ausgeführt werden. Aus diesem Grund wird hier der gesamte Ablauf der Bildgenerierung noch einmal im Überblick dargestellt.

Am Anfang eines neu zu generierenden Bildes werden zunächst die Eingabewerte des interaktiven Benutzers, wie z.B. Lenkradeinschlag, Gaspedalstellung oder Position der Computer-Maus eingelesen. Daraus errechnet eine Dynamiksimulation – evtl. unter Berücksichtigung einer Kollisionserkennung – die neue Position und Orientierung des Augenpunkts, von dem aus das Bild gerendert werden soll. Anschließend startet der sogenannte *application traversal* (Abschnitt 15.1), bei dem der Szenen Graph und die darin enthaltenen Transformationsmatrizen entsprechend dem neuen Augenpunkt und aller sonstigen Veränderungen angepasst werden. Diese Aktionen, die der Anwendungsprogrammierer selbst codiert, werden im sogenannten „Applikations-Prozess“ (App) zusammengefasst.

Mit dem aktualisierten Szenen Graph kann nun im Rahmen mehrerer *cull traversals* zuerst das *Viewing Frustum Culling*, die entfernungsabhängige Auswahl jeweils einer Detailstufe der LOD-Knoten, die Ausrichtung der Vertices der Billboard-Knoten auf die neue Augenpunktposition und evtl. noch das abschließende Occlusion Culling durchgeführt werden. Für ein effizientes Rendern müssen die verbliebenen Objekte noch nach OpenGL-

[7]Impostor heißt wörtlich aus dem Englischen übersetzt „Betrüger“ und Sprite „Geist“.

Zuständen sortiert und abschließend in eine einfache Display Liste geschrieben werden. All diese Aufgaben werden im sogenannten „Cull-Prozess" zusammengefasst.

Nach ihrer Fertigstellung wird die Display Liste vom sogenannten „Draw-Prozess" an die Grafikhardware übergeben.

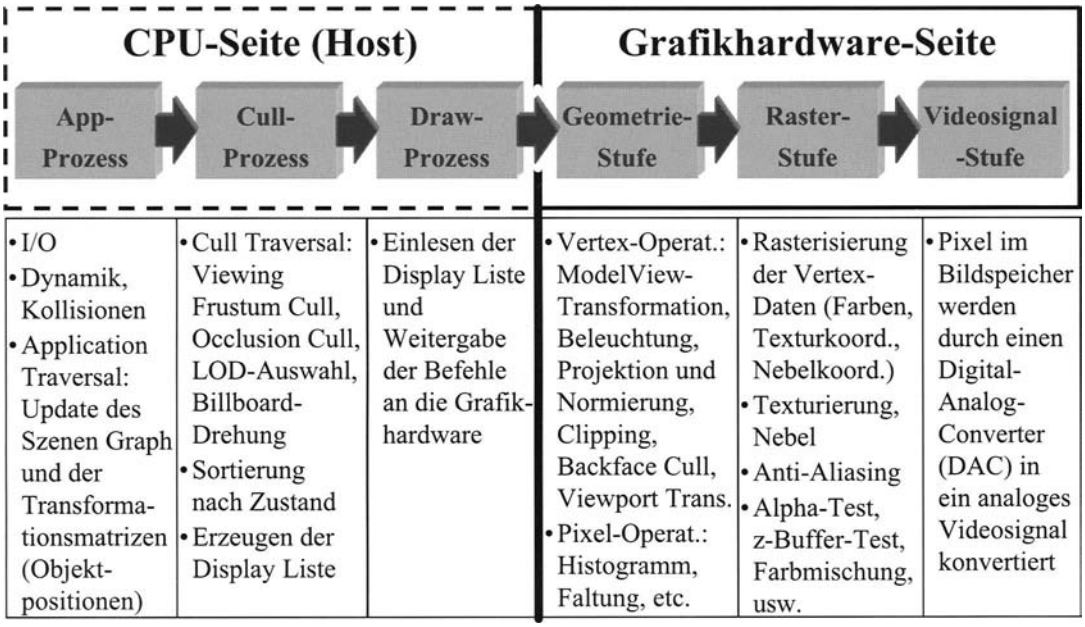


Bild 15.16: Die gesamte „Echtzeit *Rendering Pipeline*". Die Geometrie- und Rasterisierungs-Stufe der Grafikhardware-Seite rechts entsprechen zusammen der „OpenGL *Rendering Pipeline*" in Bild 5.1.

Danach beginnen die Stufen der Rendering Pipeline der Grafikhardware, wie sie ausführlich in Abschnitt 5.2 und den folgenden Kapiteln dargestellt wurden.

In der sogenannten „Geometrie-Stufe" (*Geometry Engine*) werden in erster Linie die vertex-bezogenen Operationen durchgeführt. Dabei werden die Vertices und Normalenvektoren zuerst mit Hilfe der Modell- und Augenpunktstransformation vom Objekt- ins Weltkoordinatensystem überführt. Im Weltkoordinatensystem läuft die Beleuchtungsrechnung für jeden Vertex ab. Nach der anschließenden Projektions- und Normierungstransformation wird das *Clipping* und das *Backface Culling* vorgenommen, bevor die verbliebenen Vertices am Ende noch mit Hilfe der Viewport-Transformation in Bildschirmkoordinaten umgewandelt werden. Parallel dazu können Texturen im Rahmen der Pixel-Operationen z.B. durch Faltungs- oder Histogrammoperationen modifiziert werden, bevor sie in den Texturspeicher geladen werden.

In der zweiten Stufe der Grafikhardware, der sogenannten „Rasterisierungs-Stufe" (*Raster Manager*), werden die vertex-bezogenen Daten (Farbe, Textur- und Nebelkoor-

dinaten) mit Hilfe des Scan-Line-Algorithmus (Abschnitt 12.2) auf das Pixel-Raster abgebildet. Im Rahmen der Fragment-Operationen erfolgt dann zunächst die Texturierung und anschließend die Nebelberechnung, das Anti-Aliasing, der Alpha-Test, der z-Buffer-Test, die Farbmischung, sowie evtl. weitere Operationen (Fenstertest, Stenciltest, Dithering, logische Operationen), die hier nicht dargestellt wurden. Letztendlich werden die Fragmente in den Bildspeicher geschrieben, in dem sie dann Pixel genannt werden.

Zur Darstellung des digital vorliegenden Bildes liest die Videosignalgenerier-Stufe (*Display Generator*) den Bildspeicher aus und konvertiert die Daten mit Hilfe eines Digital-Analog-Converters (DAC) in ein analoges Videosignal, so dass das Bild an einem Bildschirm oder Projektor ausgegeben werden kann.

Nach diesem Überblick kann man nun besser verstehen, wieso die sequentielle Abfolge der einzelnen Rendering-Stufen bei einem Ein-Prozessor-System zu einer schlechten Ausnutzung der Grafikhardware führt. Denn das Grundproblem bei einem Ein-Prozessor-System ist, dass für jedes neue Bild die App- und Cull-Prozesse erst vollständig abgearbeitet sein müssen, bevor der Draw-Prozess durch das Einspeisen von Befehlen aus der Display Liste die Graphik-Hardware starten kann. Bei realen Anwendungen benötigen die App- und Cull-Prozesse zusammen häufig 10 Millisekunden oder mehr, so dass die Graphikhardware während dieser Zeit einfach brach liegt. Bei einer Bildgenerierrate von 60 Hz beträgt die Taktdauer zwischen zwei Bildern (*frame time*) $1/60 \approx 0,0166sec = 16,6$ *msec*, so dass der Grafikhardware nur noch 6,6 *msec* pro Bild, also nur etwas mehr als ein Drittel der maximal möglichen Zeit für das Rendern zur Verfügung stehen. Nicht zu vergessen ist außerdem, dass auch das Betriebssystem eine gewisse Zeit für sich in Anspruch nehmen muss. In Bild 15.17-a ist eine Situation für ein Ein-Prozessor-System dargestellt. Alle drei Prozesse App, Cull und Draw müssen sequentiell innerhalb einer Taktdauer von 16,6 *msec* ablaufen, so dass jedem einzelnen Prozess nur noch ca. ein Drittel dieser Zeit zur Verfügung steht.

Bei einem Zwei-Prozessor-System kann man z.B. die App- und Cull-Prozesse auf der CPU 1 ablaufen lassen und den Draw-Prozess auf CPU 2. Der große Vorteil dabei ist, dass durch die zusätzlichen Prozessoren die Bildgenerierrate um einen Faktor 2-3 oder sogar noch mehr gesteigert werden kann. Denn während auf der CPU 1 schon der App- und Cull-Prozess für das Bild Nr. 2 gerechnet wird, steht dem Draw-Prozess und somit auch der Grafikhardware die gesamte Taktdauer von 16,6 *msec* zum Rendern von Bild Nr. 1 zur Verfügung[8] (Bild 15.17-b). Falls drei CPUs zur Verfügung stehen, kann sogar jeder der drei Prozesse auf einer eigenen CPU ablaufen (Bild 15.17-c). Dies ermöglicht aufwändigere Berechnungen im App-Prozess (z.B. detailliertere Dynamik und Kollisionen) als auch im Cull-Prozess (z.B. *Occlusion Culling*).

Neben dem höheren Preis ist der Hauptnachteil der Multiprozessor-Pipeline die sprung-

[8]In den eher seltenen Fällen, in denen der App-Prozess z.B. aufgrund komplexer numerischer Berechnungen den Flaschenhals darstellt und die Anforderungen an die Grafikhardware relativ gering sind, kann es auch sinnvoll sein, ein anderes Prozess-Modell zu wählen: Der App-Prozess wird alleine auf CPU 1 gelegt, so dass ihm die gesamte Taktdauer zur Verfügung steht und der Cull- und der Draw-Prozess laufen auf CPU 2.

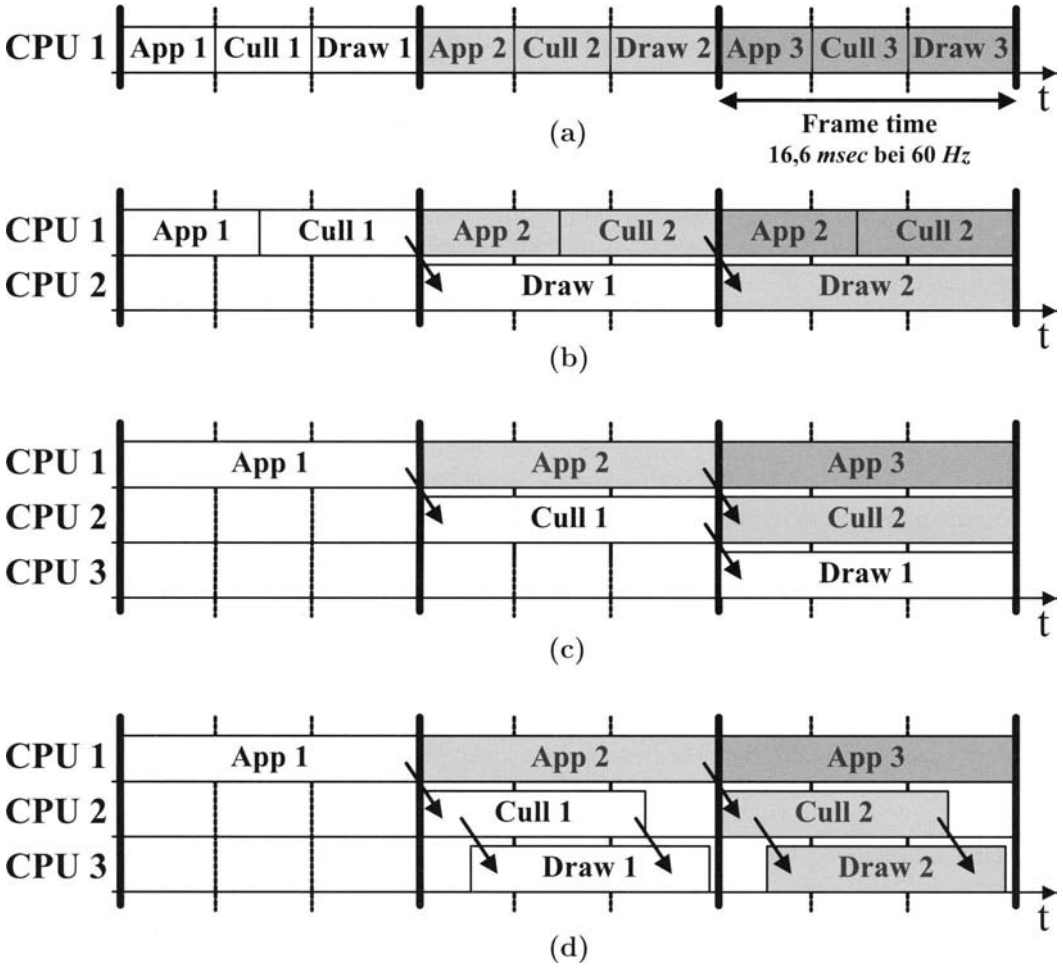


Bild 15.17: Prozessverteilung bei Multiprozessor-Systemen [Ecke04]: Die dicken Striche repräsentieren die Synchronisation zwischen den Prozessen und die Grauwerte der Balken sowie die Indizes geben die Nummer des Bildes an. **(a)** Bei einem Ein-Prozessor-System müssen die drei Prozesse App, Cull und Draw nacheinander und innerhalb eines *Frames* ablaufen, so dass für den Draw-Prozess, der die Grafikhardware antreibt, relativ wenig Zeit übrig bleibt. **(b)** Falls zwei Prozessoren zur Verfügung stehen, ist es meistens sinnvoll, den App- und den Cull-Prozess auf CPU 1 zu legen und den Draw-Prozess auf CPU 2. Dadurch steht den Prozessen deutlich mehr Rechenzeit zur Verfügung. Nachteil dieser Variante ist eine Transport-Verzögerung von einem *Frame*. **(c)** Bei drei Prozessoren kann jeder Prozess auf einen eigenen Prozessor gelegt werden und somit die gesamte Taktdauer ausnutzen. Das fertige Bild verzögert sich aber um zwei *Frames*. **(d)** Durch eine Überlappung von Cull- und Draw-Prozess kann die Transport-Verzögerung um ein Frame gegenüber (c) verringert werden.

artig steigende Transport-Verzögerung. Darunter versteht man die Zeitspanne für den Durchlauf der Signalverarbeitungskette von der Eingabe (z.B. Lenkradeinschlag) bis zur Ausgabe des Bildes auf dem Bildschirm. Bei einer Simulation treten normalerweise mindestens drei *Frames* Transport-Verzögerung auf (ein *Frame* für die Sensorsignaleingabe, ein *Frame* für die Bildgenerierung und ein *Frame* wegen des *Double Bufferings* (Abschnitt 14.1) bzw. der Darstellung des Bildes am Bildschirm). Bei einem Zwei-Prozessor-System, wie in Bild 15.17-b gezeigt, kann der Draw-Prozess erst mit einem *Frame* (d.h. 16,6 *msec*) Verzögerung starten, so dass sich die Transport-Verzögerung um diesen Wert erhöht. Bei drei Prozessoren, wie in Bild 15.17-c gezeigt, startet der Cull-Prozess mit einem *Frame* Verzögerung und der Draw-Prozess erst zwei *Frames* nach dem App-Prozess. Die Transport-Verzögerung erhöht sich in diesem Fall schon auf insgesamt fünf *Frames*, d.h. 83,3 *msec* bei 60 *Hz* Bildgenerierrate. Für „Man-In-The-Loop"-Simulationen ist dies problematisch, denn der Wert liegt über der Wahrnehmungsschwelle von Menschen (ca. 50 *msec*). Neben den in Abschnitt 3.1 bereits besprochenen generellen Maßnahmen zur Verringerung der Transport-Verzögerung (höhere Bildgenerierrate und Bewegungsprädiktion), kann die zusätzliche Verzögerung durch Multiprozessorsysteme teilweise kompensiert werden. Die erste Maßnahme besteht darin, die Aktualisierung der Augenpunktposition erst kurz vor Ende des App-Prozesses vorzunehmen. Dadurch lässt sich die Transport-Verzögerung um knapp ein *Frame* senken. Zweitens kann man durch eine Überlappung von Cull- und Draw-Prozess die Transport-Verzögerung nochmal um ein *Frame* verringern (Bild 15.17-d). Dabei schreibt der Cull-Prozess bereits während des *cull traversals* Geometrie-Daten in einen Ringspeicher, die dann unmittelbar vom Draw-Prozess ausgelesen und an die Grafikhardware übergeben werden. Allerdings muss in diesem Fall die Sortierung nach OpenGL-Zuständen im Cull-Prozess entfallen, was häufige Zustandswechsel verursacht und somit die Rendering-Geschwindigkeit der Grafikhardware senkt. Die Überlappung von Cull- und Draw-Prozess ist also nur dann sinnvoll, wenn die Anwendung nicht durch die Grafikhardware limitiert ist.

Es gibt jedoch einige weitere Gründe für den Einsatz einer noch größeren Anzahl an Prozessoren. Falls in einer Anwendung z.B. für viele bewegte Objekte komplexe Dynamik- und Kollisionsberechnungen im App-Prozess nötig sind, können dafür eigene Prozesse von der Applikation abgespaltet und zusätzlichen Prozessoren zugewiesen werden. Bei sehr großen visuellen Datenbasen ist es in der Regel unmöglich oder unbezahlbar die gesamte Datenbasis in den Hauptspeicher zu laden. Deshalb teilt man die Datenbasis auf und lädt die benötigten Teile vorausschauend von der Festplatte während der Laufzeit nach. Dafür generiert man einen eigenen Datenbasis-Nachlade-Prozess (*dbase*), der zur Sicherheit auf einem extra Prozessor abläuft [Ecke04]. Ein weiteres Beispiel für den Bedarf einer großen Anzahl von Prozessoren sind Mehrkanal-Anwendungen. Für einen LKW Fahr- und Verkehrsimulator (Bild 3.1) sind z.B. drei Frontsichtkanäle aufgrund des sehr großen horizontalen Sichtwinkels (210°) erforderlich, sowie drei Rückspiegelkanäle (zwei außen, einer innen). Es sind also sechs eigenständige Bildkanäle gleichzeitig zu rendern. Dies erfordert die sechsfache Grafikleistung, so dass mehrere Grafiksubsysteme (*pipes*) parallel eingesetzt werden müssen. Für jedes Grafiksubssystem sollte ein eigener Draw-Prozess auf einer separaten CPU zuständig sein, damit die Grafikhardware optimal ausgelastet werden kann.

Außerdem ist pro Kanal ein zusätzlicher Cull-Prozess erforderlich, der bei entsprechender Komplexität der visuellen Datenbasis evtl. die volle Taktdauer in Anspruch nehmen muss und somit eine eigene CPU benötigt. Ein solches System kann demnach schnell auf bis zu 16 Prozessoren und 6 Grafiksubsysteme anwachsen. Eine nochmalige Steigerung dieser Zahlen ist erforderlich, falls mehrere komplexe Simulatoren miteinander interagieren sollen.

15.6 Geschwindigkeits-Optimierung

In Bild 15.16 sind die Stufen der „Echtzeit *Rendering Pipeline*" dargestellt, die während der Generierung eines Bildes nacheinander durchlaufen werden müssen. Die langsamste Stufe dieser Pipeline, der sogenannte Flaschenhals (*bottleneck*), bestimmt die erreichbare Bildgenerierrate. Irgend eine Stufe, oder die Kommunikation dazwischen (die Pfeile zwischen den Blöcken), ist immer der Flaschenhals der Anwendung. In diesem Abschnitt werden Tipps gegeben, wie man den oder die Flaschenhälse heraus findet und beseitigt. Die Kunst der Geschwindigkeits-Optimierung besteht jedoch nicht nur im Beseitigen von Flaschenhälsen, denn wenn ein Flaschenhals beseitigt ist, taucht unvermeidlich ein neuer auf, sondern in einer gleichmäßigen Verteilung der Last auf alle Komponenten der Pipeline, um eine harmonische Balance zu erreichen.

Im Folgenden wird davon ausgegangen, dass eine Anforderungsanalyse durchgeführt wurde, um die Auslegung der Hardware, der visuellen Datenbasis und der Applikations-Software auf ein sicheres Fundament zu stellen. Es wird deshalb angenommen, dass die Anforderungen ohne eine Rekonfiguration des Systems ausschließlich mit Hilfe der beschriebenen Optimierungsmaßnahmen erfüllbar sind.

15.6.1 Leistungsmessung

Bevor man sich mit den Maßnahmen zur Leistungsoptimierung beschäftigt, ist es notwendig, sich darüber im Klaren zu sein, „welche" Leistungen „wie" gemessen werden, d.h. welche Messgröße mit welcher Messmethode gemessen wird. Außerdem sollte man sich immer der Tatsache bewusst sein, dass jede Messung an sich das Messergebnis mehr oder weniger verfälscht. Für die Geschwindigkeitsmessung in der Echtzeit 3D-Computergrafik heißt das, je mehr Messdaten über den Renderingvorgang erfasst werden, desto stärker sind diese Messdaten verfälscht, da ja der Messvorgang selber die Resourcen (CPU, Speicher, Bussysteme) der Grafikpipeline in Anspruch nimmt und somit die Rendering-Geschwindigkeit senkt. Um die Auswirkungen der Messung möglichst klein zu halten, kann es daher notwendig sein, viele Testläufe mit einer Anwendung durchzuführen, um bei jedem einzelnen Testlauf nur einen kleinen Teil der Messdaten zu erfassen. Bei einer Verringerung der erfassten Messwerte sollte sich das Ergebnis asymptotisch an einen Grenzwert annähern, der dem „wahren" Wert entspricht.

15.6.1.1 Messgrößen

Die zentrale Größe der Geschwindigkeits-Optimierung ist die Bildgenerierrate (*frame rate*), d.h. die Anzahl der Bilder, die pro Sekunde generiert werden können. Die Bildgenerierrate ist ein Maß für die Gesamtleistung aller Komponenten einer Grafik-Applikation. Für eine detailliertere Analyse ist die Leistung der einzelnen Komponenten zu messen. In der ersten Detailstufe unterscheidet man die CPU-Seite und die Grafikhardware-Seite (Bild 15.16). Die Leistung der CPU-Seite wird durch die Dauer des App- und des Cull-Prozesses bestimmt. Die Leistung der Grafikhardware-Seite kann in erster Näherung durch die Dauer des Draw-Prozesses charakterisiert werden. Dies gilt allerdings nur dann, wenn weder die Draw-CPU noch die Busbandbreite zwischen Draw-CPU und Grafikhardware den Flaschenhals darstellen. Eine genauere Leistungsmessung der Grafikhardware-Seite kann nur indirekt erfolgen, da die Grafikhardware in der Regel keine hochauflösenden *Timer*-Funktionen für den Anwendungsprogrammierer zur Verfügung stellt. Die Leistung der einzelnen Stufen der Grafikhardware-Seite werden durch die folgenden Messgrößen beschrieben:

- Polygon-Rate:
  Die Polygon-Rate ist ein Maß für die Leistung der Geometrie-Stufe, die angibt, wie viele Dreiecke pro Sekunde transformiert, beleuchtet und geclippt werden können.

- Pixelfüll-Rate:
  Die Pixelfüll-Rate ist ein Maß für die Leistung der Rasterisierungs-Stufe, die angibt, wie viele Pixel pro Sekunde rasterisiert, texturiert usw. und in den Bildspeicher geschrieben werden können.

- Verdeckungsgrad (*depth complexity*):
  Um zu beurteilen, ob eine gegebene Pixelfüll-Rate für eine bestimmte Auflösung und Bildgenerierrate ausreichend ist, muss allerdings noch der Verdeckungsgrad der Szene (*depth complexity*) gemessen werden. Darunter versteht man, wie viele Fragmente pro Pixel im Mittel berechnet werden bzw. wie viele Flächen im Durchschnitt hintereinander liegen. Angenommen zwei bildschirmfüllende Polygone liegen übereinander, dann müssen für jedes Pixel zwei Fragmente berechnet werden, so dass der Verdeckungsgrad 2 ist. Kommt noch ein weiteres Polygon dazu, das die Hälfte des Bildschirms füllt, müssen für die eine Hälfte der Pixel je drei Fragmente berechnet werden und für die andere Hälfte je zwei, so dass der Verdeckungsgrad 2,5 beträgt. Verhältnismäßig hohe Werte beim Verdeckungsgrad ergeben sich z.B. in einer Stadt, in der viele Häuser hintereinander stehen und sich verdecken (Bild 15.18).

- Videobandbreite:
  Die Videobandbreite ist ein Maß für die Leistung der Videosignalgenerier-Stufe, die angibt, wie viele Pixel pro Sekunde aus dem Bildspeicher ausgelesen und mit Hilfe des Digital-Analog-Converters (DAC) in ein analoges Videosignal umgewandelt werden können.

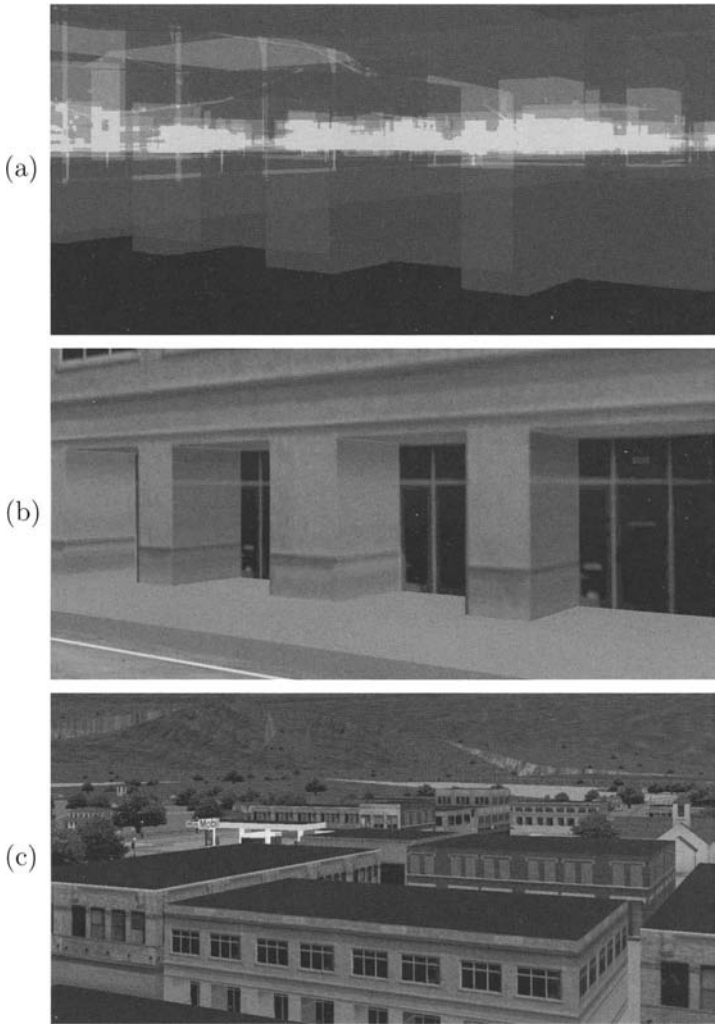


Bild 15.18: Verdeckungsgrad (*depth complexity*): Darunter versteht man, wie viele Fragmente pro Pixel im Mittel berechnet werden bzw. wie viele Flächen im Durchschnitt hintereinander liegen. (a) „Röntgenblick" auf eine Stadtszene: je mehr Flächen hintereinander liegen, desto höher der Grauwert. (b) Die selbe Szene wie unter (a) im normalen Renderingmodus. Die Hauswand verdeckt alle dahinter liegenden Objekte. (c) Die Stadtszene aus der Vogelperspektive. Der Blick auf die dahinger liegenden Objekte ist frei. Diese Bilder stammen aus der „*Town*"-Demodatenbasis, die bei dem Echtzeit-Renderingtool „OpenGL Performer" von der Firma SGI mitgeliefert wird.

Eine wichtige Rolle spielt auch die Bandbreite der Verbindungen zwischen der Grafikhardware, dem Hauptspeicher und den CPUs.

Generell ist zu beachten, dass die Herstellerangaben zu diesen Messgrößen in der Regel nur unter extrem eingeschränkten Bedingungen gültig sind. Für reale Anwendungen müssen die Leistungsangaben der Hersteller oft um eine Größenordnung oder mehr abgesenkt werden. Bei der Auslegung der Hardware sollte man daher nicht auf die Zahlen in den Prospekten vertrauen, sondern sie möglichst mit der eigenen Anwendung oder eigenen Messprogrammen auf einem Testsystem messen.

15.6.1.2 Messmethoden und -werkzeuge

Voraussetzung für verlässliche Messergebnisse ist, dass alle externen Einflüsse auf das System abgestellt und alle unnötigen Dienste (*services / deamons*) abgeschaltet werden. Gemessen wird in erster Linie die Zeit (Dauer des App-, Cull- oder Draw-Prozesses sowie der Bildgenerierung). Deshalb benötigt man hochauflösende *Timer*-Funktionen, die das Betriebssystem zur Verfügung stellen muss. Bei der Messung von Bildgenerierraten ist außerdem die Quantisierung durch die eingestellte Bildwiederholrate für den Bildschirm zu beachten. Im *Double Buffer* Modus wird normalerweise die Vertauschung von *front* und *back buffer* (GLUT-Befehl `glutSwapBuffers()`, Abschnitt 14.1) mit dem vertikalen Rücksprung des Bildschirm-Elektronenstrahls synchronisiert. Deshalb wartet eine solche Applikation mit dem Beginn des Renderns des nächsten Bildes, bis das vertikale Rücksprung-Signal eingegangen ist. Um diesen Quantisierungseffekt bei der Messung der Bildgenerierrate zu vermeiden, sollte man deshalb im Single-Buffer Modus rendern.

Das Echtzeit-Renderingtool „OpenGL Performer" von SGI bietet einen ausgezeichneten Satz an Statistik-Werkzeugen, mit denen nicht nur die Bildgenerierrate und die Dauer von App-, Cull- oder Draw-Prozessen gemessen werden kann, sondern z.B. auch der Verdeckungsgrad einer Szene (Bild 15.18), Anzahl und Typ von Grafikprimitiven, Texturen, Lichtquellen und OpenGL-Zuständen im sichtbaren Volumen, sowie die Anzahl der getesteten und eliminierten Knoten des Szenen Graphen während des *Cullings* [Ecke04]. Eine Alternative dazu ist die Programmierung eines eigenen *Benchmarks* mit Hilfe der „*SPECglperf*"-Applikation[9]. Diese Applikation ermöglicht die Vermessung der gesamten OpenGL Rendering Pipeline mit Hilfe skriptbasierter Testrahmen. Weitere Hinweise zur Leistungsmessung und zu Optimierungsmaßnahmen sind in [Cok01] zu finden.

15.6.1.3 Schnelltests zur Bestimmung des Flaschenhalses

Im Folgenden sind einige einfache, aber effektive Testmethoden beschrieben, um den Flaschenhals einer Anwendung ohne großen Aufwand schnell bestimmen zu können.

- CPU-Limitierung:
  Zur Feststellung, ob die CPU-Seite der Flaschenhals der Anwendung ist, löscht man alle Aufrufe der Grafikprogrammierschnittstelle, d.h. alle OpenGL-Befehle oder ersetzt sie durch *Dummies*. Somit wird die Grafikhardware überhaupt nicht eingesetzt.

[9]Ein Link zur „*SPECglperf*"-Applikation ist auf der Webseite zu diesem Buch zu finden

Falls die Taktrate der Anwendung konstant bleibt, kann die Grafikhardware offensichtlich nicht der Flaschenhals sein, sondern es muss die CPU-Seite sein. Um heraus zu finden, ob der App- oder der Cull-Prozess der Flaschenhals ist, löscht man auch noch den Aufruf des Cull-Prozesses. Bleibt die Taktrate der Anwendung immer noch unverändert, ist der App-Prozess der Flaschenhals, andernfalls der Cull-Prozess.

- Grafikhardware-Limitierung:
  Falls der erste Test eine deutliche Erhöhung der Taktrate der Anwendung ergibt, ist der Flaschenhals höchstwahrscheinlich die Grafikhardware oder evtl. auch die Busbandbreite zwischen der CPU-Seite und der Grafikhardware. Weitere Einzelheiten findet man durch folgende Tests heraus:

  - Geometrie-Limitierung:
    Die Geometrie-Stufe der Grafikhardware ist häufig dann der Flaschenhals, wenn die Modelle eine sehr hohe Polygonzahl aufweisen oder wenn viele lokale Lichtquellen aktiv sind. Deshalb sollte man testen, ob durch eine Verringerung der Polygonzahl und/oder der Lichtquellen die Bildgenerierrate steigt. Ist dies der Fall, so wird höchstwahrscheinlich die Geometrie-Stufe der Flaschenhals sein (evtl. könnte auch die Busbandbreite zwischen CPU-Seite und Grafikhardware der Flaschenhals sein, siehe unten). Eine elegante Methode, um die Polygonzahl kontinuierlich zu verringern, ist die Einführung eines globalen LOD-Skalierungsfaktors, der vor der LOD-Auswahl auf alle LOD-Bereichsgrenzen multipliziert wird (Abschnitt 15.3). Erhöht man den LOD-Skalierungsfaktor, wird bei geringeren Entfernungen auf eine höhere LOD-Stufe umgeschaltet, so dass sich die Polygonzahl entsprechend verringert.
  - Pixelfüll-Limitierung:
    Bei sehr hoher Auflösung oder hohem Verdeckungsgrad ist häufig die Rasterisierungs-Stufe der Grafikhardware der Flaschenhals. Um dies zu testen, bietet es sich daher an, einfach die Auflösung des Bildschirmfensters zu verkleinern. Führt dies zu einer Erhöhung der Bildgenerierrate, ist die Pixelfüllleistung der Rasterisierungs-Stufe der limitierende Faktor der Anwendung. Alternative Tests sollten auf die Fragment-Pipeline abzielen: Ausschalten der Texturierung, des Nebels, des Anti-Aliasings, der Farbmischung, sowie des Alpha- und des z-Buffer-Tests. Falls eine dieser Aktionen die Bildgenerierrate erhöht, befindet sich der Flaschenhals in der Rasterisierungs-Stufe.
  - Videobandbreiten-Limitierung:
    Bei extrem hoher Auflösung oder bei Mehrkanal-Applikationen kann es vorkommen, dass die Bandbreite des Digital-Analog-Converters (DAC) nicht ausreicht, um in der verfügbaren Taktdauer alle digitalen Pixel in ein analoges Videosignal zu konvertieren. Dieser Fall ist allerdings sehr selten, da die Videobandbreite normalerweise sehr großzügig bemessen ist und auf die Bildspeichergröße abgestimmt wird. Um diesen Fall auszuschließen, genügt es meist, die Spezifikation

der Grafikhardware zu lesen und mit den Anforderungen der Anwendung abzugleichen. Ansonsten sollte man ein Bild ohne Objekte rendern, so dass die vorhergehenden Stufen der Grafikhardware sicher nicht der Flaschenhals sein können. Steigt durch diese Maßnahme die Bildgenerierrate nicht an, hat man es vermutlich mit einer Videobandbreiten-Limitierung zu tun.

- Busbandbreiten-Limitierung:
  Falls anspruchsvolle Anwendungen, mit einer großen Menge an Geometriedaten und/oder Texturdaten auf einem Rechnersystem mit geringer Busbandbreite (z.B. kein eigener Grafikbus oder „nur" AGP 4-fach) laufen, ist öfters auch die Busbandbreite der Flaschenhals. Hier bieten sich zwei Tests an: Erstens, das Verpacken der Geometrie in compilierten OpenGL Display Listen (Abschnitt 5.2.1), die im Hauptspeicher abgelegt werden und dort von der Grafikhardware über einen „*Direct Memory Access*" (DMA) abgerufen werden[10]. Der separate DMA-Kanal erhöht die effektive Busbandbreite. Zweitens, eine Reduktion der Texturen, so dass sie sicher in den Texturspeicher passen (Vermeidung von Texturnachladen während der Laufzeit).

15.6.2 Optimierungsmaßnahmen

Mit dem Wissen, an welchen Stellen der „Echtzeit *Rendering Pipeline*" die Flaschenhälse sitzen, kann jetzt in einem iterativen Verfahren – Flaschenhals beseitigen, nächsten Flaschenhals bestimmen usw. – die Rendering-Geschwindigkeit so weit gesteigert werden, dass die geforderte Bildgenerierrate (hoffentlich) überall eingehalten werden kann. Im Prinzip gibt es drei Bereiche, in denen Optimierungsmaßnahmen zur Beseitigung eines Flaschenhalses durchgeführt werden können: Die Hardware, die visuelle Datenbasis und die *Rendering Pipeline*. Auf die Hardware wird hier nicht näher eingegangen, da man davon ausgeht, dass eine sinnvolle Auslegung der Konfiguration stattgefunden hat[11] (Abschnitt 15.5). Im Folgenden werden die wichtigsten Optimierungsmöglichkeiten für die Software aufgelistet.

Optimierungsmöglichkeiten bei der visuellen Datenbasis:

- Räumlich hierarchische Organisation der Szene:
  Dies ist eine wesentliche Voraussetzung für einen effizientes *Viewing Frustum Culling* (Abschnitt 15.1).

- LOD-Einsatz:
  Einsatz der *Level-Of-Detail*-Technik bei möglichst allen Teilen der visuellen Datenbasis, insbesondere auch dem Terrain. Verwendung von Morph-LOD, um bei möglichst

[10] Falls die Hardware überhaupt einen DMA-Kanal anbietet

[11] Falls doch ein Fehler bei der Auslegung der Hardware begangen wurde, ist es manchmal wirtschaftlich besser, zusätzliche Hardware zu kaufen, als einen großen Aufwand in die Optimierung der Software zu stecken.

geringen Abständen vom Augenpunkt auf niedrigere Detailstufen umschalten zu können (Abschnitt 15.3).

- Billboard-Einsatz:
  Ersetzung komplexer geometrischer Objekte durch Billboards (Abschnitt 15.4).

- Minimierung von Zustandswechseln:
  Jedes Objekt in einem Endknoten des Szenen Graphen sollte mit einem einzigen OpenGL-Zustand darstellbar sein. Das bedingt z.B., dass alle verwendeten Texturen für das Objekt in einer großen Textur zusammengefasst werden, dass alle Polygone mit der gleichen Einstellung für Beleuchtung und Schattierung gerendert werden, dass für alle Polygone dasselbe Material verwendet wird und dass nur ein Grafikprimitivtyp verwendet wird.

- Minimierung von Texturen:
  Reduktion der verwendeten Texturen, bzw. Verwendung von Texturen mit weniger Quantisierungsstufen (z.B. 16 bit RGBA), so dass alle Texturen in den Texturspeicher passen. Damit entfällt ein Nachladen von Texturen während der Laufzeit.

- Einsatz effizienter Grafikprimitive:
  Verwendung langer *Triangle Strips* (verbundener Dreiecke), da sie am wenigsten Vertices benötigen und die Hardware in der Regel darauf optimiert ist.

Optimierungsmaßnahmen bei den Stufen der „Echtzeit *Rendering Pipeline*“:

- App-Prozess:
  Der Applikations-Prozess wird optimiert, indem der Code effizienter gestaltet wird, Compiler-Optimierungsoptionen aktiviert werden und die Speicherverwaltung bzw. -zugriffe minimiert wird. Für eine detailliertere Darstellung wird auf [Cok01] verwiesen.

- Cull-Prozess:
  Das größte Potential liegt hier in der oben schon beschriebenen effizienten Organisation des Szenen Graphen nach räumlich-hierarchischen Gesichtspunkten. Eine geringere Testtiefe beim *Viewing Frustum Culling* senkt den Rechenwand im Cull-Prozess und die Auswirkungen in der Grafikhardware sind vielleicht verkraftbar. Außerdem sollte man testen, ob einer der vielen verschiedenen Sortier-Modi oder das vollständige Weglassen der Sortierung des Szenen Graphen den Cull-Prozess ausreichend entlastet und die Grafikhardware in einem akzeptablen Maße belastet. Das Gleiche gilt für das Weglassen des sehr rechenzeitaufwändigen *Occlusion Cullings*.

- Geometrie-Stufe:
  Für die Geometrie-Stufe der Grafikhardware gibt es zahlreiche Möglichkeiten zur Optimierung, die im Folgenden stichpunktartig aufgelistet sind:

- Verwendung von indizierten Vertex Arrays und vorcompilierten Display Listen zur effizienten Darstellung von Geometrie [Shre05].
- Verwendung der Vektorform von Befehlen, um vorab berechnete Daten einzugeben, und Verwendung der Skalarform von Befehlen, um Daten zu spezifizieren, die zur Laufzeit berechnet werden müssen.
- Vereinfachung der Beleuchtungsrechnung (Abschnitt 12.1.3): Verzicht auf zweiseitige Beleuchtung, lokalen Augenpunkt, lokale Lichtquellen, Spotlichtquellen (bzw. Ersatz durch projektive Lichttexturen), Reduktion der Anzahl von Lichtquellen, bestimmte Anteile der Szene müssen evtl. überhaupt nicht beleuchtet werden, bei statischen Szenen kann die Beleuchtungsrechnung vorab durchgeführt und in Texturen gespeichert werden, Normalenvektoren sollten vorab normiert werden.
- Vermeidung von OpenGL-Abfragen (`glGet*()`) während der Laufzeit. Nötige Abfragen sollten bei der Initialisierung durchgeführt werden.
- Verwendung der OpenGL-Befehle `glTranslate*()`, `glRotate*()`, `glScale*()` und `glLoadIdentity()` anstatt eigener Routinen zur Festlegung von Transformationsmatrizen.
- Erhöhung der Genauigkeit des *Viewing Frustum Culling* zur Verringerung der Polygone (Abschnitt 15.2.1).
- Einsatz von *Occlusion Culling* zur Vermeidung verdeckten Polygone (Abschnitt 15.2.2).
- Verzicht auf *Backface Culling* (Abschnitt 15.2.3). Dies senkt zwar den Aufwand in der Geometrie-Stufe, erhöht ihn aber in der Rasterisierungsstufe. Verlagerung des *Backface Culling* in den Cull-Prozess auf der CPU-Seite.
- Erhöhung des globalen LOD-Skalierungsfaktors (Abschnitt 15.3).
- Impostor-Einsatz (Abschnitt 15.4).

- Rasterisierungs-Stufe:
  Eine Auswahl an Optimierungsmöglichkeiten für die Rasterisierungs-Stufe der Grafikhardware sind im Folgenden stichpunktartig aufgelistet:
  - Einsatz von *Occlusion Culling* zur Vermeidung verdeckten Polygone (Abschnitt 15.2.2). Dies rentiert sich vor allem bei Szenen mit einem hohen Verdeckungsgrad (*depth complexity*).
  - Aktivierung von *Backface Culling* (Abschnitt 15.2.3).
  - Falls der Einsatz von *Occlusion Culling* nicht möglich ist, bringt es Effizienzvorteile, wenn die Objekte von vorne nach hinten sortiert werden, da verdeckte Flächen nach dem z-Buffer-Test nicht mehr in den Bildspeicher geschrieben werden müssen.

- Ausschalten des z-Buffer-Algorithmus für Hintergrundpolygone, die ganz zu Beginn gerendert werden.
- Vermeidung des Bildspeicherlöschens (`glClear()`), wenn sichergestellt ist, dass alle Pixel des Bildes jedesmal neu bedeckt werden.
- Falls der Bildspeicher gelöscht werden muss, sollten Farbwerte und z-Werte mit dem Befehl `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)` gleichzeitig initialisiert werden.
- Verwendung von OpenGL Textur-Objekten zur Vermeidung von Zustandswechseln (Abschnitt 13.1.8).
- Ersetzen eines Texturausschnitts mit dem Befehl `glTexSubImage2D()`, anstatt jedesmal eine neue Textur mit `glTexImage2D()` zu definieren (Abschnitt 13.1.1.1).
- Einsatz einfacherer Texturfilter (Abschnitte 13.1.2 und 13.1.3.3).
- Ersetzung von Multipass Algorithmen durch Mehrfachtexturen (Abschnitte 13.1.1.2 und 13.2).
- Reduktion der Auflösung.
- Ausschalten des Anti-Aliasing (Abschnitt 10.2).

Das oberste Ziel bei visuellen Simulationen ist eine konstante Bildgenerierrate, damit man einen glatten Bewegungseindruck erzielen kann. Deshalb wird häufig ein sehr großer Aufwand bei der Herstellung visueller Datenbasen betrieben, um auch an komplexen Stellen der Datenbasis die Bildgenerierrate halten zu können. Das Echtzeit-Renderingtool „OpenGL Performer“ von SGI bietet als Alternative dazu eine einfache Regelung an [Ecke04]. Als Sollwert vorgegeben wird eine maximal zulässige Dauer der Bildgenerierung (die Führungsgröße), um eine bestimmte Bildgenerierrate halten zu können, als Istwert gemessen wird die benötigte Zeitspanne zur Generierung eines Bildes (die Regelgröße) und zur Beeinflussung der Regelstrecke (hier die *Rendering Pipeline*) dienen als Stellgrößen ein globaler LOD-Skalierungsfaktor (vermindert die Polygon-Last) und die dynamische Verringerung der Bildspeicherauflösung (DVR[12], vermindert die Pixelfüll-Last). Falls die Bildgenerierzeitspanne (der Istwert) einen bestimmten oberen Grenzwert überschreitet (den Sollwert), wird ein Stressfaktor hochgesetzt, der, abhängig von der Einstellung des Reglers, den globalen LOD-Skalierungsfaktor erhöht und/oder die Bildauflösung verringert. Dadurch wird der Stress für die *Rendering Pipeline* vermindert, so dass die Bildgenerierrate gehalten werden kann. Falls die Bildgenerierzeitspanne einen bestimmten unteren Grenzwert unterschreitet, wird der Stressfaktor erniedrigt. Dadurch wird der globale LOD-Skalierungsfaktor wieder zurückgefahren und/oder die Bildauflösung erhöht, so dass

[12] *Dynamic Video Resolution* ist ein spezielles Leistungsmerkmal der „*Infinite Reality*“-Grafikhardware von SGI, bei der von Bild zu Bild die Auflösung des Bildspeichers verändert werden kann. Der Anwender merkt davon fast nichts, da die Videosignalgenerier-Stufe das Bild mit Hilfe von bilinearen Interpolationsfiltern auf die Originalauflösung vergrößert

die ursprüngliche Bildqualität wieder erreicht wird. Diese einfache Zwei-Punkt-Regelung funktioniert bei gutmütigen visuellen Datenbasen bzw. Anwendungen befriedigend. Bei komplexeren Situationen reicht diese Art der Regelung jedoch nicht mehr aus. Das Grundproblem bei dieser Regelung ist die reaktive Natur der Rückkopplung. Steigt die Grafiklast sehr schnell an (z.B. aufgrund eines komplexen Objekts, das gerade ins Sichtfeld kommt), kann erst nach dem Rendern des Bildes festgestellt werden, dass die zulässige Taktdauer überschritten wurde. Beim nächsten Bild, für das jetzt der Stressfaktor hochgesetzt wurde, ist die Grafiklast vielleicht schon gar nicht mehr so groß wie vorher. Die Bildgenerierzeitspanne sinkt jetzt evtl. sogar unter den unteren Grenzwert. Dies führt sehr leicht zu einem Schwingungsverhalten der Regelung.

Deshalb bietet es sich hier an, die Last in der Geometrie- und Rasterisierungs-Stufe der Grafikhardware zu prädizieren. Dazu berechnet man auf der Basis der aktuellen Position, Blickrichtung und Geschwindigkeit des Beobachters, die Position und Blickrichtung zu einem geeigneten zukünftigen Zeitpunkt. Bei konstanter Geschwindigkeit des Beobachters kann die zukünftige Position exakt berechnet werden. Falls sich aufgrund von Manövern die Geschwindigkeit des Beobachters ändert, ist der Fehler, den man bei der Prädiktion der Position begeht, gering, da die Extrapolation nur für einen kurzen Zeitraum erfolgt. Vor der Simulation misst man mit einem adäquaten Raster an allen Positionen und aus allen Winkeln die Grafiklast, die die visuelle Datenbasis verursacht, und speichert die Werte in einer geeigneten Datenstruktur (z.B. einem *quad tree*). Damit kann man für die zukünftige Beobachterposition und -blickrichtung die zu erwartende Grafiklast (mit einem gewissen Fehler) aus der Datenstruktur auslesen, und den Stressfaktor entsprechend anpassen. Diese Art der Lastregelung führt zu sehr viel besseren Ergebnissen, als die einfache Zwei-Punkt-Regelung.

Teil II

Bildverarbeitung

Kapitel 16

Digitale Bilddaten

16.1 Prinzipielle Vorgehensweise bei sichtbeeinflussten Anwendungen

An den Anfang des Bildverarbeitungsteils und dieses Grundlagenkapitels ist die Beschreibung einer „prinzipiellen Vorgehensweise", wie sie bei Anwendungen der digitalen Bildverarbeitung und Mustererkennung beobachtet werden kann, gestellt. Grundsätzlich gibt es zwei Zielsetzungen der digitalen Bildverarbeitung: Die *Bildverbesserung* und die *Extraktion von Information*. Der Schwerpunkt des Bildverarbeitungsteils liegt auf der Informationsextraktion. Die Techniken der Bildverbesserung werden immer dann herangezogen, wenn sie die Informationsextraktion erleichtern oder überhaupt erst ermöglichen.

16.1.1 Sensoren

Die richtige Wahl von geeigneten Sensoren für eine aktuelle Problemstellung ist von grundlegender Bedeutung. Dabei können Kriterien maßgebend sein wie z.B.:

- Welche physikalischen Messgrößen sind für die Charakterisierung der Objekte der aktuellen Problemstellung notwendig?
- Welche Ausschnitte aus dem elektromagnetischen Spektrum liefert ein bestimmter Sensor?
- Welche Datenmenge liefert der Sensor?
- In welcher Zeit muss die Datenmenge verarbeitet werden?
- Ist eine multisensorielle Lösung notwendig?
- usw.

Dabei spielt die richtige Beleuchtung eine wesentliche Rolle. Beispiele dazu sind: Auflicht, Durchlicht, diffuses Licht, Verwendung von Farbfiltern oder Blitz. Ähnliche grundsätzliche Überlegungen müssen auch bei nicht-optischen Verfahren angestellt werden, z.B. bei Nuklear-Scannern, Röntgengeräten, usw.

Ein allgemeiner Grundsatz für diesen Bereich ist: Je mehr Sorgfalt man bei der Sensor- und Lichtauswahl verwendet, desto einfacher werden möglicherweise anschließende Bearbeitungsverfahren. Es ist immer besser, wenn möglich gewisse Probleme schon vor der Digitalisierung zu beseitigen, als dies danach mit hohem Aufwand an Hard- und Software zu versuchen.

16.1.2 Digitalisierung

Zur Digitalisierung der Bilddaten wird man in der Regel auf handelsübliche Geräte wie Framegrabber oder Scanner angewiesen sein. Man sollte sich hier Gedanken über die benötigte Auflösung machen: Ist sie zu gering, so verliert man eventuell benötigte feine Bildstrukturen. Bei zu hoher Auflösung belastet man das verarbeitende System mit großen Datenmengen und, daraus resultierend, mit langen Rechenzeiten.

Hier sollte man nochmals überprüfen, ob der gewählte Sensor auch die benötigten Daten liefert. Wenn sich z.B. zwei Objekte nur in ihrem Gewicht unterscheiden, so wäre es unsinnig, das Merkmal „Gewicht" mit einem Videosensor erfassen zu wollen. Wenn somit der Sensor die charakteristischen Merkmale, die zur Unterscheidung der Objekte einer Problemstellung notwendig sind, nicht aufzeichnet, ist er nicht geeignet. Diese Binsenweisheit wird manchmal vergessen! Es wird dann oft versucht, mit riesigem Aufwand aus den digitalisierten Daten Merkmale zur Unterscheidung zu berechnen, die man mit anderen Sensoren problemlos erhalten hätte.

16.1.3 Vorverarbeitung der Rohdaten

In diesem Schritt werden Fehler, die durch die Aufzeichnung in das digitalisierte Datenmaterial eingefügt wurden, aus den Rohdaten entfernt. Beispiele dazu sind: Rauschen durch die Atmosphäre oder die Elektronik, Verzerrungen durch das Linsensystem, Randabschattungen durch die Beleuchtung, Unschärfe durch Bewegung, usw.

Um diese Fehler zu korrigieren, können Verfahren eingesetzt werden wie die Veränderung von Helligkeit und Kontrast, das Angleichen von Farbauszügen, die digitale Filterung im Orts- und Frequenzbereich oder geometrische Korrekturen.

16.1.4 Berechnung von Merkmalen

Nachdem die Rohdaten vorverarbeitet sind, können Merkmale zur Charakterisierung der Objekte berechnet werden. Dabei wird die Vorgehensweise meistens abgestuft sein: Zunächst werden Merkmale für die Segmentierung des Bildes berechnet, dann Merkmale zur Beschreibung der Segmente und schließlich Merkmale, die es erlauben, die erkannten Segmente zu sinnvollen Objekten zusammenzufügen.

Bei Verwendung eines Videosensors können aus den vorverarbeiteten Rohdaten verschiedene Merkmalstypen berechnet werden:

- Rein *bildpunktbezogene Merkmale* wie: Helligkeit, Farbton oder multispektrale Eigenschaften (Signatur).
- *Umgebungsbezogene Merkmale* wie: Kontrast, Gradient oder andere Maßzahlen für die Oberflächenstruktur (Textur) oder die Form.
- *Zeitbezogene Merkmale*, die die Bewegung oder das allgemeine zeitliche Verhalten erfassen.

16.1.5 Segmentierung des Bildes

Bei der Segmentierung wird das Bild nach Maßgabe der berechneten Merkmale in einheitliche Bereiche (*Segmente*) aufgeteilt. Hier kommen Verfahren zum Einsatz wie die Binärbilderzeugung, multivariate statistische Klassifikatoren, geometrische Klassifikatoren, neuronale Netze oder Operatoren auf der Basis der *fuzzy logic*.

Auch hier ist es wichtig, die passende Wahl zu treffen. Wenn z.B. ein Bild durch ein einfaches statisches Schwellwertverfahren in die Klassen „Bildhintergrund“ und „Objekt“ segmentiert werden kann, so wäre der Einsatz eines neuronalen Netzes überdimensioniert.

16.1.6 Kompakte Speicherung der Segmente

Zur Reduzierung der weiteren Datenmengen und damit der Rechenzeiten wird es häufig sinnvoll sein, die Segmente mit geeigneten Datenstrukturen so zu speichern, dass redundante Informationen eliminiert und die jeweiligen Zugriffsmechanismen erleichtert werden. Beispiele hierzu sind Datenstrukturen wie *run-length*-Code, *quadtrees*, *chain*-Code, Laplace-Pyramiden, usw. Bei der Implementierung von Systemen findet an dieser Stelle der Übergang von der *pixelorientierten Verarbeitung* zur *listenorientierten Verarbeitung* statt.

16.1.7 Beschreibung der Segmente

Als Nächstes schließen sich in der Regel Verfahren zur Beschreibung der Segmente an. Hier können einfache Segmenteigenschaften wie der Flächeninhalt, die Länge des Umfangs oder die Lage des Schwerpunktes schon ausreichend sein. Aufwändigere Algorithmen untersuchen die Form des Segments durch die Analyse der relativen Lage der Bildpunkte des Segments. Aber auch kanten- und linienorientierte Verfahren werden hier verwendet.

16.1.8 Synthese von Objekten

Die in den vorhergehenden Verarbeitungsschritten erkannten und beschriebenen Segmente werden jetzt zu sinnvollen Objekten zusammengefügt. Neben heuristischen Verfahren

werden hier Methoden der künstlichen Intelligenz, wie z.B. semantische Netze oder Expertensysteme, eingesetzt.

16.1.9 Ableitung einer Reaktion

Als letzter Schritt muss aus den analysierten Bilddaten eine Reaktion abgeleitet werden, z.B. das Auslösen eines Alarms, die Steuerung eines Roboters oder das Nachführen einer Kamera. Diese Komponente bezeichnet man als die *Reaktionskomponente* oder *Exekutive* des Systems.

16.1.10 Schlussbemerkung zu Abschnitt 16.1

Bei vielen Anwendungen müssen nicht alle oben aufgezählten Punkte durchlaufen werden. Auch ist es möglich, dass einzelne Verarbeitungsstufen trivial lösbar sind oder einfach ausgelassen werden können.

In den folgenden Abschnitten und Kapiteln, die grob nach diesem Schema gegliedert sind, steht immer die Anwendung im Vordergrund. Der vorliegende Bildverarbeitungsteil gibt somit keine Abfolge von logisch, mathematisch zusammenpassenden Algorithmen wieder. Vielmehr kann es sein, dass bei einer Anwendung die unterschiedlichsten Verfahren kombiniert werden müssen.

16.2 Unterabtastung und Quantisierung

Um Bilder mit Computersystemen verarbeiten zu können, müssen sie in Datenformate umgesetzt werden, die rechnerkompatibel sind. Diese Umsetzung heißt *Digitalisierung*. Für fotografische Vorlagen oder Strichzeichnungen werden hierzu *Zeilenabtaster* (*Scanner*) mit unterschiedlicher technischer Realisierung der Abtastung eingesetzt. Auch Videokameras, die über Analog/Digital-Wandler (*frame grabber*, *video capture card*) an das Computersystem angeschlossen sind, werden zur Digitalisierung verwendet. Moderne digitale Fotoapparate (Digitalkameras) oder Videokameras führen die Digitalisierung schon im Gerät durch und liefern somit bereits digital aufbereitete Daten. In manchen Bereichen, z.B. in der Fernerkundung (Luft- und Satellitenbildauswertung) wird die Digitalisierung mit flugzeug- oder satellitengetragenen *Multispektralscannern* durchgeführt. Im Folgenden wird nicht auf die technische Realisierung der einzelnen Abtastsysteme, sondern vielmehr auf einige grundlegenden Aspekte der Digitalisierung eingegangen.

Zur Einführung werden zunächst eindimensionale Funktionen $s(t)$ einer Variablen t verwendet. Wenn man eine eindimensionale Funktion $s(t)$ digital aufbereiten will, muss man sie *abtasten* und *quantisieren*. Die Abtastung erfolgt entlang der t-Achse (Abszisse). Da nicht alle Werte von $s(t)$ erfasst werden, spricht man hier auch von *Unterabtastung* (*subsampling*). Die Quantisierung wird senkrecht dazu entlang der Ordinate durchgeführt (Bild 16.1).

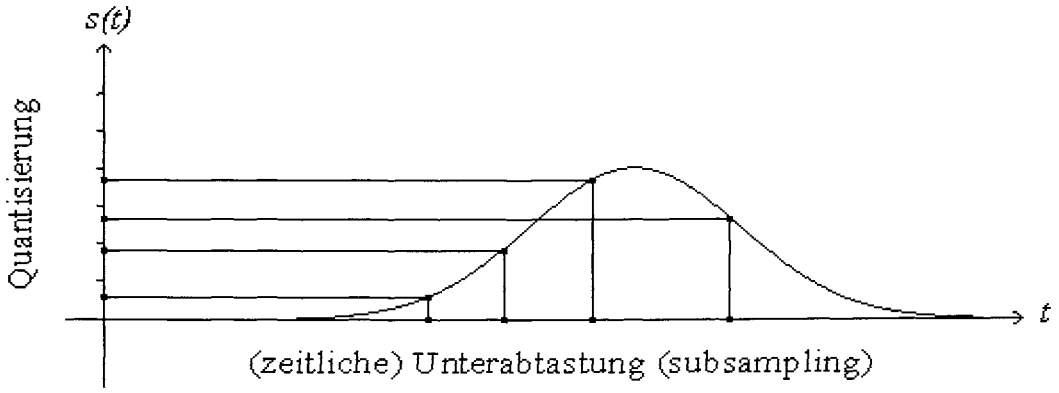


Bild 16.1: Unterabtastung und Quantisierung: Die Unterabtastung erfolgt entlang der Abszisse, die Quantisierung entlang der Ordinate

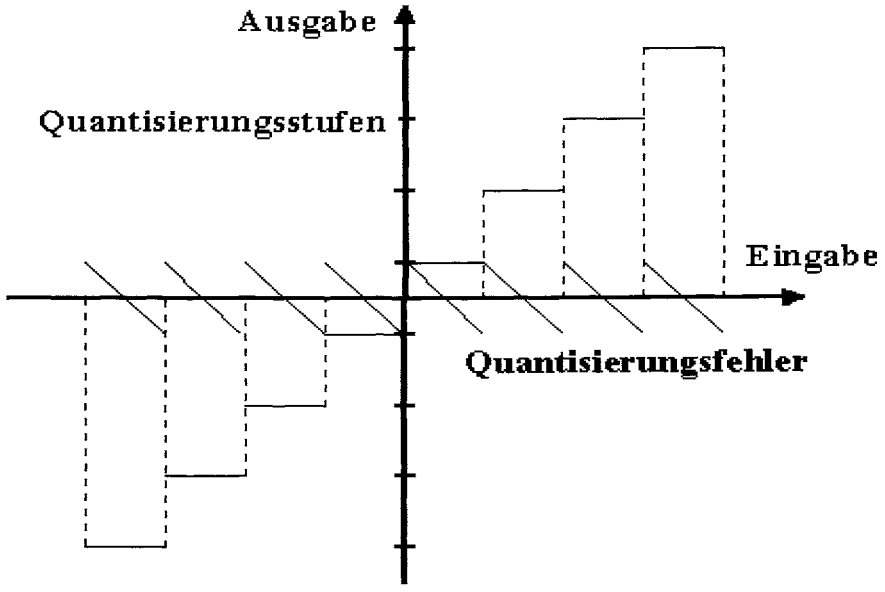


Bild 16.2: Das Prinzip Quantisierung: Intervalle des Eingabesignals werden im Ausgabesignal festen Quantisierungsstufen zugeordnet.

Zunächst einige Bemerkungen zur Abtastung. Man kann sich natürlich fragen, wie eng man die Abtastwerte auf der Abszisse legen muss, damit die originale Funktion $s(t)$ möglichst genau erfasst wird. Dazu gibt das *Abtasttheorem von Shannon* eine Hilfestellung. Man kann unter bestimmten nicht sehr restriktiven Bedingungen eine Funktion $s(t)$ als eine Überlagerung von endlich oder unendlich vielen Sinus- und Cosinus-Funktionen mit unterschiedlichen Frequenzen und Amplituden darstellen (endliche oder unendliche Reihe). Wenn es unendliche viele sind, muss man sich für eine maximale Frequenz entscheiden, da man ja nicht unendlich viele Abtastpunkte erfassen kann. Man spricht dann von einem bandbegrenzten Signal. Das Shannon'sche Abtasttheorem besagt nun, dass man die Abtastfrequenz, also das Setzen der Abtastpunkte, so wählen muss, dass sie mindestens doppelt so groß ist, wie die höchste im bandbegrenzten Signal auftretende Frequenz. Das ist auch anschaulich klar, weil man dann sicher sein kann, dass zwischen zwei Abtastpunkten „nicht mehr viel passiert". Dazu ein Beispiel aus der Praxis: Das menschliche Gehör kann Frequenzen bis etwa 20000 *Hz* wahrnehmen (bei zunehmendem Alter sinkt diese Grenze deutlich). Will man nun ein Audiosignal $s(t)$ auf eine Audio-CD bringen, so muss man also Frequenzen bis maximal etwa 20000 *Hz* erfassen, d.h. dass die Abtastfrequenz bei ca. 40000 *Hz* liegen muss (tatsächlich: 44000 *Hz*).

Nun zur Quantisierung. Sie ist in Bild 16.2 schematisch dargestellt. Das Prinzip: Intervalle des Eingabesignals $s_e(t)$ werden im Ausgabesignal $s_a(t)$ festen Quantisierungsstufen zugeordnet. Sind die Quantisierungsstufen gleichabständig, so spricht man von einer gleichmäßigen Quantisierung (*uniform quantiser*). Sollen Eigenheiten der menschlichen Perzeption berücksichtigt werden, so kann man auch eine nicht gleichmäßige Quantisierung (*nonuniform quantiser*) verwenden. So können z.B. betragsmäßig kleine Werte des Eingabesignals feiner quantisiert werden als betragsmäßig große Werte.

Das digitalisierte Signal $s_a(t)$ ist eine diskrete Funktion der diskreten Variablen t. Da es endlich viele Punkte $(t_i, s_a(t_i)), i = 0, 1, ..., n-1$ sind, kann man sie in einer Tabelle unterbringen:

t	t_0	t_1	...	t_{n-1}
$s_a(t)$	$s_a(t_0)$	$s_a(t_1)$	...	$s_a(t_{n-1})$

Fotografische Bilder kann man als zweidimensionale Funktionen $s(x, y)$ interpretieren: An der Stelle (x, y) tritt die Schwärzung $s(x, y)$ auf. Die digitale Aufbereitung erfolgt im Prinzip wie oben. Bei der (Unter-)Abtastung, die hier oft auch *Rasterung* genannt wird, werden diskrete Positionen (x_i, y_i) festgelegt, an denen der Wert $s_e(x_i, y_i)$ abgelesen wird. Dieser Wert wird dann quantisiert. Da nach der Digitalisierung auch hier endlich viele Werte vorliegen, können sie in einer Tabelle oder besser in einer Matrix gespeichert werden:

	y_0	y_1	y_2	...
x_0	$s_a(x_0, y_0)$	$s_a(x_0, y_1)$	$s_a(x_0, y_2)$	...
x_1	$s_a(x_1, y_0)$	$s_a(x_1, y_1)$	$s_a(x_1, y_2)$	...
x_2	$s_a(x_2, y_0)$	$s_a(x_2, y_1)$	$s_a(x_2, y_2)$	...
...	...	...	...	...

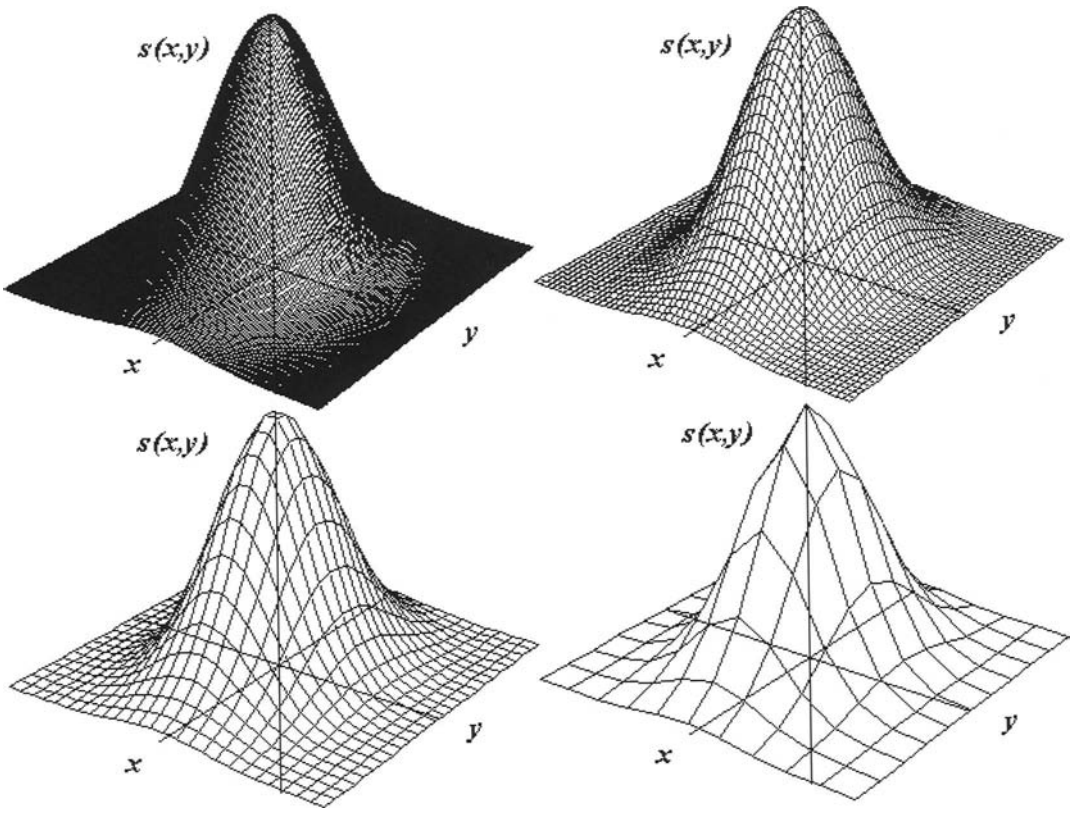


Bild 16.3: Bilder sind zweidimensionale Funktionen, die wie ein eindimensionales Signal gerastert und quantisiert werden. Hier sind vier verschieden große Unterabtastungsbeispiele einer zweidimensionalen Gauß'schen Normalverteilung dargestellt.

Bild 16.3 zeigt ein Beispiel einer zweidimensionalen Funktion (zweidimensionale Gauß'sche Normalverteilung) mit unterschiedlich großer Unterabtastung.

16.3 Digitalisierung von Schwarz/Weiß-Bilddaten

Nachdem in Abschnitt 16.2 das allgemeine Konzept bei der Digitalisierung von Funktionen kurz beispielhaft erläutert wurde, wird in diesem Abschnitt die digitale Aufbereitung einer Schwarz/Weiß-Bildvorlage untersucht. Ein Schwarz/Weiß-Bild ist nur aus zwei Grautönen (schwarz und weiß) zusammengesetzt und könnte z.B. als fotografisches Papierbild oder als Transparent vorliegen. Die Darstellung des Motivs erfolgt durch Schwärzung der fotografischen Schicht: An Stellen, die in der Helligkeit unter einer bestimmten Schwelle liegen, ist die Schicht geschwärzt, an den anderen Stellen ist sie weiß. Zur Digitalisierung sind die zwei Schritte (Unter-)Abtastung (Rasterung) und Quantisierung notwendig.

Bei der Rasterung wird die Bildvorlage durch die Überlagerung eines rechteckigen oder quadratischen Gitters in einzelne Rasterflächenstücke unterteilt. Zur Quantisierung wird jede dieser Rasterflächen entweder nur schwarz oder weiß dargestellt, je nachdem, ob der größere Teil der Rasterfläche ursprünglich weiß oder schwarz war. Wird nun ein weißes Rasterflächenstück symbolisch durch die Zahl 1 und ein schwarzes Rasterflächenstück durch die Zahl 0 dargestellt, so kann das digitalisierte Bild als eine rechteckige Zahlenanordnung (*Bildmatrix*) interpretiert werden. Bild 16.4 veranschaulicht diese grundlegenden Schritte der Digitalisierung.

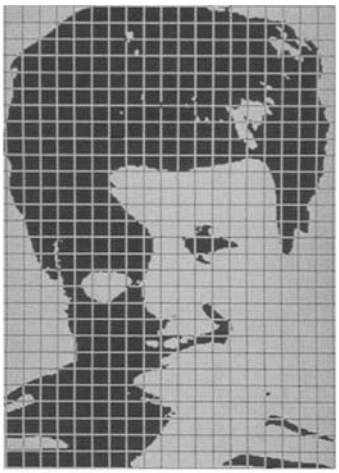
Leider ist in der Praxis die Form der Rasterflächen nicht einheitlich. Wenn z.B. ein Framegrabber oder Scanner die Vorlage mit rechteckigen, nicht quadratischen Rasterflächen verarbeitet, so wird das digitalisierte Bild, dargestellt auf einem Displaysystem mit quadratischem Raster, verzerrt erscheinen. Das ist bei vielen Operationen zwar nur ein Schönheitsfehler. Es gibt aber auch Anwendungen, z.B. bei geometrischen Operationen, wo dieser Effekt störend ist oder sogar zu falschen Ergebnissen führt.

Eine Zeile der Bildmatrix wird als *Bildzeile*, eine Spalte wird als *Bildspalte* und ein Element der Bildmatrix wird als *Bildpunkt* (*Pixel*, *picture element*) bezeichnet. Die Zählung der Bildzeilen erfolgt von oben nach unten, beginnend mit der Bildzeile 0, und die Zählung der Bildspalten erfolgt von links nach rechts, beginnend mit der Bildspalte 0 (Bild 16.5).

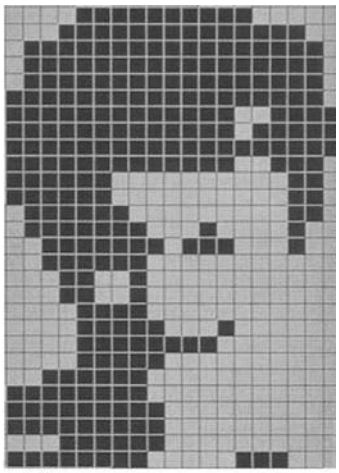
Diese Lage des Koordinatensystems wurde aus zwei Gründen gewählt: Wie bei der Bezeichnung von Matrixelementen in der Mathematik laufen die Indizes von oben nach unten und von links nach rechts. Wenn dieses Koordinatensystem um 90° gegen den Uhrzeigersinn gedreht wird, ist es mit dem in der Mathematik üblichen Koordinatensystem identisch. Das hat den Vorteil, dass Begriffe, wie z.B. die mathematisch positive oder negative Drehrichtung direkt übernommen werden können. Es wird darauf hingewiesen, dass in anderer Bildverarbeitungsliteratur oder in Grafiksystemen das Koordinatensystem anders definiert sein kann. Es wird empfohlen, im jeweiligen Anwendungsfall sich über die konkrete Definition des Koordinatensystem zu informieren.

Die einem Bildpunkt zugeordnete Zahl 0 oder 1 ist der *Grauwert* des Bildpunktes. Da ein digitalisiertes Schwarz/Weiß-Bild nur durch die zwei Grauwerte 0 und 1 dargestellt

(a)



(b)



(c)

(d)

```
1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1
1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 1
0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 1
1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 1
1 0 0 0 0 0 0 0 1 1 1 0 1 1 1 0 1 1
1 1 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 1 1
1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1
```

Bild 16.4: Digitalisierung eines Schwarz/Weiß-Bildes. (a) Original. (b) Rasterung des Originals durch die Überlagerung eines quadratischen Gitters. (c) Quantisierung durch Auffüllen der Rasterflächen mit weiß oder schwarz. (d) Darstellung der schwarzen oder weißen Rasterflächenstücke durch 0 oder 1.

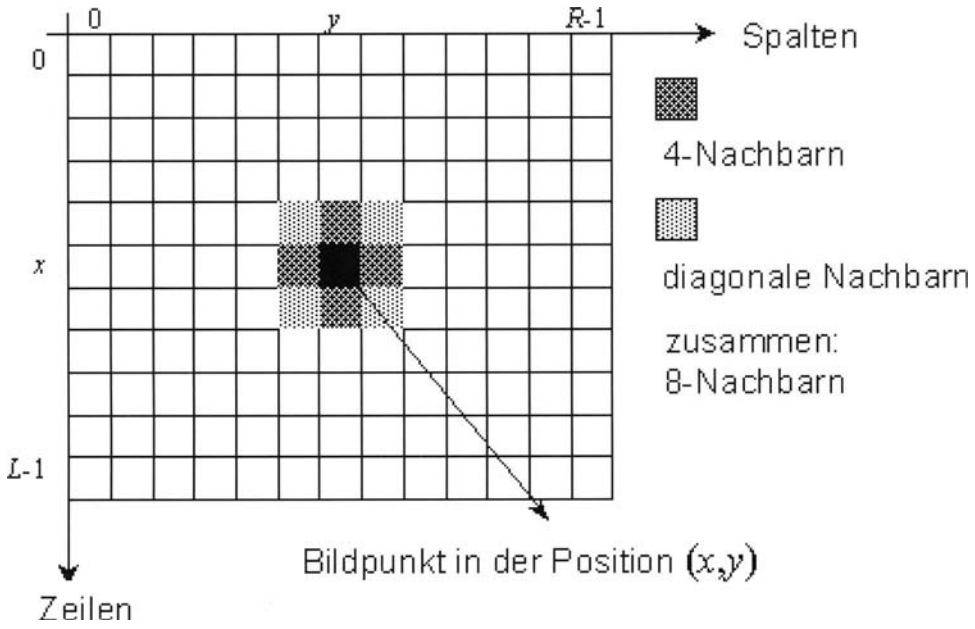


Bild 16.5: Bildmatrix eines digitalisierten Bildes mit Koordinatensystem und 4-Nachbarn und 8-Nachbarn.

wird, bezeichnet man es oft als *Binärbild*. Werden anstatt 0 und 1 beliebige andere Zahlen verwendet, so spricht man von einem *Zweipegelbild*.

Auf Displaysystemen ist es meistens notwendig, Binärbilder mit den Grauwerten 0 (schwarz) und 255 statt 1 (weiß) darzustellen, da man die Graustufen 0 und 1 mit den Augen nicht unterscheiden kann.

Im quadratischen Raster hat jeder Bildpunkt (x, y) zwei Arten von Nachbarn. Die vier Nachbarbildpunkte in den Positionen $(x, y-1)$, $(x, y+1)$, $(x-1, y)$ und $(x+1, y)$ heißen die *4-Nachbarn* des Bildpunktes in der Position (x, y). Zusätzlich hat der Bildpunkt (x, y) noch die vier *diagonalen Nachbarn* in den Positionen $(x-1, y-1)$, $(x-1, y+1)$, $(x+1, y-1)$ und $(x+1, y+1)$. Die 4-Nachbarn und die vier diagonalen Nachbarn zusammen werden als *8-Nachbarn* oder schlicht *Nachbarn* des Bildpunktes (x, y) bezeichnet (Bild 16.5).

In manchen Algorithmen ist es wichtig anzugeben, ob 4- oder 8-Nachbarschaft zugrunde gelegt wird. Bild 16.6 zeigt dazu ein Beispiel: Wenn bei einer Anwendung gezählt werden muss, wie viele Segmente (hier die schwarzen Bildpunkte) sich in einem Bild oder Bildausschnitt befinden, so ist es wichtig, welche Bildpunkte als Nachbarn zählen und somit zum selben Segment gehören. Bei 4-Nachbarschaft enthält Bild 16.6 vier Segmente, bei 8-Nachbarschaft nur zwei.

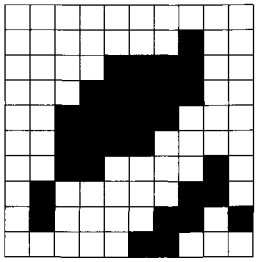


Bild 16.6: Wenn gezählt werden soll, wie viele Segmente (hier die schwarzen Bildpunkte) das Bild enthält, ist es wichtig anzugeben, ob 4- oder 8-Nachbarschaft zugrunde gelegt wird. Bei 4-Nachbarschaft enthält das Bild vier und bei 8-Nachbarschaft zwei Segmente.

Zur Berechnung des *Abstandes* zwischen zwei Bildpunkten (x_1, y_1) und (x_2, y_2) werden unterschiedliche *Distanzmaße* verwendet. Die *euklidische Distanz* berechnet sich gemäß:

$$d_e = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}. \tag{16.1}$$

Für den Abstand eines Bildpunktes zu seinen 4-Nachbarn ergibt sich $d = 1$, während sich für die vier diagonalen Nachbarn $d = \sqrt{2}$ ergibt. Ein anderes Distanzmaß ist die *Schachbrettdistanz*. Hier haben alle 8-Nachbarn dieselbe Distanz $d = 1$ zum Bildpunkt in der Position (x, y):

$$d_s = \max\{|x_1 - x_2|, |y_1 - y_2|\}. \tag{16.2}$$

Abschließend zu diesem Kapitel wird ein erstes mathematisches Modell zur Beschreibung von digitalisierten Bilddaten formuliert. Eine Möglichkeit ist die Notation als (Bild-) Matrizen. Im einfachsten Fall ist das Bild $\mathbf{S}$ (die *Szene* $\mathbf{S}$) eine rechteckige (Bild-)Matrix $\mathbf{S} = (s(x, y))$ mit Bildzeilen und Bildspalten. Der Zeilenzähler (Zeilenindex) ist x und der Spaltenzähler (Spaltenindex) ist y. Der Bildpunkt an der Stelle (Zeile, Spalte)=(x, y) besitzt den Grauwert $s(x, y)$, hier 0 oder 1. Bei exakter Anlehnung an die mathematische Matrizenschreibweise (z.B. $\mathbf{A} = (a_{ij})$) müsste man den Zeilen- und Spaltenzähler tiefgestellt als Index s_{xy} schreiben. Davon wird hier abgewichen, zum einen, weil bei manchen Verfahren die vielen hoch- und tiefgestellten Indizes verwirrend sein können, und zum anderen, weil die Notation $s(x, y)$ auch ausdrücken kann, dass man den Grauwert an der Stelle (x, y) als Funktionswert $s(x, y)$ auffassen kann.

Somit wird ein digitalisiertes Schwarz/Weiß-Bild mit dem folgenden *mathematischen*

Modell beschrieben:

$G = \{0, 1\}$	Grauwertmenge mit den Grauwerten 0 und 1
$\mathbf{S} = (s(x, y))$	Bildmatrix des Bildes
$x = 0, 1, ..., L - 1$	L Bildzeilen
$y = 0, 1, ..., R - 1$	R Bildspalten
(x, y)	Ortskoordinaten des Bildpunktes
$s(x, y) \in G$	Grauwert des Bildpunktes

(16.3)

16.4 Digitalisierung von Grautonbildern

Bei einem Grautonbild erfolgt die Abbildung des Motivs durch unterschiedliche Schwärzung der Fotoschicht. Es treten somit neben schwarz und weiß auch andere Grautöne auf. In der Umgangssprache werden Grautonbilder oft fälschlich als Schwarz/Weiß-Bilder bezeichnet (z.B. Schwarz/Weiß-Fernsehapparat). Bei der Digitalisierung eines Grautonbildes muss nach der Rasterung jeder Rasterfläche ein Grauwert zugeordnet werden. Die Bestimmung dieses Grauwertes kann punktuell erfolgen (Deltafunktion) oder durch Mittelung über die Rasterfläche. Die Mittelung wird dabei in der Regel nicht gleichgewichtig über die gesamte Rasterfläche durchgeführt. Als Grauwertmenge wird meistens $G = \{0, 1, 2, ..., 255\}$ verwendet, da diese 256 Grauwerte mit einem Byte (1 Byte = 8 Bit und $2^8 = 256$) dargestellt werden können. In byteorientierten Rechenanlagen ist damit der Speicherplatz für digitalisierte Bilddaten optimal ausgenützt und die Adressierung der Bildpunkte ist einfach. Ein digitalisiertes Grautonbild ist somit wieder eine Matrix $\mathbf{S} = (s(x, y))$, nur dass jetzt die Grauwerte aus der erweiterten Grauwertmenge G sind.

Die 256 Grauwerte von 0 bis 255 sind in den meisten Fällen ausreichend. Der Grauwert 0 wird dabei in der Regel als schwarz, der Grauwert 255 als weiß und ein Grauwert um 127 als grau interpretiert. Bild 16.7 zeigt die Quantisierung eines Originals mit 64, 32, 16, und 8 Grauwerten. Die Verallgemeinerung der Grauwertmenge G auf eine allgemeine Grauwertmenge $G' = \{z_1, z_2, ..., z_k\}$ ist jederzeit möglich, im Rahmen dieser Darstellung wird jedoch darauf verzichtet.

Mit diesen Festlegungen lässt sich ein digitalisiertes Grauwertbild wie folgt beschreiben:

$G = \{0, 1, ..., 255\}$	Grauwertmenge
$\mathbf{S} = (s(x, y))$	Bildmatrix des Grauwertbildes
$x = 0, 1, ..., L - 1$	L Bildzeilen
$y = 0, 1, ..., R - 1$	R Bildspalten
(x, y)	Ortskoordinaten des Bildpunktes
$s(x, y) = g \in G$	Grauwert des Bildpunktes

(16.4)

In praktischen Problemlösungen werden im Rahmen eines Algorithmus häufig unterschiedliche Verfahren der Reihe nach angewendet: Nach einer nicht linearen Skalierung der Grauwerte könnte z.B. eine Filterung im Ortsbereich folgen, darauf eine geometrische Korrektur, dann die Berechnung von Texturmerkmalen für Umgebungen von Bildpunkten

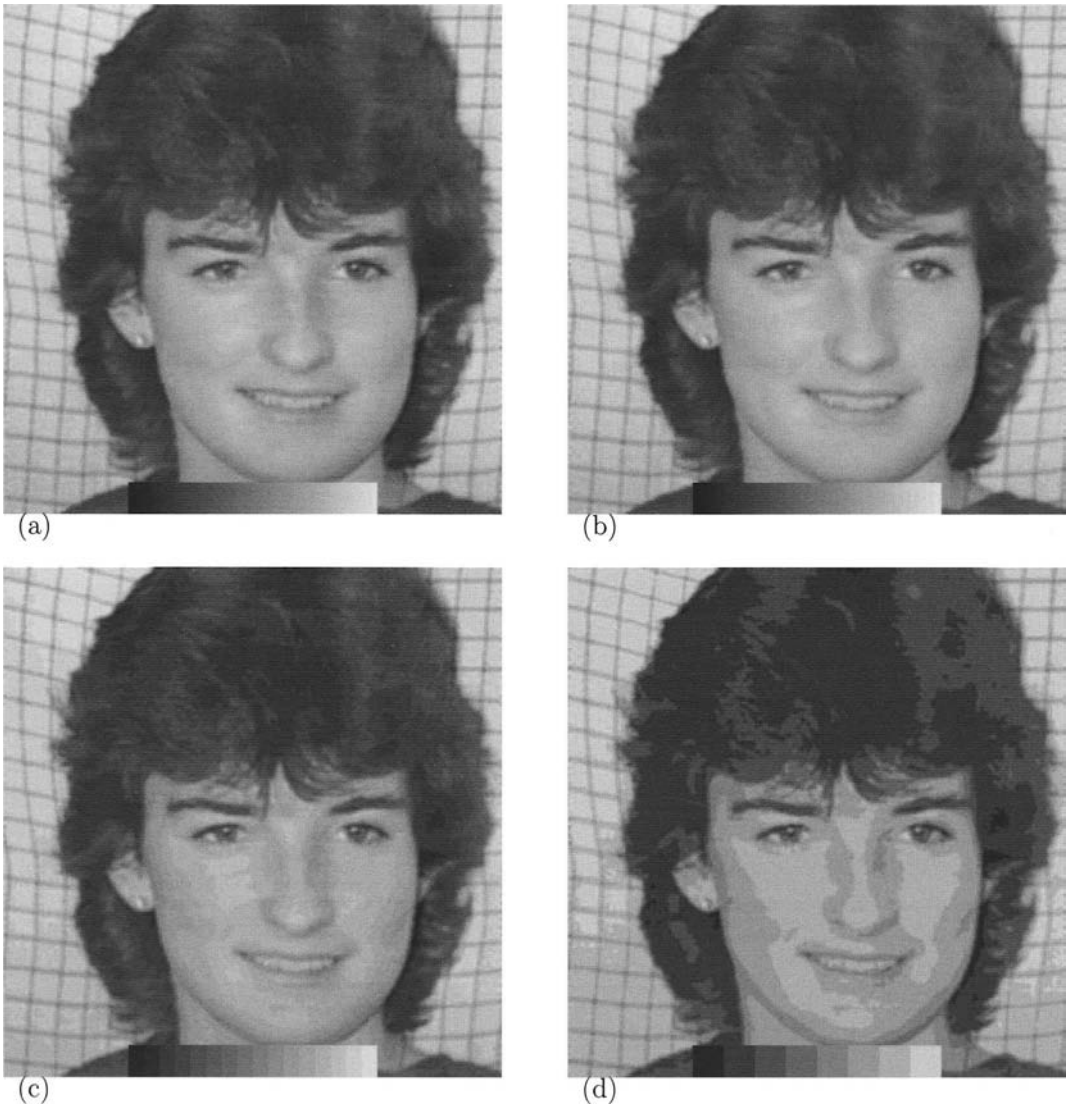


Bild 16.7: Quantisierung eines Testbildes mit (a) 64, (b) 32, (c) 16, (d) 8 Grauwerten. Für viele Anwendungsbeispiele sind weniger als 256 Grauwerte ausreichend.

und schließlich eine Klassifizierung mit einem neuronalen Netz. Es ist nun aber nicht notwendig und oft auch nicht sinnvoll, nach jedem Verarbeitungsschritt den Wertebereich des Ergebnisbildes wieder in die Grauwertmenge $G = \{0, ..., 255\}$ abzubilden. Vielmehr wird z.B. der nichtlineare Skalierungsoperator sein Ergebnisbild als *float-* (*real-*) Werte an den nächsten Operator weitergeben. Die Abbildung in die Grauwertmenge $G = \{0, ..., 255\}$ ist nur notwendig, wenn das Zwischenergebnis auf einem Displaysystem dargestellt werden soll, das diese Grauwertmenge verwendet.

Nun noch einige Bemerkungen zur Rasterung. Diese Thematik wurde in Kapitel 16.2 schon angesprochen. Es ist offensichtlich, dass die gewählte Rasterflächengröße die Qualität des digitalisierten Bildes wesentlich beeinflusst: Wird bei einer gegebenen Vorlage die Rasterfläche zu groß gewählt, so gehen feine Details des Originals verloren. Ist dagegen die Rasterfläche zu klein, so wird das Rechensystem mit zu vielen Daten belastet (Bild 16.8).

Für die Wahl der richtigen Rasterung bei der Digitalisierung werden sowohl pragmatische als auch theoretische Hilfestellungen angeboten. Zunächst kann die richtige Rasterung durch „Augenschein" festgelegt werden. Soll z.B. eine Strichzeichnung digitalisiert werden, deren minimale Linienbreite etwa $0.1mm$ ist, so könnte man eine Rasterung wählen, bei der man sicher ist, dass auf eine Breite von $0.1mm$ mindestens zwei Bildpunkte kommen. Man müsste also eine Rasterung wählen, bei der ein Quadratmillimeter in $20 \cdot 20 = 400$ oder mehr Bildpunkte zerlegt wird.

Die theoretische Grundlage hierzu bildet das Shannon'sche Abtasttheorem, das in Kapitel 16.2 schon erwähnt wurde. Eine ausführliche Darstellung dazu wird in [Bose04] oder [Grün02] gegeben.

In der Praxis liegen die Bilddaten bei vielen Anwendungsfällen bereits digitalisiert vor, so dass die Wahl der Rastergröße bereits vorweggenommen ist.

In Bild 16.9 wird ein anderer Effekt bei der Digitalisierung von Bilddaten gezeigt. Neben einer Digitalisierung des Testbildes mit $512 \cdot 512$ Bildpunkten, 256 Graustufen, ist eine Digitalisierung mit $32 \cdot 32$ Bildpunkten, 256 Graustufen abgebildet, bei der die Bildpunkte vergrößert wurden, um denselben Maßstab wie bei Bild 16.9-a zu erreichen. Bild 16.9-b ist in der Qualität deutlich schlechter als Bild 16.9-a. Durch die vergrößerte Wiedergabe der Bildpunkte als homogene graue Flächen werden Grauwertkanten in das Bild eingeführt, die im Original nicht vorhanden sind. Das menschliche Auge spricht auf diese Kanten sehr stark an, wodurch die eigentliche Bildstruktur verloren geht. Betrachtet man dagegen Bild 16.9-b aus der Ferne oder kneift man bei der Betrachtung die Augen etwas zusammen (oder nimmt man die Brille ab), so bemerkt man, dass doch mehr Information des Originals enthalten ist als zunächst vermutet wird. Bild 16.9-c, in dem über die Grenzen benachbarter Bildpunkte gemittelt wurde, zeigt einen ähnlichen Effekt.

Ein Sonderfall von Grauwertbildern sind *logische Bilder*. Sie haben gewöhnlich eine Bildmatrix der Form $\mathbf{S} = (s(x, y))$. Die Grauwerte g haben aber hier nicht die Bedeutung von digitalisierten Grautönen, sondern sind Codes für bestimmte „Klassen", die im Bild enthalten sind. Ein einfaches Beispiel ist ein Binärbild einer digitalisierten Strichzeichnung: Die weißen Bildpunkte mit dem Grauwert 1 werden als Hintergrund interpretiert und die schwarzen Bildpunkte mit dem Grauwert 0 als Bestandteile der Strichzeichnung. Ein anderes Beispiel ist ein digitalisierter Stadtplan, in dem Häuser rot, Straßen gelb, Grünflächen

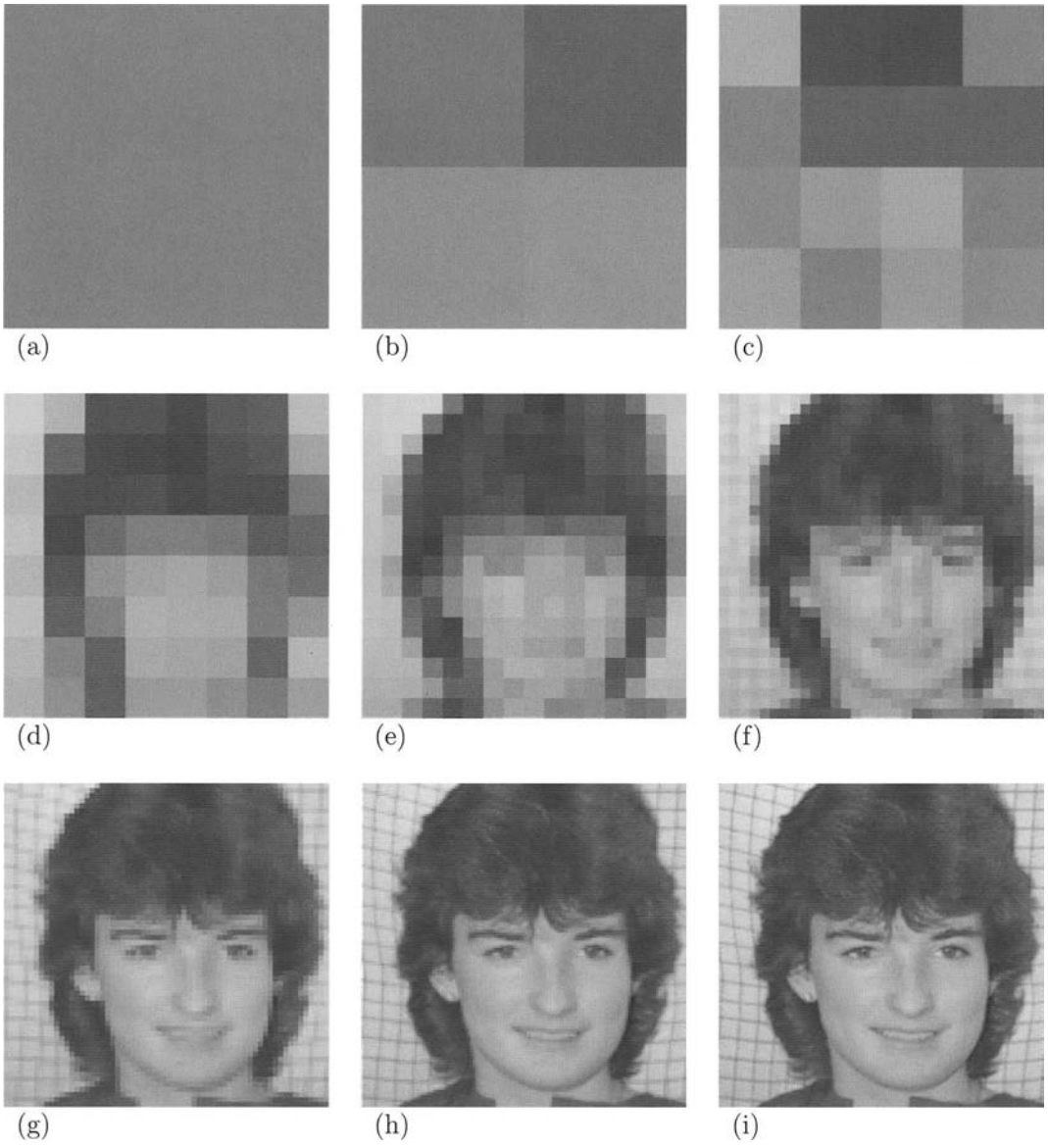
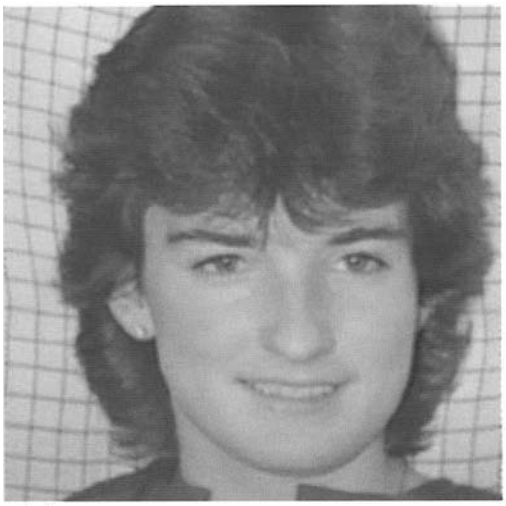


Bild 16.8: Rasterung einer Grautonvorlage mit einem quadratischen Gitter mit (a) $1 \cdot 1$, (b) $2 \cdot 2$, (c) $4 \cdot 4$, (d) $8 \cdot 8$, (e) $16 \cdot 16$, (f) $32 \cdot 32$, (g) $64 \cdot 64$, (h) $128 \cdot 128$ und (i) $256 \cdot 256$ Gitterpunkten. In der Darstellung sind alle Bildbeispiele durch Vergrößerung der Rasterflächen auf denselben Maßstab gebracht. Die neun Teilbilder sind auch ein Beispiel für die Darstellung eines Bildes als Baumstruktur (*quad tree*, Kapitel 30).



(a)

(b)

(c)

Bild 16.9: Quantisierung und Bildqualität. Die Qualität der Digitalisierung hängt nicht nur von der Wahl der Rasterflächengröße und der Quantisierung, sondern auch von der Art der Wiedergabe ab. (b) Wurde mit $32 \cdot 32$ Bildpunkten und 256 Graustufen digitalisiert. Die deutliche Verschlechterung gegenüber (a), $512 \cdot 512$ Bildpunkte, 256 Graustufen, ergibt sich auch aus der vergrößerten Darstellung der Bildpunkte als homogene graue Flächen. Bei (c) wurde über die Ränder der Bildpunkte hinweg gemittelt, und man erkennt, dass in (b) mehr Information des Originals vorhanden ist als zunächst vermutet.

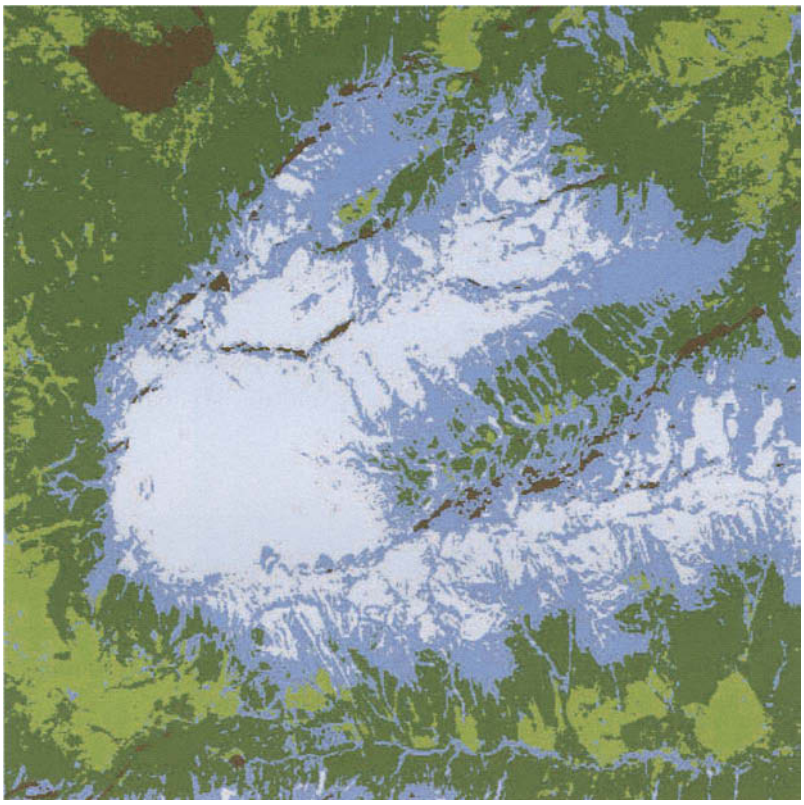


Bild 16.10: Beispiel eines klassifizierten Satellitenbildes. Als Original dienten SPOT-Bilddaten. Es zeigt das Wettersteingebirge mit der Zugspitze. Das Bild wurde mit einem neuronalen Netz klassifiziert. Die Farben bedeuten: weiß - Eis/Schnee, grau - Gestein, dunkelbraun - Gewässer, dunkelgrün - Wald und hellgrün - Grünland.

grün, Eisenbahnlinien schwarz, usw. dargestellt sind. Jeder „Klasse" wird im digitalisierten Bild ein Grauwert als *Klassencode* zugeordnet. Bei der bildlichen Reproduktion kann dann der Klassencode wieder mit dem zugehörigen Farbton ausgegeben werden. Logische Bilder sind oft Zwischenprodukte auf dem Weg vom Original zu einem interpretierten Bild, z.B. das Ergebnis einer Bildsegmentierung (Bild 16.10 und die Kapitel 23 und 31).

16.5 Farbbilder

In diesem wird Kapitel die Verarbeitung von Farbbildern erläutert. Bevor das mathematische Modell für Farbbilder formuliert wird, werden unterschiedliche Aspekte des Phänomens „Farbe" betrachtet:

- Die physikalischen Aspekte,
- die physiologischen Aspekte,
- die normativen, technischen Aspekte und
- die darauf aufbauenden Farbmodelle.

16.5.1 Farbe: Physikalische Aspekte

Physikalisch betrachtet wird Farbe durch Licht erzeugt. Licht ist eine elektromagnetische Welle und kann somit durch die Frequenz f und die Wellenlänge λ beschrieben werden. Über die Lichtgeschwindigkeit c sind die Frequenz und die Wellenlänge miteinander verbunden:

$$c = f \cdot \lambda. \tag{16.5}$$

Für den Menschen ist nur ein sehr schmaler Ausschnitt aus dem elektromagnetsichen Spektrum sichtbar: Er liegt bei Licht der Wellenlänge 400 *nm* (Nanometer) bis 700 *nm*. Trifft Licht mit einer bestimmten Wellenlänge (*monochromatisches Licht*) auf das Auge, so wird eine Farbempfindung hervorgerufen, die durch die Farben des Regenbogenspektrums, von Violett (400 *nm*) bis Rot (700 *nm*), beschrieben werden kann.

16.5.2 Farbe: Physiologische Aspekte

Damit wird aber bereits der zweite, der physiologische Aspekt, diskutiert. Ausschlaggebend für das Farbempfinden sind *Fotopigmente* im Auge, die auf unterschiedliche Ausschnitte des sichtbaren Spektrums reagieren. Sie sprechen vor allem auf Rot, Grün und Blau an, reagieren aber auch auf andere Wellenlängen (Bild 16.11). Diese Grafik zeigt, dass das Auge am intensivsten auf Grün und weniger stark auf Rot und Blau reagiert.

Aus diesem Grund entspricht die Lichtstärke (die wahrgenommene Helligkeit) nicht direkt seiner Energie: Um die Helligkeit von grünem Licht zu erhöhen, wird weniger Energie benötigt als für die Erhöhung der Helligkeit von rotem oder blauem Licht. Bild 16.12 zeigt die relative Empfindlichkeit des Auges für die Wellenlängen des sichtbaren Spektrums.

Monochromatisches Licht erzeugt die Empfindung „reiner" Farben. Dasselbe oder ähnliches Farbempfinden ist aber auch durch die Mischung verschiedener Wellenlängen zu erzeugen. Man kann jedoch monochromatisches Licht so mischen, dass Farbempfindungen hervorgerufen werden, die durch monochromatisches Licht nicht erzeugt werden können. Eine solche Mischfarbe kann in weißes Licht und eine reine Farbe oder ihr Komplement aufgeteilt werden. Die Empfindung „weißes Licht" entsteht, wenn alle „Wellenlängen in derselben Menge" vorkommen. Die reine Farbe, z.B. Rot, bestimmt die *Färbung* (*hue*), der Anteil an weißem Licht bestimmt die *Tönung*, z.B. Rosa. Mit zunehmendem Anteil an weißem Licht sinkt die *Sättigung* (*saturation*) der reinen Farbe. Die Gesamtmenge des Lichtes bestimmt die *Intensität* (*Luminanz*, *intensity*).

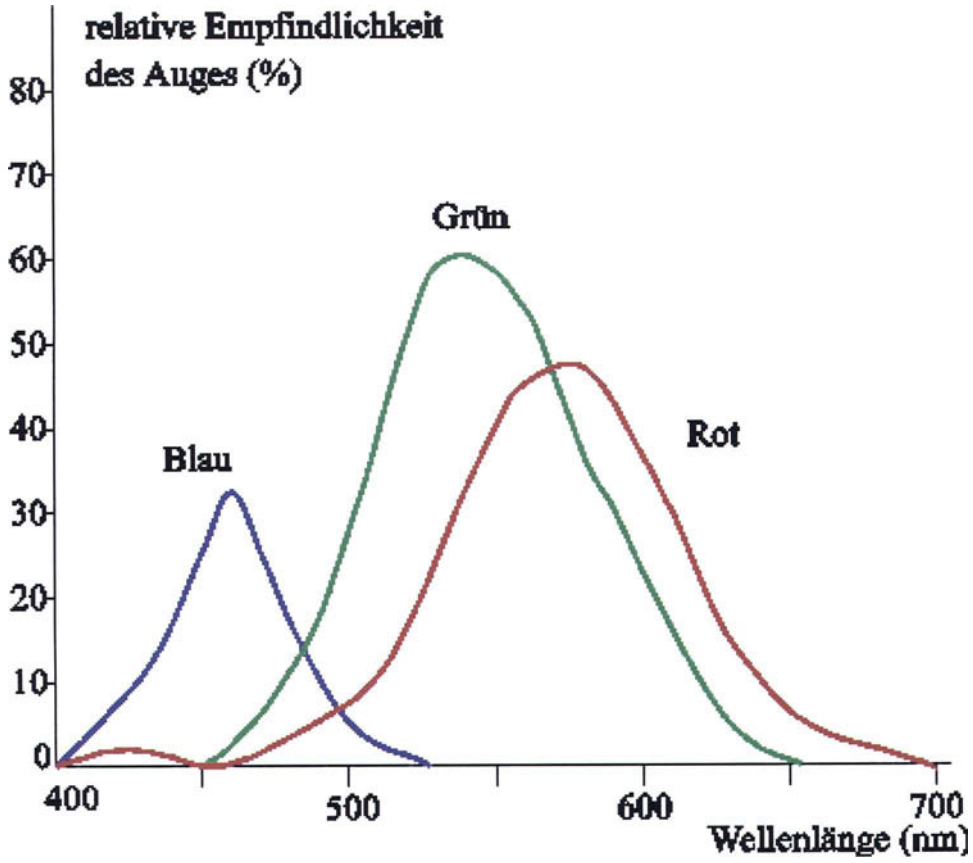


Bild 16.11: Relative Reaktion der Fotopigmente des menschlichen Auges.

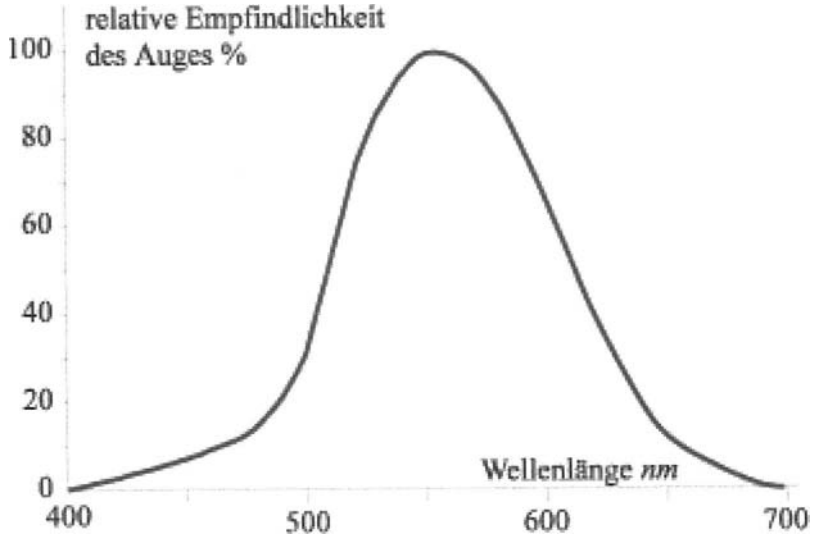


Bild 16.12: Relative Empfindlichkeit des Auges für die Wellenlängen des sichtbaren Spektrums.

16.5.3 Das CIE-Farbdreieck

Nun zum normativen, technischen Aspekt. Auf der Basis der physiologischen Gegebenheiten, dass die Farbreaktion im Auge durch die Mischung der Signale der rot-, grün- und blauempfindlichen Fotopigmente hervorgerufen wird, geht man davon aus, dass sich Farben durch drei Grundfarben zusammensetzen lassen. Allerdings gibt es keine drei Grundfarben, aus denen sich alle Farben erzeugen lassen. Um hier eine Normierung und Vergleichbarkeit zu erreichen, hat die Commission Internationale de l'Eclaire (CIE) 1931 drei künstliche Grundfarben, die *Normfarbwerte* oder *Primärfarben* eingeführt. Sie werden mit X, Y und Z bezeichnet und sind durch Energieverteilungskurven charakterisiert. So wurde z.B. für Y die Lichtstärkenreaktion des menschlichen Auges (Bild 16.12) verwendet. Außerdem wurde von der CIE die *Normfarbtafel* entwickelt (Bild 16.13), die so aufgebaut ist, dass

- jeder Punkt des Diagramms eine Farbe repräsentiert,
- alle Farben auf der Strecke zwischen zwei Farbpunkten durch Mischen der Farben der Endpunkte hergestellt werden können und
- alle Punkte innerhalb eines Dreiecks durch Mischen der Farben der Eckpunkte zu erhalten sind.

Entlang des zungenförmigen Randes liegen die Farben des monochromatischen Lichtes, und im unteren Bereich, nahe der x-Achse, die *Purpurfarben*, die nicht monochromatisch

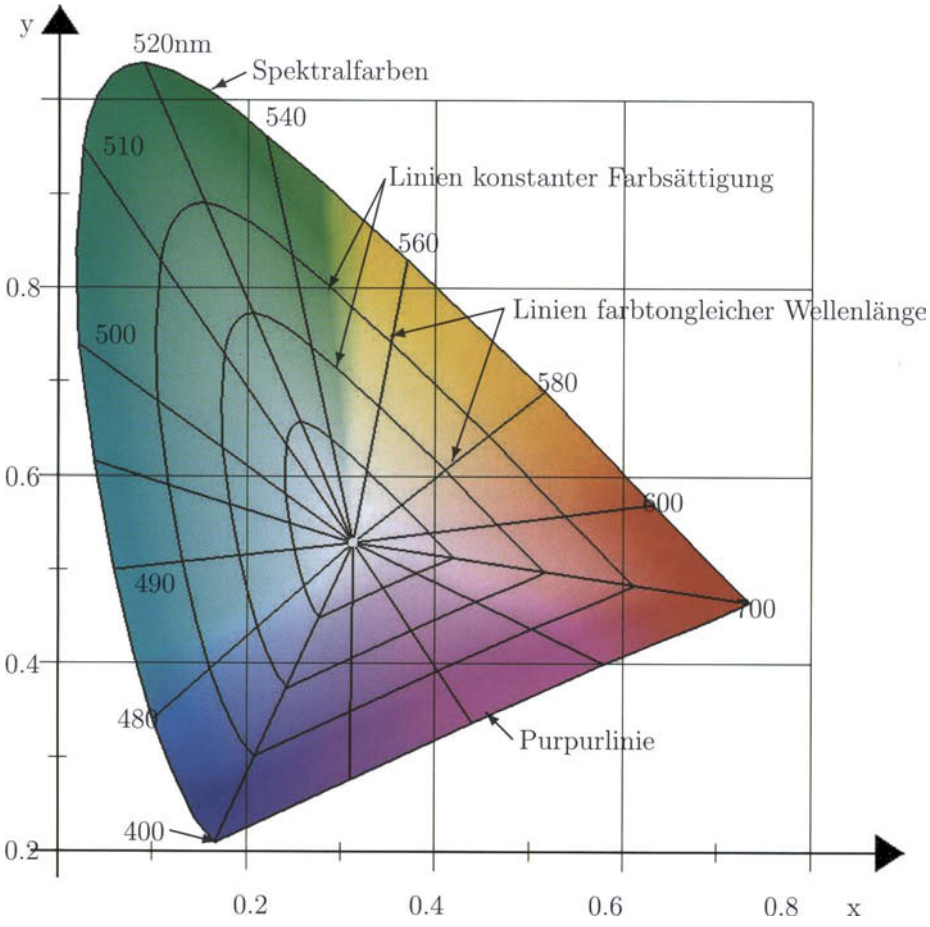


Bild 16.13: Normfarbtafel der Commission Internationale de l'Eclaire (CIE), 1931.

dargestellt werden können. In der Mitte ist ein Bereich, in dem die Farben für das menschliche Auge weiß erscheinen.

Wenn X, Y und Z als Anteile der Primärfarben einer beliebigen Farbe gemessen werden, so kann die Position dieser Farbe im CIE-Farbdreieck leicht ermittelt werden:

$$x = \frac{X}{X+Y+Z}, \qquad y = \frac{Y}{X+Y+Z}, \qquad z = \frac{Z}{X+Y+Z}. \tag{16.6}$$

Im CIE-Diagramm werden die (x, y)-Werte gezeigt, auf den z-Wert wird verzichtet, da er wegen der Beziehung

$$z = 1 - x - y \tag{16.7}$$

leicht berechnet werden kann und daher ein zweidimensionales Diagramm ausreicht.

Wenn nun eine Farbe durch die (x, y)-Koordinaten im CIE-Dreieck ausgewählt wird, so muss, um die Anteile an den Primärfarben berechnen zu können, noch die Helligkeit (Intensität , Luminanz) vorgegeben werden. Dies ist aber gerade der Y-Wert. Die X- und Z-Werte berechnen sich dann gemäß:

$$X = x \cdot \frac{Y}{y} \quad \text{und} \quad Z = z \cdot \frac{Y}{y}. \tag{16.8}$$

Wenn man im CIE-Farbdreieck drei Punkte als Ecken eines Dreiecks festlegt, so sind alle Farben innerhalb des Dreiecks durch Mischen der Eckfarben zu erhalten. Man sagt, dass durch die drei Eckpunkte eine *Farbskala* definiert wurde. Je nach Festlegung der drei Eckpunkte erhält man eine andere Farbskala. Die CIE hat für Rot, Grün und Blau einen Standard eingeführt, den die folgende Tabelle enthält:

CIE-Farbe	Wellenlänge (nm)	x	y	z
Spektral-Rot	700.0	0.73467	0.26533	0.0
Spektral-Grün	546.1	0.27367	0.71741	0.00892
Spektral-Blau	435.8	0.16658	0.00886	0.82456

Auf Farbmonitoren werden die Farben durch rot, grün und blau leuchtende Phosphore erzeugt, denen Punkte im CIE-Dreieck entsprechen. Für moderne Farbmonitore werden folgende Werte für die Grundfarben angegeben:

Grundfarbe	x	y	z
Monitor-Rot	0.628	0.346	0.026
Monitor-Grün	0.268	0.588	0.144
Monitor-Blau	0.150	0.070	0.780

In den folgenden Abschnitten werden einige Farbmodelle vorgestellt, die auf diesen Grundlagen aufbauen.

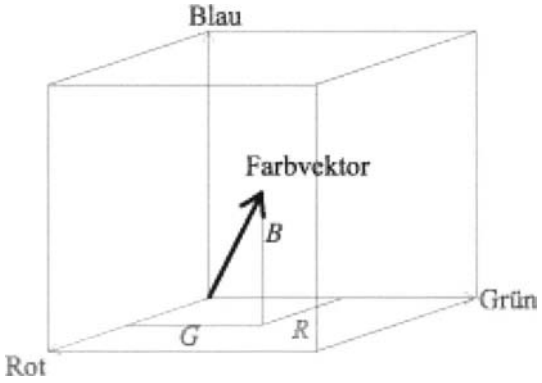


Bild 16.14: Der RGB-Farbraum. Jeder Punkt in diesem kartesischen Koordinatensystem entspricht einer Farbe mit einer bestimmten Helligkeit.

16.5.4 Das RGB-Farbmodell

Wenn die drei Punkte für Rot, Grün und Blau im CIE-Farbdreieck festgelegt sind, kann man alle Farben in dem durch die Eckpunkte festgelegten Dreieck mischen. Eine bestimmte Farbe ist somit eine Linearkombination der drei Grundfarben. Man stellt deshalb den RGB-Farbraum als dreidimensionales, kartesisches Koordinatensystem dar (Bild 16.14). Die Größen werden so normiert, dass die $R-$, $G-$ und $B-$Komponenten zwischen 0 und 1 liegen.

Ein Problem ist noch zu lösen: Durch die Wahl der drei Grundfarben im CIE-Farbdreieck wird eine bestimmte Farbskala definiert. Es soll aber die Eigenschaft erhalten bleiben, dass die Mischung der drei Grundfarben mit maximaler Intensität die Farbe Weiß ergibt. Dazu muss man im CIE-Dreieck einen Punkt als Weiß definieren. Häufig wird dazu das Weiß eines auf 6500 *Grad Kelvin* erhitzten schwarzen Körpers verwendet:

$$x = 0.313, \quad y = 0.329 \quad \text{und} \quad z = 0.358. \tag{16.9}$$

Bei einer Luminanz von $Y_w = 1$ ergeben sich die Anteile der CIE-Normfarben an diesem Weiß mit Formel (16.8):

$$X_w = 0.951, \quad Y_w = 1.000 \quad \text{und} \quad Z_w = 1.088. \tag{16.10}$$

Den Grundfarben (z.B. eines Monitors) entsprechen die CIE-Normfarbanteile (X_r, Y_r, Z_r), (X_g, Y_g, Z_g) und (X_b, Y_b, Z_b). Diese zu gleichen Anteilen gemischt sollen den Weißpunkt ergeben:

$$\begin{aligned} X_r + X_g + X_b &= X_w, \\ Y_r + Y_g + Y_b &= Y_w, \\ Z_r + Z_g + Z_b &= Z_w. \end{aligned} \tag{16.11}$$

Mit (16.8) erhält man:

$$\begin{aligned} x_r \cdot \frac{Y_r}{y_r} + x_g \cdot \frac{Y_g}{y_g} + x_b \cdot \frac{Y_b}{y_b} &= X_w, \\ y_r \cdot \frac{Y_r}{y_r} + y_g \cdot \frac{Y_g}{y_g} + y_b \cdot \frac{Y_b}{y_b} &= Y_w, \\ z_r \cdot \frac{Y_r}{y_r} + z_g \cdot \frac{Y_g}{y_g} + z_b \cdot \frac{Y_b}{y_b} &= Z_w; \end{aligned} \tag{16.12}$$

Das bedeutet, dass man die Intensitäten Y_r, Y_g und Y_b der drei (Monitor-)Grundfarben so einstellen muss, dass sich das festgelegte Normweiß ergibt. Fasst man die Quotienten zu den Linearfaktoren r, g und b zusammen und setzt man die (x, y, z)-Koordinaten der (Monitor-) Grundfarben ein, so erhält man folgendes lineares Gleichungssystem:

$$\begin{aligned} 0.628r + 0.268g + 0.150b &= 0.951, \\ 0.346r + 0.588g + 0.070b &= 1.000, \\ 0.026r + 0.144g + 0.780b &= 1.088. \end{aligned} \tag{16.13}$$

Als Lösung ergibt sich für r, g und b:

$$r = 0.761, \quad g = 1.114 \quad \text{und} \quad b = 1.164. \tag{16.14}$$

Wenn eine Farbe im RGB-Farbraum durch die Koordinaten (R, G, B) gegeben ist, so kann man z.B. den Anteil X des Normrot an dieser Farbe wie folgt berechnen:

$$\begin{aligned} X &= X_r R + X_g G + X_b B = \\ &= x_r r R + x_g g G + x_b b B = \\ &= 0.628 \cdot 0.761 \cdot R + 0.268 \cdot 1.114 \cdot G + 0.150 \cdot 1.164 \cdot B = \\ &= 0.478 \cdot R + 0.299 \cdot G + 0.175 \cdot B. \end{aligned} \tag{16.15}$$

Analog kann man den Y- und den Z-Anteil berechnen. In Matrizenschreibweise erhält man dann die beiden Gleichungen für die Umrechnung $(R, G, B) \rightarrow (X, Y, Z)$ und invers dazu $(X, Y, Z) \rightarrow (R, G, B)$:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.478 & 0.299 & 0.175 \\ 0.263 & 0.655 & 0.081 \\ 0.020 & 0.160 & 0.908 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}, \tag{16.16}$$

und

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 2.741 & -1.147 & -0.426 \\ -1.118 & 2.028 & 0.034 \\ 0.137 & -0.332 & 1.105 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}. \tag{16.17}$$

An dieser Stelle sei darauf hingewiesen, dass diese Umrechnungsformeln nur gelten, wenn die Farbskala der Monitorgrundfarben gemäß obiger Tabelle vorausgesetzt werden und wenn der angegebene Weißpunkt verwendet wird. Sollte man diese Umrechnung für ein anderes Farbdreieck benötigen, so lässt sie sich anhand dieses Beispiels leicht nachvollziehen.

Das RGB-Farbssystem bezeichnet man als *additives Farbmodell*, da sich z.B. bei einem Farbmonitor die Farbempfindung im Auge durch die Überlagerung des von den einzelnen Phosphorpunkten ausgestrahlten Lichts ergibt.

16.5.5 Das CMY-Farbmodell

Im Druckereiwesen wird ein *subtraktives Farbmodell* verwendet. Es verwendet die Grundfarben Zyan (*cyan*), Magenta (*magenta*) und Gelb (*yellow*) und wird deshalb in der Literatur als *CMY-Farbmodell* bezeichnet. Der Farbeindruck im Auge entsteht dadurch, dass die Grundfarben mit unterschiedlicher Intensität auf das Papier aufgebracht werden und diese aus weißem Licht Farben absorbieren. Die gesehene Farbe ist somit die Überlagerung des nicht absorbierten, also reflektierten Lichts. Da Zyan Rot absorbiert, Magenta Grün und Gelb Blau, sagt man auch Rot, Grün und Blau sind die *komplementären Farben* von Zyan, Magenta und Gelb. Das CMY-Farbmodell definiert wie das RGB-Farbmodell einen dreidimensionalen kartesischen Raum. Zusammen mit dem RGB-Raum kann man sich einen Farbwürfel vorstellen, dessen gegenüberliegende Ecken die Komplementärfarben sind (Bild 16.15).

Beim RGB-System liegt im Koordinatenursprung die Farbe Schwarz, beim CMY-System die Farbe Weiß. Die Umrechnung zwischen RGB- und CMY-System ist einfach:

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix}, \tag{16.18}$$

und

$$\begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \tag{16.19}$$

Beim *Vierfarbendruck* wird zusätzlich noch als vierte Grundfarbe Schwarz verwendet, um satte Schwarztöne nicht durch Überdrucken von Zyan, Magenta und Gelb erzeugen zu müssen.

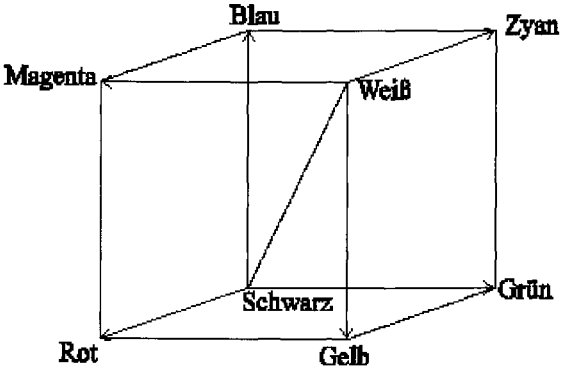


Bild 16.15: Der RGB-CMY-Farbwürfel.

16.5.6 Das YIQ-Farbmodell

Das *YIQ-Farbmodell* wird beim Farbfernsehen verwendet. Das Problem ist hier die Farbinformation so zu übertragen, dass auch Schwarz-/Weiß-Fernsehgeräte vernünftige Bilder erzeugen können. Dazu verwendet man als Y dasselbe Y wie bei den CIE-Normfarben, das ja die Intensität des Lichts wiedergibt. Schwarz-/Weiß-Fernsehgeräte verwenden nur dieses Y. Die Komponenten I und Q bestimmen die Färbung und die Sättigung. Das YIQ-Modell wird in dem in den USA eingeführten *NTSC-System* verwendet. Das europäische *PAL-System* verwendet die Komponenten Y, $R-Y$ und $B-Y$.

16.5.7 Das HSI-Farbmodell

Das *HSI-Farbmodell* wird in der Farbfotografie und in der Videotechnik verwendet. Hier steht H für *hue* (*Färbung* , *Tönung*), S für *saturation* (*Sättigung* , *Chroma*) und I für *Intensity* (*Intensität* , *Helligkeit*).

Die Sättigung gibt an, wieviel weißes Licht der Tönung beigemischt wird. Bei Sättigung 0 sieht man keine Farbe, somit je nach Intensität Schwarz, Weiß oder Grautöne. Die Intensität gibt an, wieviel Gesamtlicht vorhanden ist.

Anstatt eines kartesischen Koordinatensystems verwendet man beim HSI-Farbmodell eine Darstellung mit Zylinderkoordinaten. Die Umrechnung von (R, G, B) nach (H, S, I) erfolgt in zwei Schritten: Zuerst wird das RGB-Koordinatensystem so gedreht, dass eine Achse mit der Raumdiagonalen des Farbwürfels zusammenfällt. Diese Achse wird mit I_1 bezeichnet, die beiden anderen mit M_1 und M_2. Diese Drehung des Koordinatensystems

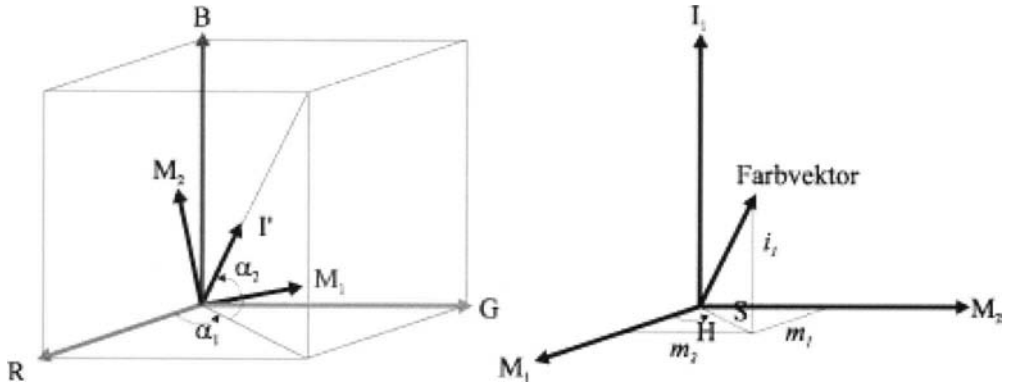


Bild 16.16: Zusammenhänge zwischen dem RGB- und dem HSI-Farbmodell.

wird mit folgender Gleichung beschrieben:

$$\begin{pmatrix} m_1 \\ m_2 \\ i_1 \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{6}} & -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{6}} \\ 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}. \tag{16.20}$$

Im zweiten Schritt werden die kartesischen (m_1, m_2, i_1)-Koordinaten in Zylinderkoordinaten umgerechnet:

$$\begin{aligned} H &= \arctan\left(\frac{m_1}{m_2}\right), \\ S &= \sqrt{m_1^2 + m_2^2}, \\ I &= \sqrt{3} \cdot i_1. \end{aligned} \tag{16.21}$$

Bild 16.16 zeigt die Zusammenhänge zwischen dem RGB- und dem HSI-Farbmodell: Einem Farbvektor (R, G, B) im RGB-Koordinatensystem entspricht ein Vektor mit den Komponenten (m_1, m_2, i_1) im gedrehten Koordinatensystem. Die H-Komponenten (also die Färbung) ist der Winkel des Farbvektors mit der M_1-Achse und die S-Komponente (die Sättigung) der Betrag des Vektors.

Die Umrechnung des HSI-Systems in das RGB-System erfolgt sinngemäß umgekehrt:

$$m_1 = S \sin H,$$

$$m_2 = S \cos H, \tag{16.22}$$

$$i_1 = \frac{I}{\sqrt{3}}$$

und

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{6}} & 0 & \frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{6}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \\ -\frac{1}{\sqrt{6}} & -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ i_1 \end{pmatrix}. \tag{16.23}$$

Die Verwendung des HSI-Farbmodells kann z.B. bei Segmentierungsoperationen mit dem Merkmal „Farbe" Vorteile bringen, da im RGB-Modell für eine bestimmte Farbe immer alle drei Komponenten benötigt werden, während im HSI-Modell die Färbung nur durch die H-Komponente bestimmt wird.

16.5.8 Mathematisches Modell für Farbbilder

Im Rahmen dieser Darstellung wird das RGB-Farbmodell verwendet. Bei der Digitalisierung eines Farbbildes werden mit Farbfiltern Rot-, Grün- und Blauauszüge angefertigt. Jeder der drei Abtastvorgänge ist, abgesehen von der Verwendung eines Farbfilters, identisch mit der Digitalisierung bei Grautonbildern (Kapitel 16.4). Ein Bildpunkt eines digitalisierten Farbbildes besteht aus drei Maßzahlen für rot, grün und blau (Bild 16.17).

Das mathematische Modell von Kapitel 16.4 muss für Farbbilder erweitert werden, da die Bildmatrix jetzt dreidimensional ist:

$G = \{0, 1, 2, ..., 255\}$	Grauwertmenge mit 256 Grauwerten
$\mathbf{S} = (s(x, y, n))$	dreidimensionale Bildmatrix des digitalisierten Farbbildes
$x = 0, 1, ..., L-1$	L Bildzeilen
$y = 0, 1, ..., R-1$	R Bildspalten
$n = 0, 1, 2$	$N = 3$ Kanäle
$\mathbf{s}(x, y) = (g_0, g_1, g_2)^T$	Bildpunkt in der Position (x, y).
$g_n \in G, n = 0, 1, 2$	Maßzahl für rot, grün und blau.

(16.24)

Die einzelnen Farbkomponenten werden in der Regel mit einem Byte codiert. Ein Bildpunkt eines digitalisierten Farbbildes umfasst also drei Byte oder 24 Bit. Bei praktischen Anwendungen treten aber wesentlich weniger als die $2^{24} = 16777216$ theoretisch möglichen Farbtöne auf. Moderne Grafikkarten verwenden allerdings heute meistens 36 Bit.

Die Werte der Farbkomponenten liegen hier nicht, wie in Kapitel 16.5.4 dargestellt, im Intervall [0,1], sondern in der Grauwertmenge $G = \{0, 1, 2, ..., 255\}$. Bei Umrechnung in andere Farbmodelle ist das unter Umständen zu beachten. Wegen der diskretisierten Werte können sich bei der Umrechnung Fehler ergeben.

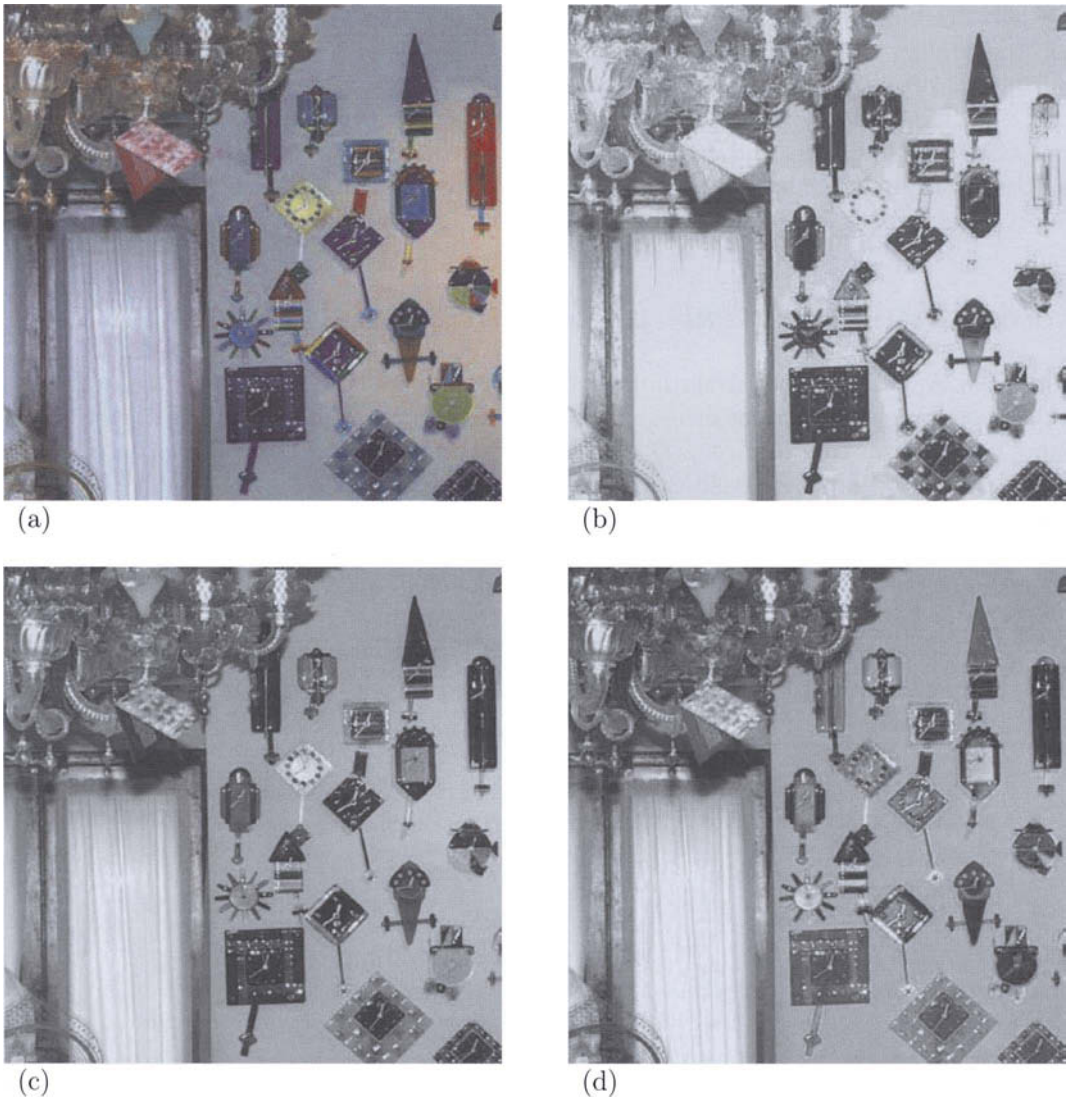


Bild 16.17: Beispiel eines Farbbildes (a), Rotauszug (b), Grünauszug (c) und Blauauszug (d).

Die Farbe ist ein wichtiges Merkmal für die Segmentierung von Bildern. Die Farbe als Merkmal für eine Bildsegmentierung wird in Kapitel 24 ausführlich besprochen. Dort werden auch einige Verfahren zur Weiterverarbeitung von Farbbildern, etwa die Erstellung von Indexbildern, erläutert. Die Verwendung des HSI-Modells bringt bei manchen Verarbeitungsschritten Zeitvorteile, da geringere Datenmengen verarbeitet werden müssen. Soll z.B. eine Bildsegmentierung mit der Farbe als Merkmal durchgeführt werden, so müssen im RGB-Modell alle drei Farbauszüge berücksichtigt werden. In der HSI-Darstellung kann die Segmentierung möglicherweise allein anhand des Farbwertes durchgeführt werden. In den Kapiteln 31 bis 32 werden häufig Beispiele mit dem Merkmal Farbe verwendet.

16.6 Multispektral- und mehrkanalige Bilder

Bei vielen Anwendungen werden multisensorielle Systeme eingesetzt, die u.a. auch Wellenlängen erfassen, die für das menschliche Auge nicht mehr sichtbar sind (z.B. ultraviolett, nahes oder thermisches infrarot). Digitalisierte Bilder dieser Art bezeichnet man als *Multispektralbilder*. Bild 16.18 stellt einen Ausschnitt aus einem Multispektralbild des Fernerkundungssatellitensystems SPOT dar, das unter anderem mit einem Multispektralscanner die Spektralbereiche $500nm - 590nm$ (grün), $610nm - 680nm$ (rot) und $790nm - 890nm$ (nahes infrarot) mit einer räumlichen Auflösung von $20m$ aufzeichnet.

Ein Multispektralbild kann wie das digitalisierte Farbbild als dreidimensionale Bildmatrix $\mathbf{S} = (s(x, y, n))$ dargestellt werden, nur dass für den Index n der Definitionsbereich erweitert wird: $n = 0, 1, ..., N - 1$. Ein digitalisiertes Farbbild ist ein Spezialfall eines Multispektralbildes:

$G = \{0, 1, 2, ..., 255\}$	Grauwertmenge mit 256 Grauwerten	
$\mathbf{S} = (s(x, y, n))$	dreidimensionale Bildmatrix	
$x = 0, 1, ..., L - 1$	L Bildzeilen	
$y = 0, 1, ..., R - 1$	R Bildspalten	(16.25)
$n = 0, 1, ..., N - 1$	N Kanäle	
$\mathbf{s}(x, y) = (g_0, g_1, ..., g_{N-1})^T$	Bildpunkt in der Position (x, y).	
$g_n \in G, n = 0, 1, ..., N - 1$	Maßzahl im Kanal n.	

Oft haben die einzelnen Kanäle eines Bildes nicht mehr die Bedeutung von Spektralinformation. Das ist z.B. dann der Fall, wenn die Spektralkanäle mit Operatoren zur Kantenextraktion behandelt wurden oder wenn ein Bild aus mehreren Spektralkanälen und zusätzlich aus einem Kanal mit dem digitalen Höhenmodell des Landschaftsausschnittes zusammengesetzt ist. Bilder dieser Art werden als *mehrkanalige Bilder* bezeichnet. Das mathematische Modell muss dazu nicht erweitert werden. Eine grafische Darstellung eines 4-kanaligen Bildes zeigt Bild 16.19. Ein Multispektralbild ist ein Spezialfall eines mehrkanaligen Bildes.

An dieser Stelle seien noch die Begriffe Pseudofarbdarstellung, Falschfarbendarstellung und Echtfarbendarstellung erläutert.

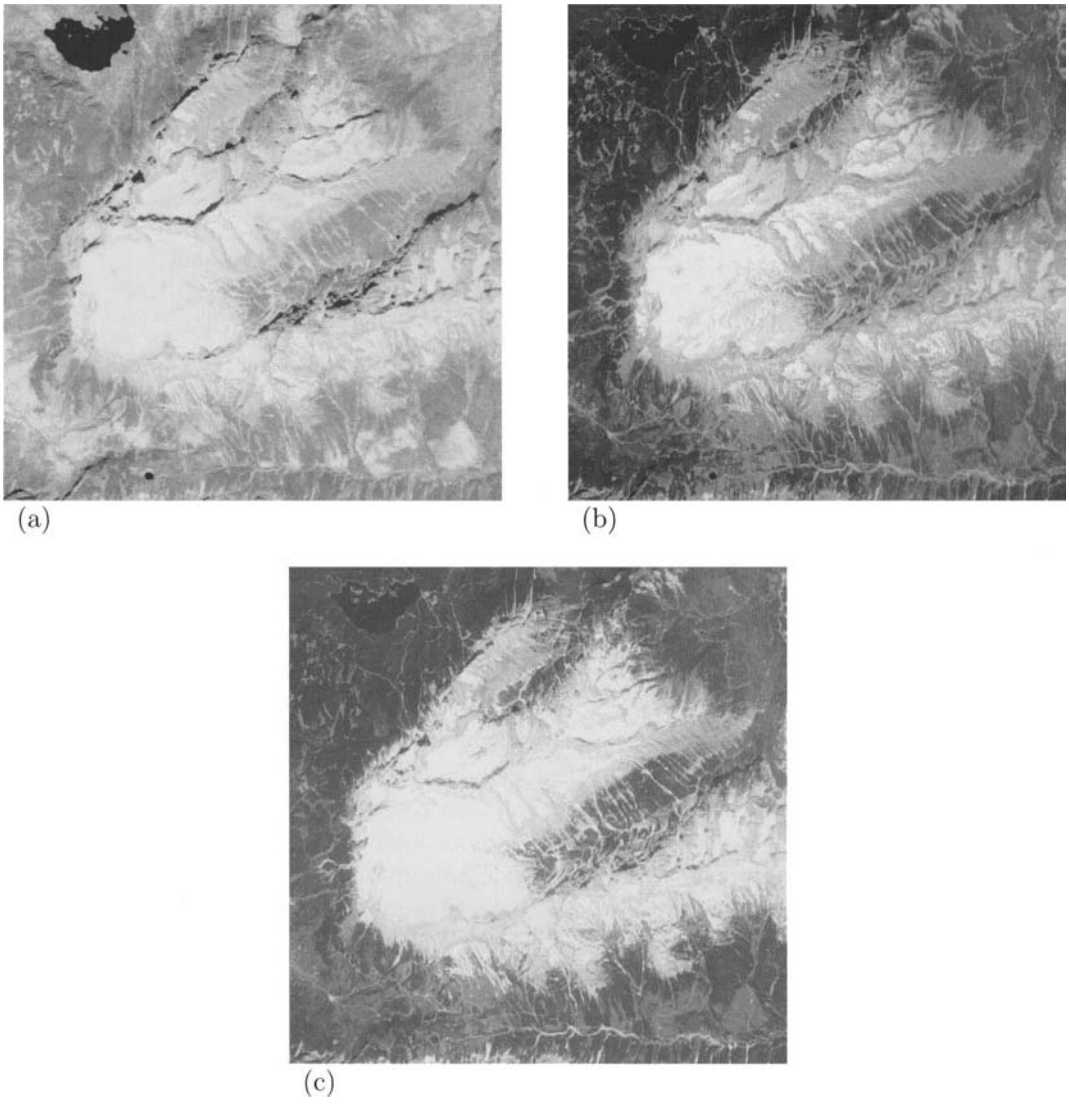


Bild 16.18: Beispiel eines Multispektralbildes des Fernerkundungssatellitems SPOT. Der Bildausschnitt zeigt das Wettersteingebirge mit der Zugspitze. Es sind die drei Spektralbereiche (a) grün ($500nm - 590nm$), (b) rot ($610nm - 680nm$) und (c) nahes infrarot ($590nm - 890nm$) abgebildet. Die Bodenauflösung beträgt $20m$.

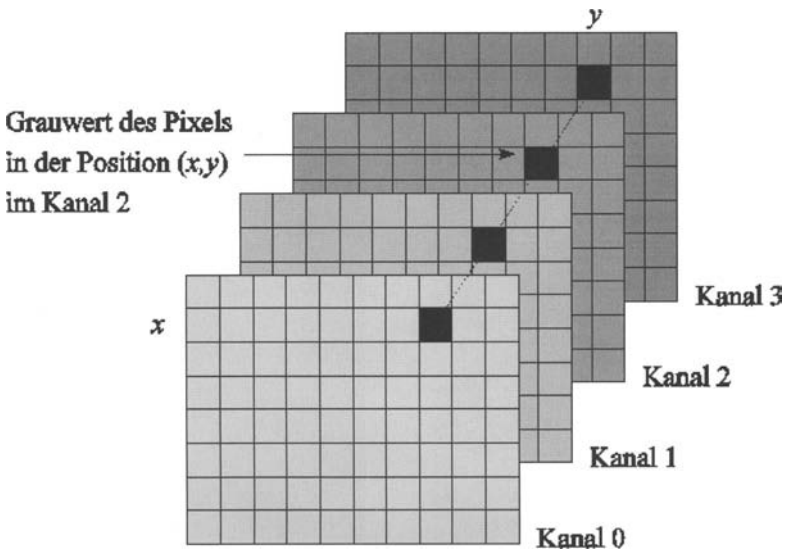


Bild 16.19: Beispiel eines 4-kanaligen Bildes mit einer dreidimensionalen Bildmatrix. Ein Bildpunkt ist hier ein 4-dimensionaler Vektor.

Bei der *Pseudofarbdarstellung* ist das Ausgangsbild ein einkanaliges Grauwertbild $\mathbf{S} = (s(x,y))$ mit $s(x,y) \in G$. Zu diesem Bild liegt eine *look-up-Tabelle* (*LuT*), oft auch *Palette*) vor. Die *look-up*-Tabelle ist eine Tabelle mit 256 Einträgen mit den relativen Adressen 0 bis 255. Der Grauwert $s(x,y)$ wird als Index interpretiert (daher auch der häufig verwendete Name *Indexbild*), der auf einen der Einträge der *look-up*-Tabelle zeigt. Zur Darstellung auf einem Displaysystem oder bei der Ausgabe auf einem Drucker wird der in der *look-up*-Tabelle eingetragene Wert $LuT[s(x,y)]$ verwendet. Dies kann ein 8-Bit-Grauwert g oder ein 24-Bit-RGB-Farbtripel (r,g,b) sein. Wenn z.B. die *look-up*-Tabelle mit den Werten $LuT[i] = i, \quad i = 0,...,255$ geladen ist, wird das Bild so dargestellt, wie es in der Bildmatrix $\mathbf{S}$ gespeichert ist. Das invertierte Bild (Negativbild) erhält man mit $LuT[i] = 255 - i, \quad i = 0,...,255$. Durch passende Wahl der *look-up*-Tabelle lassen sich, in Verbindung mit einem geeigneten Displaysystem, meistens in Videoechtzeit, Helligkeits- und Kontrastveränderungen durchführen. Ist der *LuT*-Eintrag ein 24-Bit-RGB-Farbtripel, so wird auf dem Displaysystem die Farbe ausgegeben, die diesem Farbtripel entspricht, vorausgesetzt, dass das Displaysystem 24-Bit-Farbtripel darstellen kann. Wie aus RGB-Farbbildern mit 24 Bit pro Bildpunkt Indexbilder mit nur 8 Bit pro Bildpunkt erstellt werden können, wird in Kapitel 24 ausführlich beschrieben.

Der Begriff *Falschfarbendarstellung* wird fast nur in der Fernerkundung verwendet. Damit ist gemeint, dass den RGB-Kanälen eines Displaysystems nicht die RGB-Kanäle eines Farb- oder Multispektralbildes, sondern andere Kanalkombinationen zugeordnet werden.

Man könnte z.B. dem Rotkanal des Displaysystems den Grünkanal eines Multispektralbildes **S**, dem Grünkanal des Displaysystems den Blaukanal von **S** und dem Blaukanal des Displaysystems einen Infrarotkanal von **S** zuordnen.

Von einer *Echtfarbendarstellung* spricht man, wenn das Darstellungssystem die durch die menschliche Physiologie gestellten Anforderungen erfüllt. Dazu gehören mindestens drei Farbkanäle, ausreichender Werteumfang der Farbsignale und hinreichend sorgfältig gewählte Grundfarben.

16.7 Bildfolgen

Die letzte Verallgemeinerung des mathematischen Modells für digitalisierte Bilddaten sind Bildfolgen. Ein in Kapitel 16.6 definiertes mehrkanaliges Bild kann zu verschiedenen Zeitpunkten $t = 0, 1, 2, ...$ aufgezeichnet werden. Das Modell wird dazu folgendermaßen erweitert:

$G = \{0, 1, 2, ..., 255\}$	Grauwertmenge mit 256 Grauwerten.
$\mathbf{S} = (s(x, y, n, t))$	vierdimensionale Bildmatrix
$x = 0, 1, ..., L-1$	L Bildzeilen
$y = 0, 1, ..., R-1$	R Bildspalten
$n = 0, 1, ..., N-1$	N Kanäle
$t = 0, 1, ..., T-1$	T Bilder (Frames)
$(\mathbf{s}(x, y)) = (\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{T-1})$	Bildpunkt in der Position (x, y) als Matrix geschrieben.
$\mathbf{g}_t = (g_{t0}, g_{t1}, ..., g_{tN-1})^T$	N-dimensionaler Vektor der Grauwerte des N-kanaligen Bildes **S** zum Zeitpunkt t.
$g_{tn} \in G$	Grauwert des Bildpunktes in der Position (x, y) zum Zeitpunkt t im Kanal n.

(16.26)

Als Beispiel kann die Digitalisierung eines Farbvideofilms dienen: Hier fallen bei der derzeitigen Technik 25 Vollbilder pro Sekunde an, die von einem Farbvideoframegrabber pro Bildpunkt in ein 24-Bit-RGB-Tripel umgesetzt werden. Ein Problem war hier lange Zeit nicht die Digitalisierung, sondern die Abspeicherung der enormen Datenmengen, die bei längeren Bildsequenzen anfallen. Das Fassungsvermögen der heutigen Festplatten hat hier Abhilfe geschaffen. Bild 16.20 zeigt eine Grauwertbildfolge mit acht Teilbildern. Bei Bildfolgen werden die einzelnen Teilbilder, die zu verschiedenen Zeitpunkten auftreten, oft als *Frames* bezeichnet.

In der *Bilddatencodierung* werden Techniken angewendet, die die enormen Datenmengen verdichten, ohne dass dabei ein merklicher Qualitätsverlust auftritt. Eine Einführung in diese Thematik wird z.B. in [Heyn03] gegeben.

Für die hier behandelten Verfahren wird oft auf Indizes, die im Rahmen eines Algorithmus nicht benötigt werden, aus Gründen der Übersichtlichkeit verzichtet. Dazu einige Beispiele:

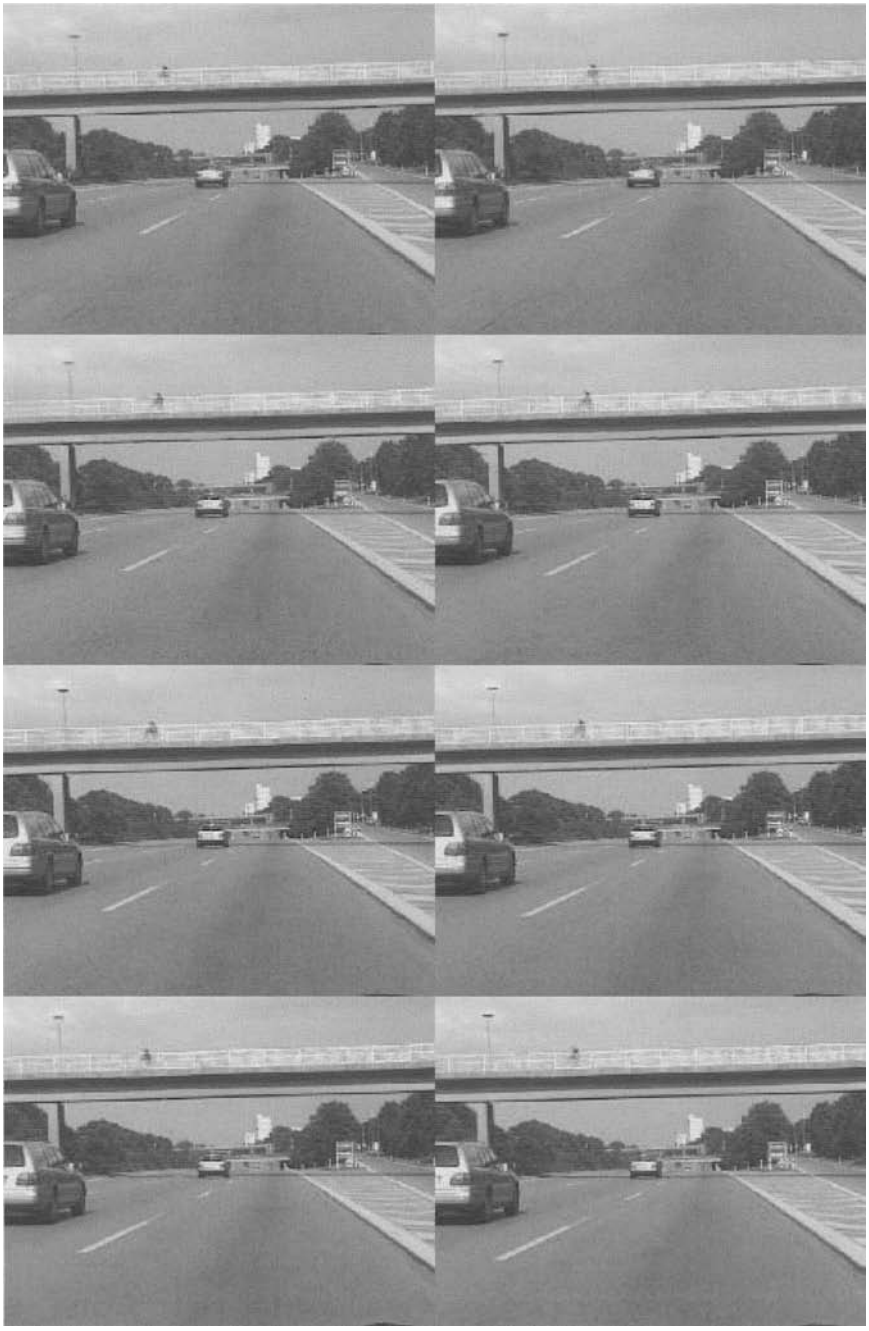


Bild 16.20: Beispiel einer Bildfolge mit acht Teilbildern (Frames).

$\mathbf{S} = (s(x,y))$	Statisches Grauwertbild ohne Zeitachse, z.B. ein digitalisiertes Grauwertbild (Halbtonbild).
$\mathbf{S} = (s(x,y,t))$	Dynamisches Grauwertbild mit Zeitachse. Zu den Zeitpunkten $t = 0, 1, ..., T-1$ fallen die Grauwertbilder $(s(x,y,t))$ an, z.B. digitalisierte Videograutonbilder mit 50 Halbbildern pro Sekunde.
$\mathbf{S} = (s(x,y,n))$	Statisches, mehrkanaliges Bild ohne Zeitachse, z.B. ein digitalisiertes Farbbild.
$\mathbf{S} = (s(x,y,n,t))$	dynamisches, mehrkanaliges Bild (Bildfolge), z.B. digitalisiertes Videofarbbild mit Rot-, Grün- und Blauauszug, 50 Halbbilder pro Sekunde.

16.8 Weitere mathematische Modelle für Bilder

In diesem Abschnitt sind weitere mathematische Modelle für Bilder zusammengestellt. Im Rahmen dieses Buches werden weitgehend die Modelle von oben verwendet, in denen digitalisierte Bilder als mehrdimensionale Felder aufgefasst wurden. In der Literatur findet man häufig auch andere Betrachtungsweisen, so z.B. Bilder als Funktionen von zwei reellen oder diskreten Variablen. Auch auf die statistische Beschreibung von Bildern mit Zufallsvariablen wird in diesem Abschnitt eingegangen.

16.8.1 Bilder als relle Funktionen zweier reeller Variablen

Ein Grauwertbild ist ein zweidimensionales Gebilde, bei dem sich die Helligkeit von Punkt zu Punkt ändert. Dieser Sachverhalt kann mathematisch mit reellen Funktionen der beiden (Orts-) Variablen x und y beschrieben werden: $s(x,y)$ ist der Grauwert an der Stelle (x,y). Gewöhnlich wird angenommen, dass sich diese Funktionen „analytisch wohlverhalten", so z.B., dass sie integrierbar sind oder dass sie eine Fouriertransformierte besitzen. Weiter wird angenommen, dass die Ortskoordinaten x und y nicht negativ und beschränkt sind. Für die Grauwerte $s(x,y)$ wird angenommen, dass sie beschränkt sind. Damit ergibt sich folgende Darstellung:

$$
\begin{aligned}
s(x,y) \quad & \text{reelle Funktion der beiden reellen (Orts-)Variablen } x \text{ und } y, \\
& \text{wobei gilt:} \\
& 0 \le x \le L-1, \\
& 0 \le y \le R-1 \text{ und} \\
& g_{min} \le s(x,y) \le g_{max}.
\end{aligned}
\tag{16.27}
$$

Soll in dieser Schreibweise ein Farb- oder Multispektralbild oder ein allgemeines N-kanaliges Bild dargestellt werden, so wird es als Vektor geschrieben:

$$
\mathbf{s} = \big(s_0(x,y), s_1(x,y), ..., s_{N-1}(x,y)\big)^T. \tag{16.28}
$$

Die Funktion $s_n(x,y)$ steht hier für den Kanal n, $n = 0, 1, ..., N-1$.

16.8.2 Bilder als (diskrete) Funktionen zweier diskreter Variablen

Die Digitalisierung (Rasterung des Ortsbereiches und Quantisierung der Grauwerte) bewirkt, dass die Grauwerte des Bildes nur mehr an bestimmten, diskreten Stellen (x, y) vorliegen. Damit ist

$$\begin{aligned} s(x,y) \quad & \text{eine Funktion der beiden diskreten (Orts-)Variablen} \\ & x \text{ und } y \text{ mit} \\ & x = 0, 1, ..., L-1, \\ & y = 0, 1, ..., R-1 \text{ und} \\ & g_{min} \leq s(x,y) \leq g_{max}. \end{aligned} \tag{16.29}$$

In (16.29) wurden die Grauwerte wie im kontinuierlichen Fall (16.27) als beschränkt angenommen. Liegen die Grauwerte ebenfalls quantisiert vor, so tritt an die Stelle des Grauwertintervalls $[g_{min}, g_{max}]$ die Grauwertmenge $G' = \{z_1, ..., z_k\}$.

Ein allgemeines N-kanaliges Bild kann wie in (16.28) als N-dimensionaler Vektor der N diskreten Funktionen $s(x, y)$ geschrieben werden.

Wird $s(x, y)$ nicht als diskrete Funktion zweier Variablen, sondern als Matrix aufgefasst, so ergibt sich der Querbezug zu den mathematischen Modellen in den Abschnitten 16.3 bis 16.7. Dort wurde einschränkend statt der allgemeinen Grauwertmenge G' die spezielle Grauwertmenge $G = \{0, 1, ..., 255\}$ verwendet.

16.8.3 Bilder als Zufallsprozesse

In manchen Anwendungsgebieten, z.B. bei der Übertragung von Bildern, ist es sinnvoll, Bilder nicht *deterministisch* sondern *statistisch* zu beschreiben. Dabei sind verschiedene Betrachtungsweisen möglich. Sie werden im Folgenden kurz erläutert.

Eine Möglichkeit ist die Interpretation *aller* Grauwerte $g = s(x, y)$ eines Bildes als Realisationen *einer* Zufallsvariablen Z, wobei hier die Anordnung der Grauwerte im (x, y)-Koordinatensystem keine Rolle spielt. Diese Zufallsvariable besitzt eine bestimmte Verteilung, die z.B. durch die Verteilungsfunktion beschrieben werden kann. Die Verteilungsfunktion ist definiert als die Wahrscheinlichkeit, dass die Zufallsvariable Z Werte annimmt, die kleiner oder gleich einem Wert z sind:

$$F(z) = p(Z \leq z). \tag{16.30}$$

Dabei kann es sich um eine *stetige* oder *diskrete Verteilung* handeln. Sind als Grauwerte eines Bildes $\mathbf{S}$ alle reellen Zahlen eines Intervalls zulässig und besitzt $F(z)$ eine stetige Ableitung $f(z)$, so wird $\mathbf{S}$ mit einer stetigen Zufallsgröße Z beschrieben. Die Funktion $f(z)$ heißt die *Dichte* der Verteilung.

Ein Bild $\mathbf{S} = (s(x,y))$ mit der Grauwertmenge $G = \{0, 1, ..., 255\}$ wird dagegen mit einer diskreten Zufallsvariablen Z beschrieben, die die Grauwerte g_i mit den Wahrscheinlichkeiten p_i, $i = 0, 1, ...255$, annimmt. Dies wird im Folgenden vorausgesetzt. Die *Wahrscheinlichkeitsfunktion* $f(z)$ ist dann definiert als

$$f(z) = \begin{cases} p_i & \text{für } z = g_i, \\ 0 & \text{für alle übrigen } z. \end{cases} \tag{16.31}$$

Das in Abschnitt 16.11 erläuterte Histogramm der relativen Häufigkeiten ist eine Schätzung der Wahrscheinlichkeitsfunktion $f(z)$. Die relative Summenhäufigkeit ist eine Schätzung der Verteilungsfunktion $F(z)$.

Der *Erwartungswert* der Zufallsgröße Z ist:

$$E(Z) = \sum_i g_i p_i \tag{16.32}$$

und die *Streuung* (*Varianz*) von Z ist

$$\sigma^2 = E\Big((Z - E(Z))^2\Big) = \sum_i \Big(g_i - E(Z)\Big)^2 p_i. \tag{16.33}$$

Der Mittelwert in Abschnitt 16.11 ist ein Schätzwert für den Erwartungswert und die mittlere quadratische Abweichung ein (allerdings nicht erwartungstreuer) Schätzwert für die Streuung. Die Größe σ wird als *Standardabweichung* bezeichnet.

Es wird jetzt ein N-kanaliges Bild $\mathbf{S} = (s(x,y,n))$, $n = 0, 1, ..., N-1$ vorausgesetzt. Die Grauwerte des Kanals n werden als Realisationen der Zufallsvariablen Z interpretiert. Das bedeutet, dass $\mathbf{S}$ durch eine N-dimensionale Zufallsgröße

$$Z = \Big(Z_0, Z_1, ..., Z_{N-1}\Big) \tag{16.34}$$

beschrieben wird.

Die *Kovarianz* des Kanals n_1 mit dem Kanal n_2 ist definiert als:

$$V_{n_1,n_2} = E\Big((Z_{n_1} - E(Z_{n_1}))(Z_{n_2} - E(Z_{n_2}))\Big). \tag{16.35}$$

Falls $n_1 = n_2$, ergibt sich die oben erläuterte Streuung. Einen Schätzwert für die Kovarianz bekommt man mit

$$v_{n_1,n_2} = \frac{1}{M} \sum_{x=1}^{L-1} \sum_{y=1}^{R-1} \Big(s(x,y,n_1) - m_{\mathbf{S},n_1}\Big)\Big(s(x,y,n_2) - m_{\mathbf{S},n_2}\Big). \tag{16.36}$$

Dieser Ausdruck kann ähnlich wie bei der einfacheren Berechnung der mittleren quadratischen Abweichung umgeformt werden (Abschnitt 16.11):

$$v_{n_1,n_2} = \frac{1}{M} \sum_{x=1}^{L-1} \sum_{y=1}^{R-1} s(x,y,n_1)s(x,y,n_2) - m_{\mathbf{S},n_1} m_{\mathbf{S},n_2}. \tag{16.37}$$

Die Kovarianz kann für jedes Paar (i,j), $i,j \in \{0,1,...,N-1\}$ von Kanälen des Bildes $\mathbf{S}$ berechnet werden. Man erhält damit die *Kovarianzmatrix* $\mathbf{C}$:

$$\mathbf{C} = (v_{i,j}) = \begin{pmatrix} v_{0,0} & v_{0,1} & \cdots & v_{0,N-1} \\ v_{1,0} & v_{1,1} & \cdots & v_{1,N-1} \\ \cdots & \cdots & \cdots & \cdots \\ v_{N-1,0} & v_{N-1,1} & \cdots & v_{N-1,N-1} \end{pmatrix}. \tag{16.38}$$

Im Weiteren wird die Schätzung $\mathbf{C}$ vereinfachend als Kovarianzmatrix bezeichnet. $\mathbf{C}$ ist wegen $v_{i,j} = v_{j,i}$ symmetrisch zur Hauptdiagonale.

Werden die Elemente der Hauptdiagonale der Kovarianzmatrix zu 1 normiert, so wird daraus die *Matrix der Korrelationskoeffizienten*:

$$\mathbf{K} = (r_{i,j}) = \left(\frac{v_{i,j}}{\sqrt{v_{i,i} v_{j,j}}} \right). \tag{16.39}$$

Die Korrelationskoeffizienten liegen im Intervall $-1 \leq r_{i,j} \leq +1$. Die Zufallsvariablen Z_i und Z_j sind *unkorreliert*, wenn $r_{i,j} = 0$ ist.

Nun zu einer etwas anderen Betrachtungsweise. Hier wird auch die Position (x,y) eines Bildpunktes berücksichtigt. Dazu ein beispielhafter Versuchsaufbau: Ein Original wird mit einer Videokamera aufgezeichnet und über einen Videodigitalisierer (*framegrabber*) in einen Rasterbildspeicher geschrieben. Die Videokamera tastet das Original mit einer Bildwiederholungsrate von 50 Halbbildern pro Sekunde ab, so dass der *framegrabber* 50 Halbbilder pro Sekunde in den Rasterbildspeicher schreibt.

Der Grauwert $s(x,y)$ des Bildpunktes in der Position (x,y) im Rasterbildspeicher ist die codierte mittlere Helligkeit eines kleinen Flächenausschnittes des Originals. Dieser an sich konstante Grauwert wird durch verschiedene Quellen verrauscht, z.B. durch die atmosphärischen Bedingungen zwischen Originalbild und Videokamera, durch elektronisches Rauschen in der Videokamera oder durch Quantisierungsrauschen im *framegrabber*. Das hat zur Folge, dass der Grauwert $s(x,y)$ nicht konstant ist, sondern um einen bestimmten Mittelwert schwankt.

Das Verhalten des Grauwertes in der Position (x,y) wird durch die Zufallsvariable $Z_{T,(x,y)}$ beschrieben. Der zusätzliche Index T steht für den Zufallsversuch T mit den Ereignissen $T = t_0, t_1, ...$. Im vorliegenden Beispiel kann dazu folgende Interpretation gegeben werden: Zum Zeitpunkt t_i tritt das Ereignis „Bild zum Zeitpunkt t_i" ein, das durch die diskrete Funktion $Z_{t_i,(x,y)}$ beschrieben wird. Damit sind unterschiedliche Interpretationen möglich:

- Für einen festen Wert t ist $Z_{T,(x,y)}$ eine Bildfunktion der beiden diskreten (Orts-)Variablen x und y. Für $i = 0,1,2,...$ ist somit eine Menge von Bildfunktionen gegeben.

- Für einen festen Punkt (x,y) und für variables t ist $Z_{T,(x,y)}$ eine Zufallsvariable für den Bildpunkt in der Position (x,y).

Unter diesen Voraussetzungen wird $Z_{T,(x,y)}$ als *zweidimensionaler Zufallsprozess* oder *stochastischer Prozess* (*random field*) bezeichnet. Damit können statistische Kenngrößen für den Zufallsprozess $Z_{T,(x,y)}$ formuliert werden:

- Der *Erwartungswert* oder *Mittelwert* des Zufallsprozesses für den Punkt (x, y):

$$E\Big(Z_{T,(x,y)}\Big). \tag{16.40}$$

- Die *Streuung* (*Varianz*) des Zufallsprozesses für den Punkt (x, y):

$$E\Big((Z_{T,(x,y)} - E(Z_{T,(x,y)}))^2\Big). \tag{16.41}$$

- Die *Autokorrelation* des Zufallsprozesses als Erwartungswert des Produktes der beiden Zufallsvariablen $Z_{T,(x_1,y_1)}$ und $Z_{T,(x_2,y_2)}$:

$$K_{(x_1,y_1),(x_2,y_2)} = E\Big(Z_{T,(x_1,y_1)} Z_{T,(x_2,y_2)}\Big). \tag{16.42}$$

- Die *Autokovarianz* des Zufallsprozesses

$$\begin{aligned} V_{(x_1,y_1),(x_2,y_2)} &= \\ &= E\Big((Z_{T,(x_1,y_1)} - E(Z_{T,(x_1,y_1)}))(Z_{T,(x_2,y_2)} - E(Z_{T,(x_2,y_2)}))\Big) = \\ &= K_{(x_1,y_1),(x_2,y_2)} - E\Big(Z_{T,(x_1,y_1)}\Big) E\Big(Z_{T,(x_2,y_2)}\Big). \end{aligned} \tag{16.43}$$

Oft wird für den Erwartungswert $E(Z_{T,(x,y)}) = 0$ angenommen. Dann kann ein Zufallsprozess allein durch seine Autokorrelation $K_{(x_1,y_1),(x_2,y_2)}$ beschrieben werden. Wird die oben beschriebene Versuchsanordnung so gewählt, dass anstelle des Originalbildes, das während des gesamten Versuchs konstant blieb, alle möglichen zweidimensionalen Grauwertverteilungen gleichwahrscheinlich auftreten können, so wird der Mittelwert $E(Z_{T,(x,y)})$ unabhängig von der Position (x, y) und die Autokorrelation verschiebungsinvariant:

$$K_{(x_1,y_1),(x_2,y_2)} = K_{(x_1+\Delta x,y_1+\Delta y),(x_2+\Delta x,y_2+\Delta y)}. \tag{16.44}$$

Setzt man für $(\Delta x, \Delta y) = (-x_2, -y_2)$ so ergibt sich

$$K_{(x_1,y_1),(x_2,y_2)} = K_{(x_1-x_2,y_1-y_2),(0,0)} = K_{(\alpha,\beta)}, \tag{16.45}$$

wobei α und β die Koordinatendifferenzen $x_1 - x_2$ und $y_1 - y_2$ sind.

Ein Zufallsprozess dieser Art heißt *homogen*. Er kann durch seine Autokorrelation beschrieben werden, die statt von einem Punktepaar (x_1, y_1) und (x_2, y_2) nur von der Koordinatendifferenz von Punktepaaren abhängt. Bei vielen Anwendungen werden in der Literatur Bilder als homogene Zufallsprozesse vorausgesetzt. Schätzungen dieser Kenngrößen berechnen sich sinngemäß wie oben. Ein Schätzwert für den Mittelwert:

$$m_{(x,y)} = \frac{1}{T} \sum_{t=0}^{T-1} s(x, y, t), \tag{16.46}$$

ein Schätzwert für die Streuung:

$$q_{(x,y)} = \frac{1}{T} \sum_{t=0}^{T-1} s(x, y, t)^2 - m_{(x,y)}^2; \tag{16.47}$$

und ein Schätzwert für die Kovarianz:

$$v_{(x_1,y_1),(x_2,y_2)} = \frac{1}{T} \sum_{t=0}^{T-1} s(x_1, y_1, t) s(x_2, y_2, t) - m_{(x_1,y_1)} m_{(x_2,y_2)}. \tag{16.48}$$

Die Kovarianz wird für alle Paare von Bildpunkten (x_k, y_k), (x_l, y_l) berechnet. Die Kovarianzmatrix ist dann die Matrix der Schätzwerte und hat die Dimension $M \cdot M$, wobei $M = L \cdot R$ die Anzahl der Bildpunkte von $\mathbf{S}$ ist.

16.9 Bildliche Reproduktion von digitalisierten Bildern

16.9.1 Geräte zur Bilddarstellung

Nachdem in den vorausgehenden Abschnitten erläutert wurde, wie Bilder digitalisiert und mit welchen Modellen sie beschrieben werden können, werden jetzt einige Möglichkeiten diskutiert, wie diese digitalisierten Daten wieder bildlich dargestellt werden können. Die bildliche Reproduktion ist notwendig, da die visuelle Beurteilung von Zwischenergebnissen einer Folge von Verarbeitungsschritten oft das weitere Vorgehen beeinflussen kann und die Endprodukte oft am anschaulichsten wieder als Bilder ausgegeben werden.

Mit den modernen Workstations und PC-Systemen und den in ihrer Umgebung installierten Zusatzgeräten hat man heute ebenfalls die Möglichkeit, gute bis sehr gute Bilddarstellungen zu erreichen. Moderne Grafikkarten erzeugen in Verbindung mit guten Monitoren meistens akzeptable Bilder. Für Videoprojektionen stehen Projektoren zur Verfügung, die z.B. an eine Grafik-Karte angeschlossen werden können. Diese Geräte, die derzeit noch teuer sind, sollten aber mindestens $1024 \cdot 768$ Pixel fehlerlos abbilden und eine ausreichende Lichtstärke besitzen.

Um die Bilder auf Grafikkarten darstellen zu können, benötigt man noch eine passende Software, die jedoch für alle Betriebssystemplattformen kostenlos oder mehr oder weniger kostengünstig angeboten wird. Man hat hier eine Vielzahl von Bildformaten definiert (BMP, TIF, JPEG, MPEG, EPS, PS, ...), die meistens kompatibel sind. Auch zur Konvertierung dieser Bilddatenformate gibt es viele Programme, zum Teil kostenlos aus dem Internet.

Zur Archivierung von Bilddaten werden gerne CD-ROM oder DVD (*digital versatile disk*) verwendet. Wenn man die passende Software hat, so kann man die Bilder so ausgeben, dass man sie auch ohne PC, nur mit einem geeigneten CD- oder DVD-Laufwerk und einem Fernsehgerät betrachten kann.

Als weitere Möglichkeit der Bildausgabe ist die Vielfalt der Drucker zu erwähnen. Hier sind Tintenstrahl- oder Laserdrucker, in Schwarz-/Weiß- oder Farbausführung im Einsatz. Für hochwertige Farbausdrucke gibt es Thermotransfer- und Thermosublimationsdrucker. Ist das Bild erst einmal auf der Grafikkarte dargestellt, so ist der Weg zum Drucker nicht mehr schwierig, da die unterschiedlichen Softwaresysteme zur Bildbearbeitung meistens Möglichkeiten zum Drucken vorsehen. Die Systeme verwenden dabei in der Regel die Druckertreiber, die im Rahmen des Betriebssystems installiert wurden. Um qualitativ hochwertiges Bildmaterial erzeugen zu können, genügen heute oft schon relativ kostengünstige Farbtintenstrahldrucker.

Schon aus dem bis jetzt Gesagten ist abzuleiten, dass der Benutzer sich nicht mit der Hard- und Softwareseite der Bilddarstellung befassen muss. Selbst wenn er eigene Programme erstellt, bieten die Programmiersysteme fast immer einfache Schnittstellen zur Bilddarstellung an.

Im folgenden Abschnitt werden einige Verfahren zur Bilddarstellung erläutert. Einige davon sind heute nicht mehr Stand der Technik. Ihre Darstellung wurde für interessierte Leser beibehalten.

16.9.2 Ausdrucken der Grauwerte

Die einfachste Möglichkeit ist das zeilenweise Ausdrucken der Grauwerte eines Bildes $\mathbf{S} = (s(x, y))$ mit $G = \{0, 1, ..., 255\}$ als Dezimalzahlen über einen Drucker. Je nach Papierbreite können mehr oder weniger Bildpunkte pro Zeile ausgegeben werden. Ein Bildpunkt braucht dabei vier Druckstellen (drei Ziffern und einen Zwischenraum). Die Anzahl der Bildzeilen ist, abgesehen vom Papierverbrauch, nicht begrenzt. Sollen breitere Ausschnitte ausgegeben werden, so werden sie auf getrennten Papierbahnen nacheinander gedruckt und anschließend zusammengeklebt. Ein bildlicher Eindruck wird mit Ausgaben dieser Art nur schwer vermittelt, da

- die Grauwerte nicht als unterschiedliches Grau, sondern als Dezimalzahlen ausgegeben werden,
- die Darstellung eines Bildpunktes mit drei Ziffern und Zwischenraum gegenüber der bei der Digitalisierung verwendeten Rasterflächengröße eine enorme Vergrößerung ist, und schließlich

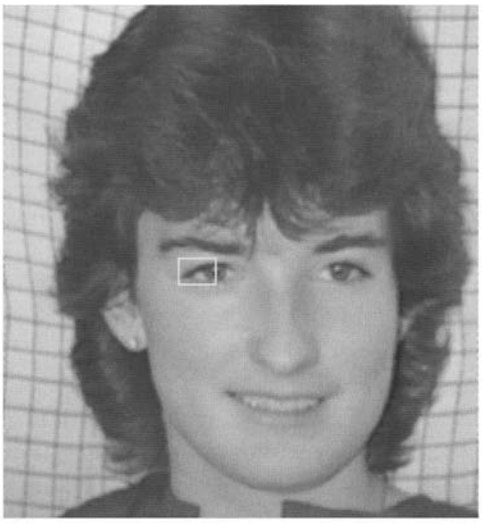
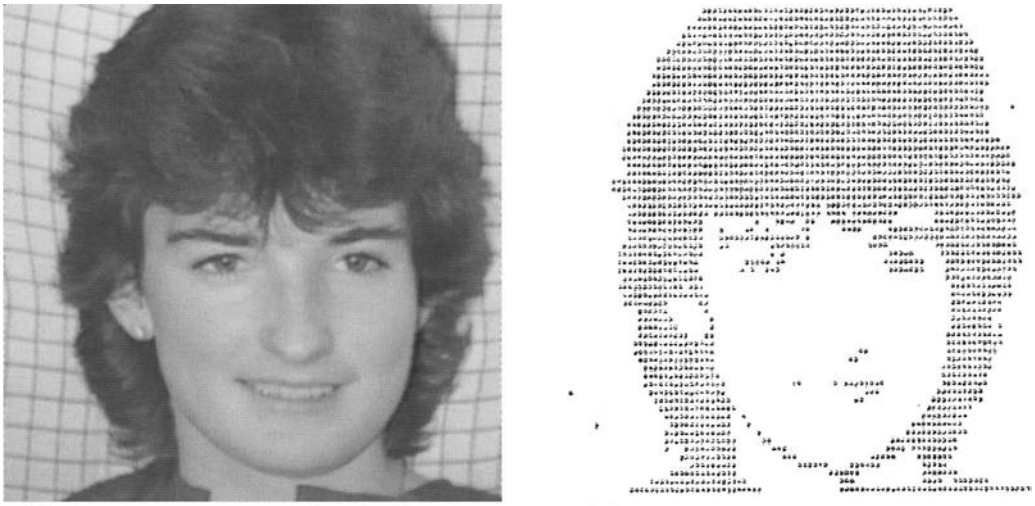


Bild 16.21: Ausgabe der Grauwerte eines Bildes in dezimaler Darstellung.

- die Rasterflächen keine Quadrate sondern Rechtecke sind, wodurch die Bilder verzerrt werden.

Trotzdem ist es manchmal hilfreich, wenn kleinere Bildausschnitte in dieser Form vorliegen. Ein Beispiel dazu zeigt Bild 16.21.



(a) (b)

Bild 16.22: Ausgabe eines Zweipegelbildes auf einem Zeilendrucker. (a) Original. (b) verkleinerter Ausdruck.

16.9.3 Ausgabe von logischen Bildern

Ein Zweipegelbild $\mathbf{S} = (s(x,y))$ mit $G = \{g_1, g_2\}$ kann über einen Zeilendrucker als Binärbild ausgegeben werden, wenn die Bildzeilen folgendermaßen aufgebaut werden:

$$\text{Ausgabebildpunkt} = \begin{cases} \text{' '} \text{ (weiße Leerstelle),} & \text{falls } s(x,y) = g_1, \\ \text{'*'} \text{ (oder beliebiges anderes Zeichen),} & \text{falls } s(x,y) = g_2. \end{cases}$$

Ein Bildpunkt benötigt bei dieser Ausgabe eine Druckstelle. Die Ausgaben vermitteln einen bildhaften Eindruck (Bild 16.22), wenn auch die starke Vergrößerung gegenüber dem Original bei großen Bildern einen hohen Papierverbrauch bedingt.

Die Verzerrung, die durch die nicht quadratischen Rasterflächen hervorgerufen wird, kann durch verschiedene Maßnahmen unterdrückt werden. Bei manchen Druckern ist es möglich, den Zeilenabstand zu verkleinern, wodurch die Streckung des Bildes in Zeilenrichtung nicht mehr so stark ist. Als weitere Maßnahme kann man die Bilder vor der Druckerausgabe in Zeilenrichtung mit einem geeigneten Faktor stauchen. Bei der Ausgabe wird dann die gerätebedingte Streckung ausgeglichen. Den Stauchungsfaktor ermittelt man am besten durch die Ausgabe eines quadratischen Bildes, bei dem man die Zeilenbreite und die Bildlänge ausmisst. Der Quotient dieser beiden Messwerte ist der gesuchte Faktor. Ein Beispiel einer Binärbildausgabe zeigt Bild 16.22-b, das fotografisch verkleinert wurde.

Auch logische Bilder $\mathbf{S} = (s(x,y))$ mit einer nicht zu umfangreichen Grauwertmenge $G = \{g_1, g_2, ...g_k\}$ können auf diese Weise ausgedruckt werden, wenn jedem Grauwert

Grauwert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Zeile																
1	M	M	M	M	H	H	O	X	X	M	*	=	¨	-	.	
2	W	W	W	W	&	*	=	=	-							
3	&	&	&	&	-											
4	0	0	0													
5	*	+														

Tabelle 16.1: Ausgabe von Grauwertbildern über einen Zeilendrucker. Die Grautöne werden durch mehrmaliges Überdrucken von Zeichen erzeugt. Die Tabelle enthält eine Codierung von 16 Graustufen durch maximal fünfmaliges Überdrucken.

ein druckbares Zeichen zugeordnet wird. Der bildhafte Charakter wird dadurch allerdings wieder verwischt, da die einzelnen Zeichen oft nicht gut zu unterscheiden sind.

16.9.4 Zeilendruckerausgabe von Grauwertbildern

Die Bedeutung der Bildausgabe auf Zeilendruckern (Walzendrucker, Trommeldrucker) ist nicht mehr sehr groß, da es kaum noch Drucker dieser Bauart gibt. Wenn es möglich ist, den Zeilenvorschub zu unterdrücken, können Grauwertbilder $\mathbf{S} = (s(x,y))$ mit $G = \{0,1,...,255\}$ durch das Übereinanderdrucken mehrerer Zeichen erzeugt werden. Damit sind natürlich nicht 256 verschiedene Graustufen zu unterscheiden, aber 16 oder 32 Stufen lassen sich, je nach Bauart des Druckers, schon erreichen. In Tabelle 16.1 ist eine Codierung von insgesamt 16 verschiedenen Graustufen durch maximal fünfmaliges Überdrucken zusammengestellt:

Die einzelnen Druckzeichen pro Grauton müssen in praktischen Anwendungsfällen an den jeweiligen Drucker angepasst werden. Zur Beurteilung der Qualität gibt man am besten einen Graukeil aus und prüft durch Betrachten aus einiger Entfernung, ob sich ein einigermaßen kontinuierlicher Übergang von dunkel nach hell ergibt. Für die Bildgröße und die auftretende Verzerrung gelten dieselben Aussagen wie in Abschnitt 16.9.3. Bild 16.23 zeigt zwei Beispiele zur Bildausgabe von Grauwertbildern über einen Zeilendrucker, die für diese Darstellung verkleinert wurden.

Wenn die so erzeugten Ausdrucke in einem Fotolabor weiterverarbeitet werden können, ist es möglich, auch Farbdarstellungen zu erzeugen. Angenommen $\mathbf{S} = (s(x,y,n)), n = 0,1,2$ sei ein dreikanaliges Bild mit Rot-, Grün- und Blauauszug. Jeder Kanal wird für sich als Grautonbild ausgedruckt und fotografisch verkleinert. Eine Farbdarstellung kann jetzt erzeugt werden, wenn bei der Belichtung des Farbmaterials der Reihe nach der Ausdruck des roten, grünen und blauen Kanals überlagert wird.

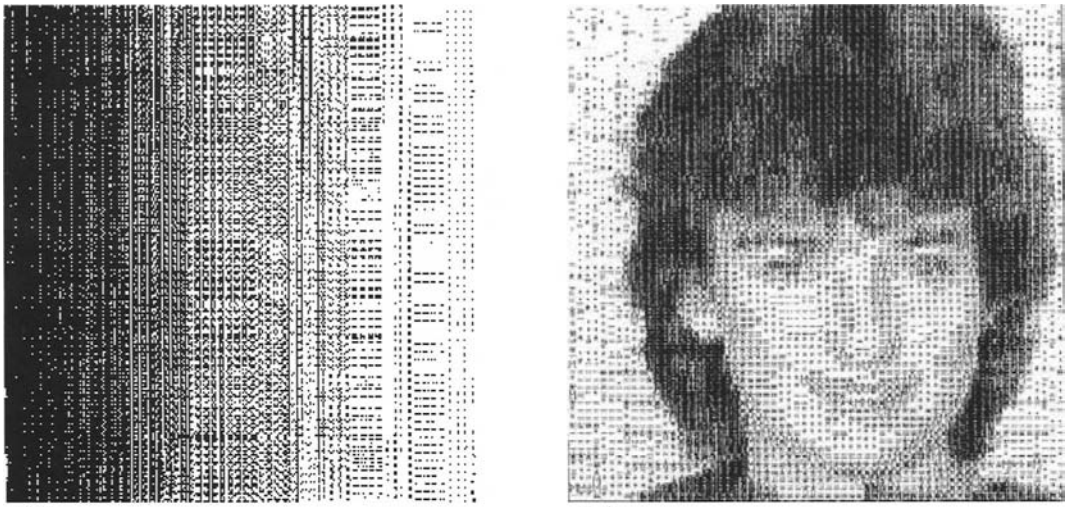


Bild 16.23: Bildausgabe von Grauwertbildern über einen Zeilendrucker durch maximal fünfmaliges Übereinanderdrucken. Die Bilder wurden verkleinert. (a) Graukeil; (b) Testbild

16.9.5 Halbtonverfahren

Halbtonverfahren können bei Schwarz-/Weiß-Ausgabegeräten (Binärdisplaygeräte) eingesetzt werden. Gebräuchliche Beispiele dazu sind Matrixdrucker, Laserdrucker oder Tintenstrahldrucker. Diese Geräte können entweder nur einen kleinen schwarzen Punkt (*dot*) oder keinen Punkt (weißen Punkt) ausgeben. Da die Auflösung der Geräte wesentlich besser ist als die des Auges, verschwimmen nahe beieinander stehende *dots* zu einer schwarzen Fläche. Übliche Auflösungen sind 600 *dpi* (*dots per inch*), 1200 *dpi* oder größer.

Durch eine geschickte Anordnung der *dots* lassen sich gute Grautonbilder erzeugen. Zur automatischen, grauwertabhängigen Anordnung der *dots* werden verschiedene Verfahren verwendet, die im Folgenden kurz zusammengefasst sind.

16.9.5.1 Binärbildausgabe

Analog zu Abschnitt 16.9.3 kann mit einem Binärdisplaygerät ein Zweipegelbild $\mathbf{S}$ mit $G = \{g_1, g_2\}$ als Binärbild ausgegeben werden, wenn für alle Grauwerte g_1 (oder alternativ g_2) ein schwarzer Punkt erzeugt wird.

Ein Grauwertbild $\mathbf{S}$ mit $G = \{0, 1, ..., 255\}$ kann mit dieser Technik ebenfalls als Binär- oder Zweipegelbild ausgegeben werden. Es muss dazu aber vor der Ausgabe mit Hilfe eines

Bild 16.24: Binärbildausgabe von Grauwertbildern auf Binärdisplaygeräten. (a) Darstellung des Testbildes als Binärbild (Schwellwert $c = 140$). (b) Größenproportionale Codierung.

geeigneten Schwellwertes c binarisiert werden:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a : s_a(x,y) = \begin{cases} 0, & s_e(x,y) \leq c \\ 255, & s_e(x,y) > c \end{cases} \tag{16.49}$$

Die Anzahl der Bildpunkte pro Zeile ist durch die Anzahl der Punkte des Gerätes begrenzt. Es können also Bilder mit 500, 1000 oder mehr Bildspalten ausgegeben werden. Die Bilder sind in der Regel nicht verzerrt, da bei diesen Geräten der Spalten- und Zeilenabstand meistens gleich ist (Bild 16.24-a).

16.9.5.2 Größenproportionale Codierung

Bei der größenproportionalen Codierung wird der Grauwert des Bildes durch eine geometrische Figur, eine Kreisscheibe, eine Quadratfläche, ein Kreuz oder eine Linie dargestellt. Die Größe der Figur wird abhängig vom Grauwert gewählt (Bild 16.24-(b)). Sehr eindrucksvolle Bilder hat der bereits verstorbene Grafiker Victor Vasarely aus Gordes, in Südfrankreich erstellt. Durch die Kombination von geometrischen Figuren und deren Verzerrung hat Vasarely erstaunliche plastische Effekte erzielt. Er hat seine Bilder allerdings ohne Computer gezeichnet.

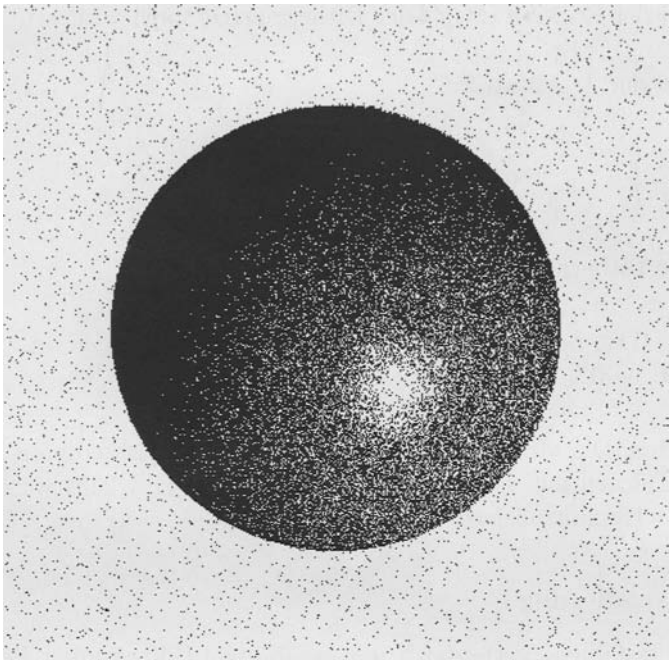


Bild 16.25: Beispiel einer Bildausgabe mit dem Zufallszahlenalgorithmus.

16.9.5.3 Multischwellwertverfahren, Dither-Verfahren

Sehr weit verbreitet sind Verfahren, bei denen nicht ein Schwellwert c, sondern mehrere verwendet werden. Die Schwellwerte, die aus einer *Dithermatrix* entnommen werden, ändern sich von Bildpunkt zu Bildpunkt. Eine $4 \cdot 4$-Dithermatrix könnte etwa wie folgt aussehen:

$$\begin{pmatrix} 0 & 128 & 32 & 160 \\ 192 & 64 & 224 & 96 \\ 48 & 176 & 16 & 144 \\ 240 & 112 & 208 & 80 \end{pmatrix} \tag{16.50}$$

Bei der Ausgabe eines Bildpunktes in der Position (x, y) werden zu den Koordinatenwerten die Größen

$$x' = x \text{ modulo } n \text{ und } y' = y \text{ modulo } n \tag{16.51}$$

berechnet, wobei n die Größe der Dithermatrix ist. Die Werte x' und y' liegen zwischen 0 und $n-1$ und können somit als Positionsangaben in der Dithermatrix angesehen werden. Für den Grauwert $s(x, y)$ wird ein schwarzer Punkt ausgegeben, wenn er kleiner ist als

der Schwellwert in der Position (x', y') der Dithermatrix, sonst wird kein schwarzer Punkt ausgegeben.

Ausgehend von der $2 \cdot 2$-Matrix

$$\mathbf{D}_2 = \begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix} \tag{16.52}$$

können größere Dithermatrizen iterativ gemäß

$$\mathbf{D}_{2n} = \begin{pmatrix} 4\mathbf{D}_n & 4\mathbf{D}_n + 2\mathbf{U}_n \\ 4\mathbf{D}_n + 3\mathbf{U}_n & 4\mathbf{D}_n + \mathbf{U}_n \end{pmatrix} \tag{16.53}$$

berechnet werden. Die Matrizen $\mathbf{U}_n$ sind dabei $n \cdot n$-Matrizen, deren Elemente 1 sind. Zur Ausgabe von Bildern mit der Grauwertmenge $G = \{0, 1, ..., 255\}$ werden die einzelnen Elemente noch mit 16 multipliziert.

16.9.5.4 Zufallscodierung

Ein anderes leistungsfähiges Verfahren zur Ausgabe von Bildern über Binärgeräte ist die Bildausgabe mit Hilfe eines Zufallszahlengenerators. Für jeden Bildpunkt wird eine Zufallszahl r erzeugt, die im Intervall [0,255] gleichverteilt sein muss. Ein schwarzer Punkt wird nur dann ausgegeben, falls der Grauwert $s(x, y) \leq r$ ist. Die Qualität der mit diesem Verfahren erzeugten Bilder hängt von der Güte des Zufallszahlengenerators ab. Diese Technik, die sehr einfach zu implementieren ist, kann mit der Spritztechnik „mit Pinsel und Zahnbürste" verglichen werden, die wohl jeder Schüler einmal im Zeichenunterricht ausprobiert hat. Ein Beispiel zu diesem Verfahren zeigt Bild 16.25.

Die Qualität der Bilder, die mit dem Multischwellwertverfahren oder der Zufallskodierung ausgegeben werden, kann noch verbessert werden, wenn die Verfahren mit einer nicht linearen Modifikation der Grauwerte (Kapitel 17) kombiniert werden.

16.10 Datenreduktion und Datenkompression

Da digitalisierte Bilddaten sehr viel Speicherplatz benötigen, wird oft eine Reduzierung des Speicherbedarfs notwendig. Zunächst kann man auf digitalisierte Bilddaten, die aus der Sicht des Betriebssystems nichts anderes als in Dateien gespeicherte Daten sind, alle Verfahren anwenden, die in der PC-Welt zur Datenkompression verwendet werden. Beispiele dazu sind Sharewareprodukte wie „lharc" oder „ZIP-Manager" oder Backup-Software, die in die Betriebssysteme integriert sind.

Andrerseits gibt es viele Verfahren, die speziell die Tatsache berücksichtigen, dass es sich um Bilddaten handelt. Zunächst aber für diesen Bereich zwei Begriffserläuterungen: Bilddatenreduktion und Bilddatenkompression.

Bei der *Bilddatenreduktion* werden von den Originalbilddaten Bestandteile weggelassen, die im speziellen Anwendungsfall nicht oder nur gering relevant sind. Aus der reduzierten Darstellung kann das Original nicht mehr rekonstruiert werden. Ein Beispiel ist die Reduzierung der Grauwertmenge von 256 auf z.B. 16 Graustufen.

Aus Bildern, die mit Verfahren der *Bilddatenkompression* verarbeitet wurden, kann das Original wieder eindeutig und fehlerfrei rekonstruiert werden. Als Beispiel hierzu kann die *run-length*-Codierung (Kapitel 34) von Binärbildern dienen.

Eine weitere Beurteilungsmöglichkeit der Eignung von Bilddatenreduktions- und -kompressionsverfahren bei einer bestimmten Anwendung ist die Frage, ob es die reduzierte oder komprimierte Form erlaubt, Bildverarbeitungs- und Mustererkennungsverfahren darauf anzuwenden. Wird ein Bild mit einer Archivierungssoftware komprimiert, so ist es sicher nicht möglich, auf die komprimierten Daten einen Bildverarbeitungsoperator, wie beispielsweise den Laplace-Operator, anzuwenden. Man muss die Bilddaten zuerst dekomprimieren, dann den Bildverarbeitungsoperator anwenden und dann das Ergebnis bei Bedarf wieder komprimieren.

Anders ist es als Beispiel bei der *run-length*-Codierung: Hier ist es möglich, mit den komprimierten Bilddaten viele sinnvolle Verarbeitungsschritte durchzuführen. Die Motivation für den Einsatz der *run-length*-Codierung kann somit nicht nur der Gesichtspunkt der Datenkompression, sondern auch die effiziente Verwendung mancher Bildverarbeitungsoperatoren sein.

Wichtige Verfahren zur Bilddatenreduktion wurden von der *Joint Photografic Experts Group*, JPEG, vorgeschlagen und standardisiert. Diese Verfahren bauen auf einer blockweisen Cosinustransformation auf. Für Bildfolgen wurde das MPEG-Format („Moving Picture Experts Group") definiert. An dieser Stelle sei nochmals auf das schon zitierte Buch [Heyn03] verwiesen.

Die „Portable Video Research Group", PRVG, an der Stanford University hat zu diesen Formaten das Softwarepaket *PRVG-JPEG/MPEG codec* entwickelt, das im Interet zusammen mit Handbüchern verfügbar ist.

16.11 Charakterisierung digitalisierter Bilder

In diesem Abschnitt sind wichtige Eigenschaften von digitalisierten Bilddaten und Kenngrößen, die diese Eigenschaften beschreiben, zusammengestellt.

16.11.1 Mittelwert und mittlere quadratische Abweichung

Es sei $\mathbf{S} = (s(x, y))$ ein einkanaliges Grauwertbild mit L Zeilen und R Spalten. Der *mittlere Grauwert* des Bildes $\mathbf{S}$, auch *Mittelwert* von $\mathbf{S}$ genannt, berechnet sich gemäß:

$$m_{\mathbf{S}} = \frac{1}{M} \sum_{x=0}^{L-1} \sum_{y=0}^{R-1} s(x, y), \tag{16.54}$$

wobei $M = L \cdot R$ die Anzahl der Bildpunkte von $\mathbf{S}$ ist. Als Beispiel ist der Mittelwert von Bild 16.26-a $m_{\mathbf{S}} = 129.26$.

Der Mittelwert eines Bildes sagt aus, ob das Bild insgesamt dunkler oder heller ist. Eine Aussage über den Kontrast lässt sich aus dem Mittelwert nicht ableiten. So haben z.B. ein Bild, das nur den Grauwert 127 enthält (Bildgröße beliebig) und ein Bild, das ein Schachbrettmuster mit den Grauwerten 0 und 254 enthält, denselben Mittelwert 127.

Eine Kenngröße, die eine Aussage über den Kontrast im Bild zulässt, ist die *mittlere quadratische Abweichung*:

$$q_{\mathbf{S}} = \frac{1}{M} \sum_{x=0}^{L-1} \sum_{y=0}^{R-1} \Big(s(x,y) - m_{\mathbf{S}}\Big)^2. \tag{16.55}$$

Bild 16.26-a hat eine mittlere quadratische Abweichung $q_{\mathbf{S}} = 4792.79$. Das oben erwähnte homogene Bild mit dem Grauwert 127 hat die mittlere quadratische Abweichung $q_{\mathbf{S}} = 0$, während sich für das Schachbrettmusterbild $q_{\mathbf{S}} = 127^2 = 16129$ ergibt.

Zur einfacheren Berechnung der mittleren quadratischen Abweichung lässt sich (16.55) durch die Anwendung der binomischen Formel umformen:

$$\begin{aligned} q_{\mathbf{S}} &= \frac{1}{M} \sum_{x=0}^{L-1} \sum_{y=0}^{R-1} \Big(s^2(x,y) - 2m_{\mathbf{S}} s(x,y) + m_{\mathbf{S}}^2\Big) = \\ &= \frac{1}{M} \sum_{x=0}^{L-1} \sum_{y=0}^{R-1} s^2(x,y) - \frac{2m_{\mathbf{S}}}{M} \sum_{x=0}^{L-1} \sum_{y=0}^{R-1} s(x,y) + \frac{1}{M} \sum_{x=0}^{L-1} \sum_{y=0}^{R-1} m_{\mathbf{S}}^2 = \\ &= \frac{1}{M} \sum_{x=0}^{L-1} \sum_{y=0}^{R-1} s^2(x,y) - m_{\mathbf{S}}^2. \end{aligned} \tag{16.56}$$

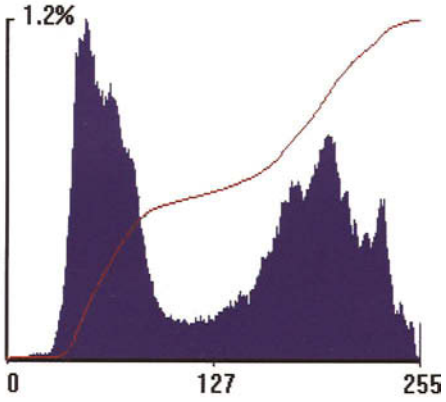
Mit (16.54) und (16.56) lassen sich der Mittelwert und die mittlere quadratische Abweichung eines Bildes in einem Durchgang berechnen.

Bei mehrkanaligen Bildern oder Bildfolgen können Mittelwert und mittlere quadratische Abweichung für jeden Kanal oder für jedes Teilbild getrennt berechnet werden. Für ein mehrkanaliges Bild $\mathbf{S} = (s(x,y,n)), \quad n = 0, 1, ..., N-1$ wird dies im Folgenden dargestellt. Die Vektoren

$$\mathbf{m_S} = \Big(m_{\mathbf{S},0}, m_{\mathbf{S},1}, ..., m_{\mathbf{S},N-1}\Big)^T \quad \text{und}$$
$$\mathbf{q_S} = \Big(q_{\mathbf{S},0}, q_{\mathbf{S},1}, ..., q_{\mathbf{S},N-1}\Big)^T$$

werden dann als *Mittelwertvektor* und *Vektor der mittleren quadratischen Abweichungen* des mehrkanaligen Bildes $\mathbf{S}$ bezeichnet.

(a)



(b)

Bild 16.26: (a) Testbild zur Veranschaulichung der in diesem Abschnitt behandelten Kenngrößen für digitalisierte Bilder. Das Bild hat einen Mittelwert $m_{\mathbf{S}} = 129.26$ und eine mittlere quadratische Abweichung $q_{\mathbf{S}} = 4792.79$. (b) Histogramm: Auf der Abszisse werden die Grauwerte g von 0 bis 255 aufgetragen. Zu jedem Grauwert g wird auf der Ordinate ein Balken gezeichnet, der in der Höhe dem Wert $p_{\mathbf{S}}(g)$ in % entspricht. Da die Werte alle sehr klein sind, wurde die Ordinate entsprechend skaliert. Die monoton steigende Kurve repräsentiert die relativen Summenhäufigkeiten. Sie liegen zwischen 0 und 1. Für diese Kurve gilt eine andere Skalierung der Ordinate.

0 - 15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01
16 - 31	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.02	0.01	0.01	0.02	0.03	0.05	0.07	0.09	0.13
32 - 47	0.16	0.18	0.24	0.31	0.42	0.48	0.59	0.72	0.88	0.99	1.05	1.04	1.10	1.09	1.09	1.12
48 - 63	1.17	1.16	1.13	1.06	1.04	1.04	0.94	0.95	0.93	0.90	0.92	0.86	0.90	0.89	0.92	0.95
64 - 79	0.90	0.90	0.89	0.87	0.83	0.79	0.75	0.73	0.73	0.71	0.72	0.69	0.68	0.68	0.65	0.59
80 - 95	0.56	0.49	0.45	0.42	0.39	0.37	0.34	0.31	0.29	0.26	0.26	0.21	0.20	0.19	0.17	0.17
96 - 111	0.16	0.15	0.14	0.14	0.14	0.15	0.13	0.13	0.13	0.13	0.13	0.12	0.13	0.14	0.13	0.12
112 - 127	0.11	0.12	0.12	0.12	0.14	0.13	0.13	0.15	0.13	0.13	0.12	0.12	0.14	0.13	0.14	0.00
128 - 143	0.13	0.15	0.15	0.16	0.16	0.16	0.17	0.17	0.18	0.17	0.20	0.18	0.17	0.20	0.19	0.22
144 - 159	0.23	0.21	0.22	0.21	0.22	0.19	0.23	0.23	0.23	0.25	0.28	0.28	0.31	0.35	0.35	0.34
160 - 175	0.35	0.37	0.36	0.37	0.41	0.41	0.45	0.43	0.48	0.51	0.54	0.53	0.54	0.54	0.54	0.59
176 - 191	0.61	0.59	0.60	0.60	0.61	0.59	0.57	0.54	0.56	0.58	0.58	0.60	0.62	0.63	0.66	0.66
192 - 207	0.68	0.72	0.72	0.72	0.76	0.76	0.77	0.76	0.76	0.74	0.74	0.67	0.64	0.62	0.55	0.56
208 - 223	0.57	0.57	0.57	0.52	0.46	0.43	0.42	0.48	0.47	0.42	0.38	0.39	0.40	0.43	0.43	0.43
224 - 239	0.41	0.38	0.40	0.42	0.43	0.49	0.52	0.55	0.52	0.55	0.47	0.38	0.34	0.29	0.25	0.19
240 - 255	0.16	0.17	0.15	0.19	0.19	0.15	0.13	0.11	0.10	0.13	0.13	0.11	0.05	0.01	0.00	0.12

Tabelle 16.2: Numerische Ausgabe der relativen Häufigkeiten in % zu Testbild 16.26-a

16.11.2 Histogramme

Mittelwert und mittlere quadratische Abweichung sind einfache Maßzahlen zur Charakterisierung der Verteilung der Grauwerte eines Bildes $\mathbf{S} = (s(x,y))$. Mehr Aufschluss über die Verteilung der Grauwerte gibt das *Histogramm der relativen Häufigkeiten* von $\mathbf{S}$:

$$p_{\mathbf{S}}(g) = \frac{a_g}{M}, \quad g = 0, 1, ..., 255. \tag{16.57}$$

Dabei sind g die Grauwerte der Grauwertmenge G und a_g die Häufigkeit des Auftretens des Grauwertes g in $\mathbf{S}$. Da das Histogramm über die Anzahl M der Bildpunkte von $\mathbf{S}$ normiert ist, gilt:

$$\sum_{g=0}^{255} p_{\mathbf{S}}(g) = 1, \text{ also auch } 0 \leq p_{\mathbf{S}}(g) \leq 1. \tag{16.58}$$

Bild 16.26-b zeigt das Histogramm von Bild 16.26-a. Auf der Abszisse werden die Grauwerte g und auf der Ordinate die dazugehörigen relativen Häufigkeiten $p_{\mathbf{S}}(g)$ in % aufgetragen. Tabelle 16.2 enthält das Histogramm in tabellarischer Form.

Der Mittelwert und die mittlere quadratische Abweichung können leicht aus dem Histogramm berechnet werden:

$$m_{\mathbf{S}} = \frac{1}{M} \sum_{g=0}^{255} gMp_{\mathbf{S}}(g) = \sum_{g=0}^{255} gp_{\mathbf{S}}(g) \tag{16.59}$$

und

$$q_{\mathbf{S}} = \frac{1}{M}\sum_{g=0}^{255}(g - m_{\mathbf{S}})^2 M p_{\mathbf{S}}(g) = \sum_{g=0}^{255}(g - m_{\mathbf{S}})^2 p_{\mathbf{S}}(g). \tag{16.60}$$

Zur Erläuterung der Bedeutung des Histogramms eines Bildes werden im Folgenden einige Beispiele gegeben. Für ein homogenes Bild $\mathbf{S} = (s(x,y)) = (g_k)$ ergibt sich trivialerweise ein Histogramm

$$p_{\mathbf{S}}(g) = \begin{cases} 1 & \text{falls } g = g_k, \\ 0 & \text{sonst.} \end{cases} \tag{16.61}$$

Ein Zweipegelbild $\mathbf{S} = (s(x,y))$, in dem nur die beiden Grauwerte $s(x,y) = g_k$ oder $s(x,y) = g_l$ auftreten, hat folgendes Histogramm:

$$p_{\mathbf{S}}(g) = \begin{cases} p_{\mathbf{S}}(g_k) & \text{falls } g = g_k, \\ p_{\mathbf{S}}(g_l) = 1 - p_{\mathbf{S}}(g_k) & \text{falls } g = g_l, \\ 0 & \text{sonst.} \end{cases} \tag{16.62}$$

Diese Form des Histogramms haben auch Binärbilder, die ein Sonderfall eines Zweipegelbildes sind.

Bei einem dunklen Bild mit wenig Kontrast sind vor allem die relativen Häufigkeiten $p_{\mathbf{S}}(g)$ für kleine Werte von g besetzt, während bei einem hellen Bild mit wenig Kontrast vor allem die großen Werte von g hohe relative Häufigkeiten aufweisen. Das Histogramm eines kontrastreichen Bildes zeichnet sich dadurch aus, dass im Idealfall alle relativen Häufigkeiten $p_{\mathbf{S}}(g) = 1/256$ sind. Das bedeutet, dass alle Grauwerte gleich oft auftreten. Ein Bild, das vorwiegend einen dunklen und einen hellen Bereich enthält, erzeugt ein Histogramm mit zwei lokalen Maxima. Histogramme dieser Art heißen *bimodal*. Abschließend sei noch darauf hingewiesen, dass aus dem Histogramm nicht auf die örtliche Anordnung der Grauwerte in der Bildmatrix $\mathbf{S}$ geschlossen werden kann. Ein Bild mit zwei deutlich abgegrenzten dunklen und hellen Bildbereichen kann dasselbe bimodale Histogramm erzeugen wie ein Bild, in dem die gleichen Grauwerte zufällig verteilt sind. Werden die relativen Häufigkeiten $p_{\mathbf{S}}(g)$ aufsummiert, so erhält man die *relativen Summenhäufigkeiten*:

$$h_{\mathbf{S}}(g) = \sum_{k=0}^{g} p_{\mathbf{S}}(k), \quad g = 0, 1, ..., 255. \tag{16.63}$$

Wegen

$$\sum_{g=0}^{255} p_{\mathbf{S}}(g) = 1 \text{ gilt: } 0 \le h_{\mathbf{S}}(g) \le 1. \tag{16.64}$$

Die relative Summenhäufigkeit wird z.B. bei der Modifikation des Histogramms zur Verbesserung des Kontrastes verwendet (Kapitel 17).

Die Berechnung des Histogramms ist nicht auf einkanalige Grauwertbilder beschränkt. In der multivariaten Klassifizierung (Kapitel 31) spielen N-dimensionale Histogramme eine wichtige Rolle. Das N-dimensionale Histogramm des N-kanaligen Bildes $\mathbf{S} = (s(x, y, n)), n = 0, 1, ..., N-1$ ist definiert als

$$p_{\mathbf{S}}(g_0, g_1, ..., g_{N-1}) = \frac{a_{g_0 g_1 \cdots g_{N-1}}}{M}, \tag{16.65}$$

wobei $a_{g_0, g_1, ..., g_{N-1}}$ die Häufigkeit der Grauwertkombination mit dem Grauwert g_n im Kanal n ist und M die Anzahl der Bildpunkte.

Im zweidimensionalen Fall kann das Histogramm bildlich dargestellt werden: Die relative Häufigkeit $p_{\mathbf{S}}(g_0, g_1)$ wird als Grauwert codiert in ein Bild $\mathbf{S}_a$ der Größe $256 \cdot 256$ Bildpunkte in der Position $(x, y) = (g_0, g_1)$ aufgetragen, also:

$$s_a(x, y) = s_a(g_0, g_1) = c \cdot p_{\mathbf{S}}(g_0, g_1), \tag{16.66}$$

wobei c ein Skalierungsfaktor ist. Bild 16.27 ist ein Beispiel einer Darstellung dieser Art. Aus der speziellen Form läßt sich ablesen, dass die Farbkanäle stark korreliert sind.

Aus dem N-dimensionalen Histogramm können die eindimensionalen Histogramme der N Kanäle abgeleitet werden. Im zweidimensionalen Fall $p_{\mathbf{S}}(g_0, g_1)$ gilt z.B.:

$$p_{\mathbf{S},\mathbf{0}}(g) = \sum_{k=0}^{255} p_{\mathbf{S}}(g, k) \tag{16.67}$$

und

$$p_{\mathbf{S},\mathbf{1}}(g) = \sum_{l=0}^{255} p_{\mathbf{S}}(l, g), \quad g = 0, 1, ..., 255. \tag{16.68}$$

Die Umkehrung gilt natürlich nicht: Aus den N eindimensionalen Histogrammen kann das N-dimensionale Histogramm nicht abgeleitet werden.

16.11.3 Entropie

Bei Fragestellungen, wie man die Datenmenge eines digitalisierten Bildes reduzieren kann, benötigt man ein Maß für den mittleren Informationsgehalt eines Bildes. Eine Maßzahl dafür ist die *Entropie*. Sie berechnet sich für ein einkanaliges Grauwertbild $\mathbf{S} = (s(x, y)), G = \{0, 1, ..., 255\}$ mit dem Histogramm $p_{\mathbf{S}}(g)$ zu

$$H = -\sum_{g=0}^{255} \Big(p_{\mathbf{S}}(g) \cdot log_2 p_{\mathbf{S}}(g) \Big). \tag{16.69}$$

Die Entropie kann auch als ein Maß für die mittlere apriori-Unsicherheit pro Bildpunkt oder die gemittelte Anzahl der notwendigen Bit pro Bildpunkt interpretiert werden. Ein Bild mit der Entropie H, bei dem die Grauwerte der Bildpunkte gleichverteilt sind,

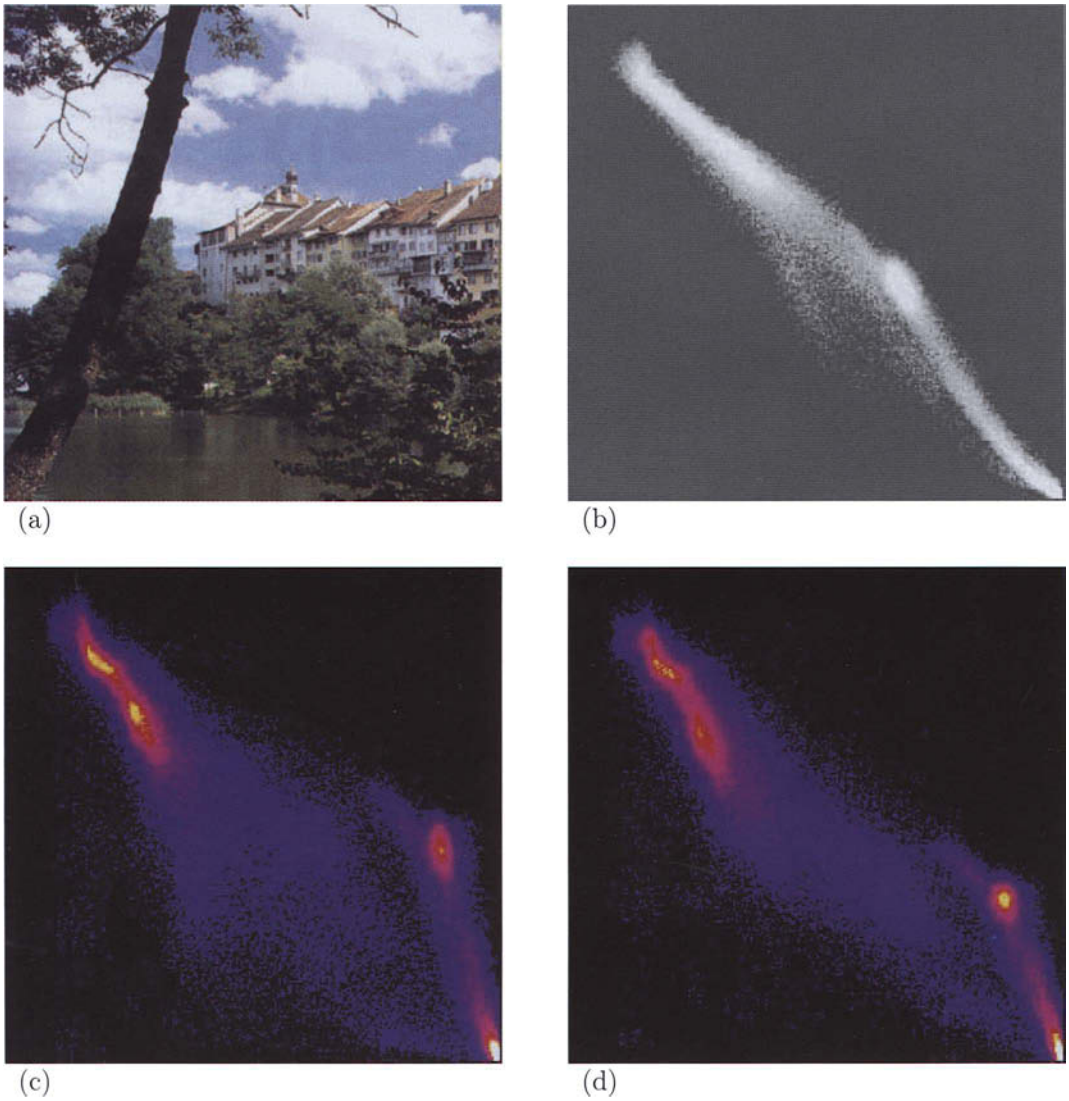


Bild 16.27: Mehrdimensionale Histogramme. (a) Farbtestbild (RGB-Bild). (b) Bildliche Darstellung des zweidimensionalen Histogramms des Rot- und des Grünauszugs. Nach unten sind die Grauwerte des ersten Kanals (Rotauszug) und nach rechts die Grauwerte des zweiten Kanals (Grünauszug) aufgetragen. Die relativen Häufigkeiten werden als Grauwert codiert. Die spezielle Form zeigt, dass die beiden Kanäle korreliert sind. (c) Zweidimensionales Histogramm: Rot-/Blauauszug. (d) Zweidimensionales Histogramm: Grün-/Blauauszug.

kann ohne Informationsverlust nicht mit weniger als H Bit pro Bildpunkt codiert werden. In der Praxis kann diese Annahme allerdings selten vorausgesetzt werden. Durch den Bildinhalt werden Korrelationen zwischen benachbarten Bildpunkten hergestellt, die nicht vernachlässigbar sind. So ist die Entropie hier nur eine Abschätzung für den extremen Fall.

Dazu noch einige Beispiele. Ein homogenes Bild $\mathbf{S} = (s(x,y)) = g$ hat die Entropie $H = 0$, während sich für ein Zweipegelbild mit $p_{\mathbf{S}}(g_1) = 0.5$ und $p_{\mathbf{S}}(g_2) = 0.5$ die Entropie $H = 1$ berechnet. Ein Bild, in dem alle Grauwerte mit derselben relativen Häufigkeit $p_{\mathbf{S}}(g) = 1/256$ auftreten, hat die Entropie $H = 8$. Das bedeutet, dass die Grauwerte der Bildpunkte mit 8 Bit codiert werden müssen. Die Entropie von Bild 16.26-a ist $H = 7.52$.

Ein aus der Entropie abgeleitetes Maß für die Symmetrie des Histogramms ist der *Anisotropiekoeffizient*:

$$\alpha = \frac{-\sum_{g=0}^{k} \left(p_{\mathbf{S}}(g) \cdot log_2 p_{\mathbf{S}}(g)\right)}{H}. \tag{16.70}$$

Hier ist k der kleinste mögliche Grauwert aus G, für den gilt:

$$\sum_{g=0}^{k} p_{\mathbf{S}}(g) \geq 0.5. \tag{16.71}$$

Symmetrische Histogramme haben ein α von 0.5. Die Abweichung von diesem Wert ist ein Hinweis auf zunehmende Asymmetrie. Der Anisotropiekoeffizient kann z.B. bei der Untersuchung, ob ein Histogramm bimodal ist oder nicht, verwendet werden.

16.11.4 Grauwertematrix (*co-occurrence*-Matrix)

Ein weiteres wichtiges Hilfsmittel zur Beschreibung von Bildeigenschaften ist die *Grauwertematrix* (*Grauwertübergangsmatrix*, *co-occurrence*-Matrix). Sie darf nicht verwechselt werden mit der Bildmatrix $\mathbf{S} = (s(x,y))$ eines digitalisierten Bildes.

Zur Definition der Grauwertematrix wird zunächst eine Relation zwischen Paaren von Bildpunktpositionen (x_1, y_1) und (x_2, y_2) festgelegt.

Beispiele:

(a) $(x_1, y_1)\rho_1(x_2, y_2)$ falls (x_2, y_2) der rechte Nachbar von (x_1, y_1) ist.

(b) $(x_1, y_1)\rho_2(x_2, y_2)$ falls gilt: $(x_2, y_2) = (x_1 + \Delta x, y_1 + \Delta y)$. (x_2, y_2) ist hier der Nachbar von (x_1, y_1) bezüglich des Verschiebungsvektors $(\Delta x, \Delta y)$.

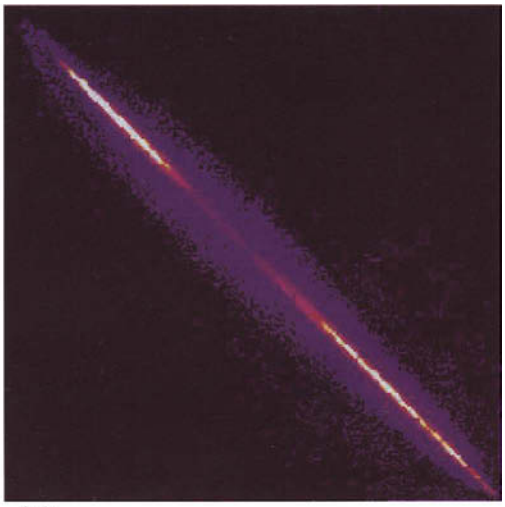
Die Grauwertematrix von $\mathbf{S} = (s(x,y)), G = \{0, 1, 2, ..., 255\}$ bezüglich der Relation ist dann die Matrix

$$\mathbf{W}_{\mathbf{S},\rho}(g_1, g_2) = \left(a_{g_1,g_2}\right), \tag{16.72}$$

wobei a_{g_1,g_2} die Häufigkeit der Grauwertkombination $(g_1, g_2) = \left(s(x_1, y_1), s(x_2, y_2)\right)$ bezüglich der Relation ρ ist.

Dazu ein Beispiel. Das Bild $\mathbf{S}$ mit $G = \{0, 1, 2, 3\}$ sei gegeben durch

(a)



(b)

Bild 16.28: (a) Testbild mit 256 Graustufen. (b) Bildliche Darstellung der Grauwertematrix (*co-occurrence*-Matrix) des Testbildes bezüglich der Relation „rechter Nachbar".

0	0	1	1	2	3
0	0	0	1	2	3
0	0	1	2	3	3
0	1	1	2	3	3
1	2	2	3	3	3
2	2	3	3	3	3

Als Relation wird $\rho = \rho_1$ („rechter Nachbar") verwendet. Es ergibt sich dann die Grauwertematrix

$$\mathbf{W}_{\mathbf{S},\rho}(g_1, g_2) = \begin{pmatrix} 4 & 4 & 0 & 0 \\ 0 & 2 & 5 & 0 \\ 0 & 0 & 2 & 6 \\ 0 & 0 & 0 & 7 \end{pmatrix}.$$

Die Grauwertematrix wird vor allem bei der Bildsegmentierung mit der Oberflächenstruktur (Textur) als Merkmal verwendet. Die Elemente der Hauptdiagonalen der Grauwertematrix repräsentieren die ungefähre Größe der homogenen Bildbereiche. Das Element in der Position $(g_1, g_2), g_1 \neq g_2$, gibt die ungefähre Länge der Grenze zwischen den Bereichen mit dem Grauwert g_1 und dem Grauwert g_2 wieder. Bei einem Bild mit wenig Kontrast wird der Bereich um die Hauptdiagonale der Grauwertematrix stark besetzt sein, während bei einem Bild mit viel Kontrast die linke untere Ecke und die rechte obere Ecke stark

besetzt sein werden. Bild 16.28-b zeigt eine bildliche Darstellung der Grauwertematrix zu Bild 16.28-a (Relation ρ: „rechter Nachbar“).

Kapitel 17

Modifikation der Grauwerte

In diesem und allen folgenden Kapiteln werden Verfahren erläutert, die letztlich auf eine, dem jeweiligen Anwendungsfall angepasste Interpretation des Bildinhaltes hinauslaufen. Die Bildinterpretation kann visuell durch einen Bearbeiter erfolgen. Dann werden Verarbeitungstechniken eingesetzt, die Bildinformationen, die bei einer speziellen Anwendung redundant oder sogar störend sind, unterdrücken und dafür wichtige Bildinhalte deutlicher hervorheben. Andererseits können Bilder auch automatisch oder zumindest teilautomatisch interpretiert werden. Auf Verfahren dieser Art wird ab Kapitel 23 näher eingegangen. Zunächst werden einfache Methoden der Bildverbesserung durch Analyse und Modifikation der Grauwertverteilung eines Bildes untersucht. Diese Verfahren dienen der besseren visuellen Interpretierbarkeit. Sie können aber auch Vorverarbeitungsschritte für nachfolgende Bildsegmentierung und Bildinterpretation sein (ab Kapitel 23).

17.1 Anwendungen

Häufig werden Bilder mit Hilfe eines Videodigitalisierers eingezogen und auf dem Monitor des Bildverarbeitungssystems dargestellt. Durch eine interaktive Veränderung der Grauwerte kann hier oft eine bessere bildliche Reproduktion erzielt werden. Die Grauwertmodifikation kann auch über im System gespeicherte *look-up*-Tabellen, die für bestimmte Anwendungsfälle vorgesehen sind, erfolgen. Auch eine dynamische Veränderung der dargestellten Grauwerte kann mit entsprechend schnellen Bildverarbeitungssystemen in Videoechtzeit durchgeführt werden.

Ein anderer Problemkreis ist die Ausgabe von Grauwertbildern über einfache Schwarz-/Weißdrucker. Es zeigt sich oft, dass die Darstellung auf dem Monitor nicht schlecht aussieht, jedoch nach dem Ausdrucken ein ziemlich kontrastarmes Bild vorliegt. Das liegt an der geringeren Anzahl der Graustufen, die ausgedruckt werden können. Ein automatisches Verfahren, das dieses Problem in vielen Fällen löst, wird in diesem Kapitel beschrieben.

In dieses Anwendungsgebiet fällt auch noch die Pseudofarbdarstellung, bei der bestimmte Grauwerte eines einkanaligen Grauwertbildes auf dem Monitor farbig dargestellt werden und so dem Betrachter einprägsamer erscheinen.

Eine andere Fragestellung liegt vor, wenn eine Trennung von Objekten vom Hintergrund durchgeführt werden soll. Dieser Problemkreis fällt in den Bereich der Bildsegmentierung. Die Binarisierung und die Äquidensitenbildung sind einfache Verfahren dazu.

Schließlich werden noch unterschiedliche Ansätze zur Reduktion der Grauwertemenge beschrieben.

17.2 Grundlagen der Grauwerttransformation

Im Folgenden wird ein einkanaliges Grauwertbild $\mathbf{S}_e = (s_e(x, y))$ mit der Grauwertmenge $G = \{0, 1, ..., 255\}$ vorausgesetzt. Eine Grauwerttransformation ist eine Abbildung f der Grauwertmenge G. Für f ist im Prinzip jede Funktion geeignet. Man wird in der Regel fordern, dass gilt:

$$f : G \rightarrow G. \tag{17.1}$$

Dann muss f beschränkt sein:

$$\min\{f\} > -\infty; \quad \max\{f\} < +\infty. \tag{17.2}$$

Ist dies der Fall, so kann f zu f_n normiert werden, sodass (17.1) erfüllt ist:

$$f_n(g) = \frac{f(g) - \min\{f\}}{\max\{f\} - \min\{f\}} \cdot c. \tag{17.3}$$

Dabei ist c ein Skalierungsfaktor, der geeignet zu wählen ist (z.B. $c = 255$). Die grafische Darstellung von $f_n(g)$ bezeichnet man als die *Gradationskurve*.

Wird f_n auf das Grauwertbild $\mathbf{S}_e$ angewendet, so berechnet sich das Ergebnisbild $\mathbf{S}_a$ wie folgt:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a :$$
$$s_a(x, y) = f_n(s_e(x, y)),\ 0 \leq x \leq L - 1,\ 0 \leq y \leq R - 1. \tag{17.4}$$

An dieser Stelle ein Hinweis zur Implementierung: In der Praxis wird man die Transformation $\mathbf{S}_e \rightarrow \mathbf{S}_a$ nie gemäß (17.4) realisieren, sondern immer über eine *look-up*-Tabelle (*LuT*). Eine *look-up*-Tabelle ist eine Tabelle mit 256 Speicherplätzen, die für die Grauwertmenge $G = \{0, 1, ..., 255\}$ die Werte von $f_n(g)$, $g \epsilon G$, enthält.

Die Transformationsfunktionen $f(g)$ oder $f_n(g)$ können auch interaktiv definiert werden. Dazu wird bei einem Grafik- oder Rasterbildspeichersystem mit dem Cursor der Verlauf einer Gradationskurve gezeichnet. Nach Maßgabe der festgelegten Kurve wird die *look-up*-Tabelle besetzt. Bei dieser Vorgehensweise kann die Skalierung der Grauwerte ganz auf das vorliegende Bildmaterial abgestimmt werden.

Der Algorithmus zur Skalierung der Grauwerte eines Bildes wird folgendermaßen skizziert:

A17.1: Skalierung der Grauwerte.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild mit der Grauwertmenge $G = \{0, 1, ..., 255\}$ (Eingabebild).

◇ $\mathbf{S}_a = (s_a(x,y))$ ein einkanaliges Grauwertbild mit der Grauwertmenge $G = \{0, 1, ..., 255\}$ (Ausgabebild).

Algorithmus:

(a) Festlegen oder Berechnen der Parameter der Funktion f_n.

(b) Berechnung der *look-up*-Tabelle. Für alle Grauwerte $g \epsilon G$:
$LuT(g) = f_n(g)$;

(c) Für alle Bildpunkte des Bildes $\mathbf{S}_e = (s_e(x,y))$:
$s_a(x,y) = LuT(s_e(x,y))$;

Ende des Algorithmus

Wenn das Eingabebild $\mathbf{S}_e$ ein mehrkanaliges Bild ist, z.B. ein Farbbild, kann die dargestellte Grauwerttransformation für jeden Kanal durchgeführt werden. Meistens ist es dann sinnvoll, für jeden Kanal eine eigene *look-up*-Tabelle zu verwenden.

17.3 Lineare Skalierung

Durch die lineare Skalierung eines Bildes wird die Grauwertverteilung eines Originalbildes $\mathbf{S}_e$ durch eine lineare Abbildung transformiert:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a : s_a(x,y) = (s_e(x,y) + c_1)c_2 = c_2 s_e(x,y) + c_1 c_2. \tag{17.5}$$

Mit der Notation von Abschnitt 17.2 erhält man folgende Transformationsfunktion:

$$f(g) = (g + c_1)c_2 = c_2 g + c_1 c_2. \tag{17.6}$$

Man sieht sofort, dass sich für $c_1 = 0$ und $c_2 = 1$ die identische Abbildung ergibt. Das Histogramm von $\mathbf{S}_a$ berechnet sich aus dem Histogramm von $\mathbf{S}_e$ gemäß:

$$p_{\mathbf{S}_a}(g_a) = \frac{a_{g_a}}{M} = \frac{a_{(g_e - c_1)/c_2}}{M} \tag{17.7}$$

Ist $c_1 > 0$, so wird zu den Grauwerten von $\mathbf{S}_e$ eine Konstante addiert und das Bild wird insgesamt heller, während für $c_1 < 0$ das Bild dunkler wird. Das zu $\mathbf{S}_a$ gehörige Histogramm

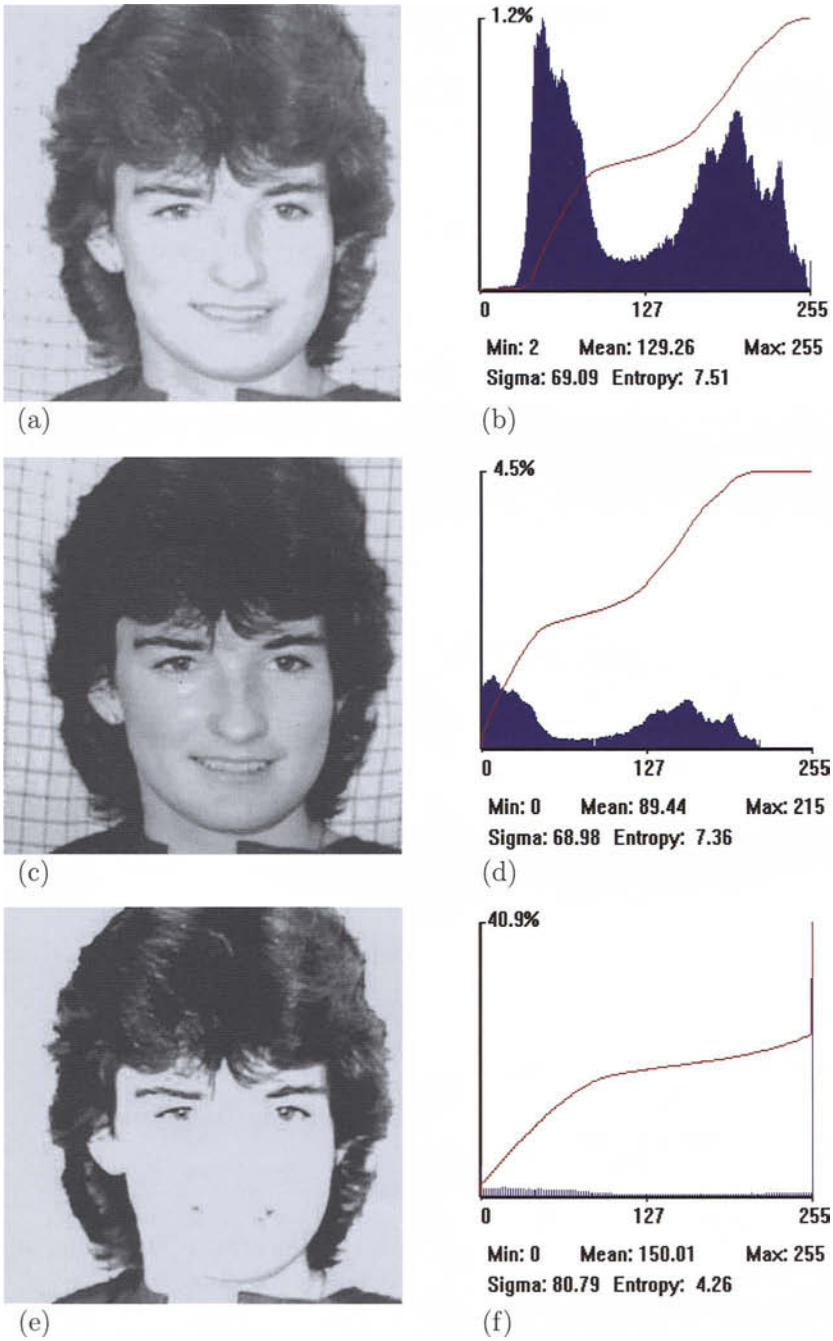


Bild 17.1: Beispiele zur linearen Skalierung eines Grauwertbildes. (a) Original. (b) Histogramm des Originals. (c) Verschiebung der Grauwerte mit $c_1 = -40$. (d) Histogramm nach der Verschiebung. Das veränderte Aussehen des Histogramms erklärt sich zum einen aus der Verschiebung der Grauwerte und zum anderen aus der unterschiedlichen Skalierung der Ordinate. Man sieht, dass Grauwerte auf der linken Seite des Histogramms „verloren gegangen" sind. (e) Kontrastanreicherung durch Multiplikation mit $c_2 = 2$. (f) Histogramm des skalierten Bildes. Auch hier wurden Grauwerte abgeschnitten.

ist gegenüber dem von $\mathbf{S}_e$ nach links ($c_1 < 0$) oder nach rechts ($c_1 > 0$) verschoben (Bild 17.1).

Die Konstante c_2 bewirkt eine Änderung des Kontrastes. Für $|c_2| > 1$ wird das Histogramm breiter und das Bild $\mathbf{S}_a$ kontrastreicher, wogegen für $|c_2| < 1$ das Histogramm schmäler und $\mathbf{S}_a$ kontrastärmer wird. Diese Sachverhalte können auch anhand des Mittelwertes und der mittleren quadratischen Abweichung verifiziert werden:

$$m_{\mathbf{S}_a} = \frac{1}{M}\sum_{x=0}^{L-1}\sum_{y=0}^{R-1} s_a(x,y) = \frac{1}{M}\sum_{x=0}^{L-1}\sum_{y=0}^{R-1}\Big(c_2 s_e(x,y) + c_1 c_2\Big) = c_2 m_{\mathbf{S}_e} + c_1 c_2. \tag{17.8}$$

$$\begin{aligned} q_{\mathbf{S}_a} &= \frac{1}{M}\sum_{x=0}^{L-1}\sum_{y=0}^{R-1}\Big(s_a(x,y) - m_{\mathbf{S}_a}\Big)^2 = \\ &= \frac{1}{M}\sum_{x=0}^{L-1}\sum_{y=0}^{R-1}\Big(c_2 s_e(x,y) + c_1 c_2 - (c_2 m_{\mathbf{S}_e} + c_1 c_2)\Big)^2 = \\ &= c_2^2 \frac{1}{M}\sum_{x=0}^{L-1}\sum_{y=0}^{R-1}\Big(s_e(x,y) - m_{\mathbf{S}_e}\Big)^2 = c_2^2 q_{\mathbf{S}_e}. \end{aligned} \tag{17.9}$$

Als Grauwertmenge bei Grauwertbildern wird $G = \{0, 1, ..., 255\}$ verwendet. Durch die Transformation (17.5) oder (17.6) kann diese Festlegung verletzt werden, da $f(g)$ nicht normiert ist. Es muss also die Transformation (17.5) so modifiziert werden, dass die Grauwerte von $\mathbf{S}_a$ immer in G liegen. Aus der linearen Skalierung wird dann eine *stückweise lineare Skalierung*:

$$\mathbf{S}_e \to \mathbf{S}_a : s_a(x,y) = \begin{cases} 0, & \text{falls } (s_e(x,y) + c_1)c_2 \le 0 \\ 255, & \text{falls } (s_e(x,y) + c_1)c_2 > 0 \\ (s_e(x,y) + c_1)c_2, & \text{sonst.} \end{cases} \tag{17.10}$$

Die normierte Skalierungsfunktion f_n ist hier stückweise linear. Sie kann somit wie folgt geschrieben werden:

$$f_n(g) = \begin{cases} 0, & \text{falls } c_2 g + c_1 c_2 \le 0, \\ 255, & \text{falls } c_2 g + c_1 c_2 > 255, \\ (g + c_1)c_2 = c_2 g + c_1 c_2, & \text{sonst.} \end{cases} \tag{17.11}$$

Die stückweise lineare Skalierung kann in $\mathbf{S}_a$ Informationsverlust bewirken, da alle Bildpunkte, deren Grauwerte durch die Transformation kleiner als 0 (oder größer als 255) würden, in $\mathbf{S}_a$ den Grauwert 0 (bzw. 255) erhalten und somit nicht mehr unterschieden werden können (siehe Bildfolge 17.1). Dieser Effekt muss aber nicht unbedingt nachteilig sein. Es gibt viele Beispiele, bei denen auf diese Weise nicht benötigte Bildinformation „ausgeblendet" wird.

Auf jeden Fall muss die Bestimmung der Parameter c_1 und c_2 sorgfältig durchgeführt werden. Eine Möglichkeit ist es, sie aus dem Histogramm zu ermitteln. Dazu werden der

minimale (min) und der maximale (max) Grauwert des Bildes $\mathbf{S}_e$ bestimmt. Die beiden Parameter berechnen sich dann gemäß:

$$c_1 = -min \text{ und } c_2 = \frac{255}{max - min}. \tag{17.12}$$

Bei dieser Transformation geht keine Bildinformation verloren, da alle Grauwerte von $\mathbf{S}_e$ im Intervall $[min,max]$ liegen und min in $\mathbf{S}_a$ auf den Wert 0 und max auf den Wert 255 abgebildet wird.

Es kann aber vorkommen, dass sich das Bild $\mathbf{S}_a$ gegenüber $\mathbf{S}_e$ kaum verändert hat, nämlich dann, wenn min und max zufällige Resultate eines leicht verrauschten Bildes sind, die keine Signifikanz für den eigentlichen Bildinhalt haben. Es ist z.B. möglich, dass durch Fehler bei der Digitalisierung die Grauwerte einiger Bildpunkte auf 0 oder 255 gesetzt werden. Dann wäre die Transformation (17.10) mit der Parameterberechnung (17.12) eine identische Abbildung. Aus diesem Grund ist es oft sinnvoll, die Skalierungsparameter in (17.12) mit zwei Werten $min' > min$ und $max' < max$ zu berechnen. Hier ist dann aber Vorsicht geboten, da durch ungeschickte Wahl Bildinformation verloren geht.

Bevor weitere Techniken zur Bestimmung der Parameter c_1 und c_2 untersucht werden noch ein Hinweis auf die Implementierung solcher Grauwertskalierungen. Um Rechenzeit zu sparen, wird die Implementierung nicht gemäß (17.10) durchgeführt, sondern wie in Abschnitt 17.2 dargestellt, über eine *look-up*-Tabelle, die einmal vor der eigentlichen Grauwerttransformation besetzt wird. Die Vorgehensweise dazu wurde in Algorithmus **A17.1** angegeben.

Bei einem Betriebssystem mit einer grafischen Oberfläche oder mit Hilfe eines Rasterbildspeichersystems, lässt sich einfach eine Skalierung der Daten im Bildfenster oder im Bildwiederholungsspeicher einrichten. Eine *look-up*-Tabelle wird anfangs mit der identischen Abbildung, also mit den Werten von 0 bis 255, vorbesetzt. Die Skalierungsparameter erhalten anfänglich die Werte $c_1 = 0$ und $c_2 = 1$. Der Cursor wird im Bildfenster in die Bildmitte gesetzt. Bei jeder Bewegung werden die Koordinaten des Cursors (x, y) ausgelesen. Die Skalierungsparameter werden dann wie in Algorithmus **A17.2** angegeben, aus der Position des Cursors berechnet.

Die *look-up*-Tabelle wird nach Maßgabe der neuen Skalierungsparameter neu besetzt. Horizontale Bewegungen des Lichtpunktes bewirken Veränderungen in der Helligkeit und vertikale Bewegungen Veränderungen im Kontrast.

Die Berechnung von c_1 und c_2 geht sehr schnell. Wenn das Lesen und Laden der *look-up*-Tabelle des Grafik- oder Rasterbildspeichersystems sehr schnell geht, kann die Skalierung interaktiv in Echtzeit durchgeführt werden, wobei die Originaldaten im Bildfenster erhalten bleiben. Es besteht mit dieser Methode somit die Möglichkeit, zu einem Bild interaktiv eine Helligkeits- und Kontrastmanipulation durchzuführen. Wenn die gewünschte Darstellung gefunden ist, wird die *look-up*-Tabelle übernommen und das Originalbild (z.B. zur Bildausgabe über einen Drucker) mit der *look-up*-Tabelle modifiziert. Diese Technik lässt sich auch bei Farbbildern einsetzen.

A17.2: Interaktive Berechnung von c_1 und c_2.

Voraussetzungen und Bemerkungen:

◇ Es wird angenommen, dass die *x_cursor_position* und die *y_cursor_position* zwischen 1 und 512 liegen.

Algorithmus:

(a) Den *cursor* des Bildverarbeitungssystems in die Bildmitte setzen.

(b) Solange kein Abbruch-Code (z.B. mit der rechten oder linken Maus-Taste) gegeben wird:

(ba) Aktuelle *cursor*-Position lesen.

(bb) Falls sich die *cursor*-Position geändert hat:

(bba) Berechnung der Parameter c_1 und c_2 etwa gemäß:

$c_2 = (512 - y\_cursor\_position)/y\_cursor\_position;$

$c_1 = 127 - c_2 \cdot x\_cursor\_position/2;$

(bbb) Berechnung der neuen *look-up*-Tabelle gemäß Algorithmus **A17.1** und Formel (17.11).

(bbc) Abspeichern der neuen *look-up*-Tabelle in der Hardware des Bildverarbeitungssystems.

Ende des Algorithmus

Die Skalierungsparameter können auch so bestimmt werden, dass das skalierte Bild $\mathbf{S}_a$ einen vorgegebenen Mittelwert und eine vorgegebene mittlere quadratische Abweichung hat. Es sei

$m_{\mathbf{S}_e}$ der Mittelwert des Originalbildes $\mathbf{S}_e$,

$q_{\mathbf{S}_e}$ die mittlere quadratische Abweichung von $\mathbf{S}_e$,

$m_{\mathbf{S}_a}$ der vorgegebene Mittelwert von $\mathbf{S}_a$,

$q_{\mathbf{S}_a}$ die vorgegebene mittlere quadratische Abweichung von $\mathbf{S}_a$.

Dann liefert die Transformation (17.10) mit den Skalierungsparametern

$$c_1 = \frac{m_{\mathbf{S}_a}\sqrt{q_{\mathbf{S}_e}}}{\sqrt{q_{\mathbf{S}_a}}} - m_{\mathbf{S}_e} \text{ und } c_2 = \frac{\sqrt{q_{\mathbf{S}_a}}}{\sqrt{q_{\mathbf{S}_e}}} \tag{17.13}$$

ein Bild $\mathbf{S}_a$ mit den vorgegebenen Werten für $m_{\mathbf{S}_a}$ und $q_{\mathbf{S}_a}$. Durch eine Transformation dieser Art können z.B. zwei Bilder desselben Objektes, die unter verschiedenen Aufnahmebedingungen erzeugt wurden, in der Helligkeit und im Kontrast angeglichen werden.

17.4 Äquidensiten (*gray level slicing*)

Eine Spezialisierung der allgemeinen Skalierung von Abschnitt 17.2 ist die Erzeugung eines *Äquidensitenbildes* $\mathbf{S}_a$ aus dem Original $\mathbf{S}_e$ (*gray level slicing*). Die normierte Skalierungsfunktion $f_n(g)$ wird dazu stückweise konstant gewählt:

$$f_n(g) = g_k \text{ für } l_k \leq g < l_{k+1} \text{ und } k = 0, 1, ... \tag{17.14}$$

wobei l_k, g_k und g aus dem Definitionsbereich $G = \{0, 1, ..., 255\}$ sind. Bild 17.2-a zeigt ein Beispiel einer Gradationskurve dieser Art, Bild 17.2-b die dazugehörige Äquidensitendarstellung des Testbildes (mit schwarzen Übergangsrändern) und Bild 17.2-c das Histogramm des Äquidensitenbildes.

Die Formel (17.15) ist ein weiteres Beispiel, wie die Skalierungsfunktion in einem Anwendungsfall aussehen könnte:

$$f_n(g) = \begin{cases} 0 & \text{falls } 0 \leq g = s_e(x,y) \leq 11, \\ 100 & \text{falls } 11 < g = s_e(x,y) \leq 111, \\ 200 & \text{falls } 111 < g = s_e(x,y) \leq 177, \\ 0 & \text{falls } 177 < g = s_e(x,y) \leq 255. \end{cases} \tag{17.15}$$

Hier werden alle Grauwerte des Originals $\mathbf{S}_e$, die z.B. zwischen 12 und 111 (eingeschlossen) liegen, im Ausgabebild $\mathbf{S}_a$ durch den Grauwert 100 dargestellt, Grauwerte, die zwischen 112 und 177 (eingeschlossen) liegen, durch den Grauwert 200. Die übrigen Grauwerte werden im Ausgabebild mit dem Grauwert 0 codiert. Das dazu gehörige Bildbeispiel ist die Bildfolge 17.3. Bild 17.3-a zeigt zwei unterschiedliche Stecker. Mit obiger Skalierungsfunktion f_n wurden die beiden „Objekte" ausgeblendet (Bild 17.3-c). Jeweils rechts daneben sind die Histogramme mit den relativen Summenhäufigkeiten. Der Bildhintergrund des Originals erfasst etwa 84% der gesamten Bildfläche. Bei Bildern, die mit stückweise konstanten Skalierungsfunktionen verarbeitet wurden, sind die relativen Summenhäufigkeiten stufenförmig (Bilder 17.2-c und 17.3-d).

Es ist auch möglich, bestimmte Grauwertbereiche unverändert aus dem Original zu übernehmen. So könnte man im obigen Beispiel den linken Stecker mit dem Grauwert 200 codieren, den Hintergrund mit 0 und den rechten Stecker mit seinen Originalgrauwerten aus dem Eingabebild $\mathbf{S}_e$:

$$f_n(g) = \begin{cases} 0 & \text{falls } 0 \leq g = s_e(x,y) \leq 11, \\ s_e(x,y) & \text{falls } 11 < g = s_e(x,y) \leq 111, \\ 200 & \text{falls } 111 < g = s_e(x,y) \leq 177, \\ 0 & \text{falls } 177 < g = s_e(x,y) \leq 255. \end{cases} \tag{17.16}$$

Die mit (17.15) erzeugten Äquidensiten heißen *Äquidensiten 1. Ordnung*. Bei *Äquidensiten 2. Ordnung* werden nur die Ränder der Flächen ausgegeben. Dazu werden zunächst die Äquidensiten 1. Ordnung berechnet. In einem zweiten Schritt werden die Differenzen zu den Nachbarn gebildet und mit einem Skalierungsfaktor ins Ausgabebild übertragen (Algorithmus **A17.3**).

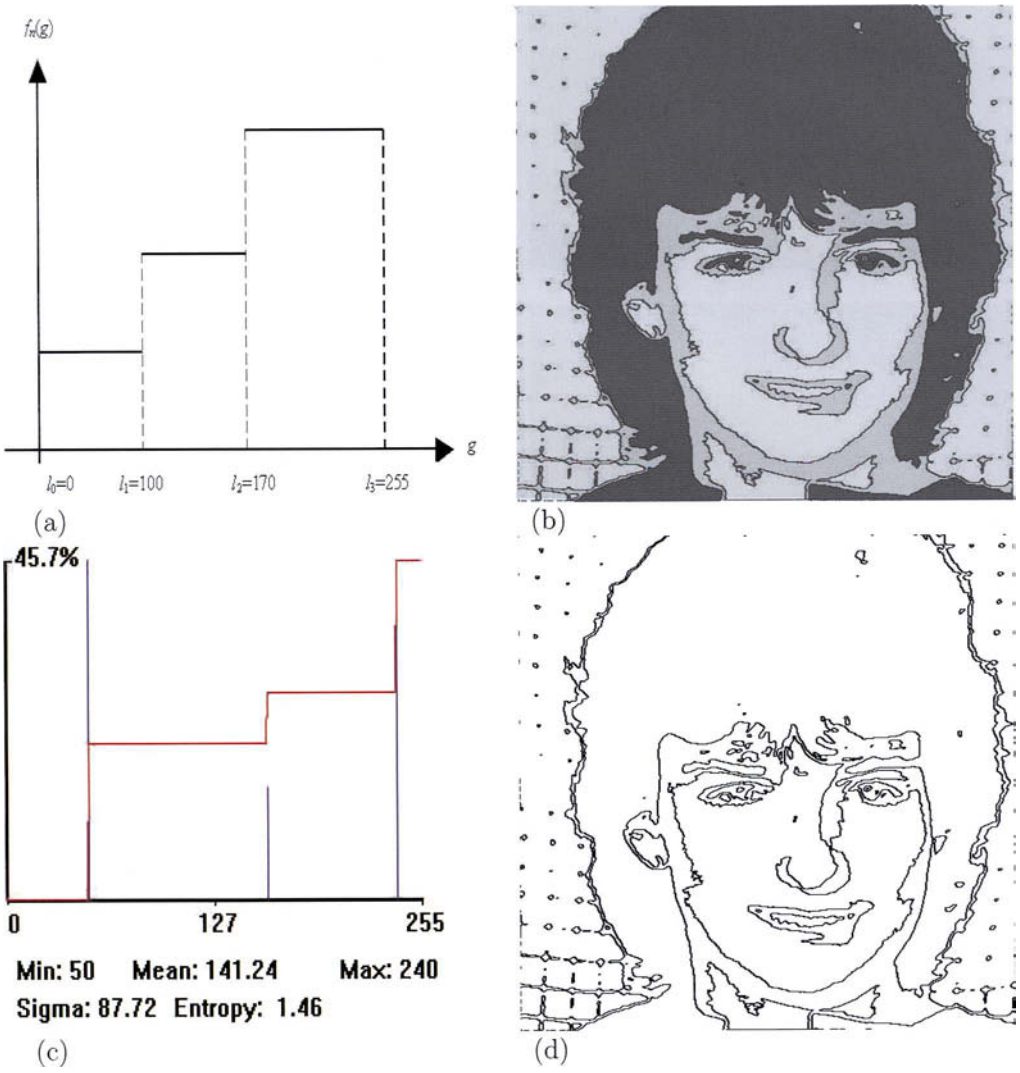
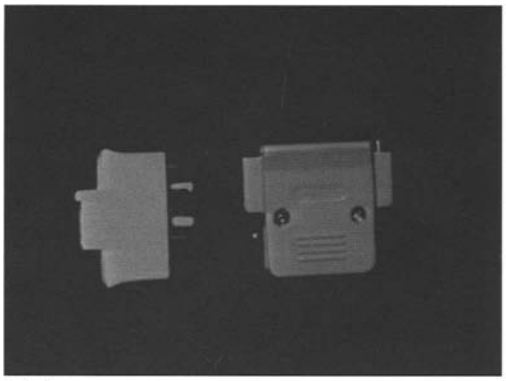
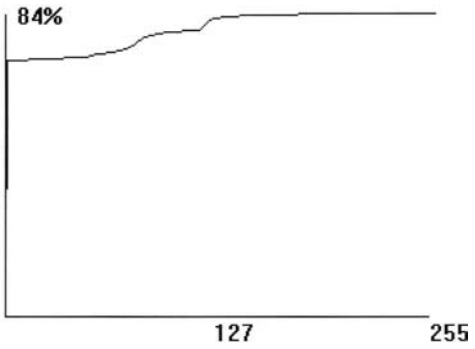


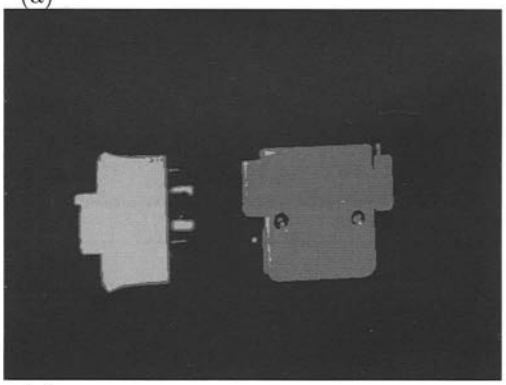
Bild 17.2: Beispiel zur Äquidensitendarstellung. (a) Stückweise konstante Skalierungsfunktion. (b) Äquidensitendarstellung des Testbildes (mit schwarzen Übergangsrändern). (c) Histogramm der Äquidensitendarstellung. (d) Äquidensiten 2. Ordnung.



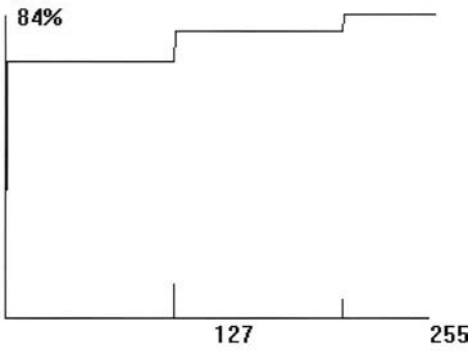
(a)



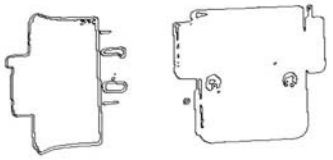
(b)



(c)



(d)



(e)

Bild 17.3: Äquidensitendarstellung. (a) Das Original zeigt zwei unterschiedliche Stecker. (b) Das Histogramm und die Summenhäufigkeiten des Originals. Es sind ca. 84% der Grauwerte sehr nahe bei null. (c) Die beiden „Objekte" wurden ausgeblendet. Da aber sowohl im rechten als auch im linken Stecker jeweils Grauwerte des anderen Steckers enthalten sind, ist eine eindeutige Trennung nicht möglich. (d) Histogramm mit Summenhäufigkeiten des Äquidensitenbildes. Bei Bildern, die mit stückweise linearen Skalierungsfunktionen verarbeitet wurden, sind die Summenhäufigkeiten stufenförmig. (e) Äquidensiten 2. Ordnung. Die doppelten Ränder im linken Stecker entstehen durch Grauwerte, mit denen der rechte Stecker codiert wurde.

A17.3: Äquidensiten 2. Ordnung.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild.

◇ $\mathbf{S}_a = (s_a(x,y))$ sei das Ausgabebild.

Algorithmus:

(a) Berechnung des Äquidensitenbildes $\mathbf{S}_{aeq}$.

(b) Für alle Pixel $s_{aeq}(x,y)$ von $\mathbf{S}_{aeq}$:

(ba) Bilde: $diff = 4s_{aeq}(x,y) - s_{aeq}(x,y-1) - s_{aeq}(x,y+1) - s_{aeq}(x-1,y) - s_{aeq}(x+1,y)$;

(bb) Falls $diff = 0 : s_a(x,y) = 255$;

(bc) Falls $diff \neq 0 : s_a(x,y) = 0$;

Ende des Algorithmus

Diese Technik kann verwendet werden, wenn auf einfache Art der Rand der „Objekte" ermittelt werden soll (Bild 17.3-e). Allerdings sind die Ergebnisse nur brauchbar, wenn die Grauwertintervalle deutlich voneinander getrennt liegen und im Hintergrund diese Grauwerte nicht auftreten. Von *Äquidensiten gemischter Ordnung* spricht man, wenn im Ausgabebild sowohl die grau codierten Flächen als auch deren Ränder dargestellt werden.

Wenn das Originalbild verrauscht ist, wird das Äquidensitenbild an den Rändern der Flächen mehr oder weniger stark ausgefranst. Dann ist es notwendig, vor der Äquidensitenbildung einen Glättungsoperator (Grundlagen dazu in Kapitel 18) anzuwenden. Sollen bei der Glättung die 8-Nachbarn eines Pixels $s(x,y)$ berücksichtigt werden, so kann der „bewegte Mittelwert" mit folgendem Filterkern verwendet werden:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{17.17}$$

Auch größere Nachbarschaften sind sinnvoll, allerdings ist dabei zu beachten, dass damit feine Bildstrukturen mehr und mehr verloren gehen.

Die Bilder 17.4-a bis 17.4-f zeigen ein Beispiel zu dieser Anwendung. Bild 17.4-a ist das Original, ein etwas verrauschtes, digitalisiertes Fernsehbild. Daneben ist das mit einem $5 \cdot 5$ bewegten Mittelwert verarbeitete Bild. Die Bilder 17.4-c und 17.4-d zeigen jeweils die Äquidensitendarstellung mit vier Grauwertintervallen. Man sieht, dass sich die homogenen Flächen im rechten Bild deutlicher ausprägen. Wenn nach der Äquidensitenbildung das Bild weiterverarbeitet werden soll, z.B. in eine Strichzeichnung für ein CAD-System, so ist das linke Bild 17.4-e unbrauchbar, während das rechte Bild 17.4-f als Urskizze in einem CAD-System weiter verwendet werden könnte.

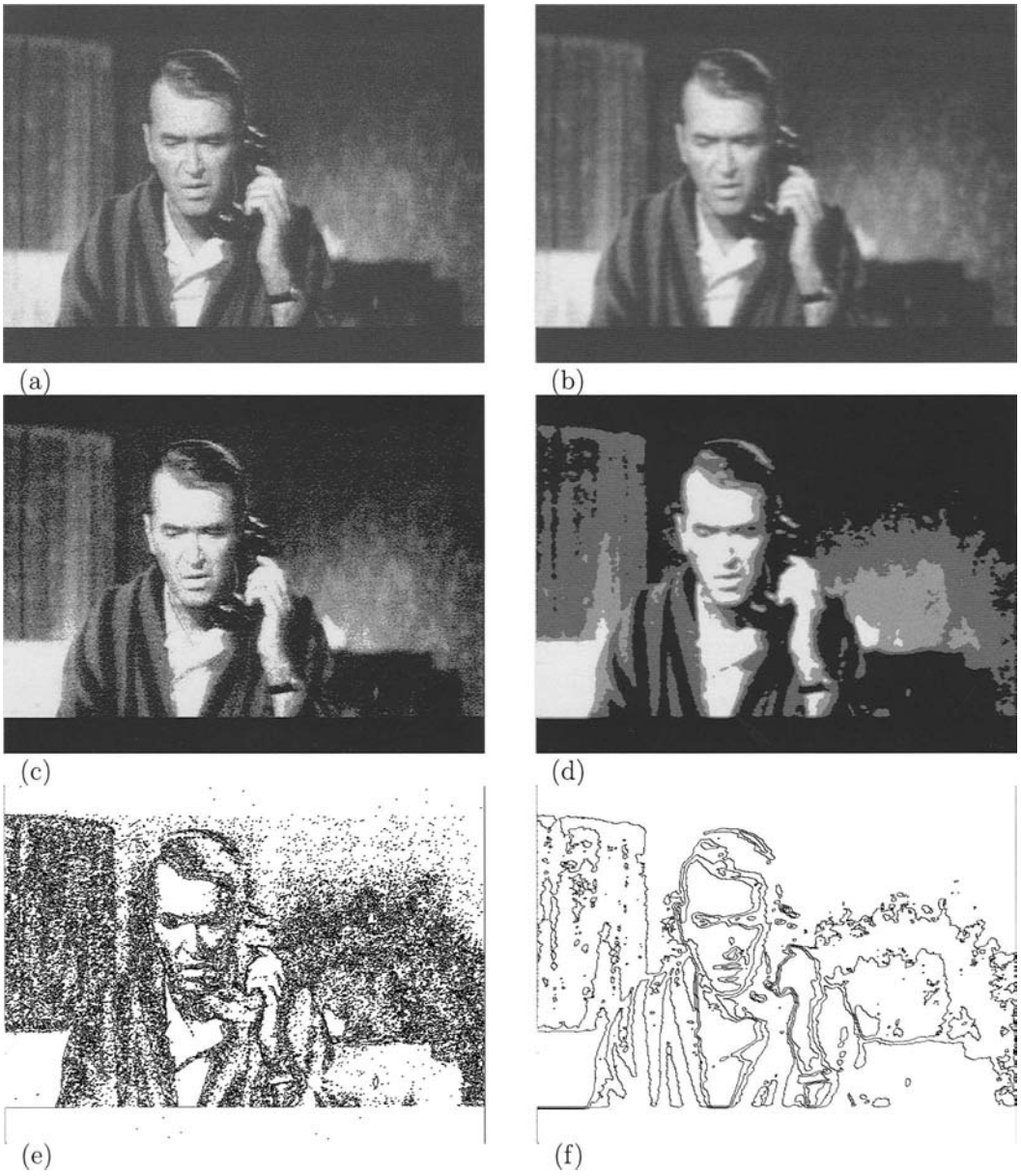
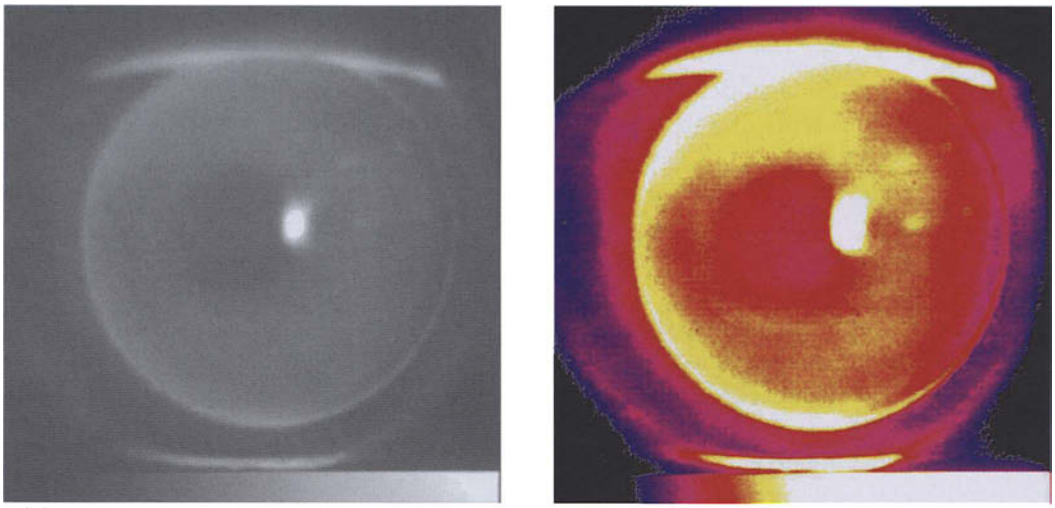


Bild 17.4: (a) Verrauschtes, digitalisiertes Fernsehbild. (b) Geglättetes Bild mit einer Umgebung von $5 \cdot 5$ Nachbarn. (c) Äquidensiten aus dem Original mit vier Graustufen. (d) Äquidensiten nach dem bewegten Mittelwert ($5 \cdot 5$). Man sieht hier, dass sich die homogenen Flächen wesentlich deutlicher ausprägen als im linken Bild. (e) Äquidensiten 2. Ordnung aus dem Original. (f) Äquidensiten 2. Ordnung aus dem geglätteten Bild. Dieses Bild wäre z.B. als Urskizze für ein CAD-System brauchbar.



(a) (b)

Bild 17.5: (a) Bei der Anpassung von Kontaktlinsen wird eine fluoreszierende Kontrastflüssigkeit verwendet. In der vorliegenden Anwendung wurden dem Linsenanpasser die Bilder zusätzlich in digitalisierter Form bewegungskompensiert auf einem Monitor angeboten. (b) Durch die dabei verwendete Pseudofarbdarstellung wurde die Verteilung der Kontrastflüssigkeit deutlich sichtbar gemacht.

Die *Pseudofarbdarstellung*, die schon in den vorhergehenden Abschnitten angesprochen wurde, ist eng mit der Äquidensitenbildung verwandt. Hier werden den Grauwerten oder Grauwertintervallen des Originalbildes Farben zugeordnet. Soll ein bestimmter Bildbereich dem Betrachter deutlich gemacht werden, so ist eine farbliche Absetzung in der Regel vorzuziehen. Das menschliche Auge kann nämlich nur etwa 30 unterschiedliche Grauabstufungen unterscheiden. Untersuchungen haben aber gezeigt, dass es bei der Unterscheidung von Farben wesentlich leistungfähiger ist (möglicherweise um mehrere Zehnerpotenzen).

Je nach Anwendungsfall können dem Betrachter durch die entsprechende Farbwahl zusätzlich bestimmte Eigenschaften wie z.B. „warm“ und „kalt“, „häufig“ und „selten“ oder „früher“ und „später“ suggeriert werden. Die Bilder 17.5-a und 17.5-b zeigen ein Beispiel dazu.

In manchen Anwendungsfällen entstehen einprägsame Äquidensitenbilder, wenn die Flächen nicht durch Grauwerte, sondern durch synthetische Muster wiedergegeben werden. So können z.B. bei einer Landnutzungskartierung aus Luft- oder Satellitenbilddaten die verschiedenen Landnutzungsflächen durch Symbole markiert werden, die sich an der Darstellung in topografischen Karten orientieren.

Ein Spezialfall der Äquidensitenbildung ist die Erzeugung eines Zweipegel- oder Binärbildes. Diese Techniken werden im nächsten Abschnitt behandelt.

17.5 Erzeugen von Binärbildern

Die Erzeugung eines Binär- oder Zweipegelbildes ist ein Sonderfall der stückweise linearen Skalierung (Äquidensitenbildung). Die Skalierungsfunktion sieht hier folgendermaßen aus:

$$f_n(g) = \begin{cases} g_0, & 0 = l_0 \leq g = s_e(x,y) < l_1, \\ g_1, & l_1 \leq g < l_2, \\ g_2 = g_0, & l_2 \leq g \leq l_3 = 255. \end{cases} \tag{17.18}$$

Bei einer reinen *Binarisierung* wird ein Schwellwert c vorgegeben. Alle Grauwerte des Bildes $\mathbf{S}_e$, die kleiner als der Schwellwert sind, werden im Ausgabebild $\mathbf{S}_a$ auf den Grauwert $g_0 = 0$ gelegt, alle anderen auf $g_1 = 1$:

$$f_n(g) = \begin{cases} g_0 = 0 & \text{falls } g = s_e(x,y) \leq c, \\ g_1 = 1 & \text{sonst.} \end{cases} \tag{17.19}$$

Bei einer anderen Variante bleiben die Grauwerte bestimmter Bildbereiche in $\mathbf{S}_a$ erhalten, während die Grauwerte anderer Bildbereiche durch einen neuen Grauwert dargestellt werden. Eine Anwendung davon ist z.B. die Trennung eines abgebildeten Objektes vom Hintergrund. Dies ist aber nur möglich, wenn sich die Grauwerte von Hintergrund und Objekt nicht überlappen. Die Skalierungsfunktion könnte dabei festgelegt sein gemäß:

$$f_n(g) = \begin{cases} g_0 = 0, & 0 \leq g = s_e(x,y) < l_1, \\ g_1 = g, & l_1 \leq g \leq 255. \end{cases} \tag{17.20}$$

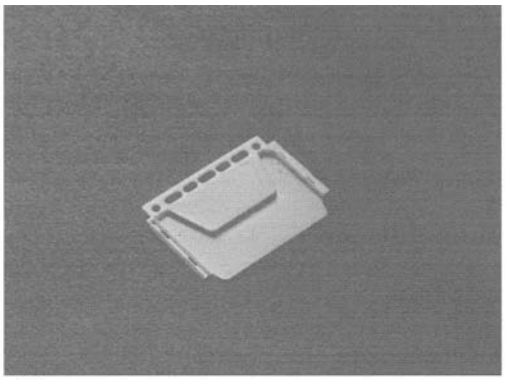
Die Grauwerte von 0 bis $l_1 - 1$ werden durch diese Skalierungsfunktion auf den Grauwert $g_0 = 0$ abgebildet, die Grauwerte von l_1 bis 255 bleiben erhalten. Verarbeitungen dieser Art fallen in den Bereich einfacher Bildsegmentierungen.

Werden anstatt der Grauwerte $g_0 = 0$ und $g_1 = 1$ beliebige Grauwerte g_0 und g_1 verwendet, so spricht man von einem *Zweipegelbild* (Kapitel 16). Die Darstellung als Zweipegelbild (etwa mit $g_0 = 0$ und $g_1 = 255$) ist dann vorzuziehen, wenn das binarisierte Bild auf einem Displaysystem dargestellt werden soll, da die Grauwerte 0 und 1 nicht zu unterscheiden sind.

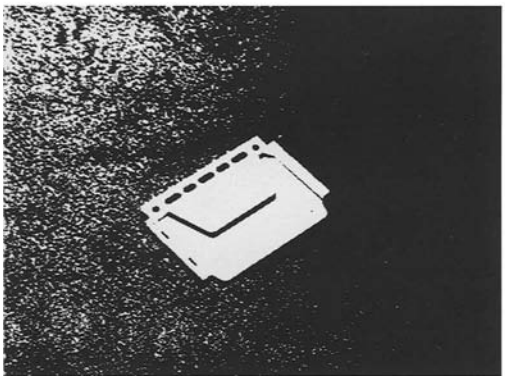
Die Binarisierung eines Grauwertbildes wird in der Praxis meistens dazu verwendet, um abgebildete Objekte vom Hintergrund zu trennen. Deshalb kann sie auch als ein einfaches Verfahren zur *Segmentierung* angesehen werden. Im Folgenden werden einige einfache Verfahren der Bildsegmentierung durch Binärbilderzeugung vorgestellt.

Wie bei der Äquidensitentechnik bilden sich die Flächen homogener aus, wenn ein leicht verrauschtes Bild zunächst mit einem Glättungsoperator behandelt wird.

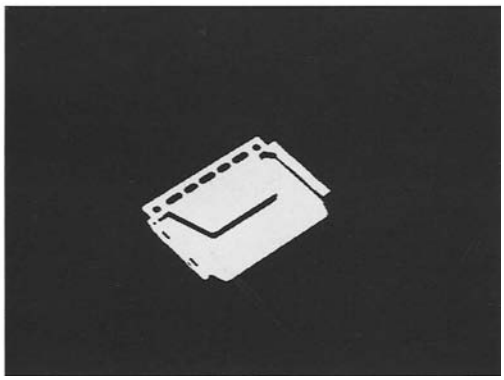
In Formel (17.19) wird für das gesamte Bild ein Schwellwert c verwendet. Man bezeichnet diese Art deshalb auch als *Binarisierung mit fixem Schwellwert*. Der Schwellwert c kann mit unterschiedlichen Verfahren festgelegt werden. Zunächst kann er natürlich aus dem Histogramm von $\mathbf{S}_e$ abgelesen oder automatisch ermittelt werden.



(a)



(b)



(c)

Bild 17.6: (a) Original. (b) Als Schwellwert wurde der Mittelwert m des Bildes verwendet, der in diesem Fall ungeeignet ist, da er das Rauschen im Bild segmentiert. (c) Hier wurde die Schwelle etwas höher gelegt.

Bei deutlich bimodalen Histogrammen ist das Minimum zwischen den beiden Maxima sicher ein geeigneter Schwellwert. Auch der mittlere Grauwert m des gesamten Bildes kann sich als Schwellwert eignen, wenn die zu trennenden hellen und dunklen Bildbereiche etwa gleich groß sind. Ist das nicht der Fall, so kann der Mittelwert m als Schwellwert c auch ungeeignet sein, da er z.B. bei einem leicht verrauschten Bild das Rauschen segmentiert (Bild 17.6).

Viele Bildverarbeitungssysteme bieten die Möglichkeit, den Schwellwert c auch interaktiv, z.B. mit einer Rollkugel oder einer Maus festzulegen. Verfahrenstechnisch wird hier ähnlich vorgegangen wie in Algorithmus **A17.2**. Hier kann das Ergebnis sofort auf dem Monitor des Displaysystems kontrolliert werden.

Soll ein Objekt segmentiert werden, dessen Größe ungefähr bekannt ist, so kann die *p%-Methode* verwendet werden. Es wird vorausgesetzt, dass das zu segmentierende Objekt etwa $p\%$ des gesamten Bildbereichs ausmacht. Man wird bei der Binarisierung den Schwellwert c so wählen, dass er $p\%$ „Objekt-Bildpunkte" und $(100-p)\%$ „Hintergrundbildpunkte" erzeugt.

A17.4: Binärbilderzeugung mit der p%-Methode.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild.

Algorithmus:

(a) Ermittle den Schwellwert c so, dass gilt:

$$\sum_{k=0}^{c} p(k) \leq 1 - \frac{p\%}{100};$$

(b) Binarisiere $\mathbf{S}_e$ mit dem Schwellwert c.

Ende des Algorithmus

Diese Verfahren sind aber nur dann geeignet, wenn der Hintergrund im gesamten Bild im gleichen Grauwertbereich liegt. Dies ist in der Praxis häufig nicht der Fall, z.B. bei ungleichmäßiger Ausleuchtung bei der Bildaufzeichnung. Die Grenze der Anwendbarkeit der einfachen Schwellwertverfahren ist auch erreicht, wenn die Grauwerte des Bildhintergrundes auch im Objekt auftreten. Dazu zeigt Bild 17.7-a ein Beispiel: Durch die unterschiedliche Schwärzung gehen die Papillaren im rechten Bildteil weitgehend verloren.

Eine Möglichkeit sind dann Verfahren mit *dynamischem Schwellwert*. Der Schwellwert c wird hier (im extremen Fall) für jeden Bildpunkt $s(x,y)$ aus einer Umgebung $U(s(x,y))$ (z.B. 3·3, 5·5, usw.) berechnet. Wie groß die Umgebung zu bemessen ist, muss im jeweiligen Anwendungsfall aus dem Bildmaterial festgelegt werden. Für die gewählte Umgebung wird das Histogramm berechnet und daraus der Schwellwert c abgeleitet. Zur Berechnung von c kann ohne weitere Prüfung der Mittelwert $m_{U(s(x,y))}$ der Umgebung verwendet werden, wenn sichergestellt ist, dass in jeder Umgebung $U(s(x,y))$ beide „Modi" (Hintergrund und zu segmentierendes Objekt) auftreten. Für eine $m \cdot m$-Umgebung sieht die Berechnung des Schwellwertes c folgendermaßen aus:

$$c = \frac{1}{m^2} \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} s(x+k-i, y+k-j), k = \frac{m-1}{2} \tag{17.21}$$

Die Schwellwertberechnung muss nicht unbedingt für jeden Bildpunkt durchgeführt werden. Wenn sich der Hintergrund nur langsam ändert, so kann das Fenster, das zur

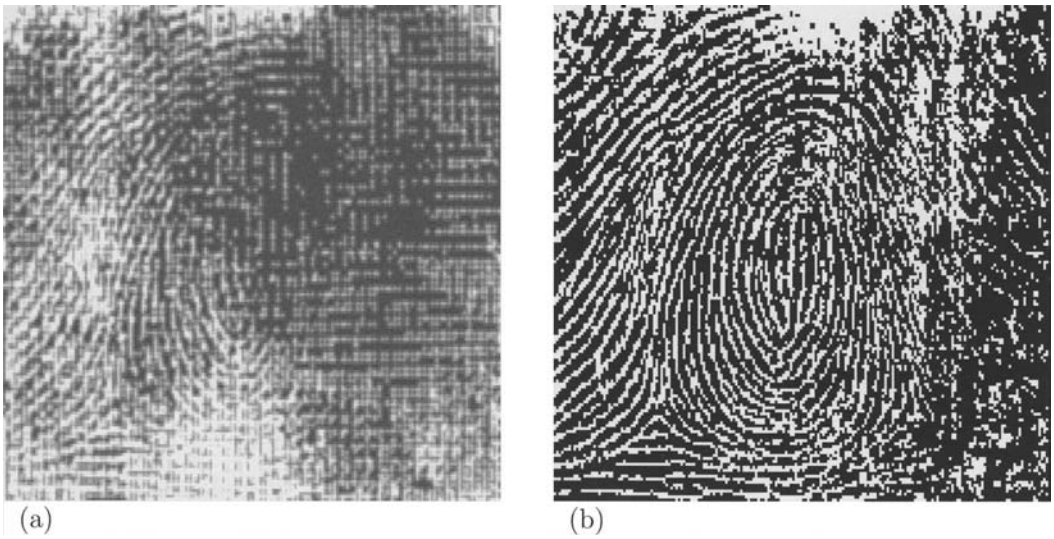


Bild 17.7: Beispiel zur Binärbilderzeugung mit dynamischem Schwellwert. (a) zeigt das Original eines Fingerabdruckes. Durch die unterschiedliche Schwärzung gehen die Papillaren im rechten Bildteil weitgehend verloren. Durch den Einsatz eines dynamischen Schwellwertverfahrens konnte die Darstellung in (b) wesentlich verbessert werden.

Berechnung verwendet wird, auch um mehr als einen Bildpunkt verschoben werden. Für die dazwischen liegenden Bildpunkte wird der jeweils letzte berechnete Wert verwendet.

Die Annahme, dass in den Umgebungen $U(s(x,y))$ der Bildpunkte immer beide Modi auftreten, ist nicht immer zutreffend. In solchen Fällen muss das Histogramm der jeweiligen Umgebung berechnet und eine *Bimodalitätsprüfung* durchgeführt werden. Dabei wird geprüft, ob das Histogramm bimodal ist, d.h., ob sich zwei lokale Maxima deutlich ausprägen. Ein Algorithmus dazu könnte etwa wie folgt aussehen:

A17.5: Bimodalitätsprüfung.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild.

◇ $U(s_e(x,y))$ sei eine Umgebung des Bildpunktes in der Position (x,y).

Algorithmus:

(a) Von Null aufsteigend wird im Histogramm des Ausschnittes das erste lokale Maximum ermittelt. Der Grauwert und der Wert des Maximums werden mit $(gmax_1, max_1)$ bezeichnet.

(b) Anschließend wird überprüft, ob in einem bestimmten Abstand von $gmax_1$ (Fangbereich) ein weiteres lokales Maximum ist, das größer als max_1 ist. Falls dies der Fall ist, wird $(gmax_1, max_1) = (gmax'_1, max'_1)$ gesetzt.

(c) Von 255 absteigend wird jetzt im Histogramm das erste lokale Maximum $(gmax_2, max_2)$ gesucht.

(d) Wie oben wird im Fangbereich geprüft, ob ein lokales Maximum $max'_2 > max_2$ existiert. Falls dies erfüllt ist, wird $(gmax_2, max_2) = (gmax'_2, max'_2)$ gesetzt.

(e) Jetzt wird untersucht, ob zwischen $gmax_1$ und $gmax_2$ ein weiteres lokales Maximum liegt. In diesem Fall wäre das Histogramm sicherlich nicht bimodal.

(f) Es sei $(gmin, min)$ das Minimum zwischen $(gmax_1, max_1)$ und $(gmax_2, max_2)$ und $minmax$ das kleinere der beiden Maxima.

(g) Falls nun der Wert von $min/minmax$ kleiner als eine vorgegebene Schranke ist, wird das Histogramm des Ausschnittes als bimodal akzeptiert. Als Schwellwert kann in diesem Fall $c = gmin$ oder der mittlere Grauwert der Umgebung verwendet werden, da ja sichergestellt ist, dass das Histogramm bimodal ist.

(h) Falls die obige Bedingung nicht erfüllt ist, wird für die Binarisierung der alte Schwellwert weiter verwendet.

Ende des Algorithmus

Bild 17.8 ist ein Beispiel zu einem dynamischen Schwellwertverfahren mit Bimodalitätsprüfung. Bild 17.8-a zeigt das synthetische Originalbild, das aus einem dezimal ausgedruckten Graukeilbalken besteht, der einem Graukeil als Hintergrund überlagert wurde. Das dynamische Schwellwertverfahren mit Bimodalitätsprüfung ermöglicht die einwandfreie Trennung von „Hintergrund" und „Objekt" (Bild 17.8-b).

Abschließend sei noch auf die mehrdimensionalen Schwellwertverfahren hingewiesen. Hier werden in einem mehrkanaligen Bild zur Segmentierung eines Objektes für jeden Kanal Schwellwerte festgelegt, so dass die Trennung in einem mehrdimensionalen Merkmalsraum durchgeführt wird. Auf diese Techniken wird später noch näher eingegangen (Abschnitt 31.7).

Eine andere Möglichkeit ist es, den Hintergrund *vor* der Binarisierung zu korrigieren. Dazu werden zwei unterschiedliche Lösungsverfahren vorgestellt.

Bei manchen Anwendungen ist es nicht möglich, die Beleuchtung so einzurichten, dass die Objekte auf einem sauber ausgeleuchteten Bildhintergrund liegen. Das ist z.B. bei mikroskopischen Aufnahmen oder bei Durchlichtaufnahmen mit einem Leuchttisch der Fall. Wenn sich aber an der Beleuchtung und am Bildausschnitt nichts ändert, so kann man ein Kalibrierungsbild $\mathbf{K}$ erzeugen, indem man den Bildausschnitt *ohne* Objekt digitalisiert (Bild 17.9-b). Dieses Kalibrierungsbild kann man jetzt zur Korrektur des Hintergrundes

1	3	3	5	5	7	7	8	9	10	11	12	14	15
1	2	3	5	6	7	8	9	9	11	12	12	13	14
2	2	3	4	6	7	8	8	9	11	11	13	13	15
1	3	4	5	5	7	7	8	9	10	11	12	14	15
9	11	12	12	13	15	16	17	17	19	20	20	21	23
10	10	11	12	13	14	16	17	18	19	20	21	22	22
10	10	12	12	14	15	16	16	17	18	19	21	22	23
10	11	12	12	14	15	15	16	18	18	19	21	21	23
9	10	11	12	13	14	16	17	18	18	20	20	22	22
1	3	3	4	6	6	8	8	9	10	11	12	14	15
2	2	3	5	6	6	7	8	9	11	12	13	14	14
1	2	4	4	5	6	8	9	9	10	12	12	13	15
2	2	4	5	5	7	7	9	10	11	11	12	14	15

(a)

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

(b)

Bild 17.8: Synthetisches Bildbeispiel zur dynamischen Schwellwertberechnung mit Bimodalitätsprüfung. (a) zeigt einen dezimal ausgedruckten Ausschnitt aus einem Graukeil, dem ein Graukeilbalken überlagert wurde. Beide „Modi" wurden zufällig gestört. (b) Das dynamische Schwellwertverfahren ermöglicht die einwandfreie Trennung zwischen „Objekt" und „Hintergrund".

eines Bildes $\mathbf{S}_e$ mit Objekt (Bild 17.9-a) verwenden, indem man z.B. als Ausgabebild $\mathbf{S}_a$ (Bild 17.9-d) die absoluten Differenzen zwischen $\mathbf{S}_e$ und $\mathbf{K}$ berechnet:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a : s_a(x,y) = |s_e(x,y) - k(x,y)|. \tag{17.22}$$

Der Vergleich der Binarisierung des Originals (Bild 17.9-c) und des Differenzbildes (Bild 17.9-e) zeigt die Verbesserung. Werden in der weiteren Verarbeitung die Originalgrauwerte des Objekts benötigt, so kann man das binarisierte Differenzbild als Maske verwenden und damit das Objekt aus dem Original ausblenden.

Bei einer anderen Variante werden morphologische Operationen im Grauwertbild (Grundlagen dazu in Kapitel 19) herangezogen: Wenn es gelingt, das Objekt ganz auszublenden, also ein Bild zu erzeugen, in dem nur mehr der Hintergrund vorhanden ist, so kann durch eine Differenzbildung zwischen Original und Hintergrundbild der Einfluss der unterschiedlichen Beleuchtung eliminiert werden. Der Algorithmus dazu lautet wie folgt:

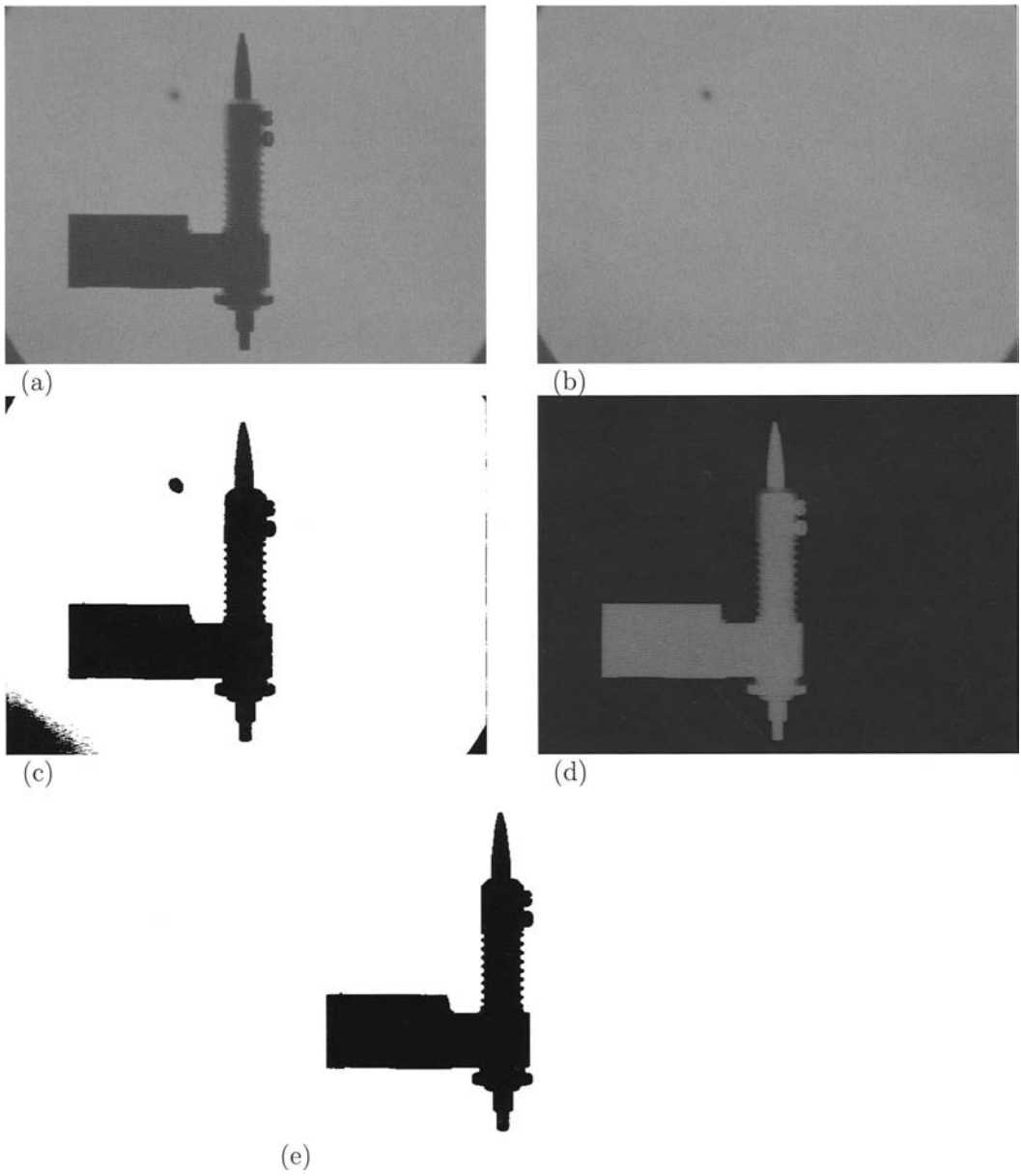


Bild 17.9: (a) Original. (b) Bildhintergrund (Kalibrierungsbild). (c) Binarisierung des Originals. Die Trennung des Objekts vom Hintergrund ist nicht sauber möglich. (d) Differenzbild: Es enthält die Absolutbeträge der Differenzen des Originals und des Kalibrierungsbildes. (e) Binarisierung des Differenzbildes. Das Ergebnis kann z.B. als Maske zum Ausblenden des Objektes aus dem Original verwendet werden.

A17.6: Elimination von Einflüssen durch die Beleuchtung.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild.

◇ Es wird angenommen, dass sich die „Objekte" dunkel vom hellen Hintergrund abheben.

Algorithmus:

(a) Berechnung eines Hilfsbildes $\mathbf{T}$ durch n Dilatationen, gefolgt von n Erosionen (*closing*-Operation).

(b) Berechnung des Differenzbildes $\mathbf{U}$: $\mathbf{U} = 127 + \mathbf{S}_e - \mathbf{T}$.

(c) Binarisiere $\mathbf{U}$ mit einem fixen Schwellwert c.

Ende des Algorithmus

Die Anzahl n der Dilatationen und Erosionen im Algorithmus **A17.6** hängt von der „Dicke" der Objekte ab. Man sieht sofort, dass das Verfahren dann geeignet ist, wenn das Objekt aus dünnen Linien besteht, die durch die Dilatationen verschwinden. Ist aufgrund des Bildinhaltes nur eine geringe Anzahl von Dilatationen durchzuführen (z.B $n = 2$ oder $n = 3$), so kann möglicherweise auf die nachfolgenden Erosionen verzichtet werden. Die Bilder 17.10-a bis 17.10-e sind Beispiele zu dieser Vorgehensweise.

Abschließend sei noch bemerkt, dass mit schnellen Bildverarbeitungssystemen die morphologischen Operationen durchaus in einem Verarbeitungszeitbereich liegen, der der Videoechtzeit nahe kommt. Damit kann auch die eben beschriebene Korrektur sehr schnell durchgeführt werden.

17.6 Logarithmische und exponentielle Skalierung

Bei der linearen Skalierung werden alle Bereiche der Grauwertmenge G gleichmäßig verändert, mit Ausnahme der Grauwerte, die durch die Transformation kleiner als 0 oder größer als 255 würden. Bei einem Bild könnte es aber wünschenswert sein, dass z.B. in dunklen Bildbereichen der Kontrast stärker angehoben wird als in hellen Bildbereichen. Dieser Effekt kann mit einer logarithmischen Funktion f_n erzielt werden. Es werden drei Parameter c_1, c_2 und c_3 benötigt, für die $0 < c_1 < c_2$ und $-255 < c_3 < +255$ gelten soll.

$$
\begin{aligned}
&min = \log(c_1); \quad max = \log(c_2);\\
&eps = (c_2 - c_1)/255;\\
&f_n(g) = 255 \cdot \frac{\log(c_1 + g \cdot eps) - min}{max - min}; \quad g \epsilon G.\\
&LuT(g + c_3) = f_n(g)
\end{aligned}
\tag{17.23}
$$

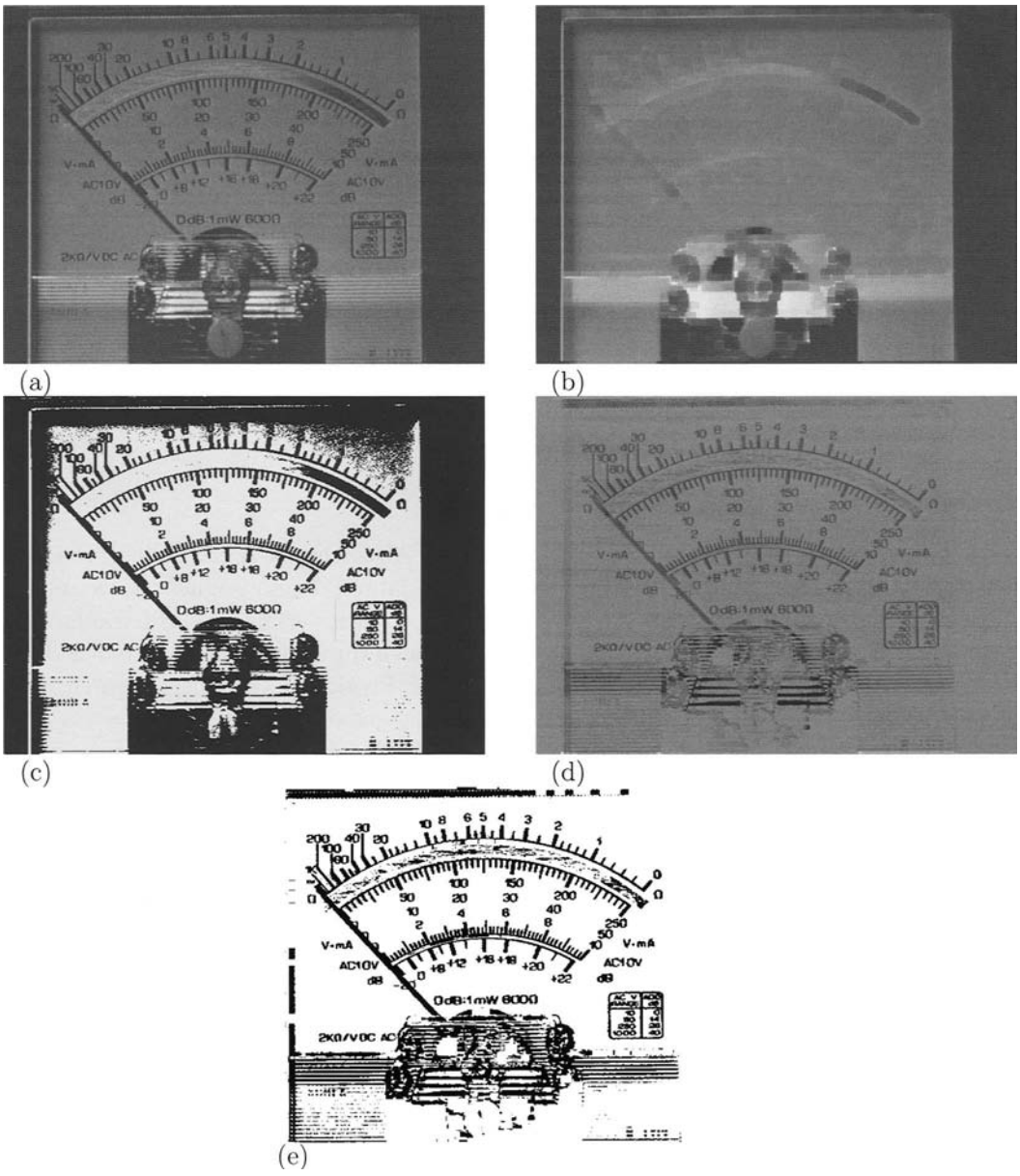


Bild 17.10: Binärbilderzeugung in Verbindung mit morphologischen Operationen. (a) Original. (b) *closing*-Operation: Vier Dilatationen gefolgt von vier Erosionen (Hintergrundbild). (c) Fixer Schwellwert ohne Korrektur des Hintergrundes. (d) Differenzbild: 127 + Original - Hintergrundbild. (e) Fixer Schwellwert nach einer Korrektur des Hintergrundes.

Durch die Wahl von c_1 und c_2 kann der Steigungsverlauf von f_n beeinflusst werden. Mit c_3 wird f_n im Grauwertintervall G nach links oder rechts verschoben.

Eine exponentielle Skalierung erhält man, wenn in (17.23) statt des Logarithmus die Exponentialfunktion verwendet wird. Es muss dann $c_1 < c_2$ gelten, die Beschränkung für c_3 bleibt wie oben bestehen.

Beispiele zur nichtlinearen Skalierung zeigen die Bilder 17.11-a bis 17.11-e. Bild 17.11-a ist das Original. Als Zielsetzung soll eine Darstellung gefunden werden, bei der sowohl das Zifferblatt als auch die LCD-Anzeige deutlich zu sehen sind. Bei Bild 17.11-b wurde eine lineare Skalierungsfunktion f_n verwendet. Das gesteckte Ziel wurde damit nicht erreicht. Bei Bild 17.11-c wurde eine logarithmische Funktion f_n mit $c_1 = 0.0001$, $c_2 = 20.0$ und $c_3 = 38$ verwendet. Der Kontrast in den dunklen Bereichen wurde merklich angehoben, was zur Folge hat, dass die LCD-Anzeige im unteren Bildbereich gut zu lesen ist. Der Kontrast in den hellen Bildbereichen wurde dagegen nicht so stark angehoben. Bild 17.11-d entstand mit Hilfe einer exponentiellen Funktion f_n ($c_1 = -6$, $c_2 = +6$, $c_3 = -155$). Hier werden die hellen Bildbereiche, also im wesentlichen der Bildhintergrund, besser kontrastiert. Die Bilder 17.11-e und 17.11-f zeigen die grafische Darstellung der Skalierungsfunktion .

An dieser Stelle sei noch vermerkt, dass die nichtlinearen Skalierungsverfahren durchaus eine Berechtigung im Rahmen der Bildvorverarbeitung für eine nachfolgende Informationsextraktion haben. Eine logarithmische Skalierung z.B. in Verbindung mit einer nachfolgenden Hochpassfilterung und einer anschließenden exponentiellen Skalierung wirkt sich so aus, dass dunkle Bildbereiche von der Hochpassfilterung stärker betroffen sind als helle.

17.7 Ebnen der Grauwerte

Das *Ebnen* eines Grauwertbildes oder die *Histogrammlinearisierung* ist ebenfalls eine nichtlineare Transformation der Grauwerte eines Bildes. Hier wird die Funktion f_n aber nicht vorgegeben, sondern aus dem zu verarbeitenden Bild berechnet und so auf die gegebene Grauwertverteilung des Bildes abgestimmt. Der Algorithmus sieht folgendermaßen aus:

A17.7: Ebnen der Grauwerte.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild mit der Grauwertmenge $G = \{0, 1, ..., 255\}$.

Algorithmus:

(a) Berechnung des Histogramms $p(g)$ des Bildes.

(b) Berechnung der relativen Summenhäufigkeiten:

$$h(g) = \sum_{k=0}^{g} p(k), \quad g = 0, 1, ..., 255;$$

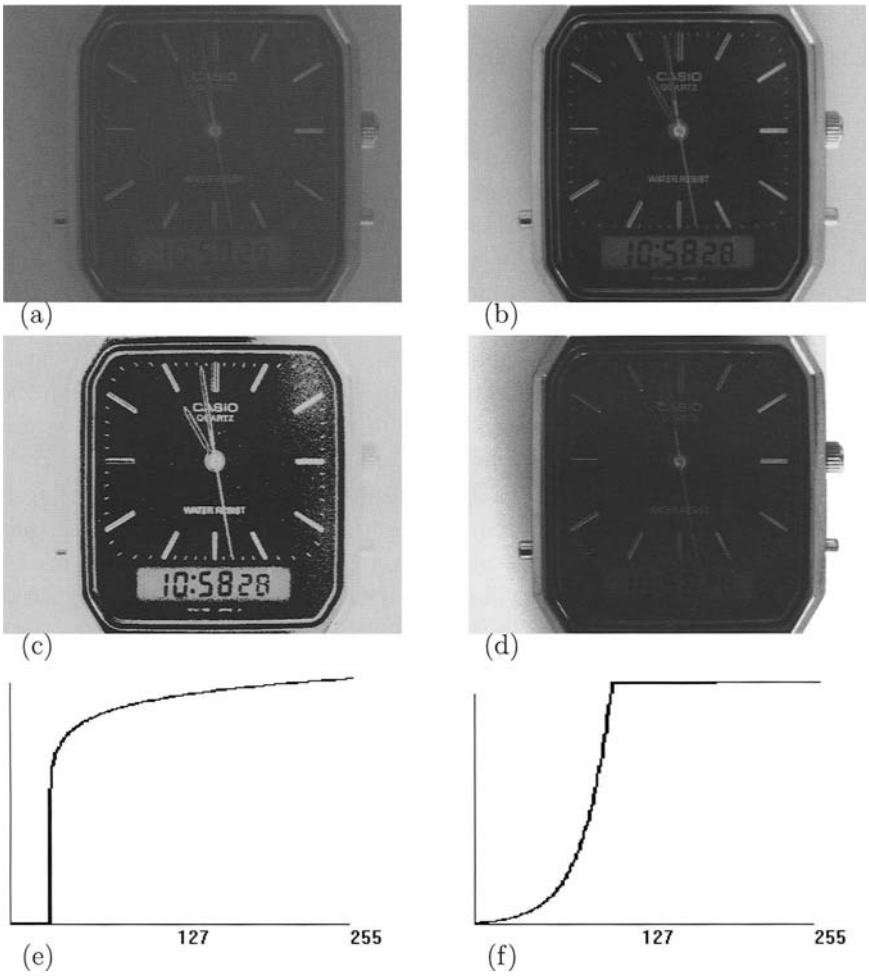


Bild 17.11: Beispiele zur nichtlinearen Skalierung. (a) Original. Als Zielsetzung soll eine Darstellung gefunden werden, bei der sowohl das Zifferblatt als auch die LCD-Anzeige deutlich zu sehen sind. (b) Lineare Skalierungsfunktion f_n: Das gesteckte Ziel wurde damit nicht erreicht. (c) Logarithmische Funktion f_n: Der Kontrast in den dunklen Bereichen wurde merklich angehoben, wodurch die LCD-Anzeige gut zu lesen ist. In den hellen Bereichen wurde der Kontrast nicht so stark verändert. (d) Exponentielle Skalierungsfunktion f_n: Es wurde vor allem der helle Bildhintergrund stark kontrastiert. (e) Skalierungsfunktion zu (c). (f) Skalierungsfunktion zu (d).

(c) Berechnung der Skalierungsfunktion f_n:
$f_n(g) = 255 \cdot h(g)$;

(d) Berechnung der *look-up*-Tabelle und Skalierung des Bildes gemäß Algorithmus **A17.1**.

Ende des Algorithmus

Wie sich diese Transformation auswirkt, wird anhand der Bilder 17.12 und 17.13 erläutert. Die Bilder 17.12-a und 17.12-b zeigen das Original und das transformierte Bild. Die Histogramme und die Skalierungsfunktionen sind in den Bildern 17.12-c und 17.12-d abgebildet. Man sieht deutlich, dass das transformierte Bild kontrastreicher ist und feine Bildstrukturen, z.B. in der linken oberen Ecke, besser zu sehen sind. Ein nochmaliges Ebnen eines bereits geebneten Bildes würde am Ergebnis nichts mehr ändern. Das sieht man auch am speziellen Verlauf der relativen Summenhäufigkeiten eines geebneten Bildes.

Wenn im Eingabebild alle 256 möglichen Grauwerte auftreten, sollte für das Ausgabebild gelten: $p(g) = \frac{1}{256} = 0.39\%$. Dass das Histogramm diesen Sachverhalt nicht richtig wiedergibt, hat drei Gründe: Erstens ist der Maßstab der Histogramme in Richtung der Ordinate sehr stark vergrößert, zweitens treten bei dieser Transformation der Grauwerte Rundungsfehler auf, und drittens fehlen im transformierten Bild einige Grauwerte, was an den Lücken im Histogramm zu erkennen ist. Wenn die daneben liegenden Grauwerte gleichmäßig auf die Lücken verteilt werden, wird der theoretische Sachverhalt besser angenähert.

Bei manchen Bildern kann die Histogrammlinearisierung aber auch nicht geeignet sein. Meistens ist das bei Bildern der Fall, die große homogene Flächen mit nahe beieinander liegenden Grauwerten haben. Als Beispiel dient Bild 17.13-a. Es hat ein bimodales Histogramm (Bild 17.13-c). Die relativen Summenhäufigkeiten und somit auch der Verlauf der Skalierungsfunktion f_n sind in das Histogramm eingeblendet. Die Funktion verdeutlicht, dass sowohl in dunklen als auch in hellen Bildbereichen der Kontrast angehoben wird, während er bei den mittleren Grauwerten nahezu unverändert bleibt. Das hat zur Folge, dass z.B. im helleren Bildbereich des transformierten Bildes 17.13-b das Rauschen stark hervortritt. Bild 17.13-d zeigt das Histogramm des skalierten Bildes. Hier gilt dasselbe wie bei Bild 17.12-d.

Man hat hier also eine parameterfreie Methode. Wo kann sie sinnvoll angewendet werden? Sollen digitalisierte Grauwertbilder über einfache Schwarz-/Weißdrucker ausgegeben werden, so hat man das Problem, dass diese Ausgabegeräte in der Regel nicht über die 256 verschiedenen Graustufen verfügen, wie ein Monitor des Bildverarbeitungsystems. Ist das Originalbild verhältnismäßig kontrastarm, so wird es nach der Ausgabe noch kontrastärmer sein. Gibt man jedoch das geebnete Bild aus, so hat man in der Regel eine ausreichend gute Darstellung.

Ein weiteres Beispiel zum Ebnen der Grauwerte ist das Angleichen von Farbauszügen. Ein RGB-Bild kann mit Hilfe einer Farbvideokamera und eines Farbvideo-Digitalisierers aufgezeichnet werden. Für einen Bildpunkt werden von Systemen dieser Art meistens 24

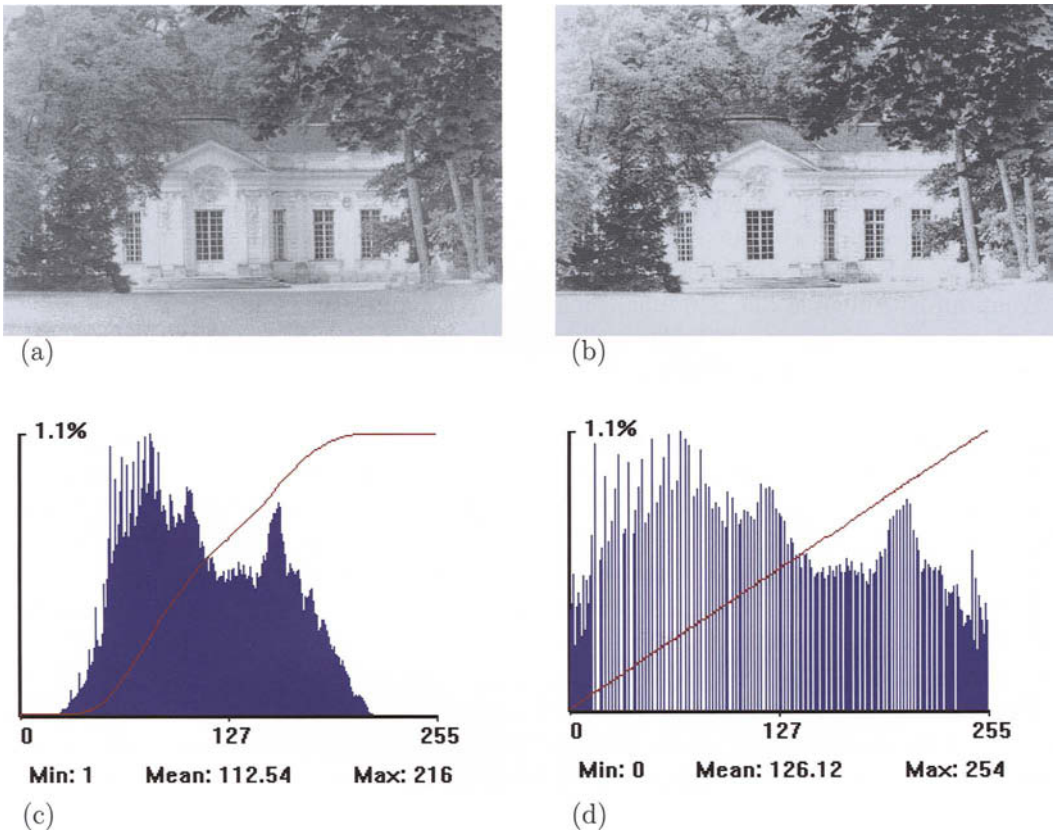


Bild 17.12: Beispiele zum Ebnen der Grauwertverteilung. (a) Original (Amalienburg im Schloßpark Nymphenburg in München; aus dem Stadtarchiv). (b) Transformiertes Bild. Man sieht deutlich, dass das Bild kontrastreicher als das Original ist. Feine Bildstrukturen, wie z.B. in der linken oberen Bildecke, sind besser zu sehen. (c) Histogramm und Skalierungsfunktion, gebildet aus den relativen Summenhäufigkeiten des Originalbildes. (d) Histogramm und relative Summenhäufigkeiten des transformierten Bildes. Ein nochmaliges Ebnen eines bereits geebneten Bildes würde nichts mehr verändern. Das sieht man auch an der speziellen Form der relativen Summenhäufigkeiten des geebneten Bildes.

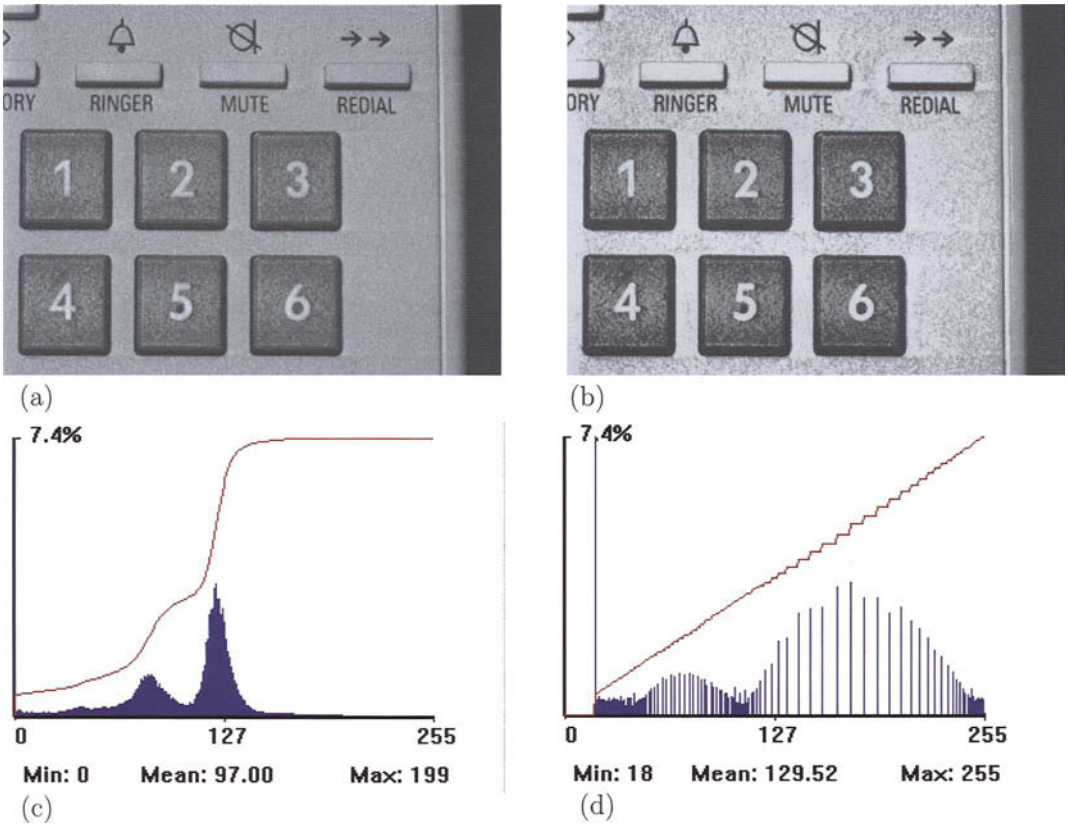
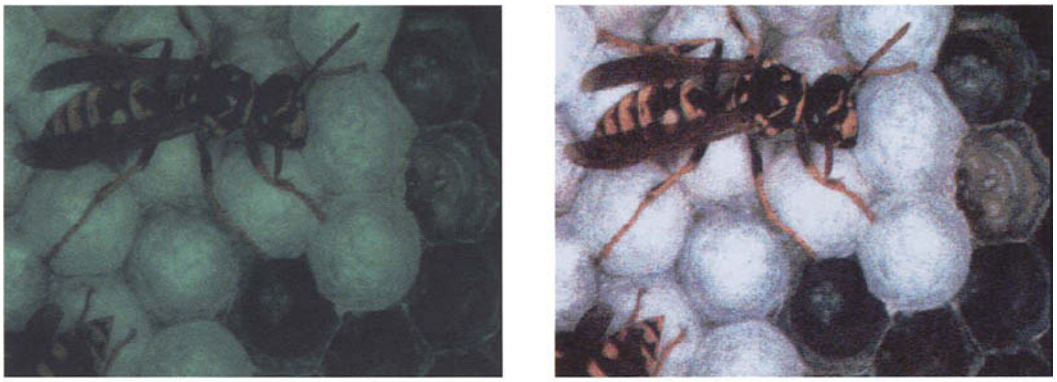


Bild 17.13: Beispiele zum Ebnen der Grauwertverteilung. (a) Testbild. (b) Transformiertes Bild. In hellen Bildbereichen tritt das Rauschen deutlich hervor. Das liegt daran, dass das Original größere homogene Bildbereiche mit ähnlichen Grauwerten besitzt. Die Grauwerte dieser Intervalle werden im Ausgabebild stark aufgespreizt. (c) Histogramm und Skalierungsfunktion des Originals. (d) Histogramm und relative Summenhäufigkeiten des transformierten Bildes. Das linke Häufigkeitsmaximum entspricht dem Bereich mit dem Grauwert 0 am rechten Bildrand des Originals. Im Histogramm des Originals ist dieser Balken durch die Ordinate verdeckt.



(a) (b)

Bild 17.14: RGB-Farbbilder, die mit Rot-, Grün- und Blaufiltern und einem normalen 8-Bit-Framegrabber erzeugt wurden. Als Vorlage zur Digitalisierung diente hier ein Farbdruck. Die Voraussetzung, dass sich der zu digitalisierende Bereich nicht bewegt, war somit gegeben. (a) Originalbild nach der Digitalisierung. (b) Werden die Farbauszüge getrennt geebnet, so ergibt sich oft eine bessere Farbqualität.

Bit verwendet. Bei Anwendungen, bei denen sich das Beobachtungsgebiet nicht bewegt, kann auch mit einer Schwarz-/Weißkamera, einem normalen 8-Bit Video-Digitalisierer und Farbfiltern gearbeitet werden. Nacheinander werden hier drei Digitalisierungen mit einem Rot-, einem Grün- und einem Blaufilter durchgeführt.

Um eine gute farbliche Darstellung zu erzielen, können die drei Farbauszüge getrennt voneinander mit den hier erläuterten Skalierungsverfahren behandelt werden. Bei der Verwendung von Farbfiltern, deren spektrale Bandbreiten nicht genau bekannt sind und die auch nicht exakt zusammenpassen, ergeben sich Farbdarstellungen, die oft weit vom Original entfernt sind. Praktische Erfahrungen haben gezeigt, dass das Ebnen der drei Farbauszüge oft die Farbqualität wesentlich verbessert. Bildbeispiele dazu zeigen die Bilder 17.14-a und 17.14-b.

17.8 Kombination einer Grauwertskalierung mit einer Hochpassfilterung

Steht bei der Anwendung der Grauwertskalierung das Erzeugen eines hinsichtlich der Helligkeit und des Kontrastes guten Bildes im Vordergrund, so kann dies durch die Kombination mit einem hochpassgefilterten Bild erzielt werden. Der theoretische Hintergrund ist in Bild 17.15 vereinfacht am Beispiel einer eindimensionalen Funktion $s(x)$ dargestellt. Der Verlauf von $s(x)$ (Bild 17.15-a) kann als Querschnittsprofil durch eine Grauwertkante in

einem Bild interpretiert werden. Die erste und zweite Ableitung zeigen die Bilder 17.15-b und 17.15-c. Wird die zweite Ableitung der Originalfunktion $s(x)$ überlagert, z.B. durch $s(x) - s''(x)$, so ergeben sich in den Bereichen der Funktion mit starken Steigungsänderungen „Überschwinger" (Bild 17.15-d).

Auf ein Grauwertbild übertragen heißt das, dass sich eine Grauwertkante besser abhebt, weil sie von einem schmalen, zuerst hellen, dann dunklen Streifen eingesäumt wird.

Als diskrete Nachbildung der zweiten Ableitung eines Bildes kann z.B. der Laplace-Operator (Grundlagen dazu in Kapitel 18) verwendet werden. Als Filterkerne können die folgenden Masken dienen:

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad \text{oder} \quad \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \tag{17.24}$$

Im Folgenden ist der Algorithmus dazu kurz zusammengestellt:

A17.8: Grauwertskalierung mit einer Hochpassfilterung.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x, y))$ ein einkanaliges Grauwertbild.

Algorithmus:

(a) Berechnung des skalierten Bildes $\mathbf{T} = (t(x, y))$ durch eine Skalierungsfunktion f_n.

(b) Anwendung des Laplace-Operators auf das Originalbild $\mathbf{S}_e$. Das Ergebnis sei $\mathbf{S}''$.

(c) Überlagerung von $\mathbf{S}''$ und des skalierten Bildes $\mathbf{T}$ etwa gemäß:

$t(x, y) - c \cdot s''(x, y)$;

Der Parameter c steuert das Gewicht der Überlagerung. Er sollte bei praktischen Anwendungsfällen anhand einiger Testbilder ermittelt werden.

Ende des Algorithmus

Die Bildfolge 17.16-a bis 17.16-c zeigt Beispiele dazu. Hier wurde das Originalbild mit einer linearen Skalierungsfunktion transformiert (Bild 17.16-b). Das skalierte Bild wurde dann mit dem Laplace-gefilterten Original überlagert. Da beide Teilverfahren mit geeigneter Bildverarbeitungshardware in Videoechtzeit durchgeführt werden können, lässt sich auch das hier erläuterte „Schärfen" eines Grauwertbildes (*image sharpening*) in Videoechtzeit durchführen.

Gute Ergebnisse erhält man in der Regel auch, wenn man bei etwas verrauschten Bildern zuerst eine Medianfilterung (Kapitel 19) durchführt und dann das Ergebnis, wie oben beschrieben, hochpassfiltert.

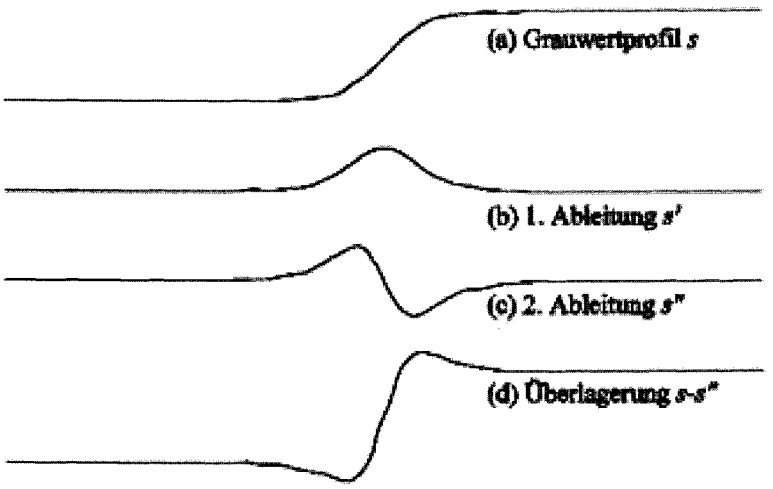


Bild 17.15: (a) Grauwertprofil $s(x)$ einer Grauwertkante in einem Bild. (b) und (c) Erste und zweite Ableitung von $s(x)$. (d) Überlagerung der Originalfunktion mit ihrer zweiten Ableitung, gemäß $s(x) - s''(x)$.

17.9 Kalibrierung der Grauwerte

Digitalisierungssysteme (allgemeiner: bildaufzeichnende Systeme) sollten monoton arbeiten. Damit ist gemeint, dass gleiche Grautöne, die im Original an verschiedenen Stellen auftreten, im digitalisierten Bild denselben Grauwert haben sollten. Das ist in der Praxis selten der Fall. Bei einer fotografischen Aufnahme können z.B. bereits bei der Belichtung des Fotomaterials *Randabschattungen* (*Vignettierungen*) auftreten, die sich je nach der Qualität des Objektivs mehr oder weniger stark auswirken. Selbst eine fehlerfreie Digitalisierung kann diesen Effekt nicht mehr eliminieren. Als weitere verfälschende Einflüsse können aufgezählt werden: die Filmentwicklung, die unterschiedliche Ablenkung des Kathodenstrahls bei der Aufzeichnung mit einer Videokamera oder die unterschiedlichen Arbeitspunkte der einzelnen Sensoren eines Abtasters mit einem Sensorfeld.

Zunächst wird untersucht, wie eine multiplikative Störung der Grauwerte kalibriert werden kann. Es sei $\mathbf{S}_e = (s_e(x,y))$ das digitalisierte Grauwertbild mit der überlagerten Störung. Dann kann für die Grauwerte von $\mathbf{S}_e$ geschrieben werden:

$$s_e(x,y) = e(x,y) \cdot s_u(x,y), \tag{17.25}$$

wobei $e(x,y)$ der multiplikative Störfaktor und $s_u(x,y)$ der ungestörte Grauwert in der Position (x,y) ist. Die Korrektur des bildaufzeichnenden Systems, einschließlich der Digitalisierung, wird mit Hilfe eines Kalibrierungsbildes, z.B. einer homogenen, grauen Vorlage, durchgeführt:

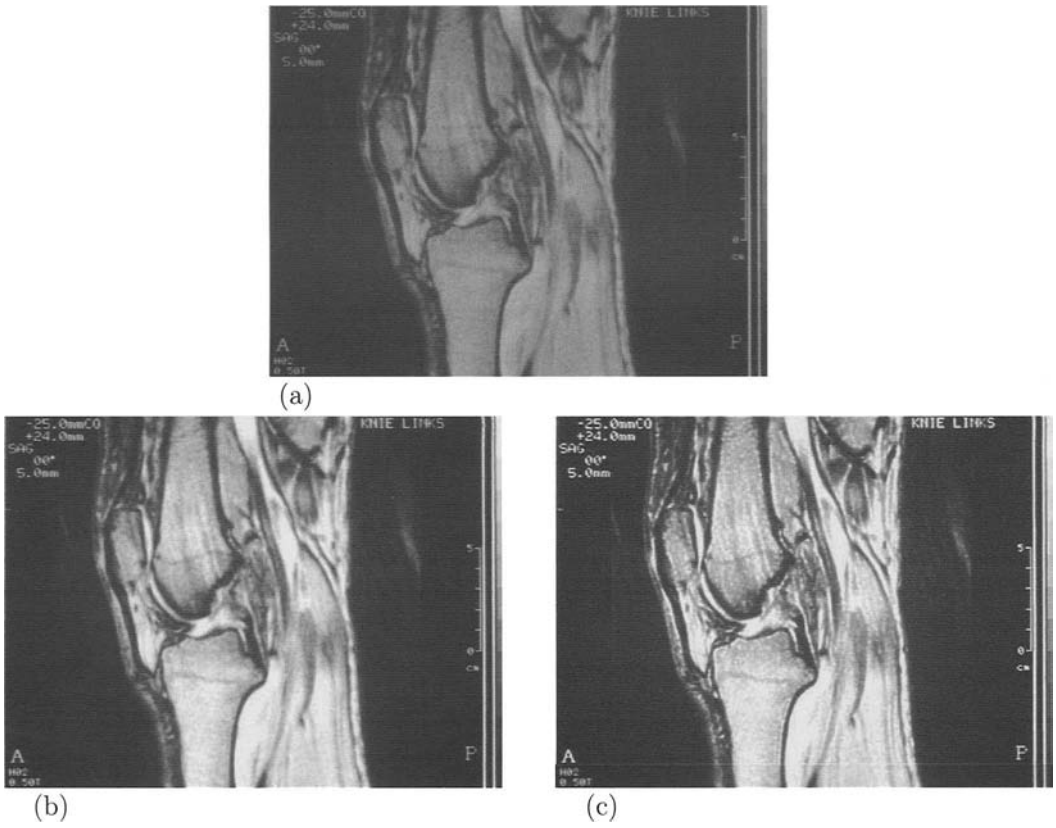


Bild 17.16: Skalierung der Grauwerte in Verbindung mit einer Hochpassfilterung. (a) Kernspintomografie des linken Knies. (b) Skaliertes Bild mit einer linearen Skalierungsfunktion f_n. (c) Zusätzliche Überlagerung des skalierten Bildes mit dem Laplace-gefilterten Original. Man sieht, dass feine Linienstrukturen besser sichtbar werden.

$\mathbf{S}_k = (s_k(x,y))$ digitalisiertes Kalibrierungsbild,
$\mathbf{S}_{k'} = (s_{k'}(x,y))$ digitalisiertes Kalibrierungsbild, das sich bei einer fehlerfreien Aufzeichnung und Digitalisierung ergeben müßte.

Für $\mathbf{S}_{k'}$ kann $s_{k'}(x,y) = c$ angenommen werden, da eine fehlerfrei aufbereitete graue Vorlage ein homogen graues Bild liefern müßte. Mit diesem Kalibrierungsbild können jetzt die Störfaktoren ermittelt werden:

$$s_k(x,y) = e(x,y) \cdot c \text{ oder } e(x,y) = \frac{s_k(x,y)}{c}. \tag{17.26}$$

Die Grauwerte des korrigierten Bildes berechnen sich daraus gemäß:

$$s_u(x,y) = \frac{s_e(x,y)}{e(x,y)} = c \cdot \frac{s_e(x,y)}{s_k(x,y)} \tag{17.27}$$

Ein anderes Kalibrierungsproblem tritt bei Abtastgeräten auf, die eine Bildzeile parallel mit einem Feld von Sensoren erfassen (Bild 17.17-a), wie z.B. ein Scanner. Ein Problem dabei ist, dass die einzelnen Sensoren nicht alle so exakt abgestimmt werden können, dass sie für denselben Grauton nach der Digitalisierung auch denselben Grauwert liefern. Ein so digitalisiertes Bild ist dann in Zeilenrichtung durch Streifen gestört.

Der Sensor y (abkürzend für: der Sensor, der die Bildspalte y abtastet) und der Sensor $y+1$ erzeugen somit für denselben Grauton die Grauwerte

$$c + f_y(g) \text{ und } c + f_{y+1}(g). \tag{17.28}$$

Die Korrekturfunktion $f_y(g)$ für die Bildspalte y ist abhängig vom Grauwert g, da der Sensor im dunklen Grautonbereich anders arbeitet als im hellen. Wenn in einem bestimmten Grautonbereich ein linearer Verlauf der Sensorempfindlichkeit angenommen werden kann, ist es möglich, mit einer hellen und einer dunklen Kalibrierungsvorlage die Korrekturfunktion $f_y(g)$ als Gerade zu ermitteln. Es sei:

$\mathbf{S}_H = \big(s_H(x,y)\big) = \big(c_H + f_y(g)\big)$ das helle, digitalisierte Kalibrierungsbild,
$\mathbf{S}_D = \big(s_D(x,y)\big) = \big(c_D + f_y(g)\big)$ das dunkle, digitalisierte Kalibrierungsbild,
$m_{y,H}$ der Mittelwert der Bildspalte y im hellen Kalibrierungsbild und
$m_{y,D}$ der Mittelwert der Bildspalte y im dunklen Kalibrierungsbild.

Mit diesen Voraussetzungen berechnet sich die Korrekturgerade für die Bildspalte y zu:

$$f_y(g) = \big(m_{y,H} - m_{y,D}\big) \cdot \frac{g - c_D}{c_H - c_D} + m_{y,D}. \tag{17.29}$$

Für c_D und c_H kann der Mittelwert des dunklen oder hellen Kalibrierungsbildes verwendet werden. Bild 17.17-b zeigt die grafische Darstellung der Kalibrierungsgeraden. Die Bilder 17.17-c und 17.17-d zeigen ein praktisches Beispiel.

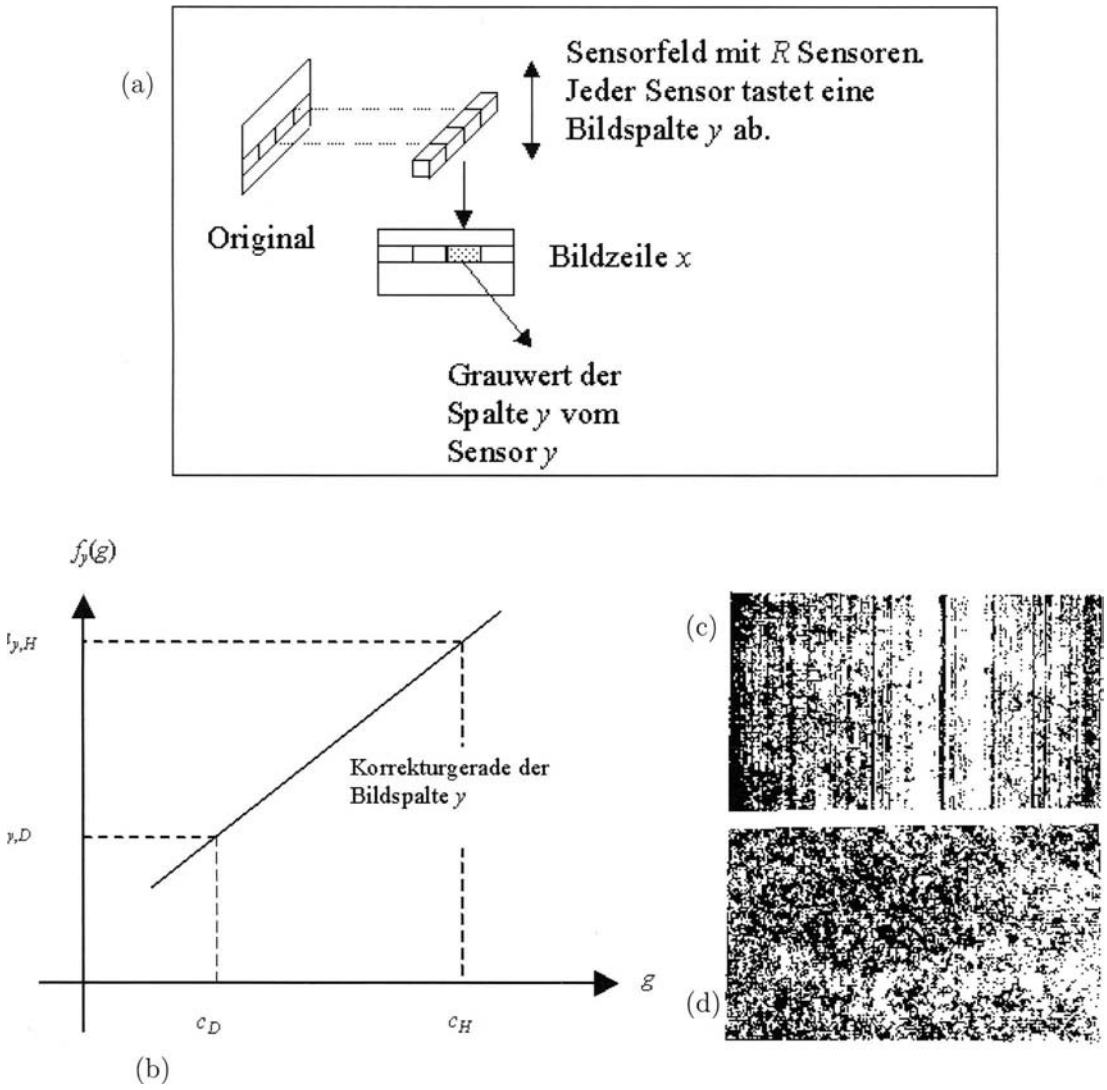


Bild 17.17: Kalibrierung bei CCD-Scannern. (a) Abtastung einer Vorlage mit einem Sensorfeld, das eine Bildzeile parallel erfasst. (b) Grafische Darstellung der Korrekturgeraden für die Bildspalte y. (c) Streifenmuster vor der Kalibrierung mit starker Kontrastverstärkung. (d) Kalibriertes Bild.

17.10 Berechnung einer neuen Grauwertmenge

Wird zu einem digitalisierten Grauwertbild $\mathbf{S}_e = (s_e(x,y))$ das Histogramm $p_{\mathbf{S}_e}(g)$ berechnet, so zeigt es oft eine Grauwertverteilung, die nicht alle Grauwerte der Grauwertmenge $G = \{0, 1, ..., 255\}$ umfasst. In solchen Fällen können die Grauwerte, wie in diesem Kapitel beschrieben, skaliert werden, sodass sie die gesamte Grauwertmenge ausnützen. Die Bilddaten können aber auch komprimiert gespeichert werden, wenn sich zeigt, dass statt der ursprünglichen 8 Bit weniger als 8 Bit pro Bildpunkt ausreichend sind. Das ist z.B. dann der Fall, wenn

$$\Delta g = g_{max} - g_{min} \leq 128, \tag{17.30}$$

wobei g_{min} und g_{max} der minimale und der maximale im Bild $\mathbf{S}_e$ auftretende Grauwert sind. Wie viele Bit zur komprimierten Speicherung notwendig sind, ergibt sich aus der Größe der Differenz Δg in (17.30). Falls z.B. $127 \geq \Delta g \geq 64$ werden 7 Bit entsprechend 128 Grauwerten, also $G' = \{0, 1, ..., 127\}$ oder falls $63 \geq \Delta g \geq 32$ werden 6 Bit, entsprechend 64 Grauwerten, also $G' = \{0, 1, ..., 63\}$ benötigt.

Einen weiteren Hinweis auf die Anzahl der benötigten Bit pro Bildpunkt gibt die Entropie (Abschnitt 16.11):

$$H = -\sum_{g=0}^{255} \Big(p_{\mathbf{S}_e}(g) \cdot \log_2 p_{\mathbf{S}_e}(g) \Big). \tag{17.31}$$

Für das Bild 17.1-a in Kapitel 17.3 ist der minimale Grauwert $g_{min} = 2$ und der maximale Grauwert $g_{max} = 255$. Die Differenz $\Delta g = 253$ zeigt, dass das Bild ohne Informationsverlust nicht nach der hier beschriebenen Methode mit weniger als 8 Bit pro Bildpunkt gespeichert werden kann. Dies bestätigt auch der Wert der Entropie $H = 7.51$.

Lässt man geringe Informationsverluste zu, die den visuellen Eindruck der Bilder oft nicht beeinflussen, so kann mit den im Folgenden beschriebenen Beispielen eine weitere Erhöhung der Speicherungsdichte erreicht werden. Zu Bild 17.1-a sind in Bild 17.18 die Binärbilder der einzelnen Bitebenen dargestellt. Das Binärbild zur Bitebene $i, 0 < i < 7$, wird dadurch erzeugt, dass von jedem Grauwert $s(x,y)$ in seiner Darstellung als Dualzahl nur das Bit in der Position i bildlich dargestellt wird:

$$s_i(x,y) = \begin{cases} 1 & \text{falls } \big(s_e(x,y).\text{AND}.2^i\big) \neq 0 \\ 0 & \text{sonst.} \end{cases} \tag{17.32}$$

Beim Binärbild zur Bitebene $i = 0$ wird so z.B. von jedem Grauwert nur das niederwertigste Bit dargestellt.

Bild 17.18 zeigt, dass die Bitebenen 0 und 1 fast keine Bildinformation enthalten und daher weggelassen werden können. Das bedeutet in diesem Beispiel eine Reduktion von 8 auf 6 Bit pro Bildpunkt, also von 256 auf 64 verschiedene Grauwerte.

Bei allen Ermittlungen einer neuen Grauwertmenge sollte noch der resultierende Kompressions- oder Reduktionsfaktor gegenüber der Tatsache abgewogen werden, dass dann

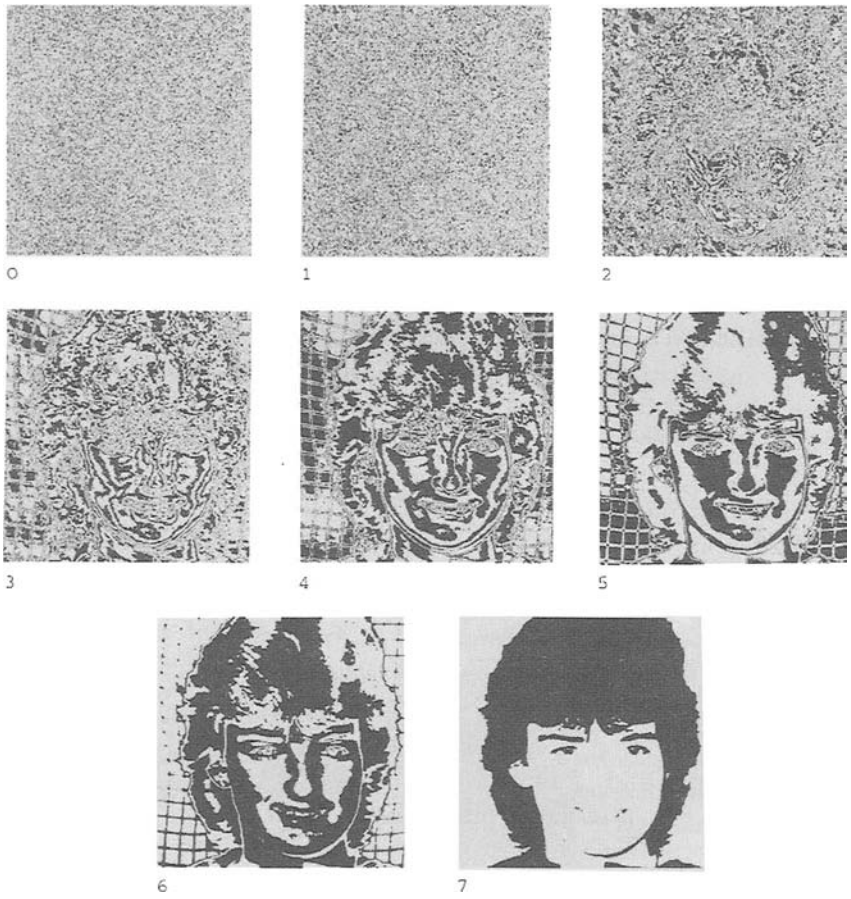


Bild 17.18: Binärbilder der einzelnen Bitebenen zu Bild 17.1-a. Man sieht deutlich, dass die Bitebenen 0 und 1 fast keine Bildinformation enthalten und daher weggelassen werden können.

der Grauwert eines Bildpunktes nicht mehr ein ganzes Byte belegt, wodurch sich u.U. erhöhte Rechenzeiten bei den Zugriffen auf die einzelnen Bildpunkte ergeben. So wäre z.B. eine Datenreduktion von 8 auf 7 Bit in einer byteorientierten Rechenanlage nicht sinnvoll, da bei jedem Zugriff auf einen Bildpunkt der zugehörige Grauwert durch Maskieren und Verschieben der Bitmuster von zwei Byte ermittelt werden muss.

17.11 Rekonstruktion des höchstwertigen Bit

Auch das höchstwertige Bit eines Grauwertbildes $\mathbf{S}_e = (s_e(x, y)$ kann zur datenkomprimierten Speicherung weggelassen werden. Das Original lässt sich aus der komprimierten Darstellung wieder rekonstruieren, wenn die Annahme zutrifft, dass zwischen benachbarten Bildpunkten kein allzu abrupter Übergang in den Grauwerten ist. Dieses Verfahren lässt sich am besten am Beispiel eines Graukeils (Bild 17.19-a) erläutern. Die grafische Darstellung des Grauwertverlaufs entlang der markierten Strecke $\overline{AB}$ ergibt ein Geradenstück. Wird das höchstwertige Bit 7, das nur bei den Grauwerten $128 \leq s(x, y) \leq 255$ gesetzt ist, maskiert, erhält man den gestrichelten Verlauf (Bild 17.19-b). Den dazugehörigen modifizierten Graukeil zeigt Bild 17.19-c. Soll aus der komprimierten Darstellung das Original rekonstruiert werden, so wird der Grauwertsprung daran erkannt, dass die Differenz zum rechten Nachbarn größer als 1 ist.

Bei natürlichen Bildern ist die Rekonstruktion nicht ganz so einfach, da man zunächst nicht weiß, ob der Grauwert am Zeilenanfang ursprünglich im Intervall $[0, 127]$ oder $[128, 255]$ lag. Wird jedoch diese Information in einem Bit zusätzlich mit abgespeichert, so kann bei der Rekonstruktion an jedem Zeilenanfang eine Hilfsgröße richtig auf 0 oder 128 gesetzt werden. Außerdem können im Verlauf der Grauwerte einer Bildzeile mehrere Sprungstellen auftreten. Bei jeder Sprungstelle muss dann die Hilfsgröße auf 128 oder 0 umgeschaltet werden. Schließlich muss noch für die Schwelle c ein geeigneter Wert festgelegt werden, der davon abhängt, wie groß die Grauwertübergänge im Original sind. Bild 17.20 zeigt das mit diesem Verfahren komprimierte Bild 17.1-a.

Nachdem in Bild 17.1-a die Bitebenen 0 und 1 weggelassen werden können und die Bitebene 7 durch stetige Fortsetzung rekonstruiert werden kann, lassen sich somit die Grauwerte ohne wesentlichen Informationsverlust mit 5 Bit pro Bildpunkt darstellen.

17.12 Reduktion der Grauwertmenge durch Differenzbildung

Eine weitere Möglichkeit der Reduktion (Kompression) der zur Darstellung eines Grauwertbildes notwendigen Grauwertmenge ergibt sich, wenn nicht wie bisher jeder Bildpunkt getrennt für sich, sondern im Rahmen seiner 4- oder 8-Nachbarschaft betrachtet wird. Ein einfaches Beispiel hierzu ist die Berechnung und Speicherung der Grauwertdifferenzen zum jeweiligen rechten Nachbarn innerhalb einer Bildzeile:

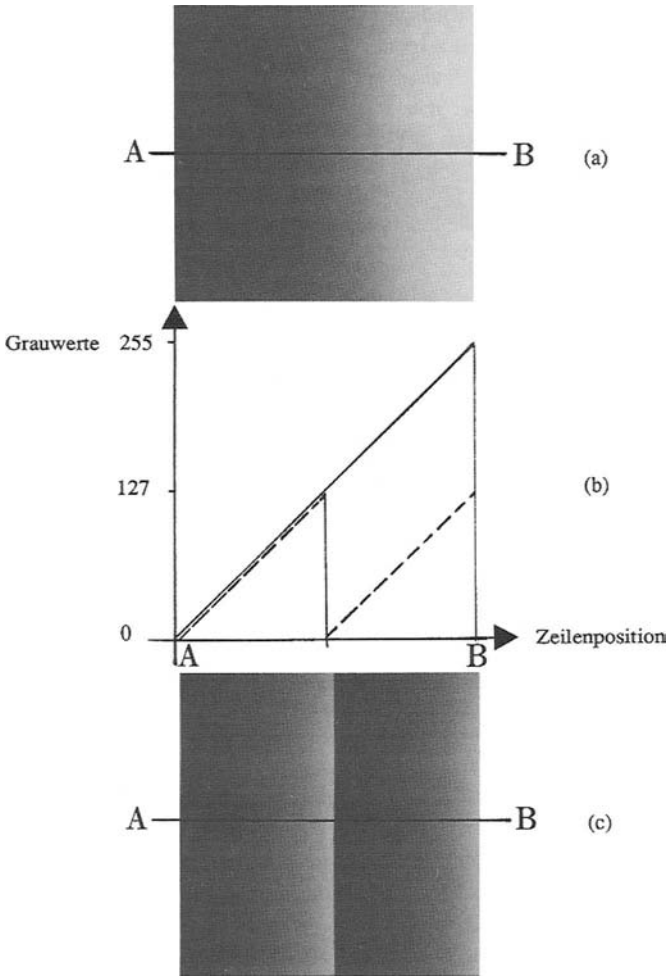


Bild 17.19: Rekonstruktion des höchstwertigen Bit. (a) Graukeil (b) Grafische Darstellung des Grauwertverlaufs entlang der Strecke $\overline{AB}$ vor und nach dem Ausblenden des höchstwertigen Bit 7. (c) Graukeil, bei dem Bit 7 ausgeblendet wurde.

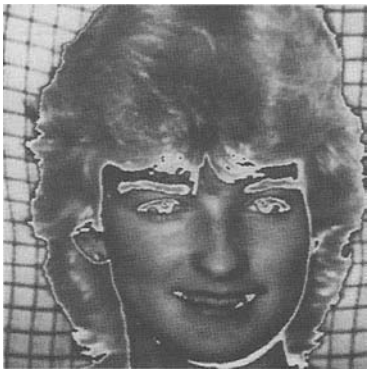


Bild 17.20: Testbild mit ausgeblendetem Bit 7. Das Original kann durch stetige Fortsetzung der Grauwerte innerhalb der Bildzeilen rekonstruiert werden.

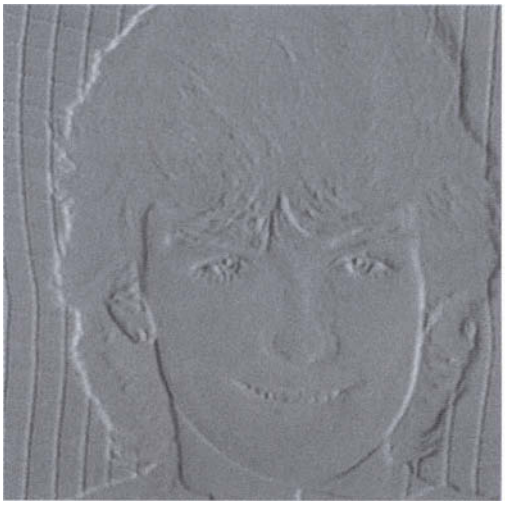
$\mathbf{S}_e \rightarrow \mathbf{S}_a$:

$$s_a(x,y) = s_e(x,y) - s_e(x,y+1) + c,\ 0 \leq x \leq L-1,\ 0 \leq y \leq R-2. \tag{17.33}$$

Die Konstante c muss dabei so gewählt werden, dass negative Grauwerte vermieden werden. Das Verfahren lässt sich am eindrucksvollsten wieder anhand des Testbildes 17.1-a darstellen. Bild 17.21-a zeigt die bildliche Darstellung des transformierten Originals, das aus den Grauwertdifferenzen zum jeweiligen linken Zeilennachbarn besteht. Bei der Differenzberechnung in (17.33) wurde $c = 127$ verwendet. Zur Verdeutlichung wurde das Differenzenbild etwas im Kontrast angereichert. Das dazugehörige Histogramm zeigt Bild 17.21-b. Daraus kann abgeleitet werden, dass die Differenzen etwa im Intervall $-23 \leq s_a(x,y) \leq 25$ liegen. Zur Darstellung dieser 48 verschiedenen Grauwerte werden dann 6 Bit pro Bildpunkt, also $G' = \{0, 1, ..., 63\}$ verwendet. Da die Standardabweichung der Differenzen sich zu $\sigma = 7.7$ berechnet, kann das Differenzenbild ohne allzu großen Informationsverlust auch mit weniger Grauwerten datenreduziert gespeichert werden, z.B. mit 4 Bit pro Bildpunkt, was einem maximalen Grauwertdifferenzbetrag von 16 Grauwerten entspricht.

Um die Rekonstruktion zu ermöglichen, wird z.B. in jeder Bildzeile der erste Bildpunkt mit seinem Originalgrauwert $s(x,0)$ gespeichert. Zu den in der Zeile folgenden Bildpunkten wird nur mehr die Differenz zum rechten Nachbarn gemäß (17.33) abgelegt. Bei der Rekonstruktion kann jetzt, ausgehend vom Originalgrauwert am Zeilenanfang, durch Addieren oder Subtrahieren der jeweiligen Differenz der ursprüngliche Grauwert ermittelt werden. Es genügt, auch nur den Originalgrauwert in der Position $(0,0)$ zu speichern. Beim Übergang in eine neue Bildzeile ist dann der Grauwert $s_a(x,0)$ die Differenz zum Originalgrauwert.

Das hier erläuterte Differenzbildungsverfahren ist nur ein einfaches Beispiel der Speicherung von Transformationskoeffizienten anstatt der Originalgrauwerte. Weitere Möglichkei-



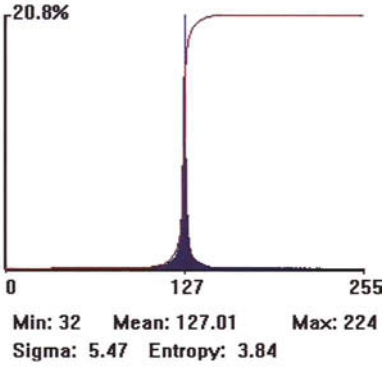


Bild 17.21: Differenzbildung (a) Bildliche Darstellung der Grauwertdifferenzen zum jeweils rechten Zeilennachbarn. Zur Verdeutlichung wurde das Bild etwas im Kontrast angehoben, (b) Histogramm zu (a). Zu den Grauwertdifferenzen wurde, um negative Grauwerte zu vermeiden, jeweils die Konstante 127 addiert.

ten sind z.B. die Speicherung der Fourierkoeffizienten (Abschnitt 21.4) oder die Speicherung der Koeffizienten einer Hauptkomponententransformation (Abschnitt 25.4).

Kapitel 18

Operationen im Ortsbereich

Die Grauwerttransformationen von Kapitel 17 waren rein bildpunktbezogen, d.h. es wurden keine Grauwertinformationen von benachbarten Bildpunkten verwendet. Es gibt nun aber auch viele Operationen, die, ausgehend von einem bestimmten Bildpunkt, eine Umgebung dieses Bildpunktes berücksichtigen. Da hier Umgebungen eines Bildpunktes mit den Ortskoordinaten (x, y) verwendet werden, bezeichnet man derartige Verfahren als *Operationen im Ortsbereich.*

Ein einfaches Beispiel ist die Mittelung der Grauwerte in einer Umgebung des Bildpunktes in der Position (x, y). Ein so erzeugtes Ausgabebild wird, verglichen mit dem Original, unschärfer wirken, da sich die Grauwertübergänge (Grauwertkanten) nicht mehr so deutlich ausprägen. Man kann auch sagen, dass das „Grauwertgebirge" geglättet wird. Zur Verdeutlichung dieser Operation ist in der Bildfolge 18.1-a bis 18.1-d ein Beispiel gegeben. Bild 18.1-a zeigt das Original eines Grauwertbildes, 18.1-b das geglättete Bild. Darunter ist in 18.1-c das Original als Projektion des dreidimensionalen „Grauwertgebirges" mit Hilfe einer Liniengrafik abgebildet. Schließlich zeigt 18.1-d das geglättete Bild als Liniengrafik.

Andere Operationen dieser Art werden verwendet, um die Grauwertkanten deutlicher hervorzuheben. Sie können damit ein Verarbeitungsschritt in Richtung Kantenextraktion sein. Wieder andere derartige Operatoren werden zur Elimination von gestört aufgezeichneten Bildpunkten oder Bildzeilen verwendet. Eine wichtige Klasse von Operatoren wird unter dem Oberbegriff der Rangordnungsoperatoren zusammengefasst. Beispiele dazu sind die Erosion und die Dilatation. Operationen dieser Art werden in Kapitel 19 behandelt.

Einige Operationen dieser Art können als digitale Filter im Ortsbereich interpretiert werden. Der Zusammenhang zwischen den digitalen Filtern im Ortsbereich und den digitalen Filtern im Frequenzbereich wird in Kapitel 21 dargestellt. Die Verfahren in diesem Kapitel können auch als Vorverarbeitungsschritte verstanden werden, wenn sich z.B. eine Bildsegmentierung (Kapitel 23) anschließt.

Nach einer Übersicht der Anwendungen, die zu den hier dargestellten Verfahren passen, folgt ein kurzes Grundlagenkapitel zu Filteroperationen im Ortsbereich. Auf der Grundlage dieser Verfahren werden im Folgenden die unterschiedlichsten Anwendungen erläutert.

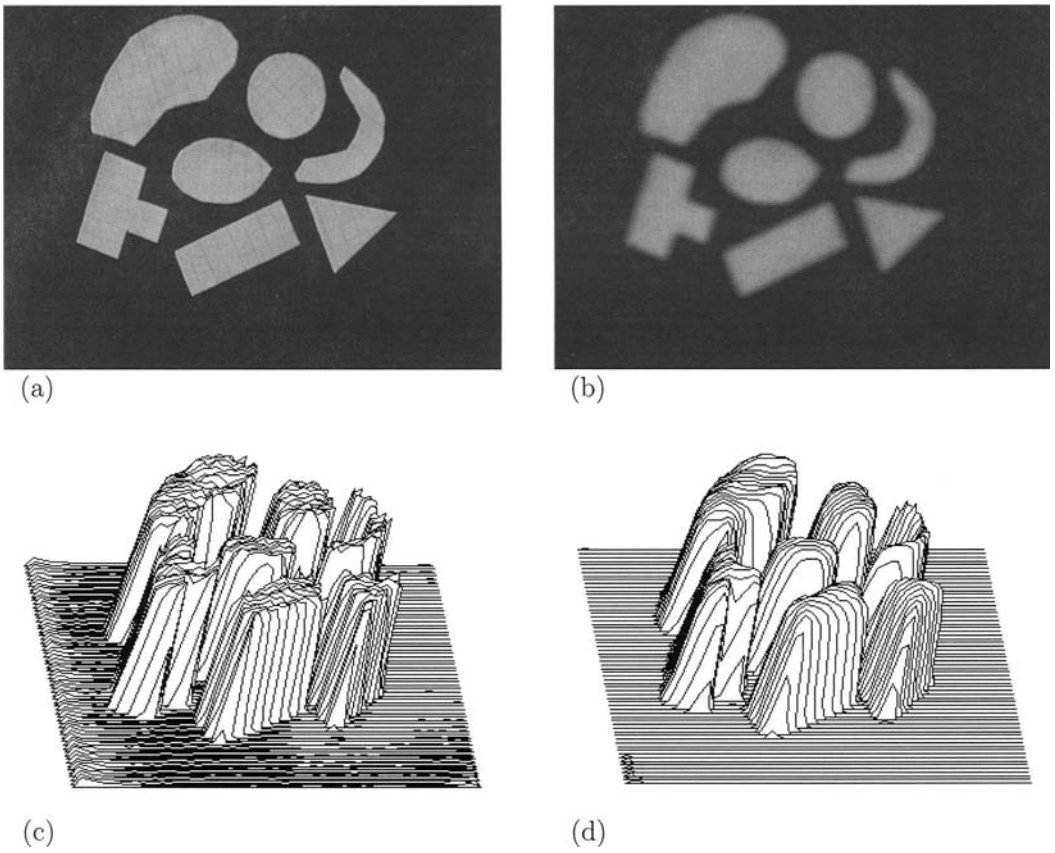


Bild 18.1: Beispiel zur Glättung des „Grauwertgebirges". (a) Original. (b) Geglättetes Original. (c) Darstellung des Originals als zweidimensionale Projektion mit Hilfe einer Liniengrafik. (d) Geglättetes Bild als Liniengrafik.

18.1 Anwendungen

Bilddaten, die mit Videosensoren aufgezeichnet werden, sind oft mehr oder weniger stark verrauscht. Dieses Rauschen ist bei einer rein visuellen Verarbeitung der Bilder in der Regel nicht störend und teilweise auch gar nicht sichtbar. Werden diese Bilddaten aber weiteren Verarbeitungsschritten, wie z.B. der Bildsegmentierung, zugeführt, so können sich die Einflüsse des Rauschens verfälschend auf die Ergebnisse auswirken.

In diesem Kapitel werden Verfahren erläutert, bei denen Glättungsoperatoren eingesetzt werden, um z.B. verrauschte Einzelbilder zu verbessern. Eine Bildverbesserung kann auch durch eine Bildakkumulation entlang der Zeitachse bei einer Bildsequenz erzielt werden.

Ein anderes Anwendungsgebiet ist das Herausarbeiten von Kanten und Linien, die in einem Bild enthalten sind. Die in diesem Kapitel erläuterten Differenzenoperatoren sind ein erster Schritt in diese Richtung. Die ausführliche Untersuchung von Kanten und Linien erfolgt in Kapitel 20.

Außerdem werden Algorithmen besprochen, die es erlauben, einzelne gestörte Pixel oder gestörte Bildzeilen zu eliminieren.

18.2 Grundlagen: Filteroperationen im Ortsbereich

Als Eingabebild wird ein einkanaliges Grauwertbild $\mathbf{S}_e = (s_e(x,y))$ verwendet. Die Bildpunkte $s_a(x,y)$ des Ausgabebildes $\mathbf{S}_a$ ergeben sich durch eine gewichtete, additive Verknüpfung des Bildpunktes $s_e(x,y)$ mit benachbarten Bildpunkten. Die Nachbarschaft und die Gewichte der Nachbarbildpunkte wird dabei durch eine Maske (*Faltungsmaske, Filterkern*) $\mathbf{H} = (h(u,v))$ festgelegt. Diese Operation, die einer *Faltung* des Bildes $\mathbf{S}_e$ mit der Maske $\mathbf{H}$ entspricht und auch als *digitale Filterung im Ortsbereich (Ortsraum)* bezeichnet wird, kann wie folgt geschrieben werden:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a : \tag{18.1}$$

$$s_a(x,y) = \frac{1}{m^2} \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} s_e(x+k-u, y+k-v) \cdot h(u,v),$$

wobei m die Größe der Maske $\mathbf{H}$ angibt. In der Praxis werden meistens quadratische Masken mit $m = 3, 5, 7, ...$ verwendet. Für den Parameter k gilt: $k = \frac{m-1}{2}$.

Für den Randbereich des Eingabebildes $\mathbf{S}_e$ ergeben sich mit (18.1) Probleme, da bei der Verknüpfung auch Bildpunkte verwendet werden, die außerhalb des Bildbereichs von $\mathbf{S}_e$ liegen (z.B. für $x = 0$ oder $y = 0$). Zur Lösung dieses Problems bieten sich unterschiedliche Möglichkeiten an.

Als erste Möglichkeit kann man bei der Berechnung des Ausgabebildes $\mathbf{S}_a$ und einer Maske mit z.B. $m = 3$ die Indizes x und y jeweils nur von eins bis zur vorletzten Zeile (Spalte) laufen lassen. Das hat aber zur Folge, dass das Ausgabebild hier um einen ein Pixel breiten Rand kleiner wird. Dieser Effekt, dass das Ausgabebild kleiner ist als das Eingabebild, ist bei manchen Anwendungen störend oder sogar unerwünscht.

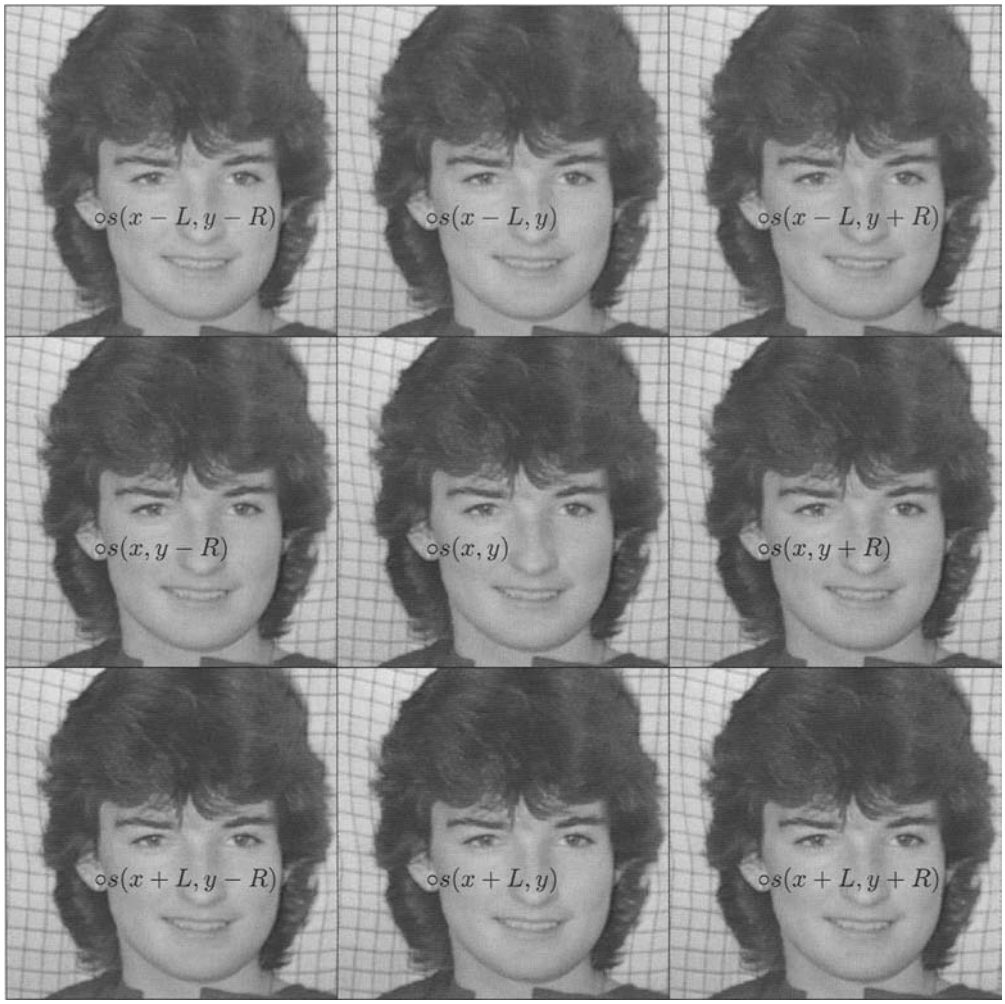


Bild 18.2: Periodische Fortsetzung des Bildes $\mathbf{S} = (s(x, y))$.

Als zweite Möglichkeit kann der Bildrand von $\mathbf{S}_e$ unverändert in das Ausgabebild $\mathbf{S}_a$ übernommen werden. Dann kann es passieren, dass bei weiteren Verarbeitungsschritten, z.B. bei einer Skalierung der Grauwerte (Kapitel 17), die Grauwerte in diesem Randbereich Störungen verursachen.

Drittens ist auch die Codierung der Bildpunkte des Randes mit einem konstanten Grauwert (etwa 0, 127 oder 255) denkbar.

Schließlich kann das Problem auch gelöst werden, indem man sich das Eingabebild $\mathbf{S}_e$ periodisch fortgesetzt vorstellt und so die fehlenden Pixel vom oberen bzw. unteren und

rechten bzw. linken Bildrand verwendet wie Bild 18.2 zeigt.

Diese Vorgehensweise ist vor dem Hintergrund der diskreten, zweidimensionalen Fouriertransformation sinnvoll, da dort eine solche periodische Fortsetzung vorausgesetzt wird (Kapitel 21). Allerdings erscheint eine solche periodische Fortsetzung nicht immer sinnvoll. Als Beispiel sei ein Bild mit einem dunklen Vordergrund im unteren Teil und einem hellen Hintergrund im oberen Teil betrachtet: Bei der periodischen Fortsetzung würden hier bei der Berechnung des Mittelwertes am oberen Bildrand die Grauwerte des dunklen unteren Bildbereichs erfasst werden. Es ist hier somit eine Spiegelung des Bildes sinnvoller.

Je nachdem, wie die Koeffizienten des Filterkerns $\mathbf{H}$ aussehen, ergeben sich unterschiedliche *digitale Filter im Ortsbereich*. Treten nur positive Koeffizienten auf, so spricht man auch von *Summenoperatoren*, bei positiven und negativen Koeffizienten von *Differenzenoperatoren*. Lassen sich die Koeffizienten als Stützstellen einer linearen Funktion auffassen, so werden die Operationen als *lineare digitale Filter im Ortsbereich*, anderenfalls als *nicht lineare digitale Filter im Ortsbereich* bezeichnet.

Im Folgenden werden einige Beispiele zu Filterkernen gegeben. Eine Operation mit dem Filterkern

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{18.2}$$

wird als *bewegter Mittelwert* bezeichnet. Er wurde in (18.2) mit $m = 3$ dargestellt. Er kann aber auch mit größeren Filterkernen angewendet werden. Der bewegte Mittelwert bewirkt eine Glättung der Grauwerte.

Es sind auch Varianten denkbar: Soll z.B. der Bildpunkt in der Position (x, y) doppelt gewichtet werden, so kann der zugehörige Koeffizient doppelt so groß wie die anderen gewählt werden. Wenn dabei der Mittelwert des gesamten Bildes nicht verändert werden soll, so ist darauf zu achten, dass die Summe der Filterkoeffizienten den Wert m^2 ergibt. Ein Beispiel dazu ist der folgende Filterkern:

$$\mathbf{H} = \begin{pmatrix} 0.9 & 0.9 & 0.9 \\ 0.9 & 1.8 & 0.9 \\ 0.9 & 0.9 & 0.9 \end{pmatrix} \tag{18.3}$$

Der nichtlineare *Gauß-Tiefpass* hat den Filterkern

$$\mathbf{H} = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}. \tag{18.4}$$

Der Name rührt daher, dass seine Elemente eine zweidimesionale Gauß'sche Glocke grob annähern. Wenn auch hier die Operation den Mittelwert des Bildes erhalten soll, so müssen die Elemente dieses Filterkerns mit $\frac{9}{16}$ multipliziert werden.

Differenzenoperatoren werden in der Regel verwendet, um Grauwertübergänge deutlicher herauszuarbeiten oder Kanten zu extrahieren. Bei der Berechnung kann hier auf

den Normierungsfaktor $\frac{1}{m^2}$ in (18.1) verzichtet werden. Wenn das Ergebnis wieder in die Grauwertmenge $G = \{0, ..., 255\}$ abgebildet werden soll, so ist zu beachten, dass die Werte geeignet skaliert werden.

Wird eine partielle Differentiation in Zeilen- und Spaltenrichtung nachgebildet, so erhält man die Filterkerne

$$\mathbf{H}_x = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \text{und} \quad \mathbf{H}_y = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix}. \tag{18.5}$$

Nach dieser Darstellung der Grundprinzipien werden in den folgenden Abschnitten die unterschiedlichen Verfahren ausführlich beschrieben.

18.3 Glätten der Grauwerte eines Bildes

Die einfachste Form der Glättung der Grauwerte eines Bildes ist die Berechnung des Mittelwertes einer vorgegebenen Umgebung eines Bildpunktes. Aus Gründen der Symmetrie wird meistens eine quadratische Umgebung mit $3 \cdot 3, 5 \cdot 5, 7 \cdot 7, ...$ Bildpunkten verwendet. Es sei $\mathbf{S}_e = (s_e(x, y))$ ein Grauwertbild mit $G = \{0, 1, ..., 255\}$. Die Größe des quadratischen Umgebungsfensters wird mit m bezeichnet. Der Operator

$$\mathbf{S}_e \rightarrow \mathbf{S}_a : s_a(x, y) = \frac{1}{m^2} \sum_{u=-k}^{+k} \sum_{v=-k}^{+k} s_e(x - u, y - v), \tag{18.6}$$

mit $m = 3, 5, 7, ...$ und $k = (m-1)/2$, weist dem Bildpunkt $s_a(x, y)$ den Mittelwert der $m \cdot m$-Umgebung des Bildpunktes $s_e(x, y)$ in $\mathbf{S}_e$ als neuen Grauwert zu. Für $m = 3$ werden z.B. folgende Bildpunkte in die Berechnung mit einbezogen:

$$\begin{array}{lll} s_e(x-1, y-1) & s_e(x-1, y) & s_e(x-1, y+1) \\ s_e(x, y-1) & s_e(x, y) & s_e(x, y+1) \\ s_e(x+1, y-1) & s_e(x+1, y) & s_e(x+1, y+1). \end{array}$$

Die Probleme, die sich für den Randbereich ergeben, können mit den unterschiedlichen Vorgehensweisen gelöst werden, die in Abschnitt 18.2 dargestellt wurden.

Bilder, die mit diesem Operator, der auch *bewegter Mittelwert* oder *gleitender Mittelwert* (*moving average*) genannt wird, verarbeitet werden, wirken im Vergleich zum Original weicher oder etwas unschärfer. In Bildbereichen mit homogenen Grauwerten hat der bewegte Mittelwert keine Auswirkung, wogegen Grauwertübergänge geglättet werden. Ein Beispiel zeigt die Bildfolge 18.3.

Der Mittelwert von $\mathbf{S}_a$ ist derselbe wie der von $\mathbf{S}_e$, während die Streuung von $\mathbf{S}_a$ kleiner wird, was z.B. bedeuten kann, dass im Bild $\mathbf{S}_e$ überlagertes Rauschen verringert wird. Dies läßt sich auch rechnerisch nachweisen. Für die Grauwerte von $\mathbf{S}_e$ wird angenommen, dass ihnen ein Rauschanteil additiv überlagert ist:

$$s_e(x, y) = s_0(x, y) + z(x, y). \tag{18.7}$$

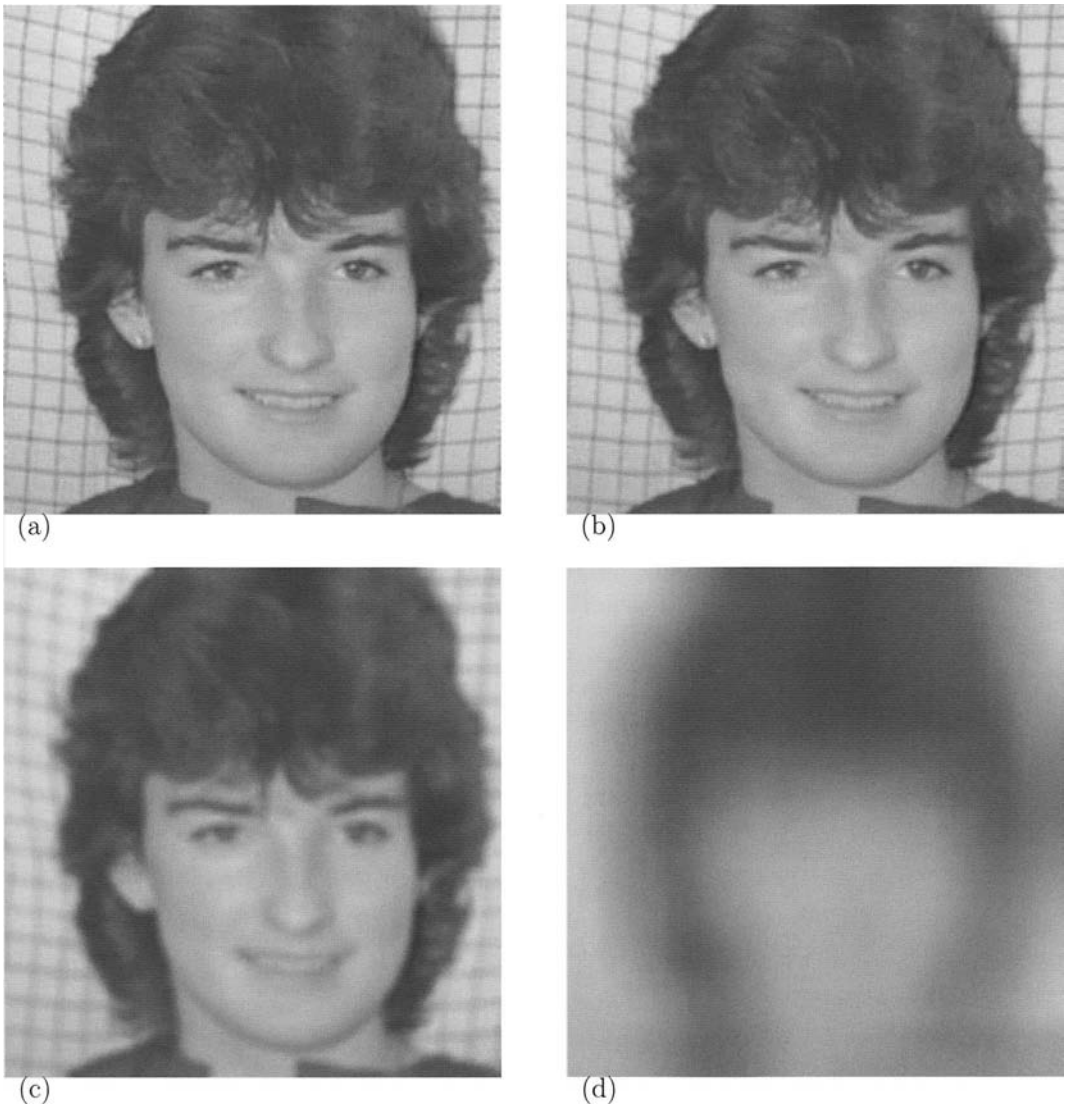


Bild 18.3: Beispiele zum bewegten Mittelwert. (a) Original. (b) Umgebung mit $3 \cdot 3$ Bildpunkten. (c) Umgebung mit $11 \cdot 11$ Bildpunkten. (d) Umgebung mit $101 \cdot 101$ Bildpunkten. Bei allen Beispielen wurde der Randbereich durch eine Spiegelung des Originals berechnet.

Die Rauschanteile $z(x,y)$ werden durch Zufallsvariable $Z_{(x,y)}$ beschrieben, die den Erwartungswert 0 und die Streuung σ^2 haben. Die Streuung des Rauschens wird auch als *Rauschleistung* bezeichnet. Die Zufallsvariablen seien unabhängig und unkorreliert. Damit läßt sich für $s_a(x,y)$ schreiben:

$$s_a(x,y) = \frac{1}{m^2}\Big(\sum_{u=-k}^{+k} \sum_{v=-k}^{+k} s_0(x-u, y-v) + \sum_{u=-k}^{+k} \sum_{v=-k}^{+k} z(x-u, y-v)\Big). \tag{18.8}$$

Für den Erwartungswert und die Streuung der Zufallsvariablen

$$Z = \frac{1}{m^2} \sum_{u=-k}^{+k} \sum_{v=-k}^{+k} Z_{x-u,y-v} \tag{18.9}$$

ergibt sich:

$$E(Z) = E\Big(\frac{1}{m^2} \sum_{u=-k}^{+k} \sum_{v=-k}^{+k} Z_{x-u,y-v}\Big) = \frac{1}{m^2} \sum_{u=-k}^{+k} \sum_{v=-k}^{+k} E\Big(Z_{x-u,y-v}\Big) = 0. \tag{18.10}$$

$$E(Z^2) = E\Big(\Big(\frac{1}{m^2} \sum_{u=-k}^{+k} \sum_{v=-k}^{+k} Z_{x-u,y-v}\Big)^2\Big) = \tag{18.11}$$

$$= \frac{1}{m^4} E\Big(\Big(\sum_{u=-k}^{+k} \sum_{v=-k}^{+k} Z_{x-u,y-v}\Big)^2\Big) =$$

$$= \frac{1}{m^4} \sum_{u=-k}^{+k} \sum_{v=-k}^{+k} E\Big(\Big(Z_{x-u,y-v}\Big)^2\Big) = \frac{1}{m^2} \cdot \sigma^2.$$

Für die Umformung in (18.11) wird die Unabhängigkeit und die Unkorreliertheit von zwei Zufallsvariablen $Z_{x-u_1,y-v_1}$ und $Z_{(x-u_2,y-v_2)}$ verwendet:

$$E\Big(Z_{(x-u_1,y-v_1)} Z_{(x-u_2,y-v_2)}\Big) = E\Big(Z_{(x-u_1,y-v_1)}\Big) \cdot E\Big(Z_{(x-u_2,y-v_2)}\Big) = 0. \tag{18.12}$$

Die Beziehungen (18.10) und (18.11) bestätigen die obigen Vermutungen: Der Erwartungswert bleibt erhalten und die Streuung des Rauschanteils wird durch den Faktor $1/m^2$ reduziert.

Der bewegte Mittelwert kann auch als Faltungsoperation, wie in Abschnitt 18.2 erläutert, aufgefasst werden. Die Faltung zweier Funktionen $s(x,y)$ und $h(x,y)$ der diskreten Variablen x und y ist definiert als:

$$\sum_{u=-\infty}^{+\infty} \sum_{v=-\infty}^{+\infty} s(x-u, y-v) \cdot h(u,v). \tag{18.13}$$

Da beim bewegten Mittelwert nur über eine endliche $m \cdot m$-Umgebung des Bildes $\mathbf{S}_e$ gemittelt wird, kann (18.6) als Faltung des Bildausschnittes mit einer Maske $\mathbf{H} = (h(u,v))$ geschrieben werden:

$$s_a(x,y) = \frac{1}{m^2} \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} s_e(x+k-u, y+k-v) \cdot h(u,v) \tag{18.14}$$

mit $k = (m-1)/2$ und $m = 3, 5, 7, ...$ Die Maske $\mathbf{H}$ hat beim bewegten Mittelwert mit z.B. $m = 3$ das Aussehen:

$$\mathbf{H} = \big(h(u,v)\big) = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}. \tag{18.15}$$

Bei der Berechnung des bewegten Mittelwertes kann berücksichtigt werden, dass beim Übergang von der Position (x,y) zu $(x,y+1)$ nicht die ganze Summe neu berechnet werden muss, sondern der neue Mittelwert durch Subtraktion des Mittelwertes des Spaltenvektors

$$\big(s_e(x-k, y-k), s_e(x-k+1, y-k), ..., s_e(x+k, y-k)\big)^T$$

und Addition des Mittelwertes des Spaltenvektors

$$\big(s_e(x-k, y+1+k), s_e(x-k+1, y+1+k), ..., s_e(x+k, y+1+k)\big)^T$$

berechnet werden kann.

Bei der Implementierung des bewegten Mittelwertes können diese speziellen Eigenschaften des Filterkerns ausgenützt werden. Ein Algorithmus dazu ist im Folgenden angegeben.

A18.1: Bewegter Mittelwert.

Voraussetzungen und Bemerkungen:

- ◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild mit $G = \{0, 1, ..., 255\}$ als Grauwertmenge (Eingabebild).
- ◇ $\mathbf{S}_a = (s_a(x,y))$ ein einkanaliges Grauwertbild $G = \{0, 1, ..., 255\}$ als Grauwertmenge (Ausgabebild).
- ◇ Die spezielle Behandlung des Bildrandes, wie in Abschnitt 18.2 dargestellt, wird hier nicht berücksichtigt.

Algorithmus:

(a) Für alle Bildzeilen $x = 0, 1, ..., L-1$ des Bildes $\mathbf{S}_e = (s_e(x,y))$:

(aa) Berechne die folgenden Spaltensummen für $j = -k, ..., +k$ und $k = \frac{m-1}{2}$:

$$s_j = \sum_{i=-k}^{+k} s_e(x+i, j);$$

(ab) Für alle Bildspalten $y = 0, 1, ..., R-1$ des Bildes $\mathbf{S}_e = (s_e(x, y))$:

(aba) Berechne den Grauwert des Ausgabebildpunktes:

$$s_a(x, y) = \frac{1}{m^2} \sum_{j=-k}^{+k} s_j;$$

(abb) Berechne eine neue Spaltensumme:

$$s_{k+1} = \sum_{i=-k}^{+k} s_e(x+i, y+k+1);$$

(abc) Verschiebe die Spaltensummen. Für $j = -k, ..., +k$:

$$s_j = s_{j+1};$$

Ende des Algorithmus

Die einzelnen Elemente von $\mathbf{H}$ geben das Gewicht an, mit dem die Grauwerte der $m \cdot m$-Umgebung in die Summation eingehen. Verwendet man für $\mathbf{H}$ die Maske

$$\mathbf{H} = \big(h(u, v)\big) = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{18.16}$$

so wird der Grauwert in der Mitte der Umgebung doppelt bewertet. Allerdings wäre dann der Mittelwert von $\mathbf{S}_a$ anders als der von $\mathbf{S}_e$. Der Mittelwert bleibt erhalten, wenn die Elemente von $\mathbf{H}$ so normiert sind, dass ihre Summe m^2 ergibt. Statt (18.16) ist dann die Maske

$$\mathbf{H} = \big(h(u, v)\big) = \begin{pmatrix} 0.9 & 0.9 & 0.9 \\ 0.9 & 1.8 & 0.9 \\ 0.9 & 0.9 & 0.9 \end{pmatrix} \tag{18.17}$$

zu verwenden.

Der bewegte Mittelwert wird in der Praxis oft zur Glättung der Grauwerte vor anderen Operationen eingesetzt. So vermittelt z.B. die Äquidensiten- oder Pseudofarbdarstellung eines geglätteten Bildes einen besseren visuellen Eindruck als die entsprechende Darstellung des Originals. Allerdings geht bei der Mittelwertbildung Bildinformation verloren, was aber bei einer anschließenden Äquidensiten- oder Pseudofarbdarstellung in der Regel keine Rolle spielt, da hier ja sowieso Grauwertintervalle zu einem Grauwert zusammengefasst werden.

Je nach der Größe von m werden durch den bewegten Mittelwert mehr oder weniger große Umgebungen des Bildpunktes in der Position (x, y) in die Mittelung einbezogen. Ein Nachteil ist dabei allerdings, dass Grauwertübergänge, also Kanten, geglättet und somit

unschärfer werden. Die Bilder 18.4-a, 18.4-b, 18.4-c und 18.4-d zeigen ein Beispiel hierzu. Bild 18.4-d ist aus der Differenz zwischen dem Original und dem zehnmal hintereinander gefilterten Bild 18.4-c entstanden. Man kann so die Bildinformation darstellen, die durch die Filteroperation verloren ging: Das sind aber gerade die deutlichen Grauwertübergänge (Kanten). Dieser Sachverhalt wird bei den Laplace-Pyramiden (Kapitel 28) verwendet.

In Kapitel 17 wurde im Rahmen der Binärbilderzeugung, der Äquidensitendarstellung, sowie der Pseudofarbdarstellung bereits auf die Verwendung des bewegten Mittelwertes als Vorverarbeitungsschritt hingewiesen.

Bei der Verwendung anderer Filterkerne, z.B. dem Gauß-Tiefpass von (18.4), erhält man ähnliche Ergebnisse.

18.4 Differenzenoperatoren

Werden für die Elemente der Maske **H** in (18.14) auch negative Werte zugelassen, so ergeben sich *Differenzenoperatoren.* Ist die Summe der Elemente von **H** gleich 0, so berechnet sich mit (18.14) für homogene Bildbereiche der Wert 0. Bei Grauwertübergängen liefert der Operator eine Maßzahl für die „Stärke" des Übergangs. Zur praktischen Berechnung kann in (18.14) auf den Normierungsfaktor $1/m^2$ verzichtet werden.

Eine einfache Anwendung ist die Berechnung der Differenzen benachbarter Bildpunkte in Zeilen- oder in Spaltenrichtung. Ist $s(x)$ eine stetige Funktion, so ist die erste Ableitung von $s(x)$ definiert als

$$\frac{\mathrm{d}s}{\mathrm{d}x} = \lim_{\Delta x \to 0} \frac{s(x + \Delta x) - s(x)}{\Delta x}. \tag{18.18}$$

Die sinngemäße Differenzenbildung bei einer Funktion $s(x)$ mit diskretem x lautet:

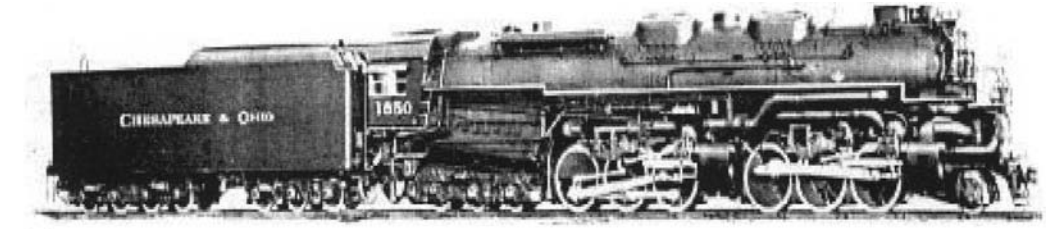
$$\frac{s(x+1) - s(x)}{1} = s(x+1) - s(x). \tag{18.19}$$

Die Maske **H** in (18.14) hat damit das folgende Aussehen:

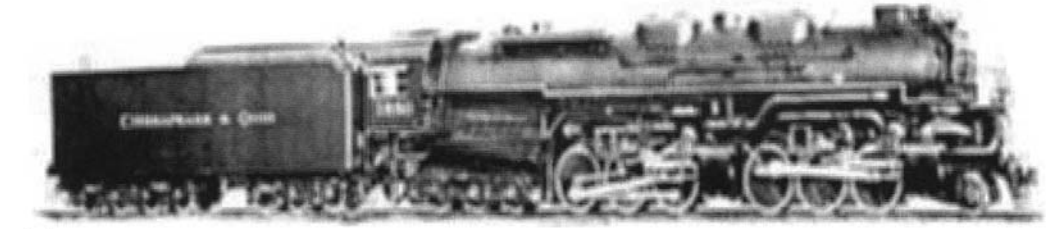
$$\mathbf{H}_x = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \qquad \mathbf{H}_y = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix}. \tag{18.20}$$

Differenzenoperatoren, die die beiden Masken $\mathbf{H}_x$ oder $\mathbf{H}_y$ verwenden, sind also die diskreten Analoga zur Berechnung der ersten Ableitung in x- oder y-Richtung einer stetigen Funktion zweier Variablen.

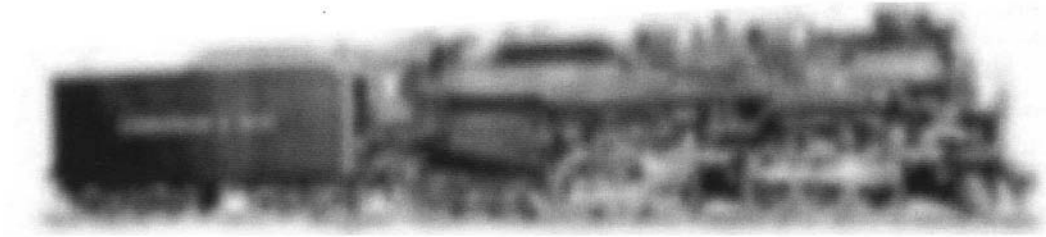
Wie schon erwähnt, werden Differenzenoperatoren vor allem da eingesetzt, wo es darum geht, Grauwertkanten und Linien aus einem Bild zu extrahieren. Die einfache Differenzbildung mit (18.20) ist dazu in der Praxis zu anfällig gegenüber Störungen, etwa durch



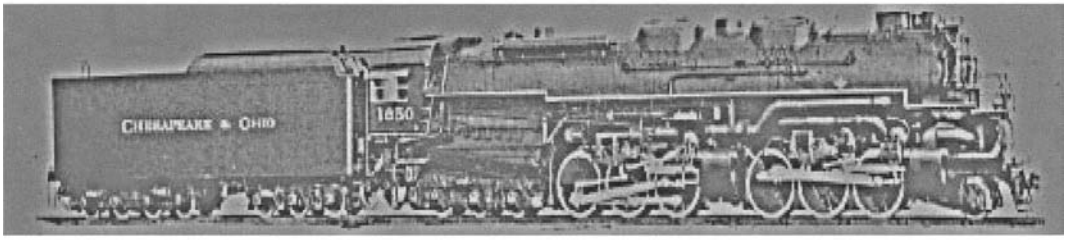
(a)



(b)



(c)



(d)

Bild 18.4: (a) Originalbild „Chesapeake & Ohio". (b) Bewegter Mittelwert($m = 3$): Man sieht deutlich, dass Kanten unschärfer sind als im Original. (c) Hier wurde der Operator zehnmal hintereinander angewendet. (d) Differenzbild zwischen Original und dem zehnmal gefilterten Bild. Man sieht hier die Bildinhalte, die durch die Filterung verloren gingen.

(a) (b)

Bild 18.5: Differenzenoperator (Sobeloperator), angewendet auf das Testbild. (a) Ergebnis der $\mathbf{H}_x$-Maske. (b) Ergebnis der $\mathbf{H}_y$-Maske.

Rauschen. Aus diesem Grund werden zur Differenzbildung auch die Grauwerte weiterer Nachbarn verwendet. Statt (18.20) könnte z.B. auch verwendet werden:

$$\mathbf{H}_x = \begin{pmatrix} 1 & 1 & 1 \\ -1 & -1 & -1 \\ 0 & 0 & 0 \end{pmatrix} \qquad \mathbf{H}_y = \begin{pmatrix} 1 & -1 & 0 \\ 1 & -1 & 0 \\ 1 & -1 & 0 \end{pmatrix}. \tag{18.21}$$

Bessere Ergebnisse liefert der *Sobeloperator* :

$$\mathbf{H}_x = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \qquad \mathbf{H}_y = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}, \tag{18.22}$$

da hier die Differenzbildung jeweils zur übernächsten Zeile (Spalte) berechnet wird und kleine Störungen benachbarter Zeilen (Spalten) nicht in das Ergebnis eingehen. Die Rauschminderung wird hier durch Mittelung quer zur Richtung des Gradienten bewirkt. Die Anwendung dieser beiden Masken auf das Testbild zeigen Bild 18.5-a und Bild 18.5-b. Für die bildliche Darstellung wurde zu den Differenzen der Grauwert 127 addiert. Außerdem wurde zur besseren bildlichen Darstellung der Bildkontrast etwas angehoben.

Die bis jetzt besprochenen Differenzenoperatoren sprechen am stärksten auf horizontal oder vertikal verlaufende Grauwertkanten an, sie sind also richtungsabhängig. Ein weiterer

richtungsabhängiger Differenzenoperator ist der *Kompassgradient*, der aus insgesamt acht Masken für acht Richtungen besteht:

$$\mathbf{H}_{\text{Ost}} = \begin{pmatrix} -1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & 1 & 1 \end{pmatrix} \mathbf{H}_{\text{Nordost}} = \begin{pmatrix} 1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & -1 & 1 \end{pmatrix} \tag{18.23}$$

$$\mathbf{H}_{\text{Nord}} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & -1 & -1 \end{pmatrix} \mathbf{H}_{\text{Nordwest}} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & -2 & -1 \\ 1 & -1 & -1 \end{pmatrix}$$

$$\mathbf{H}_{\text{West}} = \begin{pmatrix} 1 & 1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & -1 \end{pmatrix} \mathbf{H}_{\text{Südwest}} = \begin{pmatrix} 1 & -1 & -1 \\ 1 & -2 & -1 \\ 1 & 1 & 1 \end{pmatrix}$$

$$\mathbf{H}_{\text{Süd}} = \begin{pmatrix} -1 & -1 & -1 \\ 1 & -2 & 1 \\ 1 & 1 & 1 \end{pmatrix} \mathbf{H}_{\text{Südost}} = \begin{pmatrix} -1 & -1 & 1 \\ -1 & -2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

Der *Gradient* einer stetigen Funktion zweier Variablen ist:

$$\left(\frac{\partial s(x,y)}{\partial x}, \frac{\partial s(x,y)}{\partial y} \right)^T. \tag{18.24}$$

Der *Betrag* des Gradienten ist:

$$\sqrt{\left(\frac{\partial s(x,y)}{\partial x} \right)^2 + \left(\frac{\partial s(x,y)}{\partial y} \right)^2} \tag{18.25}$$

und die *Richtung* des Gradienten berechnet sich aus:

$$\left(\frac{\partial s(x,y)}{\partial y} \right) \Big/ \left(\frac{\partial s(x,y)}{\partial x} \right). \tag{18.26}$$

Im diskreten Fall werden z.B. mit den Masken des Sobeloperators (18.22) die Funktionen $s_x(x,y)$ und $s_y(x,y)$ berechnet und anschließend Betrag und Richtung des Gradienten:

$$\sqrt{s_x(x,y)^2 + s_y(x,y)^2} \text{ und } s_y(x,y) \big/ s_x(x,y). \tag{18.27}$$

Soll eine bildliche Darstellung des Betrags des Gradienten erfolgen, so ist in (18.27) zu beachten, dass die berechneten Werte in die Grauwertmenge G abgebildet werden. Die Bilder 18.6-a/b zeigen den so aus Bild 18.5-a und Bild 18.5-b berechneten Gradienten (Betrag und Richtung).

(a) (b)

Bild 18.6: Gradienten zum Testbild nach der Anwendung des Sobeloperators. (a) Betrag. (b) Richtung. Die Winkel wurden so codiert, dass gleiche Richtungen durch den gleichen Grauwert dargestellt sind.

Richtungsunabhängige Differenzenoperatoren liegen vor, wenn die Maske $\mathbf{H}$ punktsymmetrisch ist. Am häufigsten wird hier der *Laplace-Operator* verwendet. Im kontinuierlichen Fall ist er definiert als:

$$\frac{\partial s^2(x,y)}{\partial x^2} + \frac{\partial s^2(x,y)}{\partial y^2}. \tag{18.28}$$

Die Maske für sein diskretes Analogon ist:

$$\mathbf{H} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}. \tag{18.29}$$

Oft werden statt dessen auch

$$\mathbf{H} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \text{ oder } \mathbf{H} = \begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix} \tag{18.30}$$

verwendet.

Der Laplace-Operator, der etwas anfälliger gegenüber Bildstörungen ist, liefert in Verbindung mit (18.14) in einem Durchgang Maßzahlen für den Betrag des Grauwertgradienten.

Allerdings ist zu beachten, dass der Laplace-Operator als diskrete Nachbildung der 2. Ableitung, angewendet auf eine beliebig geneigte Ebene, den Wert 0 ergibt, obwohl die Gradienten dieser Fläche nicht notwendig den Werte 0 haben.

Die Differenzenoperatoren werden, wie schon oben erwähnt, vor allem zur Extraktion von Grauwertkanten eingesetzt. Sie sind hier nur ein erster Schritt in einer Folge von weiteren Verarbeitungen, wie z.B. Relaxation, Schwellwertbildung oder Linienverfolgung. Diese Problemkreise werden in Kapitel 20 noch näher untersucht.

18.5 Elimination isolierter Bildpunkte

Manchmal treten in Bildern einzelne Pixelstörungen auf. Ein Beispiel dazu sind Luftbilddaten aus der Fernerkundung, bei denen durch störende Einflüsse auf den Sensor gelegentlich einzelne Bildpunkte übersteuert werden. Eine derartige Störung könnte etwa wie folgt aussehen:

...	...	...	...	...	...	...
...	18	22	27	24	16	...
...	19	21	28	25	20	...
...	18	21	255	24	31	...
...	17	24	26	25	32	...
...	14	19	22	19	25	...
...	...	...	...	...	...	...

In einer Umgebung von Bildpunkten mit einem Mittelwert um etwa 20 hat ein Bildpunkt den Grauwert 255.

Mit einem bewegten Mittelwert oder ähnlichen Glättungsoperatoren (Abschnitt 18.3) wird das gesamte Bild unschärfer und der fehlerhafte Wert wird etwas verkleinert. Bei der Verwendung eines Medianfilters (Kapitel 19) kann die Bildpunktstörung eliminiert werden. Die Grauwerte der Umgebung der Störung werden dabei aber auch verändert, was bei manchen Anwendungen nicht wünschenswert ist.

Um den gestörten Bildpunkt zu erkennen und zu eliminieren, ohne dabei die Grauwerte der Umgebung zu verändern, vergleicht man den Grauwert jedes Bildpunktes mit dem Mittelwert m seiner 8-Nachbarn. Übersteigt der Unterschied einen bestimmten Schwellwert c, so wird der Bildpunkt als gestört erkannt und durch den Mittelwert m ersetzt:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a : \tag{18.31}$$

$$s_a(x,y) = \begin{cases} m, & \text{falls } \mid m - s_e(x,y) \mid > c, \\ s_e(x,y), & \text{sonst.} \end{cases}$$

Dabei ist

$$m = \frac{1}{8} \sum_{u=0}^{2} \sum_{v=0}^{2} s_e(x+1-u, y+1-v) \cdot h(u,v). \tag{18.32}$$

mit der Filtermaske

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{pmatrix}. \tag{18.33}$$

Dieses Verfahren kann immer eingesetzt werden, wenn sich die Bildstörungen im Grauwert deutlich von den korrekten Bildpunkten abheben. Wird der Schwellwert c zu klein gewählt, so ist die Wirkung dieses Operators mit der des bewegten Mittelwertes (Abschnitt 18.3) zu vergleichen.

Durch sinngemäße Modifikation der verwendeten Maske können auch Bildstörungen, die mehr als einen Bildpunkt verfälscht haben, mit diesem Operator detektiert und eliminiert werden. Ist z.B. bekannt, dass immer zwei aufeinanderfolgende Bildpunkte in einer Bildzeile gestört sind, so kann die folgende Maske eingesetzt werden:

$$\mathbf{H} = \big(h(u,v)\big) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}. \tag{18.34}$$

Es ist aber zu bemerken, dass bei ausgedehnten Bildstörungen die Effektivität dieses Operators mehr und mehr abnimmt.

18.6 Elimination gestörter Bildzeilen

Manchmal treten in Bildern gestörte Bildzeilen auf, die entweder ganz weiß oder schwarz sind oder ein zufälliges Grauwertmuster aufweisen. Die Detektion solcher Bildzeilen kann mit Hilfe der Berechnung der Korrelation von aufeinander folgenden Bildzeilen durchgeführt werden: Bei natürlichen Bildern ist der Übergang der Grauwerte von einer Bildzeile zur nächsten in der Regel nicht allzu abrupt, was bedeutet, dass der Korrelationskoeffizient zwischen benachbarten, ungestörten Bildzeilen relativ groß ist. Dagegen ist er bei einer korrekten und einer gestörten Bildzeile klein. Damit ist also die folgende Vorgehensweise angebracht:

- Die Verarbeitung wird mit einer korrekten Bildzeile begonnen.
- Zu je zwei aufeinanderfolgenden Bildzeilen x und $x+1$ wird der Korrelationskoeffizient $r_{x,x+1}$ berechnet.
- Liegt der Betrag $\mid r_{x,x+1} \mid$ über einem gewissen Schwellwert c, so wird angenommen, dass die Bildzeile $x+1$ korrekt ist.
- Ist der Betrag $\mid r_{x,x+1} \mid < c$, so wird die Bildzeile $x+1$ als fehlerhaft betrachtet.

Der Korrelationskoeffizient für die Bildzeilen des Eingabebildes $\mathbf{S}_e$ wird dazu wie folgt berechnet. Es sei $v_{x,x+1}$, die Kovarianz der Bildzeilen x und $x+1$:

$$v_{x,x+1} = \frac{1}{R} \sum_{y=0}^{R-1} \Big(s_e(x,y) - m_x\Big)\Big(s_e(x+1,y) - m_{x+1}\Big). \tag{18.35}$$

m_x und m_{x+1} sind die mittleren Grauwerte der Bildzeilen x und $x+1$. Durch Normierung der Kovarianz durch die Wurzel des Produktes der Streuungen $v_{x,x}$ und $v_{x+1,x+1}$ erhält man den Korrelationskoeffizienten:

$$r_{x,x+1} = \frac{v_{x,x+1}}{\sqrt{v_{x,x} v_{x+1,x+1}}}. \tag{18.36}$$

Der Betrag $\mid r_{x,x+1} \mid$ liegt zwischen 0 und 1, wobei sich starke Korrelation durch Werte nahe der 1 ausdrückt. Zur Bestimmung eines passenden Schwellwertes c ist zu sagen, dass er am besten durch Testen am jeweiligen Bildmaterial zu ermitteln ist.

A18.2: Elimination gestörter Bildzeilen.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild mit $G = \{0,1,...,255\}$ als Grauwertmenge (Eingabebild).

◇ $\mathbf{S}_a = (s_a(x,y))$ ein einkanaliges Grauwertbild mit $G = \{0,1,...,255\}$ als Grauwertmenge (Ausgabebild).

◇ Die Verarbeitung wird mit einer korrekten Bildzeile begonnen.

Algorithmus:

(a) Für je zwei aufeinander folgende Bildzeilen x und $x+1$ wird der Korrelationskoeffizient $r_{x,x+1}$ berechnet.

(aa) Liegt der Betrag $\mid r_{x,x+1]} \mid$ über einem Schwellwert c, so wird angenommen, dass die Bildzeile $x+1$ korrekt ist.

(ab) Ist der Betrag $\mid r_{x,x+1]} \mid < c$, so wird die Bildzeile $x+1$ als fehlerhaft betrachtet und z.B. durch die Zeile x ersetzt.

Ende des Algorithmus

Bei der praktischen Anwendung dieses Verfahrens sind noch einige Punkte zu berücksichtigen. Wie schon oben erwähnt, muss die Startbildzeile korrekt sein. Eine passende Startzeile kann entweder interaktiv festgelegt oder durch den Vergleich der Grauwerte von benachbarten Bildpunkten ermittelt werden. Falls gestörte Bildzeilen auch direkt aufeinanderfolgend auftreten können, muss berücksichtigt werden, dass zwei gestörte Bildzeilen,

die z.B. beide ganz weiß oder schwarz sind, maximal korreliert sind. Andererseits werden gestörte Bildzeilen, die zufällige Grauwertmuster aufweisen, in der Regel geringer korreliert sein.

Abschließend ist noch zu untersuchen, wie eine detektierte, gestörte Bildzeile eliminiert werden kann. Die einfachste Methode ist die Ersetzung dieser Bildzeile durch die vorhergehende korrekte, die zur Berechnung des Korrelationskoeffizienten verwendet wurde. Bei dieser Methode kann allerdings bei schräg verlaufenden Grauwertkanten eine ungewünschte Stufung auftreten. Andere Methoden interpolieren die Grauwerte zwischen zwei oder mehreren benachbarten Bildzeilen.

18.7 Bildakkumulation bei Bildfolgen

Wenn Bildfolgen, etwa der Form $\mathbf{S} = (s(x, y, t))$, $t = 0, 1, 2, ...$, vorliegen, so können Rauschanteile auch durch eine Summation entlang der Zeitachse vermindert werden:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a : \tag{18.37}$$

$$s_a(x, y) = \frac{1}{T} \sum_{t=0}^{T-1} s_e(x, y, t);$$

Hier wird über eine Bildsequenz von T Einzelbildern akkumuliert. Es muss allerdings vorausgesetzt werden, dass sich im Bildausschnitt nichts bewegt. Der Vorteil gegenüber einem bewegten Mittelwert über ein Einzelbild liegt darin, dass durch die Akkumulation entlang der Zeitachse zwar Rauschen verringert werden kann, jedoch Kanten im Bild erhalten bleiben.

Bei sehr stark verrauschten Bildsignalen, bei denen sich die eigentliche Bildinformation vom Rauschen kaum mehr unterscheiden läßt, kann durch die Akkumulation über einen genügend langen Zeitraum eine deutlichere Hervorhebung der Bildinformation erzielt werden. In der Radioastronomie werden diese Techniken der Bildakkumulation erfolgreich eingesetzt. Über einen längeren Zeitraum ist hier allerdings die Voraussetzung, dass sich das Beobachtungsgebiet nicht bewegt, nicht gegeben. Da aber die Bewegung bekannt ist, kann sie durch ein Nachführen des Sensors (des Radioteleskops) kompensiert werden.

In der Medizin wird die Bildakkumulation ebenfalls verwendet, z.B. in der Angiocardiografie. Hier werden durch die Kombination von Bildakkumulation, Verwendung von Kontrastmitteln und der Differenzbildung von akkumulierten Zwischenergebnissen gute Bildverbesserungserfolge erzielt. Eine ausführliche Darstellung der Auswertung von Bildfolgen ist in Kapitel 25 zusammengestellt.

Kapitel 19

Rangordnungsoperatoren und mathematische Morphologie

19.1 Anwendungen

In den folgenden Abschnitten werden Verfahren beschrieben, die in den Bereich der Rangordnungsoperatoren und der mathematischen Morphologie gehören. Nach einem Grundlagenabschnitt über mathematische Morphologie wird besprochen, wie sich z.B. vereinzelte Bildpunkte eliminieren lassen oder wie sich der Rand eines Segments ermitteln lässt. Auch bestimmte Formen können durch die Kombination verschiedender morphologischer Operationen gezielt extrahiert werden.

19.2 Grundlagen: Mathematische Morphologie

Die mathematische Morphologie ist eine Theorie, die sich mit der Verknüpfung von Mengen befasst. Angewendet auf Bildvorlagen können durch die Kombination verschiedener morphologischer Operatoren viele Bildverarbeitungsprobleme, wie z.B. eine Kantenextraktion, eine Skelettierung, eine Segmentierung oder das Zählen von Segmenten behandelt werden. In diesem Abschnitt werden die Grundlagen dazu zusammengefasst.

Sehr allgemein wird die mathematische Morphologie in [Serr82] und [Serr88] behandelt. Wichtige Mengenoperationen sind hier, neben den bekannten Operationen wie Durchschnitt, Vereinigung, Differenz und Komplement, die *Dilatation* und die *Erosion* von Mengen. Dilatation und Erosion werden in dieser mengentheoretischen Darstellung folgendermaßen definiert: Es seien X und K beliebige Mengen. Die Menge K wird als *strukturierendes Element* (manchmal auch *Kern*) bezeichnet. Die Menge K wird in alle möglichen Positionen y verschoben. Die so verschobene Menge wird mit K_y bezeichnet. Die Dilatation der Menge X durch das strukturierende Element K stellt die Menge aller Punkte y dar, bei denen der Durchschnitt von X und K_y nicht leer ist:

$$Y = X \oplus K = \{y \mid K_y \cap X \neq \emptyset\}. \tag{19.1}$$

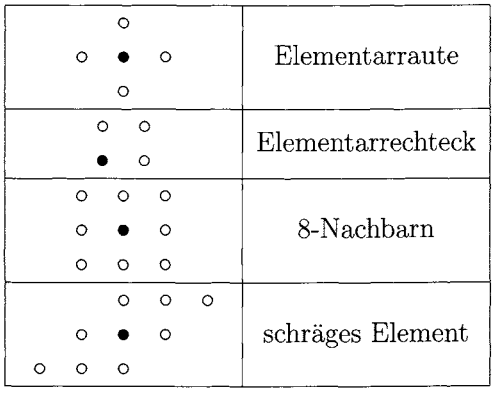


Bild 19.1: Beispiele für strukturierende Elemente. Mit • ist der Bezugspunkt und mit ○ sind die Nachbarn markiert.

Bei der Erosion wird gefordert, dass K_y eine Teilmenge von X ist:

$$Y = X \ominus K = \{y \mid K_y \subseteq X\} = \{y \mid K_y \cap X = K_y\}. \tag{19.2}$$

Aufbauend auf diesen grundlegenden Operationen werden in den oben zitierten Arbeiten weitere morphologische Operationen definiert und auf ihre algebraischen Eigenschaften untersucht. Auf diese zum Teil recht anspruchsvollen Ausführungen soll im Rahmen dieser Darstellung verzichtet werden. Vielmehr werden im folgenden die morphologischen Operationen praxisbezogen für Binär- und Grauwertbilder definiert. Eine ausführliche Zusammenstellung dazu ist in [Abma94] zu finden.

Es sei $\mathbf{S}_e = (s_e(x, y))$ ein Grauwertbild. Als strukturierendes Element wird eine Maske K (ein *Kern* K) verwendet, die festlegt, welche Bildpunkte zum Bildpunkt in der Position (x, y) als benachbart betrachtet werden. Der Bildpunkt des strukturierenden Elements, der auf die Position (x, y) gelegt wird, heißt *Bezugspunkt*. In Bild 19.1 sind einige Beispiele dazu gegeben.

Der Bezugspunkt des strukturierenden Elements wird bei der Verarbeitung auf die Bildpunkte in den Positionen (x, y), $x = 0, 1, ..., L-1$ und $y = 0, 1, ..., R-1$ gelegt. Probleme, die dabei mit den Bildrändern auftreten, sind in ähnlicher Weise wie in Abschnitt 18.2 beschrieben zu behandeln. Die Grauwerte der durch das strukturierende Element definierten Nachbarn werden der Größe nach geordnet und bilden eine *Rangfolge* f. Operatoren, die auf dieser Vorgehensweise aufbauen, werden *Rangordnungsoperatoren* genannt.

Dazu ein Beispiel: Gegeben sei der folgende Ausschnitt aus einem Grauwertbild:

	0	1	2	3	4	...	Spalte y
0	53	68	47	36	27	...	
1	57	67	52	41	31	...	
2	61	64	19	43	35	...	
3	63	38	58	47	41	...	
...	...	...	...	...	...	...	
Zeile x							

Als Bezugspunkt wird der Punkt $(x, y) = (2, 2)$ gewählt. Mit der Elementarraute als strukturierendes Element ergibt sich dann:

$$\begin{array}{ccc} & 52 & \\ 64 & 19 & 43 \\ & 58 & \end{array} \quad \begin{array}{c} \text{geordnete} \\ \rightarrow \\ \text{Rangfolge} \end{array} \quad f = \{19, 43, 52, 58, 64\}$$

Mit dem schrägen Element als strukturierendes Element ergibt dagegen:

$$\begin{array}{ccccc} & & 52 & 41 & 31 \\ & 64 & 19 & 43 & \\ 63 & 38 & 58 & & \end{array} \quad \begin{array}{c} \text{geordnete} \\ \rightarrow \\ \text{Rangfolge} \end{array} \quad f = \{19, 31, 38, 41, 43, 52, 58, 63, 64\}$$

Je nachdem, durch welches Element der geordneten Rangfolge f der Bildpunkt in der Position des Bezugspunktes ersetzt wird, ergeben sich unterschiedliche Operationen:

Eine *Dilatation* erhält man, wenn der Grauwert in der Position des Bezugspunkts durch den *maximalen* Wert der Rangfolge f ersetzt wird. Hier dehnen sich die helleren Bildbereiche auf Kosten der dunkleren Bildbereiche aus. Die Ausdehnungsrichtung kann dabei durch die Form des strukturierenden Elements beeinflusst werden.

Bei einer *Erosion* wird der Grauwert des Bezugspunktes durch den *minimalen* Wert der Rangfolge f ersetzt, was eine Ausdehnung der dunklen Bildbereiche auf Kosten der hellen bewirkt. Ein Beispiel zu Dilatationen und Erosionen wurde in Abschnitt 17.5 gegeben.

Eine weitere Operation ist das *Medianfilter*: Hier wird der Grauwert des Bezugspunkts durch den *mittleren* Wert der Rangfolge f ersetzt. Ein Medianfilter ist z.B. zur Verbesserung verrauschter Bilder geeignet. Eine Anwendung dazu ist in Abschnitt 19.3 enthalten.

Ist das Eingabebild $\mathbf{S}_e$ ein Binärbild, so können die Grauwerte der Bildpunkte auch als Werte 0 und 1 von Boole'schen Variablen aufgefasst werden. In diesem Fall können die morphologischen Operationen auch durch logische Operatoren realisiert werden, so die Dilatation etwa durch die ODER-Verknüpfung: Die 0/1-Werte der durch das strukturierende Element festgelegten Nachbarschaft werden mit ODER verknüpft und das Ergebnis wird dem Bildpunkt in der Position des Bezugspunktes zugewiesen. Die Erosion lässt sich sinngemäß durch die UND-Verknüpfung der benachbarten Bildpunkte realisieren.

Hier sei noch auf eine schnelle Implementierungsmöglichkeit der morphologischen Operationen hingewiesen ([Abma94]). Anstatt das strukturierende Element auf die Positionen (x, y) zu legen und die benachbarten Bildpunkte zu betrachten, kann das Eingabebild als Ganzes in die einzelnen Positionen des strukturierenden Elements verschoben und mit dem

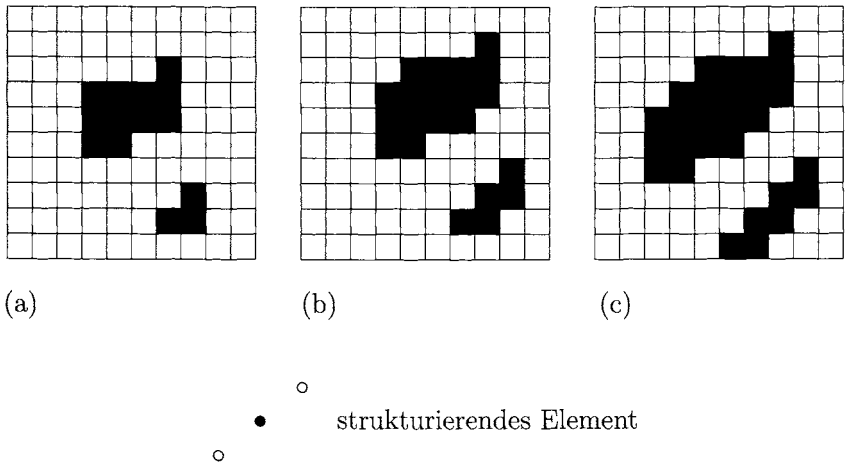


Bild 19.2: Beispiel für eine schnelle Implementierung von morphologischen Operationen durch SHIFT- und logische Operationen: Dilatation mit dem abgebildeten strukturierenden Element. (a) Originalbild (schwarze Bildpunkte: 1, weiße Bildpunkte: 0). (b) Verschiebung des Originals in die rechte obere Position des strukturierenden Elements und ODER-Verknüpfung mit dem Original. (c) Verschiebung des Originals in die linke untere Position des strukturierenden Elements und ODER-Verknüpfung mit dem Ergebnis von Teilbild (b).

Original entsprechend verknüpft werden (ODER bei der Dilatation oder UND bei der Erosion). Bild 19.2 zeigt ein Beispiel dazu. Voraussetzung für diese Verarbeitungstechnik sind schnelle Verschiebungsoperationen von Binärbildern und schnelle logische Verknüpfungen von zwei Binärbildern.

Auch bei Grauwertbildern kann die oben beschriebene Technik verwendet werden. An die Stelle des logischen ODER tritt dann die Maximumbildung und an die Stelle des logischen UND die Minimumbildung.

Bei Grauwertbildern werden manchmal die morphologischen Operationen etwas allgemeiner verwendet. Im Folgenden wird mit $\mathbf{K} = (k(i,j))$ das strukturierende Element bezeichnet. Dilatation und Erosion lassen sich dann wie folgt schreiben:

$$\mathrm{dil}(x,y) = \max_{i,j}\{s_e(x+i, y+j) + k(i,j)\}, \tag{19.3}$$

$$\mathrm{ero}(x,y) = \min_{i,j}\{s_e(x+i, y+j) - k(i,j)\}. \tag{19.4}$$

Dabei laufen die Indizes i und j über den Geltungsbereich des strukturierenden Elements. Stellt man sich ein Grauwertbild als „Grauwertgebirge" (Funktionswerte $s(x,y)$ an der Stelle (x,y)) vor, so ist das strukturierende Element $\mathbf{K}$ hier ein dreidimensionales Gebil-

de, das über das Grauwertgebirge gleitet. Bei einer Dilatation werden dann die Grauwerte entsprechend inkrementiert, bei der Erosion dekrementiert.

Nun noch eine Bemerkung, wie man beliebig geformte strukturierende Elemente verwenden kann, wenn man aus Programmierungsgründen **K** als rechteckigen Bereich vereinbart hat: Man kann z.B. festlegen, dass nur Werte mit $k(i,j) \geq 0$ als Nachbarn gewertet werden. Negative Werte markieren dann Positionen, die nicht in die Nachbarschaft einbezogen werden sollen.

Häufig werden auch mehrere Operationen miteinander kombiniert. Eine Folge von n Erosionen und anschließenden n Dilatationen wird *opening*-Operation genannt. n Dilatationen, gefolgt von n Erosionen heißen *closing*-Operation. Mit diesen Operatoren können gezielt dunkle Strukturen auf hellem Hintergrund bzw. helle Strukturen auf dunklem Hintergrund bearbeitet werden.

Eine weitere Verknüpfungsmöglichkeit ist die Verbindung einer *opening*- oder *closing*-Operation mit einer Differenzbildung mit dem Originalbild $\mathbf{S}_e$. Diese Operationen sind geeignet, wenn man gezielt bestimmte Strukturen aus einem Bild ausblenden will.

Werden bei einem Binärbild, das mit dem Grauwert 1 codierte Segmente auf einem mit 0 codierten Hintergrund enthält, abwechselnd Dilatationen mit einer anschließenden UND-Verknüpfung mit dem Original durchgeführt, so kann man gezielt Segmente ausblenden. Es ist damit möglich, alle Segmente zu extrahieren, die den Bildrand berühren oder die Löcher enthalten. Wird ein Segment „markiert", so kann durch diese Operation, die auch *grassfire*-Operation genannt wird, dieses Segment aus dem Bild ausgeblendet werden.

19.3 Median Filter

Wie im Grundlagenabschnitt 19.2 erläutert wird bei der Medianfilterung der Grauwert des Bezugspunktes durch den mittleren Wert der geordneten Folge f ersetzt. Im Beispiel von Abschnitt 19.2 mit der Elementarraute ergibt sich

$$\begin{array}{ccc} & 52 & \\ 64 & 19 & 43 \\ & 58 & \end{array} \quad \begin{array}{c} \text{Ersetzung} \\ \rightarrow \\ \text{med}(f) = 52 \end{array} \quad \begin{array}{ccc} & 52 & \\ 64 & 52 & 43 \\ & 58 & \end{array}$$

und mit dem schrägen Element ergibt sich

$$\begin{array}{cccc} & 52 & 41 & 31 \\ 64 & 19 & 43 & \\ 63 & 38 & 58 & \end{array} \quad \begin{array}{c} \text{Ersetzung} \\ \rightarrow \\ \text{med}(f)=43 \end{array} \quad \begin{array}{cccc} & 52 & 41 & 31 \\ 64 & 43 & 43 & \\ 63 & 38 & 58 & \end{array}$$

Für die Medianfunktion med$f$ gelten die folgenden Rechenregeln, die die Addition einer Konstanten zu den Werten der Folge und die Multiplikation der Folge mit einem Faktor beschreiben:

$$\text{med}\{c + f(k)\} = c + \text{med}\{f(k)\} \text{ und } \text{med}\{c \cdot f(k)\} = c \cdot \text{med}\{f(k)\}. \tag{19.5}$$

(a) (b)

Bild 19.3: (a) Original eines Fernsehbildes, das aufgrund eines schlecht eingestellten Senders stark verrauscht ist. (b) Mediangefiltertes Bild. Man sieht deutlich die verbesserte Bildqualität.

Es gilt jedoch nicht allgemein, dass der Medianwert der Summe von zwei Folgen gleich der Summe der Medianwerte ist.

Aus der Beschreibung eines Medianfilters ist zu ersehen, dass ein Medianfilter gut zur Elimination isolierter, fehlerhafter Bildpunkte geeignet ist. Auch Rauschen in einem Bild kann mit einem Medianfilter abgeschwächt werden. Gegenüber dem bewegten Mittelwert (Abschnitt 18.3) ergibt sich hier der Vorteil, dass die Grauwertübergänge (Kanten) besser erhalten bleiben. Allerdings ist im Vergleich mit dem bewegten Mittelwert ein erhöhter Rechenaufwand zu bewältigen. Mit schneller Bildverarbeitungshardware ist das aber kein Problem.

Die Bilder 19.3-a und 19.3-b zeigen ein Beispiel. Es wurde ein Fernseh-Videobild digitalisert, das aufgrund eines schlecht eingestellten Senders stark verrauscht ist. Die Anwendung eines Medianfilters hat hier eine wesentliche Verbesserung der Bildqualität gebracht.

Die Medianfilterung lässt sich auch mit anderen Vorverarbeitungsschritten kombinieren. In Abschnitt 17.8 wurde die Grauwertskalierung in Verbindung mit einer Hochpassfilterung erläutert. Bei manchen Anwendungen lässt sich die Bildqualität noch verbessern, wenn das skalierte und hochpassgefilterte Bild anschließend mit einem Medianfilter bearbeitet wird.

Neben der eben diskutierten Anwendung können mit einem Medianfilter auch einzelne, gestörte Bildpunkte eliminiert werden (Abschnitt 18.5).

19.4 Dilatation und Erosion im Binärbild

In diesem Abschnitt wird ein Binärbild $\mathbf{S}_e = (s_e(x,y))$ mit $G = \{0,1\}$ vorausgesetzt. Als strukturierendes Element wird das Element „8-Nachbarn" verwendet. Bei der Dilatation wird der binäre Grauwert des Bezugspunktes durch das Maximum der geordneten Folge f ersetzt. Wenn also in der durch das strukturierende Element definierten Nachbarschaft ein Bildpunkt den Grauwert 1 besitzt, so wird der Bezugspunkt durch 1 ersetzt. Dies lässt sich auch durch die logische Verknüpfung der benachbarten Bildpunkte darstellen:

$$\begin{aligned} &\mathbf{S}_e \rightarrow \mathbf{S}_a : \qquad (19.6)\\ &s_a(x,y) = s_e(x-1,y-1) \vee s_e(x-1,y) \vee s_e(x-1,y+1) \vee \\ &\qquad s_e(x,y-1) \vee s_e(x,y) \vee s_e(x,y+1) \vee \\ &\qquad s_e(x+1,y-1) \vee s_e(x+1,y) \vee s_e(x+1,y+1). \end{aligned}$$

Durch diese Verknüpfung behält ein Bildpunkt mit $s_e(x,y) = 1$ in $\mathbf{S}_a$ seinen Grauwert, während ein Bildpunkt $s_e(x,y) = 0$ in $\mathbf{S}_a$ den Grauwert 1 erhält, wenn in seiner 8-Nachbarschaft eine 1 auftritt. Dies bewirkt die Vergrößerung (Expansion) aller Flächen mit dem Grauwert 1 um einen Rand der Breite eines Bildpunktes. Außerdem werden kleine Flächen mit dem Grauwert 0 eliminiert. Bild 19.4-b und Bild 19.4-c zeigen die ein- bzw. zweimalige Expansion der weißen Flächen, ausgehend von Bild 19.4-a.

Die Erosion ist die entgegengesetzte Operation: Der binäre Grauwert des Bezugspunktes wird durch das Minimum der geordneten Folge f ersetzt. Das bewirkt mit obigem strukturierenden Element das Verkleinern (Kontraktion) aller Flächen mit dem Grauwert 1 um einen Rand der Breite eines Bildpunktes. Ein Bildpunkt mit $s_e(x,y) = 0$ behält in $\mathbf{S}_a$ seinen Grauwert, während $s_e(x,y) = 1$ in $\mathbf{S}_a$ nur dann den Grauwert 1 behält, wenn alle Nachbarn den Grauwert 1 besitzen. Dies kann durch die Boole'sche Und-Verknüpfung erreicht werden:

$$\begin{aligned} &\mathbf{S}_e \rightarrow \mathbf{S}_a : \qquad (19.7)\\ &s_a(x,y) = s_e(x-1,y-1) \wedge s_e(x-1,y) \wedge s_e(x-1,y+1) \wedge \\ &\qquad s_e(x,y-1) \wedge s_e(x,y) \wedge s_e(x,y+1) \wedge \\ &\qquad s_e(x+1,y-1) \wedge s_e(x+1,y) \wedge s_e(x+1,y+1). \end{aligned}$$

Da in der Boole'schen Algebra der Zusammenhang $\neg(a \wedge b) = \neg a \vee \neg b$ gilt, kann (19.7) so umgeformt werden, dass die Kontraktion des Grauwertes 1 durch eine Expansion des Grauwertes 0 erreicht werden kann. Diese Umformung ist auch anschaulich einleuchtend. Bild 19.4-d und Bild 19.4-e zeigen die ein- bzw. zweimalige Kontraktion des Grauwertes 1.

Es ist klar, dass durch die inverse Operation das Original nicht mehr reproduziert werden kann, da ja z.B. isolierte Bildpunkte mit $s_e(x,y) = 0$ bei der Dilatation eliminiert werden und bei der Erosion nicht mehr auftauchen. Diese aufeinander folgende Anwendung

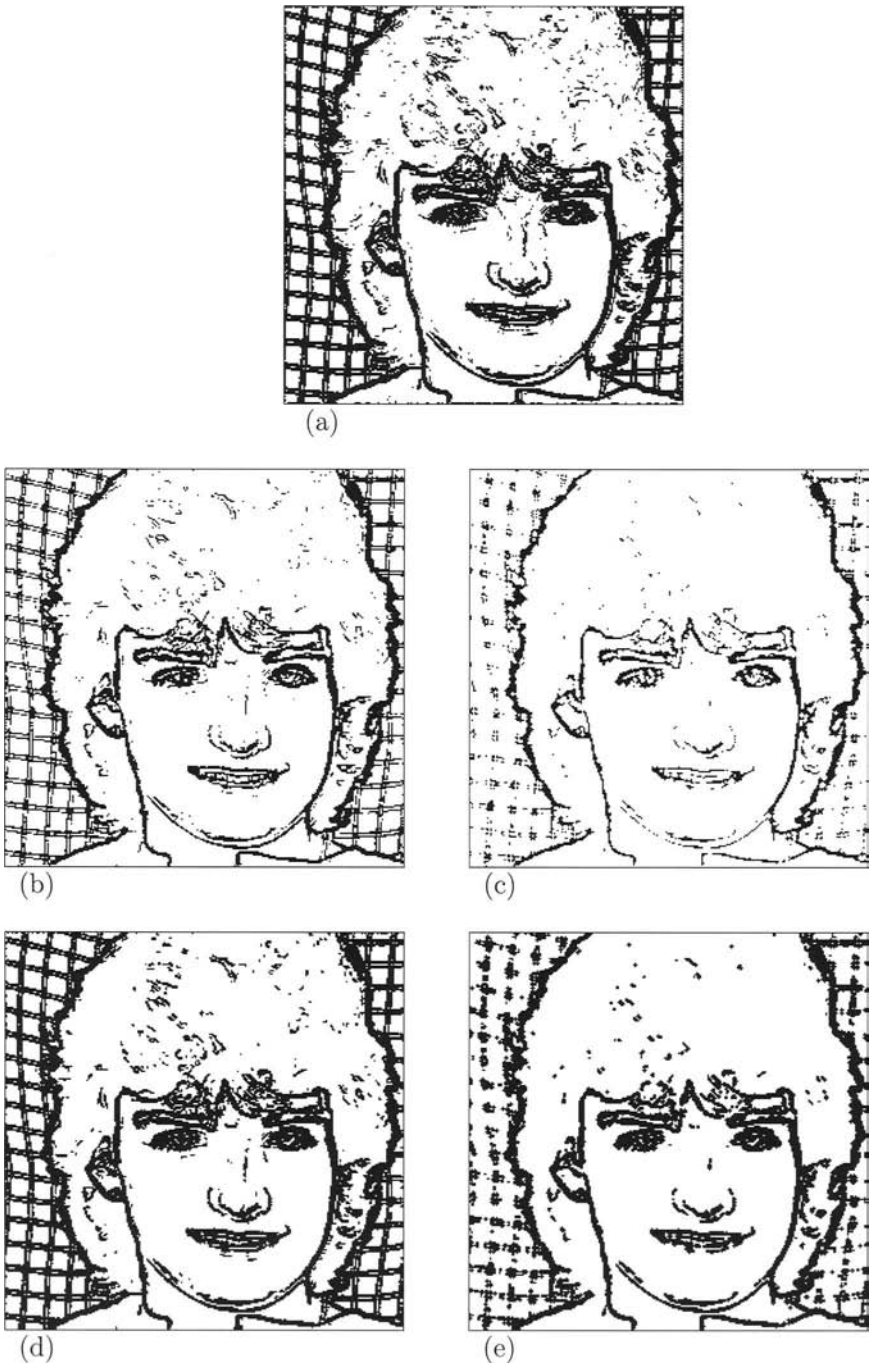
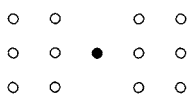


Bild 19.4: Expansions- und Kontraktionsoperatoren. (a) Originalbinärbild der Kantenextraktion. (b) und (c) Expansion (Dilatation) der Flächen mit dem Grauwert 1. (d) und (e) Kontraktion (Erosion) der Flächen mit dem Grauwert 1 durch Expansion der Flächen mit dem Grauwert 0.

von Dilatationen und Erosionen wird in der Praxis oft als *closing* (*Fermeture*) bezeichnet. Die Kombination von Erosionen mit anschließenden Dilatationen heißt dementsprechend *opening* (*Ouverture*).

Ein anderes einfaches Beispiel ist die Extraktion des Randes einer mit 1-en codierten Fläche in einem Binärbild. Dazu wird z.B. eine Dilatation durchgeführt und das Ergebnis mit dem Original mit der Boole'schen Operation XOR (eXclusives OR) verknüpft.

Die Richtung der Wirkungsweise einer Dilatation oder Erosion lässt sich durch die Gestalt des strukturierenden Elementes beeinflussen. In den obigen Beispielen zur Verarbeitung von Binärbildern wurde ein symmetrisches Element verwendet. Soll z.B. eine Dilatation verstärkt in Spaltenrichtung wirken, so könnte etwa das folgende strukturierende Element verwendet werden:



Durch entsprechendes apriori-Wissen kann somit durch geeignete Wahl des strukturierenden Elements die Wirkung dieser morphologischen Operationen auf den jeweiligen Anwendungsfall angepasst werden.

19.5 Morphologie im Grauwertbild

Die morphologischen Operationen lassen sich sinngemäß auch auf Grauwertbilder $\mathbf{S}_e = (s_e(x,y))$ mit $G = \{0,1,...,255\}$ anwenden. Es wird im Folgenden mit SE das strukturierende Element und mit $U(SE(x,y))$ die Menge der durch das strukturierende Element festgelegten Nachbarn eines Bildpunktes in der Position (x,y) bezeichnet. Die Dilatation und die Erosion können dann geschrieben werden:

$$\begin{aligned} \mathrm{dil}(x,y) &= \max\Big(U(SE(x,y))\Big) + c_g \\ \mathrm{ero}(x,y) &= \min\Big(U(SE(x,y))\Big) - c_g \end{aligned} \tag{19.8}$$

Dabei ist c_g ein Gewichtsfaktor, der abhängig von den Grauwerten von $g \in G$ als „Zuschlag" verwendet werden kann. Dann ist allerdings zu beachten, dass im Ergebnisbild z.B. nach einer Dilatation Grauwerte auftreten, die im Originalbild nicht vorhanden waren.

Anhand eines einfachen Beispiels werden im Folgenden die Verwendung von morphologischen Operationen im Grauwertbild erläutert: Die Problemstellung sei, die Messmarken in den vier Ecken des Bildes 19.5-a zu extrahieren. Durch eine Folge von Erosionen verschwinden die Messmarken allmählich (Bild 19.5-b). Wieviele Erosionen hintereinander ausgeführt werden, hängt von der Größe der Bildstrukturen ab. In diesem Beispiel wurden sechs Erosionen mit der Elementarraute durchgeführt. Besitzen die Bildstrukturen eine Vorzugsrichtung, so kann dies durch das strukturierende Element berücksichtigt werden. Im nächsten Schritt werden ebensoviele Dilatationen wie vorher Erosionen ausgeführt

(Bild 19.5-c). Die hellen Messmarken erscheinen dabei nicht mehr, aber größere helle Bildstrukturen erhalten wieder etwa ihre ursprüngliche Form. Das Differenzbild zwischen dem Original und dem erodierten/dilatierten Bild (*opening*-Operation) enthält nur diejenigen Bildinformationen, die durch die morphologischen Operationen verlorengegangen sind, hier im wesentlichen die Messmarken. Da die rechteckigen Objekte an den Ecken nicht exakt reproduziert wurden, treten in diesem Bereichen Störungen auf. Zu beachten ist auch, dass die Spiegelung auf der Kappe des Faserschreibers durch die *opening*-Operation eliminiert wurde. Eine Binarisierung von Bild 19.5-d liefert Bild 19.5-e. Die zusätzlichen Störungen sind deutlich zu erkennen. In einem weiteren Verarbeitungsschritt, in dem die Fläche und die Länglichkeit der schwarzen Bildstrukturen verwendet wurde, konnten die Messmarken extrahiert werden (Bild 19.5-f). Die weitere Verarbeitung besteht in diesem Beispiel aus der geometrischen Entzerrung des Originals, so dass die Messmarken mit ihren Schwerpunkten in den vier Bildecken liegen.

Bei segmentierten Bildern, bei denen die einzelnen Grauwerte der Bildpunkte Codes für die Zugehörigkeit zu einer Klasse sind (logische Bilder), können mit morphologischen Operationen punktweise Störungen eliminiert werden. Die beiden Bilder 19.6-a und 19.6-b zeigen ein Beispiel dazu. Hier wurden durch eine Reihe von Dilatationen und anschließenden Erosionen die punktweisen Störungen eliminiert. Allerdings gehen bei diesen Operationen feine Bildstrukturen verloren. Es kann auch passieren, dass der Zusammenhang der einzelnen Segmente unterbrochen wird. Das ist dann der Fall, wenn z.B. durch Dilatationen schmale Segmentstellen gelöscht und durch die anschließenden Erosionen nicht mehr rekonstruiert werden können (Bilder 19.6-c und 19.6-d).

Mit morphologischen Operationen wie Medianfilter, Dilatation und Erosion, können auch ganze Bildzeilen, die gestört sind, eliminiert werden. Wie bei der Korrektur einzelner Bildpunktstörungen ist hier auch der Nachteil in Kauf zu nehmen, dass die Grauwerte der Umgebungen ebenfalls verändert werden. Soll dies nicht der Fall sein, so können derartige Störungen durch die Berechnung der Korrelation aufeinander folgender Bildzeilen detektiert werden (Abschnitte 18.5 und 18.6).

Auch zur Verarbeitung von Kanten sind die morphologischen Operationen geeignet. Auf diese Thematik wird im Kapitel 20 näher eingegangen.

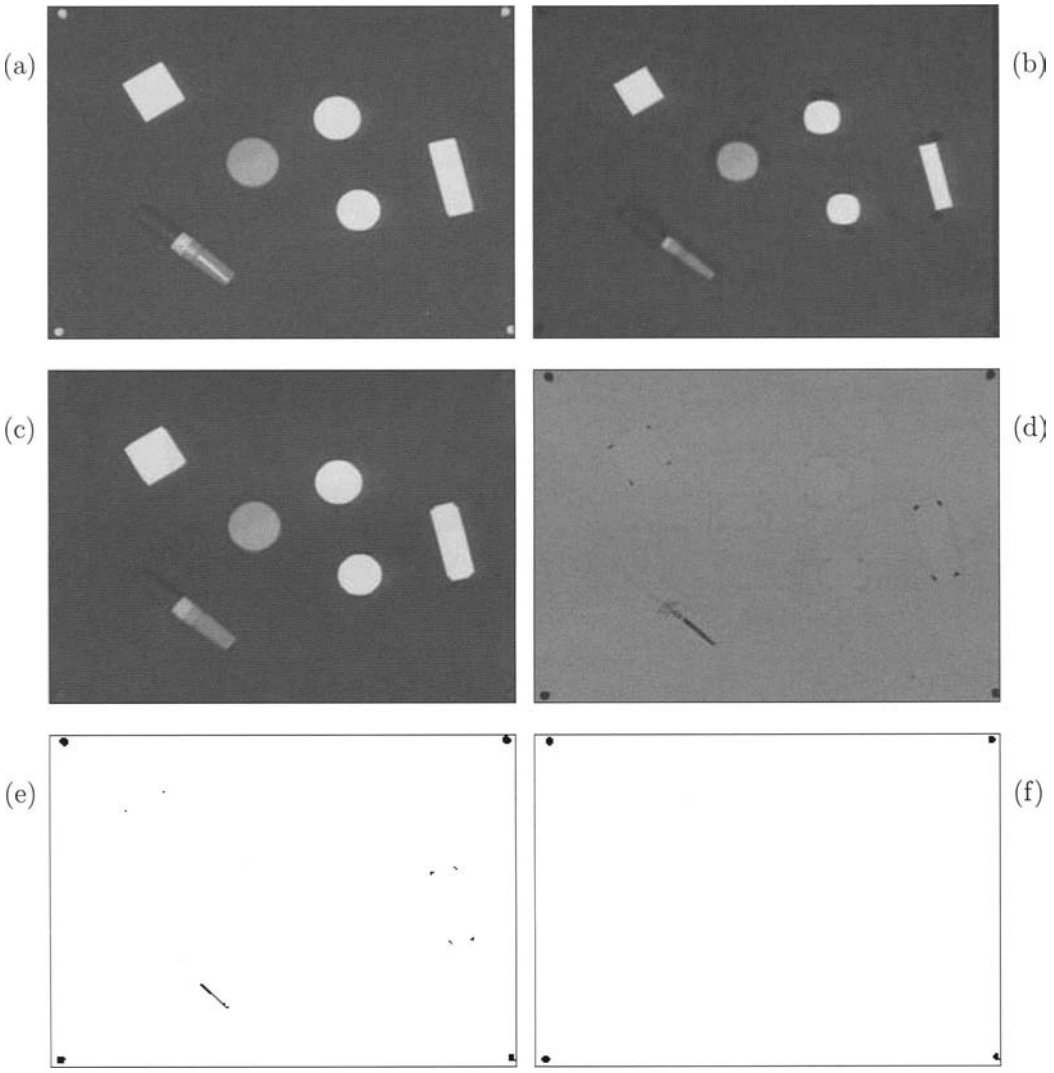
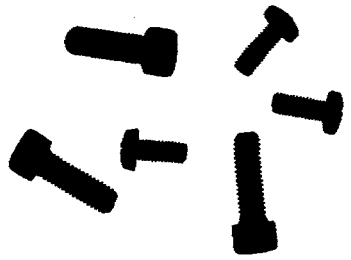
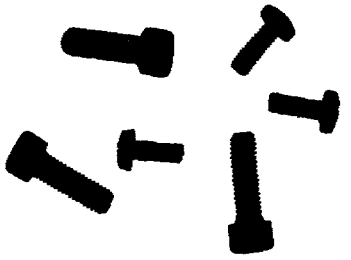


Bild 19.5: Beispiel zur Grauwertbildmorphologie. (a) Testbild mit mehreren Objekten und vier Messmarken in den Ecken. (b) Durch eine Folge von Erosionen verschwinden die hellen Messmarken. (c) Weitere Dilatationen reproduzieren etwa die Form größerer Bildstrukturen, die Messmarken erscheinen nicht mehr. (d) Das Differenzbild des Originals und des erodierten/dilatierten Bildes enthält die Bildinformationen, die verloren gingen. (e) Nach einer Binarisierung (Schwellwertbildung) sind deutlich die Messmarken und weitere Bildstörungen zu sehen. (f) Nach einer Formanalyse der schwarzen Bildteile (Flächeninhalt und Länglichkeit) konnten die Messmarken extrahiert werden. Die weitere Verarbeitung ist in diesem Beispiel eine geometrische Entzerrung des Originals, so dass die Messmarken in den Bildecken liegen.



(a)



(b)

(c)

(d)

Bild 19.6: (a) Binarisiertes Testbild. (b) Durch Dilatationen und anschließende Erosionen wurden die punktweisen Störungen eliminiert. Bei diesen Operationen gehen allerdings feine Bildstrukturen verloren. (c) Testbild mit Bereichen, bei denen die Segmente sehr schmal werden. (d) Die Dilatationen unterbrechen hier den Zusammenhang der Segmente.

Kapitel 20

Kanten und Linien

20.1 Anwendungen

In den folgenden Abschnitten werden Verfahren beschrieben, bei denen das Herausarbeiten von Grauwertübergängen wichtig ist. Dazu gehören einfache Verfahren der Kantenextraktion, aber auch komplizierte Algorithmen, die eine Kante über den Gradient beschreiben. Anschließend wird besprochen, wie sich der Rand eines Segments mit den Methoden der mathematischen Morphologie ermitteln lässt. Weiter folgen Beispiele zur Verarbeitung von Linien. Hier wird unter anderem auch die Skelettierung dargestellt.

20.2 Grundlegendes über Kanten und Linien

Physiologische Untersuchungen haben gezeigt, dass das menschliche Auge besser auf Diskontinuität als auf Kontinuität geeicht ist. Das ist auch plausibel, da diejenigen Bereiche, die Helligkeitsübergänge aufweisen, uns bei der Betrachtung eines Bildes in der Regel mehr Information geben, als weitgehend homogen graue oder einfarbige Bereiche. Für die Verarbeitung digitalisierter Bilder heißt das, dass der Versuch, die Verarbeitung, z.B. eine Segmentierung, kanten- und linienorientiert anzugehen, durchaus erfolgversprechend sein kann.

Dieser Abschnitt beschäftigt sich vorwiegend mit der Extraktion und der Weiterverarbeitung von Kanten und Linien. Das Ausgangsbildmaterial, auf das die Verfahren zur Kantenextraktion angewendet werden, kann dabei auf unterschiedliche Weise zustande kommen.

Ein Grauwertbild, das mit einem Videosensor oder einem Scanner aufgezeichnet wurde, ist ein mögliches Ausgangsbild für die Hervorhebung von Kanten. Zunächst muss untersucht werden, was unter einer Kante oder, präziser ausgedrückt, unter einer *Grauwertkante* zu verstehen ist. Dazu wird ein Grauwertbild $\mathbf{S}_e = (s_e(x, y))$ betrachtet, das einen hellen und einen dunklen Bildbereich enthält (Bild 20.1, oberes Teilbild)

Der Funktionsverlauf ist in der grafischen Darstellung der Grauwerte (*Grauwertprofil*) entlang der Strecke $\overline{AB}$ durch eine große Steigung gekennzeichnet. Stellt man sich diesen

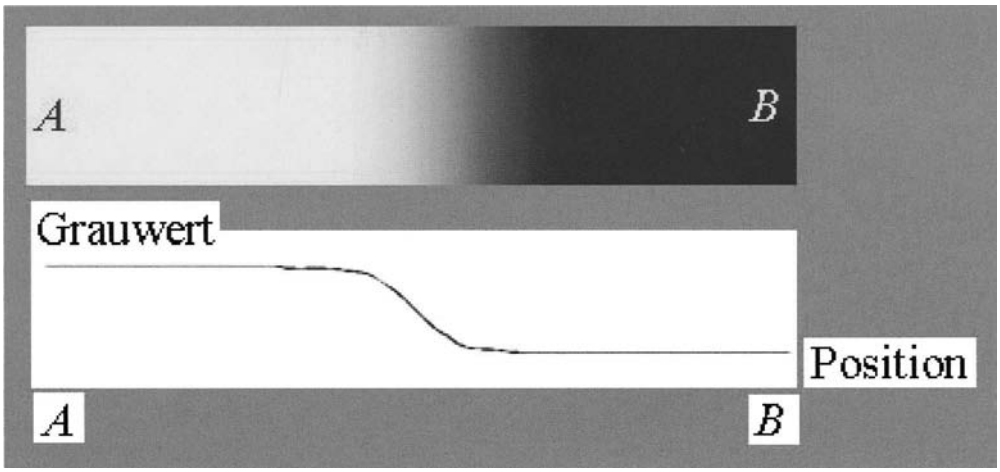


Bild 20.1: Oberes Teilbild: Bildausschnitt mit einer vergrößerten Grauwertkante. Unteres Teilbild: Grafische Darstellung des Grauwertverlaufs entlang der Strecke $\overline{AB}$ (Grauwertprofil).

Sachverhalt zweidimensional als „Grauwertgebirge" vor, so wird sich im Übergangsbereich eine „Rampe" ausprägen.

Eine Grauwertkante wird mit Hilfe des Gradienten (Abschnitt 18.4) charakterisiert: In homogenen oder nahezu homogenen Bildbereichen ist der Betrag des Gradienten gleich null oder sehr klein, während er im Bereich der Kante einen großen Betrag aufweist.

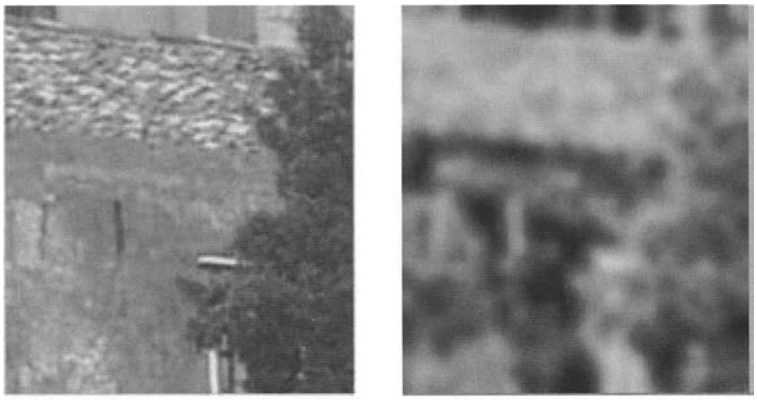
Die Charakterisierung einer Kante allein mit Hilfe des Gradientenbetrags ist noch nicht ausreichend, da ja z.B. ein Bildbereich mit starker Oberflächenstruktur auch viele Punkte mit großem Betrag des Gradienten aufweist. Allerdings werden hier die Richtungen mehr oder weniger zufällig verteilt sein. So wird man von einer Kante zusätzlich verlangen, dass sie sich in einem bestimmten Bildausschnitt durch eine Vorzugsrichtung des Gradienten auszeichnet, also kontinuierlich verläuft. Ab welcher Ausdehnung von einer Kante gesprochen wird, kann nicht pauschal gesagt werden, dies muss vielmehr im jeweiligen Anwendungsfall festgelegt werden.

In obiger Beschreibung wurde der Begriff der Kante als Grauwertkante erläutert. Kanten können aber auch in anderen Merkmalen auftreten, so z.B. eine Farbkante . Eine *Farb-* oder *Signaturkante* liegt vor, wenn in mindestens einem Farb- (Spektral-) Kanal eine Grauwertkante auftritt.

Werden durch ein Merkmal, das Bewegungen in einer Bildfolge erfasst (Kapitel 26), bewegte Bildbereiche von unbewegten Bildbereichen abgegrenzt, so könnte man von einer *Bewegungskante* sprechen.

Eine *Texturkante* entsteht im Übergangsbereich von unterschiedlichen Oberflächenstrukturen (Bild 20.2).

Um den Übergangsbereich der beiden Texturen zu finden, werden hier texturbeschrei-



(a) (b)

Bild 20.2: Beispiel zu unterschiedlichen Oberflächenstrukturen (Texturen). (a) Bildausschnitt mit verschiedenen Texturen. (b)Durch die Anwendung eines Texturmaßes werden Texturkanten berechnet.

bende Merkmale verwendet (Kapitel 27). Einige Beispiele dazu sind:

- Die Berechnung der Streuung für Umgebungen.
- Die Berechnung des Betrags und/oder der Richtung des Gradienten.
- Texturmaße, die aus der *co-occurrence*-Matrix abgeleitet werden (Abschnitt 16.11).
- Texturbeschreibende Parameter aus der Autokorrelation von Bildausschnitten, der Verarbeitung von Fourier-Koeffizienten oder der Verwendung anderer Transformationen.

Eine ausführliche Darstellung der Texturmerkmalsberechnung, auch in Verbindung mit der fraktalen Geometrie, wird in den Kapiteln 27, 28 und 29 gegeben.

Wenn die Texturmaßzahlen in die Grauwertmenge $G = \{0, ..., 255\}$ abgebildet werden, so ergibt sich ein Bild, dessen Grauwerte Maßzahlen für Texturen sind. Ein derartiges Bild kann natürlich als Eingabebild für eine Kantenextraktion dienen. Allerdings wird der berechnete Kantenverlauf, je nach der Größe der jeweiligen Texturen, den tatsächlichen Verlauf nur angenähert wiedergeben.

Nun zum Begriff der Linie. Eine Linie ist ein Gebilde, das beidseitig durch Kanten begrenzt ist, die einerseits nicht zu weit voneinander entfernt sind und zum anderen weitgehend parallel verlaufen (Bild 20.3). Bei welchem Abstand und welcher Ausdehnung der begrenzenden Kanten ein Gebilde als Linie angesehen wird oder nicht, ist wieder abhängig vom jeweiligen Problemkreis.

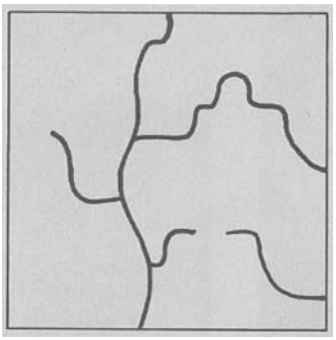


Bild 20.3: Verlauf einer Linie. Verzweigungen und Unterbrechungen können den Linienverlauf bestimmen.

Bei einer Linie in einem Luftbild kann es sich z.B. um eine Straße, eine Eisenbahnlinie oder um einen Flusslauf handeln. In einer Strichzeichnung sind die dunklen (oder hellen) Linien auf dem hellen (oder dunklen) Hintergrund die eigentlichen Informationsträger. Bei einem Fingerabdruck wiederum prägen sich die Papillaren als helle oder dunkle Linien aus.

Wird auf eine Grauwertkante ein lokaler Differenzenoperator (Abschnitt 18.4) angewendet, so wird im gefilterten Bild der Verlauf der Grauwertkante durch eine Linie wiedergegeben. Aus diesem Grund ist der Übergang zwischen kanten- und linienorientierter Bildsegmentierung fließend.

Eine kantenorientierte Bildsegmentierung läuft im wesentlichen so ab (Bild 20.4):

- Aus dem Originalbild (Bild 20.4-a) wird im ersten Schritt die Grauwertkante zwischen Objekt und Hintergrund extrahiert. Sie ist in Bild 20.4-b als Linie dargestellt.
- Im nächsten Schritt wird das kantenextrahierte Bild durch ein Schwellwertverfahren binarisiert. Dadurch wird logisch zwischen Kantenbereichen und keinen Kantenbereichen unterschieden.
- Mit Hilfe der zusätzlichen Information, ob das zu segmentierende Objekt links oder rechts der Kante liegt, kann jetzt eine Maske angefertigt werden, die logisch zwischen Hintergrund und dem Bereich des Objektes unterscheidet (Bild 20.4-c).
- Schließlich kann, falls benötigt, der Bereich des Objektes mit Hilfe der Maske aus dem Original ausgeblendet werden (Bild 20.4-d).

Im Folgenden ist zu untersuchen, wie die Bildpunkte auf der Grauwertkante ermittelt werden können. Dabei unterscheidet man zwischen zwei Vorgehensweisen. Bei der *parallelen Methode* werden alle Bildpunkte des zu verarbeitenden Bildes einem Extraktionsprozess unterzogen, während bei der *sequentiellen Methode*, ausgehend von einem bestimmten

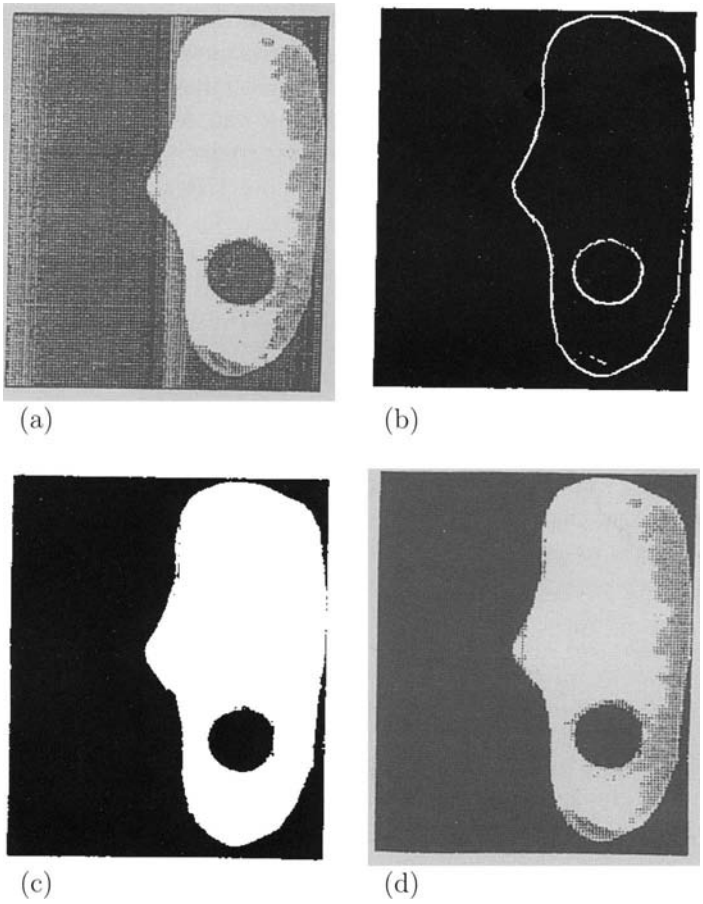


Bild 20.4: Kantenorientierte Bildsegmentierung. (a) Originalbild. (b) Extraktion der Kante zwischen Objekt und Hintergrund. (c) Maske: Hintergrund/Objekt. (d) Original mit ausgeblendetem Hintergrund.

Anfangspunkt, Nachfolgepunkte gesucht werden. Die sequentielle Methode wird auch als *Linienverfolgung* bezeichnet.

20.3 Einfache Verfahren zur Kantenextraktion

Mit Differenzenoperatoren (Abschnitt 18.4) werden Grauwertübergänge deutlicher herausgearbeitet. Einige Anwendungsbeispiele wurden bereits in vorhergehenden Kapiteln besprochen: Z.B. wurde in Abschnitt 17.8 dargestellt, dass eine Kontrastanreicherung durch die Veränderung der Grauskala und die Überlagerung mit dem Laplace-gefilterten Originalbild die Bildqualität für die visuelle Betrachtung verbessern kann.

Dieser Abschnitt beschäftigt sich vorwiegend mit der Extraktion und der Weiterverarbeitung von Kanten. Das Ausgangsbildmaterial, auf das die Verfahren zur Kantenextraktion angewendet werden, kann dabei, wie in Abschnitt 20.2 erläutert, auf unterschiedliche Weise zustande kommen.

Die Anwendung eines Differenzenoperators auf das (eventuell vorverarbeitete) Eingabebild ist der erste Schritt in Richtung Kantenextraktion. Geeignet sind dazu z.B. die Operatoren, die im Grundlagenabschnitt 18.4 aufgeführt sind.

Die Filterkerne (18.20) sind in der Praxis zu anfällig für Störungen. Häufig wird der Sobeloperator (18.22) verwendet. Er bringt bei den meisten Anwendungen gute Ergebnisse, auch wenn er dazu neigt, die Grauwertkanten verhältnismäßig dick darzustellen. Dadurch können feine Strukturen verloren gehen.

Bei Verwendung von (18.20) oder (18.22) ist zu beachten, dass mit diesen Filterkernen eine partielle Ableitung in Zeilen- und in Spaltenrichtung nachvollzogen wird. Um die Kanten herauszuarbeiten, muss anschließend noch der Betrag des Gradienten mit (18.25) berechnet werden.

Der Laplace-Operator (18.29) liefert direkt ein kantenextrahiertes Bild. Welche der vorgeschlagenen Varianten am besten geeignet ist, muss in jedem Anwendungsfall anhand des aktuellen Bildmaterials entschieden werden.

Für alle Filterkerne gilt die Bemerkung, dass die vorgeschlagene Größe nicht auf $m = 3$ beschränkt werden muss. Vielmehr können sie sinngemäß auch auf größere Umgebungen ausgedehnt werden. Allerdings ist bei größeren Filterkernen zu bedenken, dass sich dadurch höhere Rechenzeiten ergeben.

Nachdem die Kanten mit einem der vorgeschlagenen Operatoren vorbereitet wurden, folgt als nächster Schritt eine Segmentierung. Die Eigenschaft „ein Bildpunkt liegt auf einer Kante“ drückt sich durch einen hohen Betrag des Gradienten aus. Als Segmentierung kann daher eine einfache Schwellwertbildung (Binarisierung, Abschnitt 17.5) verwendet werden, bei der Bildpunkte mit einem hohen Gradientenbetrag als Kantenpunkte gewertet werden.

A20.1: Extraktion von Kanten.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild mit $G = \{0,1,...,255\}$ als Grauwertmenge (Eingabebild). Dieses Bild kann auch das Ergebnis einer längeren Vorverarbeitung (z.B. einer Berechnung von Texturparametern) sein.

◇ $\mathbf{S}_a = (s_a(x,y))$ ein einkanaliges Ausgabebild. Der Grauwert 0 bedeutet „keine Kante", der Grauwert 1 oder 255 bedeutet „Kante".

Algorithmus:

(a) Anwendung eines Differenzenoperators (z.B. Sobeloperator) auf das Eingabebild $\mathbf{S}_e$.

(b) Berechnung des Betrags des Gradienten aus den beiden Richtungskomponenten $s_x(x,y)$ und $s_y(x,y)$ gemäß:

$$g_b(x,y) = \sqrt{s_x(x,y)^2 + s_y(x,y)^2}.$$

(c) Binarisierung der Gradientenbeträge mit einem geeigneten Schwellwert c:

$$s_a(x,y) = \begin{cases} 0, & \text{falls } g_b(x,y) \leq c, \\ 1, & \text{sonst.} \end{cases}$$

Ende des Algorithmus

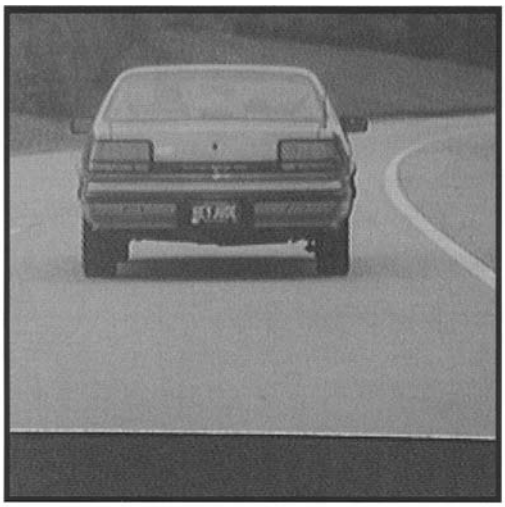
Die Bilder 20.5-a, 20.5-b und 20.5-c zeigen ein Beispiel eines Originals und des daraus berechneten und binarisierten Kantenbildes.

Bei der hier geschilderten, einfachen Methode zur Kantenextraktion tritt das Problem auf, dass Kanten, die im Eingabebild zwar vorhanden sind, die sich aber nicht so deutlich abheben wie andere Kanten, eventuell nicht erkannt werden. Das liegt z.B. daran, dass sich für derartige Kanten ein niederer Wert für den Gradientenbetrag ergibt, der bei der Binarisierung dann unter dem Schwellwert liegt. In der Bildfolge 20.5-a bis 20.5-c ist dieser Effekt deutlich zu sehen.

Wenn das Kantengebilde vollständig extrahiert werden soll, so müssen aufwändigere Methoden angewendet werden. Bevor auf die weitere Verarbeitung des binarisierten Kantenbildes, wie z.B. Skelettierung oder das Schließen unterbrochener Linien, eingegangen wird, werden in den folgenden Abschnitten Anregungen zu einer gezielteren Kantendetektion gegeben.

20.4 Parallele Kantenextraktion

Bei der parallelen Kantenextraktion wird für jeden Bildpunkt ein Maß berechnet, das als Wahrscheinlichkeit interpretiert werden kann, dass er zu einer Kante gehört. Die wesentlichen Schritte dabei sind im Folgenden wiederholt:



(a)

(b)

(c)

Bild 20.5: (a) Originalbild. (b) Kantenbild nach einem Sobeloperator und einer Berechnung der Gradientenbeträge. (c) Binarisiertes Bild. Hier werden z.B. im oberen Bildhintergrund Bildpunkte als Kanten ausgewiesen, wo im Original keine Kanten sind. Kanten, deren Gradientenbeträge unter dem Schwellwert liegen, werden nicht segmentiert.

Zuerst wird ein lokaler Differenzenoperator (z.B. Laplace Operator oder Sobel Operator) oder ein Hochpassfilter im Frequenzbereich angewendet (Abschnitt 18.4 und Kapitel 21). Jedem Bildpunkt wird dadurch eine Maßzahl zugewiesen, die sich entweder nur aus dem Betrag des Gradienten oder aus Betrag und Richtung zusammensetzt. Im nächsten Schritt werden die Gradientenbeträge einer Schwellwertbildung unterzogen. Ein so erzeugtes Binärbild enthält dann z.B. die Kanten des Originalbildes als Linien mit dem Grauwert 1. Es kann jetzt noch sinnvoll sein, das Binärbild von isolierten Einsen zu säubern, die Linien zu verdünnen und eventuelle Unterbrechungen zu überbrücken (z.B. mit morphologischen Operationen, Kapitel 19). Nachdem das Binärbild „gesäubert" wurde, kann dann die Maskenbildung, wie in Abschnitt 20.2 erläutert, durchgeführt werden.

Soll mit dieser parallelen Methode eine Linie extrahiert werden, so ist zu beachten, dass eine Linie nach der Filterung als Doppellinie erscheint. Dieser Sachverhalt kann u.U. als zusätzliche Information bei der weiteren Verarbeitung verwendet werden.

Bei der geschilderten Vorgehensweise wurde die Entscheidung, ob ein Bildpunkt auf einer Kante liegt oder nicht, nur vom Betrag des Gradienten und einer anschließenden Schwellwertbildung abhängig gemacht. Um hier mehr Sicherheit zu gewinnen, kann anstatt der Schwellwertbildung auch ein Klassifikator (Kapitel 31) verwendet werden. Dazu ein Beispiel (Bild 20.6-a): In einem Bild sei ein dunkles Objekt auf einem hellen Hintergrund abgebildet. Wird in einem zweidimensionalen Merkmalsraum auf der Abszisse der Grauwert eines Bildpunktes im Original aufgetragen und als Ordinate der Betrag des Gradienten, so werden sich die Verhältnisse ergeben, wie sie Bild 20.6-b zeigt. Der Klasse „Kante" ist in diesem Beispiel also der rechte obere Bereich des Merkmalsraumes zugeordnet. Dieser Sachverhalt kann z.B. mit einem festdimensionierten Klassifikator, der anhand einer Stichprobe trainiert wird, erfasst werden (Kapitel 31).

Da eine Grauwertkante meistens breiter als ein Bildpunkt ist, kann die Entscheidung, ob ein bestimmter Bereich auf der Kante liegt oder nicht, auch anhand von benachbarten Bereichen getroffen werden (Bild 20.7). Man könnte etwa immer $3 \cdot 3$ Bildpunkte große Bildausschnitte auf einmal zuordnen. Dazu berechnet man den Mittelwert des Gradientenbetrags dieses Ausschnittes und vergleicht ihn mit den entsprechenden Mittelwerten der 4-benachbarten oder 8-benachbarten $3 \cdot 3$-Umgebungen. Wenn der Mittelwert des Zentrums über einer Schwelle liegt, werden die $3 \cdot 3$ Bildpunkte der Klasse „Kante" zugeordnet.

Eine andere Vorgehensweise bietet sich an, wenn man eine Vermutung über den Verlauf einer Kante oder Linie in einem Bildausschnitt hat. In diesem Fall wird ein mathematisches Modell des Kanten- oder Linienverlaufs aufgestellt und mit dem tatsächlichen Verlauf im Bildausschnitt verglichen. Auch dazu ein Beispiel: Im Rahmen einer Kantenextraktion ergebe sich die Vermutung, dass in einem Bildausschnitt $s_e(x+k, y+k), k = 0, 1, ..., m-1$ eines Bildes $\mathbf{S}_e = (s_e(x, y))$ ungefähr diagonal von links oben nach rechts unten eine Grauwertkante verläuft. Diese Vermutung kann durch eine funktionale Approximation des Verlaufs der Kante modelliert werden, z.B. durch die Aufstellung einer Geradengleichung oder durch die Berechnung einer „Schablone" (*Template*) $\mathbf{H} = (h(u, v))$ der Größe $m \cdot m$,

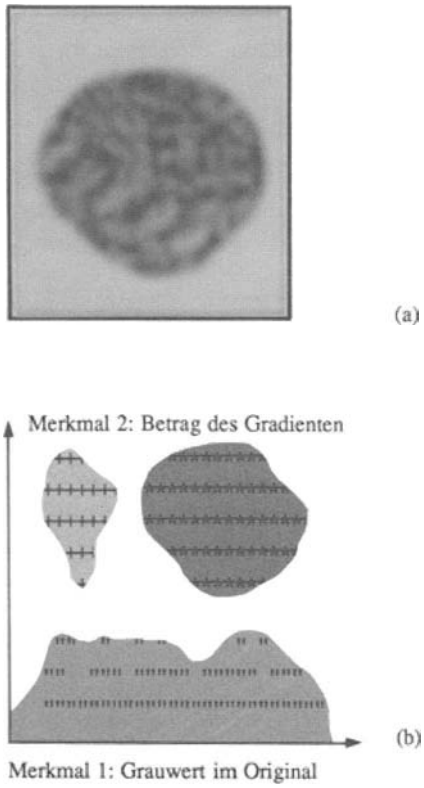


Bild 20.6: Klassifizierung der Bildpunkte im Merkmalsraum Grauwert/Betrag des Gradienten. (a) Originalbild: heller Hintergrund, dunkles Objekt. (b) Merkmalsraum: Grauwert/Betrag des Gradienten. + Bildpunkte mit kleinem Grauwert und großem Gradientenbetrag; * Bildpunkte mit großem Grauwert und großem Gradientenbetrag; " Bildpunkte mit kleinem Gradientenbetrag.

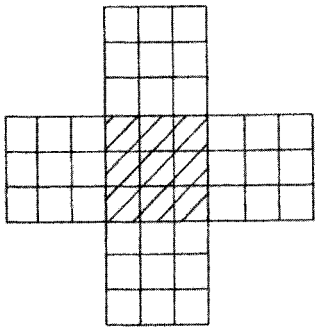


Bild 20.7: Zuordnung von $3 \cdot 3$-Ausschnitten anhand des mittleren Gradientenbetrags der Umgebung.

in der die Bildpunkte folgende Grauwerte besitzen:

$$h(u,v) = \begin{cases} 0, & u > v, \\ 1, & \text{sonst.} \end{cases} \tag{20.1}$$

Der zu überprüfende Bildausschnitt wird nun binarisiert und führt zur binären Bildmatrix $s_{bin}(x+k, y+k)$. Jetzt wird ein Maß d der Abweichung des Ausschnittes vom Modell berechnet:

$$d = \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} \Big(s_{bin}(x+u, y+v) - h(u,v) \Big)^2 = \tag{20.2}$$

$$= \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} s_{bin}(x+u, y+v)^2 - 2 \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} s_{bin}(x+u, y+v) h(u,v) + \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} h(u,v)^2.$$

Der erste Term der ausquadrierten binomischen Formel (20.2) ist für den betrachteten Bildausschnitt konstant, der letzte Term für den Modellausschnitt. Zur Berechnung des Maßes für die Abweichung ist somit der mittlere Term ausreichend. Die modellierte Lage der Kante im Bildausschnitt wird dann akzeptiert, wenn die Größe

$$d' = \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} s_{bin}(x+u, y+v) h(u,v) > c \tag{20.3}$$

ist, mit einer geeigneten Schwelle c. Die Berechnung der Übereinstimmung zwischen Bildausschnitt und Modell gemäß (20.3) läuft somit auf die Berechnung der Korrelation hinaus.

20.5 Kantenextraktion mit Betrag und Richtung des Gradienten

Bei dem hier geschilderten *Kantendetektor* werden neben Gradientenbeträgen auch die Gradientenrichtungen ausgewertet. Der erste Schritt ist wie oben die Anwendung eines Differenzenoperators, der die Richtungskomponenten $s_x(x,y)$ und $s_y(x,y)$ des Gradienten liefert. Die Gradientenbeträge $g_b(x,y)$ werden mit (18.22) berechnet. Um zu einer betragsunabhängigen Kantenbewertung zu kommen, werden die Gradientenbeträge einer *Rangordnungsoperation* unterworfen. Die Gradientenbeträge in den $3 \cdot 3$–Umgebungen zweier Bildpunkte könnten etwa folgende Werte haben:

$$\begin{array}{ccc} 18 & 31 & 26 \\ 17 & 44 & 16 \\ 19 & 54 & 20 \end{array} \quad \text{und} \quad \begin{array}{ccc} 78 & 234 & 112 \\ 81 & 227 & 104 \\ 65 & 103 & 230 \end{array}. \tag{20.4}$$

Mit einem fixen Schwellwert ist es unmöglich, aus beiden Umgebungen die richtigen Kantenstücke zu erhalten.

Die geordneten Folgen (die *Rangordnungen*) dieser Umgebungen lauten:

$$16 \quad 17 \quad 18 \quad 19 \quad 20 \quad 26 \quad 31 \quad 44 \quad 54 \tag{20.5}$$

und

$$65 \quad 78 \quad 81 \quad 103 \quad 104 \quad 112 \quad 227 \quad 230 \quad 234. \tag{20.6}$$

Nun wird der Wert jedes Bildpunktes durch den Wert seiner Position in dieser Rangordnung ersetzt (*Positionsrang*). Man erhält dann:

$$\begin{array}{ccc} 3 & 7 & 6 \\ 2 & 8 & 1 \\ 4 & 9 & 5 \end{array} \quad \text{und} \quad \begin{array}{ccc} 2 & 9 & 6 \\ 3 & 7 & 5 \\ 1 & 4 & 8 \end{array}. \tag{20.7}$$

Es ist deutlich zu sehen, dass die beiden Kantenbereiche, ausgezeichnet durch hohe Gradientenwerte, im Positionsrang ebenfalls die hohen Werte erhalten. Außerdem werden beide Kantenausschnitte gleich bewertet, obwohl die Gradientenwerte im linken Ausschnitt wesentlich kleiner sind. Legt man als Schwelle für die Klasse „Kante“ den Schwellwert $c \geq 7$ fest, so erhält man folgende binarisierte Ausschnitte, die den Kantenverlauf richtig wiedergeben:

$$\begin{array}{ccc} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{array} \quad \text{und} \quad \begin{array}{ccc} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}. \tag{20.8}$$

Nun einige Bemerkungen zu Problemen, die sich bei der Berechnung und Auswertung der Positionsrangbilder ergeben. Zunächst können in einer sortierten Folge gleiche Grauwerte auftreten. Es stellt sich die Frage, welchen Positionsrang man ihnen zuordnet. Der

höchste Rang schließt sich aus, da sonst in homogenen Gebieten alle Punkte diesen Rang erhalten würden und somit als Kante markiert würden. Der niederste Rang ist ebenfalls problematisch, da dadurch in Linienstücken häufige Unterbrechungen auftreten. Am besten ist wohl ein mittlerer Rang geeignet, der sich durch eine Mittelung und Rundung über die in Frage kommenden Rangpositionen berechnet.

Ein anderer Parameter, der das Ergebnis beeinflusst, ist die Umgebungsgröße, die zur Positionsrangberechnung verwendet wird. Hier ist, wie häufig in der Bildverarbeitung und Mustererkennung, zwischen repräsentativen Ergebnissen und benötigter Rechenzeit ein Kompromiss zu schließen. Um einigermaßen zusammenhängende Kantenstücke zu erhalten, sollte man schon eine $5 \cdot 5$– oder $7 \cdot 7$–Umgebung verwenden. Bei größeren Umgebungen wird der Rechenaufwand zu groß.

Schließlich beeinflusst die Schwelle, die zwischen „Kante“ und „nicht Kante“ unterscheidet, das Ergebnis wesentlich. Sie ist abhängig von der jeweiligen Umgebung und muss anhand des aktuellen Bildmaterials durch einige Versuche ermittelt werden. Bei $5 \cdot 5$– bzw. $7 \cdot 7$–Umgebungen kann das Probieren etwa mit einer Schwelle von 14 bzw. 25 beginnen.

Das binarisierte Positionsrangbild (im folgenden als *Rangbild* bezeichnet) wird auch in relativ homogenen Bildbereichen Kanten ausweisen. Diese können aber in Verbindung mit einer Richtungssystematik eliminiert werden.

Für die *Richtungssystematik* wird aus den Richtungskomponenten $s_x(x, y)$ und $s_y(x, y)$ ein Maß für die Gradientenrichtung an dieser Stelle ermittelt. Die Richtungswinkel φ könnten dazu gemäß

$$\varphi = \arctan \frac{s_x(x, y)}{s_y(x, y)} \tag{20.9}$$

berechnet werden. Aus Laufzeitgründen und aufgrund der Vorgehensweise, wie die Richtungsystematik ermittelt wird, ist es zu empfehlen, nicht die arctan–Berechnung von (20.9) zu verwenden, sondern die Werte $s_x(x, y)$ und $s_y(x, y)$ zu skalieren und auf ein umgebendes Quadrat mit einer Richtungseinteilung abzubilden. Die skalierten Werte zeigen dann direkt auf ein Richtungsfeld. Bild 20.8 zeigt ein Beispiel eines Richtungsquadrats der Größe $7 \cdot 7$.

Um in der Umgebung des Bildpunktes in der Position (x, y) systematisch nach Kantenstücken zu suchen, werden benachbarte Bildpunkte inspiziert, ob sie gleiche Gradientenrichtungen haben. Dabei sind unterschiedliche Strategien möglich, zwei davon werden kurz vorgestellt.

Beim ersten Verfahren (Bild 20.9) werden, ausgehend von der in der Position (x, y) gefundenen Richtung (im Beispiel die Richtung 5), die mit einem * markierten Bildpunkte untersucht. Richtungen, die innerhalb einer festzulegenden Toleranz liegen, erhöhen die Systematik. Die Richtungstoleranz ist dabei notwendig, da ja an der gerade untersuchten Position die Kante einen Knick machen könnte. Im Ergebnisbild (*Systematikbild*) sind die Grauwerte Maßzahlen für die Systematik, wobei ein großer Wert eine hohe Wahrscheinlichkeit für eine Kante ausdrückt.

Beim zweiten Verfahren wird ein schmälerer Richtungskanal verwendet als im ersten Verfahren (Bild 20.10). Er wird in drei Teile zerlegt und für jeden Teil werden Eigenschaften

9	8	7	6	5	4	3
10						2
11						1
12			x, y			0
13						23
14						22
15	16	17	18	19	20	21

Bild 20.8: Richtungsquadrat der Größe $7 \cdot 7$.

9	8	7	6*	5*	4*	3
10			*	*	*	2
11		*	*	*		1
12		*	5	*		0
13		*	*	*		23
14	*	*	*			22
15	16*	17*	18*	19	20	21

Bild 20.9: Richtungsquadrat der Größe $7 \cdot 7$. Die Richtungen der markierten Positionen werden bei einer vorgegebenen Richtung (z.B. 5) für den Bildpunkt in der Position (x, y) untersucht, ob sie einen positiven Beitrag zur Systematik liefern oder nicht.

berechnet, wie z.B. die Anzahl der vorkommenden Bildpunkte mit $s_x(x, y) = s_y(x, y) = 0$ (homogene Bildbereiche), die durchschnittliche Richtung auf einem Teilstück und die Summe der absoluten Abweichungen von der Hauptrichtung in der Position (x, y). Aus einer kombinierten Auswertung dieser Eigenschaften kann abgeleitet werden, ob Richtungssystematik vorliegt oder nicht. Das Ergebnisbild (Systematikbild) ist hier ein Binärbild.

Zur letztlich gewünschten Kantenextraktion wird in der Auswertung das Rangbild mit dem Systematikbild verknüpft. Versuche haben gezeigt, dass es besser ist, vom Rangbild auszugehen. Eine einfache UND-Verknüpfung der beiden Binärbilder hat sich nicht bewährt. Vielmehr werden im Rangbild zusammenhängende Kantenstücke von einer festzulegenden Mindestlänge ermittelt. Dann wird versucht, diese Kantenstücke im Systema-

9	8	7	6	5	4	3
10^1						2
11	1	2				1
12			10^2			0
13				2	3	23
14						22^3
15	16	17	18	19	20	21

Bild 20.10: Der Richtungskanal wird in drei Teilstücke aufgespaltet (hier markiert mit ^{1}, ^{2} und ^{3}). Zu jedem Teilstück werden aus den Richtungen der betroffenen Bildpunkte Eigenschaften berechnet.

tikbild ebenfalls zu finden. Ist dies mit einer bestimmten Toleranz erfolgreich, so wird das Kantenstück in das endgültige Kantenbild übernommen. Der Toleranzbereich für die Übereinstimmung kann dabei wie folgt modelliert werden:

$$\alpha = \textit{Größe des Kantenstücks}; \tag{20.10}$$

$$\beta = \frac{\textit{Anzahl der übereinstimmenden Punkte}}{\alpha}.$$

Um zu einem akzeptablen Kantenbild zu kommen, wird eine minimale Kantengröße α und ein minimaler Übereinstimmungsgrad β festgelegt. In der Bildfolge 20.11 ist ein Beispiel zusammengestellt. Bild 20.11-a ist das Originalbild, Bild 20.11-b zeigt das Rangbild mit einer Umgebungsgröße von $7 \cdot 7$, Bild 20.11-c das dazu passende Systematikbild und 20.11-d das Kantenbild als Ergebnis.

Zusammenfassend lautet der gesamte Algorithmus zu diesem Kantendetektor folgendermaßen:

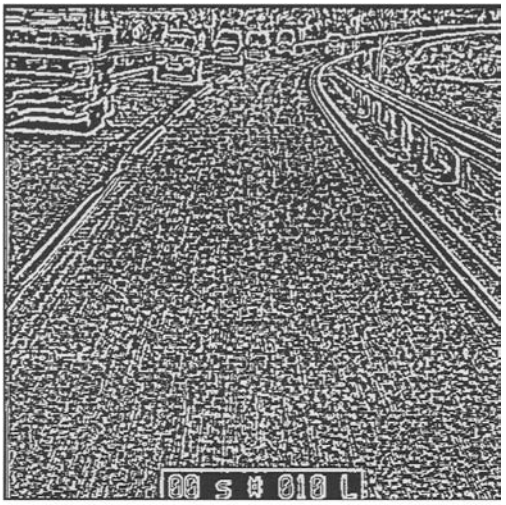
A20.2: Extraktion von Kanten mit Gradientenrichtung und Gradientenbetrag.

Voraussetzungen und Bemerkungen:

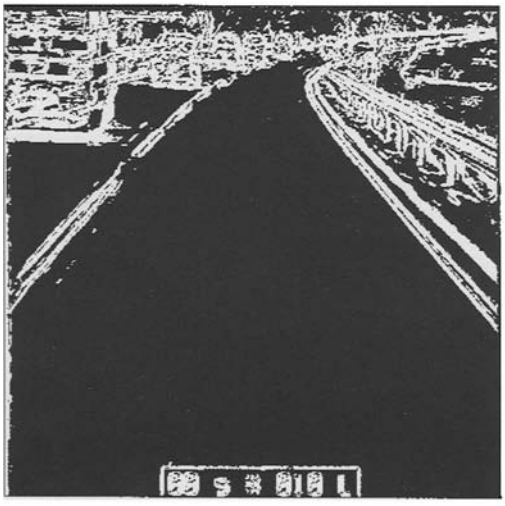
◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild mit $G = \{0,1,...,255\}$ als Grauwertmenge (Eingabebild). Dieses Bild kann auch das Ergebnis mehrerer Vorverarbeitungsschritte sein (z.B. Texturparameter).

◇ $\mathbf{S}_a = (s_a(x,y))$ ein einkanaliges Ausgabebild. Der Grauwert 0 bedeutet „keine Kante“, der Grauwert 1 oder 255 bedeutet „Kante“.

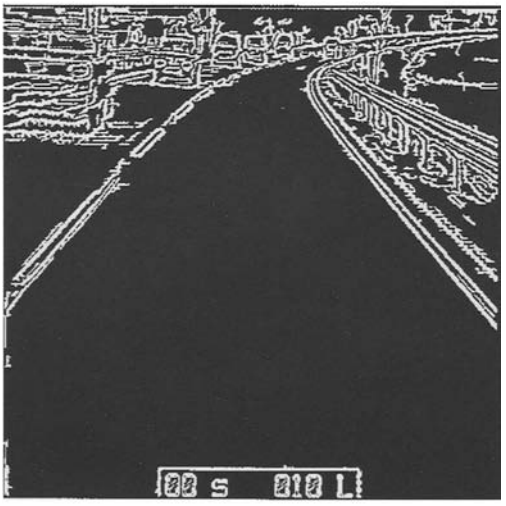
(a)



(b)



(c)



(d)

Bild 20.11: (a) Originalbild einer Straßenszene. (b) Rangbild. (c) Systematikbild. (d) Ergebnis der Kantenextraktion. Bemerkenswert ist, dass selbst die im Original kaum sichtbaren Stützen der Leitplanken am rechten Straßenrand deutlich zu sehen sind.

Algorithmus:

(a) Anwendung eines Differenzenoperators (z.B. Sobeloperator) auf das Eingabebild $\mathbf{S}_e$.

(b) Berechnung des Betrags des Gradienten aus den beiden Richtungskomponenten $s_x(x,y)$ und $s_y(x,y)$ gemäß:

$$g_b(x,y) = \sqrt{s_x(x,y)^2 + s_y(x,y)^2}.$$

(c) Berechnung des Rangbildes, wie es in den obigen Beispielen beschrieben wurde.

(d) Berechnung der Gradientenrichtung $g_r(x,y)$ durch Skalierung der Richtungskomponenten $s_x(x,y)$ und $s_y(x,y)$ auf die Größe des Richtungsquadrats gemäß Bild 20.10. Dies kann mit dem Bresenham-Algorithmus[1] durchgeführt werden. Die skalierten Werte d_x und d_y zeigen direkt auf einen Wert des Richtungsquadrats.

(e) Berechnung der Systematik (hier gemäß dem zweiten dargestellten Beispiel). Hier erfolgt eine Einteilung des Richtungskanals in drei Teile. Die Auswertung der Richtungen der drei Teile kann z.B. folgenden Weg gehen, bei dem Fälle für eine positiv bewertete Richtungssystematik aufgeführt werden:

(e1): Die Richtungsabweichung der gemittelten Richtung aller drei Teile von der Hauptrichtung $g_r(x,y)$ ist kleiner als ein Schwellwert. Im mittleren Teil des Richtungskanals treten keine Richtungen mit $d_x = d_y = 0$ auf. In den beiden anderen Teilen tritt mindestens eine mit der Hauptrichtung identische Richtung auf (gerade Kante).

(e2): Die Richtungsabweichung der gemittelten Richtungen von Teil 1 und Teil 2 ist akzeptabel. Teil 2 und Teil 3 erfüllen diese Forderung jedoch nicht. Die Streuung der Richtungen von Teil 3 liegen über der Schranke (gekrümmte Kante).

(e3): Wie die vorhergehende Bedingung, jedoch sinngemäß für Teil 2 und Teil 3 (gekrümmte Kante).

(e4): In Teil 1 liegen nur Punkte mit $d_x = d_y = 0$. Die Streuungen in Teil 2 und Teil 3 sind akzeptabel. Teil 3 enthält keine Punkte mit $d_x = d_y = 0$ (90°- Ecke).

(e5): Wie die vorhergehende Bedingung, jedoch Teil 1 und Teil 3 vertauscht (90°-Ecke).

(e6): In Teil 1 liegen nur Punkte mit $d_x = d_y = 0$. Teil 2 beginnt mit einem solchen Punkt. Sonst wie die vierte Bedingung (90°-Ecke).

[1] Bresenham-Algorithmus: Eine Fehlergröße e misst den Abstand zwischen dem tatsächlichen Geradenverlauf und dem gerade erzeugten Punkt im Koordinatenraster. Ist e positiv, so ist der Geradenverlauf über dem akuellen Punkt. Es wird die y-Koordinate erhöht und 1 von e abgezogen. Ist e negativ, so bleibt die y-Koordinate unverändert. Der Vorteil von diesem Algorithmus ist, dass er auch rein ganzzahlig implementiert werden kann.

(e7): In Teil 3 liegen nur Punkte mit $d_x = d_y = 0$. Teil 2 endet mit einem solchen Punkt. Sonst wie die fünfte Bedingung (90^o-Ecke).

Mit diesen Bedingungen wird für den Bildpunkt in der Position (x, y) ermittelt, ob eine Systematik vorliegt oder nicht.

(f) Berechnung des Kantenbildes durch eine Kombination des Rang- und des Systematikbildes wie oben beschrieben. Die Bewertung erfolgt dabei gemäß (20.10).

Ende des Algorithmus

Abschließend zu diesem Algorithmus noch einige Bemerkungen: Durch die Auswertung des Rang- und des Systematikbildes werden hier positive Eigenschaften der einzelnen Verarbeitungsschritte kombiniert. Sowohl Rang- als auch Systematikbild ermöglichen eine betragsunabhängige Kantendetektion. Die Rangauswertung liefert eine gute Geschlossenheit von Kantenzügen, während die Systematikauswertung bei der Entfernung von irrelevanten Kantenstücken in homogenen oder verrauschten Bildbereichen herangezogen wird. Beide Zwischenschritte (Rang und Systematik) und das Endergebnis sind Binärbilder (logische Bilder), in denen die Grauwerte der Bildpunkte die Bedeutung „liegt auf einer Kante“ und „liegt auf keiner Kante“ haben.

An vielen Stellen dieses Kantendetektors werden Schwellwerte verwendet. Das hat zur Folge, dass das Verfahren sehr sorgfältig auf die jeweilige Problemstellung zu adaptieren ist. Das Ergebnis wird dabei sicher verbessert, wenn in der Vorverarbeitung Störungen (Rauschen, Scannerfehler, usw.) korrigiert werden.

Da einzelne Verarbeitungschritte durchaus rechenzeitintensiv sind, vor allem, wenn größere Umgebungen verwendet werden, wird man bei Echtzeitanwendungen möglicherweise Probleme bekommen. Hier kann die Verwendung von sehr schnellen allgemeinen Prozessoren, die Parallelisierung von Teilabläufen oder die Verwendung von Spezialhardware Abhilfe schaffen. Bei vielen praktischen Anwendungen ist es außerdem nicht notwendig, den gesamten Bildbereich zu verarbeiten, sondern die Verarbeitung kann auf einen kleinen Bildausschnitt (*area-of-interest*) beschränkt werden.

20.6 Der Canny-Kantendetektor

Einen Kantendetektor mit einigen interessanten Eigenschaften stellt der Canny-Kantendetektor dar. Er wurde von Canny 1983 [Cann83, Cann86] vorgestellt. Damals waren schon eine Vielzahl von Operatoren bekannt, Canny's Beitrag war, mit einem mathematisch-signaltheoretischen Ansatz aus (mathematisch formulierten) Designkriterien einen bezüglich dieser Kriterien optimalen Kantendetektor abzuleiten.

Informell ausgedrückt sind dies folgende Kriterien:

- Geringe Fehler erster und zweiter Art, d.h. es sollen keine Kanten übersehen werden und keine Kanten an Stellen detektiert werden, an denen sich keine befinden,

- Genaue Lokalisierung von Kanten, d.h. ein Detektor sollte Kanten möglichst genau dort lokalisieren, wo sie auch im Originalbild zu finden sind,
- Nur eine Antwort des Detektors auf eine Kante, nicht mehrere Antworten an mehreren Stellen (im wesentlichen als Ergänzung zum zweiten Kriterium).

Canny betrachtet dabei das eindimensionale Signal senkrecht zur jeweils betrachteten Kante im Bild. Als bezüglich der obigen Kriterien optimalen Kantendetektor ermittelt er das Maximum der Ableitung des rauschgefilterten Signals.

Die Realisierung des Canny-Kantendetektors erfolgt - wie schon in Canny's Originalarbeit skizziert - meist in einem dreischrittigen Verfahren:

- Glättung des Bildes mit einem Gauß'schen Kern
- Bestimmung des Gradienten (Kantenfilterung)
- Kantentracking (Nicht-Maxima-Unterdrückung und Tracking mit Hysterese)

A20.3: Canny-Kantendetektor.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x, y))$ ein Grauwertbild;

◇ Eingabeparameter des Canny-Kantendetektors: σ: Standardabweichung für die Glättung mit einem Gauß'schen Kern; h, l: oberer und unterer Schwellwert für das Kantentracking mit Hysterese;

Algorithmus:

(a) Glätte $\mathbf{S}_e$ mit einem Gauß'schen Kern mit Standardabweichung σ;

(b) Bestimmung des Gradienten (Kantenfilterung):

(ba) Wende auf $\mathbf{S}_e$ Kantenfilteroperatoren in zwei orthogonalen Richtungen an;

(bb) Bilde aus dem Ergebnis Gradientenbetrag und -richtung;

(c) Kantentracking:

(ca) Unterdrücke im Gradientenbetragsbild diejenigen Pixel, die entlang der Gradientenrichtung kein Maximum darstellen;

(cb) Verfolge Kanten mit Hysterese: Ausgehend von Pixeln, die über der oberen Schwelle h liegen, markiere alle Pixel, die über der unteren Schwelle l liegen als zur Kante gehörend.

Ende des Algorithmus

Bei der Glättung des Bildes wird ein Gauß'scher Kern mit - je nach Verrauschtheit des Bildmaterials - vorwählbarer Standardabweichung σ verwendet. Diese ist einer der Parameter des Canny-Kantendetektors. Ziel dieses Schrittes ist die Verminderung des Bildrauschens, da die hochfrequenten Rauschanteile sich nachteilig auf den nachfolgenden Schritt der Gradientenbildung auswirken.

Zur Gradientenbildung wird meist einer der bekannten Kantenfilteroperatoren in zwei orthogonalen Richtungen angewendet (Robert's Cross, Prewitt, Sobel, Bild 20.12-b,c), um daraus Betrag und Richtung des Gradienten zu bestimmen.

Im Gradienten-Betragsbild (Bild 20.12-d) werden dann diejenigen Pixel "unterdrückt", d.h. zu Null gesetzt (*non-maxima-suppression*), die entlang der Gradientenrichtung (also senkrecht zur Kante) kein Maximum aufweisen. Canny schlägt vor, für die Bestimmung der Nachbarpixel entlang des Gradienten zwischen den entsprechenden Pixeln der 8-er Nachbarschaft zu interpolieren. Allerdings wird hierzu auch häufig die Gradientenrichtung meist in die vier (betragsfreien) Richtungen der Achter-Nachbarschaft (*horizontal, vertikal, entlang der Hauptdiagonalen, entlang der Nebendiagonalen*) diskretisiert und die Maximumsunterdrückung entlang der drei benachbarten Pixel in dieser Richtung durchgeführt.

Auf den übrigbleibenden, Kanten im Bild entsprechenden Pixeln wird dann ein Linienverfolgungsverfahren mit Hysterese angewendet: Um schwache, oft auch aus Restrauschen entstehende, Kanten zu unterdrücken, werden nur diejenigen Kanten verwendet, bei denen mindestens ein Pixelwert über einer oberen Schranke h liegt. Um für diese Kanten Unterbrechungen durch schwächere Kantenbereiche zu vermeiden, wird eine Hysterese eingeführt, d.h. für eine einmal erkannte Kante reicht es aus, wenn alle Pixel über einer unteren Schranke l liegen.

Die obere und untere Schwellwerte (h, l) bilden neben der oben erwähnten Standardabweichung die anderen beiden der drei Parameter des Canny-Kantendetektors.

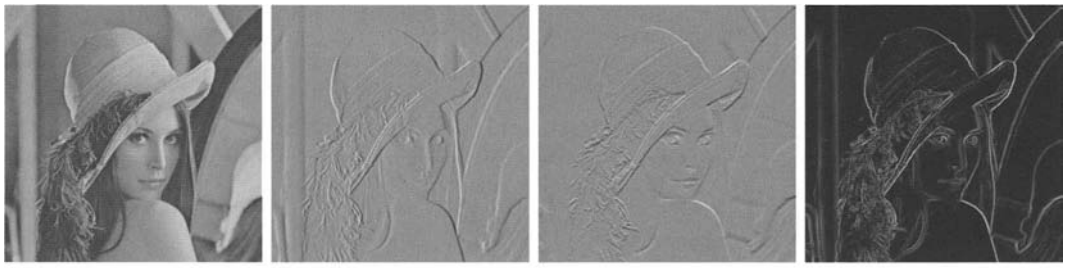


Bild 20.12: (a) Die Original-Lena, (b) Gradient dx, (c) Gradient dy, (d) Betrag des Gradienten

Bild 20.13: Auswirkungen unterschiedlicher Standardabweichungen zum Glätten des Bildes: (a) $\sigma = 1.0$, (b) $\sigma = 2.0$, (c) $\sigma = 3.0$, (d) $\sigma = 4.0$,

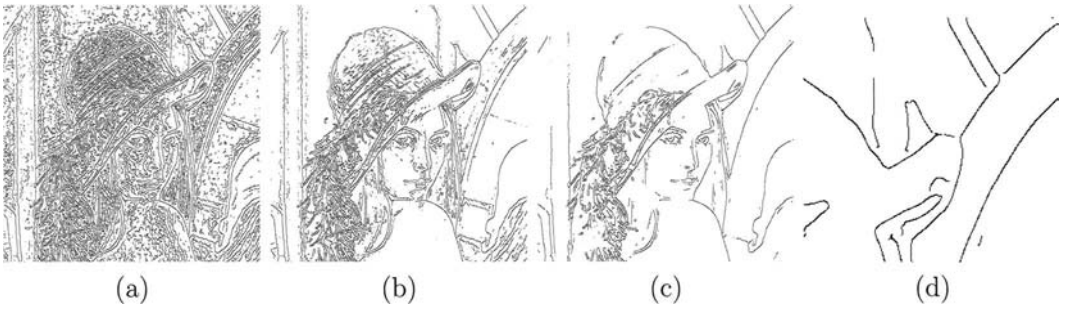


Bild 20.14: Auswirkung unterschiedlicher Hystereseschwellen für das Kantentracking, leichte Glättung ($\sigma = 4.0$), starkes verbleibendes Rauschen: (a) $h = 0.8, l = 0.3$,(b) $h = 1.0, l = 0.6$,(c) $h = 1.0, l = 0.8$,(d) Detail mit offenen Y-Verbindungen

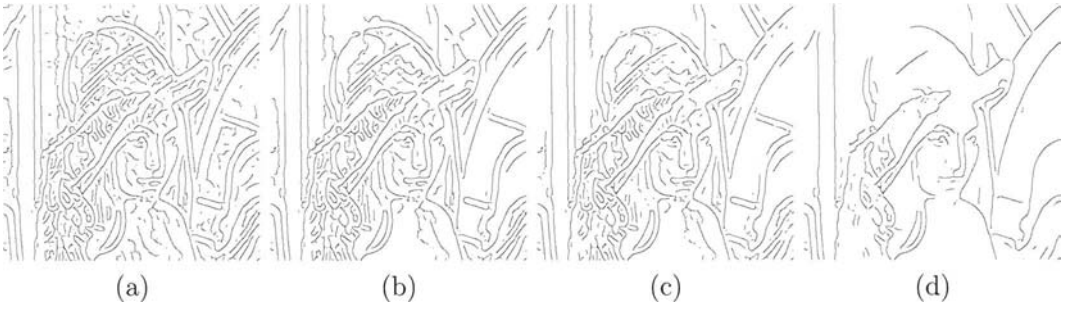


Bild 20.15: Auswirkung unterschiedlicher Hystereseschwellen für das Kantentracking, starke Glättung ($\sigma = 4.0$), leichtes verbleibendes Rauschen: (a) $h = 1.0, l = 0.1$,(b) $h = 0.3, l = 0.3$,(c) $h = 1.0, l = 0.3$,(d) $h = 1.0, l = 0.6$

Sollte die Topologie der detektierten Kanten in der jeweiligen Anwendung eine besondere Rolle spielen, so ist auf ein spezielles 'Feature' des Canny-Kantendetektors zu achten: Bedingt durch die Nicht-Maxima-Unterdrückung beim Kantentracking werden 'Y'-Verbindungen dreier (oder Kreuzungen mehrerer) Kanten nicht vollständig detektiert (siehe Detail aus der rechten, oberen Ecke des Originalbilds im Teilbild 20.14-d). Hier ist gegebenenfalls ein Nachbearbeitungsschritt zum Schließen der Y-Verbindungen oder eine alternative Trackingstrategie erforderlich.

20.7 Kanten und Linien mit morphologischen Operationen

Die mathematische Morphologie stellt der digitalen Bildverarbeitung und Mustererkennung mächtige Instrumente zur Bewältigung vieler Problemstellungen zur Verfügung. Die Grundlagen dazu sind in Kapitel 19 zusammengestellt. In diesem Abschnitt werden morphologische Verfahren erläutert, die im Zusammenhang mit der Verarbeitung von Kanten und Linien interessant sind.

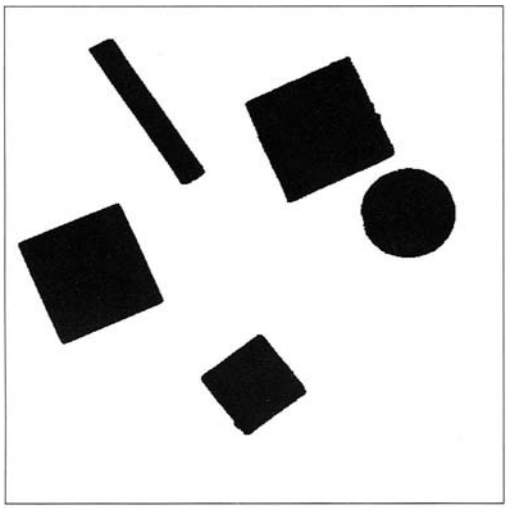
Zuvor aber noch eine Zusammenfassung, wie Kanten- oder Linienbilder entstehen können: Grauwertkanten treten in Grauwertbildern an Helligkeitsübergängen auf, die eine gewisse Systematik aufweisen. Grauwertkanten können aber auch, wie schon oben dargestellt, das Ergebnis einer Textur- oder Farbmerkmalsverarbeitung sein. Wird auf ein derartiges Grauwertbild ein Verfahren zur Kantenextraktion angewendet (Abschnitte 18.4, 20.3 und 20.4), so sind im Ergebnisbild die markanten Grauwertübergänge in der Regel durch Linien dargestellt. Das so erzeugte Linienbild kann ein Grauwertbild oder ein Binärbild sein. Ist es ein Grauwertbild, so sagt ein hoher Wert aus, dass die Wahrscheinlichkeit, dass der Bildpunkt auf einer Grauwertkante liegt, hoch ist. Bei einem binären Linienbild bedeutet der Grauwert 0, dass der Bildpunkt zum Hintergrund, und der Grauwert 1, dass er zu einer Kante gehört.

Linienbilder können natürlich auch das direkte Ergebnis einer Digitalisierung sein, so z.B. wenn eine Strichgrafik entweder per Software oder mit einem Digitalisierungsgerät (Videosensor, Scanner) in ein Rasterbild umgewandelt wird.

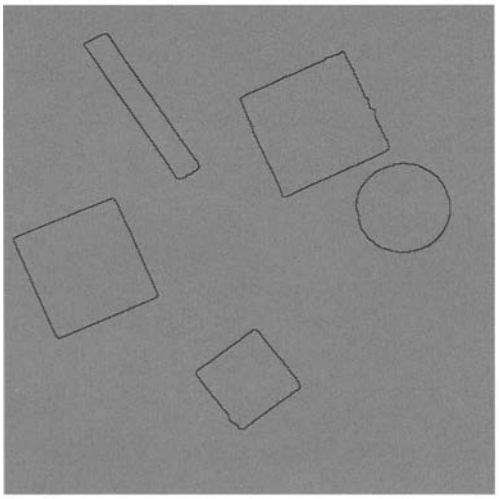
20.7.1 Extraktion des Randes von Segmenten

Ein einfaches Verfahren, wie man aus einem Binärbild, das unterschiedliche Segmente enthält, den Rand dieser Segmente gewinnen kann, ist eine Dilatation oder Erosion mit einer anschließenden Differenz zum Originalbild. Vorausgesetzt wird ein Binärbild (Zweipegelbild), in dem sich die Segmente dunkel (Grauwert 0) vor einem hellen Hintergrund (Grauwerte 1 oder 255) abheben.

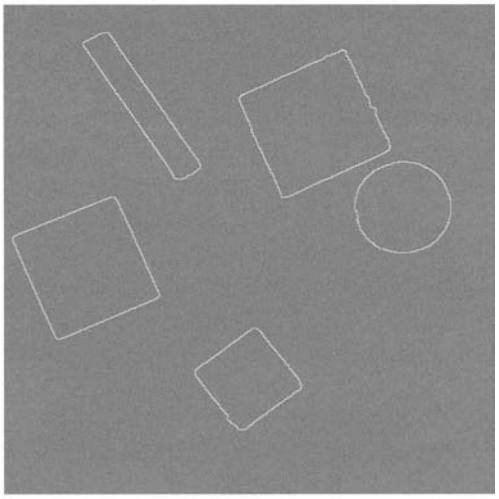
Bei einer Dilatation und einer anschließenden Differenzbildung des dilatierten Bildes mit dem Original erhält man den inneren Rand der Segmente, während bei der Erosion und einer anschließenden Differenzbildung Ränder erzeugt werden, die die Segmente ganz enthalten (Bilder 20.16).



(a)



(b)



(c)

Bild 20.16: (a) Originalbild: Dunkle Segmente auf hellem Hintergrund. (b) Dilatation und Differenz (Grauwert 127 addiert). (c) Erosion und Differenz (Grauwert 127 addiert).

Dieses Verfahren ist auch bei Grauwertbildern möglich. Wird eine Dilatation durchgeführt, so vergrößern sich die hellen Bildbereiche auf Kosten der dunklen. Nach einer Differenz mit dem Originalbild erhält man die Grauwertübergänge als Linien. Ähnlich wie bei den Differenzenoperatoren aus Abschnitt 18.4 ergibt sich auch hier das Problem, dass die Beträge dieses *morphologischen Gradienten* unterschiedliche Werte erhalten und so eine einfache Binarisierung oft nicht immer das gewünschte Ergebnis erzielt. Hier kann auch eine Betrachtung mit Positionsrang und Systematik (Abschnitt 20.5) weiter helfen. Die Bildfolge 20.17 zeigt ein Beispiel zum morphologischen Gradienten.

20.7.2 Verarbeitung von Linien

Bei der Binarisierung von Linienbildern lassen sich ebenfalls morphologische Operationen einsetzen. Handelt es sich z.B. um eine digitalisierte Strichgrafik, die mit einem Scanner oder einer Videokamera aufgezeichnet wurde, so kann es vorkommen, dass der Hintergrund durch ungleichmäßige Ausleuchtung oder durch Randabschattungen (Vignettierungen) gestört ist. Der Schwellwert bei der Binarisierung wird dann ortsabhängig. Dieses Problem kann manchmal mit einem dynamischen Schwellwert (Abschnitt 17.5) gelöst werden.

Wenn es möglich ist, durch eine *closing*-Operation in einem Eingabebild mit hellem Hintergrund die Linien ganz verschwinden zu lassen, so bleibt nur mehr der Hintergrund übrig. Durch eine Differenzbildung mit dem Original und eventuell der Addition einer Konstanten erhält man ein Ausgabebild, in dem der störende Hintergrund ausgeblendet ist. Dieses Bild kann dann gut binarisiert werden (Bildfolge 20.18).

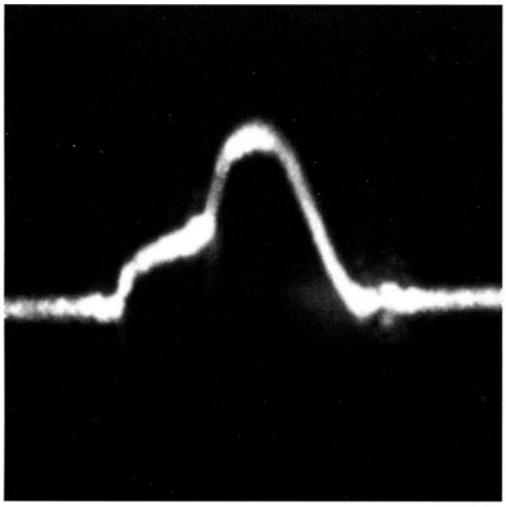
Auch nach einer Binarisierung können Linienbilder mit morphologischen Operationen korrigiert, verbessert und weiter verarbeitet werden. Eine einfache Korrektur besteht z.B. in der Elimination einzelner, verstreuter Bildpunkte (Abschnitt 18.5).

Häufig tritt auch der Fall ein, dass Linien unterbrochen sind, die eigentlich geschlossen sein sollten. Bei einer Vorzugsrichtung der Linien ist das Schließen von Lücken mit einem strukturierenden Element, das die Vorzugsrichtung berücksichtigt, einfach möglich. Bei einer Erosion des Vordergrundes schließen sich die Lücken, da die Erosion hier nur in der gewählten Richtung wirkt. Ein Beispiel dazu zeigt die Bildfolge 20.19.

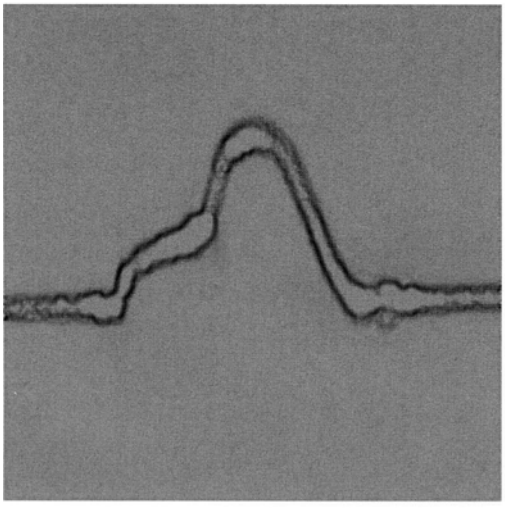
20.8 Skelettierung mit morphologischen Operationen

Nachdem ein Linienbild mit den in Abschnitt 20.7 beschriebenen Verfahren binarisiert wurde, ist eine oft benötigte Weiterverarbeitung das Verdünnen der Linien, so dass nur mehr eine ein Pixel breite Linie verbleibt, die in der Mitte der ursprünglichen, breiteren Linie verläuft. Diese Vorgehensweise nennt man *Skelettierung*. Ein Beispiel einer Anwendung hierzu wurde im vorhergehenden Abschnitt mit der Bildfolge 20.19 gegeben.

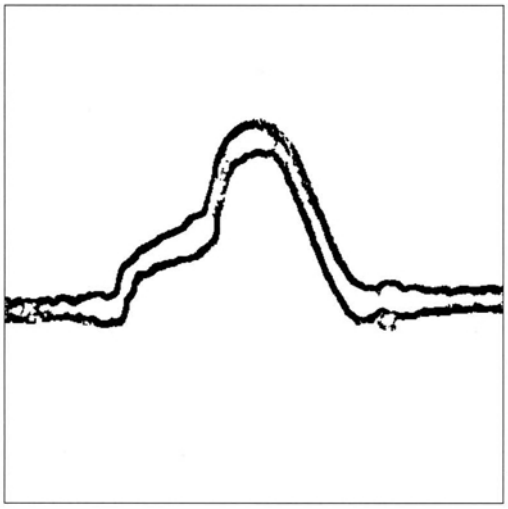
Die Skelettlinie oder das Skelett einer binär dargestellten Fläche (eines Segments) ist durch eine Reihe von Forderungen festgelegt, die jedoch nicht als exakte, mathematische Definitionen aufgefasst werden können. Das ist auch der Grund, warum sich in der ein-



(a)



(b)



(c)

Bild 20.17: Beispiel zum morphologischen Gradienten im Grauwertbild. (a) Laserlichtband, das mit einer Spaltlinse auf eine Kleberaupe projiziert wurde. (b) Ergebnis nach zwei Dilatationen und einer Subtraktion vom Original (skaliert). (c) Binarisiertes Ergebnis.

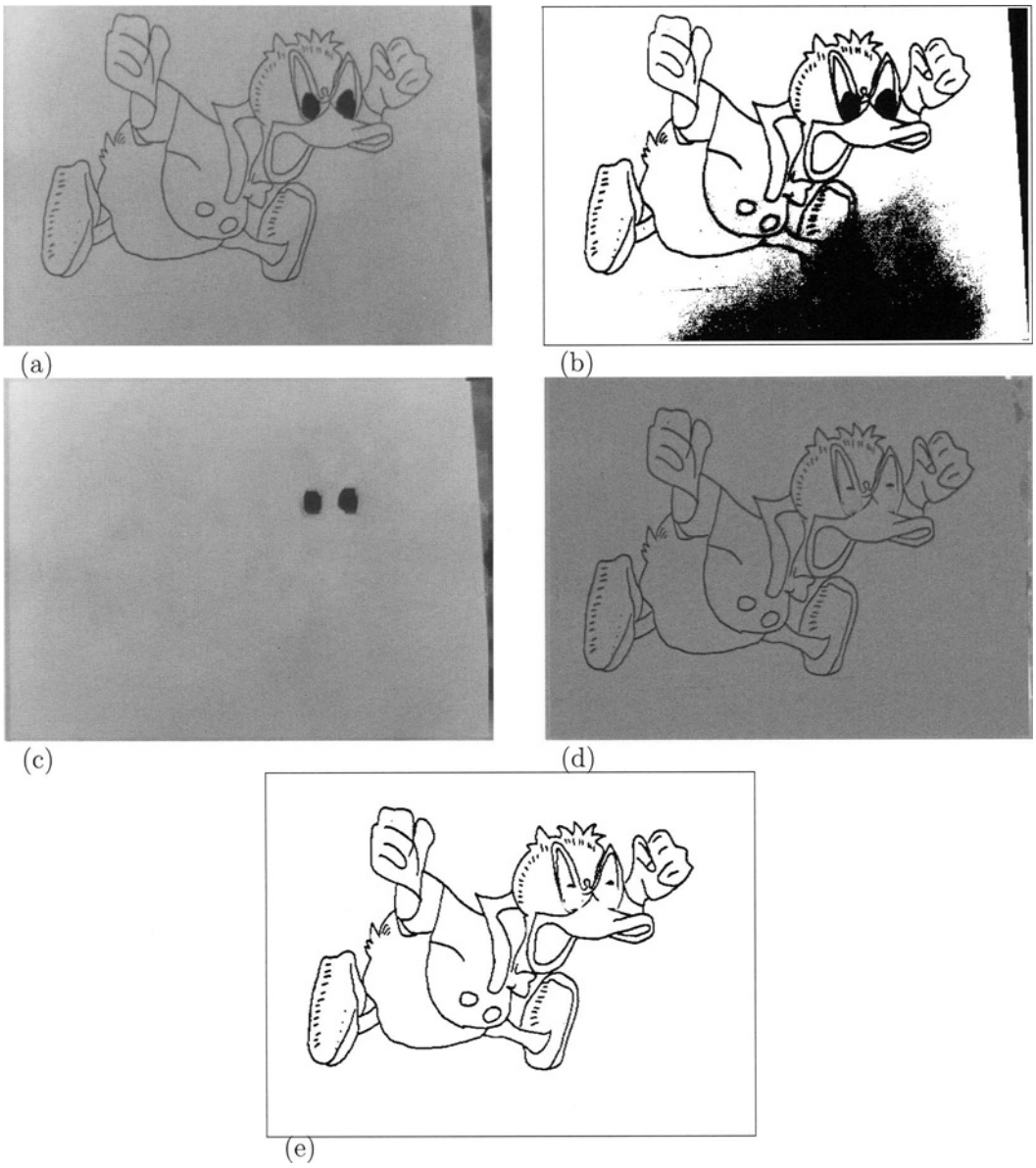


Bild 20.18: Beispiel zur Korrektur eines ungleichmäßig ausgeleuchteten Bildhintergrundes. (a) Originalbild. (b) Binärbild mit fixem Schwellwert: Der rechte untere Rand läuft zu. (c) Bildhintergrund nach einer *closing*-Operation (fünf Iterationen). (d) Differenz mit dem Original. (e) Binarisierung. Im Bereich der Augen ist der Bildvordergrund durch die *closing*-Operation nicht vollständig eliminiert worden. Damit sind die Störungen zu erklären. Das kann aber auch nutzbringend eingesetzt werden. In diesem Beispiel konnte dadurch der Keil am rechten Bildrand entfernt werden.

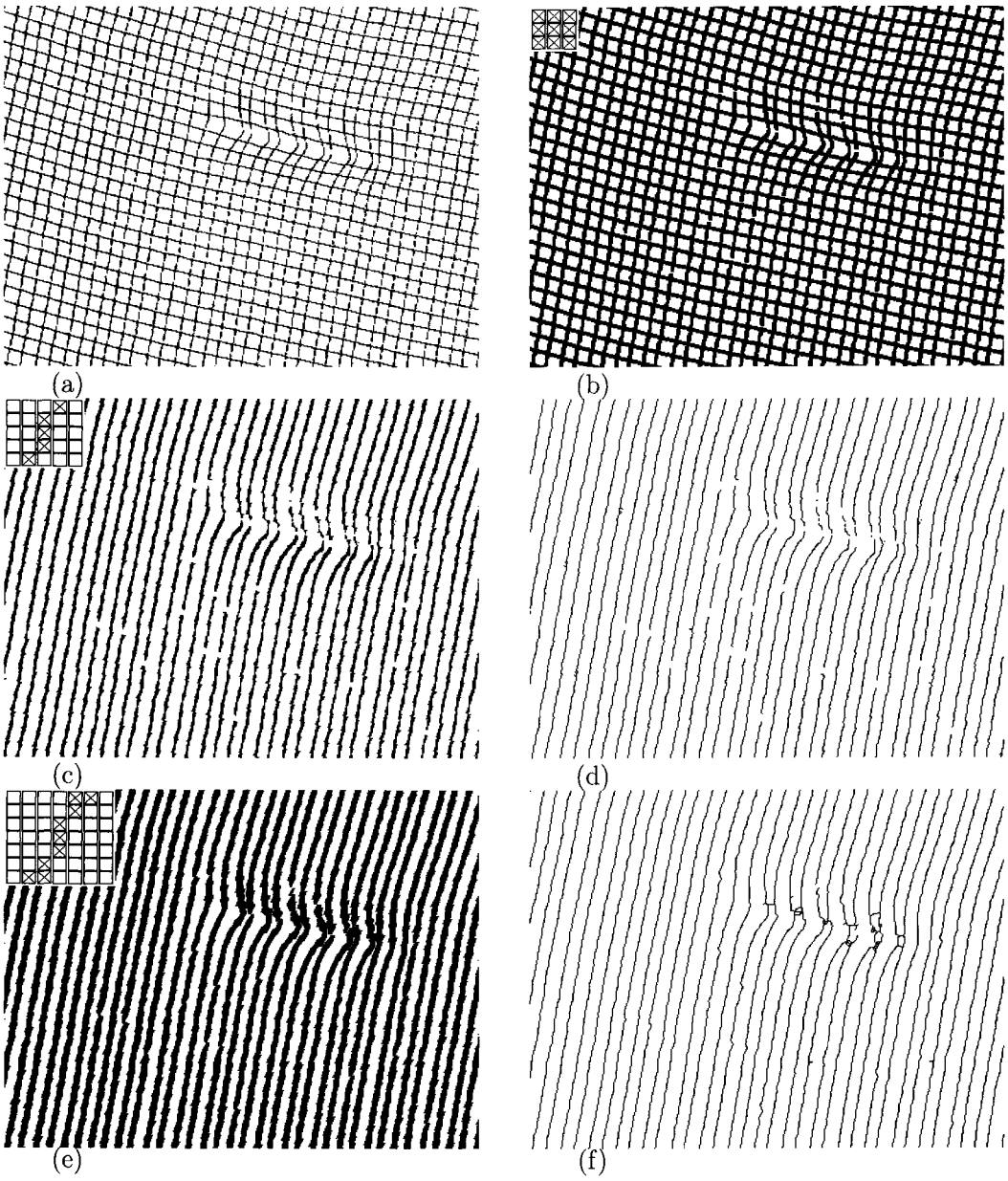


Bild 20.19: Beispiel zum Schließen von Lücken in einem binarisierten Linienbild. (a) Originalbild. Die Aufgabe besteht darin, die vertikalen Linien möglichst geschlossen zu extrahieren. (b) Um die Linien etwas zu verdicken, wurde eine Erosion mit dem links oben eingeblendeten strukturierenden Element durchgeführt. (c) Zur Elimination der horizontalen Linien wurde eine Dilatation mit dem links oben eingeblendeten strukturierenden Element durchgeführt. (d) Skelettierung (siehe nächster Abschnitt) zur Verdünnung der Linien. Einige Linien sind unterbrochen. (e) Erosion von Teilbild (c) mit dem links oben eingeblendeten strukturierenden Element. (f) Skelettierung zur Verdünnung der Linien. Die Linien sind, bis auf die Fehlerstelle, geschlossen.

schlägigen Literatur eine große Anzahl von Arbeiten mit dieser Thematik befassen und dabei zu recht unterschiedlichen Ergebnissen kommen. Forderungen zur Skelettlinie lauten:

- Die Skelettlinie soll nur einen Bildpunkt breit sein.
- Die Skelettlinie soll ungefähr in der Mitte des Segments liegen.
- Das Skelett soll die ursprüngliche Form des Segments widerspiegeln. So darf das Skelett einer einfach zusammenhängenden Fläche nicht in mehrere Teile zerfallen.
- Das Skelettierungsverfahren soll unempfindlich gegen kleinere Störungen am Rand des Segments sein. Empfindliche Verfahren erzeugen bei ausgefransten Rändern viele Verästelungen.
- Der Skelettierungsalgorithmus muss nach einer bestimmten Anzahl von Iterationen stabil werden, d.h. es dürfen sich keine Änderungen mehr ergeben.

Es gibt unterschiedliche Verfahren zur Skelettierung. Eine Klasse von Verfahren geht vom Binärbild aus und entscheidet für alle Vordergrundpixel, ob sie entfernt werden können oder nicht. Die Entscheidung, ob ein gesetzter Vordergrundbildpunkt entfernt, also dem Hintergrund zugewiesen werden kann (das *Tilgungskriterium*), wird anhand von Umgebungen (meistens $3 \cdot 3$) entschieden.

Eine andere Klasse von Skelettierungsalgorithmen baut auf Konturverfolgungsverfahren auf. Hier liegt der Gedanke zugrunde, dass man nur diejenigen Bildpunkte überprüfen will, die auch tatsächlich zum Rand des Segments gehören. Man wird hier also die Randpunkte des Segments in einer Listenstruktur führen. Wenn ein Randpunkt aus dieser Liste entfernt wird, wird für diesen ein neuer, bisher nicht auf dem Rand liegender Segmentpunkt eingefügt.

Im Folgenden werden zwei Skelettierungsverfahren beschrieben, die beide zur ersten der oben dargestellten Klassen gehören.

In Kapitel 19 wurden neben den Grundlagen zur mathematischen Morphologie auch schnelle Implementierungsmöglichkeiten durch Verschiebeoperationen und Boole'sche Operationen erläutert. Diese Technik wird hier verwendet. Es wird im Folgenden angenommen, dass ein Binärbild mit den Grauwerten 0 (Hintergrundbildpunkt) und 1 (Vordergrundbildpunkt) vorliegt. Zunächst werden acht $3 \cdot 3$–Masken vorgegeben:

$$\begin{pmatrix} 0 & b & b \\ 0 & 1 & 1 \\ 0 & b & 1 \end{pmatrix} \begin{pmatrix} b & 1 & b \\ 0 & 1 & 1 \\ 0 & 0 & b \end{pmatrix} \begin{pmatrix} b & 1 & 1 \\ b & 1 & b \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} b & 1 & b \\ 1 & 1 & 0 \\ b & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 1 & b & 0 \\ 1 & 1 & 0 \\ b & b & 0 \end{pmatrix} \begin{pmatrix} b & 0 & 0 \\ 1 & 1 & 0 \\ b & 1 & b \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 \\ b & 1 & b \\ 1 & 1 & b \end{pmatrix} \begin{pmatrix} 0 & 0 & b \\ 0 & 1 & 1 \\ b & 1 & b \end{pmatrix} \tag{20.11}$$

Diese Masken beschreiben Konstellationen von $3 \cdot 3$ Bildpunkten, bei denen der mittlere Punkt von der Segmentfläche entfernt werden darf. Die Masken werden von links oben nach rechts unten mit $\mathbf{M_0}$ bis $\mathbf{M_7}$ bezeichnet. Dabei bedeuten die mit b (für „beliebig") besetzten Positionen, dass an diesen Stellen entweder eine 0 oder eine 1 stehen kann.

Nach Maßgabe dieser Masken wird nun das Originalbild $\mathbf{S}_e$ in ein Hilfsbild $\mathbf{H}$ verschoben. Die Verschiebungsrichtungen werden dabei von 0 (östlich), bis 7 (südöstlich) numeriert. Abhängig davon, ob in der Maske in der Verschiebungsposition eine 0 oder 1 steht, wird das Hilfsbild negiert oder nicht. Positionen, die mit b besetzt sind, werden nicht berücksichtigt. Die Hilfsbilder für jede Verschiebungsrichtung werden alle mit UND verknüpft. Das so entstehende Hilfsbild $\mathbf{T}$ enthält alle Bildpunkte, die im Eingabebild $\mathbf{S}_e$ die Konstellation einer bestimmten Maske erfüllen. Dieses Bild $\mathbf{T}$ wird mit EXODER (exklusives ODER) mit dem Original verknüpft, wodurch die entsprechenden Pixel gelöscht werden. Für die Maske $\mathbf{M_0}$ ist diese Vorgehensweise im folgenden dargestellt:

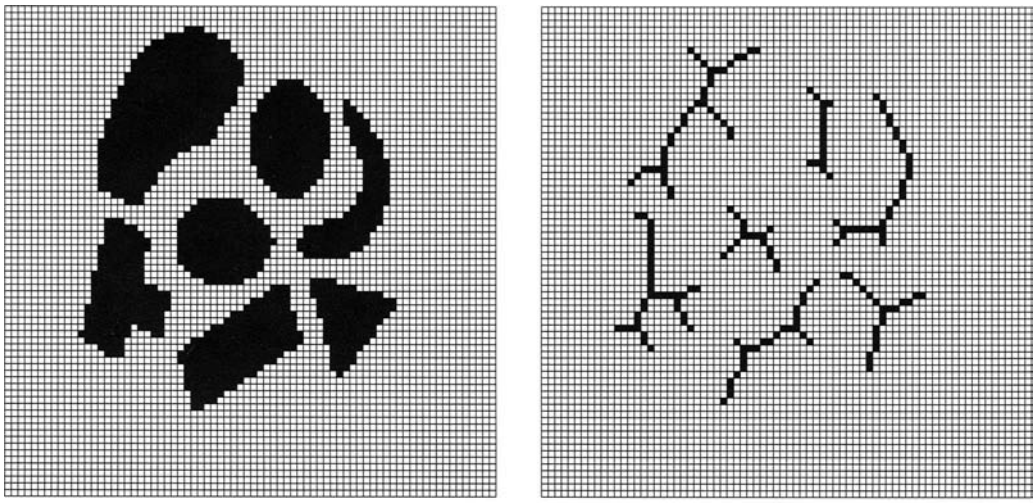
Maske M_0 :

$$
\begin{aligned}
&\text{SHIFT}(\mathbf{S}_e, \mathbf{H}, 0)\\
&\text{NOT}(\mathbf{H}, \mathbf{H})\\
&\text{AND}(\mathbf{S}_e, \mathbf{H}, \mathbf{T})\\
&\text{SHIFT}(\mathbf{S}_e, \mathbf{H}, 1)\\
&\text{NOT}(\mathbf{H}, \mathbf{H})\\
&\text{AND}(\mathbf{T}, \mathbf{H}, \mathbf{T})\\
&\text{SHIFT}(\mathbf{S}_e, \mathbf{H}, 3)\\
&\text{AND}(\mathbf{T}, \mathbf{H}, \mathbf{T})\\
&\text{SHIFT}(\mathbf{S}_e, \mathbf{H}, 4)\\
&\text{AND}(\mathbf{T}, \mathbf{H}, \mathbf{T})\\
&\text{SHIFT}(\mathbf{S}_e, \mathbf{H}, 7)\\
&\text{NOT}(\mathbf{H}, \mathbf{H})\\
&\text{AND}(\mathbf{T}, \mathbf{H}, \mathbf{T})\\
&\text{EXODER}(\mathbf{S}_e, \mathbf{T}, \mathbf{S}_e)
\end{aligned}
\tag{20.12}
$$

Der dritte Parameter bei den SHIFT-Operationen gibt die Verschiebungsrichtung, wie oben festgelegt, an. Die Operationen von (20.12) werden nacheinander sinngemäß für alle Masken $\mathbf{M_0}$ bis $\mathbf{M_7}$ in einem Iterationsschritt durchgeführt. Es werden so viele Iterationen angeschlossen, bis in einem Schritt keine Bildpunkte mehr entfernt werden. Das Ergebnisbild enthält dann das Skelett. Der gesamte Algorithmus lautet somit wie folgt:

A20.4: Skelettierung mit morphologischen Operationen.

Voraussetzungen und Bemerkungen:



(a) (b)

Bild 20.20: Beispiel zur Sklettierung mit morphologischen Operationen. (a) Originalbild. (b) Skelett.

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Binärbild; $G = \{0,1\}$.

◇ $\mathbf{H}$ und $\mathbf{T}$ zwei Hilfsbilder.

Algorithmus:

(a) Setze $removed = 1$;

(b) Solange gilt $removed = 1$:

(ba) Setze $removed = 0$.

(bb) Führe die Operationen (20.12) sinngemäß für alle Masken $\mathbf{M_0}$ bis $\mathbf{M_7}$ durch.

(bc) Falls in einem Iterationsschritt Bildpunkte entfernt wurden: Setze $removed = 1$.

Ende des Algorithmus

Ein Beispiel zu diesen Verfahren zeigt Bild 20.20.

Bei dieser Skelettierungsmethode sind viele Verschiebeoperationen und logische Operationen notwendig. Das setzt voraus, dass diese grundlegenden Verknüpfungen von (Binär-)Bildern sehr schnell durchgeführt werden. Nur dann wird der dargestellte Algorithmus in akzeptabler Zeit ablaufen. Da es sich um einfache Einzeloperationen handelt, kann dieses Verfahren gut mit spezieller Hardware realisiert werden.

Ein anderes Skelettierungsverfahren, das bezüglich der Rechenzeit wesentlich schneller abläuft, ist im nächsten Abschnitt beschrieben. Allerdings liefert dieses Verfahren eine etwas „schlechtere" Skelettlinie.

20.9 Skelettierung mit der Euler'schen Charakteristik

Dem zweiten Verfahren zur Skelettierung liegt die *Euler'sche Charakteristik* zugrunde. Die Euler'sche Charakteristik für Polygonnetze und für Polyeder ist die Zahl $E = e - k + f$, wobei e für die Zahl der Ecken, k für die Zahl der Kanten und f für die Zahl der Flächen (Polygonbereiche) steht. Der Euler'sche Polygonsatz sagt nun, dass für ein Polygonnetz $E = 1$ gilt. Im dreidimensionalen Raum gilt für (schlichtartige) Polyeder $E = 2$. Dazu ein einfaches Beispiel: Ein Würfel besitzt 8 Ecken, 12 Kanten und 6 Flächen, also ist $E = 8 - 12 + 6 = 2$.

Diese Betrachtungsweise lässt sich auch auf Binärbilder anwenden. Die Zahl E gibt hier die Anzahl der Segmente an. Die Bedeutung der Größen e, k und f ist etwas anders als oben. Hier ist e die Anzahl der Ecken der (z.B. schwarzen) Vordergrundbildpunkte, k die Anzahl der Kanten der Vordergrundbildpunkte und f die Anzahl der Vordergrundbildpunkte. Einen Vordergrundbildpunkt stellt man sich hier am besten als ein Quadrat mit vier Ecken, vier Kanten und einer Flächeneinheit vor. Gemeinsame Eckpunkte oder Kanten von Bildpunkten werden nur einmal gezählt. Bild 20.21 zeigt dazu einige Beispiele: Für das Binärbild 20.21-a ergibt sich $e = 11$, $k = 15$ und $f = 5$, also $E = 1$. Teilbild 20.21-b ergibt ebenfalls $E = 1$ ($e = 11$, $k = 14$ und $f = 4$). Das $5 \cdot 5$–Bild 20.21-c besitzt die Werte $e = 29$, $k = 38$ und $f = 12$, somit $E = 3$. Man sieht, dass E hier die Anzahl der einfach zusammenhängenden Segmente angibt. Diese Eigenschaft der Euler'schen Charakteristik, auf die in Abschnitt 35.10 nochmals eingegangen wird, kann zum Zählen der Segmente in einem Binärbild verwendet werden.

Wie kann die Euler'sche Charakteristik zur Skelettierung verwendet werden? Zur Berechnung der Entscheidung, ob ein Vordergrundbildpunkt entfernt werden soll, werden 3·3–Umgebungen betrachtet. Ein Bildpunkt darf entfernt werden, wenn sich dadurch die Anzahl der zusammenhängenden Flächen nicht ändert. So kann bei Bild 20.21-a der Bildpunkt in der Mitte dem Hintergrund zugewiesen werden. Wenn sich die Anzahl der zusammenhängenden Flächen nicht ändert, ist die Euler'sche Charakteristik vor und nach dem Entfernen gleich (in diesem Fall $E = 1$). Bei Bild 20.21-b darf der mittlere Punkt nicht entfernt werden, sonst würden zwei nicht zusammenhängende Flächen entstehen. Hier würde sich die Euler'sche Charakteristik von $E = 1$ auf $E = 2$ ändern.

Für den Skelettierungsalgorithmus heisst das, dass alle Punkte des Binärbildes im Rahmen ihrer $3 \cdot 3$-Umgebung untersucht werden. Ein Bildvordergrundpixel darf nur dann entfernt werden, wenn die Euler'sche Charakteristik vor und nach dem Entfernen gleich bleibt.

Probleme ergeben sich noch bei Linienenden. Linienendpunkte sind Pixel, die nur einen Nachbarn besitzen, etwa wie Bild 20.22 zeigt.

Diese Bildpunkte werden gesondert behandelt. Es kann etwa gefordert werden, dass

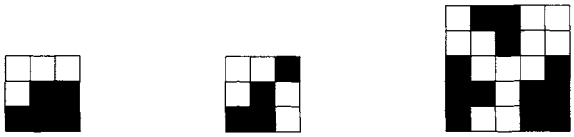


Bild 20.21: Beispiele zur Euler'schen Charakteristik in Binärbildern. (a) e=11, k=15 und f=5, also E=1. (b) e=11, k=14 und f=4, also E=1. (c) e=29, k=38 und f=12, somit E=3. Man sieht, dass E hier die Anzahl der einfach zusammenhängenden Segmente angibt.

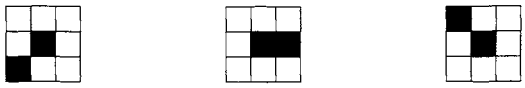


Bild 20.22: Vordergrundbildpunkte, die an Linienenden liegen, müssen gesondert behandelt werden.

Vordergrundpunkte, die nur einen Nachbarn besitzen, nicht entfernt werden, obwohl sich die Euler'sche Charakteristik nicht ändert.

Als nächstes ist noch zu beachten, in welcher Reihenfolge das Bild durchlaufen wird. Zeilenweise von links oben nach rechts unten ist unpassend, da dann z.B. bei einem horizontal liegenden Balken die Skelettlinie am unteren Ende verlaufen würde, was einer der Forderungen widerspricht. Um dies zu umgehen, werden in einem Iterationsschritt mehrere Subiterationen durchgeführt. Bei jeder Subiteration wird nur eine Teilmenge der Bildpunkte des gesamten Bildes untersucht. Vorgeschlagen werden mindestens zwei Subiterationen. In der ersten werden nur diejenigen Pixel verarbeitet, bei denen beide (x, y)-Ortskoordinaten gerade oder beide ungerade sind. In der zweiten Subiteration werden die restlichen Bildpunkte verarbeitet.

Schließlich ist noch zu untersuchen, wann das Verfahren abbricht. In der Regel wird das Abbruchkriterium so formuliert, dass der Algorithmus beendet wird, wenn in einer Iteration keine Vordergrundpixel mehr entfernt wurden. Es müssen also in jeder Iteration alle Pixel untersucht werden, auch wenn vielleicht nur mehr ein oder zwei Pixel entfernt werden. Um hier Rechenzeit zu sparen kann das Bild in rechteckige Teilbilder unterteilt werden. Ein Teilbild wird in einem Iterationsschritt nur dann verarbeitet, wenn noch Bildpunkte entfernt werden können. Teilbilder, in denen das Skelett schon berechnet ist, werden nicht mehr berücksichtigt.

Nun noch einige Bemerkungen zur Implementierung: Die Entscheidung, ob ein Vordergrundbildpunkt entfernt werden darf oder nicht, erfordert ein zweimaliges Berechnen der Euler'schen Charakteristik. Um diesen Rechenaufwand zu vermeiden, kann man eine Tabelle aufstellen, in der vermerkt ist, für welche Konstellationen von $3 \cdot 3$–Umgebungen

der zentrale Punkt entfernt werden kann. Bei $3 \cdot 3$–Umgebungen gibt es 512 Konstellationsmöglichkeiten. Zu jeder Konstellation wird eine Adresse berechnet, die zwischen 0 und 511 liegt und somit einem Eintrag in der Tabelle zugeordnet ist.

Der gesamte Algorithmus läuft wie folgt ab:

A20.5: Skelettierung mit der Euler'schen Charakteristik.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein einkanaliges Grauwertbild mit $G = \{0,1,...,255\}$ als Grauwertmenge (Eingabebild).

◇ *threshold*: Schwellwert zur Binarisierung.

◇ *ETab* sei die Tabelle mit den Einträgen, ob ein Vordergrundpixel entfernt werden darf oder nicht.

◇ Es wird mit zwei Bildpuffern gearbeitet. Am Anfang wird der erste Puffer mit dem Eingabebild vorbesetzt. Das Ergebnis einer Iteration wird im zweiten Puffer erzeugt. Nach einer Iteration wird der Ausgabepuffer zum Eingabepuffer.

Algorithmus:

(a) Setze $removed = 1$;

(b) Solange gilt $removed = 1$:

(ba) Setze $removed = 0$.

(bb) Erste Subiteration: Betrachte alle Pixel in den Positionen (x,y), bei denen x und y beide gerade oder ungerade sind:

(bba) Betrachte für alle Vordergrundpixel die $3 \cdot 3$–Umgebung und berechne die Adresse *position* in der Tabelle *Etab*.

(bbb) Falls gilt $Etab[position] = 1$ und falls der Bildpunkt in der Position (x,y) mehr als einen Nachbarn besitzt, darf er im Ausgabebild dem Hintergrund zugewiesen werden. In diesem Fall wird $removed = 1$ gesetzt.

(bc) Zweite Subiteration: Setze das Ausgabebild zum Eingabebild und betrachte Positionen (x,y), für die x gerade und y ungerade oder x ungerade und y gerade ist:

(bca) (wie (bba)).

(bcb) (wie (bbb)).

(bcc) Vertausche die Puffer.

Ende des Algorithmus

Es sei nochmals darauf hingewiesen, dass aufgrund der etwas vagen Festlegung der Skelettlinie die verschiedenen Verfahren alle im Detail unterschiedliche Ergebnisse bringen. Auch das Verhalten an ausgefransten Segmenträndern ist nicht immer vorhersagbar.

Die weitere Verarbeitung von skelettierten Bildern hängt vom jeweiligen Anwendungsfall ab. Es können z.B. in einem nächsten Schritt die Skelettlinien vektorisiert werden. Das bedeutet, dass die Skelettlinien nicht mehr als Pixelfolgen, sondern durch Polygonzüge dargestellt werden. Dabei werden in der Regel die Skelettlinien „gesäubert", d.h., dass ähnlich verlaufende Linienrichtungen zu einer Richtung zusammengefasst werden.

Ein weiterer Verarbeitungsschritt könnte die Transformation der Vektordarstellung in das Format von gängigen CAD-Systemen sein. Wenn das gelingt, kann eine ursprünglich als Rasterbild vorliegende Information letztlich in einem CAD-System weiter verarbeitet werden. Diese Anwendung kann hilfreich sein, wenn Bilder von (Bau-)Teilen, die mit einem Videosensor erfasst wurden, letztlich in einem CAD-System weiterverarbeitet werden sollen.

20.10 Relaxation

Die mathematisch statistische Methode der Relaxation ist ein interessantes Verfahren, das zur Kanten- und Linienextraktion verwendet wird. Zunächst wird der theoretische Hintergrund der Relaxation dargestellt, dann folgt eine praktische Anwendung aus dem Bereich der Kantenextraktion.

Bei der Relaxation wird folgender Sachverhalt vorausgesetzt:

(20.13)

$S = \{\mathbf{g}_1, \mathbf{g}_2, ..., \mathbf{g}_M\}$	sei eine Menge von M Objekten (z.B. eine Menge von Bildpunkten);
$L = \{l_1, l_2, ..., l_N\}$	sei eine Menge von N Eigenschaften, die die Objekte der Menge S besitzen können;
$p_i(l)$	sei die Wahrscheinlichkeit, dass l eine richtige Eigenschaft von $\mathbf{g}_i$ ist;
L_i	sei die Menge aller Eigenschaften, die für das Objekt $\mathbf{g}_i$ zulässig sind.

Es soll gelten:

$$\sum_{l \in L_i} p_i(l) = 1, \text{für } i = 1, 2, ..., M. \tag{20.14}$$

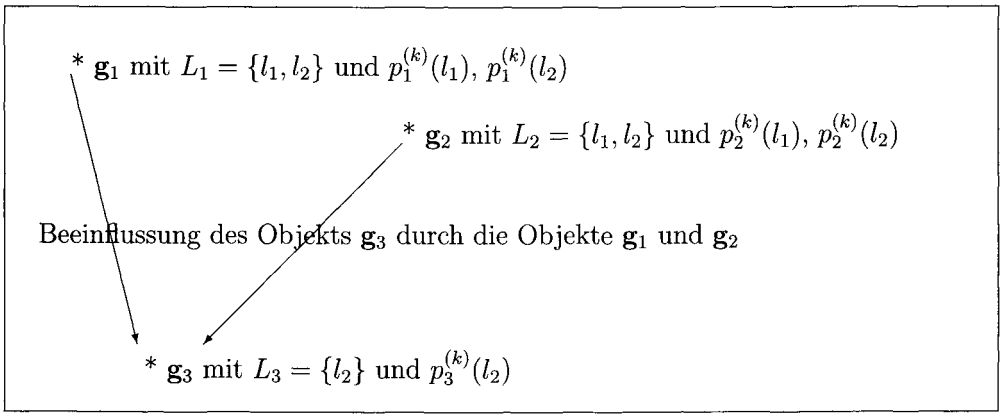


Bild 20.23: Relaxationsmodell mit $S = \{\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3\}$ und $L = \{l_1, l_2\}$ im Iterationsschritt k

Das bedeutet, dass die Summe der Wahrscheinlichkeiten der Eigenschaften für das Objekt $\mathbf{g}_i$ eins ist. Die Wahrscheinlichkeiten werden jetzt als Startwerte einer Iteration betrachtet: $p_i^{(1)}(l) = p(l)$. Im k-ten Iterationsschritt ist

$$p_i^{(k)}(l), i = 1, 2, ..., M \tag{20.15}$$

die Wahrscheinlichkeit, dass die Eigenschaft l auf das Objekt $\mathbf{g}_i$ zutrifft. Die Objekte sollen sich nun gegenseitig beeinflussen, so dass beim Übergang von der k-ten zur $(k+1)$-ten Iteration die Wahrscheinlichkeiten

$$p_i^{(k+1)}(l), i = 1, 2, ..., M \tag{20.16}$$

neu berechnet werden. Bild 20.23 zeigt diesen Sachverhalt an einem Beispiel.

Die Wahrscheinlichkeiten $p_i^{(k+1)}(l)$ hängen dabei von den Wahrscheinlichkeiten $p_i^{(k)}(l)$ des Objektes $\mathbf{g}_i$ und von den Wahrscheinlichkeiten $p_n^{(k)}(l')$ aller benachbarten Objekte ab. Dabei muss in jedem Anwendungsfall festgelegt werden, was unter „benachbart" zu verstehen ist, ob z.B. als Nachbarn alle oder nur bestimmte Objekte zulässig sind. Die Einflussnahme „hängen ab von" wird wie folgt modelliert:

- Falls die Eigenschaften l und l' zusammenpassen (*vereinbar sind*), leistet $p^{(k)}(l')$ einen positiven Beitrag zu $p^{(k+1)}(l)$.
- Falls die Eigenschaften l und l' nicht zusammenpassen (*nicht vereinbar sind*), leistet $p^{(k)}(l')$ einen negativen Beitrag zu $p^{(k+1)}(l)$.
- Falls die Eigenschaften l und l' in keiner gegenseitigen Beziehung stehen (*gegenseitig neutral sind*), leistet $p^{(k)}(l')$ keinen Beitrag zu $p^{(k+1)}(l)$.

Dies kann durch eine Kompatibilitätsfunktion ausgedrückt werden:

$$r_{ij} : L_i \times L_j \rightarrow [-1, 1]\,. \tag{20.17}$$

Zur Neuberechnung der $p^{(k+1)}(l)$ wird zunächst ein Korrekturwert ermittelt:

$$q_i^{(k)}(l) = \sum_j d_{ij} \Big(\sum_{l'} r_{ij}(l, l') p_j^{(k)}(l') \Big). \tag{20.18}$$

Die erste Summe wird dabei über alle Objekte erstreckt, die das Objekt $\mathbf{g}_i$ beeinflussen können und die zweite Summe über alle Marken l' in der Menge L_j. Die Faktoren d_{ij} sind zusätzliche Gewichtsfaktoren, die den Abstand des Objektes $\mathbf{g}_j$ von $\mathbf{g}_i$ beschreiben und für die

$$\sum_j d_{ij} = 1 \tag{20.19}$$

vorausgesetzt wird. Damit kann $p^{(k)}(l)$ korrigiert werden:

$$p_i^{(k+1)}(l) = \frac{p_i^{(k)}(l) \left[1 + q_i^{(k)}(l)\right]}{\sum_{l \in L_i} \left(p^{(k)}(l) \left[1 + q_i^{(k)}(l)\right] \right)}. \tag{20.20}$$

Der Nenner in (20.20) gewährleistet, dass die Summe der neu berechneten Wahrscheinlichkeiten für das Objekt $\mathbf{g}_i$ wieder 1 ergibt.

Nach der Darstellung der Theorie jetzt ein praktisches Beispiel, bei dem die Relaxation zur Kantenverstärkung eingesetzt wird. Die Objekte der Relaxation sind die Bildpunkte $\mathbf{S} = (s(x, y))$ eines Grauwertbildes. Zu jedem Bildpunkt (x, y) wird (mit Hilfe eines lokalen Differenzenoperators, Abschnitt 18.4) der Betrag $mag(x, y)$ des Gradienten berechnet.

Die Menge der Eigenschaften sei $L = \{E, N\}$ mit der Bedeutung:

- E: der Bildpunkt liegt auf einer Kante und
- N: der Bildpunkt liegt auf keiner Kante.

Damit ist klar, dass für jeden Bildpunkt beide Eigenschaften möglich sind, also $L_{(x,y)} = L$ zu setzen ist.

Als Anfangswerte der Iteration werden verwendet:

$$p_{(x,y)}^{(1)}(E) = \frac{mag(x, y)}{\max\{mag(x, y)\}} \text{und} \tag{20.21}$$

$$p_{(x,y)}^{(1)}(N) = 1 - p_{(x,y)}^{(1)}(E). \tag{20.22}$$

Die Bestimmung des maximalen Betrags des Gradienten wird dabei in einer bestimmten Umgebung von (x, y), etwa in der 4- oder 8-Nachbarschaft oder auch in einer größeren Umgebung durchgeführt. Damit ist auch die mögliche Einflussnahme der Objekte festgelegt,

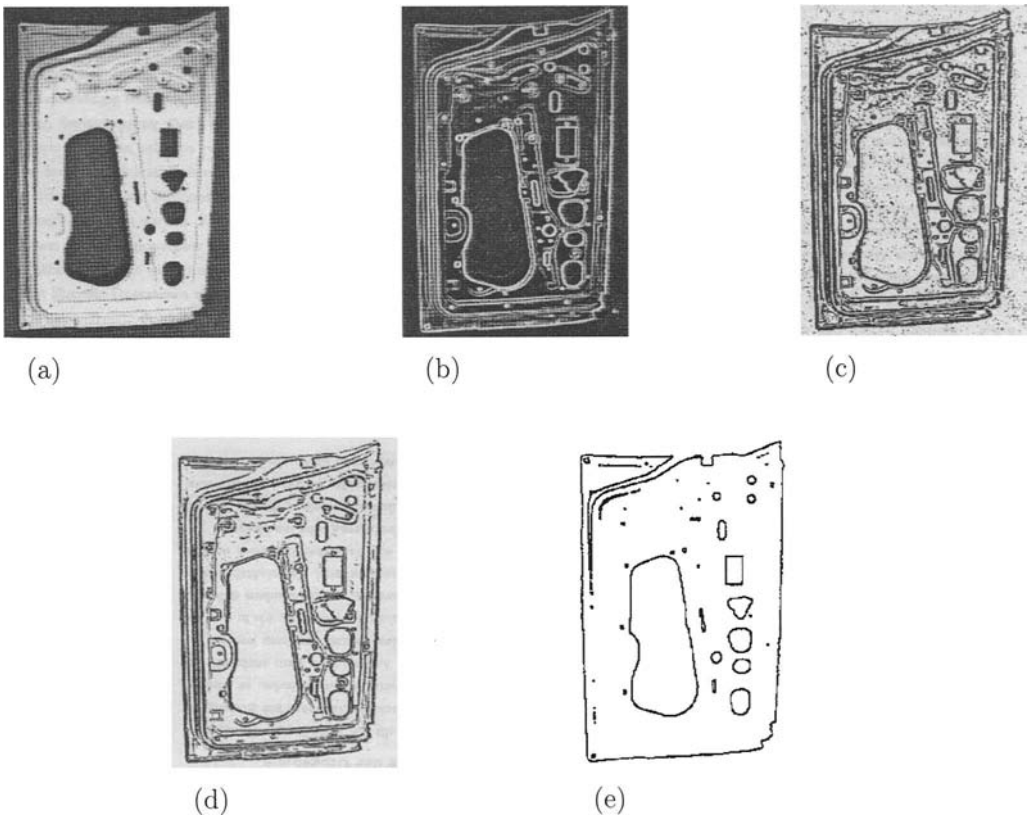


Bild 20.24: (a) Originalbild: Blechteil. (b) Betrag des Sobeloperators. (c) Invertierung von (b). (d) Binarisierung des invertierten Sobeloperatorbetrags. Dieses Bild ist das Ausgangsbild für die Relaxation. (e) Ergebnis der Relaxation.

deren Gewichtung z.B. für alle gleich sein kann. Schließlich muss noch die Kompatibilitätsfunktion angegeben werden. Dies geschieht hier als Kompatibilitätsmatrix:

$$\begin{pmatrix} r_{EE} & r_{EN} \\ r_{NE} & r_{NN} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{20.23}$$

Damit sind alle Parameter der Relaxation festgelegt und die Iterationen können gemäß (20.18) und (20.20) berechnet werden. Nach einer festgelegen Anzahl von Iterationen werden z.B. die Grauwerte des Originals mit den berechneten Wahrscheinlichkeiten multipliziert und so die Grauwertkanten verstärkt.

Die Bildfolge 20.24 zeigt ein Beispiel dazu. Ausgangspunkt der Relaxation war hier ein Sobelbetragsbild des Originals.

20.11 Grundlagen: Houghtransformation

Zielsetzung der *Houghtransformation* (gesprochen: „Haff"-Transformation) ist das Finden vorgegebener geometrischer Strukturen (*Referenzstrukturen*) in einem (segmentierten) Bild. Es wird geprüft, ob einzelne Segmente der Referenzstruktur ähnlich sind. Als geometrische Strukturen kommen z.B. Geraden, Kreise, Ellipsen, aber auch beliebige andere Formen in Frage. Eine wesentliche Eigenschaft der Houghtransformation ist ihre Robustheit. Damit ist gemeint, dass die im Bild auftretenden Strukturen der Referenzstruktur nicht gleich sein müssen, sondern durch Rauschen oder systematische Fehler gestört sein können. Die Houghtransformation ist auch erfolgreich einsetzbar, wenn die zu detektierenden Strukturen teilweise verdeckt, also nicht vollständig sind. Man kann daher von einer Rekonstruktion geometrischer Strukturen nach Maßgabe einer Referenzstruktur sprechen.

Die Houghtransformation wurde ursprünglich zur *Detektion kollinearer Bildpunkte* in Binärbildern verwendet. Die geometrische Struktur, die als Referenzstruktur dient, ist in diesem Fall die Gerade. Eine Gerade ist durch die Gleichung $ax + by + c = 0$ eindeutig bestimmt. Für $b \neq 0$ ist sie durch die beiden Parameter $m = -\frac{a}{b}$ (die Steigung) und $t = -\frac{c}{b}$ (den $y-$Achsenabschnitt) ebenfalls eindeutig bestimmt. Auch andere Gleichungen legen eine Gerade fest: Wird z.B. vom Koordinatenursprung das Lot auf die Gerade gefällt, so ist durch die Länge r des Lots und den Winkel φ des Lots mit der x- oder y-Achse die Gerade ebenfalls eindeutig festgelegt (Bild 20.25). Den Zusammenhang zwischen (x, y)-Koordinaten und (r, φ)-Koordinaten beschreibt die *Hesse'sche Normalform*:

$$r = x \cdot \cos\varphi + y \cdot \sin\varphi. \tag{20.24}$$

In einem Koordinatensystem, in dem auf der Abszisse die Werte von r und auf der Ordinate die Werte von φ aufgetragen werden, entspricht der Geraden ein Punkt. Dieses Koordinatensystem wird als (r, φ)-*Raum* bezeichnet.

Trägt man für alle Geraden des Geradenbüschels in einem Punkt A die Werte in den (r, φ)-Raum ein, so erhält man eine sinoidale Kurve (Bild 20.26). Das Büschel in einem zweiten Punkt B erzeugt ebenfalls eine derartige Kurve. Da die Büschel eine Gerade gemeinsam haben, schneiden sich beide Kurven im (r, φ)-Raum in einem Punkt. Wenn man somit die Büschel von vielen Punkten, die alle auf einer Geraden liegen, in den (r, φ)-Raum einträgt, so werden sich die dazugehörigen Kurven alle in einem Punkt schneiden.

Dieser Sachverhalt kann zur Detektion kollinearer Bildpunkte in einem Bild oder Bildausschnitt verwendet werden. Dazu wird der (r, φ)-Raum diskretisiert, z.B. die φ-Achse in 1°- oder 5°-Stufen und die Radiusachse je nach aktuellem Genauigkeitsbedarf. Der (r, φ)-Raum geht dann in ein endliches, zweidimensionales Feld über, das im Folgenden auch als *Akkumulator* bezeichnet wird. Jedes Element des Akkumulators entspricht genau einer Geraden im (x, y)-Koordinatensystem. Alle Elemente des Akkumulators werden anfänglich mit null vorbesetzt.

Im nächsten Schritt werden zu allen Vordergrundbildpunkten des binären Eingabebildes die Geradenbüschel berechnet und in den (r, φ)-Raum eingetragen, d.h. es werden die betroffenen Elemente des Akkumulators um eins erhöht. Falls nun im Eingabebild viele

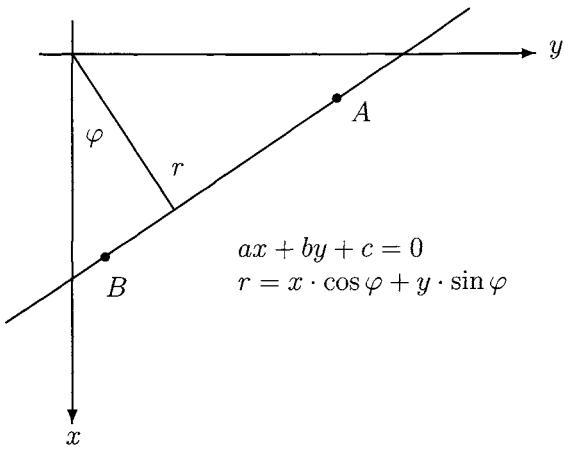


Bild 20.25: Gerade im (x, y)-Koordinatensystem. Wird vom Koordinatenursprung das Lot auf die Gerade gefällt, so legen die Länge r und der Winkel φ die Gerade eindeutig fest. Den Zusammenhang zwischen (x, y)-Koordinaten und (r, φ)-Koordinaten beschreibt die Hesse'sche Normalform.

Vordergrundbildpunkte näherungsweise auf derselben Geraden liegen (Bild 20.27-b), so wird im Akkumulator dasjenige Element einen hohen Wert aufweisen, in dem sich alle Büschelkurven schneiden (Bild 20.27-c). Die Aufgabe, im Originalbild kollineare Bildpunkte zu detektieren, reduziert sich nach dieser Verarbeitung auf die Suche des maximalen Elements des Akkumulators. Die zu diesem Element gehörigen r- und φ-Werte bestimmen die Gerade im (x, y)-Koordinatensystem, auf der die Vordergrundbildpunkte näherungsweise liegen.

Der Verlauf der Geraden kann dabei natürlich nur im Rahmen der gewählten Diskretisierung des (r, φ)-Koordinatensystems angegeben werden. Die Bilder 20.28-a bis 20.28-c zeigen weitere Beispiele und Problematiken zur Houghtransformation.

Die Auswertung des (r, φ)-Raumes kann eine reine Maximumsuche sein oder ein Clusteralgorithmus, mit dem dann auch versucht werden kann, bestimmte Bildstrukturen im Original zu erkennen. Als Beispiel dazu dient Bild 20.29. Es zeigt in Bild 20.29-a ein Quadrat und in Bild 20.29-b den (r, φ)-Raum, in dem sich vier Punkthäufungen ergeben, die den Geraden des Quadrates entsprechen.

Der Vorteil der Houghtransformation liegt darin, dass die Linien auch kleinere Unterbrechungen aufweisen dürfen und dass auch andere, verstreute Punkte auftreten können. Die Wahl des Bildausschnittes, auf den jeweils die Houghtransformation angewendet wird, ist von der Linienbreite abhängig und muss in konkreten Anwendungsfällen sorgfältig abgestimmt werden.

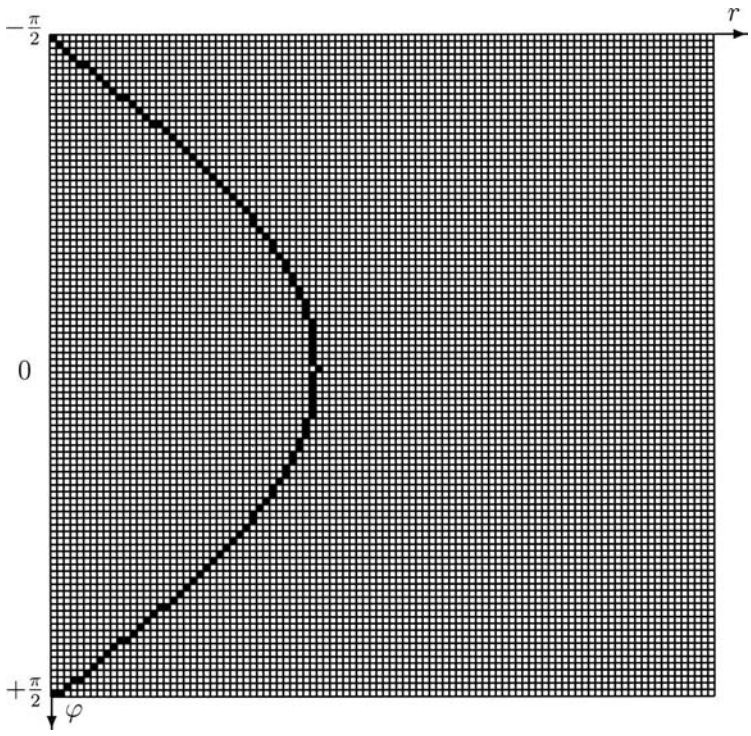


Bild 20.26: In einem (r, φ)-Koordinatensystem ((r, φ)- Raum) entspricht einer Geraden im (x, y)-Koordinatensystem ein Punkt. Die Geraden eines Büschels ergeben sinoidale Kurven.

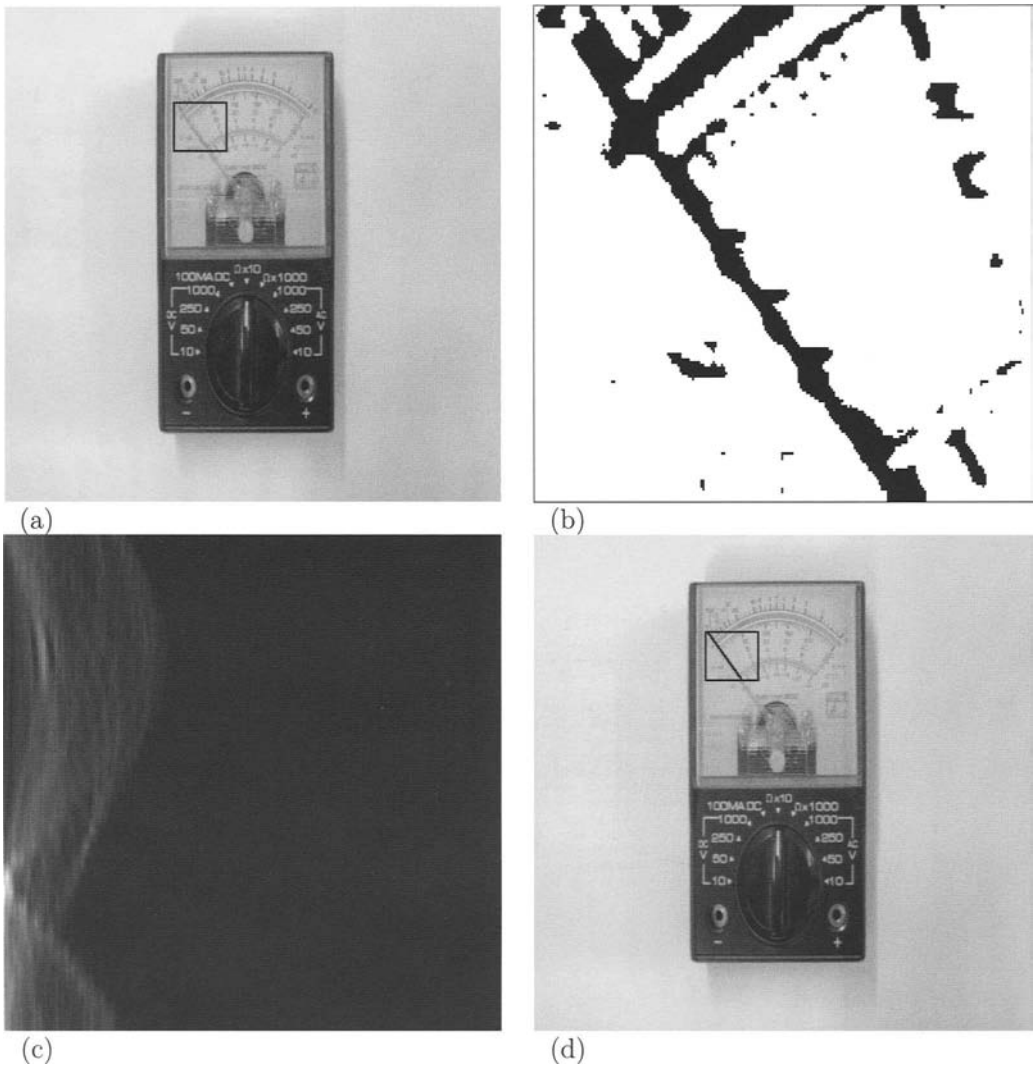


Bild 20.27: (a) Originalbild. Im markierten Ausschnitt soll die Stellung des Zeigers detektiert werden. (b) Markierter Ausschnitt, vergrößert und binarisiert. (c) Houghraum. Die Geradenbüschel aller schwarzen Vordergrundbildpunkte von Teilbild (b) wurden in den (r, φ)-Raum eingetragen. Da sich alle Büschelkurven von kollinearen Bildpunkten im (r, φ)-Raum in einem Punkt schneiden, haben nach dieser Auswertung einige Elemente des Akkumulators hohe Werte, die im Bild durch helle Grautöne dargestellt sind. Die Aufgabe der Detektion des Zeigers besteht jetzt nur mehr aus der Suche des maximalen Wertes des Akkumulators. (d) Das Maximum des Akkumulators liegt bei $r = 2Pixel$ und $\varphi = 124°$. Die detektierte Gerade ist in den Ausschnitt eingezeichnet. Das Lot von der linken oberen Ecke des Bildausschnitts auf diese Gerade bildet mit der nach unten gehenden x-Achse einen Winkel von 124°. Der Betrag ist etwa zwei Pixel.

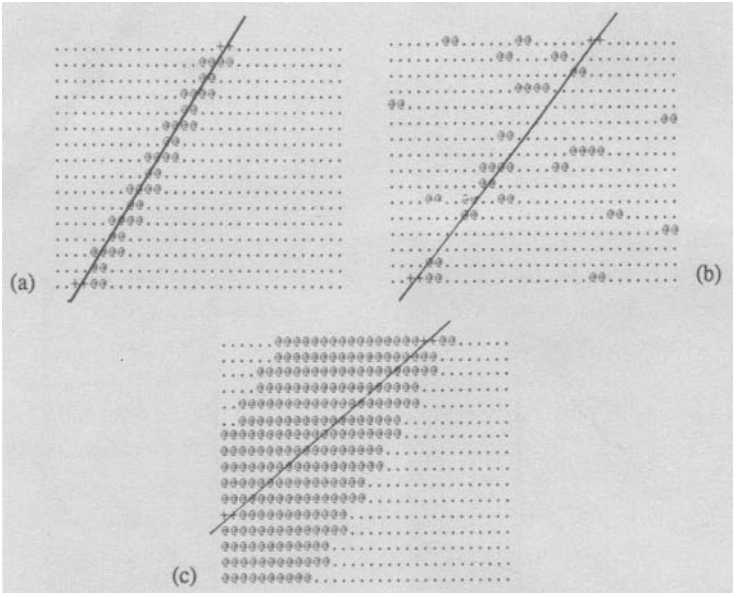


Bild 20.28: Beispiele zur Houghtransformation. (a) Detektion eines Geradenstücks (die Schnittpunkte der erkannten Gerade mit dem Bildrand sind durch '++' gekennzeichnet). (b) Detektion eines unterbrochenen Geradenstücks mit zusätzlichen Störungen. (c) Falls die zu detektierende Linie zu breit im Verhältnis zum Bildausschnitt ist, ergeben sich Ungenauigkeiten.

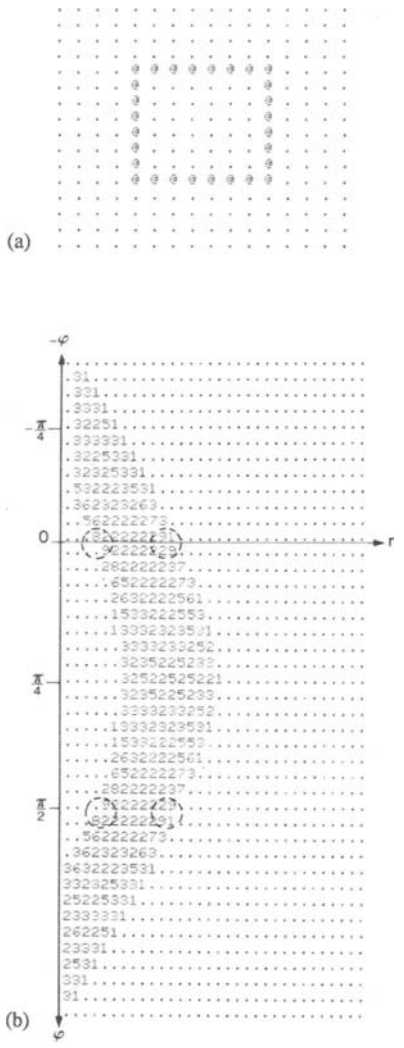


Bild 20.29: Beispiele zur Houghtransformation. (a) Beispiel eines Quadrates im Bildausschnitt. (b) Im (r, φ)-Raum ergeben sich vier Punkthäufungen.

20.12 Verallgemeinerte Houghtransformation

Die Ausführungen von Abschnitt 20.11 lassen allgemeinere Konzepte erkennen, die im Folgenden näher untersucht werden. Es besteht die Aufgabe, in einem Merkmalskanal eines (vorverarbeiteten) Bildes Bildstrukturen nach Maßgabe einer Referenzstruktur zu detektieren. Dabei können die Bildstrukturen bis zu einer bestimmten Grenze verrauscht, gestört oder verdeckt sein. Als geometrische Strukturen kommen z.B. Geraden, Kreise oder Ellipsen in Frage. In Abschnitt 20.13 wird eine Erweiterung auf beliebige Formen vorgestellt. Da bei Geraden, Kreisen oder Ellipsen nur eindimensionale Konturen untersucht werden, ist als Vorverarbeitung eine Kanten- oder Liniendetektion sinnvoll. Man könnte z.B. zu einem Grauwertbild die Gradientenbeträge (Abschnitt 18.4) berechnen und nur die Bildpunkte mit hohem Betrag in die weitere Verarbeitung einbeziehen.

Mit der Houghtransfomation werden im nächsten Schritt solche Bildpunkte gesucht, die ähnliche Merkmalseigenschaften besitzen. Es werden also lokale Merkmalsausprägungen zu globalen Ausprägungen kombiniert.

Die Darstellung der Standard-Houghtransformation von Abschnitt 20.11 zeigt drei Schritte, die der Reihe nach behandelt werden müssen:

- Die Parametrisierung der Referenzstruktur.
- Die akkumulierende Abbildung der Merkmalspunkte.
- Die Auswertung des Akkumulators.

20.12.1 Parametrisierung der Referenzstruktur

Für die Referenzstruktur muss eine geeignete Parametrisierung gefunden werden. Ein Beispiel für die Gerade wurde in Abschnitt 20.11 schon gegeben. Wie bereits erwähnt, können aber auch die Steigung und der y-Achsenabschnitt als Parameterpaar verwendet werden. Ein weiteres Beispiel sind zwei Werte s_1, s_2, die die Schnittpunkte der Geraden mit dem Bildrand angeben. Die Werte für s_1 und s_2 werden in einer festgelegten Orientierung entlang des Bildrandes gemessen (Bild 20.30).

Analog hierzu ergibt sich, dass für den Kreis drei Parameter (z.B. (x, y)-Koordinaten des Mittelpunkts und Radius) und für die Ellipse fünf Parameter (z.B. (x, y)-Koordinaten des Mittelpunkts, Längen der beiden Halbachsen und Drehwinkel) zur Definition notwendig sind. Eine geometrische Referenzstruktur wird somit durch ein *Parametertupel* beschrieben, das einen *Parameterraum* aufspannt. Die Anzahl der Parameter bestimmt die Dimension des Parameterraums, der auch *Transformationsraum* oder *Houghraum* genannt wird. Ein Parametertupel $(p_1, p_2, ..., p_N)$ entspricht genau einer Ausprägung der Referenzstruktur in einer definierten Lage.

Da der Parameterraum diskretisiert werden muss, sind nur solche Parametrisierungen von Bedeutung, bei denen sich für alle in Frage kommenden Lagen der Referenzstruktur endliche Werte ergeben. Werden Geraden als Referenzstruktur verwendet, so erfüllen die Parameterpaare (r, φ) oder (s_1, s_2) diese Forderung.

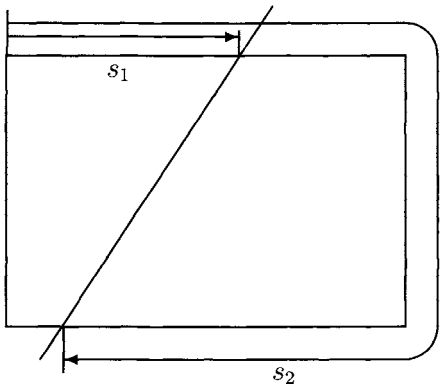


Bild 20.30: Parametrisierung einer Geraden durch die Angabe ihrer Schnittpunkte mit dem Bildrand.

Zur Diskretisierung werden die einzelnen Parameterskalen abgeschätzt und in (meistens äquidistante) Intervalle eingeteilt. Bei dieser Einteilung können Genauigkeitsanforderungen berücksichtigt werden. Durch die Diskretisierung des Houghraums wird das Akkumulatorfeld definiert. Jedem Element des Akkumulators entspricht ein Parametertupel und damit eine Referenzstruktur. Der Akkumulator wird anfangs auf null gesetzt. Bei einigen Varianten sind die Elemente des Akkumulators nicht Skalare, sondern Listen. In diesem Fall wird das Element in den Zustand „leere Liste“ gesetzt.

20.12.2 Akkumulierende Abbildung der Merkmalspunkte

In dieser Phase werden für jeden Merkmalspunkt (Vordergrundpunkt) die Lageparameter aller möglichen durch die Referenzstruktur festgelegten Strukturen berechnet, deren Mitglied der Punkt sein kann. Im Fall einer Geraden als Referenzstruktur ist das das Geradenbüschel im aktuellen Punkt. Für jedes sich ergebende Parametertupel wird die entsprechende Zelle des Akkumulators inkrementiert.

Die Inkrementierung kann im einfachsten Fall eine Erhöhung um eins sein. Aber auch kompliziertere Inkrementfunktionen sind möglich. Werden z.B. diejenigen Bildpunkte als Vordergrundpunkte betrachtet, deren Gradientenbetrag größer als ein vorgegebener Wert ist, so kann die Inkrementierung auch um diesen Betrag erfolgen.

20.12.3 Auswertung des Akkumulators

Der Auswertung des Akkumulatorfeldes liegt folgende Überlegung zugrunde: Strukturen, die der Referenzstruktur entsprechen und durch viele Merkmalspunkte gestützt sind, werden im Akkumulator einen hohen Wert aufweisen. Wenn man den maximalen Wert des

Akkumulators bestimmt, so kann man ziemlich sicher sein, dass das zugehörige Parametertupel die Lage einer Bildstruktur beschreibt, die der Referenzstruktur sehr ähnlich ist.

Bei der Auswertung des Akkumulators werden somit das globale Maximum oder lokale Maxima gesucht. Je mehr eine Struktur gestört, verrauscht oder verdeckt ist, je unähnlicher sie somit der Referenzstruktur ist, desto weniger markant werden sich die Maxima ausprägen. Es ist offensichtlich, dass das Problem der Auswertung des Akkumulators ein *cluster*-Analyseproblem ist und daher mit den einschlägigen Verfahren (z.B. mit dem *fuzzy*-Klassifikator von Kapitel 33) behandelt werden kann.

Nachdem eine Struktur auf diese Weise gefunden ist, können aus dem zugeordneten Parametertupel weitere Eigenschaften, wie z.B. die Größe, die Orientierung, die Länge der Kontur, usw. berechnet werden.

Im folgenden Abschnitt wird eine Erweiterung der Houghtransformation vorgestellt, die es erlaubt, beliebige Segmentstrukturen als Referenz zu verwenden. Dabei kann neben der konturbezogenen Vorgehensweise auch eine flächenhafte Analyse durchgeführt werden.

20.13 Erweiterte Houghtransformation

Die Houghtransformation wurde ursprünglich, wie oben dargestellt, zur Detektion von Geraden und Kurven in Bildausschnitten definiert. In diesem Abschnitt wird die Houghtransformation verallgemeinert und für Segmente verwendet, die durch ihre Konturkurve in einem binarisierten Kantenbild gegeben sind.

20.13.1 Erweiterte Houghtransformation der Randpunkte

Es wird ein Bild vorausgesetzt, in dem unterschiedliche Segmente enthalten sind. Es besteht die Zielsetzung, zu entscheiden, ob ein bestimmtes Segment mit einer für dieses Segment charakteristischen Form im Bild enthalten ist.

Als Vorverarbeitungsschritte sei eine Segmentierung (etwa eine Binarisierung, Abschnitt 17.5) und eine Randerkennung (Abschnitte 20.3 bis 20.11) der einzelnen Segmente durchgeführt worden. Die Randerkennung kann z.B. mit Hilfe einer *run-length*-Codierung, einer Segmentvereinzelung und einer Markierung der Randpunkte der einzelnen Segmente durchgeführt werden (Kapitel 34). Das gesamte Bild kann durch Rauschen gestört sein. Auch eine teilweise Verdeckung der Segmente ist zulässig.

Zu jedem Segment wird ein *Bezugspunkt* festgelegt. Als Bezugspunkt kann ein beliebiger Punkt verwendet werden. Es bietet sich z.B. der Flächenschwerpunkt an. Bild 20.31 zeigt ein Beispiel, wie ein derartiges Eingabebild aussehen könnte. Die Bezugspunkte der Segmente sind schraffiert markiert.

Die Aufgabe besteht darin zu entscheiden, ob ein vorgegebenes Segment, das in Form einer bestimmten Datenstruktur vorliegt, im Bild enthalten ist oder nicht. Dieses Segment wird im Folgenden als *Referenzsegment* und die dazugehörige Kontur als *Referenzkontur* bezeichnet. Vorausgesetzt wird, dass das zu detektierende Segment in gleicher Größe und Drehlage im Eingabebild enthalten sein muss.

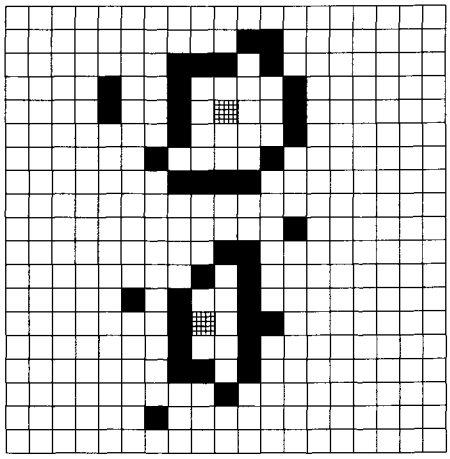


Bild 20.31: Beispiel zu einem vorverarbeiteten Bild, in dem zwei Segmente durch ihren Rand darstellt sind. Die Bezugspunkte der Segmente sind schraffiert markiert. Außerdem enthält das Bild einzelne Störungen.

Zu Beginn der Verarbeitung werden zunächst die Elemente eines zweidimensionalen *Akkumulatorfeldes* $\mathbf{A} = (a(x, y))$, das die Größe des Eingabebildes besitzt, auf null gesetzt. Es wird ein beliebiger Vordergrundpunkt P des binären Eingabebildes $\mathbf{S}_e$ betrachtet. Dieser Punkt könnte auch ein Punkt der Referenzkontur sein. Falls es ein solcher Punkt ist, müsste der Bezugspunkt des Referenzsegments in einer bestimmten, bekannten relativen Lage dazu liegen. Es werden der Reihe nach alle Möglichkeiten untersucht, wie die Referenzkontur relativ zu diesem Punkt liegen könnte. Jedesmal ergibt sich eine andere Lage für den Bezugspunkt. Diese Menge von möglichen Lagen des Bezugspunktes wird *Lösungsmenge für den Bezugspunkt* genannt. Bild 20.32 zeigt die Lösungsmenge für den markierten Punkt P. Das Referenzsegment ist darüber abgebildet. Man sieht, dass die Lösungsmenge punktsymmetrisch zur Referenzkontur liegt.

Im nächsten Schritt werden die Zähler im Akkumulatorfeld für die Punkte der Lösungsmenge um eins erhöht. Die geschilderten Schritte werden für alle in Frage kommenden Vordergrundpunkte des Eingabebildes durchgeführt. Abschließend wird im Akkumulatorfeld das Maximum ermittelt. Die Koordinaten des Maximums werden festgehalten. Falls der Wert des Maximums einen bestimmten Schwellwert übersteigt, ist das Segment gefunden und der Bezugspunkt liegt an der Stelle der Maximumkoordinaten. Je nach Wahl des Schwellwertes können im Eingabebild $\mathbf{S}_e$ mehr oder weniger große Abweichungen von der Form des zu detektierenden Referenzsegments auftreten.

Zusammengefasst lautet der Algorithmus für die erweiterte Houghtransformation wie folgt:

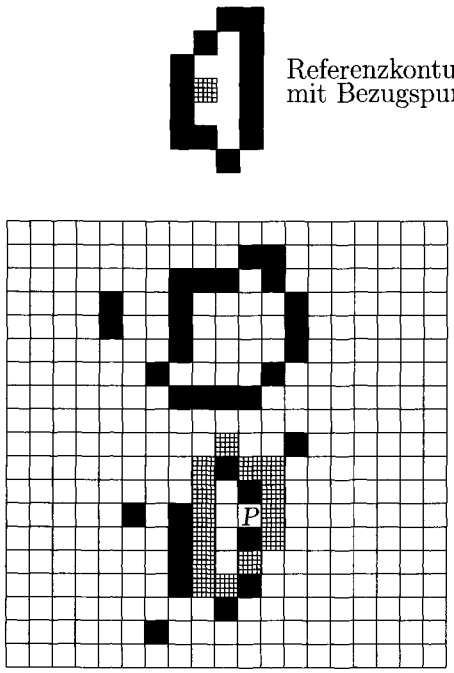


Bild 20.32: Wenn der Punkt P ein Punkt der Referenzkontur ist, müsste der Referenzpunkt in einer bestimmten relativen Position dazu liegen. Es wird der Reihe nach angenommen, dass der Punkt P ein Punkt der Referenzkontur ist. Jedesmal ergibt sich dann eine andere Position für den Referenzpunkt. Die Lösungsmenge für den markierten Punkt P ist schraffiert. Das zu suchende Segment ist darüber abgebildet. Die Lösungsmenge liegt punktsymmetrisch zur Referenzkontur. Für jeden Vordergrundbildpunkt wird die Lösungsmenge bestimmt und die entsprechenden Elemente im Akkumulator erhöht. Am Ende wird dasjenige Akkumulatorelement einen hohen Wert aufweisen, das die Lage des Referenzpunktes im Eingabebild angibt.

A20.6: Erweiterte Houghtransformation.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y))$ ein Binärbild, in dem nur die Ränder der Segmente und eventuelle Störungen auftreten (z.B. Hintergrundpunkte: Grauwert 0, Vordergrundpunkte: Grauwert 1);

◇ $\mathbf{A} = (a(x,y))$ sei ein Akkumulatorfeld, das anfangs mit null vorbesetzt ist.

◇ Das Referenzsegment mit Bezugspunkt liegt in einer geeigneten Datenstruktur vor.

Algorithmus:

(a) Für alle Vordergrundpunkte $s_e(x,y)$ mit dem Grauwert 1:

(aa) Ermittle die Lösungsmenge für den Bezugspunkt.

(ab) Erhöhe die entsprechenden Zähler $a(x,y)$ im Akkumulatorfeld um eins.

(b) Ermittle das Maximum des Akkumulatorfeldes.

(c) Falls das Maximum einen bestimmten Schwellwert übersteigt, geben die Koordinaten des Maximums die Lage des Bezugspunktes für das gesuchte Segment an.

Ende des Algorithmus

Die Bildfolge 20.33 zeigt ein Beispiel. Bild 20.33-a ist das Originalbild, Bild 20.33-b das vorverarbeitete Binärbild. Bild 20.33-c zeigt die Referenzkontur und Bild 20.33-d das Original mit dem eingeblendeten, detektierten Segment.

Ein großer Nachteil dieses Verfahrens ist, dass es weder größen- noch rotationsinvariant ist. Im Rahmen einer praktischen Anwendung müssten die zu untersuchenden Strukturen so vorverarbeitet werden, dass dieser Nachteil keine Rolle spielt. Ein Möglichkeit dazu wäre es, die Teile bereits ausgerichtet dem Videosensor zuzuführen. Ist das nicht möglich, so könnte man versuchen, eine Skalen- und Rotationsinvarianz durch geometrische Operationen zu erreichen. Auch die Berechnung von Momenten (Abschnitt 35.9) wäre anwendbar.

Wenn das Referenzsegment in diskreten Skalen- und Rotationsstufen vorliegt, wird der obige Algorithmus für jede Größen- und Rotationskombination durchlaufen. Das Verfahren ist dann größen- und rotationsinvariant.

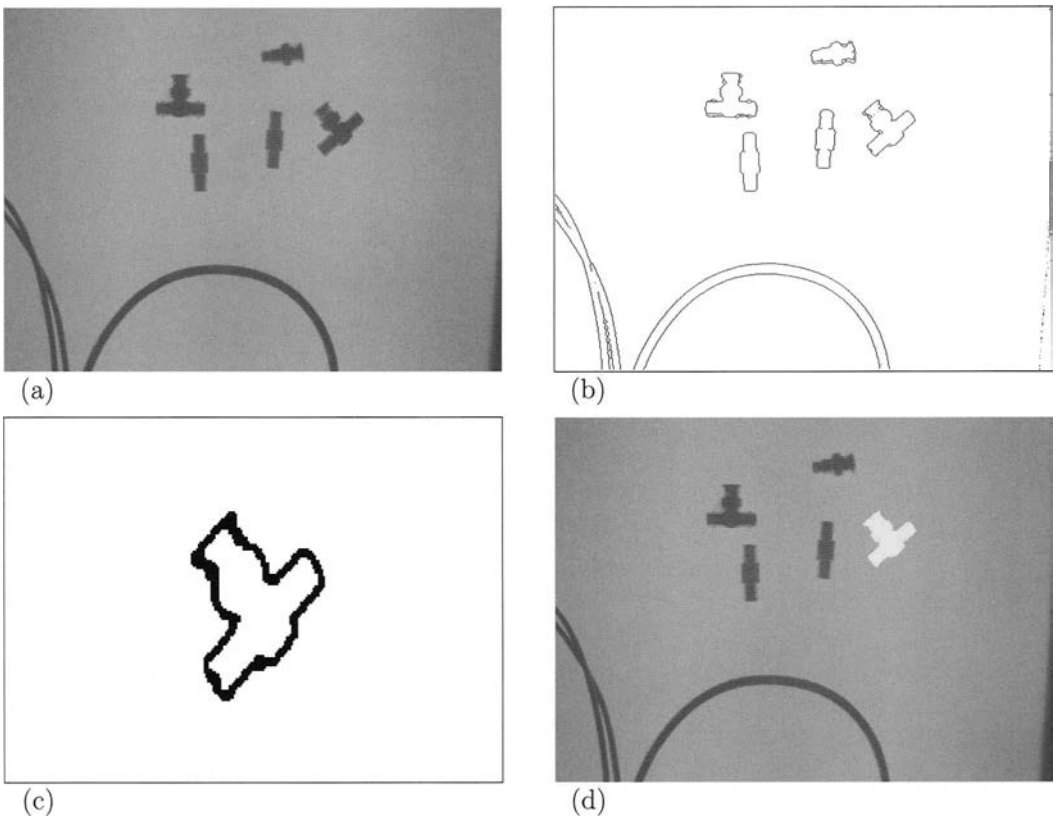


Bild 20.33: (a) Originalbild. (b) Vorverarbeitetes Binärbild. (c) Referenzkontur. (d) Original mit eingeblendetem Segment.

20.13.2 Erweiterung auf flächenhaft gegebene Segmente

Für die Betrachtungen in diesem Abschnitt sei ein Segment flächenhaft durch eine Menge von Punkten (nicht nur Konturpunkten) gegeben, die alle eine bestimmte Eigenschaft besitzen. Beispielsweise kann ein Segment einen bestimmten Grauwert, eine bestimmte Gradientenrichtung oder eine bestimmte Oberflächenstruktur (Textur) besitzen.

In einem Vorverarbeitungsschritt wird für jeden Bildpunkt $s_e(x,y)$ des Eingabebildes $\mathbf{S}_e$ aus einer Umgebung ein (Textur-)Maß $s_t(x,y) = f(U(s_e(x,y))$ berechnet. Dabei wird mit $U(s_e(x,y))$ die Menge der Umgebungsbildpunkte von $s_e(x,y)$ bezeichnet und mit f die Funktion, die zur Berechnung des (Textur-)Maßes auf die Umgebung angewendet wird (Kapitel 27 bis 29).

Das Referenzsegment mit Bezugspunkt besitzt einen bekannten Wert ms_t für das (Textur-)Maß. Für alle Punkte (x,y), für die die Distanz $d(ms_t, s_t(x,y))$ kleiner als ein vorgegebener Schwellwert ist, wird die Lösungsmenge für den Bezugspunkt bestimmt. Die weitere Auswertung erfolgt wie in Abschnitt 20.12.

Da hier im Gegensatz zu Abschnitt 20.12 die Lösungsmenge für wesentlich mehr Bildpunkte bestimmt werden muss, ergeben sich längere Rechenzeiten. Man kann deshalb die Segmente *run-length* codieren (Kapitel 34) und statt des Akkumulatorfeldes $\mathbf{A} = (a(x,y))$ ein Feld $\mathbf{D} = (d(x,y))$ mit

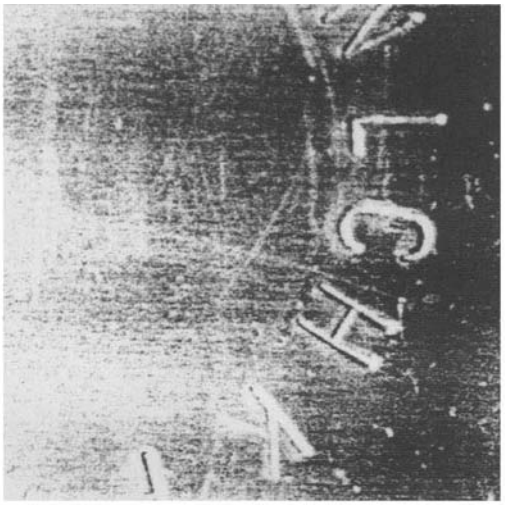
$$d(x,y) = a(x,y) - a(x,y-1) \tag{20.25}$$

zu verwenden. Tritt nämlich eine bestimmte Zeile des *run-length*-Codes in der Lösungsmenge auf, so müssten alle entsprechenden Felder $a(x,y)$ des Akkumuators um eins erhöht werden. Bei der Verwendung von $d(x,y)$ müssen nur zwei Felder verändert werden: Am linken Rand wird die Differenz um eins größer und am rechten Rand um eins kleiner. Für die Felder dazwischen ändert sich die Differenz nicht. Mit dieser Verbesserung erhöht sich die Rechenzeit nur geringfügig.

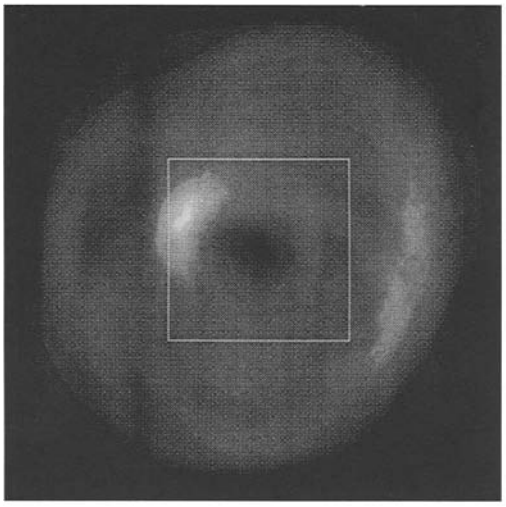
Zur Maximumsuche kann das Akkumulatorfeld $\mathbf{A} = (a(x,y))$ einfach berechnet werden: Für jeden Zeilenanfang wird $a(0,y) = 0$ gesetzt. Die weiteren Zeilenelemente berechnen sich dann wie folgt:

$$a(x,y) = a(x-1,y) + d(x,y). \tag{20.26}$$

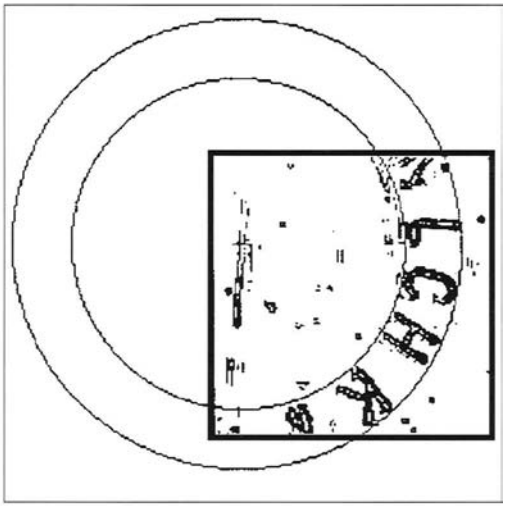
Abschließend zeigt die Bildfolge 20.34 ein Anwendungsbeispiel. Dieses Beispiel und die Bilder wurden [Ever90] entnommen. Zu detektieren sind in Blech eingestanzte Schriftzeichen, die auf einem Kreisbogen angeordnet sind (Bild 20.34-a). Um die Zeichen lokalisieren zu können, ist die Position eines Ringes mit bekanntem Innen- und Außenradius gesucht, in dem die Schriftzeichen liegen. Die Schriftzeichen heben sich durch starke Hell-Dunkel-Übergänge ab. Deshalb wird als Merkmal für die Akkumulation gefordert, dass ein Punkt Vordergrundpunkt im binarisierten Gradientenbild ist. Als Refenenzpunkt für den gesuchten Ring wird sein Mittelpunkt verwendet. Der Akkumulatorwert $a(x,y)$ enthält somit die Anzahl der Vordergrundpunkte, die im Ring liegen, wenn sich sein Mittelpunkt in der Position (x,y) befindet. Bild 20.34-b zeigt das Akkumulationsergebnis (helle Grauwerte entsprechen hohen Werten). Bild 20.34-c zeigt das Gradientenbild mit der detektierten optimalen Lage des Rings.



(a)



(b)



(c)

Bild 20.34: (a) In Blech eingestanzte Schriftzeichen, die innerhalb eines Kreisrings angeordnet sind. Aus [Ever90]. (b) Ergebnis der Akkumulation. (c) Gradientenbild mit detektierter optimaler Lage des Rings.

20.14 Sequentielle Kantenextraktion, Linienverfolgung

Der Grundgedanke bei den sequentiellen Verfahren ist, dass, um eine Grauwertkante oder eine Linie zu extrahieren, nicht alle Bildpunkte des Bildes untersucht werden müssen, sondern nur diejenigen, die Kandidaten sind. Damit werden auch gleich die zwei Teilprobleme der Linienverfolgung deutlich: das *Finden von Ansatzstellen für eine Linienverfolgung* und das *Finden von Nachfolgepunkten*, wenn schon ein bestimmter Abschnitt extrahiert wurde.

Bevor aber diese Probleme näher untersucht werden, noch einige generelle Bemerkungen zur Linienverfolgung. Hier treten eine Reihe von Sachverhalten auf, mit denen alle verwendeten Verfahren konfrontiert werden. Was passiert z.B., wenn die verfolgte Linie unterbrochen ist? Bricht das Verfahren ab oder besteht die Möglichkeit, kleinere Unterbrechungen zu überbrücken? Was geschieht bei Verzweigungen? Werden sie überhaupt erkannt? Können die verschiedenen Äste extrahiert werden? Merkt das Verfahren, wenn es einen Ast ein zweitesmal durchläuft? Was passiert bei Helligkeits- und/oder Kontraständerungen entlang des Linienverlaufs? Schließlich: Besteht die Möglichkeit, auch parallel verlaufende Linien zu erkennen? Bei den folgenden Verfahren kann ihre Eignung anhand dieser Bewertungskriterien gemessen werden.

Jetzt zur ersten Fragestellung, dem Finden von Ansatzstellen für eine Linienverfolgung. Die einfachste und sicherste Möglichkeit ist natürlich die interaktive Festlegung eines Ansatzpunktes für eine Linie. Das klingt zunächst nicht sehr praktikabel, aber es gibt Anwendungsbereiche, wo die interaktive Festlegung durchaus eingesetzt werden kann, z.B. im Rahmen eines Systems, in dem digitalisierte Luft- oder Satellitenbilder teilautomatisch ausgewertet werden. Mit den Methoden der multivariaten Klassifizierung können hier die flächenhaften Objekte (Gewässer, Grünland, Ackerland, usw.) extrahiert werden. Ein Bearbeiter wird dann diese Ergebnisse interaktiv beurteilen und, wenn nötig, korrigieren. Bei dieser Nachbearbeitung ist es dann aber leicht möglich, das verarbeitende System zur Linienverfolgung von Straßen, Gewässerläufen, Eisenbahndämmen usw. jeweils auf geeignete Ansatzstellen zu setzen. In vielen Fällen genügen hier schon wenige Punkte pro Bildausschnitt, wenn der Linienverfolger auch Verzweigungen erkennt und richtig verarbeitet.

Eine andere, automatische Möglichkeit ist die Bewertung eines in Frage kommenden Bildpunktes durch Klassifizierung. Dazu kann so vorgegangen werden, wie in Abschnitt 20.4, Bild 20.6 angedeutet wurde. Zunächst werden die Bildpunkte im Merkmalsraum „Grauwert/Betrag des Gradienten" klassifiziert. Wird ein Bildpunkt als „auf einer Kante liegend" erkannt, so kann ausgehend von diesem Punkt die Verfolgung der Linie begonnen werden.

Oft bieten sich aus der Abfolge der Verarbeitungsschritte Bildpunkte als Kandidaten für eine Ansatzstelle an. Dazu kann als Beispiel die Segmentierung eines Bildes mit Hilfe von Baumstrukturen gebracht werden (Kapitel 30). Hier wird die Baumstruktur zunächst nach homogenen (unstrukturierten) Bildbereichen untersucht, dann werden inhomogene (strukturierte) Bildbereiche extrahiert. Diese Bereiche lassen sich, grob gesprochen, in Bildbereiche mit flächenhaft ausgeprägter Struktur und in Strukturbereiche einteilen, die von

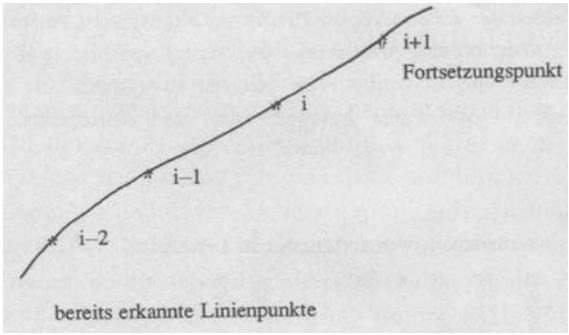


Bild 20.35: Voraussetzungen für die Linienverfolgung: Die Bildpunkte $..., i-2, i-1, i$ sind bereits als Linienpunkte erkannt; der Bildpunkt $i+1$ soll als Fortsetzungspunkt ermittelt werden.

linienhaften Bildinhalten erzeugt werden. Nach der Segmentierung von den Bildbereichen, die von der flächenhaften Struktur eingenommen werden, bleiben also im wesentlichen die Linienstrukturen übrig, sodass hier Bildpunkte gezielt als Ansatzstellen ausgewählt werden können.

Weitere Möglichkeiten der Suche von Ansatzstellen sind die Überlagerung mit einem Gitternetz und die Auswertung der einzelnen Flächenstücke oder die Bewertung des Bildinhaltes im Rahmen eines Modells, das im jeweiligen Anwendungsfall bestimmte Bildbereiche als wahrscheinlicher für die Beinhaltung einer Linie erscheinen lässt.

Jetzt zur zweiten Fragestellung, dem Finden von Nachfolgepunkten, wenn schon ein Teil der Linie verarbeitet ist. Die Darstellung der einzelnen Vorgehensweisen geht immer von folgenden Voraussetzungen aus (Bild 20.35): Es wurden bereits die Punkte $..., i-2, i-1, i$ als auf der Linie liegend erkannt. Im nächsten Schritt soll der Punkt $i+1$ als Fortsetzungspunkt ermittelt werden.

Eine Möglichkeit besteht nun darin, vom Punkt i ausgehend Suchstrahlen zu berechnen und entlang dieser Suchstrahlen den mittleren Grauwert zu ermitteln (Bild 20.36).

Wird z.B. eine dunkle Linie auf einem hellen Hintergrund angenommen, so werden die mittleren Grauwerte entlang der Suchstrahlen etwa den Verlauf von Bild 20.37 zeigen. Der Suchstrahl mit der Richtung Nummer 4 wäre dann der zu wählende, der Abstand des Fortsetzungspunktes $i+1$ von Punkt i könnte etwa einer vorgegebenen Konstante entsprechen.

Bei diesem (aber auch bei allen folgenden) Verfahren kann der Suchbereich aus der Vorgeschichte der bereits gefundenen Bildpunkte eingeschränkt werden. Wird zu den letzten k bereits gefundenen Bildpunkten eine Ausgleichsgerade berechnet, so kann ein bestimmtes Winkelfeld (*Suchfeld*) auf beiden Seiten einer Geraden als Suchbereich verwendet werden.

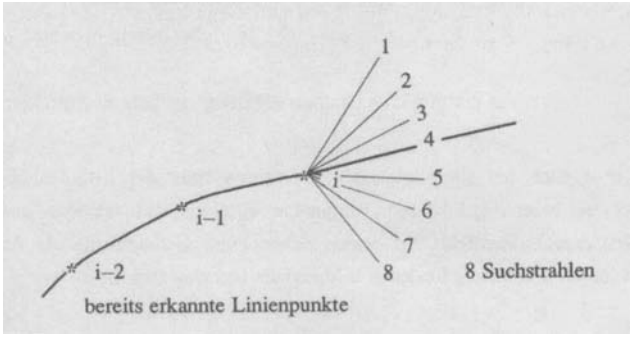


Bild 20.36: Linienverfolgung durch Berechnung von Suchstrahlen.

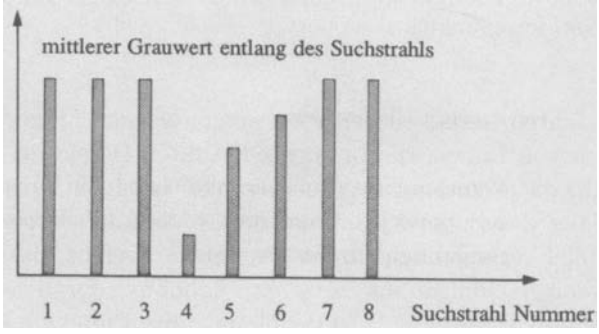


Bild 20.37: Verlauf der mittleren Grauwerte entlang der acht Suchstrahlen.

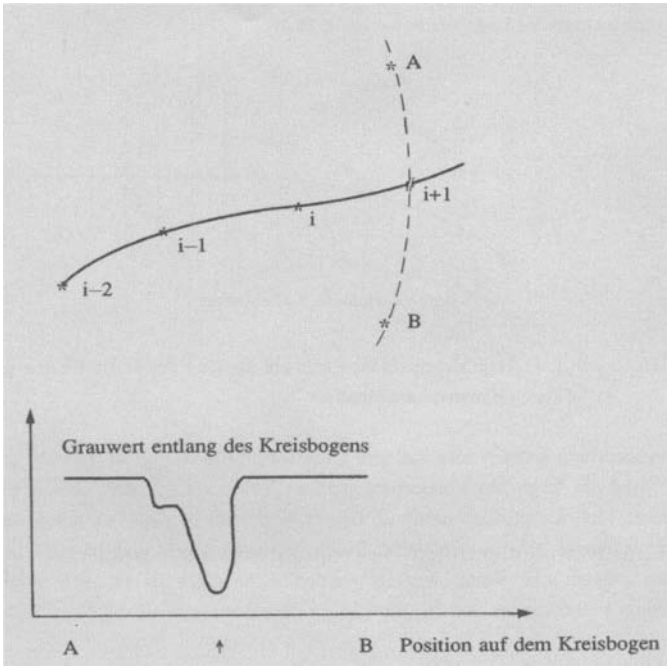


Bild 20.38: Kreisbogenverfahren (a) Kreisbogen vom Punkt A zum Punkt B, begrenzt durch den Suchbereich. (b) Grauwertprofil entlang des Kreisbogens. Die markierte Position wird als Lage des Nachfolgepunktes akzeptiert.

Wie ist dieses Verfahren bezüglich der oben aufgeworfenen Fragestellungen zu bewerten? Das Überbrücken von Linien, die in eingeschränkten Bereichen eine Unterbrechung aufweisen, ist möglich, da ja entlang der Suchstrahlen gemittelt wird. Bei starken Kontraständerungen entlang des Linienverlaufs ist dieses Verfahren nicht geeignet. Bei Verzweigungen oder parallel verlaufenden Linien besteht die Gefahr, dass das Suchverfahren an der falschen Stelle den Nachfolgepunkt setzt. Zur Erhöhung der Sicherheit kann hier vom gefundenen Nachfolgepunkt $i+1$ eine Rückverfolgung zum Punkt i angeschlossen werden: Wenn die Rückverfolgung nicht in einer vorgegebenen Kreisumgebung des Linienpunktes i endet, so wird der Punkt $i+1$ nicht als Nachfolger akzeptiert.

Nun eine andere Strategie: Vom Linienpunkt i als Mittelpunkt werden entlang eines Kreisbogens, der durch den Suchbereich definiert ist, die Grauwerte untersucht (Bild 20.38). Lässt sich im Grauwertprofil (im betrachteten Beispiel) ein lokales Minimum ermitteln, so wird die Position des Minimums als Lage des Nachfolgepunktes akzeptiert.

Bei oft unterbrochenen Linien ist dieses Verfahren nicht geeignet. Verbessert kann es

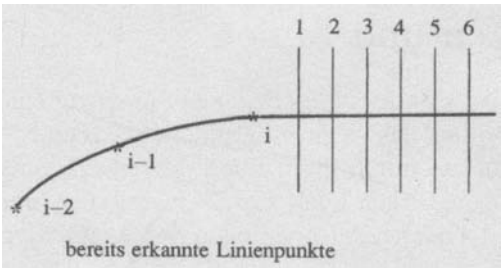


Bild 20.39: Das Mehrbereichsverfahren. Entlang der sechs Richtungen werden die Grauwerte untersucht.

dadurch werden, dass mehrere Suchkreisbögen in unterschiedlichen Abständen berechnet werden. Für die anderen Fragen gelten wohl dieselben Aussagen wie im ersten Verfahren. Eine Verzweigung drückt sich u.U. im Grauwertprofil der Kreisbögen aus. Allerdings kann zunächst nicht zwischen „Verzweigung" und „paralleler Linie" unterschieden werden.

Die Erweiterung des eben besprochenen Verfahrens, das auch als *Einbereichsverfahren* bezeichnet wird, ist das *Mehrbereichsverfahren*, bei dem, wie schon oben angedeutet, der Suchbereich in unterschiedlichen Abständen untersucht wird. Um Rechenzeit zu sparen, müssen hier nicht unbedingt Kreisbögen berechnet werden, sondern es genügt auch die Überlagerung eines Linienrasters, bei dem die Grauwertprofile etwa senkrecht zur Fortsetzungsrichtung der Linie verlaufen (Bild 20.39).

Zur Auswertung müssen hier mehrere Grauwertprofile betrachtet werden. Im obigen Beispiel wird die Lage des Fortsetzungspunktes aus der Lage der lokalen Minima in den einzelnen Grauwertprofilen ermittelt. Dieses Verfahren ist natürlich gut geeignet, wenn die Linien teilweise unterbrochen sind. Auch Verzweigungen und parallel laufende Linien können erkannt und weiter verfolgt werden. Schließlich ist es auch möglich, Kontraständerungen zu erkennen und die Verfolgung darauf abzustimmen.

Abschließend seien noch stichwortartig einige weitere Möglichkeiten aufgezeigt. Die in Abschnitt 20.11 erläuterte Houghtransformation kann ebenfalls bei sequentiellen Methoden eingesetzt werden. Dazu wird der Suchbereich, etwa ein Rechteck, in dem die zu verfolgende Linie wahrscheinlich verläuft, in den (r, φ)-Raum transformiert und der Verlauf der Linien aus der Auswertung des (r, φ)-Raumes abgeleitet. Für diese Vorgehensweise gelten etwa dieselben Gütekriterien wie für das Mehrbereichsverfahren.

Schließlich kann der weitere Verlauf der Linie auch modelliert und durch Vergleich des Modells mit dem tatsächlichen Verlauf ermittelt werden (Abschnitt 20.4).

20.15 Abschließende Bemerkungen zu Kanten und Linien

Bei vielen Anwendungen ist es notwendig, Segmente in einem Bild zu erkennen und einer bestimmten Klasse zuzuordnen. Die Segmentierung wird in den Kapiteln 23 bis 33 ausführlich erläutert. Wenn Segmente durch ihren Rand oder durch ihr Skelett dargestellt sind, so kann diese Information natürlich auch zur Segmentierung und Erkennung verwendet werden. Hier sei z.B. auf die beiden Ausprägungen des Strahlenverfahrens (Kapitel 36 und 37) hingewiesen.

Ein anderer Gesichtspunkt sei an dieser Stelle ebenfalls kurz angesprochen. Bei der Verarbeitung von Bildfolgen (Bewegungsdetektion, Tracking, Stereobildauswertung) werden oft *Verschiebungsvektorfelder* (*optical flow*) berechnet (Kapitel 26). Sie geben an, wie z.B. in einem Bild zum Zeitpunkt t einzelne Bildblöcke verschoben werden müssen, um daraus das Bild zum Zeitpunkt $t + 1$ zu erhalten. Natürlich kann man nicht für alle Bildpunkte einen Verschiebungsvektor berechnen: In homogenen Bildbereichen ist es nicht möglich (und evtl. auch nicht notwendig), Verschiebungsvektoren zu ermitteln. Es werden daher Bildpunkte gesucht, die bevorzugt zur Berechnung von Verschiebungsvektoren geeignet sind. Diese Punkte werden *markante Punkte* genannt. Ein Algorithmus dazu heißt *Punktefinder* oder auch *Deskriptor*.

Welche Kriterien können in einem Punktefinder verwendet werden? Ein markanter Punkt zeichnet sich dadurch aus, dass an dieser Stelle eine starke Grauwertänderung stattfindet, also an dieser Stelle der Gradient einen hohen Wert aufweist. Somit können markante Punkte dadurch gefunden werden, dass in einem Bild die Gradientenbeträge (z.B. Sobeloperator, Abschnitt 18.4) mit einem Schwellwert verarbeitet werden. Alle Punkte, deren Gradientenbetrag über dem Schwellwert liegt, werden als Kandidaten für markante Punkte betrachtet.

Kapitel 21

Operationen im Frequenzbereich

21.1 Anwendungen

Im Rahmen der digitalen Bildverarbeitung werden Operationen im Frequenzbereich häufig eingesetzt. Vor allem die Fouriertransformation wird zur digitalen Filterung verwendet. Mit den Methoden der digitalen Filterung wird versucht, Grauwertstörungen im Bild zu eliminieren. Diese Grauwertstörungen können z.B. durch systematische Fehler bei der Aufzeichnung oder der Digitalisierung des Bildes verursacht werden. Ein anderes Anwendungsgebiet der Fouriertransformation und der Cosinustransformation ist die Bildcodierung. Hier wird z.B. untersucht, ob ein digitalisiertes Bild weniger Bandbreite bei der Übertragung benötigt, wenn man es anstatt der Grauwerte im Ortsbereich anhand der Transformationskoeffizienten (den Linearfaktoren der Fourier- oder Cosinustransformation) im Ortsfrequenzbereich beschreibt. Schließlich wird die Fouriertransformation auch bei der Charakterisierung von Oberflächenstrukturen oder der Beschreibung der Form eines Objektes verwendet.

Es folgen einige Grundlagenabschnitte, die den mathematischen Hintergrund der linearen Transformationen etwas beleuchten. Dann folgt die Darstellung der diskreten zweidimensionalen Fouriertransformation, wie sie in der Bildverarbeitung verwendet wird. Schließlich folgen Bildbeispiele zur digitalen Filterung im Frequenzbereich.

21.2 Lineare Approximation

In der Mathematik, vor allem in der Numerik, wird eine Funktion $f(x)$ häufig durch eine *Approximationsfunktion* (Ersatzfunktion) $\phi(x)$ angenähert. Dies ist notwendig, wenn $f(x)$ entweder mathematisch ungünstig zu handhaben ist (Berechnung der Funktionswerte, Integration) oder nur an diskreten Stützpunkten $(x_i, y_i = f(x_i))$ bekannt ist. Zur Approximation wird ein abgeschlossenes Intervall $[a, b]$ der reellen Zahlen betrachtet. Mit $C[a, b]$ wird die Menge aller stetigen Funktionen auf $[a, b]$ bezeichnet. Es soll $f(x) \in C[a, b]$ durch $\phi(x) \in C[a, b]$ angenähert werden. Die Approximationsfunktion $\phi(x)$ wird in der Regel von Parametern $c_0, c_1, ..., c_n$ abhängen, sodass auch $\phi(x) = \phi(x, c_0, c_1, ..., c_n)$ geschrieben

werden kann.

Wenn sich die Approximationsfunktion $\phi(x)$ in der Form

$$\phi(x) = \phi(x, c_0, c_1, ..., c_n) = c_0\varphi_0(x) + c_1\varphi_1(x) + ... + c_n\varphi_n(x) \tag{21.1}$$

darstellen lässt, so spricht man von einer *linearen Approximation*. Die Funktionen $\varphi_i(x), i = 0, 1, ..., n$ werden als ein *Funktionssystem* (ein *System von Basisfunktionen*) bezeichnet und die $c_i, i = 0, 1, ..., n$ als *Linearfaktoren*. Bekannte Beispiele sind die Darstellung von Funktionen als endliche oder unendliche Reihen, etwa:

$$f(x) = e^x = 1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + \frac{1}{3!}x^3 + ... = 1 + \frac{1}{1!}x + \frac{1}{2!}x^2 + ... + \frac{1}{n!}x^n + R. \tag{21.2}$$

Hier ist das Funktionensystem

$$\varphi_0(x) = 1, \varphi_1(x) = x, \varphi_2(x) = x^2, ..., \varphi_n(x) = x^n. \tag{21.3}$$

Für die Linearfaktoren gilt:

$$c_0 = 1, c_1 = \frac{1}{1!}, c_2 = \frac{1}{2!}, ..., c_n = \frac{1}{n!}. \tag{21.4}$$

R ist ein Restglied, das den Fehler enthält der entsteht, wenn die an sich unendliche Reihenentwicklung zur praktischen Berechnung nach $n + 1$ Komponenten abgebrochen wird.

Die Linearfaktoren $c_i, i = 0, 1, ...n$ werden nach der Methode der *kontinuierlichen* oder *diskreten linearen Approximation im quadratischen Mittel* berechnet. Dies führt zu den Gauß'schen Normalgleichungen, einem System von linearen Gleichungen für die Berechnung [1] der c_i.

Zunächst zum stetigen Fall. Hier wird folgende Norm zugrunde gelegt:

$$\| g \|_2 = \left(\int_a^b w(x) g^2(x) dx \right)^{\frac{1}{2}}. \tag{21.5}$$

Dabei ist $g \in C[a, b]$ und $w(x) > 0$ eine auf $[a, b]$ integrierbare Gewichtsfunktion. Die Gauß'schen Normalgleichungen lauten dann:

$$\sum_{k=0}^{n} c_k \int_a^b w(x)\varphi_j(x)\varphi_k(x)dx = \int_a^b w(x)f(x)\varphi_j(x)dx, \quad j = 0, ..., n. \tag{21.6}$$

[1] Aus numerischen Gründen werden die Linearfaktoren in der Praxis jedoch nicht direkt aus den Gauß'schen Normalgleichungen berechnet, sondern über andere Methoden, z.B. die Householdertransformation.

Im diskreten Fall wird $\phi(x)$ als Approximation für eine Funktion $f(x) \in C[a,b]$ gesucht, die an $N+1$ vorgegebenen Stützpunkten $(x_i, y_i = f(x_i)), \quad i = 0, ..., N$ bekannt ist. Als Norm verwendet man hier:

$$\| g \|_2 = \left(\sum_{i=0}^{N} w_i g^2(x_i) \right)^{\frac{1}{2}}, \tag{21.7}$$

die $w_i > 0$ sind Gewichtskoeffizienten. Hier ergibt sich für die Gauß'schen Normalgleichungen:

$$\sum_{k=0}^{n} c_k \sum_{i=0}^{N} w_i \varphi_j(x_i) \varphi_k(x_i) = \sum_{i=0}^{N} w_i f(x_i) \varphi_j(x_i), \quad j = 0, ..., n, \quad N \geq n. \tag{21.8}$$

Von besonderem Interesse sind solche Systeme von Basisfunktionen, die *orthogonal* oder *orthonormiert* sind. Im kontinuierlichen Fall heisst $\varphi_0(x), \varphi_1(x), ..., \varphi_n(x)$ ein orthogonales System, wenn gilt:

$$\int_a^b w(x) \varphi_j(x) \varphi_k(x) dx = 0, \text{ für } j, k = 0, 1, ..., n, j \neq k. \tag{21.9}$$

Das System heisst orthonormiert, wenn zusätzlich gilt:

$$\int_a^b w(x) \varphi_j(x) \varphi_j(x) dx = 1, \text{ für } j = 0, 1, ..., n. \tag{21.10}$$

Sinngemäß im diskreten Fall:

$$\sum_{i=0}^{N} w_i \varphi_j(x_i) \varphi_k(x_i) = 0, \text{ für } j, k = 0, 1, ..., n, j \neq k, \quad N \geq n \tag{21.11}$$

und

$$\sum_{i=0}^{N} w_i \varphi_j(x_i) \varphi_j(x_i) = 1, \text{ für } j = 0, 1, ..., n, \quad N \geq n. \tag{21.12}$$

Wenn man in (21.6) oder (21.8) ein Funktionssystem einsetzt, das orthogonal oder orthonormiert ist, gestaltet sich die Berechnung der Linearfaktoren über die Gauß'schen Normalgleichungen einfacher als bei einem nicht orthogonalen System.

Legt man nun eine Menge von Basisfunktionen fest, so kann man eine Funktion $f(x)$ auch durch die Linearfaktoren beschreiben, die in der Approximationsfunktion $\phi(x)$ auftreten. So könnte z.B. zwischen einem Coder (als Sender von Informationen) und einem Decoder (als Empfänger von Informationen) ein bestimmtes Basisfunktionssystem festgelegt werden. Der Coder berechnet die Linearfaktoren c_i der Approximationsfunktion $\phi(x)$

und überträgt sie dem Decoder. Dieser kann aufgrund der bekannten Basis die Approximationsfunktion $\phi(x)$ und damit näherungsweise $f(x)$ rekonstruieren. Beispiele dazu sind die Fourier- oder die Cosinustransformation (Abschnitte 21.4 und 21.9).

Es zeigt sich außerdem, dass manche Eigenschaften der Funktion $f(x)$ anhand der Linearfaktoren der Approximationsfunktion besser zu erkennen sind als in der Originalfunktion. So können z.B. störende Frequenzen eines zeitabhängigen Signals $f(t)$ nach einer Fouriertransformation erkannt und eliminiert (gefiltert) werden.

21.3 Trigonometrische Approximationsfunktionen

Periodische Funktionen mit der Periode 2π ($f(x + 2\pi) = f(x)$, der Übergang zu einer anderen Periode ist möglich) lassen sich durch ihre Fourierreihe darstellen:

$$f(x) = \frac{\alpha_0}{2} + \sum_{k=1}^{\infty} \Big(\alpha_k \cos(kx) + \beta_k \sin(kx)\Big). \tag{21.13}$$

Die Linearfaktoren (Fourierkoeffizienten) α_k und β_k berechnen sich mit den Euler'schen Formeln:

$$\alpha_k = \frac{1}{\pi} \int_{-\pi}^{+\pi} f(x) \cos(kx) dx, k = 0, 1, ... \text{ und } \beta_k = \frac{1}{\pi} \int_{-\pi}^{+\pi} f(x) \sin(kx) dx, k = 1, 2, ... \tag{21.14}$$

Zur praktischen Berechnung wird $f(x)$ durch ein endliches trigonometrisches Polynom $\phi(x)$ approximiert:

$$f(x) \approx \phi(x) = \frac{a_0}{2} + \sum_{k=1}^{n} \Big(a_k \cos(kx) + b_k \sin(kx)\Big). \tag{21.15}$$

Durch einen Vergleich lassen sich leicht die Basisfunktionen und die Linearfaktoren des trigonometrischen Polynoms mit der allgemeinen linearen Approximationsformel (21.1) in Beziehung setzen. Allerdings ist zu beachten, dass in (21.15) der Index n eine andere Bedeutung hat als in (21.1):

$$\begin{array}{ll} \varphi_0(x) = 1 & c_0 = \frac{a_0}{2} \\ \varphi_1(x) = \cos(x) & c_1 = a_1 \\ \varphi_2(x) = \sin(x) & c_2 = b_1 \\ \varphi_3(x) = \cos(2x) & c_3 = a_2 \\ \varphi_4(x) = \sin(2x) & c_4 = b_2 \\ ... & ... \end{array}$$

Es lässt sich zeigen, dass die Basisfunktionen des trigonometrischen Polynoms orthogonal sind. Die Linearfaktoren berechnen sich auch hier aus den Gauß'schen Normalgleichungen. Im Folgenden wird nur mehr der diskrete Fall behandelt. Dazu wird vorausgesetzt, dass von $f(x)$ die $2N$ äquidistanten Stützpunkte $(x_j, y_j = f(x_j))$ bekannt sind, mit $x_j = j(\pi/N)$.

Hier sind $2n+1$ unbekannte Parameter $a_0, a_1, ..., a_n, b_1, b_2, ..., b_n$ zu berechnen. Somit muss für die Anzahl der Stützpunkte gelten: $2n+1 \leq 2N$. Im Fall $2n+1 < 2N$ werden die Parameter wie oben nach der Methode Approximation im quadratischen Mittel berechnet. Es ergibt sich:

$$a_0 = \frac{1}{N}\sum_{j=1}^{2N} y_j, \quad a_k = \frac{1}{N}\sum_{j=1}^{2N} y_j \cos(kx_j), \quad b_k = \frac{1}{N}\sum_{j=1}^{2N} y_j \sin(kx_j), \tag{21.16}$$

für $k = 1, ..., n$.

Falls $n = N$ gilt, ergibt sich für $b_N = 0$. Es sind dann nur $2n$ Parameter zu berechnen und die Anzahl der Parameter ist gleich der Anzahl der Stützpunkte. In diesem Fall liegt eine trigonometrische Interpolation vor, die im Weiteren näher untersucht wird.

Das trigonometrische Interpolationspolynom lautet:

$$\phi(x) = \frac{a_0}{2} + \sum_{k=1}^{N-1} \Big(a_k \cos(kx) + b_k \sin(kx)\Big) + a_N \cos(Nx) \tag{21.17}$$

und für die Koeffizienten gilt:

$$\begin{aligned} a_0 &= \frac{1}{N}\sum_{j=1}^{2N} y_j, \quad a_N = \frac{1}{N}\sum_{j=1}^{2N} (-1)^j y_j, \\ a_k &= \frac{1}{N}\sum_{j=1}^{2N} y_j \cos(kx_j), \quad b_k = \frac{1}{N}\sum_{j=1}^{2N} y_j \sin(kx_j), k = 1, ..., N-1. \end{aligned} \tag{21.18}$$

Dieser Sachverhalt lässt sich auch in vektorieller Form schreiben. Es sei

$$\mathbf{f} = (f(x_1), f(x_2), ..., f(x_{2N}))^T \text{ und } \mathbf{\Phi} = (\phi(x_1), \phi(x_2), ..., \phi(x_{2N}))^T, \tag{21.19}$$

wobei wegen der Periodizität $x_0 = x_{2N}$ gilt. Da $\phi(x)$ die Funktion $f(x)$ interpoliert, gilt $\mathbf{f} = \mathbf{\Phi}$.

Nun lässt sich $\mathbf{f}$ als Linearkombination von $2N$ linear unabhängigen und orthogonalen Basisvektoren $\mathbf{v}_j, j = 0, ..., 2N-1$ schreiben:

$$\mathbf{f} = \begin{pmatrix} f(x_1) \\ f(x_2) \\ ... \\ f(x_{2N}) \end{pmatrix} = \frac{a_0}{2}\begin{pmatrix} 1 \\ 1 \\ ... \\ 1 \end{pmatrix} + a_1 \begin{pmatrix} \cos(x_1) \\ \cos(x_2) \\ ... \\ \cos(x_{2N}) \end{pmatrix} + b_1 \begin{pmatrix} \sin(x_1) \\ \sin(x_2) \\ ... \\ \sin(x_{2N}) \end{pmatrix} + \tag{21.20}$$

$$+ \quad a_2 \begin{pmatrix} \cos(2x_1) \\ \cos(2x_2) \\ ... \\ \cos(2x_{2N}) \end{pmatrix} + b_2 \begin{pmatrix} \sin(2x_1) \\ \sin(2x_2) \\ ... \\ \sin(2x_{2N}) \end{pmatrix} + \quad ... \quad + a_{N-1} \begin{pmatrix} \cos((N-1)x_1) \\ \cos((N-1)x_2) \\ ... \\ \cos((N-1)x_{2N}) \end{pmatrix} +$$

$$+ \quad b_{N-1} \begin{pmatrix} \sin((N-1)x_1) \\ \sin((N-1)x_2) \\ \dots \\ \sin((N-1)x_{2N}) \end{pmatrix} + a_N \begin{pmatrix} \cos(Nx_1) \\ \cos(Nx_2) \\ \dots \\ \cos(Nx_{2N}) \end{pmatrix} =$$

$$= \quad \frac{a_0}{2}\mathbf{v}_0 + a_1\mathbf{v}_1 + b_1\mathbf{v}_2 + \dots + a_{N-1}\mathbf{v}_{2N-3} + b_{N-1}\mathbf{v}_{2N-2} + a_N\mathbf{v}_{2N-1}.$$

Aufbauend auf dieser Darstellung wird in Abschnitt 21.4 die diskrete zweidimensionale Fouriertransformation erläutert. An die Stelle des Vektors $\mathbf{f}$ tritt eine Bildmatrix $\mathbf{S} = (s(x,y))$ die im allgemeinen Fall auch komplex sein kann. Statt der Basisvektoren $\mathbf{v}_j$ treten Basismatrizen $\mathbf{B}_{(u,v)}$ auf.

21.4 Diskrete zweidimensionale Fouriertransformation

Mit $Q_{L,R}$ wird die Menge aller $L \cdot R$-Matrizen bezeichnet, wobei die Elemente einer Matrix auch komplexe Zahlen sein können. $Q_{L,R}$ wird zu einem linearen Raum über dem Körper C der komplexen Zahlen, wenn die Addition von zwei Matrizen $\mathbf{S}$ und $\mathbf{T}$ und die linearen Operationen mit Elementen a, b aus C definiert werden:

$$\begin{aligned} &\mathbf{S} + \mathbf{T} = \Big(s(x,y)\Big) + \Big(t(x,y)\Big) = \Big(s(x,y) + t(x,y)\Big). \\ &(ab)\mathbf{S} = a(b\mathbf{S}) = a\Big(b \cdot s(x,y)\Big) = \Big(ab \cdot s(x,y)\Big). \\ &(a+b)\mathbf{S} = a\mathbf{S} + b\mathbf{S}. \\ &a(\mathbf{S} + \mathbf{T}) = a\mathbf{S} + a\mathbf{T}. \end{aligned} \tag{21.21}$$

Schließlich wird noch ein Skalarprodukt $< \mathbf{S}, \mathbf{T} >$ definiert:

$$< \mathbf{S}, \mathbf{T} >= \frac{1}{L \cdot R} \sum_{x=0}^{L-1} \sum_{y=0}^{R-1} s(x,y) \cdot t^*(x,y), \tag{21.22}$$

wobei mit $t^*(x,y)$ die konjugiert komplexe Zahl zu $t(x,y)$ bezeichnet wird. Damit ist $Q_{L,R}$ ein unitärer Raum über dem Körper C der komplexen Zahlen. Die Elemente $\mathbf{S}$ von $Q_{L,R}$ sind als Linearkombinationen von $M = L \cdot R$ linear unabhängigen Basismatrizen (einer Basis) darstellbar:

$$\begin{aligned} &\mathbf{B}_{u,v} = \Big(b(x,y)\Big)_{u,v}; \quad u = 0, \dots, L-1; v = 0, \dots, R-1. \\ &s(x,y) = \sum_{u=0}^{L-1} \sum_{v=0}^{R-1} f(u,v) b(x,y)_{u,v}; \quad x = 0, \dots, L-1; y = 0, \dots, R-1. \end{aligned} \tag{21.23}$$

Für eine orthonormierte Basis muss gelten:

$$< \mathbf{B}_{u_1,v_1}, \mathbf{B}_{u_2,v_2} >= 0, \text{ falls } (u_1, v_1) \neq (u_2, v_2) \text{ und} \tag{21.24}$$
$$< \mathbf{B}_{u_1,v_1}, \mathbf{B}_{u_2,v_2} >= 1, \text{ falls } (u_1, v_1) = (u_2, v_2).$$

Es lässt sich zeigen, dass die Basismatrizen

$$\mathbf{B}_{u,v} = \left(\exp\left(i2\pi\left(\frac{x \cdot u}{L} + \frac{y \cdot v}{R}\right)\right)\right); \quad u = 0, ..., L-1; v = 0, ..., R-1 \tag{21.25}$$

linear unabhängig und orthonormiert sind. Somit kann eine $(L \cdot R)$-Matrix $\mathbf{S} = (s(x, y))$ als Linearkombination dieser Basismatrizen geschrieben werden:

$$s(x, y) = \sum_{u=0}^{L-1} \sum_{v=0}^{R-1} f(u, v) \exp\left(i2\pi\left(\frac{x \cdot u}{L} + \frac{y \cdot v}{R}\right)\right); \tag{21.26}$$
$$x = 0, ..., L-1; y = 0, ..., R-1.$$

Da die Basismatrizen orthonormiert sind, berechnen sich die Linearfaktoren $f(u, v)$ gemäß:

$$f(u, v) = \frac{1}{L \cdot R} \sum_{x=0}^{L-1} \sum_{y=0}^{R-1} s(x, y) \exp\left(-i2\pi\left(\frac{x \cdot u}{L} + \frac{y \cdot v}{R}\right)\right); \tag{21.27}$$
$$u = 0, ..., v-1; y = 0, ..., R-1.$$

Die in (21.25) definierten Matrizen sind die Basis der *diskreten, zweidimensionalen Fouriertransformation*, (21.27) wird als *diskrete, zweidimensionale Fouriertransformation* und (21.26) als *inverse, diskrete, zweidimensionale Fouriertransformation* bezeichnet. $\mathbf{S} = (s(x, y))$ und $\mathbf{F} = (f(u, v))$ sind ein *Fouriertransformationspaar* und $\mathbf{F}$ heißt die *Fouriertransformierte* von $\mathbf{S}$.

Es ist nun der Bezug zu digitalisierten Bildern herzustellen. Eine Bildmatrix $\mathbf{S} = (s(x, y))$ mit L Bildzeilen und R Bildspalten ist ein Element der oben definierten Menge $Q_{L,R}$. Damit können auf Bilder die Transformationen (21.27) und (21.26) angewendet werden. Wenn man die obige Ableitung der diskreten, zweidimensionalen Fouriertransformation nicht über einem unitären Raum von $L \cdot R$ Matrizen, sondern über einem unitären Raum von periodischen Funktionen $s(x, y)$ der diskreten Variablen x und y durchführt, stellt man sich das Bild $\mathbf{S}$, wie in Kapitel 18, Bild 18.2 gezeigt, periodisch in $x-$ und $y-$Richtung fortgesetzt vor. Auch die Fouriertransformierte $f(u, v)$ ist dann eine periodische Funktion der diskreten Varibalen u und v, die dann als *Ortsfrequenzen* bezeichnet werden.

Die Fouriertransformierte von $\mathbf{S}$ ist in der Regel komplex: $\mathbf{F} = \mathbf{P} + i\mathbf{T}$, wobei $\mathbf{P} = (p(u,v))$ und $\mathbf{T} = (t(u,v))$ der Realteil und der Imaginärteil von $\mathbf{F} = (f(u,v))$ sind. Die Beträge

$$\big(\mid f(u,v) \mid \big) = \Big(\sqrt{p^2(u,v) + t^2(u,v)}\Big) \tag{21.28}$$

werden als *Fourierspektrum* oder *Spektrum der Ortsfrequenzen* von $\mathbf{S}$ bezeichnet. Die höchste auf einem bestimmten Trägermedium darstellbare Ortsfrequenz bezeichnet man auch als *Auflösung*. Man versteht darunter die durch die physikalischen Eigenschaften des Trägermediums bedingte dichteste Folge von noch unterscheidbaren hellen und dunklen Linien.

Für die qualitative Beurteilung der in einem Bild $\mathbf{S} = (s(x,y))$ vorhandenen Ortsfrequenzen wird für das Spektrum $(\mid f(u,v) \mid)$ häufig eine bildhafte Darstellung angestrebt. Die Grauwerte werden z.B. so gewählt, dass sie zum Betrag $\mid f(u,v) \mid$ der jeweiligen, komplexen Spektralkomponente $f(u,v)$ proportional sind. Sollen in der bildlichen Darstellung Spektralkomponenten mit kleinem Betrag stärker betont werden als solche mit großem Betrag, so können die Grauwerte $p(u,v)$ einer bildlichen Darstellung von $\mid f(u,v) \mid$ berechnet werden gemäß

$$p(u,v) = c \cdot \log\big(1 + \mid f(u,v) \mid\big), \tag{21.29}$$

wobei c ein Normierungsfaktor ist, der so gewählt werden muss, dass die Grauwerte in der Grauwertmenge G liegen. Bild 21.1 zeigt ein Testbild und daneben die bildliche Darstellung des Spektrums. Andere diskrete (unitäre) Transformationen von $\mathbf{S} = (s(x,y))$ erhält man, wenn die Basismatrizen in (21.25) anders gewählt werden. Beispiele dazu sind die Cosinus-, die Hadamar- oder die Haartransformation.

21.5 Digitale Filterung im Ortsfrequenzbereich

Eine digitale Filterung im Bereich der Ortsfrequenzen eines Bildes $\mathbf{S}_e$ besteht aus der Veränderung der Fouriertransformierten $(f_e(u,v))$ durch die Multiplikation mit einer Übertragungsfunktion $g(u,v)$. Man erhält dann die gefilterte Fouriertransformierte $f_a(u,v)$ des Ausgabebildes $\mathbf{S}_a$:

$$f_a(u,v) = g(u,v) \cdot f_e(u,v); u = 0, ..., L-1; v = 0, ..., R-1. \tag{21.30}$$

Je nach der Beschaffenheit der Übertragungsfunktion $g(u,v)$ spricht man von verschiedenen Filtertypen. Werden durch die Übertragungsfunktion hohe Ortsfrequenzanteile abgeschwächt oder sogar unterdrückt, so spricht man von einer *Tiefpassfilterung*. Im Gegensatz dazu werden bei einer *Hochpassfilterung* niedere Ortsfrequenzanteile abgeschwächt oder unterdrückt. Bei *Bandpassfiltern* wird die Übertragungfunktion so gewählt, dass nur die Ortsfrequenzanteile eines bestimmten Ausschnittes unverändert bleiben, während bei der

(a) (b)

Bild 21.1: Testbild zur Fouriertransformation. (a) Original. (b) Bildliche Darstellung des Spektrums.

Bandsperre die Ortsfrequenzanteile nur des Ausschnittes unterdrückt werden. Das gefilterte Bild $\mathbf{S}_a$ im Ortsbereich des (x, y)-Koordinatensystems berechnet sich mit der inversen Fouriertransformation gemäß (21.26):

$$s_a(x,y) = \sum_{u=0}^{L-1} \sum_{v=0}^{R-1} f_a(u,v) \exp\left(i2\pi\left(\frac{x \cdot u}{L} + \frac{y \cdot v}{R}\right)\right); \qquad (21.31)$$
$$x = 0, ..., L-1; y = 0, ..., R-1.$$

Da bei einem Tiefpassfilter hohe Ortsfrequenzanteile abgeschwächt werden, erscheint das gefilterte Bild $\mathbf{S}_a$ im Vergleich zum Original unschärfer. Die hohen Ortsfrequenzen treten im Originalbild $\mathbf{S}_e$ vor allem in den Grauwertkanten auf. Werden durch ein Hochpassfilter die niederen Ortsfrequenzen unterdrückt, so werden im gefilterten Bild $\mathbf{S}_a$ vor allem die Grauwertkanten deutlich hervortreten. An dieser Stelle sei auf den Zusammenhang zwischen der Tiefpassfilterung im Ortsfrequenzbereich mit den Summenoperatoren von Abschnitt 18.3 einerseits und zwischen der Hochpassfilterung im Frequenzbereich und den Differenzenoperatoren von Abschnitt 18.4 hingewiesen. Dieser Zusammenhang wird im folgenden Abschnitt noch näher erläutert.

Zunächst aber noch zwei Beispiele: Eine Übertragungsfunktion mit Tiefpass- und eine mit Hochpasscharakter. Für das Folgende wird angenommen, dass das Bild $\mathbf{S}_e = (s_e(x, y))$ quadratisch ist, also $L = R = N$. Es lässt sich zeigen, dass die Fouriertransformierte von $\mathbf{S}_e$

punktsymmetrisch zu $(N/2, N/2)$ ist, wenn vor der Fouriertransformation die Originalwerte mit $(-1)^{(x+y)}$ multipliziert werden. Diese Form wird auch bei der bildlichen Darstellung der Beträge Fouriertransformierten (des Betragsspektrums) gewählt.

Ein Tiefpassfilter ist dann z.B.:

$$g_T(u,v) = \frac{1}{2} + \frac{1}{2}\left\{ \cos\left(\frac{2\pi}{N\sqrt{2}} \sqrt{(u-\frac{N}{2})^2 + (v-\frac{N}{2})^2} \right) \right\}; \quad (21.32)$$
$$u = 0, ..., N-1; v = 0, ..., N-1$$

und ein Hochpassfilter z.B.:

$$g_T(u,v) = \frac{1}{2} - \frac{1}{2}\left\{ \cos\left(\frac{2\pi}{N\sqrt{2}} \sqrt{(u-\frac{N}{2})^2 + (v-\frac{N}{2})^2} \right) \right\}; \quad (21.33)$$
$$u = 0, ..., N-1; v = 0, ..., N-1.$$

Zu weiteren Fragen über den Entwurf von digitalen Filtern im Frequenzbereich, also zur Berechnung von geeigneten Übertragungsfunktionen, sei z.B. auf [Abma94] verwiesen.

Auf den folgenden Seiten sind als Abschluss zu diesem Abschnitt Bildbeispiele zu verschiedenen Filterungen zusammengestellt. Sie zeigen jeweils das Original, daneben das jeweils verwendete Filter und rechts oben das Ergebnis der Filterung. Unter dem Original ist die bildliche Darstellung der Beträge der Fouriertransformierten zu sehen und unter dem Ergebnis die bildliche Darstellung der Beträge der gefilterten Fouriertransformierten.

21.6 Zusammenhang mit dem Ortsbereich

Der Zusammenhang zwischen Ortsfrequenzbereich und Ortsbereich wird durch den *Faltungssatz* vermittelt. Die diskrete Faltung zweier Funktionen $s(x,y)$ und $h(x,y)$ ist definiert als

$$\sum_{k=-\infty}^{+\infty} \sum_{l=-\infty}^{+\infty} s(k,l) \cdot h(x-k, y-l) = \sum_{k=-\infty}^{+\infty} \sum_{l=-\infty}^{+\infty} s(x-k, y-l) \cdot h(k,l); \quad (21.34)$$
$$x, y = ..., -2, -1, 0, 1, 2...$$

Abkürzend wird dafür oft $s(x,y) \star h(x,y)$ geschrieben. Im Folgenden wird mit FT bzw. mit FT^{-1} die diskrete, bzw. die inverse, diskrete, zweidimensionale Fouriertransformation bezeichnet. Dann lautet der Faltungssatz:

$$\mathrm{FT}\big(s(x,y)\big) \cdot \mathrm{FT}\big(h(x,y)\big) = \mathrm{FT}\big(s(x,y) \star h(x,y)\big). \quad (21.35)$$

Das bedeutet, dass das Produkt der Fouriertransformierten von $s(x,y)$ und $h(x,y)$ gleich der Fouriertransformierten der Faltung dieser beiden Funktionen ist. In (21.30) und

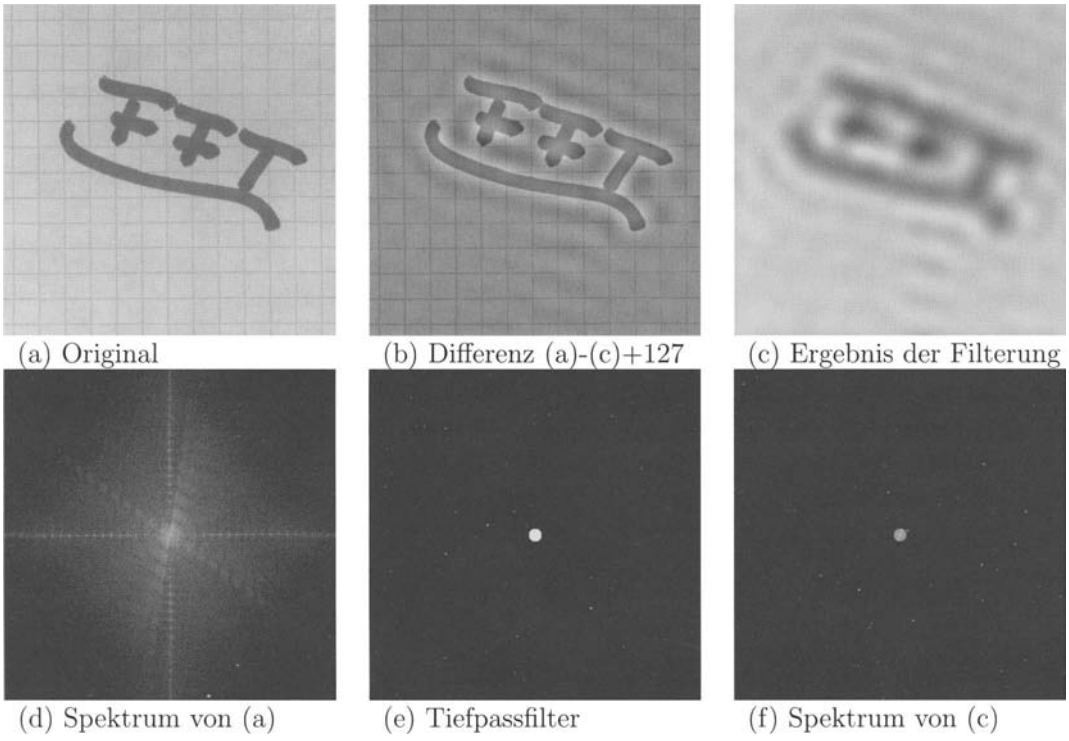


Bild 21.2: Beispiel zu einer Tiefpassfilterung. (a) Original. (b) Differenzbild: (Original - gefiltertes Bild + 127). (c) Ergebnis der Filterung: Das Bild ist deutlich unschärfer geworden. Die Wellenstruktur im Hintergrund wird durch das scharfkantige. Filter erzeugt. (d) Bildliche Darstellung des Spektrums des Originals. (e) Verwendetes Tiefpassfilter: Die Frequenzanteile im schwarzen Bereich werden auf null gesetzt, die im weißen Bereich werden unverändert übernommen. (f) Bildliche Darstellung des gefilterten Spektrums.

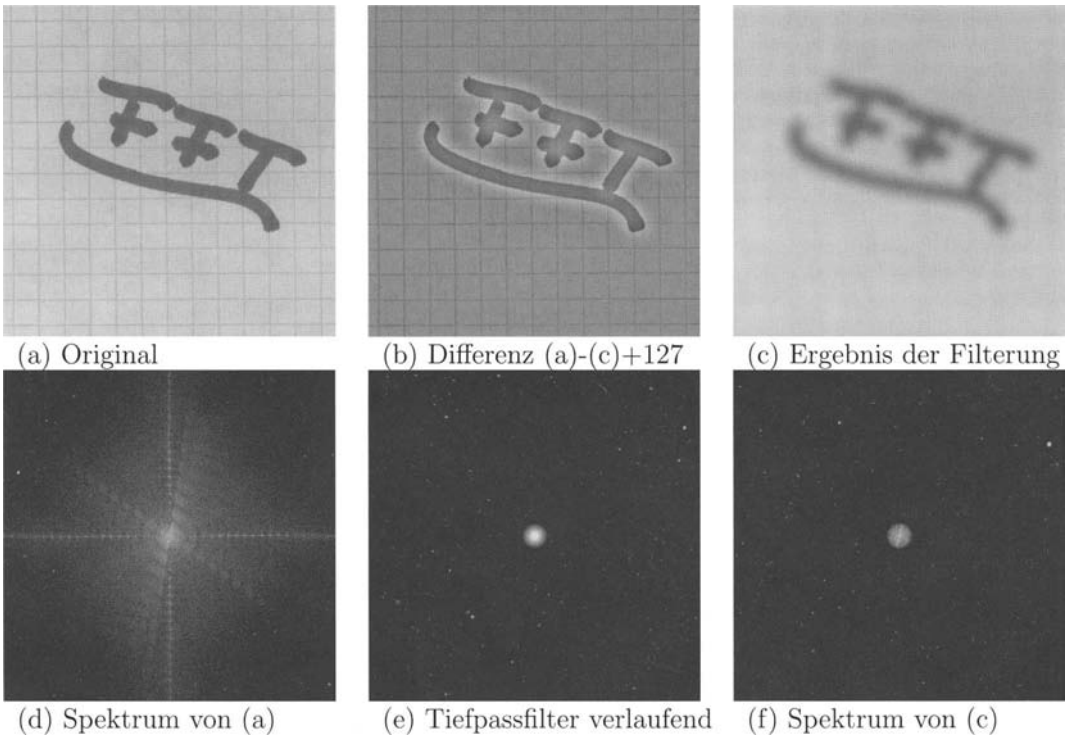


Bild 21.3: Beispiel zu einer Tiefpassfilterung. (a) Original. (b) Differenzbild: (Original - gefiltertes Bild + 127). (c) Ergebnis der Filterung: Das Bild ist deutlich unschärfer geworden. Die Wellenstruktur im Hintergrund tritt hier wegen der Verwendung eines verlaufenden Filters nicht auf. (d) Bildliche Darstellung des Spektrums des Originals. (e) Verwendetes Tiefpassfilter: Die Frequenzanteile im schwarzen Bereich werden auf null gesetzt, die im weißen Bereich werden unverändert übernommen, die Anteile im Übergangsbereich werden abgedämpft. (f) Bildliche Darstellung des gefilterten Spektrums.

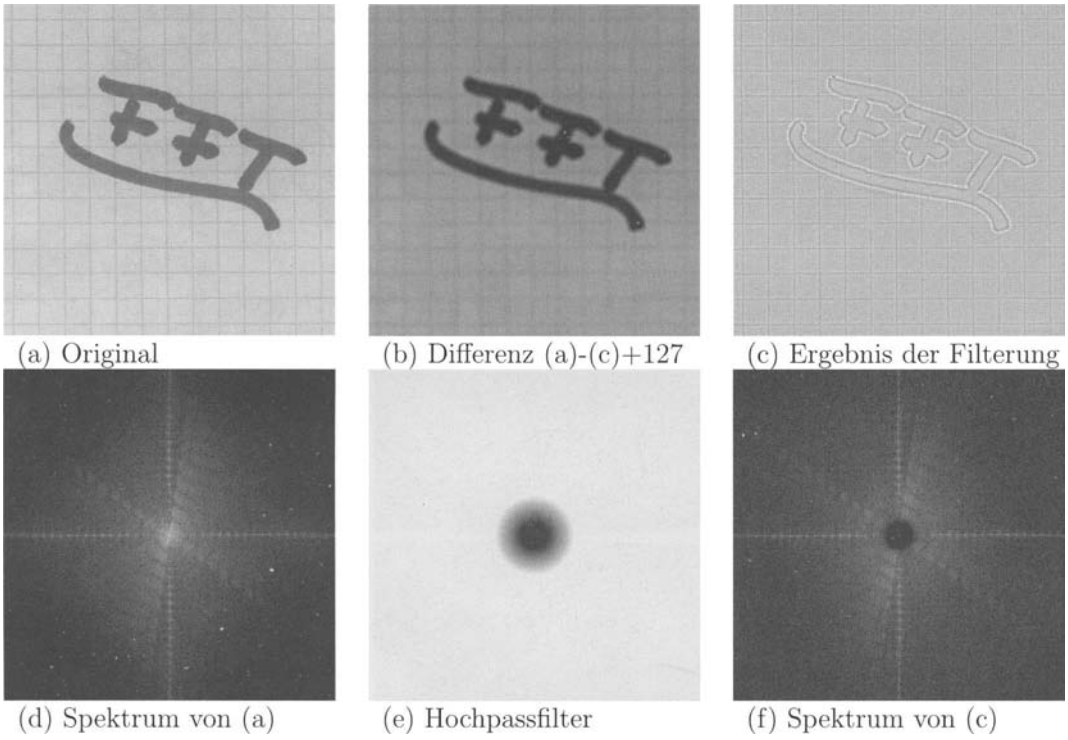


Bild 21.4: Beispiel zu einer Hochpassfilterung. (a) Original. (b) Differenzbild: (Original - gefiltertes Bild + 127). (c) Ergebnis der Filterung: Es sind im Wesentlichen nur mehr die Grauwertübergänge zu sehen, also die Bildbereiche mit hohen Ortsfrequenzen. (d) Bildliche Darstellung des Spektrums des Originals. (e) Verwendetes Hochpassfilter: Die Frequenzanteile im schwarzen Bereich werden auf null gesetzt, die im weißen Bereich werden unverändert übernommen, die Anteile im Übergangsbereich werden abgedämpft. (f) Bildliche Darstellung des gefilterten Spektrums.

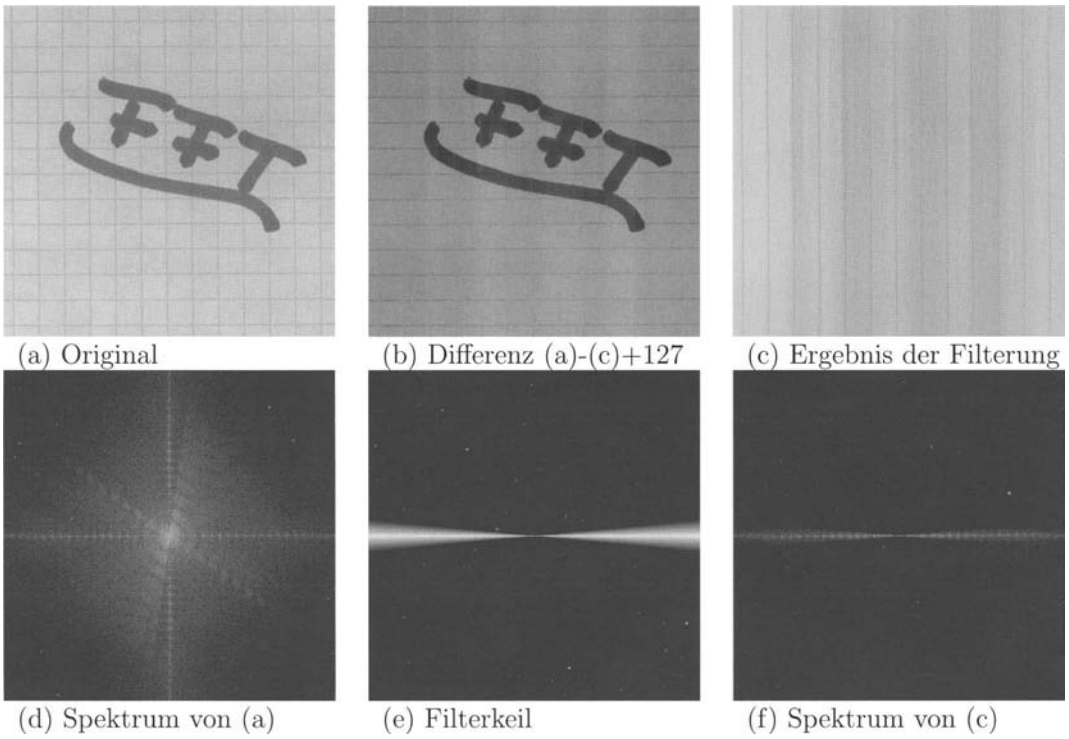


Bild 21.5: Beispiel zu einer Filterung mit einem keilförmigen Filter. (a) Original. (b) Differenzbild: (Original - gefiltertes Bild + 127). (c) Ergebnis der Filterung: Es sind im Wesentlichen nur mehr die vertikalen Linien zu sehen. (d) Bildliche Darstellung des Spektrums des Originals. (e) Verwendetes Keilfilter: Die Frequenzanteile im schwarzen Bereich werden auf null gesetzt, die im weißen Bereich werden unverändert übernommen, die Anteile im Übergangsbereich werden abgedämpft. (f) Bildliche Darstellung des gefilterten Spektrums.

Ortsbereich		Ortsfrequenzbereich
$s(x,y)$	FT $\longrightarrow$	$f(u,v)$
$\downarrow$		$\downarrow$
$s \star h$ Faltung mit der Implusantwort	$h \rightarrow \mathrm{FT} \rightarrow g$	$f \cdot g$ Multiplikation mit der Übertragungsfunktion
$\downarrow$	$h \leftarrow \mathrm{FT}^{-1} \leftarrow g$	$\downarrow$
$s'(x,y)$ Ergebnis der Filterung	FT^{-1} $\longleftarrow$	$f'(u,v)$ gefilterte Fourier-transformierte

Bild 21.6: Zusammenhänge zwischen dem Ortsbereich und dem Ortsfrequenzbereich.

(21.31) wurde dargestellt, dass die digitale Filterung im Ortsfrequenzbereich durch die Multiplikation der Fouriertransformierten $f(u,v)$ von $s(x,y)$ mit einer Übertragungsfunktion $g(u,v)$ und die anschließende inverse Fouriertransformation erreicht werden kann. Aus dem Faltungssatz kann der Bezug zum Ortsbereich abgeleitet werden: Die Filterung kann auch im Ortsbereich durchgeführt werden, wenn $s(x,y)$ mit einer Impulsantwortfunktion $h(x,y)$ gefaltet wird. Die Impulsantwort- und die Übertragungsfunktion bilden dabei ein Fouriertransformationspaar. Diese Zusammenhänge sind in Bild 21.6 grafisch dargestellt.

Digitalisierte Bilder, interpretiert als Funktionen der diskreten Variablen x und y, sind nur in einem endlichen Rechteckbereich mit der Seitenlänge $L \cdot R$ definiert. Außerhalb dieses Bereichs kann man sich vorstellen, dass sie nur den Wert Null annehmen. Dadurch reduzieren sich die unendlichen Summen auf endliche und (21.34) lautet dann:

$$\sum_{k=0}^{L-1} \sum_{l=0}^{R-1} s(x-k, y-l) \cdot h(k,l). \tag{21.36}$$

Bei praktischen Anwendungen wird die Impulsantwort auf kleinere Bereiche als das Gesamtbild begrenzt, z.B. auf $3 \cdot 3$ Bildpunkte. Der Vergleich mit (18.8) zeigt dann, dass man die in Kapitel 18 erläuterten Summen- oder Differenzenoperatoren erhält.

21.7 Logarithmische Filterung

Die logarithmische Filterung ist ein Beispiel einer digitalen Filterung im Orts- oder Frequenzbereich, verbunden mit einer nicht linearen Transformation der Grauwerte. Ein Anwendungsbeispiel ist die Elimination von ungleichmäßiger Beleuchtung, ohne dabei auf die Bildqualität bei feinen Details verzichten zu müssen.

In Bildern treten unterschiedliche Einflüsse der beleuchtenden Lichtquellen bei der Aufnahme auf, so z.B. Schatteneffekte oder Reflexionen. Die Reflexionen sind dabei oft erwünscht, da sie Details oft deutlicher hervorheben, während sich die Schatten aufgrund von unterschiedlicher Beleuchtung in der Regel nicht positiv auswirken. Die Unterschiede in der Beleuchtung treten in einem Bild meistens nicht so abrupt auf wie die Lichtreflexionen. Das bedeutet also vor dem Hintergrund der Fouriertransformation, dass sich die Lichtreflexionen mehr in den hohen Frequenzanteilen und die Beleuchtungseffekte mehr in den niederen Frequenzanteilen ausdrücken. Um nun solche Beleuchtungseffekte zu eliminieren oder zu mindestens abzudämpfen, ist eine Filterung mit Betonung der hohen Frequenzanteile angebracht.

Es wird angenommen, dass der Beleuchtungs- und der Reflexionsanteil sich multiplikativ zum Grauwert eines Bildpunktes ergänzen:

$$s(x,y) = s_B(x,y) \cdot s_R(x,y), \tag{21.37}$$

wobei $s_B(x,y)$ der Beleuchtungsanteil und $s_R(x,y)$ der Reflexionsanteil ist. Durch Logarithmieren von (21.37) erhält man:

$$\log s(x,y) = \log s_B(x,y) + \log s_R(x,y). \tag{21.38}$$

Da die Fouriertransformation eine lineare Operation ist, können die Summanden in (21.38) gezielt durch digitale Filter beeinflusst werden. Man wird hier also ein Filter im Orts- oder Frequenzbereich einsetzen, das die hohen Bildfrequenzen beeinflusst. Durch die Logarithmierung werden außerdem die hellen Bildteile mehr angesprochen als die dunklen.

Praktisch sind also folgende Schritte durchzuführen:

- nicht lineare Skalierung durch die Logarithmierung der Grauwerte,
- digitale Filterung mit Betonung hoher Bildfrequenzen,
- Rücktransformation in eine lineare Grauskala durch Exponentiation.

21.8 Inverse und Wiener Filterung

Bei der inversen Filterung wird angenommem, dass ein Bild $\mathbf{S}_a = (s_a(x,y))$ durch eine Überlagerung gestört ist, deren Impulsantwort- bzw. Übertragungsfunktion bekannt ist:

$$s_e(x,y) = \frac{1}{L \cdot R} \sum_{k=0}^{L-1} \sum_{l=0}^{R-1} s_a(x-k, y-l) \cdot h(k,l). \tag{21.39}$$

Im Frequenzbereich gilt dann sinngemäß:

$$f_e(u,v) = f_a(u,v) \cdot g(u,v), \tag{21.40}$$

wobei $f_e(u,v)$, $f_a(u,v)$ und $g(u,v)$ die Fouriertransformierten des gestörten Bildes $(s_e(x,y))$, des korrekten Bildes $(s_a(x,y))$ und der Impulsantwortfunktion $h(x,y)$ sind.

Aus (21.40) folgt, dass rein rechnerisch die Fouriertransformierte des korrekten Bildes $(s_a(x,y))$ aus dem Quotienten von $f_e(u,v)$ und $g(u,v)$ gewonnen werden kann:

$$f_a(u,v) = \frac{f_e(u,v)}{g(u,v)} \tag{21.41}$$

und $s_a(x,y)$ durch inverse Fouriertransformation in den Ortsbereich.

Bei praktischen Anwendungen von (21.41) treten jedoch Schwierigkeiten auf und zwar an den Stellen, an denen $g(u,v)$ den Wert null annimmt. Um dies zu umgehen, wird statt des korrekten inversen Filters $1/g(u,v)$ eine Näherung $m(u,v)$ verwendet, die nur diejenigen Frequenzen rekonstruiert, die ein hohes Signal-Rauschverhältnis haben. Dabei wird $m(u,v)$ so gewählt, dass an Stellen, an denen $g(u,v) \neq 0$ ist, das exakte Filter $1/g(u,v)$ verwendet wird.

Bei der Wiener Filterung wird der Fehler zwischen dem korrekten und dem gestörten Bild minimiert und zwar durch einen statistischen Ansatz. Das zu berechnende Bild $(s_a(x,y))$ wird als stochastischer Prozess beschrieben, der durch die Überlagerung der stochastischen Prozesse des korrekten Bildes $(s_a(x,y))$ mit einem Rauschanteil $(z(x,y))$ zustande kommt. Das Bild $(s_a(x,y))$ wird dabei so bestimmt, dass Standardabweichungen der Differenz des korrekten Bildes und des berechneten Bildes minimiert werden.

21.9 Diskrete, zweidimensionale Cosinustransformation

Die unitären (orthogonalen) Transformationen, zu denen auch die Fouriertransformation gehört, haben Vor- und Nachteile, abhängig vom jeweiligen Einsatzgebiet. In der *Bilddatencodierung* sucht man orthogonale Transformationen, bei denen sich die Transformationskoeffizienten möglichst dicht zusammen konzentrieren. Außerdem soll durch das Weglassen von Transformationskoeffizienten, die im Frequenzspektrum „weiter weg liegen" die Bildqualität nach der Rücktransformation nur gering verschlechtert werden.

Als erstes kann die *Hadamartransformation* erwähnt werden: Sie ist eine orthogonale lineare Transformation, die auf den Hadamar-Matrizen als Basis-Matrizen aufbaut. Hadamar-Matrizen sind quadratisch und haben in den Zeilen und Spalten nur die Werte 1 oder -1 (ein gemeinsamer Faktor wird vor die Matrix geschrieben). Hadamar-Matrizen haben die Eigenschaft $\mathbf{H}\mathbf{H}^T = \mathbf{I}$. Die kleinste Hadamartransformations-Matrix ist die $(2 \cdot 2)$-Matrix $\mathbf{H}_2$. Es gilt: Falls eine Hadamar-Matrix der Größe $(L \cdot L)$ existiert, wobei L

eine Potenz von 2 ist, so gibt es auch eine der Größe $(2L \cdot 2L)$, die nach folgender Vorschrift erhalten wird:

$$\mathbf{H}_2 = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{21.42}$$

$$\mathbf{H}_{2L} = \frac{1}{\sqrt{2}} \begin{pmatrix} \mathbf{H}_L & \mathbf{H}_L \\ \mathbf{H}_L & -\mathbf{H}_L \end{pmatrix}, \tag{21.43}$$

Ein Beispiel:

$$\mathbf{H}_4 = \frac{1}{2} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}. \tag{21.44}$$

Ob für beliebige Werte für L Hadamar-Matrizen existieren, ist nicht bewiesen. Es gibt aber für fast alle Werte bis $L = 200$ Konstruktionsmöglichkeiten. Die Hadamartransformation führt multidimensionale 45°-Drehungen durch. Die Annahme, dass diese Drehungen für die Zwecke der Bilddatencodierung immer passend sind, trifft häufig nicht zu.

Die *Hauptkomponententransformation* (Abschnitt 25.4) (*Karhunen-Loeve-Transformation*) liefert zwar immer optimale Ergebnisse, da die Transformationsmatrizen und damit die Drehwinkel aus dem Bild ermittelt werden. Aber die dazu notwendige Berechnung der Kovarianzmatrix, der Eigenwerte und der Eigenvektoren ist für eine schnelle Bilddatencodierung zu aufwendig.

Die Fouriertransformation liefert ein anschaulich interpretierbares Spektrum eines Bildes, das bei anderen Anwendungen gut zu verwenden ist. Ein wesentlicher Nachteil für die Bilddatencodierung sind die komplexen Transformationskoeffizienten. Ein weiterer Nachteil ist die schlechte Konvergenz bei der Rekonstruktion des Bildes, wenn hochfrequente Transformationskoeffizienten weggelassen werden. Dies erklärt sich aus der periodischen Fortsetzung des Eingabebildes, die bei der Fouriertransformation angenommen werden muss.

Wenn die Eingabefunktion reell und symmetrisch ist, sind die Koeffizienten der Fouriertransformation reell. Sie resultieren aus den Cosinus-Anteilen der Basisfunktionen. Man erhält somit die *Cosinustransformation*, wenn man die Eingabefunktion reell wählt und sie symmetrisch fortsetzt. Theoretische und praktische Untersuchungen haben gezeigt, dass sie für die Bilddatencodierung gut geeignet ist.

Die Cosinustransformation lautet:

$$f(u,v) = \frac{2}{L} c(u)c(v) \sum_{x=0}^{L-1} \sum_{y=0}^{L-1} s(x,y) \cos\Big(\frac{\pi(2x+1)}{2L}u\Big) \cos\Big(\frac{\pi(2y+1)}{2L}v\Big) \tag{21.45}$$

$$s(x,y) = \frac{2}{L} \sum_{u=0}^{L-1} \sum_{v=0}^{L-1} c(u)c(v)f(u,v) \cos\Big(\frac{\pi(2x+1)}{2L}u\Big) \cos\Big(\frac{\pi(2y+1)}{2L}v\Big), \tag{21.46}$$

wobei $c(0) = \frac{1}{\sqrt{2}}$ und $c(k) = 1, k = 1, 2, ..., L-1$.

In der Bilddatencodierung wird die Cosinustransformation z.B. im JPEG-Verfahren eingesetzt. Hier werden jeweils Blöcke von $8 \cdot 8$ Bildpunkten transformiert. Die Transformationskoeffizienten der Cosinustransformation werden dann quantisiert und mit *run length Codierung* (Abschnitt 34.2) weiter verdichtet. Wenn man die Cosinustransformation auf $8 \cdot 8$-Blöcke beschränkt, ergeben sich aus (21.45) und (21.46) folgende Spezialisierungen:

$$f(u,v) = \frac{c(u)c(v)}{4} \sum_{x=0}^{7} \sum_{y=0}^{7} s(x,y) \cos\Big(\frac{\pi(2x+1)}{16}u\Big) \cos\Big(\frac{\pi(2y+1)}{16}v\Big) \tag{21.47}$$

$$s(x,y) = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} c(u)c(v)f(u,v) \cos\Big(\frac{\pi(2x+1)}{16}u\Big) \cos\Big(\frac{\pi(2y+1)}{16}v\Big), \tag{21.48}$$

wobei wie oben $c(0) = \frac{1}{\sqrt{2}}$ und $c(k) = 1, k = 1, 2, ..., 7$.

Die Bilder 21.7 zeigen Beispiele zur Cosinustransformation. Bild 21.7-a ist das Original ($512 \cdot 512$ Bildpunkte). Bild 21.7-b zeigt die als Grauwerte codierten Transformationskoeffizienten, wenn die Cosinustransformation über das gesamte Bild erstreckt wird (ein $512 \cdot 512$ Block). Bei den beiden folgenden Teilbildern 21.7-c und 21.7-d wurden Blöcke der Größe $32 \cdot 32$ und $8 \cdot 8$ verwendet.

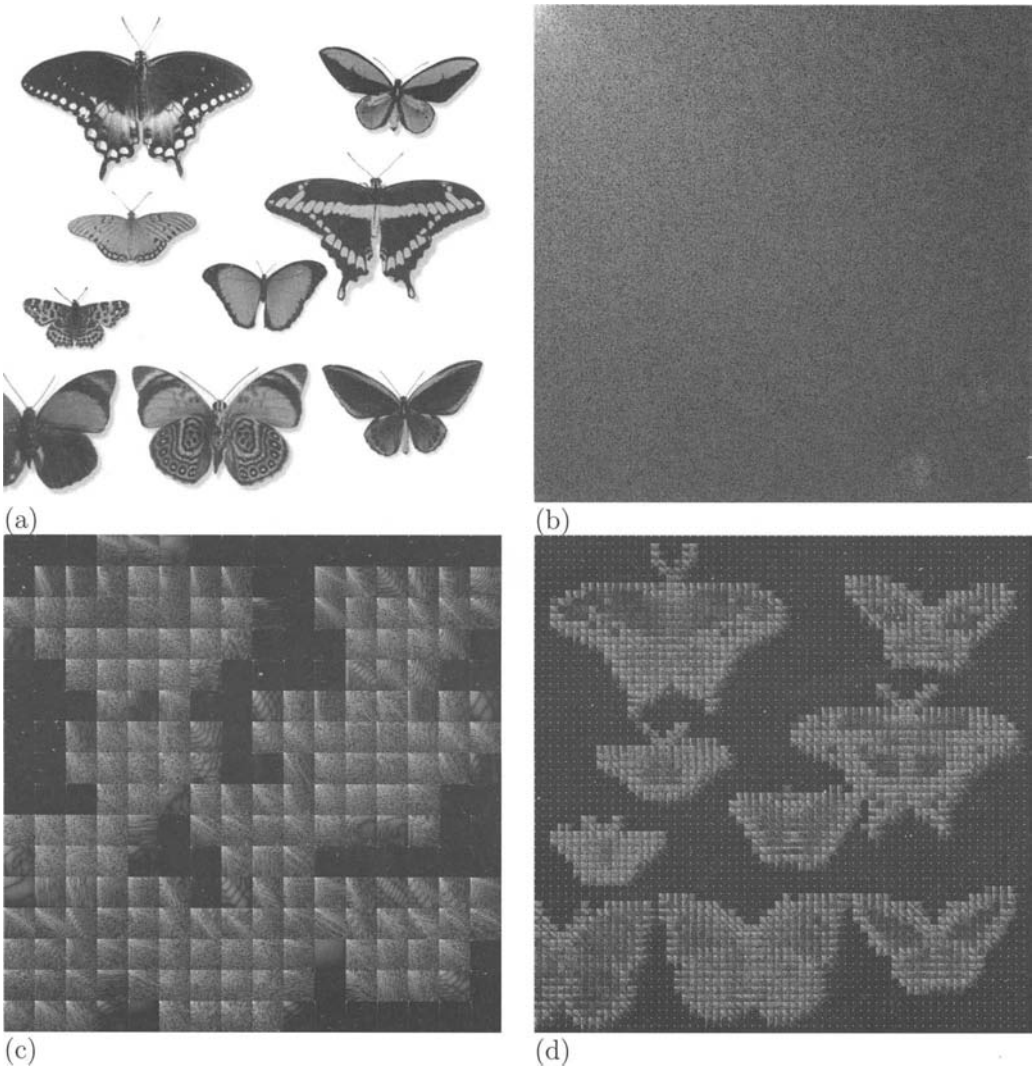


Bild 21.7: Beispiele zur Cosinustransformation. (a) Original. (b) Bildliche Darstellung des Spektrums. Die Cosinustransformation wurde über das gesamte Bild erstreckt (ein $512 \cdot 512$ Block). (c) Bildliche Darstellung des Spektrums. Die Cosinustransformation wurde über $32 \cdot 32$ Blöcke erstreckt. (d) Bildliche Darstellung des Spektrums. Die Cosinustransformation wurde über $8 \cdot 8$ Blöcke durchgeführt. Das entspricht der Vorgehensweise beim JPEG-Verfahren.

Kapitel 22

Modifikation der Ortskoordinaten

22.1 Anwendungen

Geometrische Transformationen werden bei vielen Problemkreisen der digitalen Bildverarbeitung und Mustererkennung benötigt. Einfache Operationen sind das Vergrößern, das Verkleinern, das Verschieben oder das Drehen von Bildern. Auch Maßstabsänderungen, evtl. unterschiedlich in Zeilen- und in Spaltenrichtung, sind möglich. Etwas allgemeiner sind die affinen Transformationen von Bilddaten.

Eine klassische Anwendung ist die geometrische Entzerrung von Luft- und Satellitenbildern, so dass sie mit Kartenkoordinatensystemen (z.B. UTM oder Gauß-Krüger) deckungsgleich sind, um so z.B. thematische Karten erzeugen zu können. Ein anderes Beispiel in dieser Richtung ist die geometrische Angleichung von zwei Bildern des gleichen Beobachtungsgebiets, die jedoch zu unterschiedlichen Zeitpunkten und damit von unterschiedlichen Standorten aufgezeichnet wurden. Als ähnliche Anwendung ist die Elimination von Bildverzerrungen, die durch den Einfluss der verwendeten Aufnahmetechnik bedingt sind (Optik, Abtastart), zu erwähnen.

Bei anderen Anwendungen werden Bildausschnitte gezielt geometrisch verändert und so an andere Bildinhalte angeglichen. Diese Techniken werden auch als *morphing* bezeichnet.

22.2 Grundlegende Problemstellung

Bei allen bisher besprochenen Verfahren wurde die Anordnung der Bildpunkte in der Bildmatrix $\mathbf{S}$ nicht geändert, d.h. die Ortskoordinaten (x, y) der Bildpunkte wurden von den Operationen nicht betroffen. Bei vielen Bildern besteht aber die Notwendigkeit, sie vor weiteren Verarbeitungsschritten einer Transformation der Ortskoordinaten zu unterziehen. Das ist z.B. der Fall, wenn zwei Bilder desselben Objekts, die jedoch unter verschiedenen Aufnahmebedingungen gemacht wurden, zueinander in Beziehung gesetzt werden sollen. Zur Gewährleistung der Deckungsgleichheit ist hier dann in der Regel eine Anpassung der Ortskoordinaten der beiden Bilder notwendig.

Die grundsätzliche Problemstellung kann wie folgt beschrieben werden: Es sei $\mathbf{S}_e =$

$(s_e(x_e, y_e))$ die Bildmatrix des Originalbildes mit den Ortskoordinaten x_e und y_e. Eine Transformation der Ortskoordinaten $\mathbf{S}_e \rightarrow \mathbf{S}_a$ wird beschrieben durch

$$s_a(x_a, y_a) = s_e(x_e, y_e), \tag{22.1}$$

wobei

$$x_a = f_1(x_e, y_e) \text{ und } y_a = f_2(x_e, y_e) \tag{22.2}$$

die *Abbildungen der Transformation* (*mapping functions*) sind. Eine Transformation dieser Art bewirkt also, dass der Grauwert $g = s_e(x_e, y_e)$ des Originalbildes $\mathbf{S}_e$ (des verzerrten Bildes) im transformierten (entzerrten) Bild $\mathbf{S}_a$ in der Position mit den Ortskoordinaten x_a und y_a erscheint. Je nach Art der Transformationsfunktionen ergeben sich verschiedene Arten der Modifikation der Ortskoordinaten.

22.3 Vergrößerung, Verkleinerung

Bei der *Vergrößerung* wird das Originalbild $\mathbf{S}_e$ mit L_e Bildzeilen und R_e Bildspalten in ein Bild $\mathbf{S}_a$ übergeführt, das $L_a > L_e$ und $R_a > R_e$ Bildzeilen bzw. Bildspalten besitzt. Ist der Vergrößerungsmaßstab in beiden Richtungen gleich, so haben die Funktionen der Transformation die einfache Form

$$f_1(x_e, y_e) = ax_e \text{ und } f_2(x_e, y_e) = ay_e. \tag{22.3}$$

Dabei ist $a > 1$ der Vergrößerungsfaktor. Durch Einsetzen der Beziehung (22.3) in (22.1) sieht man den Zusammenhang:

$$s_a(ax_e, ay_e) = s_e(x_e, y_e). \tag{22.4}$$

Mit dieser Zuordnungsvorschrift kann die Bildmatrix $\mathbf{S}_a$ gefüllt werden, etwa durch:

```
FOR x_e:=0 TO L_e-1
FOR y_e:=0 TO R_e-1
BEGIN
 x_a:= a*x_e;
 y_a:= a*y_e;
 s_a(x_a,y_a) := s_e(x_e,y_e)
END;
```

Diese Vorgehensweise bezeichnet man als *direkte Methode*: Ausgehend von den Koordinaten des verzerrten Bildes wird die Position des Grauwertes im entzerrten Bild berechnet.

Die Umrechnung der Bildkoordinaten mit der direkten Methode hat in dieser Form allerdings den Nachteil, dass bestimmte Positionen in $\mathbf{S}_a$ nicht besetzt werden, so bei $a = 3$ z.B. die Positionen

$$(x_a, y_a) = (k, l) \text{ und } (x_a, y_a) = (k+1, l+1) \text{ mit } k, l = 1, 4, 7, 10, ...$$

Dieser Nachteil wird vermieden, wenn der Laufindex in Zeilen und Spaltenrichtung durch eine Größe `delta` inkrementiert wird:

```
delta := 1/a;
x_e  := 0;
WHILE x_e < L_e DO
BEGIN
 y_e := 0;
 WHILE y_e < R_e DO
 BEGIN
  x_a:= TRUNC(a*x_e);
  y_a:= TRUNC(a*y_e);
  i := TRUNC(x_e);
  j := TRUNC(y_e);
  s_a(x_a,y_a) := s_e(i,j);
  y_e := y_e + delta
 END;
 x_e := x_e + delta
END;
```

Bei der *indirekten Methode* wird der umgekehrte Weg eingeschlagen. Zu jeder Position des entzerrten Bildes wird im verzerrten Bild der zugehörige Grauwert bestimmt. Dazu werden allerdings die Umkehrfunktionen $f_1^{-1} = f_2^{-1} = 1/a$ benötigt, die auch überall definiert sind, da $a > 1$ vorausgesetzt wurde. Der Algorithmus lautet dann:

```
FOR x_a:=0 TO a*(L_e-1)
FOR y_a:=0 TO a*(R_e-1)
BEGIN
 x_e := ROUND(x_a/a);
 y_e := ROUND(y_a/a);
 s_a(x_a,y_a) := s_e(x_e,y_e)
END;
```

Bei diesen Koordinatenberechnungen tritt der Fall ein, dass die berechneten Werte x_a/a und y_a/a nicht ganzzahlig sind und somit keine Bildpunktposition im verzerrten oder entzerrten Bild bezeichnen. Es ist dann die Frage, welche Position gewählt und wie der zugehörige Grauwert berechnet werden soll. Diese Fragestellungen werden mit dem Schlagwort *resampling* bezeichnet.

Im obigen Algorithmus zur indirekten Methode wird das Problem durch Rundung gelöst, d.h. es wird in der Bildmatrix diejenige Koordinatenposition gewählt, die den Werten x_a/a und y_a/a am nächsten liegt (*nearest neighbor resampling*).

Eine andere Möglichkeit ist die *bilineare Interpolation*, bei der der Grauwert als gewichteter Mittelwert der vier benachbarten Bildpunkte berechnet wird:

$$\begin{aligned} s_a(x_a, y_a) &= a_1 \cdot s\Big(\text{int}(x_e), \text{int}(y_e)\Big) + a_2 \cdot s\Big(\text{int}(x_e), \text{int}(y_e) + 1\Big) + \\ &\quad a_3 \cdot s\Big(\text{int}(x_e) + 1, \text{int}(y_e)\Big) + a_4 \cdot s\Big(\text{int}(x_e) + 1, \text{int}(y_e) + 1\Big). \end{aligned} \tag{22.5}$$

Die Parameter a_i können dabei so gewählt werden, dass sie die Lage der berechneten Position bezüglich der vier Nachbarpunkte berücksichtigen.

Da (22.5) eine Mittelung von Grauwerten beinhaltet, hat die bilineare Interpolation auch grauwertglättende Eigenschaften (vergleichbar mit dem bewegten Mittelwert oder einer Tiefpassfilterung, Kapitel 18). Durch diese Mittelung werden neue Grauwerte erzeugt, die womöglich im Originalbild nicht auftreten. Das begrenzt die Verwendungsmöglichkeit des Verfahrens: Bei logischen Bildern, bei denen die Grauwerte Codes für bestimmte Eigenschaften sind, kann die bilineare Interpolation nicht eingesetzt werden. Im Vergleich mit der nächster-Nachbar-Methode, die z.B. schräg verlaufende Grauwertkanten im entzerrten Bild oft stufig erscheinen lässt, werden bei der bilinearen Interpolation die Grauwertübergänge, bedingt durch das Tiefpassverhalten, etwas gedämpft. Je nach Anwendungsfall mag dieser Effekt mehr oder weniger wünschenswert sein.

Eine weitere Methode zum *resampling* ist die *kubische Faltung* (*cubic convolution*). Sie benutzt eine Umgebung von insgesamt 16 Bildpunkten, die, mit entsprechenden Gewichten versehen, die Berechnung des Grauwertes beeinflussen:

$$\begin{aligned} &i = \text{int}(x_a/a) \text{ und } j = \text{int}(y_a/a) \\ &s_a(x_a, y_a) = \sum_{m=-1}^{2} \sum_{n=-1}^{2} a_{mn} \cdot s_e(i+m, j+n). \end{aligned} \tag{22.6}$$

Die kubische Faltung, bei der durch entsprechende Wahl der Parameter der Tiefpasseffekt vermieden werden kann, liefert im Vergleich mit den beiden anderen Verfahren die besten Ergebnisse. Allerdings ist auch der erhöhte Rechenaufwand zu berücksichtigen.

Die Problemstellung des *resampling* tritt bei allen folgenden Umrechnungen der Ortskoordinaten auf. Es wird jedoch im Weiteren nicht mehr näher darauf eingegangen. Auch ob die direkte oder die indirekte Methode verwendet wird, ist im Folgenden ohne Bedeutung.

Die *Verkleinerung* kann mit der Beziehung (22.3) ausgedrückt werden, wenn für den Parameter $0 < a < 1$ zugelassen wird. Unterschiedliche Maßstabsfaktoren in Zeilen- und Spaltenrichtung können durch die folgende Wahl der Funktionen erreicht werden:

$$f_1(x_e, y_e) = a \cdot x_e \text{ und } f_2(x_e, y_e) = b \cdot y_e \text{ mit } a, b > 0. \tag{22.7}$$

22.4 Affine Abbildungen

Bei der allgemeinen affinen Transformation der Ortskoordinaten sind die Funktionen lineare Polynome in x_e und y_e:

$$\begin{aligned} x_a &= f_1(x_e, y_e) = a_0 + a_1 x_e + a_2 y_e, \\ y_a &= f_2(x_e, y_e) = b_0 + b_1 x_e + b_2 y_e. \end{aligned} \tag{22.8}$$

Die in Abschnitt 22.3 besprochene Vergrößerung und Verkleinerung sind natürlich Spezialfälle von (22.8). In Matrizenschreibweise lautet (22.8):

$$\mathbf{x}_a = \mathbf{A}\mathbf{x}_e + \mathbf{v} \text{ oder} \tag{22.9}$$
$$\begin{pmatrix} x_a \\ y_a \end{pmatrix} = \begin{pmatrix} a_1 & a_2 \\ b_1 & b_2 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \end{pmatrix} + \begin{pmatrix} a_0 \\ b_0 \end{pmatrix}$$

Durch entsprechende Wahl der Parameter a_i und b_i erhält man verschiedene Transformationen der Ortskoordinaten.

Bei einer *Verschiebung* (*Translation*) ist die Matrix $\mathbf{A}$ die Einheitsmatrix. Der Verschiebungsvektor $\mathbf{v}$ gibt die Verschiebungsanteile in x_e- und y_e-Richtung an:

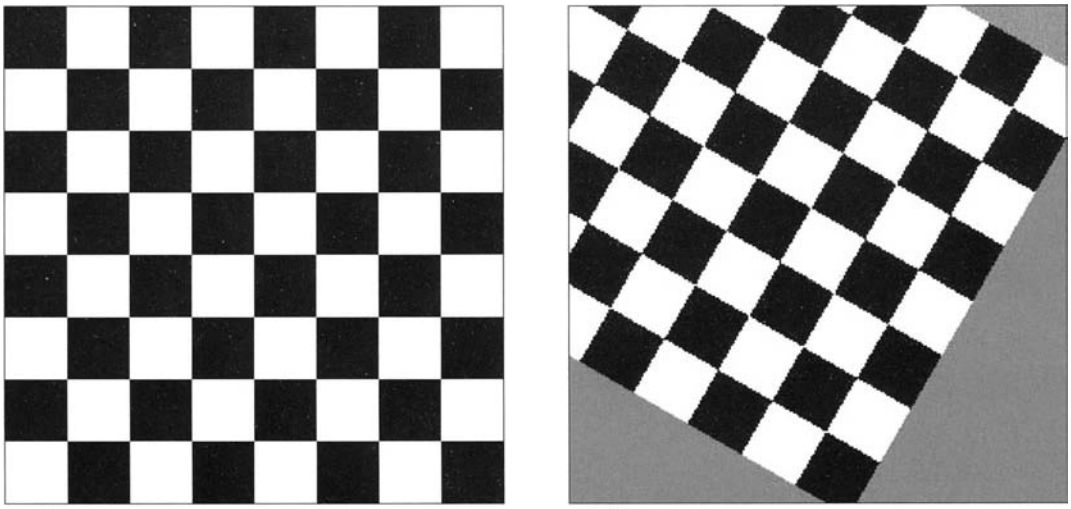
$$\begin{aligned} x_a &= x_e + a_0, \\ y_a &= y_e + b_0. \end{aligned} \tag{22.10}$$

Drehungen (*Rotationen*) können ebenfalls durch (22.8) dargestellt werden. Die Matrix $\mathbf{A}$ hat bei einer Drehung um den Winkel α die Gestalt:

$$\mathbf{A} = \begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}. \tag{22.11}$$

Ist mit der Drehung auch eine Verschiebung verbunden, so drückt der Verschiebungsvektor $\mathbf{v}$ die Anteile in x_e- und y_e-Richtung aus. Da die Bildmatrix $\mathbf{S}_a$ des gedrehten Bildes wieder eine rechteckige Zahlenanordnung ist, muss unter Umständen ein spezieller Grauwert als Hintergrund festgelegt werden, mit dem diejenigen Bildpunkte codiert werden, die zwar in der Bildmatrix $\mathbf{S}_a$ liegen, aber nicht zum gedrehten Bild gehören (Bild 22.1).

Bei einer allgemeinen Affintransformation werden an die Parameter in (22.8) keine weiteren Forderungen gestellt. Es muss lediglich gewährleistet sein, dass die Determinante der Matrix $\mathbf{A}$ ungleich 0 ist. Es können dann geometrische Transformationen wie *Scherung* oder *Spiegelung* erfasst werden. Auf die Bestimmung der Parameter a_i und b_i wird in den folgenden Abschnitten noch näher eingegangen.



(a) (b)

Bild 22.1: Koordinatentransformation eines Bildes. (a) Originalbild. (b) Transformiertes Bild (Drehung und Verschiebung) mit Hintergrundgrauwert.

22.5 Interpolation mit Polynomen

22.5.1 Polynome

Sind die Funktionen der geometrischen Transformation $f_1(x_e, y_e)$ und $f_2(x_e, y_e)$ nicht weiter bekannt, so werden Funktionen verwendet, die den tatsächlichen Sachverhalt so gut wie möglich annähern sollen (*Modellbildung*). In der Praxis werden dazu oft Polynome n-ten Grades in den Unbekannten x_e und y_e verwendet:

$$\begin{aligned} x_a &= f_1(x_e, y_e) = a_0 + a_1 x_e + a_2 y_e + a_3 x_e^2 + a_4 x_e y_e + a_5 y_e^2 + ..., \\ y_a &= f_2(x_e, y_e) = b_0 + b_1 x_e + b_2 y_e + b_3 x_e^2 + b_4 x_e y_e + b_5 y_e^2 + ... \end{aligned} \tag{22.12}$$

Ein Sonderfall von (22.12) sind die affinen Abbildungen von Abschnitt 22.4. Auch Polynome zweiten Grades werden in der Praxis oft verwendet. Es kann auch sinnvoll sein, in Zeilen- und in Spaltenrichtung Polynome unterschiedlichen Grades zu verwenden, wenn der jeweilige Anwendungsfall dies rechtfertigt.

Es stellt sich nun die Frage, wie die Parameter a_i und b_i der Transformationspolynome ermittelt werden können. Eine Möglichkeit besteht darin, einen Lösungsansatz über sogenannte *Systemkorrekturen* zu wählen. Hier liegen aufgrund der Aufzeichnungsart eines Bildes Informationen über die geometrischen Verzerrungen, also über die Funktionen der Transformation und deren Parameter vor. Durch einen mathematischen Ansatz können

dann die unbekannten Parameter abgeleitet werden. Als Beispiel hierzu kann ein Bild dienen, das durch eine projektive Abbildung entstanden ist. Der Ansatz mit linearen Polynomen ist dann exakt, und die unbekannten Parameter können z.B. aus der Position des Aufnahmegerätes berechnet werden. Ein weiteres Beispiel ist ein Bild, das mit einem Weitwinkelobjektiv aufgenommen wurde. Liegen vom Hersteller Angaben (Tabellen oder funktionaler Zusammenhang) über die Aufnahmeeigenschaften des Objektivs vor, so kann auch hier, in gewissen Grenzen, eine Systemkorrektur durchgeführt werden.

In allen Fällen, in denen die Parameter nicht über einen Ansatz zur Systemkorrektur berechnet werden können, wird die *Passpunktmethode* verwendet. Zur Erläuterung dieser Methode werden folgende Annahmen gemacht (Bild 22.2):

$\mathbf{S}_e = (s_e(x_e, y_e))$	sei das verzerrte Bild.
$\mathbf{R} = (r(u, v))$	sei ein Referenzbild, an das das verzerrte Bild angeglichen werden soll. Über den funktionalen Zusammenhang der Verzerrung sei weiter nichts bekannt. Es wird jedoch angenommen, dass er durch Polynome n-ten Grades approximiert werden kann.
$\mathbf{S}_a = (s_a(x_a, y_a))$	sei das entzerrte Bild, das dann deckungsgleich mit dem Referenzbild $\mathbf{R}$ ist.

Vereinfachend wird ein linearer Ansatz gemäß (22.8) gewählt. Es sind dann die sechs unbekannten Parameter a_i und b_i mit $i = 0, 1, 2$ zu bestimmen. Zur Ermittlung dieser Parameter sind ingesamt drei *Passpunkte* notwendig. Passpunkte sind dabei einander entsprechende Punkte, die sowohl im Referenzbild $\mathbf{R}$ als auch im verzerrten Bild $\mathbf{S}$ eindeutig ermittelt werden können. Die drei Passpunkte liefern also drei einander entsprechende Koordinatenpaare gemäß:

$$i\text{-ter Passpunkt: } (u_i, v_i) \Longleftrightarrow (x_{e,i}, y_{e,i}), i = 1, 2, 3.$$

Diese Koordinatenpaare werden in die Funktionen (22.8) eingesetzt und liefern zwei lineare Gleichungssysteme:

$$\begin{aligned} u_i &= a_0 + a_1 x_{e,i} + a_2 y_{e,i} \\ v_i &= b_0 + b_1 x_{e,i} + b_2 y_{e,i}, \quad i = 1, 2, 3. \end{aligned} \tag{22.13}$$

In Matrixschreibweise:

$$\begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} 1 & x_{e,1} & y_{e,1} \\ 1 & x_{e,2} & y_{e,2} \\ 1 & x_{e,3} & y_{e,3} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix} \text{ und } \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} 1 & x_{e,1} & y_{e,1} \\ 1 & x_{e,2} & y_{e,2} \\ 1 & x_{e,3} & y_{e,3} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \end{pmatrix}.$$

Aus diesen Gleichungssystemen können die sechs unbekannten Polynomparameter berechnet und nach Maßgabe der drei Passpunkte kann das verzerrte Bild $\mathbf{S}_a = (s_a(x_a, y_a))$ aufgebaut werden.

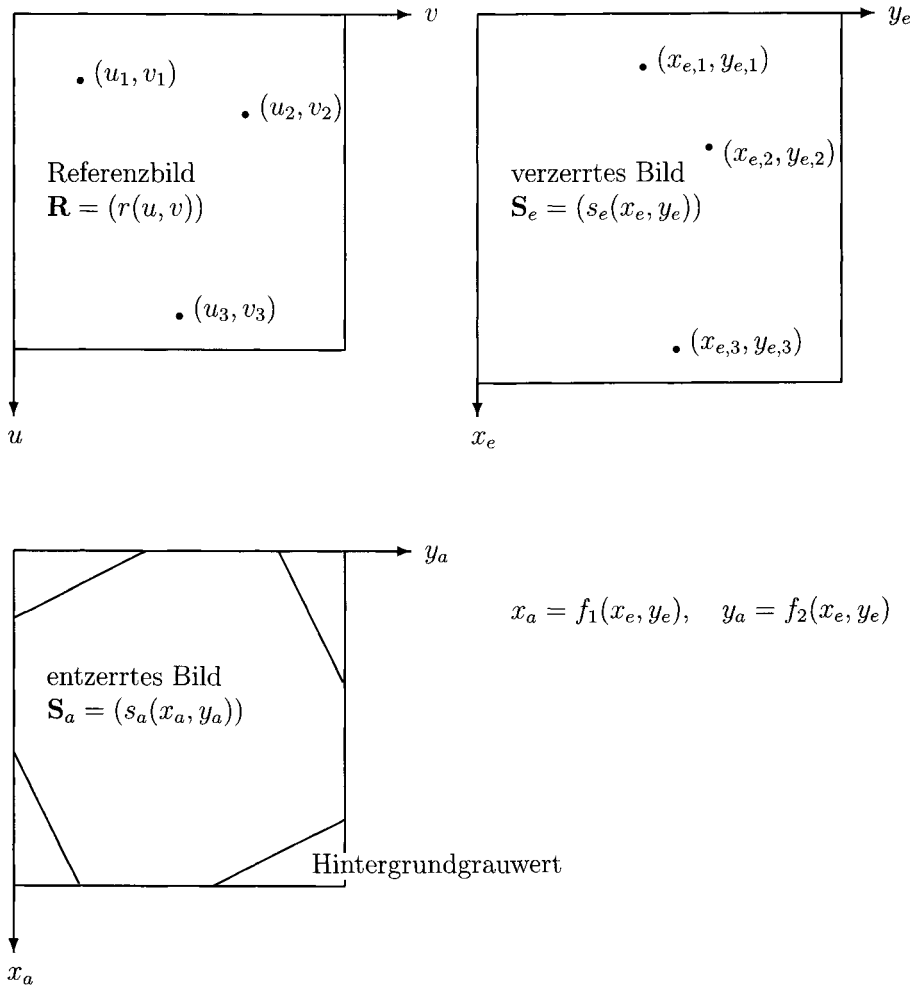


Bild 22.2: Bestimmung der Parameter der Transformationsfunktionen mit der Passpunktmethode. Im Referenzbild $\mathbf{R}$ und im verzerrten Bild $\mathbf{S}_e$ werden die Koordinaten von einander entsprechenden Bildpunkten ermittelt. Anhand dieser Koordinaten werden die unbekannten Parameter der Funktionen der Transformation berechnet.

Die Passpunktmethode kann sinngemäß auf Polynome mit einem höheren Grad oder allgemeinere Abbildungsgleichungen angewendet werden. Bei Polynomen zweiten Grades (12 unbekannte Parameter) sind sechs Passpunkte und bei Polynomen dritten Grades (20 unbekannte Parameter) sind zehn Passpunkte erforderlich.

Bei der Vermessung der Passpunkte müssen in der Praxis immer Messfehler in Kauf genommen werden. Das hat aber zur Folge, dass diese Messfehler in die Berechnung der Parameter eingehen und somit auch die Güte der Entzerrung beeinflussen. So ist es sinnvoll, mehr Passpunkte als für das jeweilige Entzerrungsmodell notwendig sind zu ermitteln und die Parameter so zu bestimmen, dass sie nach der *Methode der kleinsten Quadrate* (*Ausgleichsrechnung*) die Messfehler minimieren. Dies wird im nächsten Abschnitt erläutert.

22.5.2 Ausgleichsrechnung

Zur Berechnung der Funktionen bei einer geometrischen Entzerrung wird die *Ausgleichsrechnung* verwendet, um die Parameter so zu bestimmen, dass die Funktionen eine vorgegebene Fehlerfunktion minimieren. Im Folgenden wird diese Methode am Beispiel von linearen Polynomen dargestellt. Die sinngemäße Übertragung auf andere Funktionen ist möglich.

Es wird vorausgesetzt, dass $p > 3$ Passpunkte vorliegen:

$$(u_i, v_i) \Longleftrightarrow (x_{e,i}, y_{e,i}), \quad i = 1, 2, 3, ..., p$$

Die linearen Gleichungsysteme für die unbekannten Parameter a_i und b_i lauten:

$$\begin{aligned} u_i &= a_0 + a_1 x_{e,i} + a_2 y_{e,i} \\ v_i &= b_0 + b_1 x_{e,i} + b_2 y_{e,i}, \quad i = 1, 2, ..., p \end{aligned} \tag{22.14}$$

In Matrixschreibweise:

$$\mathbf{u} = \mathbf{Pa} \text{ und } \mathbf{v} = \mathbf{Pb}.$$

Dabei sind $\mathbf{u}$ und $\mathbf{v}$ Vektoren der Dimension p, $\mathbf{P}$ eine Matrix mit drei Spalten und p Zeilen und $\mathbf{a}$ und $\mathbf{b}$ Vektoren der Dimension drei (die unbekannten Parameter der Polynome).

Die lineare Ausgleichung wird so durchgeführt, dass die Vektoren $\mathbf{a}$ und $\mathbf{b}$ die folgenden Fehlerfunktionen minimieren:

$$\begin{aligned} &\sum_{i=1}^{p} \Big(u_i - (a_0 + a_1 x_{e,i} + a_2 y_{e,i})\Big)^2 \rightarrow min_a \\ &\sum_{i=1}^{p} \Big(v_i - (b_0 + b_1 x_{e,i} + b_2 y_{e,i})\Big)^2 \rightarrow min_b \end{aligned} \tag{22.15}$$

oder in Matrixschreibweise:

$$(\mathbf{u} - \mathbf{Pa})^T(\mathbf{u} - \mathbf{Pa}) \rightarrow min_a \text{ und } (\mathbf{v} - \mathbf{Pb})^T(\mathbf{v} - \mathbf{Pb}) \rightarrow min_b. \tag{22.16}$$

Da die Funktionen (22.15) stetige partielle Ableitungen nach den a_i und b_i besitzen, lässt sich sofort eine notwendige Bedingung für die Minmimierung angeben:

$$\begin{aligned} \frac{\partial \sum_{i=1}^{p} \left(u_i - (a_0 + a_1 x_{e,i} + a_2 y_{e,i})\right)^2}{\partial a_j} &= 0, \quad j = 0,1,2 \\ \frac{\partial \sum_{i=1}^{p} \left(v_i - (b_0 + b_1 x_{e,i} + b_2 y_{e,i})\right)^2}{\partial b_j} &= 0, \quad j = 0,1,2 \end{aligned} \tag{22.17}$$

Die Bedingungen (22.17) heißen die *Normalgleichungen*, die bei der linearen Ausgleichung auch durch die folgenden linearen Gleichungssysteme für **a** und **b** ausgedrückt werden können:

$$\mathbf{P}^T\mathbf{Pa} = \mathbf{Pu} \text{ und } \mathbf{P}^T\mathbf{Pb} = \mathbf{Pv}. \tag{22.18}$$

Die Lösungen dieser linearen Gleichungssysteme sind die gesuchten Parameter **a** und **b** der Polynome. Die direkte Lösung der Normalgleichungen (22.18) ist möglicherweise numerisch instabil. Aus diesem Grund werden in der Praxis auch andere Methoden verwendet, so z.B. das *Householder-Verfahren* ([Enge96]).

Im Weiteren ist nun zu untersuchen, wie die Qualität des funktionalen Ansatzes und der Ausgleichung beurteilt werden können.

22.5.3 Beurteilung der Qualität

Die Beurteilung der Qualität sollte bei einer geometrischen Entzerrung immer durchgeführt werden, denn sie gibt Aufschluss, ob der funktionale Ansatz mit den gewählten Funktionen (im vorliegenden Beispiel Polynome ersten Grades) gerechtfertigt ist und wie gut die Funktionen an die gegebenen Passpunkte angepasst wurden. Da bei der Vermessung der Passpunkte manchmal auch größere Fehler auftreten können (z.B. Tippfehler bei den Koordinatenwerten), ist durch die Beurteilung auch eine Möglichkeit gegeben, fehlerhafte Passpunkte zu entdecken.

Zur Beurteilung können mehrere Möglichkeiten kombiniert werden. Als Erstes sind die in (22.15) durch die Ausgleichung erzielten Minima min_a und min_b natürlich Beurteilungsgrößen. Hier empfiehlt es sich, noch eine Normierung der beiden Minima durch die Anzahl p der Passpunkte durchzuführen:

$$min_{a,normiert} = \frac{min_a}{p} \quad \text{und} \quad min_{b,normiert} = \frac{min_b}{p}. \tag{22.19}$$

Besser noch als (22.19) ist eine Normierung über die Überbestimmtheit der Gleichungssysteme. Es sei m die Anzahl der Parameter der Funktionen (im vorliegenden Beispiel: $m = 3$). Als Größen zur Beurteilung der Güte kann dann verwendet werden:

$$min_{a,normiert} = \frac{min_a}{p-m} \quad \text{und} \quad min_{b,normiert} = \frac{min_b}{p-m}. \tag{22.20}$$

Dabei muss natürlich die Anzahl p der Passpunkte größer sein als die Anzahl m der auftretenden Parameter. Für $p = m$ wäre das Fehlermaß (22.20) unendlich groß, was auch plausibel ist, da ja im Falle der exakten Bestimmung der Parameter der Funktionen keine Aussagen über ihre Güte gemacht werden können.

Eine weitere Möglichkeit, die auch gut zur grafischen Darstellung geeignet ist, ist die Berechnung von *Restfehlervektoren*. Dazu werden, wie oben beschrieben, die Parameter der Funktionen ermittelt. Dann werden die Koordinaten der Passpunkte in die Transformationsgleichungen eingesetzt und zu jedem Passpunkt (u_i, v_i) im Referenzbild die Koordinaten (u_i', v_i') berechnet. Die Differenz der gemessenen und der berechneten Koordinaten $(u_i - u_i', v_i - v_i')$ wird als Restfehlervektor des i-ten Passpunktes bezeichnet. Zur grafischen Darstellung werden die Restfehlervektoren in ein (u, v)-Koordinatensystem eingetragen (Bild 22.3). Da die Abweichungen oft nur sehr gering sind, empfiehlt es sich, sie um einen geeigneten Maßstabsfaktor c vergrößert darzustellen.

Sind die Richtungen der Restfehlervektoren mehr oder weniger gleichverteilt und die Beträge klein, so kann angenommen werden, dass bei der Vermessung keine groben Fehler gemacht wurden und das gewählte Modell den tatsächlichen funktionalen Zusammenhang zwischen Referenzbild und verzerrtem Bild gut approximiert. Treten in bestimmten Bildbereichen Vorzugsrichtungen auf, so ist das ein Hinweis, dass in diesem Bereich bei der Bestimmung der Passpunkte Fehler gemacht wurden. Allerdings würde sich ein systematischer Fehler, der bei der Vermessung aller Passpunkte in gleicher Weise zum Tragen kam, in der grafischen Darstellung der Restfehlervektoren nicht unbedingt ausdrücken. Um Passpunkte zu entdecken, die, wie oben angedeutet, durch gröbere Koordinatenfehler gestört sind, wird der Betrag der Restfehlervektoren verwendet:

$$\sqrt{(u_i - u_i')^2 + (v_i - v_i')^2} \tag{22.21}$$

Ist der Betrag größer als ein bestimmter Schwellwert c, so wird der zugehörige Passpunkt als möglicherweise falsch vermessen markiert und kann dann noch einmal kontrolliert werden. Der Schwellwert c kann dabei nach Maßgabe der mittleren quadratischen Abweichung der Beträge der Restfehlervektoren aller Passpunkte festgelegt werden.

Eine weitere Variante zur Beurteilung besteht darin, dass zur Berechnung der Restfehlervektoren nicht die Passpunkte verwendet werden, mit denen die Parameter berechnet wurden, sondern eigens dafür vermessene Kontrollpunkte. Allerdings können dann fehlerhafte Passpunkte nicht automatisch erkannt werden. So ist es am besten, wenn alle angebotenen Möglichkeiten kombiniert werden, um ein Höchstmaß an Informationen zur Beurteilung der Güte zu erhalten.

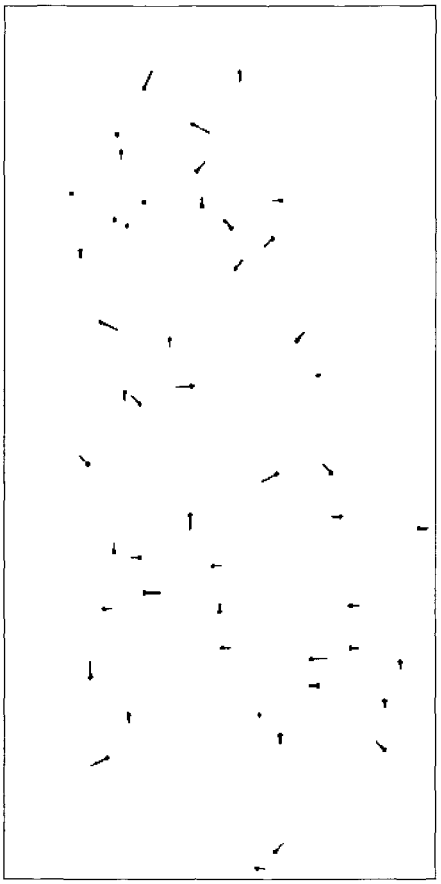


Bild 22.3: Beurteilung der Qualität der Ausgleichung bei der Berechnung der Parameter der Transformationsfunktionen. In einem (u, v)-Koordinatensystem sind die Restfehlervektoren, die sich aus der Differenz der gemessenen und der berechneten Passpunktkoordinaten ergeben, grafisch dargestellt, wobei sie für die Wiedergabe vergrößert wurden.

22.5.4 Vermessung der Passpunkte

Die Vermessung der Koordinaten der Passpunkte kann auf unterschiedliche Weise durchgeführt werden. Liegen das Referenzbild und das verzerrte Bild in digitalisierter Form vor, so können die Koordinaten mit Hilfe der interaktiven Möglichkeiten eines Bildverarbeitungssystems ermittelt werden. Die Koordinaten der Passpunkte werden dabei mit Hilfe einer Maus und eines Fadenkreuzes auf dem Bildschirm festgelegt und im (u, v)-Koordinatensystem des Referenzbildes bzw. im (x_e, y_e)-Koordinatensystem des verzerrten Bildes vermessen. Ist das Referenzbild eine Karte, so werden die (u, v)-Koordinaten im Koordinatensystem der jeweiligen Kartenprojektion (z.B. Gauß-Krüger- oder UTM-Gitter) angegeben. Dabei ist ein Digitalisierungstisch hilfreich, auf den die Kartenvorlage aufgespannt ist. Mit Hilfe des dazugehörigen Fadenkreuzes können dann die Koordinaten relativ genau bestimmt werden. In diesem Fall muss noch eine Zuordnung der (u, v)-Koordinaten des Referenzbildes $\mathbf{R}$ zu den Zeilen- und Spaltenkoordinaten (x_a, y_a) des entzerrten Bildes $\mathbf{S}_a$ hergestellt werden. Einfacher gestaltet sich das Ausmessen der (u, v)-Koordinaten, wenn die Karte schon digitalisiert vorliegt.

Eine aufwändigere Technik ist die Passpunktvermessung mit Hilfe einer *Flächenkorrelation*. Dies sei am Beispiel eines digitalisierten Referenzbildes $\mathbf{R} = (r(u, v))$ und eines verzerrten Bildes $\mathbf{S}_e = (s_e(x_e, y_e))$ erläutert. Zunächst wird im Referenzbild ein markanter Punkt bestimmt, der auch im verzerrten Bild erkannt werden kann und somit als Passpunkt in Frage kommt. Die Koordinaten dieses Punktes im Referenzbild seien (u, v). Im nächsten Schritt wird um den Punkt ein Ausschnitt der Größe $m \cdot m, m = 3, 5, 7, \ldots$ mit (u, v) als Mittelpunkt definiert. Aufgrund von apriori-Informationen kann in der Regel die ungefähre Lage $(x_{e,0}, y_{e,0})$ des Punktes im verzerrten Bild berechnet oder interaktiv ermittelt werden. Bei apriori-Information über die Art der Verzerrung kann auch der Ausschnitt im Referenzbild an die Geometrie des verzerrten Bildes angepasst werden. Nun wird ein Maß der Nicht-Übereinstimmung zwischen dem Ausschnitt im Referenzbild und dem entsprechenden Ausschnitt im verzerrten Bild berechnet, etwa:

$$\frac{1}{m^2} \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} \Big(s_e(x_{e,i-n+k}, y_{e,i-n+l}) - h(k, l) \Big)^2, \tag{22.22}$$

wobei $n = (m-1)/2$, $(x_{e,i}, y_{e,i})$ mit anfänglich $i = 0$ die berechnete Position im verzerrten Bild und $(h(u, v))$ der evtl. geometrisch angepasste Ausschnitt aus dem Referenzbild ist. Durch Ausquadrieren von (22.22) erhält man:

$$\begin{aligned} &\frac{1}{m^2} \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} s_e(x_{e,i-n+k}, y_{e,i-n+l})^2 - \\ &\frac{2}{m^2} \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} \Big(s_e(x_{e,i-n+k}, y_{e,i-n+l})) \cdot h(k, l) \Big) + \\ &\frac{1}{m^2} \sum_{k=0}^{m-1} \sum_{l=0}^{m-1} h(k, l)^2. \end{aligned} \tag{22.23}$$

Der mittlere Term in (22.23) ist die Korrelation des Ausschnittes des Referenzbildes $(h(u,v))$ mit dem verzerrten Bild im Bereich der Position $(x_{e,i}, y_{e,i})$.

Das Maß der Nicht-Übereinstimmung ist also klein, wenn die Korrelation in (22.23) groß ist. Soll das Maß von Helligkeit und Kontrast unabhängig sein, so kann (22.23) noch durch eine Division durch den ersten Term normiert werden.

Ausgehend von der Berechnung des Maßes in der Position mit dem Index $i = 0$ im verzerrten Bild wird jetzt eine Suche nach dem Korrelationsmaximum durchgeführt. Die gefundene Position des Korrelationsmaximums, die auch auf Bruchteile von Bildpunkten berechnet werden kann, wird dann als Lage des Passpunktes im verzerrten Bild verwendet.

Diese Methode der Passpunktbestimmung über die Korrelation ist gut bei automatischen oder teilautomatischen Entzerrungssystemen geeignet, bei denen die Passflächen der Referenzbilder (z.B. Satellitenbilddaten) in einer Datenbank verwaltet werden. Bei der Integration eines neuen Bildes können dann die Passpunktkoordinaten mit Hilfe der Datenbank automatisch ermittelt werden.

22.6 Abschließende Bemerkungen zur geometrischen Entzerrung

Die Probleme der Transformation der Ortskoordinaten wurden hier am Beispiel der Verwendung von Polynomen erläutert. In der Praxis werden auch andere Funktionen, wie z.B. zweidimensionale Splinefunktionen oder Flächen zweiter Ordnung verwendet.

Bei vielen praktischen Anwendungen findet im zunehmenden Maß die Verarbeitung von dreidimensionalen Koordinaten Bedeutung. Die Stereoluftbildauswertung in der Fotogrammetrie erlaubt es z.B. mit einem Stereobildpaar mit nur fünf Passpunkten, die vollständigen dreidimensionalen Verhältnisse zu rekonstruieren. Auch für Zeilenabtasteraufnahmen sind Entzerrungsmodelle entwickelt worden, die drei verschiedene Blickwinkel auf das zu entzerrende Gebiet verwenden.

Auch bei der Auswertung von Bildfolgen ist die Berechnung der dreidimensionalen Koordinaten wichtig. Hier wird neben dem Biokularstereo auch das Bewegungsstereo verwendet, bei dem z.B. aus aufeinanderfolgenden Zeitreihenbildern räumliche Informationen berechnet werden.

Kapitel 23

Szenenanalyse

23.1 Einleitung, Beispiele, Merkmale

Die *Szenenanalyse* befasst sich mit der Extraktion von Informationen aus einem Bild oder einer Bildfolge mit automatischen oder halbautomatischen Methoden. Sie kann in die Verarbeitungsschritte der Vorverarbeitung, der Segmentierung und der Interpretation eingeteilt werden.

Bei der *Vorverarbeitung* von digitalisierten Bildern werden Techniken wie Bildverbesserung, geometrische Entzerrung oder digitale Filterung eingesetzt. Auch die Verwendung von speziellen Speicherungstechniken oder die Berechnung von geeigneten Merkmalen zur Bildsegmentierung kann zum Bereich der Bildvorverarbeitung gerechnet werden.

Bei der *Segmentierung* wird der Ortsbereich des Bildes, also der Bereich der Zeilen- und Spaltenkoordinaten x und y, in Bereiche eingeteilt, die nach Maßgabe von *Einheitlichkeitsprädikaten* P zusammengehören. Die resultierenden Bereiche mit einheitlichen Bildeigenschaften werden *Segmente* (oder *Regionen*) genannt. Es wird in der Regel verlangt, dass

- jeder Bildpunkt in genau einer der Regionen erfasst ist,
- das Einheitlichkeitsprädikat, angewendet auf die Bildpunkte der Region, den Wahrheitswert `TRUE` ergibt und
- die Regionen maximal sind, d.h., dass die Hinzunahme von Bildpunkten zu einer Region den Wahrheitswert `FALSE` bewirkt.

Bei der *Interpretation* wird versucht, auf der Basis des segmentierten Bildes einen Zusammenhang zwischen den einzelnen Segmenten abzuleiten, um so letztlich komplexe Objekte erkennen zu können. Dazu einige Beispiele:

In den meisten Anwendungsbereichen der digitalen Bildverarbeitung und Mustererkennung ist es notwendig, aus einem Bild oder einer Bildfolge zusammengehörige Bildstrukturen zu erkennen und diese als Ganzes weiter zu verarbeiten. Beispiele dazu sind in Bild 23.1 zusammengestellt. In der Automobilindustrie werden häufig Teile verklebt. Mit einem

Roboter wird dazu eine Kleberaupe auf eines der zu verklebenden Teile aufgebracht. Eine Aufgabe der optischen Qualitätskontrolle ist es hier zu überprüfen, ob die Kleberaupe vorhanden ist, ob sie unterbrochen ist und ob sie die richtige Form besitzt. Zunächst muss man somit auf dem zu untersuchenden Bauteil, meistens in einer festgelegten *area-of-interest* (Bild 23.1-a), die Kleberaupe, falls sie vorhanden ist, vom Bildhintergrund trennen. Dies kann, nach einer geeigneten Vorverarbeitung, im einfachsten Fall durch eine Binarisierung geschehen (Bild 23.1-b). Wenn die Binarisierung in der *area-of-interest* nicht eine Mindestanzahl an Vordergrundbildpunkten liefert, könnte man entscheiden, dass die Kleberaupe nicht vorhanden ist. Eine weitere Analyse des Bandes der Kleberaupe kann die Information liefern, ob Unterbrechungen vorliegen.

Wenn auch die Form (der Querschnitt) der Kleberaupe überprüft werden soll, so kann man mit einem Laser und einer Spaltlinse ein schmales Lichtband schräg auf die Kleberaupe projizieren und dieses Lichtband mit einem Videosensor von oben aufzeichnen. Die Verformung des Lichtbandes zeigt dann die Form der Kleberaupe. Hier muss also das Lichtband vom Bildhintergrund separiert werden (Bild 23.1-c). Die Berechnung markanter Punkte auf dem Lichtband und deren relative Lage zueinander liefert die Information, ob die Form des Querschnitts der Kleberaupe im zulässigen Bereich liegt.

Dieses einfache Beispiel enthält die grundlegenden Schritte der Segmentierung, bei der eine Bildstruktur (ein *Objekt*) vom Hintergrund zu trennen ist: Es werden Bildpunkte, die gleiche oder ähnliche Eigenschaften besitzen, zu einer Einheit zusammengefasst. Die Einheitlichkeit ist beim Lichtband durch das Merkmal „Helligkeit" gegeben, d.h. alle Bildpunkte, deren Grauwert über einem bestimmten Schwellwert liegt, werden als „zum Lichtband gehörig" gewertet. Die weiteren Auswertungen zur Form des Lichtbandes gehören nicht mehr zum Segmentierungsschritt. Sie können erst dann durchgeführt werden, wenn die Bildpunkte des Lichtbandes gefunden sind.

In diesem Beispiel ist das Objekt (das Lichtband) vom Hintergrund zu trennen. Die Segmentierung wurde hier durch eine Binarisierung durchgeführt. Im Ergebnisbild (23.1-c) sind die weißen Bildpunkte die Kleberaupe und die schwarzen der Hintergrund. Zu dem Objekt „Lichtband" wurde hier also ein *Segment* erzeugt.

In der Regel werden komplexe Objekte in mehrere Segmente aufgelöst. Erste wichtige Aufgabe, die meistens nicht trivial ist und die weiteren Schritte wesentlich beeinflußt, ist die Festlegung, wie ein komplexes Objekt in einfache Bestandteile zerlegt werden kann. Auch hierzu einige Beispiele (Bildfolge 23.2):

Ein Widerstand (Bild 23.2-a) kann z.B. in die Bestandteile „linker/rechter Anschlussdraht", „Keramikkörper" und „Farbcodierung" zerlegt werden. Man kann sich gut vorstellen, dass bei einer geeigneten Zuführung von Widerständen zu einem Videosensor diese Teile segmentiert werden können. Es werden dann Segmente für den linken und den rechten Anschlussdraht und für den Keramikkörper erzeugt. Die Farbcodierung ist eine zusätzliche Eigenschaft des Keramikkörpers, die erst dann analysiert werden kann, wenn das Segment zum Keramikkörper gefunden ist.

Schwieriger ist es schon bei einem Messgerät (Bild 23.2-b): Es hat ein Gehäuse mit einer bestimmten Abmessung. Im oberen Teil befindet sich die Anzeige mit einer Skaleneinteilung und einem Zeiger, im unteren Teil links und rechts zwei Anschlussbuchsen und in der Mitte

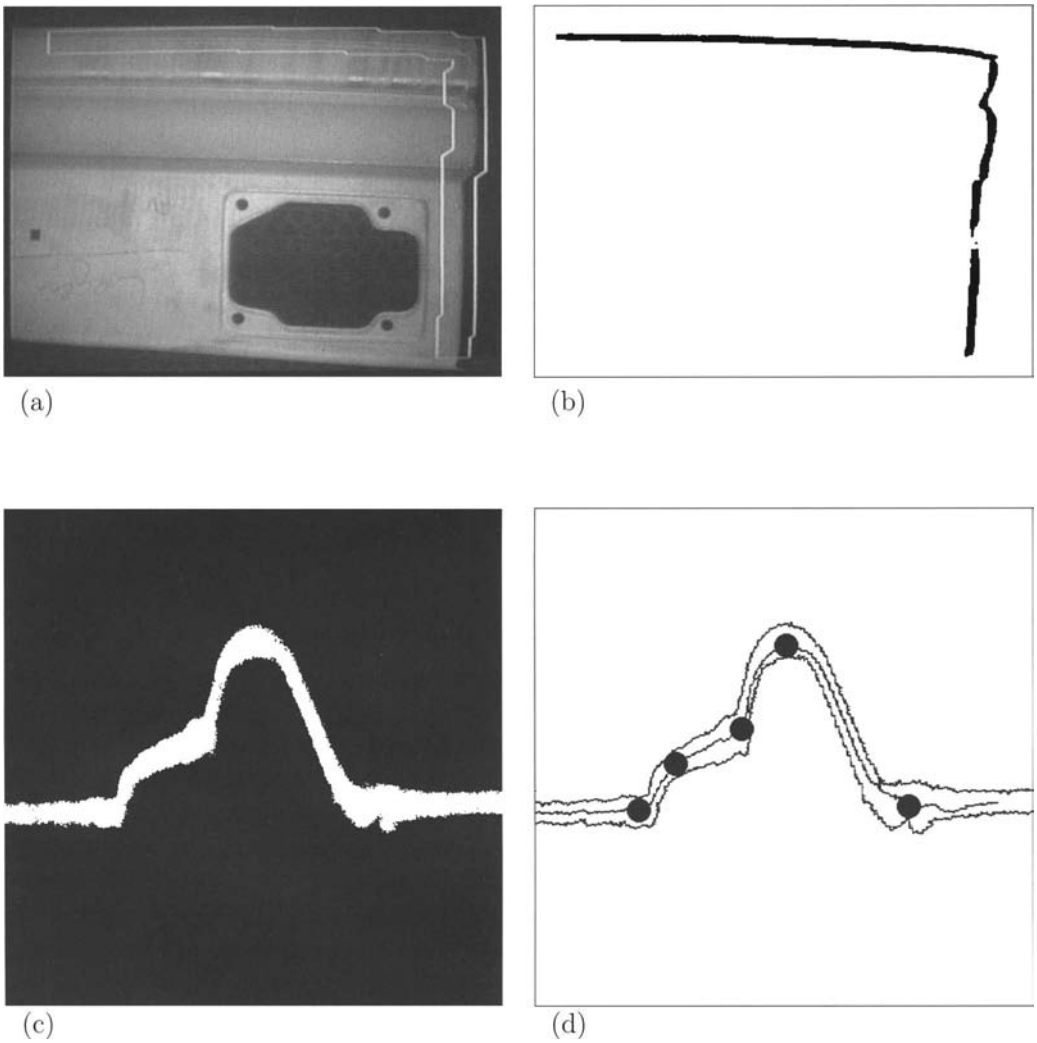
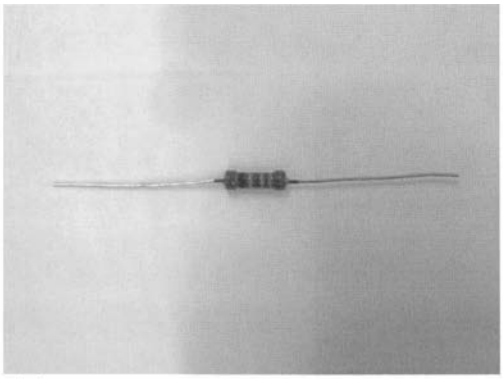
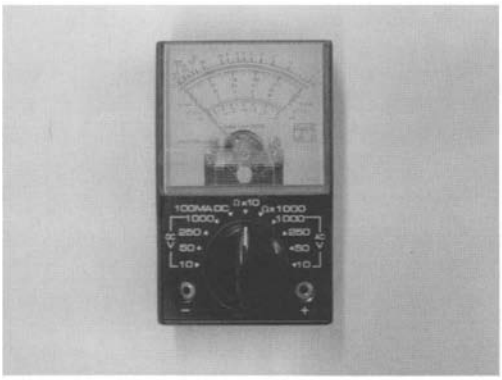


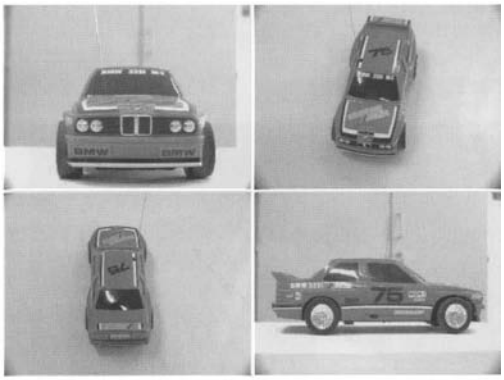
Bild 23.1: Qualitätskontrolle bei Kleberaupen. (a) Blechteil mit eingeblendeter *area-of-interest*. (b) Binarisierte Kleberaupe. Erzeugt die Binarisierung zu wenig Vordergrundbildpunkte in der *area-of-interest*, so kann man entscheiden, dass die Kleberaupe vermutlich nicht vorhanden ist. Eine weitere Analyse des Verlaufs liefert etwaige Unterbrechungen. (c) Zur Untersuchung der Form wird ein schmales Lichtband schräg auf die Kleberaupe projiziert und von oben aufgezeichnet. (d) Die Form des Lichtbandes kann durch die Bestimmung markanter Punkte und deren relativer Lage untersucht werden.



(a)



(b)



(c)

Bild 23.2: Beispiele für Objekte. (a) Widerstand. (b) Messgerät. (c) Fahrzeug.

ein Drehknopf, der von einer Skala umgeben ist. Bei einer passenden Zuführung eines derartigen Messgeräts, so dass es in einer definierten Lage ist, wäre es sicher möglich, die einzelnen Segmente zu erkennen.

Beim dritten Beispiel (Auto, 23.2-c) ist es nicht mehr ohne weiteres möglich, eine Zerlegung in einzelne erkennbare, einfache Bestandteile anzugeben. Wenn das Auto bei einer gegebenen Problemstellung aus verschiedenen Richtungen beobachtet wird, sieht es für den Betrachter ganz unterschiedlich aus. Bei manchen Anwendungen kann man mit apriori-Wissen, z.B. dass man das Fahrzeug immer nur von der Seite sieht, eine Zerlegung in einfache Bestandteile angeben.

Anhand dieser drei Beispiele sollte gezeigt werden, dass die Beschreibung der Zusammensetzung eines komplexen Objektes aus einfachen Bestandteilen die wichtigste Voraussetzung für eine erfolgreiche Segmentierung und eine daran anschließende Erkennung ist. Es macht sich auf jeden Fall bezahlt, wenn hier sorgfältig analysiert wird. Für die weiteren

Betrachtungen wird nun angenommen, dass dieses Problem gelöst ist. Zum Teil wird auch vereinfachend angenommen, dass ein Objekt nur aus einem Segment besteht, wie das z.B. bei einer Beilagscheibe der Fall ist.

In manchen Anwendungsgebieten ist es schwierig, die einzelnen Objekte, mit denen das System konfrontiert wird, exakt anzugeben. Ein Beispiel dazu ist die Erkennung von Handschriften: Ein Objekt dieses Problemkreises könnte der Buchstabe „A" sein. Man hat eine Vorstellung, wie ein „A" aussehen sollte. Aber bei handgeschriebenen Texten wird dieser Buchstabe von Schreiber zu Schreiber und sogar beim selben Schreiber immer unterschiedlich ausfallen. So wird man festlegen müssen, bis zu welcher Grenze man eine Struktur noch als „A" akzeptieren will. In diesem Fall spricht man von der *Objektklasse* „A", in der alle zulässigen Ausprägungen zusammengefasst sind.

Die Objekte (oder Objektklassen) werden in unserer Umwelt durch eine Vielzahl von physikalisch messbaren Größen charakterisiert [1]. Im Rahmen eines bestimmten Problemkreises wird man sich auf eine spezielle Sensorik festlegen. Damit hat man sich aber auch auf spezielle physikalische Größen festgelegt, die die Objekte charakterisieren. Falls man mit den gewählten Sensoren gerade diejenigen Eigenschaften nicht erfasst, in denen sich die Objekte unterscheiden, so hat man die falschen Sensoren gewählt (Abschnitt 16.1).

Die aufgezeichneten Originaldaten spannen einen N-dimensionalen *Merkmalsraum* auf, wobei N die Anzahl der Informationsschichten (Kanäle) ist. Wird z.B. mit einer Farbvideokamera ein RGB-Bild aufgezeichnet, so ist der Merkmalsraum der Originaldaten dreidimensional. In diesem Merkmalsraum sind den aufgezeichneten Objekten (Objektklassen) bestimmte Bereiche zugeordnet. Diese Bereiche werden als *Muster* (*Musterklassen*) bezeichnet. Die Lage der Musterklassen im Merkmalsraum ist nicht bekannt. Wenn die aufgezeichneten charakteristischen Messgrößen richtig gewählt sind, so kann man hoffen, dass Musterklassen kompakte Bereiche sind, die für die verschiedenen Klassen getrennt liegen.

Nun kann es aber sein, dass die aufgezeichneten Originaldaten es nicht ermöglichen, die Musterklassen in kompakte, getrennte Cluster zu transformieren. Dann müssen aus den Originaldaten abgeleitete Merkmale berechnet werden. Es wird dazu eine N_e-kanalige Bildfolge $\mathbf{S}_e = (s_e(x, y, n, t))$ als Originaldaten betrachtet. Man kann Merkmale aus den Kanälen oder aus Kanalkombinationen verwenden. Das wird in den Kapiteln 24 und 25 näher untersucht. Auch entlang der Zeitachse können Merkmale berechnet werden. Beispiele dazu sind Verschiebungsvektorfelder. Diese Thematik wird in Kapitel 26 erläutert. Schließlich kann man auch in Umgebungen eines Bildpunktes in der Position (x, y) Merkmale berechnen. Beispiele dazu sind Maßzahlen zur Textur, die in den Kapiteln 27 bis 30 ausführlich behandelt werden. Nach dem Schritt der Merkmalsberechnung kann man das Ergebnis mit einer N_a-kanaligen Szene $\mathbf{S}_a = (s_a(x, y, n))$ beschreiben, bei der jeder Kanal $n = 0, 1, ..., N_a - 1$ Maßzahlen für das jeweilige Merkmal enthält.

Diese Szene $\mathbf{S}_a$, die natürlich auch mit der Originalszene $\mathbf{S}_e$ identisch sein kann (wenn

[1] Natürlich werden Objekte auch noch durch andere Eigenschaften charakterisiert, wie z.B. Tiere oder Menschen durch bestimmte Verhaltensweisen. Diese Eigenschaften sind in der Regel aber nicht physikalisch messbar. Auf die Betrachtung derartiger „Merkmale" wird im Folgenden verzichtet.

keine aus den Originaldaten abgeleiteten Merkmale berechnet werden), ist die Eingabe für die eigentlichen Segmentierungsverfahren (Klassifikatoren), die ab Kapitel 31 besprochen werden.

In den folgenden Kapiteln werden nun die unterschiedlichsten Möglichkeiten vorgestellt, wie man Merkmale für eine Segmentierung aus den Originaldaten gewinnnen kann. Da einige der vorgestellten Verfahren (z.B. die Gauß- und Laplace-Pyramiden in Kapitel 28) nicht nur zur Merkmalsberechnung eingesetzt werden können, sondern neben diesem Gesichtspunkt auch noch andere Anwendungsmöglichkeiten haben, werden diese ebenfalls dargestellt.

Kapitel 24

Merkmale: Grauwert und Farbe

24.1 Anwendungen

Der Grauwert eines Bildes kann als einfaches Merkmal für eine Bildsegmentierung angesehen werden. In einem mikroskopischen Bild können z.B. dunkle Zellen vor einem hellen Bildhintergrund zu sehen sein. Die Binarisierung eines derartigen Bildes ist ein einfaches Beispiel für eine Segmentierung.

Ein Grauwertbild kann aber auch das Ergebnis einer aufwändigen Merkmalsberechnung sein. So können z.B. die Grauwerte Maßzahlen für die Textur oder die Bewegung sein. Wenn die verschiedenen Klassen in diesem Bild in deutlich getrennten Grauwert- (Merkmals-) Intervallen liegen, können sie mit einfachen Schwellwertverfahren segmentiert werden.

Die Verarbeitung von Multispektral- und Farbbildern war lange Zeit fast ausschließlich eine Domäne der Fernerkundung (Luft- und Satellitenbildauswertung). Hier werden, meistens ohne allzu großen „Echtzeitdruck", die riesigen Datenmengen, nach geometrischen Korrekturen und weiteren Vorverarbeitungsschritten, mit multivariaten Klassifikatoren verarbeitet, um so z.B. thematische Karten und Landnutzungskartierungen zu erstellen.

Leistungsfähige Prozessoren, großer Hauptspeicherausbau, sowie Hintergrundspeicher mit gewaltigem Fassungsvermögen machen es möglich, die Verarbeitung und Auswertung von Farbbildern in vielen neuen Bereichen einzusetzen. So werden in der Industrie multisensorielle Systeme entwickelt, die u.a. auch Farb- und Multispektralbilder aufzeichnen. Die so gewonnenen Farb- oder Multispektraldaten können unter Umständen direkt als Merkmale für eine Segmentierung verwendet werden.

Aber auch im privaten Bereich sind digitalisierte Farbbilder heute ein Standard. Man denke nur an die Möglichkeit der Digitalisierung von Videoaufzeichnungen mit kostengünstigen Farbframegrabbern, die Digitalisierung mit hochauflösenden Scannern oder den Einsatz von Digitalkameras.

In diesem Kapitel werden neben dem Gesichtspunkt des „Merkmals Farbe" einige Verfahren beschrieben, die sich bei der Verarbeitung und Darstellung von Farbbildern einsetzen lassen. Als erstes wird gezeigt, wie man auch ohne Farbframegrabber Farbbilder digitali-

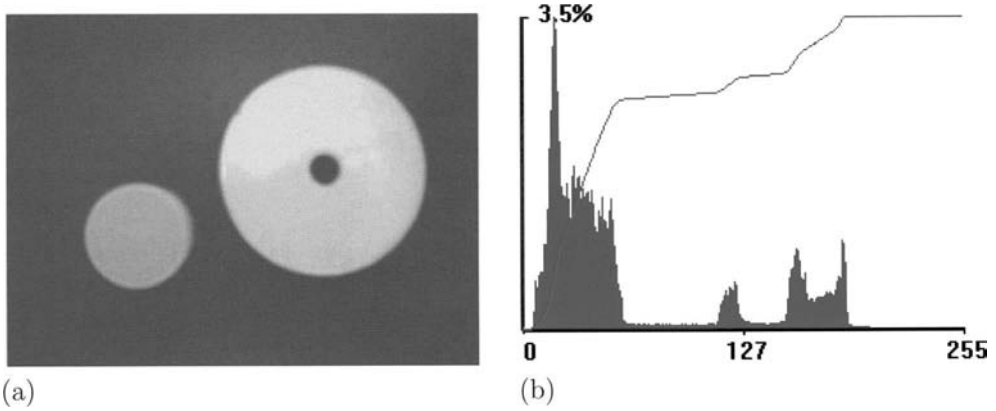


Bild 24.1: (a) Grauwertbild mit zwei hellen Teilen auf einem dunklen Hintergrund. Das Bild enthält somit die drei Klassen „Hintergrund", „Objekt links unten" und „Objekt rechts oben". (b) Histogramm zu Bild (a): Man sieht deutlich die drei lokalen Maxima der drei Klassen.

sieren kann und wie man Farbauszüge, die mit unterschiedlichen Farbfiltern digitalisiert wurden, angleichen kann.

Außerdem werden Techniken erläutert, die es erlauben, RGB-Bilder auf Ausgabesystemen mit nur acht Bit pro Pixel (z.B. eine ältere Grafikkarte oder ein Farbdrucker) in befriedigender Qualität darzustellen. In diesem Zusammenhang wird auch ein unüberwachter Klassifikator erläutert.

24.2 Merkmal: Grauwert

Wenn sich in einem Grauwertbild helle Objekte vor einem dunklen Hintergrund oder dunkle Objekte vor einem hellen Hintergrund abzeichnen, kann man sie mit einfachen Schwellwertverfahren segmentieren. Bild 24.1-a zeigt ein Beispiel eines Grauwertbildes mit zwei Objekten vor einem dunklen Hintertgrund. Das Bild enthält somit die drei Klassen „Hintergrund", „Objekt links unten" und „Objekt rechts oben". Rechts daneben ist in Bild 24.1-b das Histogramm gezeigt. Man sieht deutlich die drei lokalen Maxima der drei Klassen: Der Bildhintergrund liegt z.B. etwa im Grauwertbereich von 0 bis 80.

Es ist einfach, den Hintergrund oder das Objekt rechts/oben zu segmentieren: Für den Hintergrund werden alle Grauwerte von 0 bis 80 der Klasse „Hintergrund" zugeordnet und alle weiteren Grauwerte einer „Zurückweisungsklasse". Für das Objekt rechts/oben werden die Grauwerte 0 bis 150 der Zurückweisungsklasse und die Grauwerte 151 bis 255 der Klasse „Objekt rechts oben" zugewiesen. Die Bilder 24.2-a und -b zeigen das Ergebnis.

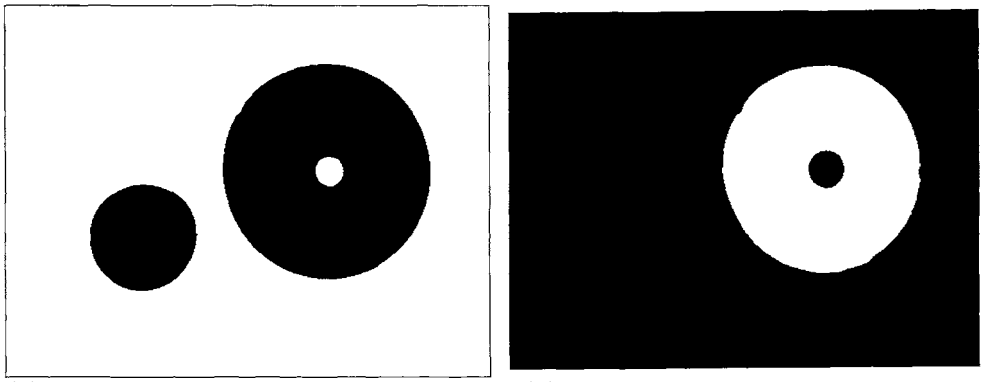
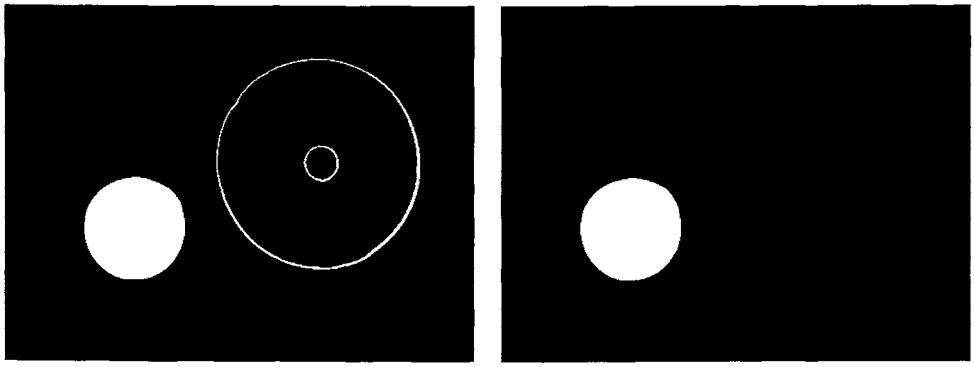


Bild 24.2: Segmentierung durch Schwellwertbildung: Die Segmente sind weiß dargestellt, die Zurückweisungsklasse schwarz. (a) Klasse „Hintergrund": Grauwertintervall 0 bis 80 im Original. (b) Klasse „Objekt rechts oben": Grauwertintervall 151 bis 255 im Original.

Probleme gibt es bei der Segmentierung der Klasse „Objekt links unten". Aus dem Histogramm in Bild 24.1-b kann man ablesen, dass die Grauwerte dieser Klasse etwa zwischen 110 und 145 liegen. Wenn man das Original mit diesem Grauwertbereich segmentiert und alle anderen Grauwerte der Zurückweisungsklasse zuordnet, erhält man das Ergebnis von Bild 24.3-a. Man sieht deutlich, dass auch Bildpunkte, die zur Klasse „Objekt rechts oben" gehören, der Klasse „Objekt links unten" zugeordnet wurden. Das liegt daran, dass im Übergangsbereich zwischen dem dunklen Hintergrund und dem hellen Objekt rechts/oben auch Grauwerte auftreten, die im Grauwertintervall des Objektes links/unten vorkommen. Unter Umständen kann man einen derartigen Segmentierungsfehler nachträglich korrigieren. In Bild 24.3-b wurde z.B. auf das Ergebnis der Segmentierung eine morphologische opening-Operation (zwei Erosionen gefolgt von zwei Dilatationen, Kapitel 19) angewendet, wodurch der Rand um das Objekt rechts/oben verschwindet.

Abschließend zu diesem Abschnitt ein Beispiel eines Grauwertbildes, das das Ergebnis einer Berechnung von Maßzahlen für die Textur ist. Bild 24.4-a zeigt einen Bildausschnitt mit zwei unterschiedlichen Stoffstrukturen. Durch die Berechnung eines Texturmaßes, hier die Kantendichte (Abschnitt 27.5), ist es gelungen, die beiden Texturen in unterschiedliche Grauwertintervalle (Merkmalsintervalle) abzubilden (Bild 24.4-b). Das Ergebnis einer Segmentierung zeigt Bild 24.4-c.



(a) (b)

Bild 24.3: Segmentierung durch Schwellwertbildung: Die Segmente sind weiß dargestellt, die Zurückweisungsklasse schwarz. (a) Klasse „Objekt links unten": Grauwertintervall 111 bis 145 im Original. Es werden auch Bildpunkte segmentiert, die zur Klasse „Objekt rechts oben" gehören. (b) Mit einer morphologischen opening-Operation (Erosionen, gefolgt von Dilatationen) können die falsch segmentieren Bildpunkte eliminiert werden.

24.3 Merkmal: Farbe

In den folgenden Betrachtungen wird das RGB-Farbmodell verwendet. Ein RGB-Bild ist, wie in Abschnitt 16.5 beschrieben, ein dreikanaliges Bild $\mathbf{S} = (s(x, y, n))$, mit $n = 0, 1, 2$ und der Kanalreihenfolge Rot, Grün und Blau. Der Wertebereich der einzelnen Kanäle ist die Grauwertmenge $G = \{0, 1, ..., 255\}$, sodass ein Farbbildpunkt aus 24 Bit zusammengesetzt ist, was bedeutet, dass $2^{24} = 16777216$ Farben dargestellt werden können.

Die Werte der Farbkomponenten liegen hier nicht, wie im Grundlagenabschnitt 16.5 dargestellt, im Intervall [0,1], sondern in der Grauwertmenge G. Bei Umrechnung in andere Farbmodelle ist das unter Umständen zu beachten. Wegen der diskretisierten Werte können sich bei der Umrechnung Fehler ergeben.

Die Farbe ist ein wichtiges Merkmal für die Segmentierung von Bildern. Ein Beispiel, wie die Muster von Objekten, in diesem Fall die verschiedenfarbigen Köpfe, im 3-dimensionalen RGB-Merkmalsraum liegen, zeigt die Bildfolge 24.5. Da der dreidimensionale RGB-Merkmalsraum schwer darzustellen ist, wurden hier die drei zweidimensionalen Merkmalsräume RG, RB und GB abgebildet. Bild 24.5-a zeigt das Farbbild. Die Bilder 24.5-b bis 24.5-d zeigen den Rot-, Grün- und Blaukanal. In den Bildern 24.5-e bis 24.5-g sind die zweidimensionalen RB- RG- und GB-Merkmalsräume abgebildet. Man sieht deutlich, wo die verschiedenfarbigen Objekte (Köpfe und Hintergrund) im Merkmalsraum die Punkthäufungen (cluster) bilden.

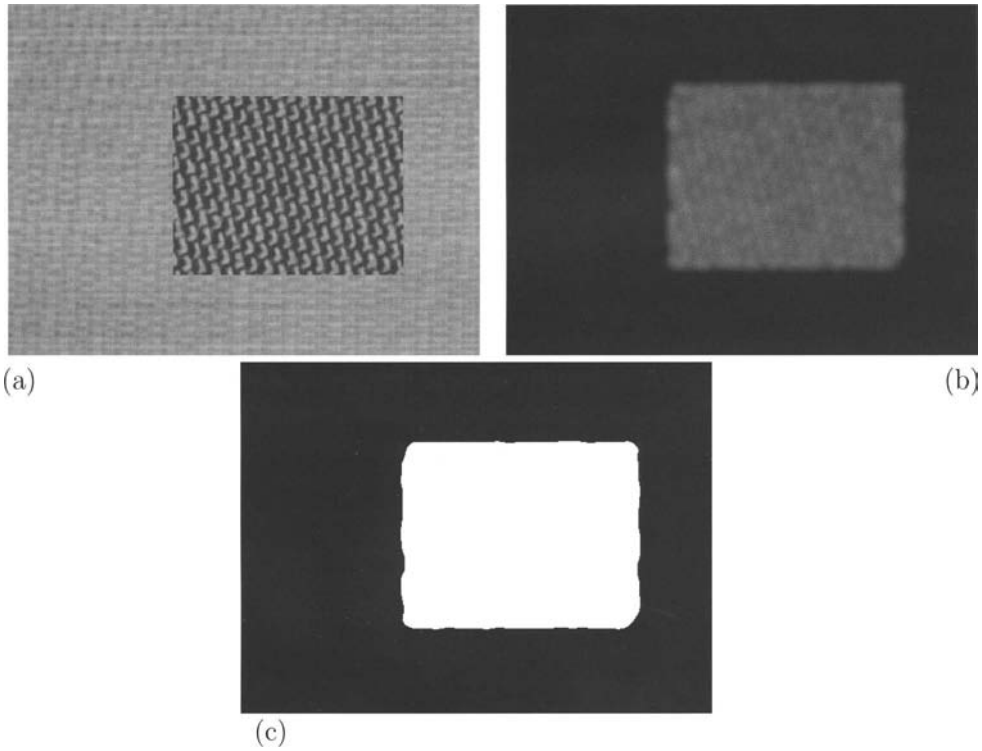


Bild 24.4: (a) Zwei unterschiedlich strukturierte Stoffe. (b) Durch die Berechnung eines Texturmaßes ist es gelungen, die beiden Texturen in getrennte Grauwertintervalle (Merkmalsintervalle) abzubilden. (c) Ergebnis der Segmentierung einer der beiden Texturen.

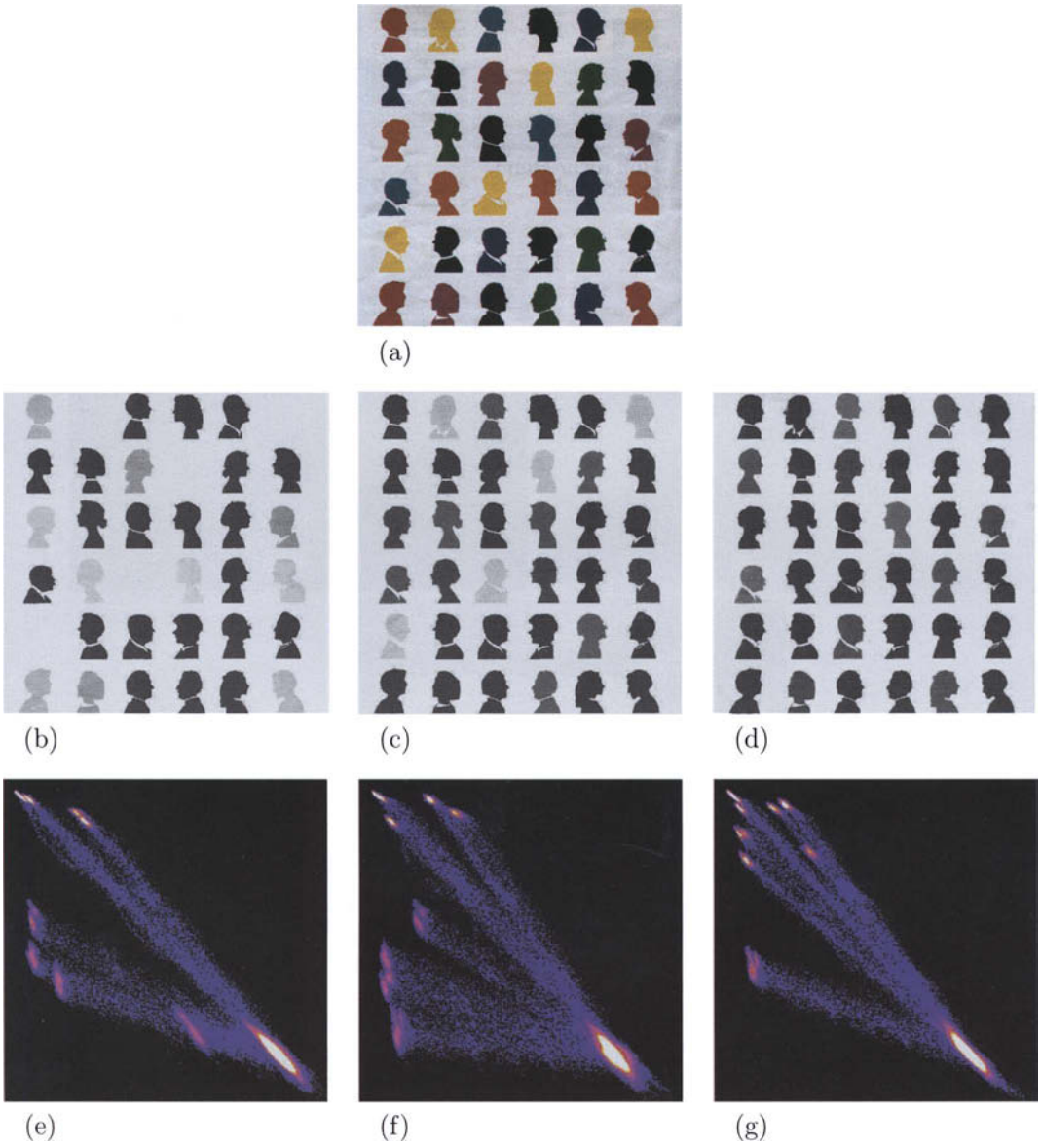


Bild 24.5: (a) RGB-Farbbild. (b) bis (d): Rot-, Grün- und Blaukanal. (e) bis (g): Zweidimensionale RG- RB- und GB-Merkmalsräume. Man sieht deutlich, wo die verschiedenfarbigen Köpfe (Objekte) im Merkmalsraum die Punkthäufungen (cluster) bilden. Zur Orientierung: Der Koordinatenursprung ist in den Merkmalsräumen links oben. Im RG-Raum z.B. ist der Rotkanal nach unten und der Grünkanal nach rechts aufgetragen. Die große Punktwolke rechts unten entspricht dem hellen Hintergrund der Köpfe. Die Zuordnung der weiteren Cluster sei dem Leser als Übung überlassen.

Bei der Besprechung der Klassifikatoren werden in den Kapiteln 31 bis 33 ebenfalls häufig Beispiele mit dem Merkmal Farbe verwendet.

Die hier folgenden Abschnitte befassen sich mit weiteren Aspekten bei der Digitalisierung und der bildlichen Wiedergabe von Farbbildern.

24.4 Reduktion der Farben in einem Farbbild durch Vorquantisierung

Bei der Verarbeitung und vor allem bei der bildlichen Darstellung von digitalisierten Farbbildern ist man oft vor das folgende Problem gestellt: Ein Farbbild, dessen Qualität mit den Verfahren der vorangehenden Kapitel verbessert wurde, soll auf einem Farbmonitor dargestellt oder über einen Farbdrucker ausgegeben werden. Manchmal sind (z.B. aus Kostengründen in sehr einfachen Umgebungen) Echtfarbsysteme, die 24-Bit (oder mehr) RGB-Bilder darstellen können, nicht verfügbar. Dagegen sind meistens Ausgabesysteme vorhanden, die 256 verschiedene Grau- oder Farbtöne darstellen können (z.B. sehr einfache Grafikkarten oder Farbdrucker).

Um ein RGB-Bild, das ja durch seine 24-Bit-Bildpunkte theoretische $2^{24} = 16777216$ verschiedene Farbtöne besitzen kann, auf einem Ausgabegerät mit nur 256 Farbtönen darstellen zu können, muss die Anzahl der Farbtöne drastisch reduziert werden. Dazu werden im Folgenden einige Beispiele gegeben.

Eine einfache Methode ist die Reduktion der 256 Grauwerte pro Farbauszug durch eine *Vorquantisierung*. In der Praxis hat es sich bewährt, z.B. den Rotanteil auf drei Bit (acht Rottöne), den Grünanteil auf drei Bit (acht Grüntöne) und den Blauanteil auf zwei Bit (vier Blautöne) zu reduzieren. Der so entstehende Farbwert kann dann in acht Bit (einem Byte) dargestellt werden:

7	6	5	4	3	2	1	0	Bitposition
B	B	G	G	G	R	R	R	Farbanteile

Bei einer einfachen Grafikkarte oder einem 8-Bit-Bildspeicher eines Bildverarbeitungssystems muss nun noch eine Farb-*look-up*-Tabelle (*cLuT*, Abschnitt 17.4) erzeugt werden, die die verwendete RGB-Aufteilung in einem Byte richtig darstellt. Der gesamte Algorithmus ist im folgenden zusammengefasst:

A24.1: 8-Bit-Echtfarben-Darstellung durch Vorquantisierung.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x, y, n))$ ist ein dreikanaliges RGB-Eingabebild.

◇ $\mathbf{S}_a = (s_a(x, y))$ ist ein einkanaliges Ausgabebild.

◇ Alle Divisionen werden ganzzahlig ausgeführt. | und & bedeuten bitweise logisches ODER bzw. UND, << bedeutet bitweiser Linksshift und >> bitweiser Rechtsshift.

Algorithmus:

(a) Für alle Bildpunkte des Bildes $\mathbf{S}_e = (s_e(x, y, n))$:

$red\_reduced = s_e(x, y, 0) >> 5;$

$green\_reduced = s_e(x, y, 1) >> 5;$

$blue\_reduced = s_e(x, y, 2) >> 6;$

$s_a(x, y) = red\_reduced|(green\_reduced << 3)|(blue\_reduced << 6);$

(b) Berechnung einer passenden $cLuT$ (Beispiel). Für alle $g \epsilon G$:

$cLuT.red[g] = (g\&7) << 5 + 16;$

$cLuT.green[g] = ((g\&56)/8) << 5 + 16;$

$cLuT.blue[g] = ((g\&192)/64) << 6 + 32;$

(c) Abspeichern der $cLuT$ in der Hardware des Displaysystems.

(d) Ausgabe des Bildes $\mathbf{S}_a$ in den Bildwiederholungsspeicher.

Ende des Algorithmus

Bild 24.6-a zeigt ein Beispiel zu diesem Verfahren. Da der Bildinhalt stark strukturiert ist, fällt beim Betrachten der geringere Umfang der Farbskala nicht auf. Bild 24.6-b besitzt dagegen größere homogene Flächen. Hier wird die verminderte Bildqualität deutlich. Diesen Effekt kann man etwas abschwächen, wenn man vor der Quantisierung der Farben das Original mit einer leichten Hochpassfilterung überlagert (Abschnitt 18.4). Dadurch werden die homogenen Flächen im Bild etwas gröber, und der störende Effekt von Bild 24.6-c tritt nicht mehr so stark auf (Bild 24.6-d).

24.5 Aufwändigere Verfahren zur Farbreduktion: Indexbilder

Bei aufwendigeren Verfahren zur Reduktion der Farben eines RGB-Bildes wird zunächst die Häufigkeitsverteilung der Farben berechnet. Dann werden die „wichtigsten" Farben des Bildes ermittelt und für die farbreduzierte Darstellung verwendet. Das farbreduzierte Bild besteht dann aus einem Indexbild, bei dem die Werte der Bildpunkte Zeiger in eine *look-up*-Tabelle sind. Der Rahmen eines derartigen Verfahrens ist im Algorithmus **A24.2** zusammengefasst. Die einzelnen Teilaspekte werden in den folgenden Abschnitten untersucht.

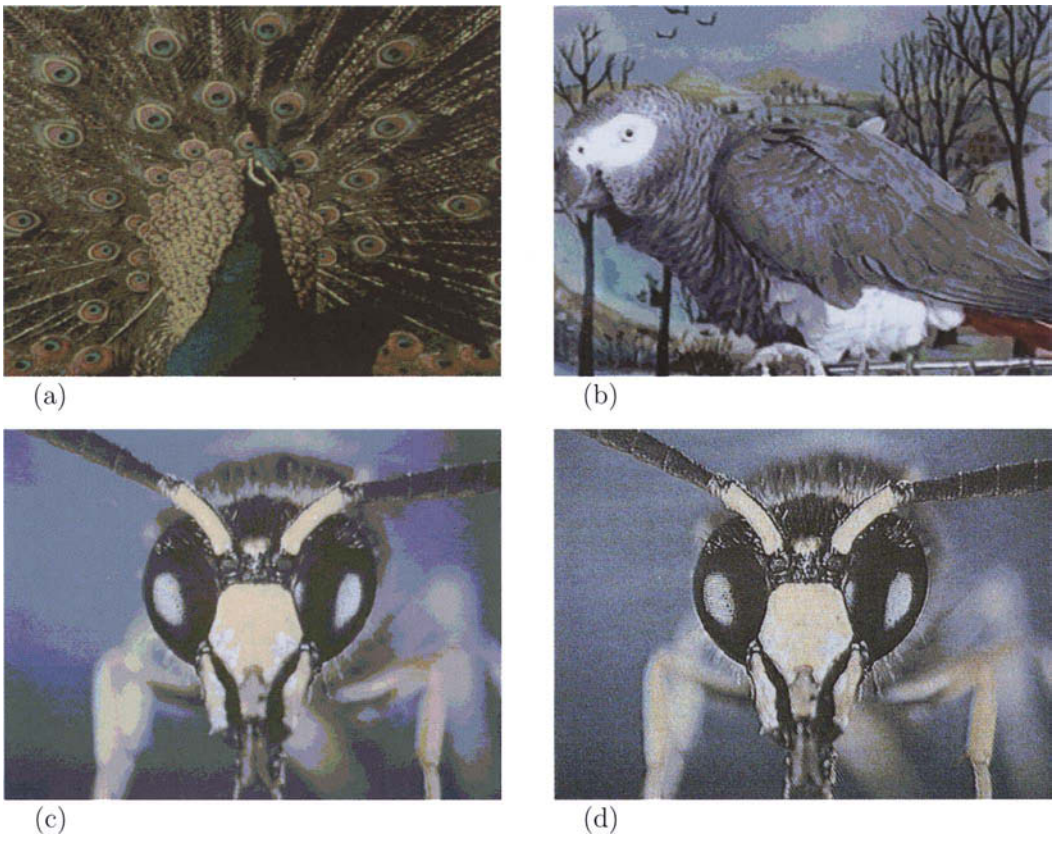


Bild 24.6: Beispiele zur Reduktion der Farbskala. (a) Bei stark strukturierten Bildern fällt der Einfluss der Farbskalareduktion nicht auf. (b) Hier sind neben strukturierten Bereichen auch homogene Bereiche vorhanden. Der Einfluss der Farb- quantisierung ist deutlich zu sehen. (c) Besitzt ein Bild größere homogene Flächen, so wird die verminderte Bildqualität sehr deutlich. (d) Hier wurde das Original vor der Quantisierung der Farben mit einem hochpassgefilterten Bild überlagert. Dadurch werden die homogenen Flächen etwas gröber und die Auswirkung der Quantisierung tritt nicht mehr so deutlich hervor.

A24.2: Erstellung von Index-Bildern.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x, y, n))$ ist ein dreikanaliges RGB-Eingabebild.

◇ $\mathbf{S}_a = (s_a(x, y))$ ist ein einkanaliges Ausgabebild (Indexbild).

Algorithmus:

(a) Berechnung der Häufigkeitsverteilung der Farbtöne des Bildes (Histogrammberechnung, Abschnitt 24.5.1).

(b) Aufstellen einer repräsentativen Farb-*look-up*-Tabelle (*cLut*) mit den wichtigsten Farbtönen des Bildes (Abschnitt 24.5.2).

(c) Abbildung der Farben des Originalbildes in die Farb-*look-up*-Tabelle *cLuT* (Abschnitt 24.5.3).

(d) Transformation des dreikanaligen Farbbildes $\mathbf{S}_e$ in ein einkanaliges Index-Bild $\mathbf{S}_a$ unter Verwendung der *cLuT* (Abschnitt 24.5.4).

Ende des Algorithmus

24.5.1 Die Farbhäufigkeitsverteilung eines Farbbildes

Vom mathematischen Standpunkt aus betrachtet können die relativen Häufigkeiten der Farben eines Farbbildes leicht beschrieben werden:

$$\mathbf{s}_e(x, y) = \mathbf{g} = (r, g, b)^T; \quad r, g, b \in G = \{0, 1, ..., 255\}; \tag{24.1}$$

$$p(\mathbf{g}) = \frac{a_\mathbf{g}}{M},$$

wobei $a_\mathbf{g}$ die Häufigkeit des Farbvektors $\mathbf{g}$ in $\mathbf{S}_e$ und M die Anzahl der Bildpunkte von $\mathbf{S}_e$ ist.

In der Praxis stößt man schnell auf Schwierigkeiten, wenn man sich überlegt, wieviel Speicherplatz man für das Histogramm $p(\mathbf{g})$ benötigt. Ein Farbvektor $\mathbf{g}$ ist aus 24 Bit (jeweils 8 Bit für Rot, Grün und Blau) zusammengesetzt. Es gibt also 2^{24} verschiedene Farbvektoren, was bedeutet, dass das Histogramm $p_S(\mathbf{g})$ $2^{24} = 16777216$ Einträge vorsehen müsste. Bei 16 Bit pro Eintrag sind das 32 MByte. Da bei einem Bild mit 16 Mio. Pixel eine Farbe maximal 2^{24}-mal auftreten könnte, würde man sogar 24 Bit pro Eintrag benötigen. Das wären dann 48 MByte.

Eine Möglichkeit, die schon in Abschnitt 24.4 angewendet wurde, besteht in der Vorquantisierung der Grauwerte der Farbkanäle. Je nach Anwendungsfall können die Farbpixel auf 8 Bit (3 Bit Rot, 3 Bit Grün und 2 Bit Blau), auf 15 Bit (5 Bit Rot, 5 Bit Grün und 5 Bit

Blau) oder 17 Bit (6 Bit Rot, 6 Bit Grün und 5 Bit Blau) reduziert werden. Entsprechend wird die Farb-*look-up*-Tabelle kleiner.

Eine andere Überlegung führt zu den folgenden Verfahren in diesem Abschnitt: Ein Farbbild mit $512 \cdot 512$ Bildpunkten kann natürlich auch maximal $512 \cdot 512 = 2^{18}$ verschiedene Farben enthalten. Das Histogramm muss also maximal für $2^{18} = 262144$ Einträge vorgesehen werden. Bei 16 Bit pro Eintrag sind das nur 0.5 MByte. Bei einem $1024 \cdot 1024$ großen Bild wären es 2 MByte. Da es aber unwahrscheinlich ist, dass in einem Farbbild die maximal mögliche Zahl von Farben auftritt, kann das Histogramm noch weiter verkleinert werden.

Es stellt sich nun die Aufgabe, eine Abbildung H des 24-Bit-Farbraums in ein Histogramm vom Umfang $h\_size < 2^{24}$ zu konstruieren. Diese Abbildung kann natürlich nicht eineindeutig (injektiv) sein. Mit einer *Hash-Codierung* lässt sich dieses Problem lösen.

Zunächst die Datenstruktur des Histogramms (der Häufigkeitstabelle): Man benötigt einen Zähler für die Häufigkeit, außerdem muss der zu jedem Eintrag gehörige RGB-Wert (als „Schlüssel") mit abgespeichert werden, um vergleichen zu können, ob an dieser Stelle in der Hash-Tabelle auch die richtige Farbe eingetragen ist. Damit ergibt sich für die Hash-Tabelle diese Struktur:

0	*count*	R	G	B
	...			
$h\_size - 1$	*count*	R	G	B

Das Histogramm (die Hash-Tabelle) wird mit *h_tab* bezeichnet, die einzelnen Datenfelder mit *h_tab.count*, *h_tab.R*, *h_tab.G* und *h_tab.B*.

Die Hash-Adresse zu einem Farbvektor $\mathbf{g} = (r, g, b)^T$ wird mit der Modulofunktion berechnet. Treten bei der Einspeicherung in die Hash-Tabelle Kollisionen auf, so werden sie durch eine Fortschaltgröße l und Sondieren eines neuen Speicherplatzes aufgelöst. Es ist zu beachten, dass durch wiederholtes Akkumulieren der Hashadresse h letztlich alle Positionen der Hash-Tabelle erreicht werden.

A24.3: Berechnung des Farbhistogramms eines RGB-Bildes.

Voraussetzungen und Bemerkungen:

- ◇ $\mathbf{S}_e = (s_e(x, y, n)), n = 0, 1, 2$ ist ein dreikanaliges RGB-Eingabebild.
- ◇ $\mathbf{g} = \mathbf{s}_e(x, y) = (r, g, b)^T; r, g, b \in G.$ ist ein RGB-Farbvektor von $\mathbf{S}_e$.
- ◇ *h_tab* ist das Farbhistogramm mit der Datenstruktur von oben und den Datenfeldern *h_tab.count*, *h_tab.R*, *h_tab.G* und *h_tab.B*.

Algorithmus:

(a) Für alle Bildpunkte $\mathbf{g} = \mathbf{s}_e(x, y)$ von $\mathbf{S}_e$:

(aa) Berechnung der Hashadresse h des Farbvektors $\mathbf{g}$:

$key = (r << 16)\&(g << 8)\&b;$

$h = key \bmod h\_size;$

$i = 1;$

(ab) Wiederhole Folgendes maximal so lange, bis alle Positionen der Hash-Tabelle inspiziert sind:

(aba) Falls der Eintrag in der Position h der Hash-Tabelle leer ist:

Farbvektor $\mathbf{g}$ eintragen;

$h\_tab.count = 1;$

Beende die Schleife (ab);

(abb) Falls der Eintrag in der Position h der Hash-Tabelle nicht leer ist und der Farbvektor $\mathbf{g}$ mit dem eingetragenen Farbvektor identisch ist:

$h\_tab.count = h\_tab.count + 1;$

Beende die Schleife (ab);

(abc) Falls der Eintrag in der Position h der Hash-Tabelle nicht leer ist und der Farbvektor $\mathbf{g}$ mit dem eingetragenen Farbvektor nicht identisch ist (Kollision):

Auflösen der Kollision durch Sondieren eines neuen Speicherplatzes:

$h = h + i;\ i = i + l;$ (z.B. $l = 1$ oder $l = 2$)

falls $h \geq h\_size:\ h = h - h\_size;$

(ac) Falls der Farbvektor $\mathbf{g}$ nicht eingetragen werden konnte, war die Hash-Tabelle zu klein gewählt.

Ende des Algorithmus

24.5.2 Erstellen einer Farb-Look-Up-Tabelle

Zur Erstellung einer Farb-*look-up*-Tabelle *cLuT* aus dem Histogramm der Farben werden unterschiedliche Verfahren angewendet. Ihr Ziel ist es, aus dem Histogramm die „wichtigsten" l_size Farbtöne (l_size: Länge der *cLuT*, z.B. $l\_size = 256$) herauszufinden. Zwei Verfahren werden im Folgenden erläutert.

24.5.2.1 Das Popularity-Verfahren

Das Popularity-Verfahren ist sehr einfach: Das Histogramm der Farben (*h_tab*) wird der Größe nach sortiert, dann werden die ersten *l_size* Farben, also die häufigsten, ausgewählt.

Es kann hier vorkommen, dass Farben, die zwar weniger häufig, jedoch subjektiv „wichtig" sind, nicht mit erfasst werden. Sie werden dann bei der reduzierten Darstellung möglicherweise völlig falsch wiedergegeben. Das liegt daran, dass die Farben im RGB-Farbraum *cluster* bilden: Ein bestimmter Farbwert tritt bei natürlichen Bildern nicht alleine häufig auf, sondern es werden auch die Farben seiner Umgebung häufig sein. Als Beispiel erzeuge ein Bild im RGB-Raum fünf *cluster*, deren Farben im Bild alle subjektiv wichtig erscheinen, die jedoch in den Häufigkeiten deutlich abnehmen. Der Popularity- Algorithmus wird dann möglicherweise die Farben der ersten drei „größeren" *cluster* erfassen, die letzten zwei aber nicht.

24.5.2.2 Das Median-Cut-Verfahren

Das *median-cut*-Verfahren versucht den RGB-Farbraum so aufzuteilen, dass auch kleinere Farbhäufungen (*cluster*) erfasst werden. Zunächst die Darstellung des Verfahrens, dann ein Beispiel.

A24.4: Median-Cut-Verfahren.

Voraussetzungen und Bemerkungen:

◇ *h_tab* sei das Farbhistogramm gemäß Algorithmus **A24.3** mit der oben angegebenen Datenstruktur und den Datenfeldern *h_tab.count*, *h_tab.R*, *h_tab.G* und *h_tab.B*.

◇ *l_size* sei die vorgegebene Anzahl der zu erzeugenden Farbrepräsentanten in der *cLuT*.

Algorithmus:

(a) Es wird eine Liste von Teiltabellen erstellt. Sie wird mit *list* bezeichnet. Zu Beginn des Verfahrens enthält *list* nur ein Element, nämlich das gesamte Histogramm *h_tab*.

(b) Wiederhole Folgendes so lange, bis die Liste der Teiltabellen *list* genau *l_size* Elemente besitzt:

(ba) Wiederhole Folgendes für alle Elemente von *list*. Das jeweils aktuell verarbeitete Element von *list* wird mit *act_tab* bezeichnet:

(baa) Ermittle zu den Spalten *act_tab.R*, *act_tab.G* und *act_tab.B* die Minima und Maxima min_R, max_R, min_G, max_G, min_B und max_B.

(bab) *act_col* sei die Spalte von *act_tab*, für die der Wertebereich maximal ist: $max = \max\{(max_R - min_R), (max_G - min_G), (max_B - min_B)\}$;

(bac) Sortiere die *act_tab* nach Maßgabe der Spalte *act_col*.

(bad) Teile die *act_tab* am Medianwert nach Maßgabe der Spalte *act_col*.

Die Teilung am Medianwert wird so durchgeführt, dass etwa gleichviele Farbvektoren in beiden Hälften auftreten.

(bae) Falls *list* jetzt *l_size* Elemente enthält: Beende beide Schleifen (weiter bei (c)).

(bb) Nachdem jetzt eine neue Liste *list* erstellt ist, sortiere die Elemente von *list*, fallend nach der Häufigkeit der Farbvektoren. In jeder Teilliste (jedem Element von *list*) werden die Felder *count* aufsummiert. Am Anfang der sortierten Liste steht dasjenige Element, das die größte Summe ergibt. Bei der Sortierung werden nur solche Elemente berücksichtigt, die mehr als einen Farbvektoreintrag enthalten.

(c) Bestimme die *l_size* Farbrepräsentanten. Falls ein Element von *list* nur aus einem Farbvektor besteht, wird dieser als Farbrepräsentant verwendet. Für Elemente von *list*, die mehr als einen Farbvektor enthalten: Bilde den Mittelwertfarbvektor und verwende ihn als Repräsentanten.

Ende des Algorithmus

Der *median-cut*-Algorithmus ist in verschiedenen Softwareprodukten, die eine Verarbeitung von Farbbildern erlauben, bereits integriert. Als Beispiele seien hier erwähnt:

- *Image Alchemy* von Handmade Software, Inc., Los Gatos, CA. oder das
- PPM-Software-Paket, das in vielen UNIX-Netzen auch als Quellcode zur Verfügung steht.

24.5.3 Abbildung der Farben des Originalbildes in die Farbtabelle

Nachdem die Farb-*look-up*-Tabelle *cLuT* berechnet ist, müssen die Farben des Originalbildes $\mathbf{S}_e$ in die *cLuT* abgebildet werden. Zu jedem Farbvektor von $\mathbf{S}_e$ wird ermittelt, zu welchem Farbvektor der *cLuT* er im RGB-Farbraum den geringsten Abstand hat. Durch diesen wird er dann repräsentiert. Es ist jedoch nicht notwendig, dies für alle Bildpunkte des Originalbildes durchzuführen. Im Farbhistogramm *h_tab* (der Hash-Tabelle) sind nämlich alle Farbvektoren von $\mathbf{S}_e$ erfasst. Es genügt also, für alle in *h_tab* auftretenden Vektoren die Zuordnung zur Farb-*look-up*-Tabelle durchzuführen. Dazu kann sogar dieselbe Tabelle verwendet werden, da die Werte von *h_tab.count* nicht mehr benötigt werden.

A24.5: Abbildung der Farben des Originals.

Voraussetzungen und Bemerkungen:

◇ *h_tab* sei das Farbhistogramm gemäß Algorithmus **A24.3** mit der angegebenen Datenstruktur und den Datenfeldern *h_tab.count*, *h_tab.R*, *h_tab.G* und *h_tab.B*.

◇ *cLuT*: Farb-*look-up*-Tabelle mit den ausgewählten Farbrepräsentanten des Bildes $\mathbf{S}_e$. Datenfelder von *cLuT*: *cLuT.R*, *cLuT.G* und *cLuT.B*.

Algorithmus:

(a) Für alle Einträge i von *h_tab*:

(ab) Für alle Einträge j von *cLuT*:

(aba) Berechne die minimale Distanz von (*h_tab.R*[i], *h_tab.G*[i], *h_tab.B*[i]) zu (*cLuT.R*[j], *cLuT.G*[j], *cLuT.B*[j]).

Dies sei der Eintrag k von *cLuT*.

(abb) Setze *h_tab.count*[i] = k.

Ende des Algorithmus

24.5.4 Segmentierung des Originalbildes

Nach diesen Vorbereitungen ist die Umsetzung des Originalbildes zu einem Index-Bild einfach: Jedem Farbvektor des Originalbildes wird ein Zeiger (Index) in die *cLut* zugeordnet. Welche Position in *cLut* das ist, ist in *h_tab* gespeichert.

A24.6: Transformation des Originals.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x, y, n))$ ist ein dreikanaliges RGB-Eingabebild.

◇ $\mathbf{S}_a = (s_a(x, y))$ ist ein einkanaliges Ausgabebild (Indexbild).

◇ *cLuT*: Farb-*look-up*-Tabelle mit den ausgewählten Farbrepräsentanten (wie oben);

Algorithmus:

(a) Für alle Bildpunkte $\mathbf{s}_e(x, y)$ des Bildes $\mathbf{S}_e = (s(x, y, n))$:

(aa) Suche den Farbton $\mathbf{s}_e(x, y) = \mathbf{g} = (r, g, b)^T$ in *h_Tab*. Dies sei der k-te Eintrag.

(ab) Setze: $s_a(x,y) = h\_tab.count[k]$.

Ende des Algorithmus

Es ist klar, dass zum farbreduzierten, einkanaligen Ergebnisbild $\mathbf{S}_a = (s_a(x,y))$ zusätzlich die Farb-*look-up*-Tabelle *cLuT* gehört, da die „Grau-"Werte $s_a(x,y)$ ja Indizes in die *cLuT* sind.

Auf Bildbeispiele zu diesen Verfahren wird verzichtet, da die oft nur geringen Unterschiede nur auf guten Farbmonitoren zu sehen sind und im Druck verloren gehen würden.

24.5.5 Segmentierung des Originalbildes mit Dithering

Mit der Technik des *dithering* kann der Segmentierungsschritt zusätzlich verbessert werden. Diese Technik ist vor allem dann geeignet, wenn die Segmentierung auf wenige Farben erfolgte, also eine mit nur wenigen Farben besetzte *cLuT* berechnet wurde.

Das Verfahren verteilt den Fehler, der bei der Darstellung eines Originalfarbvektors $\mathbf{s}_e(x,y)$ durch seinen Repräsentanten aus der *cLuT* gemacht wird, auf die Nachbarvektoren. Um dies in einem einzigen Durchlauf berechnen zu können, wird der Fehler nur auf die rechts und unten vom aktuellen Bildpunkt $\mathbf{s}_e(x,y)$ liegenden Nachbarn übertragen, wobei die Nachbarn unterschiedlich gewichtet werden:

$\mathbf{s}_e(x,y)$	$\mathbf{s}_e(x,y+1) + \frac{3}{8}\mathbf{q}$
$\mathbf{s}_e(x+1,y) + \frac{3}{8}\mathbf{q}$	$\mathbf{s}_e(x+1,y+1) + \frac{1}{8}\mathbf{q}$

Der Algorithmus dazu ist in **A24.7** zusammengestellt.

A24.7: Abbildung der Farben des Originals mit dithering.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y,n))$ ist ein dreikanaliges RGB-Eingabebild.

◇ $\mathbf{S}_a = (s_a(x,y))$ ist ein einkanaliges Ausgabebild (Indexbild).

◇ *h_tab* das Farbhistogramm (wie oben).

◇ *cLuT*: Farb-*look-up*-Tabelle mit den ausgewählten Farbrepräsentanten (wie oben).

Algorithmus:

(a) Für alle Bildpunkte $\mathbf{s}_e(x,y)$ des Bildes $\mathbf{S}_e$:

(aa) Suche den Farbton $\mathbf{s}_e(x,y) = \mathbf{g} = (r,g,b)^T$ in *h_tab*. Dies sei der j-te Eintrag, der auf den k-ten Eintrag von *cLut* weist.

(ab) Setze: $s_a(x,y) = h\_tab.count[j]$.

(ac) Berechne den Quantisierungsfehler $\mathbf{q}$:

$$\mathbf{q} = \begin{pmatrix} r - cLuT.R[k] \\ g - cLut.G[k] \\ b - cLut.B[k] \end{pmatrix}$$

(ad) Verteile den Quantisierungsfehler auf benachbarte Pixel:

$$\mathbf{s}_e(x,y+1) = \mathbf{s}_e(x,y+1) + \frac{3}{8} * \mathbf{q};$$

$$\mathbf{s}_e(x+1,y) = \mathbf{s}_e(x+1,y) + \frac{3}{8} * \mathbf{q};$$

$$\mathbf{s}_e(x+1,y+1) = \mathbf{s}_e(x+1,y+1) + \frac{1}{8} * \mathbf{q};$$

Ende des Algorithmus

24.5.6 Unüberwachte Klassifikatoren zur Reduktion der Farben

Ein etwas anderer Weg wird beschritten, wenn der RGB-Farbraum mit *multivariaten Klassifikatoren* (Kapitel 31) untersucht und aufgeteilt wird. Hier sind vor allem unüberwachte Klassifikatoren gut geeignet, da sie die Aufteilung des RGB-Merkmalsraums ohne ein vom Bearbeiter zur Verfügung gestelltes Trainingsset (Stichprobe) analysieren.

Dabei geht man folgendermaßen vor: Es wird die Anzahl der zu erzeugenden repräsentativen Farben, also der Umfang *c_size* der Farb-*look-up*-Tabelle *cLuT* vorgegeben. Dann wird ein unüberwachter *cluster*-Algorithmus angewendet, der den RGB-Farbraum in ebenso viele (*c_size*) *cluster* aufteilt. Die Mittelwertvektoren der *cluster* werden dann als Farbrepräsentanten verwendet.

Aus der Vielfalt der möglichen unüberwachten *cluster*-Algorithmen sei der einfache *Minimum-Distance-Cluster-Algorithmus* vorgestellt.

A24.8: Minimum-Distance-Cluster-Algorithmus.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y,n)), n = 0,1,2; \mathbf{s}_e(x,y) = \mathbf{g} = (r,g,b)^T$ sei ein RGB-Eingabebild.

Algorithmus:

(a) Wähle einen beliebigen Bildpunkt von $\mathbf{S}_e$ als Zentrum $\vec{z}_0$ des ersten *clusters* aus.

(b) Für alle Bildpunke $\mathbf{g} = \mathbf{s}_e(x, y)$ von $\mathbf{S}_e$:

(ba) Für alle bereits festgelegten *cluster*-Zentren $\vec{z}_i$:

(baa) Ermittle die minimale Distanz $d_i = d(\mathbf{g}, \vec{z}_i)$. Dies sei d_j.

(bb) Falls d_j kleiner als der vorgegebene Schwellwert c ist: Ordne $\mathbf{g}$ dem *cluster* j zu.

(bc) Falls d_j nicht kleiner als der vorgegebene Schwellwert c ist:

(bca) Falls die maximale Anzahl an *cluster*-Zentren noch nicht erreicht ist: Setze $\mathbf{g}$ als neues *cluster*-Zentrum $\vec{z}$.

(bcb) Falls die maximale Anzahl an *cluster*-Zentren erreicht ist: Ordne $\mathbf{g}$ dem *cluster* j zu.

Ende des Algorithmus

Ein großer Vorteil dieses Verfahrens liegt in der schnellen Verarbeitungszeit, da die *cluster*-Bildung in einem einzigen Durchlauf durch das gesamte Bild $\mathbf{S}_e$ berechnet wird.

Ein Nachteil ist sicher, dass das Ergebnis abhängig von der Verarbeitungsreihenfolge ist. Es kann z.B. der Fall auftreten, dass Bildpunkte einem *cluster* j zugeordnet werden, solange ein neues *cluster* k noch nicht erzeugt wurde. Wird dieses erzeugt, so werden Farbvektoren, die vorher *cluster* j zugewiesen wurden, ab diesem Zeitpunkt dem *cluster* k zugeordnet.

Dieser Effekt wird vermieden, wenn die *cluster*-Bildung iterativ mit mehreren Durchläufen durch die Bilddaten erfolgt. Ein Verfahren dieser Art, das auf der Basis der *fuzzy logic* aufbaut, wird in Kapitel 33 beschrieben.

Abschließend sei noch bemerkt, dass der Minimum-Distance-Cluster-Algorithmus nicht auf dreidimensionale RGB-Farbräume beschränkt ist, sondern auch als unüberwachter Klassifikator bei N-dimensionalen Merkmalsräumen eingesetzt werden kann.

Kapitel 25

Merkmale aus mehrkanaligen Bildern

25.1 Anwendungen

In den vorhergehenden Kapiteln wurden Bildverarbeitungsverfahren beschrieben, die jeweils nur einen Bildkanal verarbeiten. In diesem Kapitel werden nun Verfahren erläutert, die die Informationen von mehreren Bildkanälen verknüpfen. Handelt es sich dabei um Farb- oder Multispektralbilder, so lassen sich mit Differenz- oder Ratiobildung oft einprägsame Darstellungen für die visuelle Weiterverarbeitung erzielen. Durch die Verknüpfung von Kanälen bei logischen Bildern lassen sich viele praktische Effekte, z.B. Einblendungen erzielen. Den letzten Teil dieses Kapitels bildet die Hauptkomponententransformation, mit der versucht wird, nach einer Rotation des Merkmalskoordinatensystems redundante Informationen zu entfernen.

25.2 Summe, Differenz, Ratio

Für die Darstellungen in diesem Kapitel wird zunächst ein mehrkanaliges Bild vorausgesetzt gemäß:

$$\begin{aligned} \mathbf{S}_e = (s_e(x,y,n)) \quad & x = 0,1,...,L-1, \text{(Bildzeilen)} \\ & y = 0,1,...,R-1, \text{(Bildspalten) und} \\ & n = 0,1,...,N-1, \text{(Kanäle).} \end{aligned} \tag{25.1}$$

Ein einfaches Beispiel einer mehrkanaligen Operation ist die *gewichtete additive Verknüpfung* der einzelnen Kanäle:

$$\begin{aligned} & \mathbf{S}_e \rightarrow \mathbf{S}_a : \\ & s_a(x,y) = \frac{1}{N} \sum_{n=0}^{N-1} a_n \cdot s_e(x,y,n); \quad \sum_{n=0}^{N-1} a_n = 1. \end{aligned} \tag{25.2}$$

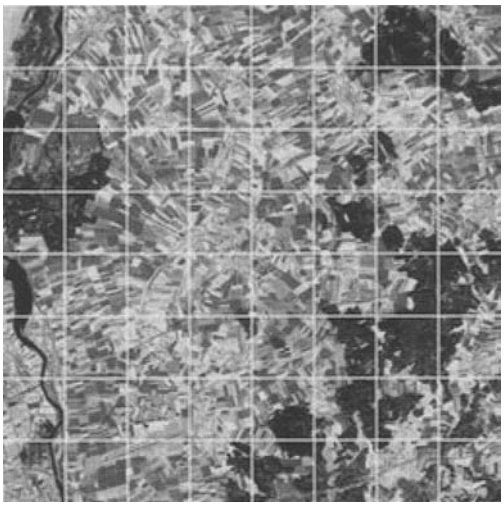


Bild 25.1: Geometrisch entzerrtes Satellitenbild mit überlagertem Gitternetz.

Ist dabei $\mathbf{S}_e$ ein Multispektralbild, so stellt der Grauwert $s_a(x, y)$ den Mittelwert des Bildpunktes in den einzelnen Kanälen dar. Speziell im Fall eines digitalisierten Farbbildes mit Rot-, Grün- und Blauauszug entsprechen dann die Grauwerte des transformierten Bildes in etwa den panchromatischen Grauwerten, die entstehen, wenn das Original ohne Farbfilter, also nur als Grautonbild, digitalisiert würde (HSI-Modell: Intensität).

Ein anderes Beispiel zur additiven Verknüpfung ist das Einblenden von grafischen Informationen in ein digitalisiertes Bild. Ist $s_e(x, y, 0)$ ein beliebiges Grauwertbild und $s_e(x, y, 1)$ ein weiterer Kanal mit grafischen Informationen wie Strichzeichnungen, Gittermuster, Text, usw., so kann durch die Operation (25.2) diese Information in den ersten Kanal eingeblendet werden (Bild 25.1). Dabei wird in dieser Darstellung nur vorausgesetzt, dass die grafische Information weiß (Grauwert 255) auf schwarzem Hintergrund (Grauwert 0) vorliegt.

Durch verschiedene Modifikationen kann (25.2) so geändert werden, dass z.B. die eingeblendete Information transparent oder nicht transparent ist oder dass überlagerte Strichzeichnungen immer einen optimalen Kontrast zum Hintergrund haben.

Die *Differenz* zweier Kanäle k und l sieht Folgendermaßen aus:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a :$$
$$s_a(x, y) = s_e(x, y, k) - s_e(x, y, l) + c. \tag{25.3}$$

Die Konstante c wird verwendet, um negative Grauwerte zu vermeiden. Sie kann z.B. c=127 gesetzt werden. Im Ausgabebild $\mathbf{S}_a$ oszillieren dann die Differenzen um den Grauwert

127. Treten dennoch Differenzen auf, die größer als 127 sind, so werden diese Werte auf 0 bzw. 255 gesetzt. Eine andere Möglichkeit besteht darin, dass zunächst die größte Differenz ermittelt wird und die Werte von $\mathbf{S}_a$ so skaliert werden, dass sie in die Grauwertmenge G passen.

Sind bei der Differenzbildung in (25.3) die Kanäle Multispektralkanäle, so sind im Bild $\mathbf{S}_a$ diejenigen Bereiche hervorgehoben, bei denen spektrale Unterschiede auftreten. Um diese Unterschiede deutlicher herauszuarbeiten, kann eine Kontrastanreicherung des Bildes $\mathbf{S}_a$ sinnvoll sein.

In Kapitel 17 wurden einige Beispiele zur Kalibrierung vorgestellt. Mit der Differenzbildung gemäß (25.3) lassen sich ebenfalls Kalibrierungsprobleme bearbeiten, so z.B. die Elimination von Vignettierungseinflüssen. Wird mit Hilfe einer Videokamera, die über AD-Wandler an ein Rasterbildspeichersystem angeschlossen ist, eine Reihe von Bildvorlagen digitalisiert, so ist durch die Einflüsse des Kameraobjektivs zu erwarten, dass die digitalisierten Bilder mehr oder weniger starke Randabschattungen aufweisen. Sollen diese Bilddaten anschließend zu Binärbildern umgewandelt werden, so ist der Schwellwert ortsabhängig und die Qualität der Binärbilder dementsprechend schlecht. Wird bei der Digitalisierung ein homogen weißes Kalibrierungsbild mit verarbeitet, so kann dieses von allen anderen Bildern abgezogen werden, so dass in den Differenzenbildern die Randabschattungen eliminiert sind. Dabei ist im jeweiligen Anwendungsfall die Konstante c passend zu wählen.

Bei der Verarbeitung von Multispektralbildern wird statt der Differenz zweier Kanäle k und l häufiger das *Ratio* verwendet:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a:$$
$$s_a(x,y) = \frac{s_e(x,y,k) - s_e(x,y,l)}{s_e(x,y,k) + s_e(x,y,l)} \cdot c_1 + c_2. \tag{25.4}$$

Das Ratio ist invariant gegen die Multiplikation der Kanäle des Originalbildes $\mathbf{S}_e$ mit einem konstanten Faktor. Wenn nun proportional zur Leuchtdichte des von der Kamera aufgenommenen Objektes quantisiert wurde, dann ist das Ratiobild $\mathbf{S}_a$ unabhängig von der Beleuchtung des Objektes. Im übrigen liefern Differenz und Ratio sehr ähnliche Bilder (25.2).

Da sich mit Falschfarbencodierung immer nur drei Bildkanäle den drei Grundfarben Rot, Grün und Blau zuordnen lassen, haben die in diesem Abschnitt aufgeführten Beispiele wichtige Anwendungen bei der farbbildlichen Darstellung von Multispektralbildern. Durch die gezeigten Operationen kann die Anzahl der Kanäle reduziert und nicht gewünschte Bildinformation eliminiert werden. Durch eine geschickte farbliche Darstellung können dann Bilder erstellt werden, die in vielen Anwendungsfällen ausgezeichnet für weitere visuelle Interpretationen geeignet sind. Umgekehrt kann man aus wenigen Kanälen durch Verwendung zusätzlicher Kenntnisse über die aufgenommenen Objekte mehrere Kanäle erzeugen. Als Beispiel seien Satellitenaufnahmen genannt, die nur mit zwei Spektralausschnitten vorliegen. Wenn man die Korrelationen der vorhandenen Kanäle bei typischen terrestrischen Objekten (z.B. Wald, Gewässer, Wiese, Wüste, Gestein, usw.) kennt, kann

(a)

(b)

(c)

Bild 25.2: Differenz und Ratio zweier Spektralbereiche eines digitalisierten Farbbildes. (a) Original. (b) Differenz: Rot- und Blauauszug (kontrastverstärkt). (c) Ratio: Rot- und Blauauszug (kontrastverstärkt).

man einen dritten „Farbauszug" berechnen und dann eine Echt- oder Falschfarbendarstellung erzeugen.

25.3 Verknüpfung von Kanälen bei logischen Bildern

In diesem Abschnitt wird anhand eines einfachen Beispiels gezeigt, wie einzelne Bildkanäle, mit Grauwertbildern und logischen Bildern miteinander verknüpft werden können. Dazu werden folgende Annahmen gemacht:

Es sei $\mathbf{S}_e = (s_e(x, y, n)), n = 0, 1, 2$ ein Bild mit drei Kanälen, die Folgendes enthalten (Bild 25.3-a, -b und -c):

Kanal $s_e(x, y, 0)$: Ein Grauwertbild.
Kanal $s_e(x, y, 1)$: Ein Grauwertbild.
Kanal $s_e(x, y, 2)$: Ein Maskenbild mit den Grauwerten 0 und 255.

In Bild 25.3-d wurden die drei Kanäle des Eingabebildes so kombiniert, dass der Bildhintergrund von Bild 25.3-a durch den Hintergrund von Bild 25.3-b ersetzt wurde.

$$\begin{aligned} s_a(x, y) &= \Big((s_e(x, y, 2).\text{EQUAL}.0) \cdot s_e(x, y, 0)\Big) + \\ &+ \Big((s_e(x, y, 2).\text{EQUAL}.255) \cdot s_e(x, y, 1)\Big). \end{aligned} \tag{25.5}$$

Die Boole'schen Ausdrücke in den inneren Klammern liefern den Wahrheitswert `TRUE`, der rechnerintern mit 1 (manchmal auch mit -1) codiert ist, falls im Kanal 2 ein Bildpunkt den Grauwert 0 bzw. 255 hat. In allen anderen Fällen liefern sie den Wahrheitswert `FALSE`, codiert mit 0. Durch die Multiplikation mit dem jeweiligen Grauwert im Kanal 0 bzw. Kanal 1 haben diese Bildpunkte im Ausgabebild $\mathbf{S}_a$ die gewünschten Grauwerte.

Wenn das Maskenbild kein Zweipegelbild ist, sondern einzelne Flächenstücke mit unterschiedlichen Grauwerten codiert sind, können gezielt bestimmte Flächen kombiniert werden. Die Bilder 25.4 zeigen dazu ein Beispiel.

Diese Beispiele zeigen die vielfältigen Auswertungsmöglichkeiten, wenn mehrkanalige Bilder dieser Art vorliegen. Bei Bilddaten z.B. aus der Fernerkundung lassen sich mit den vorgestellten Techniken hilfreiche Zusatzinformationen für Anwendungen in der Raumplanung, Geologie, Ozeanografie, Ökologie, erzeugen.

25.4 Die Hauptkomponententransformation

Die Hauptkomponententransformation ist eine Standardmethode, die zur Datenreduktion eingesetzt wird. Zur Erläuterung wird zunächst ein zweikanaliges Bild $\mathbf{S}_e = (s_e(x, y, n)), n = 0, 1$, betrachtet, dessen Kanäle relativ stark korreliert sind. Die Verteilung der Grauwerte könnte dann etwa durch das zweidimensionale Histogramm von Bild 25.5-d wiedergegeben

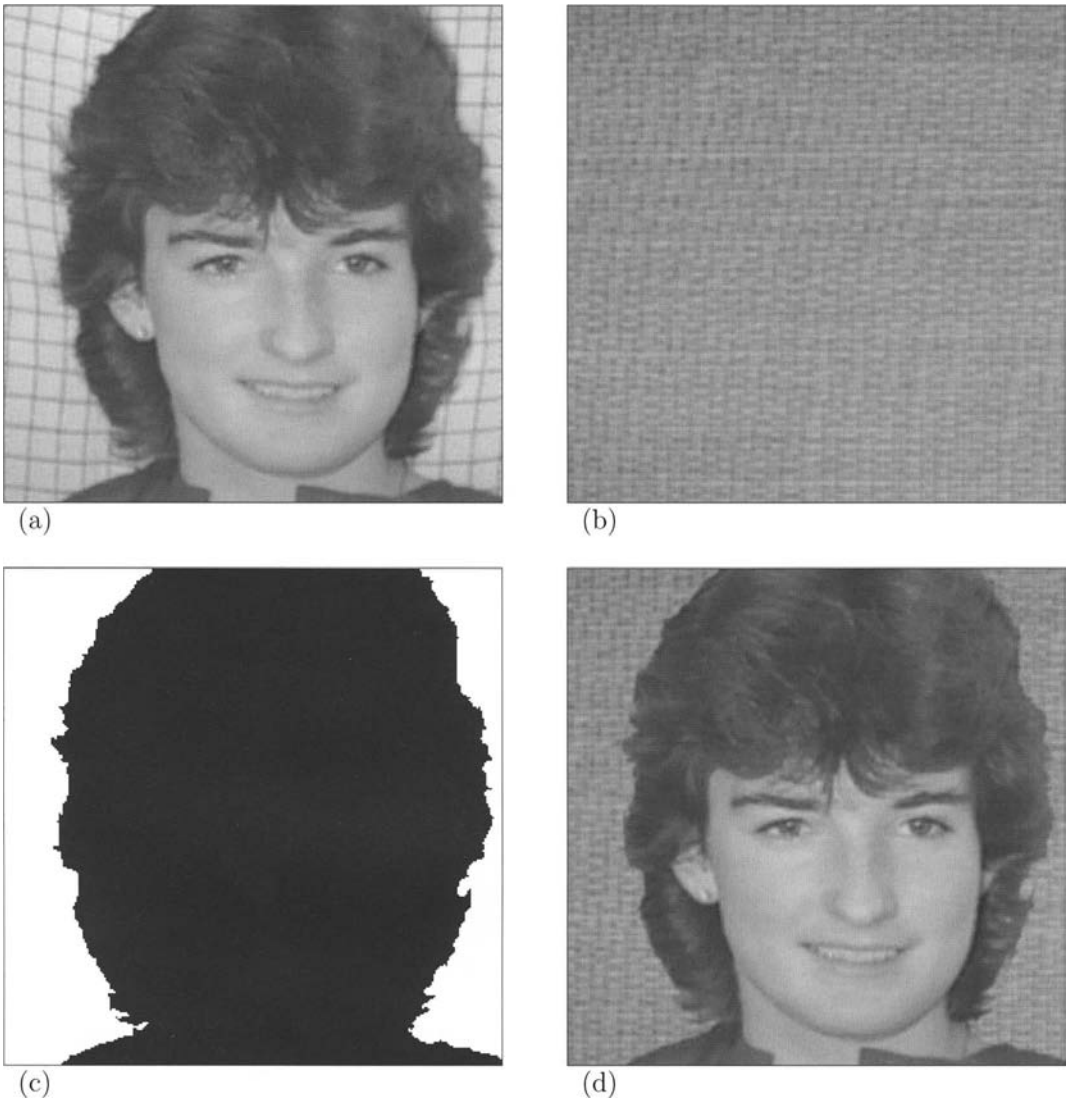


Bild 25.3: Beispiel zur Kombination von Bildkanälen. (a) Grauwertbild im ersten Kanal des Eingabebildes. (b) Grauwertbild im zweiten Kanal des Eingabebildes. (c) Maskenbild (logisches Bild) im dritten Kanal des Eingabebildes mit den Grauwerten, 0 und 255. (d) Ergebnis der Verknüpfung.

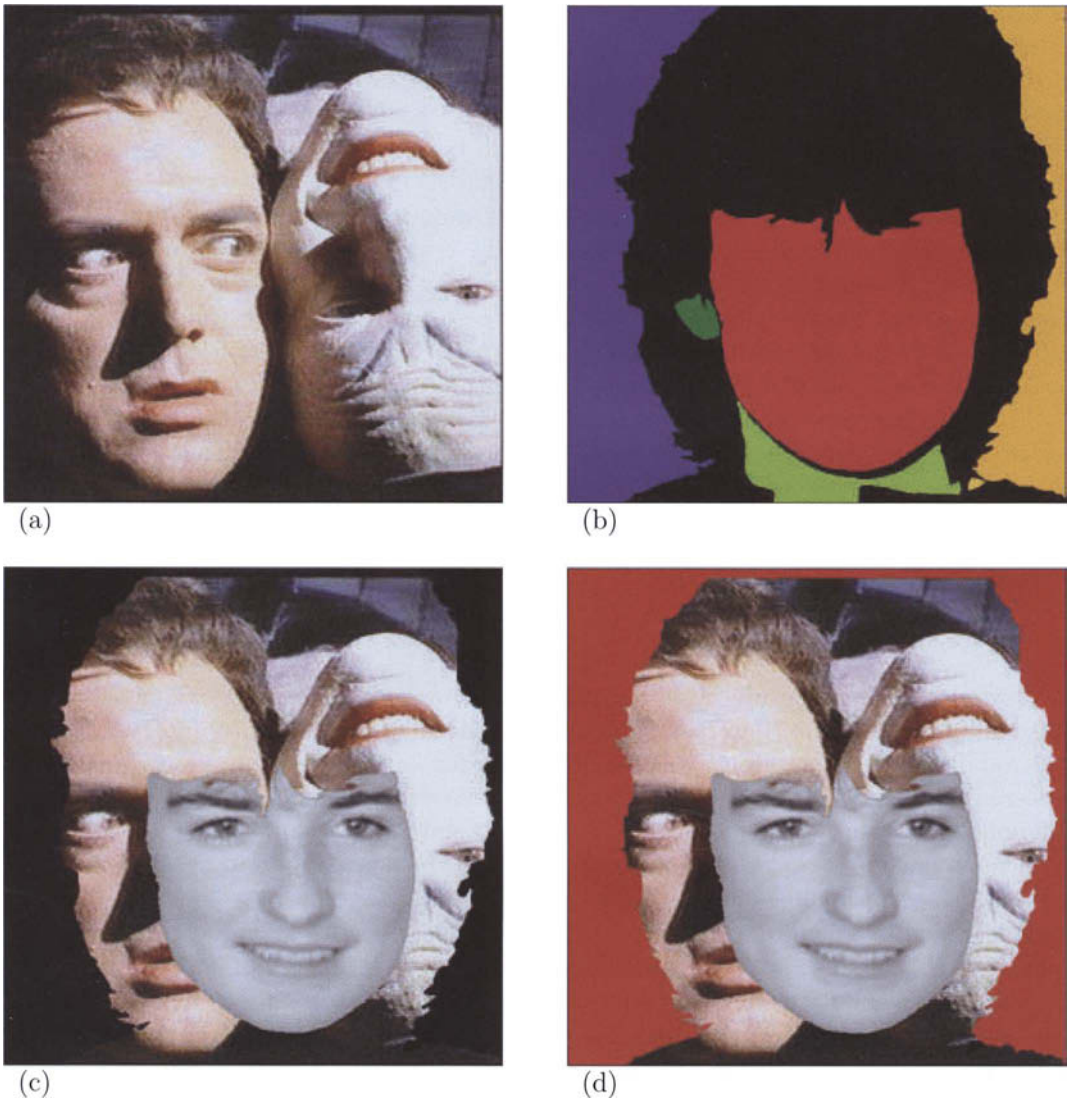


Bild 25.4: Beispiel zur Kombination von Bildkanälen. (a) Originalbild. (b) Maskenbild, in dem Teilflächen mit unterschiedlichen Grauwerten codiert sind. Diese Grauwerte sind hier mit Pseudofarben dargestellt. (c) Beispiel einer Bildkombination. (d) Hier sind die Hintergrundflächen zusätzlich mit einem roten Farbton aufgefüllt.

werden. Durch die starke Korrelation der Grauwerte der beiden Kanäle kann zu jedem Grauwert im ersten Kanal ein mehr oder weniger breites Grauwertintervall im zweiten Kanal angegeben werden. Der Extremfall wäre die lineare Abhängigkeit der beiden Kanäle. In diesem Fall könnte zu jedem Grauwert im ersten Kanal der zugehörige Grauwert im zweiten Kanal exakt angegeben werden. Der zweite Kanal wäre dann redundant. Aber nicht nur bei linearer Abhängigkeit, sondern auch bei starker Korrelation, beinhalten die beiden Kanäle redundante Grauwertinformationen.

Die Hauptkomponententransformation besteht nun in einer Drehung der Koordinatenachsen der beiden Kanäle, und zwar so, dass die erste Achse in Richtung der größten Streuung der Grauwertkombinationen von Kanal 0 und 1 liegt und die zweite senkrecht dazu. Die neuen Koordinatenachsen heißen die *erste* und die *zweite Hauptkomponente*. In Bild 25.5-d sind sie angedeutet. Die neuen Achsen werden, um negative Grauwerte zu vermeiden, jeweils um den Wert 127 verschoben.

In diesem zweidimensionalen Fall besteht die Hauptkomponententransformation aus einer Drehung (und Verschiebung) des Merkmalskoordinatensystems um einen Winkel α, der die Richtung der stärksten Streuung angibt. Die Transformationsmatrix ist die bekannte Drehmatrix:

$$\begin{pmatrix} \cos\alpha & \sin\alpha \\ -\sin\alpha & \cos\alpha \end{pmatrix}. \tag{25.6}$$

Nach der Drehung hat die erste Hauptkomponente die Richtung der maximalen Streuung, während die zweite Hauptkomponente in der Regel eine geringe Streuung aufweist, so dass auf diesen neuen zweiten Kanal verzichtet werden kann, ohne allzuviel Information zu verlieren. Bild 25.5-e zeigt das gedrehte Koordinatensystem der ersten und zweiten Hauptkomponente zu den n Bildkanälen von Bild 25.5-b und 25.5-c. Um negative Grauwerte zu vermeiden, müssen bei praktischen Anwendungen die beiden Hauptkomponenten so gelegt werden, dass der jeweilige mittlere Grauwert gleich 127 ist.

Die Bilder 25.5-f und 25.5-g zeigen die erste und die zweite Hauptkomponente. Man sieht deutlich, dass die zweite Hauptkomponente nur mehr geringe Bildinformation enthält.

Nach dieser einführenden Darstellung am zweidimensionalen Fall soll nun ein N-kanaliges Bild $\mathbf{S}_e = (s_e(x,y,n)), n = 0,1,...,N-1$ vorausgesetzt werden. Die Ableitung der Hauptkomponententransformation wird jetzt anhand statistischer Methoden durchgeführt. Die Bildpunkte

$$\mathbf{s}_e(x,y) = (g_0, g_1, ..., g_{N-1})^T$$

von $\mathbf{S}_e$ werden als Realisierungen einer N-dimensionalen Zufallsvariablen

$$\mathbf{x} = (X_0, X_1, ..., X_{N-1})^T$$

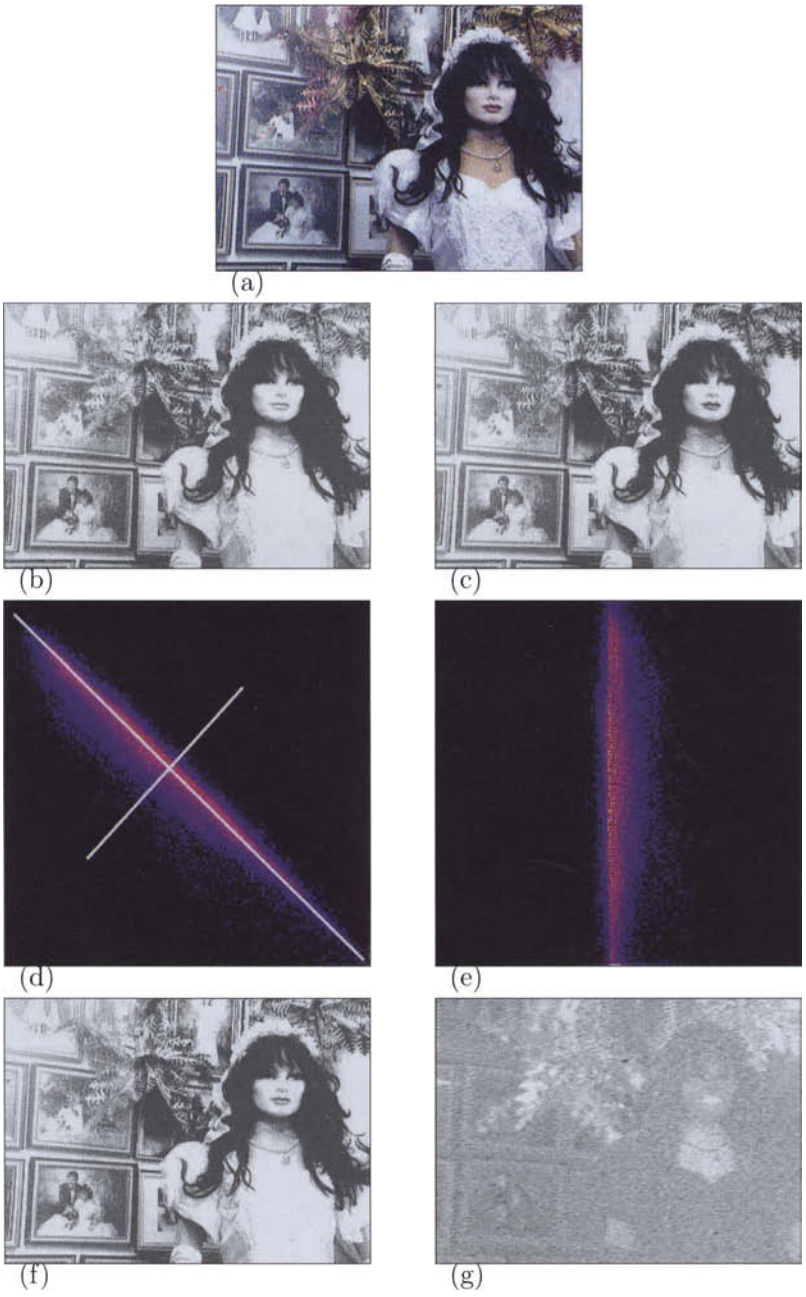


Bild 25.5: Zweidimensionale Grauwertverteilung eines zweikanaligen Bildes, dessen Kanäle stark korreliert sind. (a) Farbbild. (b) Roter Kanal. (c) Blauer Kanal. (d) zweidimensionales Histogramm roter/blauer Kanal. (e) zweidimensionales Histogramm, erste und zweite Hauptkomponente. (f) Erste Hauptkomponente. (g) Zweite Hauptkomponente.

aufgefasst. Zur Herleitung wird o.B.d.A. angenommen, dass der Erwartungswert $E(\mathbf{x}) = \mathbf{0}$ ist und $\mathbf{x}$ die Kovarianzmatrix $\mathbf{C}$ besitzt. Die Verteilung der Zufallsvariablen $\mathbf{x}$ ist im Folgenden nicht wesentlich.

Die Hauptkomponenten sind Linearkombinationen der Zufallsvariablen $X_0, X_1, ..., X_{N-1}$, wobei an die Streuungen spezielle Anforderungen gestellt werden. Die erste Hauptkomponente ist diejenige Linearkombination $\mathbf{a}^T\mathbf{x}$ mit

$$\mathbf{a} = (a_0, a_1, ..., a_{N-1}) \text{ und } \mathbf{a}^T\mathbf{a} = 1 \text{ (normiert)},$$

deren Streuung maximal ist. Für die Streuung von Linearkombination $\mathbf{a}^T\mathbf{x}$ ergibt sich:

$$E\big((\mathbf{a}^T\mathbf{x})^2\big) = E\big(\mathbf{a}^T\mathbf{x}\mathbf{a}^T\mathbf{x}\big) = E\big(\mathbf{a}^T\mathbf{x}\mathbf{x}^T\mathbf{a}\big) = \mathbf{a}^T E\big(\mathbf{x}\mathbf{x}^T\big)\mathbf{a} = \mathbf{a}^T\mathbf{C}\mathbf{a}. \tag{25.7}$$

Die Nebenbedingung $\mathbf{a}^T\mathbf{a} = 1$ wird durch einen Ansatz mit einem Lagrange'schen Multiplikator berücksichtigt:

$$\Phi = \mathbf{a}^T\mathbf{C}\mathbf{a} - \lambda(\mathbf{a}^T\mathbf{a} - 1). \tag{25.8}$$

Die Streuung von Φ ist maximal, falls die Ableitung

$$\frac{\partial\Phi}{\partial\mathbf{a}} = 2\mathbf{C}\mathbf{a} - 2\lambda\mathbf{a} = \mathbf{0} \text{ oder } (\mathbf{C} - \lambda\mathbf{I})\mathbf{a} = \mathbf{0}. \tag{25.9}$$

λ ist also ein Eigenwert der Kovarianzmatrix $\mathbf{C}$. Das charakteristische Polynom

$$\det(\mathbf{C} - \lambda\mathbf{I}) = 0 \tag{25.10}$$

ergibt N Eigenwerte $\lambda_0 \geq \lambda_1 \geq ... \geq \lambda_{N-1}$. Die Streuung von $\mathbf{a}^T\mathbf{x}$ kann mit diesen Voraussetzungen durch die Eigenwerte λ_i ausgedrückt werden. Dazu wird (25.9) von links mit $\mathbf{a}^T$ multipliziert:

$$\begin{aligned} &\mathbf{a}^T(\mathbf{C} - \lambda\mathbf{I})\mathbf{a} = 0, \\ &\mathbf{a}^T\mathbf{C}\mathbf{a} - \lambda\mathbf{a}^T\mathbf{a} = 0, \\ &E\big((\mathbf{a}^T\mathbf{x})^2\big) = \mathbf{a}^T\mathbf{C}\mathbf{a} = \lambda. \end{aligned} \tag{25.11}$$

Die Streuung von $\mathbf{a}^T\mathbf{x}$ ist also maximal, wenn λ den Wert des größten Eigenwertes λ_0 annimmt und der Koeffizientenvektor $\mathbf{a}$ der zu λ_0 gehörige Eigenvektor $\mathbf{a}_0$ ist. Die erste Hauptkomponente ist dann die Linearkombination $\mathbf{a}_0^T\mathbf{x}$ mit der Streuung λ_0.

Die Bestimmung der zweiten Hauptkomponente ergibt sich aus der Forderung, dass $\mathbf{a}_1^T\mathbf{x}$ maximale Streuung unter all jenen Linearkombinationen besitzt, die mit $\mathbf{a}_0^T\mathbf{x}$ unkorreliert sind. Die weitere Berechnung erfolgt sinngemäß wie oben.

Es zeigt sich letztlich, dass die N Hauptkomponenten der Zufallsvariablen $\mathbf{x}$ die Linearkombinationen

$$\mathbf{a}_0^T\mathbf{x}, \quad \mathbf{a}_1^T\mathbf{x}, \quad ..., \quad \mathbf{a}_{N-1}^T\mathbf{x} \tag{25.12}$$

sind, wobei $\mathbf{a}_i$ der zum Eigenwert $\lambda_i(\lambda_0 \geq \lambda_1 \geq ... \geq \lambda_{N-1})$ gehörige Eigenvektor der Kovarianzmatrix $\mathbf{C}$ ist. Die Streuung von $\mathbf{a}_i^T\mathbf{x}$ ist $lambda_i$.

Die praktische Berechnung wird nun wie folgt durchgeführt:

- Berechnung der Kovarianzmatrix $\mathbf{C}$ des Bildes $\mathbf{S}_e$.
- Berechnung der Eigenwerte und der Eigenvektoren von $\mathbf{C}$.
- Ordnen der Eigenwerte, so dass gilt: $\lambda_0 \geq \lambda_1 \geq ... \geq \lambda_{N-1}$. Die Matrix $\mathbf{V}$ der Eigenvektoren wird sinngemäß umgeordnet.
- Für alle Bildpunkte $\mathbf{s}_e(x,y) = (g_0, g_1, ..., g_{N-1})^T$ von $\mathbf{S}_e$ wird die neue Grauwertkombination $\mathbf{s}_a(x,y)$ berechnet gemäß: $\mathbf{s}_a(x,y) = \mathbf{V} \cdot \mathbf{s}_e(x,y)$. $\mathbf{s}_a(x,y)$ ist ein Bildpunkt des hauptkomponententransformierten Bildes $\mathbf{S}_a$.

Die Histogramme von Bild 25.6 sind aus dem Rot-, Grün- und Blau-Kanal des Farbbildes 25.5-a berechnet. Darunter sind die Histogramme der drei Hauptkomponenten abgebildet. Man sieht deutlich die abnehmende Streuung der Histogramme.

Abschließend noch einige Bemerkungen zur Hauptkomponententransformation. In diesem Abschnitt wurde sie anhand eines mehrkanaligen Bildes $\mathbf{S}_e = (s_e(x,y,n), n = 0,1,...,$ $N-1$ erläutert. Sie bringt hier eine Reduktion der Kanäle, da sie das neue Merkmalskoordinatensystem so legt, dass es möglichst redundanzfrei ist. Es ist allerdings zu beachten, dass in manchen Fällen durch diese Drehung des Merkmalskoordinatensystems für spezielle abgebildete Objekte die charakteristischen Eigenschaften verloren gehen oder verschlechtert werden können. Die Kovarianzmatrix $\mathbf{C}$ kann auch nur für Bildausschnitte von besonderem Interesse berechnet werden. In diesem Fall sind die Transformierten an die jeweiligen Bildausschnitte angepasst.

Eine andere Anwendungsmöglichkeit besteht in der Elimination redundanter Bildinformation im Ortsbereich. Hier wird dann die Kovarianzmatrix zu bestimmten Umgebungen (z.B. $8 \cdot 8$ Bildpunkte) berechnet. Allerdings ergeben sich hier sehr große Matrizen, sodass diese Art der Anwendung oft nur von theoretischer Bedeutung ist.

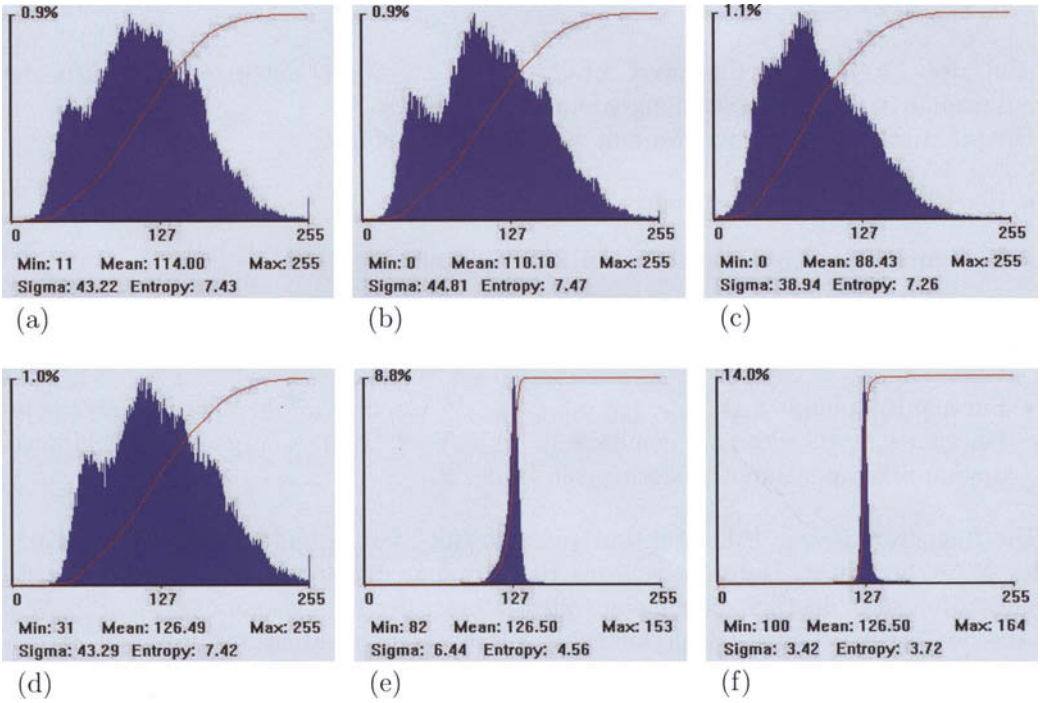


Bild 25.6: Zweidimensionale Grauwertverteilung eines zweikanaligen Bildes, dessen Kanäle stark korreliert sind. (a) - (c): Histogramm zum Rot- , Grün- und Blau-Kanal des Farbbildes 25.5-a. (d) - (f): Histogramme der drei Hauptkomponenten. Man sieht deutlich die abnehmende Streuung der Histogramme.

Kapitel 26

Merkmale aus Bildfolgen

26.1 Anwendungen

In diesem Kapitel werden Bildfolgen $\mathbf{S}_e = (s_e(x, y, t)), t = 0, 1, ..., T-1$, vorausgesetzt. Es kann sich dabei z.B. um eine Bildsequenz handeln, die von einer Videokamera erzeugt wird, die fest installiert ist und einen definierten Bildausschnitt liefert. Wenn es sich um Farbbildfolgen oder Multispektralbildfolgen handelt, kann die Darstellung der Szene $\mathbf{S}_e$ noch um die Kanalparameter erweitert werden: $\mathbf{S}_e = (s_e(x, y, n, t)), n = 0, 1, ..., N-1, t = 0, 1, ..., T-1$. In der folgenden Darstellung werden jedoch nur einkanalige Bildfolgen behandelt.

Bei leicht verrauschten Bildfolgen kann durch eine Mittelung entlang der Zeitachse das Rauschen abgeschwächt werden. Wenn sich in der Bildfolge etwas bewegt, ergeben sich durch die Mittelung Artefakte. Bei manchen Anwendungen, z.B. in der (Radio-)Astronomie können derartige Störungen durch ein Nachführen des Videosensors vermieden werden. Auch die Differenz zwischen einzelnen Teilbildern einer Bildfolge wird in der Praxis, z.B. in der Medizin, verwendet.

Bei medizinischen Anwendungen sind die Bildakkumulation und die Differenzbildung wichtige Verarbeitungsschritte von Bildfolgen. Die Detektion von bewegten Bildausschnitten lässt sich ebenfalls mit den Bildfolgen erzielen.

Ein wichtiger Bereich ist die Erfassung von zeitlichen Veränderungen in einer Bildfolge, die z.B. durch Bewegungen im Beobachtungsgebiet und/oder durch Bewegungen des Videosensors entstanden sind. Hier kann man z.B. zur Überwachung von Sicherheitsbereichen Bilder erzeugen, in denen abgebildet ist, wo sich etwas verändert oder bewegt hat. Ein nächster Schritt ist die Darstellung von Bewegungen und die qualitative Erfassung von Bewegungen. Anwendungsbeispiele gibt es hier aus dem Bereich der Bilddatencodierung bei den MPEG-Verfahren, bei denen Bewegungsschätzung und Bewegungskompensation zur effektiven Codierung von Bildfolgen eingesetzt werden.

26.2 Akkumulation und Differenz

Ist der Videosensor fest installiert und bewegt sich im Bildausschnitt nichts, so sind die Einzelbilder der Bildfolge im Wesentlichen alle gleich, bis auf etwaige Helligkeitsveränderungen und auf das Rauschen, das durch die Aufnahmebedingungen in die Bilddaten integriert wird. Eine Mittelung der zeitlich aufeinanderfolgenden Bilder liefert dann ein Bild, in dem die Rauscheinflüsse geringer sind (Gauß'sches Rauschen mit Erwartungswert 0):

$$s_a(x,y) = \frac{1}{T}\sum_{t=0}^{T-1} s_e(x,y,t). \tag{26.1}$$

Das gemittelte Bild $\mathbf{S}_a$ ist in der Bildqualität verbessert, wobei die Mittelung in (26.1) die Schärfe des Bildes nicht beeinflusst hat.

Auch die Differenz von Teilbildern einer Bildfolge zu den Zeitpunkten t und $t+1$ lässt mehrere Anwendungen zu:

$$s_a(x,y) = s_e(x,y,t) - s_e(x,y,t+1) + c. \tag{26.2}$$

Die Bildfolge 26.1 zeigt ein Beispiel dazu. Die ersten beiden Bilder sind die Teilbilder der Bildfolge zu den Zeitpunkten t und $t+1$. Bild 26.1-c ist das Differenzbild von Bild 26.1-a und Bild 26.1-b. Zu den Werten der Differenz wurde die Konstante $c = 127$ addiert, um negative Grauwerte zu vermeiden. Man sieht deutlich den neu verdeckten und den frei werdenden Hintergrund. Da die Videokamera mit der Hand gehalten wurde, war sie nicht ganz unbewegt. Dadurch ergeben sich auch im Bildhintergrund geringe Differenzsignale. Bild 26.1-d entstand sinngemäß wie Bild 26.1-c, jedoch wurde als zweites Bild der Differenz das Bild zum Zeitpubkt $t+7$ verwendet.

In Kombination mit der zeitlichen Mittelwertbildung wird die Differenzenbildung bei medizinischen Fragestellungen häufig angewendet. So wird z.B. bei der Angiocardiografie (Untersuchung des Herzens) die folgende Versuchsanordnung gewählt:

- Für die Zeitpunkte $t_0, t_1, ..., t_k$ wird eine Bildsequenz ohne Kontrastmittel zur Reduktion des Rauschens aufsummiert:

$$s_1(x,y) = \frac{1}{k+1}\sum_{t=0}^{k} s_e(x,y,t). \tag{26.3}$$

- Dann wird ein Kontrastmittel eingespritzt und für die Zeitpunkte $t_{k+1}, t_{k+2}, ..., t_{k+l}$ eine Bildsequenz mit Kontrastmittel aufsummiert:

$$s_2(x,y) = \frac{1}{l-k}\sum_{t=k+1}^{l} s_e(x,y,t). \tag{26.4}$$

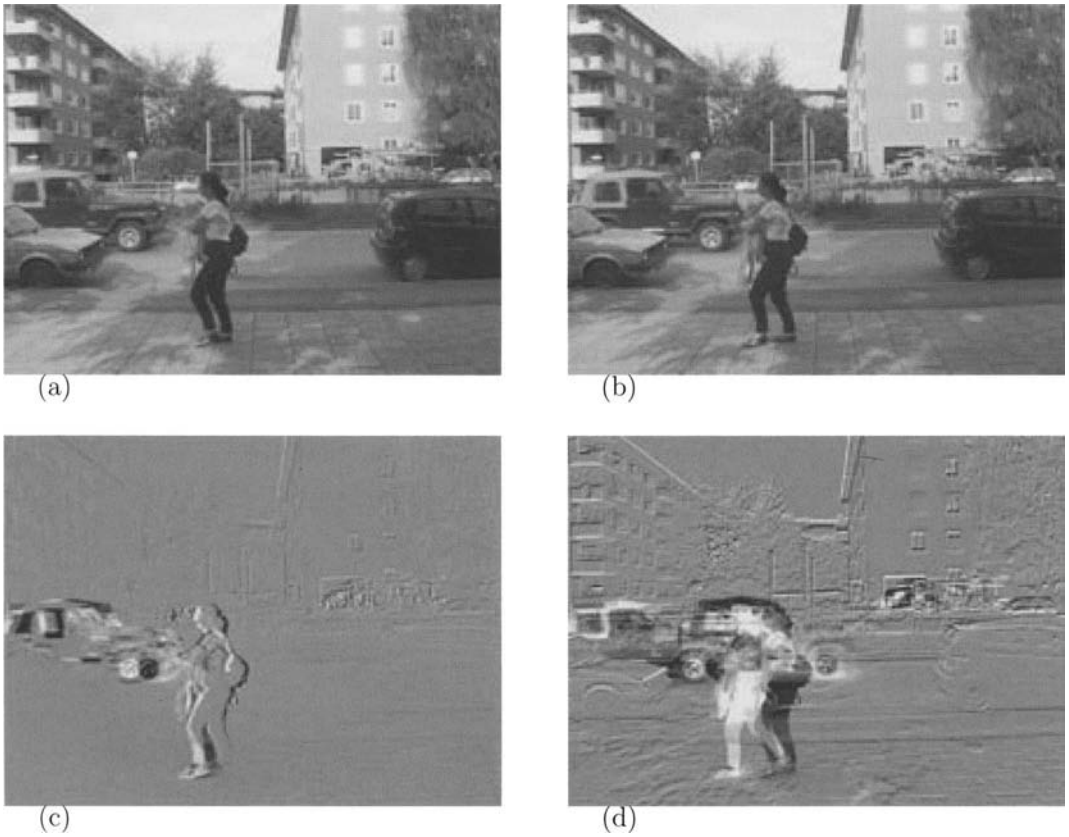


Bild 26.1: Differenz von Teilbildern einer Bildfolge. (a) Bild zum Zeitpunkt t. (b) Bild zum Zeitpunkt $t+1$. (c) Differenzbild von (a) und (b). Zu den Werten der Differenz wurde 127 addiert. Man sieht deutlich den neu verdeckten und den frei werdenden Hintergrund. Da die Videokamera mit der Hand gehalten wurde, war sie nicht ganz unbewegt. Dadurch ergeben sich auch im Bildhintergrund geringe Differenzsignale. (d) Wie Bild (c), jedoch wurde als zweites Bild der Differenz das Bild zum Zeitpunkt $t+7$ verwendet.

- Auch diese Mittelung wird zur Reduktion des Rauschens durchgeführt. Das Differenzbild

$$s_a(x,y) = s_1(x,y) - s_2(x,y) + c \tag{26.5}$$

zeigt dann in der Regel die zu untersuchenden Bildbereiche deutlich (evtl. nach einer abschließenden Kontrastanreicherung), während störender Hintergrund ausgeblendet ist. Allerdings erzeugen bei dieser Vorgehensweise Bewegungen im Untersuchungsgebiet Artefakte.

Eine weitere medizinische Anwendung der Verarbeitung von Bildfolgen ist die bildliche Darstellung des zeitlichen Verlaufs der Ausbreitung eines Kontrastmittels. Voraussetzung ist hier wieder ein unbewegtes Beobachtungsgebiet. Mit der Bildakkumulation wird vor dem Einspritzen des Kontrastmittels begonnen und während der Eingabe des Kontrastmittels und eine bestimmte Zeit anschließend fortgesetzt. Zu bestimmten Zeitpunkten können Zwischenergebnisse abgespeichert werden, sodass schließlich eine Bildfolge mit unterschiedlicher Akkumulationsdauer vorliegt. Wird diese Bildfolge mit einer geeigneten Pseudofarbdarstellung ausgegeben, so ist die unterschiedliche Färbung ein Hinweis auf die zeitliche Ankunft und die Konzentration des Kontrastmittels an bestimmten Stellen.

26.3 Merkmal: Bewegung

Ein weiterer wichtiger Problemkreis bei der Verarbeitung von Bildfolgen ist die *Sichtbarmachung von Bewegungen*, die *Detektion von Bewegungen* und u.U. die *Verfolgung von bewegten Objekten*. Neben militärischen gibt es hier auch viele zivile Anwendungen, etwa die Verfolgung von Werkstücken auf einem Fließband im Rahmen einer industriellen Handhabungsmaschine.

Eine Möglichkeit Bewegungen sichtbar zu machen besteht in der Akkumulation über mehrere Einzelbilder der Bildfolge. In Bild 26.2 wurde z.B. über acht Teilbilder der Bildfolge von Bild 26.1 akkumuliert.

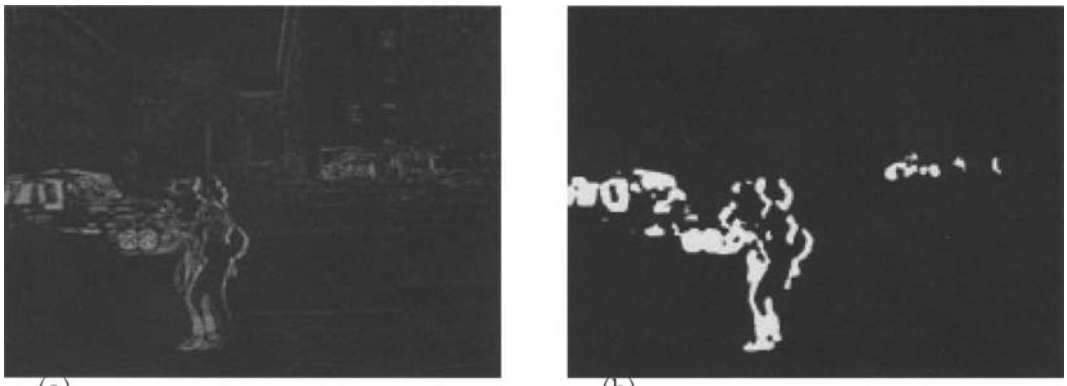
Ein anderer einfacher Lösungsansatz bei Bildfolgen mit einem festen Bildausschnitt, in dem sich ein Objekt bewegt, ist die Berechnung des Differenzbetrags von aufeinanderfolgenden Bildern, etwa zu den Zeitpunkten t und $t+1$, gemäß:

$$s_1(x,y) = |s_e(x,y,t) - s_e(x,y,t+1)|. \tag{26.6}$$

Anschließend kann mit einem passenden Schwellwert z noch eine Binarisierung durchgeführt werden:

$$s_a(x,y) = \begin{cases} 0, & s_1(x,y) < z \\ 1, & \text{sonst.} \end{cases} \tag{26.7}$$

Bild 26.2: Akkumulation über acht Teilbilder der Bildfolge von Bild 26.1.



(a) (b)

Bild 26.3: (a) Differenzbetrag der beiden Teilbilder von Beispiel 26.1. (b) Binarisierung des Differenzbetrags, auf den vor der Schwellwertbildung ein Unschärfeoperator angewendet wurde. Die Bildpunkte mit dem Grauwert 1 sind weiss dargestellt.

In $\mathbf{S}_a$ sind dann Bildbereiche, die von der Bewegung betroffen waren, mit 1 codiert. Damit kann eine Entscheidung getroffen werden, ob sich etwas bewegt hat oder nicht (Bild 26.3).

Die bis jetzt vorgestellten Verfahren zur Verarbeitung von „Bewegung" sind nur dazu geeignet, Bewegungen oder bewegte Ausschnitte darzustellen. Will man „Bewegung" als Merkmal für eine spätere Segmentierung, so muss die Bewegung zahlenmäßig erfasst werden. Man muss somit einem Bildpunkt ein Maß für seine Bewegung zuordnen. Dabei kann man nicht bildpunktweise vorgehen, sondern muss Umgebungen von Bildpunkten (Blöcke) im (x, y)-Ortsbereich betrachten. Wie groß die Blöcke sind, hängt von der gewünschten Genauigkeit ab. Zusätzlich müssen auch Informationen entlang der Zeitachse t ausgewertet werden.

Bei einem Bewegungsschätzer wird z.B. eine Kombination aus Differenzen entlang der Zeitachse t und im Ortsbereich der (x, y)-Koordinaten verwendet:

$$
\begin{aligned}
diff_t(x,y) &= s_e(x,y,t) - s_e(x,y,t+1). \\
td &= \sum_x \sum_y |diff_t|. \\
diff_o(x,y) &= s_e(x,y,t) - s_e(x,y-1,t). \\
od &= \sum_x \sum_y |diff_o|. \\
v &= \frac{td}{od}
\end{aligned}
\tag{26.8}
$$

Die Summation wird über die Bildpunkte des gewählten Bildausschnitts durchgeführt. Der Wert td ist ungefähr proportional zur Geschwindigkeit der Bewegung in diesem Aus-

schnitt. Allerdings ist der Wert noch von der Größe des Ausschnitts abhängig. Aus diesem Grund wird eine Normalisierung durchgeführt und der Wert *od* berechnet. Der Wert *od* hängt von der Größe des Ausschnitts ab, ist aber unabhängig von der darin auftretenden Bewegung. Als Maß für die Geschwindigkeit wird schließlich der Wert v verwendet. Praktische Erfahrungen haben gezeigt, dass v ein gutes Maß ist, wenn die Objektgeschwindigkeit zwischen dem Bild zum Zeitpunkt $t-1$ und dem Bild zum Zeitpunkt t nicht größer als drei Bildpunkte ist.

Ein Beispiel zu diesem Bewegungsmaß zeigt die Bildfolge 26.4. In den beiden oberen Bildern, die jeweils dasselbe Bild zum Zeitpunkt t darstellen, sind zwei unterschiedliche Bildausschnitte markiert. Für diese Ausschnitte wurde das obige Bewegungsmaß berechnet. Die beiden mittleren Bilder sind die Differenzbilder (+127) zwischen den Zeitpunkten t und $t+1$. Man sieht deutlich die Bewegungen. Für den linken Ausschnitt ergibt sich $v = 0.05$ und für den rechten, in dem sich ein Objekt bewegt hat, $v = 1.39$. Die beiden unteren Bilder sind die Differenzbilder (+127) zwischen den Zeitpunkten t und $t+2$. Hier ergibt sich sinngemäß für den Ausschnitt im linken Bild $v = 0.05$ und für den Ausschnitt im rechten Bild $v = 2.08$. Der Vergleich zeigt, dass sich bei größeren Bewegungen auch größere Werte berechnen. Man könnte nun die Werte von v geeignet skalieren und in das Grauwertintervall G abbilden. Das ergibt ein Bild, in dem die Werte der Bildpunkte Maßzahlen für die Bewegung sind.

26.4 Differentielle Ermittelung von Verschiebungsvektoren

Ein aufwändiger Ansatz zur Erfassung von Bewegungen ist die Ermittlung von *Verschiebungsvektorfeldern*. Der Ansatz beschränkt sich zunächst auf die Erkennung einer reinen Translation parallel zur Bildebene. Er kann aber, wenn auch eingeschränkt, auf andere Bewegungssituationen angewendet werden. Als Zielsetzung sind die Verschiebungsvektoren von zwei aufeinanderfolgenden Bildern (Zeitpunkte t und $t+1$) zu bestimmen. Eine vorausgesetzte Annahme ist, dass sich Bildausschnitte gleichmäßig verschieben und somit alle Bildpunkte des Ausschnitts denselben Verschiebungsvektor besitzen. Das trifft bei natürlichen Bildfolgen nicht zu, sodass daraus zum Teil auch die Fehler des Verfahrens zu erklären sind. Trotzdem ist die Berechnung von Verschiebungsvektorfeldern eine sehr effektive Methode, um von Bewegungen sowohl die Richtung als auch den Betrag der Bewegung näherungsweise zu ermitteln. Zur Berechnung der Verschiebungsvektoren werden im Folgenden zwei Verfahren vorgestellt.

In diesem Abschnitt wird die differentielle Methode zur Ermittlung der Verschiebungsvektoren beschrieben.

Zur Berechnung wird zunächst eine Differenzfunktion $d(x,y)$ gebildet:

$$d(x,y) = s(x,y,t+1) - s(x,y,t) = s(x+\Delta x, y+\Delta y, t) - s(x,y,t). \tag{26.9}$$

Diese Umformung ist möglich, da gilt $s(x,y,t+1) = s(x+\Delta x, y+\Delta y, t)$, wenn $(\Delta x, \Delta y)$ der Verschiebungsvektor des bewegten Objektes ist, dessen Helligkeit sich bei der Verschie-

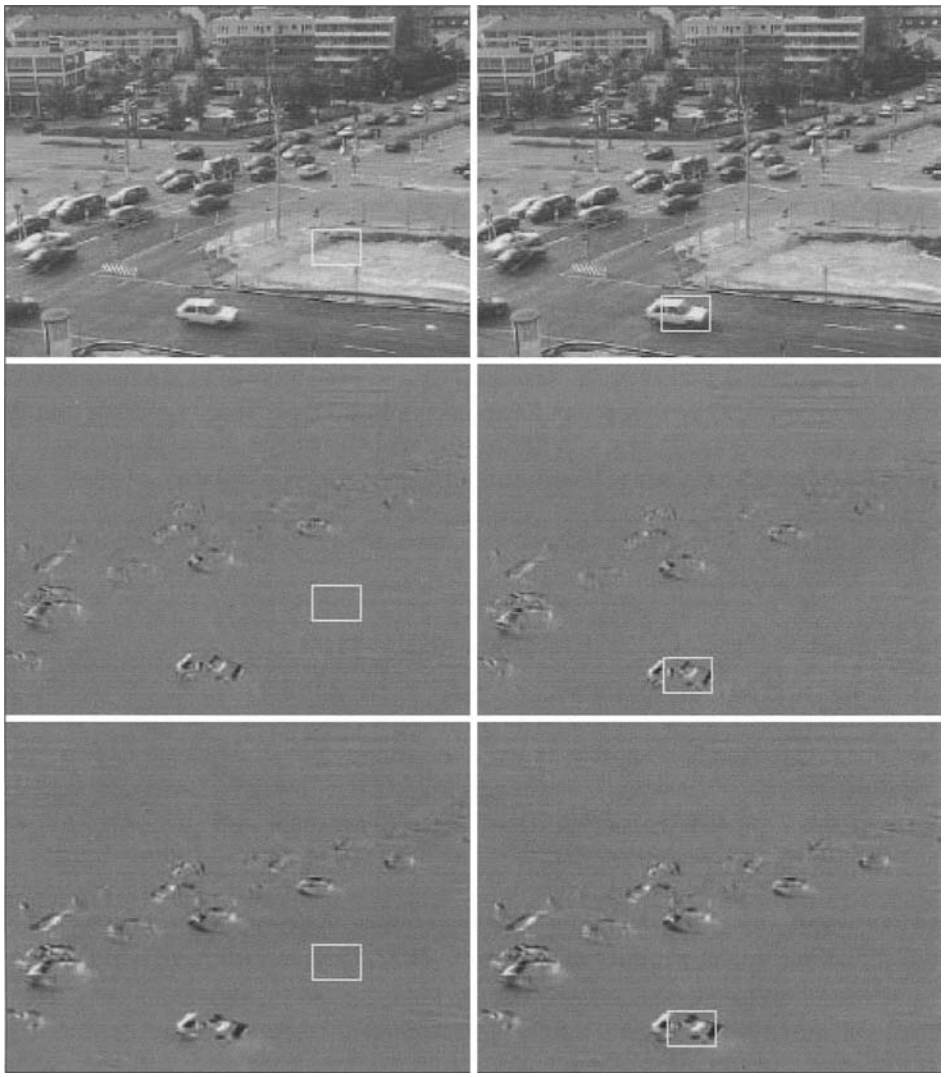


Bild 26.4: Beispiel für ein Maß zur Bewegungsschätzung. In den beiden oberen Bildern sind zwei unterschiedliche Bildausschnitte markiert. Die beiden mittleren Bilder sind die Differenzbilder (+127) zwischen den Zeitpunkten t und $t+1$. Für den linken Ausschnitt ergibt sich $v = 0.05$ und für den rechten, in dem sich ein Objekt bewegt hat, $v = 1.39$. Die beiden unteren Bilder sind die Differenzbilder (+127) zwischen den Zeitpunkten t und $t+2$. Hier ergibt sich sinngemäß $v = 0.05$ und $v = 2.08$.

bung nicht geändert hat. Eine Taylorreihenentwicklung von (26.9) gestattet die weitere Umformung:

$$\begin{aligned} d(x,y) &= s(x,y,t) + \Delta x \frac{\partial s(x,y,t)}{\partial x} + \Delta y \frac{\partial s(x,y,t)}{\partial y} - s(x,y,t) + E = \\ &= \Delta x \frac{\partial s(x,y,t)}{\partial x} + \Delta y \frac{\partial s(x,y,t)}{\partial y} + E. \end{aligned} \tag{26.10}$$

Dabei sind in E Glieder der Taylorreihenentwicklung mit höherer Ordnung zusammengefasst, die weggelassen werden. Es wird nun angenommen, dass für einen Bildausschnitt mit $m \cdot m$ Bildpunkten die Verschiebungsvektoren näherungsweise gleich sind. Dann gilt für alle Bildpunkte in diesem Bereich die folgende Beziehung:

$$\begin{pmatrix} \frac{\partial s(x-k,y-l,t)}{\partial x} & \frac{\partial s(x-k,y-l,t)}{\partial y} \\ \cdots & \cdots \\ \frac{\partial s(x+k,y+l,t)}{\partial x} & \frac{\partial s(x+k,y+l,t)}{\partial y} \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} d(x-k,y-l) \\ \cdots \\ d(x+k,y+l) \end{pmatrix}. \tag{26.11}$$

Die Indizes k und l laufen dabei über die Bildpunktpositionen des jeweiligen Ausschnittes. Aus (26.11) können jetzt die Verschiebungskomponenten Δx und Δy für den entsprechenden Bildausschnitt mit Hilfe der Ausgleichsrechnung näherungsweise bestimmt werden und stehen dann z.B. als Eingangsgrößen für eine Nachführelektronik zur Verfügung. Da die Berechnung für jedes Fenster getrennt erfolgt, können mit dieser Methode auch sich unterschiedlich bewegende Objekte erkannt werden, sofern sie in verschiedenen Fenstern liegen. Durch den gewählten Ansatz wird nur eine lineare Variation des Grauwertes mit den Bildpunktkoordinaten zur Berechnung einer Verschiebung berücksichtigt. Dadurch entstehen Ungenauigkeiten an Kanten, die z.B. dann zu falschen Verschiebungsvektoren führen, wenn die Übergangszone zwischen dem Objektinneren und dem Hintergrund schmaler ist als die beobachtete Verschiebung des Objektes.

Die Bildfolge 26.5 zeigt ein Beispiel. Die Bilder 26.5-a und 26.5-b sind zwei aufeinander folgende Bilder einer Bildfolge, die aus dem Frontfenster eines nach vorne fahrenden Autos aufgenommen wurde. Die Bilder 26.5-c und 26.5-d sind die Δx- und Δy- Komponenten der Verschiebungsvektoren als Grauwerte codiert, die für $15 \cdot 15$ Bildpunkte berechnet wurden. Zur besseren Beurteilung wurden in den Bildern 26.6-a und 26.6-b die Verschiebungsvektoren grafisch eingezeichnet. Der weiße Punkt markiert den Fußpunkt eines Verschiebungsvektors, die schwarzen Pixel zeigen die Länge und die Richtung.

26.5 Ermittelung von Verschiebungsvektoren mit Blockmatching

Bei den meisten praktischen Anwendungen (z.B. dem MPEG-Verfahren) werden die Verschiebungsvektoren mit der Methode des *Blockmatching* berechnet. Dazu wird wie folgt vorgegangen:

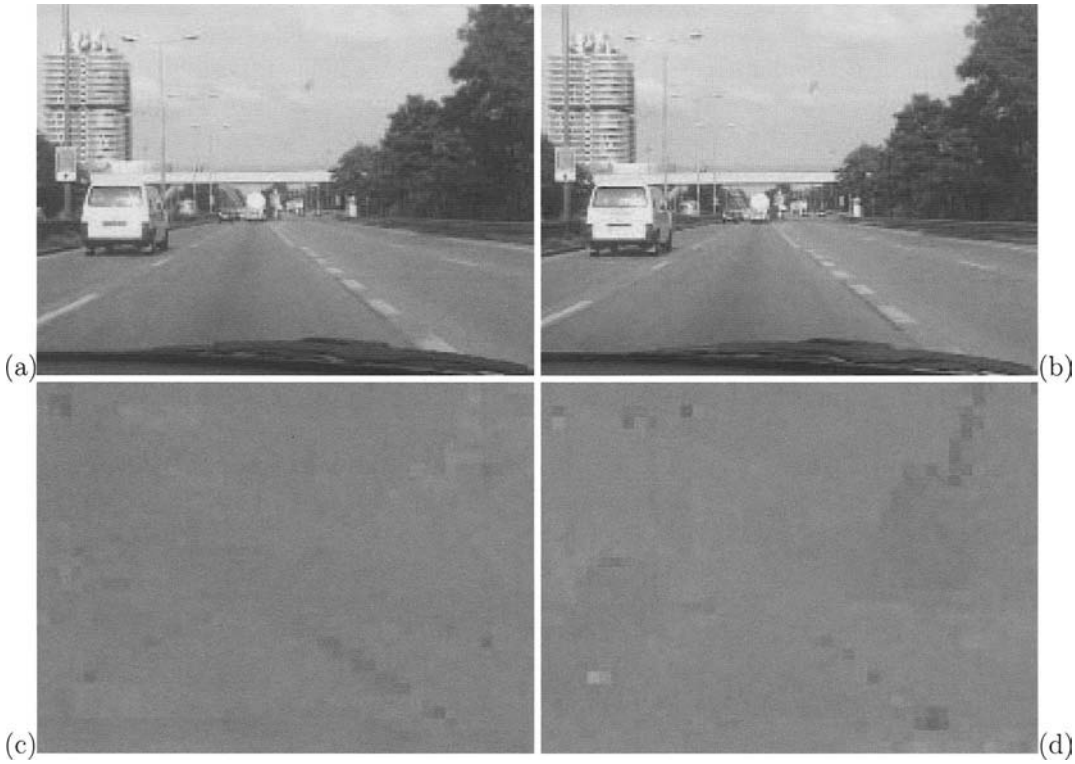


Bild 26.5: Berechnung von Verschiebungsvektorfelderen mit der differentiellen Methode. (a) und (b): Bilder zu den Zeitpunkten t und $t+1$ einer Bildfolge, die aus dem Frontfenster eines nach vorne fahrenden Autos aufgenommen wurde. (c) und (d): Δx- und Δy-Komponenten der Verschiebungsvektoren als Grauwerte codiert, die für $15 \cdot 15$ Bildpunkte berechnet wurden.

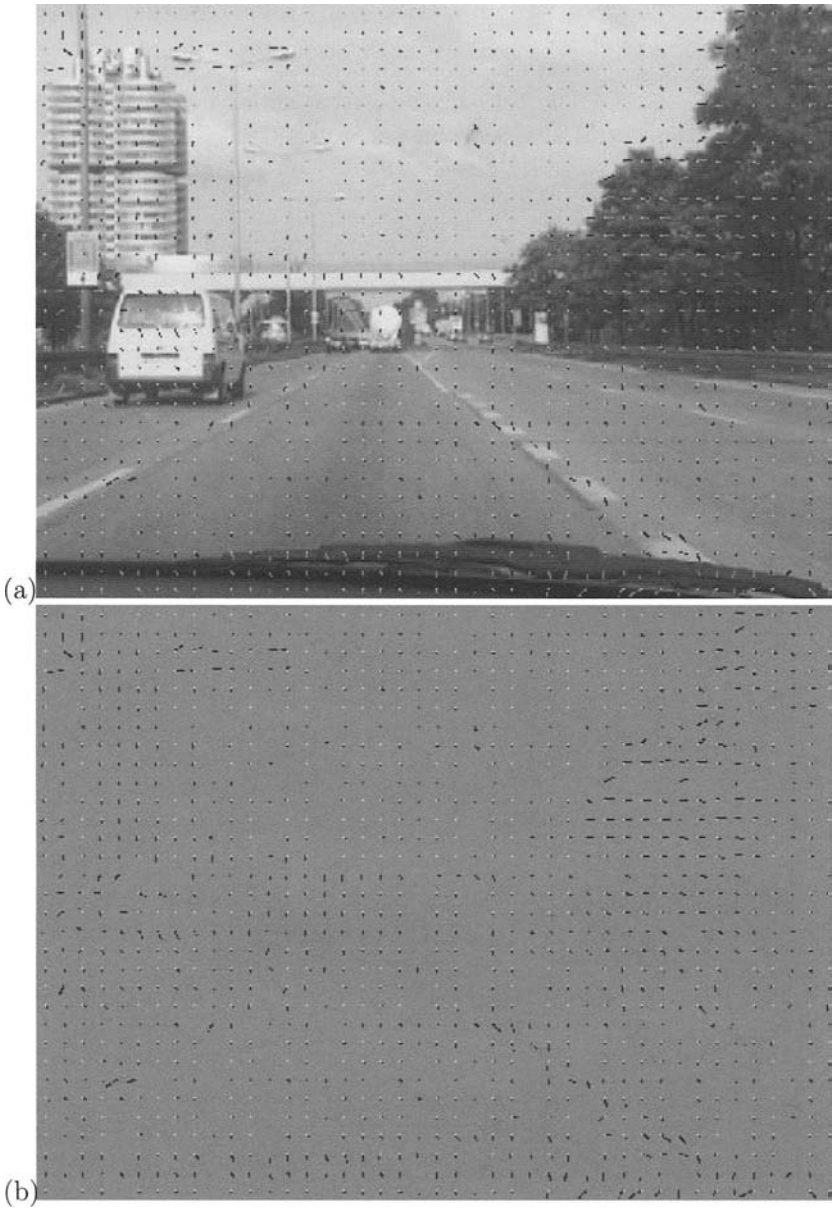


Bild 26.6: Berechnung von Verschiebungsvektorfeldern mit der differentiellen Methode. (a) Die Verschiebungsvektoren wurden hier in das erste Bild eingeblendet. Der weiße Punkt markiert den Fußpunkt eines Verschiebungsvektors, die schwarzen Pixel zeigen die Länge und die Richtung. (b) Hier sind nur die Verschiebungsvektoren gezeigt.

- Die Verschiebungsvektoren werden zu den Bildern zum Zeitpunkt t (Frame t) und $t+1$ der Bildfolge berechnet.
- Das Bild zum Zeitpunkt t wird blockweise verarbeitet. Es wird versucht, den Bildausschnitt $block(m,n)$ des Frames t im Frame $t+1$ zu lokalisieren. Die Parameter m und n laufen dabei über die Größe p Zeilen und q Spalten der Umgebung (des Blocks, z.B. $p \cdot q = 8 \cdot 8$ Bildpunkte).
- Im Frame $t+1$ wird dazu in einer $umgebung(m,n)$ der $block(m,n)$ verschoben und berechnet, in welche Position er am besten passt. Die $umgebung(m,n)$ hat eine Größe von $(p+2c) \cdot (q+2c)$ Bildpunkten, wobei der Parameter c dabei eine Konstante ist, die die maximal erlaubte Verschiebung festlegt.
- Als Matching-Maß werden verwendet:

$$d_1(i,j) = \frac{1}{pq} \sum_{m=0}^{p-1} \sum_{n=0}^{q-1} \Big(block(m,n) - umgebung(m+i, n+j) \Big)^2 \tag{26.12}$$

oder

$$d_2(i,j) = \frac{1}{pq} \sum_{m=0}^{p-1} \sum_{n=0}^{q-1} \Big| block(m,n) - umgebung(m+i, n+j) \Big| \tag{26.13}$$

Die Parameter i und j durchlaufen dabei das Intervall $-c \leq i,j \leq +c$.

- Als passende Position wird die Position (i_{min}, j_{min}) verwendet, für die $d(i_{min}, j_{min})$ minimal ist. Der $block(m,n)$ im Frame t ist somit im Frame $t+1$ um i_{min} Zeilen und j_{min} Spalten verschoben.

Wenn die geschilderte Vorgehensweise verwendet wird, sind bei der Suche $(2c+1)^2$ Positionen zu inspizieren. Man bezeichnet sie als *full search*. Hier ist sichergestellt, dass man ein unter den gegebenen Umständen optimales Ergebnis erzielt. Allerdings muss für jede Position das Matching-Maß (26.12) oder (26.13) berechnet werden, was bei engen Zeitvorgaben zu Problemen führen kann. Aus diesem Grund verwendet man hier auch Verfahren, bei denen zunächst nur an ausgewählten Positionen i,j das Matching-Maß berechnet wird. Dazu wird das Minimum bestimmt. Dann wird um das gefundene Minimum mit kleineren Abständen der Positionen i,j wieder das Matching-Maß berechnet und das Minimum bestimmt. Dieser Vorgang wird mehrstufig durchgeführt. Man hat hier weniger Berechnungen als beim *full search* durchzuführen, es kann jedoch sein, dass man nur ein suboptimales Ergebnis erhält.

Abschließend ein Beispiel: Zu den beiden Teilbildern 26.5-a und 26.5-b wurden die Verschiebungsvektoren mit Blockmatching und *full search* berechnet. Das Ergebnis zeigen die Bilder 26.7-a (Frame t mit Verschiebungsvektoren) und 26.7-b (nur die Verschiebungsvektoren). Als Blockgröße wurden $15 \cdot 15$ Bildpunkte verwendet. Für den Umgebungsparameter

wurde $c = 11$ verwendet. Außerdem wurden nur solche Verschiebungsvektoren dargestellt, deren Betrag über einem Schwellwert von 5 liegt. Man sieht recht gut, dass der Sachverhalt beim Blick durch die Frontscheibe eines nach vorne fahrenden Autos auf einer Stadtautobahn richtig wiedergegeben wird. Allerdings haben sich auch einige Fehler eingeschlichen, die sich in der Praxis nie vermeiden lassen.

Die Berechnung von Verschiebungsvektoren zur Bewegungsschätzung wird in der Bilddatencodierung verwendet: Mit Hilfe der Verschiebungsvektoren wird der Frame $t + 1$ aus dem Frame t durch Verschieben von Blöcken angenähert. Beim Codieren kann das Differenzbild zum originalen Frame $t + 1$ berechnet werden. Das Differenzbild kann, bei guter Bewegungsschätzung und weiteren Verarbeitungsschritten, effektiv entropiecodiert werden (z.B. Huffmancode). Beim Decodieren ist es möglich, mit dem Frame t, den Verschiebungsvektoren und dem Differenzbild den Frame $t + 1$ fehlerfrei zu rekonstruieren. Techniken dieser Art werden, mit verschiedenen Modifikationen, bei den MPEG-Codierungsverfahren für Videodaten eingesetzt.

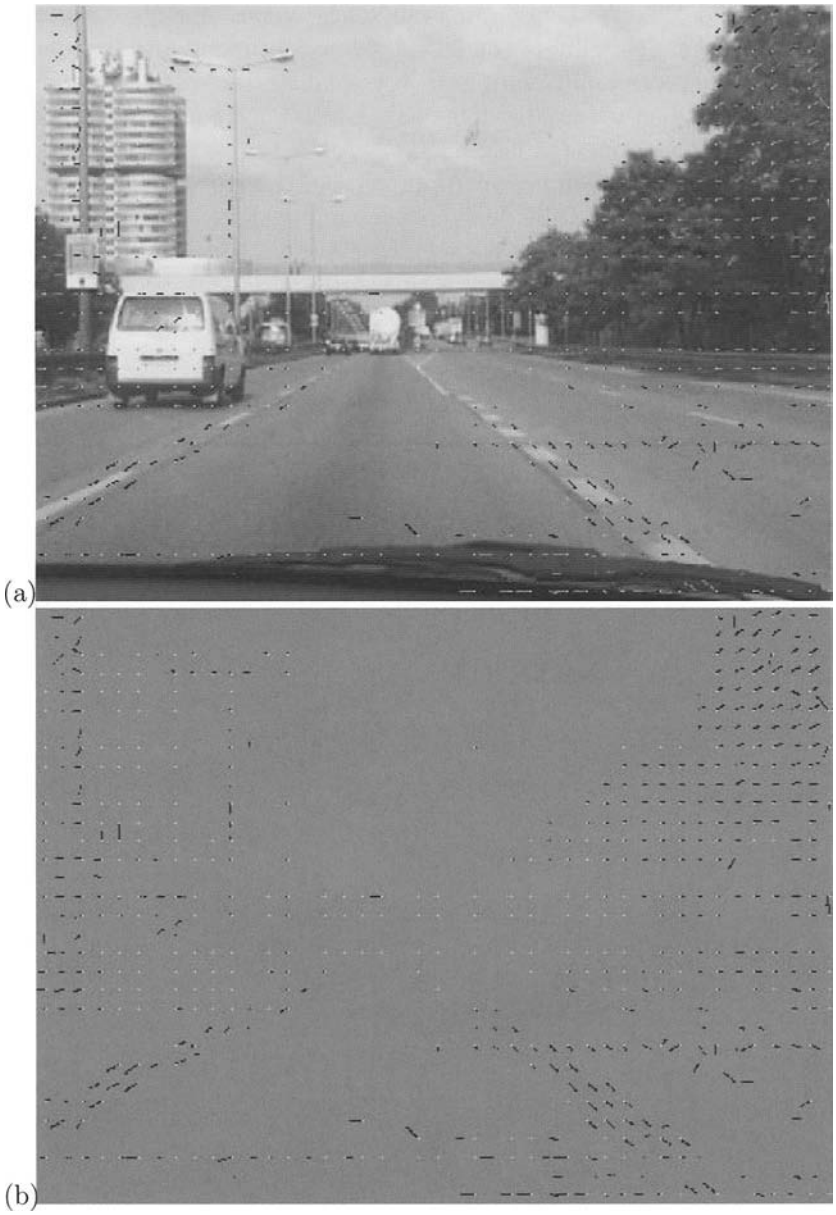


Bild 26.7: Berechnung von Verschiebungsvektorfeldern mit Blockmatching. (a) Die Verschiebungsvektoren wurden hier in das erste Bild eingeblendet. Der weiße Punkt markiert den Fußpunkt eines Verschiebungsvektors, die schwarzen Pixel zeigen die Länge und die Richtung. (b) Hier sind nur die Verschiebungsvektoren gezeigt.

Kapitel 27

Merkmale aus Umgebungen: Texturen

27.1 Anwendungen

Die bis jetzt betrachteten Merkmale wurden für jeden Bildpunkt berechnet, ohne dabei seine Umgebung im Ortsbereich zu berücksichtigen. Häufig ist aber das kennzeichnende Merkmal, das die zu untersuchenden Objekte unterscheidet, eine mehr oder weniger regelmäßige, unterschiedliche Struktur, die als *Textur* bezeichnet wird. Eine exakte Definition des Begriffs „Textur" ist schwer zu geben, auch wird der Begriff bei verschiedenen Autoren etwas unterschiedlich gebraucht: Manche verwenden den Begriff „Textur" nur dann, wenn eine gewisse Regelmäßigkeit in der Struktur zu erkennen ist. In der vorliegenden Betrachtung soll „Textur" gleichbedeutend mit „Oberflächenstruktur" verwendet werden. Es sind dann künstliche oder natürliche regelmäßige Strukturen, aber auch künstliche oder natürliche unregelmäßige Strukturen gemeint.

27.2 Grundlagen zu Texturmerkmalen

Eine Eigenschaft haben alle Texturen gemeinsam: Wenn man Merkmale für die Textur (*Texturmerkmale*) berechnen will, so kann man sich nicht auf einzelne Bildpunkte beschränken, sondern muss Umgebungen von Bildpunkten betrachten. Man will erreichen, dass die berechneten Texturmerkmale für die verschiedenen Texturen im (Textur-) Merkmalsraum kompakte Bereiche ergeben, die anschließend mit Segmentierungsverfahren getrennt werden können. Soll z.B. für eine bestimmte Textur ein sinnvolles Merkmal berechnet werden, so müssen die Merkmalswerte auf der reellen Zahlenachse (oder in der Grauwertmenge G) in einem eng begrenzten Intervall liegen. Im Idealfall ergibt das Texturmerkmal, abgebildet in die Grauwertmenge, einen homogenen Bereich. Dieser Bereich kann dann mit einem einfachen Schwellwertverfahren segmentiert werden.

Man sieht hier auch den fließenden Übergang zwischen der rein bildpunktorientierten und der umgebungsorientierten Vorgehensweise: Nachdem z.B. ein Texturmerkmal in die

Grauwertmenge abgebildet ist, ist das entstehende Bild ein Grauwertbild, das mit allen Verfahren der Grauwertbildverarbeitung bearbeitet werden kann. Wird z.B. darauf ein Kantenextraktionsverfahren angewendet, so können die im Kantenbild auftretenden Kanten als *Texturkanten* interpretiert werden. Werden mehrere Texturmerkmale berechnet, so entsteht eine N-dimensionale Szene $\mathbf{S}$, deren Kanäle die einzelnen Texturmerkmale sind. Ein Bildpunkt in der Position (x, y) ist dann ein N-dimensionaler Merkmalsvektor $\mathbf{g} = \mathbf{s}(x, y)$, dessen Komponenten Maßzahlen für das jeweilige Texturmaß an der Stelle (x, y) des Originalbildes sind. Für nachfolgende Segmentierungsverfahren, wie z.B. Klassifikatoren oder neuronale Netze, spielt es keine Rolle, durch welche Vorverarbeitungsschritte die Merkmalsvektoren zustande gekommen sind. Es ist offensichtlich, dass die N-kanalige Merkmalsszene $\mathbf{S}$ auch aus einer Kombination von bildpunktorientierten und umgebungsorientierten Merkmalen aufgebaut sein kann.

Wenn eine Textur eine bestimmte Regelmäßigkeit aufweist, so muss man eine *Grundtexturfläche* finden, die die charakteristischen Eigenschaften der Textur enthält. Die Textur ist dann so beschaffen, dass sich bei vielfacher Wiederholung der Grundtexturfläche die Textur ergibt. Je unregelmäßiger eine Textur ist, desto größer wird die Grundtexturfläche und desto schwieriger ist sie zu bestimmen. Im Extremfall ist die Grundtexturfläche identisch mit der gesamten Oberfläche.

Häufig ist man hier mit zwei entgegengesetzten Sachverhalten konfrontiert: Einerseits soll die Grundtexturfläche so groß sein, dass sie die wesentlichen Struktureigenschaften erfasst, andererseits soll die Grundtexturfläche so klein wie möglich sein. Gründe dafür sind:

- Große Grundtexturflächen verursachen lange Rechenzeiten.
- In einem Bild werden in der Regel unterschiedliche Texturen mit unterschiedlich großen Grundtexturflächen auftreten. Wählt man Umgebungen, für die Texturmerkmale berechnet werden, so müssen sich diese zwangsläufig an den größeren Grundtexturflächen orientieren. In Übergangsbereichen von verschiedenen Texturen wird man dann Mischtexturbereiche erhalten, in denen die gewählten Texturmerkmale falsche Werte liefern.

In [Abma94] wird auch der Gesichtspunkt erwähnt, dass sich eine Textur aus Einzelobjekten zusammensetzt. Demnach kann man versuchen, eine Textur durch die Beschreibung der Einzelobjekte zu charakterisieren. Dazu bieten sich Eigenschaften der Einzelobjekte an, wie die Form, die Größe (und damit die Anzahl), die Grauwertverteilung oder die Anordnung.

Da in der Praxis die Grundtexturflächen nicht oder nur schwer zu ermitteln sind, werden als Kompromiss für jeden Bildpunkt eines Bildes in der Position (x, y) aus einer (u, v)–Umgebung $U = U_{(u,v)}$ (*Texturfenster*) die Werte für die unterschiedlichen Texturparameter berechnet. Dabei werden meistens $m \cdot m$–Umgebungen, mit $m = 3, 5, 7, ...$ verwendet. Es ist klar, dass diese Umgebungen nur eine vage Annäherung an die Grundtexturflächen sind.

Welche Merkmale kann man nun als Texturmerkmale verwenden? In den folgenden Abschnitten werden einige einfache Beispiele zur Beantwortung dieser Frage gegeben.

27.3 Streuung (Varianz)

Die Schätzung der *Streuung* im Texturfenster ist eine erste Möglichkeit, eine Textur zu beschreiben. Intuitiv ist dies auch einleuchtend: Bei einer homogenen Fläche ergibt sich null, je inhomogener die Fläche ist, desto größer werden die Werte. Die Streuung wird über die mittlere quadratische Abweichung im Texturfenster geschätzt:

$$q_U(x,y) = \frac{1}{(M-1)} \sum_{(i,j)\in U} \Big(s(x-i,y-j) - m_U\Big)^2, \tag{27.1}$$

wobei M die Anzahl der Bildpunkte und m_U der Mittelwert des Texturfensters U sind. Bei der Berechnung kann man auch gegen den Rand des Texturfensters die Bildpunkte von q_U geringer gewichten.

In der Bildfolge 27.1 sind Beispiele zur Streuung als Texturmaß gegeben. In Bild 27.1-a ist das Original gezeigt, das auf einem texturierten Hintergrund 16 unterschiedliche Texturen enthält. Die Bilder 27.1-b,c und d zeigen für jeden Bildpunkt die Werte q_U für Umgebungen mit $3 \cdot 3$, $5 \cdot 5$ und $7 \cdot 7$ Bildpunkten. Deutlich ist zu sehen, dass die Textur 6 bereits bei einer $3 \cdot 3$–Umgebung sehr hohe einheitliche Merkmalswerte ergibt, während sich bei den Texturen 5 und 7 einheitlich niedere Werte für q_U berechnen. Dieses Maß wäre somit ausreichend, um die Texturen 5 und 7 von der Textur 6 zu unterscheiden. Die Texturen 9, 10, 11 und 15 tendieren bei den größeren Umgebungen ebenfalls zu homogenen Merkmalswerten. Auffallend ist, dass bei einigen Texturen deutlich die Texturkanten zur Hintergrundtextur hervortreten. In der Textur 13, die sicher keine kleine Grundtexturfläche besitzt, sind Grauwertkanten innerhalb der Textur gut zu sehen. Bei den anderen Texturen sind keine markanten Eigenschaften zu beobachten.

27.4 Gradient

Der *Gradient* kann ebenfalls als Texturmerkmal verwendet werden: In homogenen Bildbereichen werden die Gradientenbeträge gering sein, während in strukturierten Bereichen hohe Gradientenbeträge auftreten. Der Gradient kann mit den in Abschnitt 18.4 angegebenen Formeln berechnet werden. Hinzu kommt hier noch ein weiteres Merkmal, das mit der Berechnung des Gradienten erfasst wird: Die Richtung des Gradienten. Wird die Richtung des Gradienten wieder als Grauwert codiert, so haben in einem entsprechenden Ausgabebild alle Bildpunkte, in deren Umgebung der Gradient dieselbe Richtung besitzt, denselben Grauwert.

Die Bilder 27.2 zeigen Beispiele hierzu. Die Gradienten wurden hier aus 3·3–Umgebungen berechnet. Bild 27.2-a zeigt die Gradientenbeträge. Es ergibt sich kein wesentlicher Unterschied zu den lokalen Streuungen im Texturfenster (Bild 27.2-b). Die Bilder 27.2-b und 27.2-c stellen die Gradientenrichtungen dar. Bei Bild 27.2-b sind deutlich die einheitlichen Richtungen der Gradienten zu sehen (z.B. bei den Texturen 1, 2, 6, 8, 13 und 15). Im Hinblick auf die Forderung, dass Texturmerkmale für die einzelnen Texturen im Merkmalsraum homogene Bereiche ergeben sollen, ist die Eigenschaft der Gradientenrichtungen

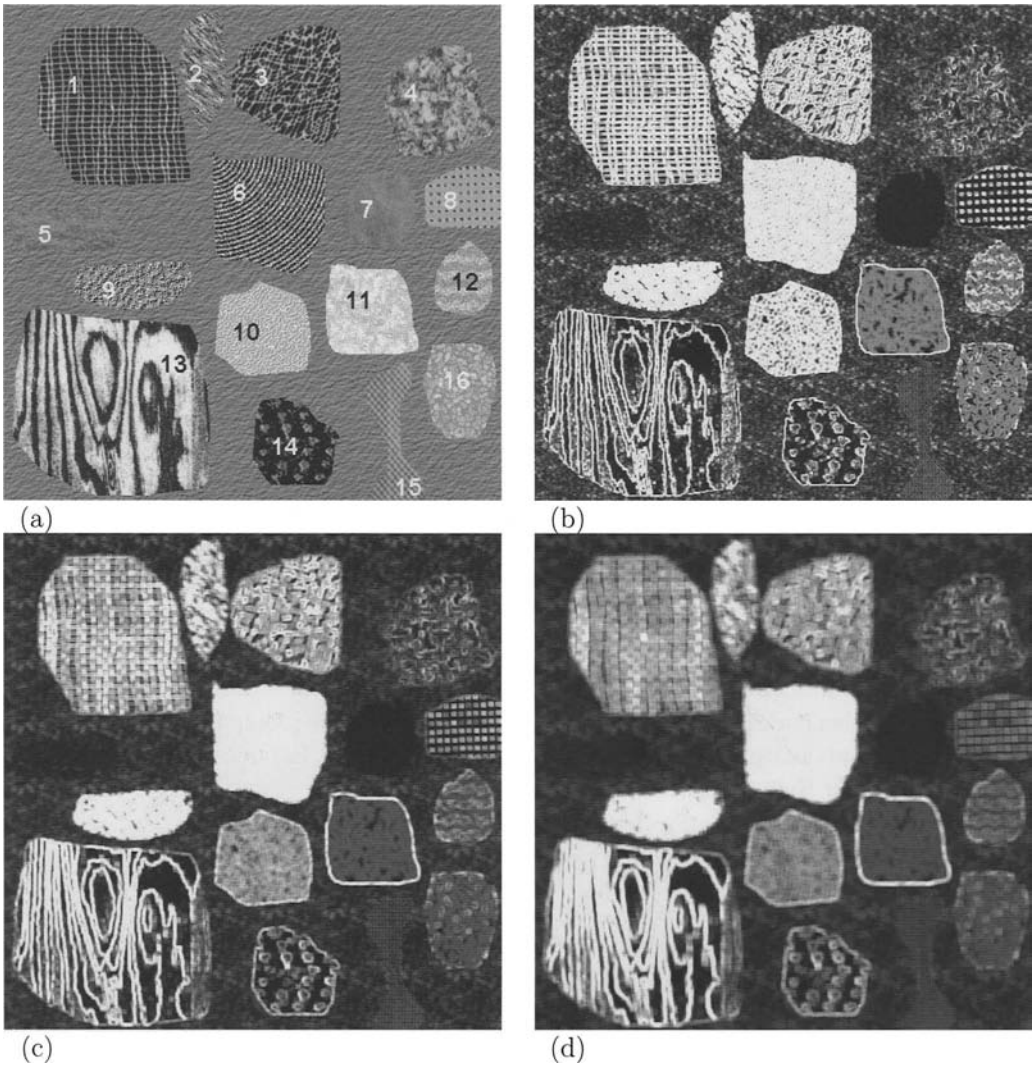


Bild 27.1: Lokale Schätzung der Streuung im Texturfenster. (a) Originalbild mit 16 Texturen vor einem texturierten Hintergrund. (b) Mittlere quadratische Abweichung q_U bei einem Texturfenster von 3·3 Bildpunkten. Die Texturen 5, 6 und 7 ergeben bereits ziemlich einheitliche Bereiche. (c) Mittlere quadratische Abweichung q_U bei einem Texturfenster von $5 \cdot 5$ Bildpunkten. Texturkanten zur Hintergrundtextur sind bei einigen Beispielen deutlich zu sehen. (d) Mittlere quadratische Abweichung q_U bei einem Texturfenster von $7 \cdot 7$ Bildpunkten. Auch die Texturen 9, 10, 11 und 15 tendieren jetzt zu einheitlichen Bereichen.

störend, dass bei Grauwertkanten gegenüberliegende Gradienten (180°) mit unterschiedlichen Grauwerten codiert werden. In Bild 27.2-c wurden aus diesem Grund gegenüberliegende Gradientenrichtungen zu einem Grauwert zusammengefasst. Deutlich hebt sich dann die Textur 15 heraus. In Textur 6 ist der Verlauf der Gradientenrichtungen gut zu verfolgen. Bei Textur 13 ergeben sich durch die ausgeprägte Vorzugsrichtung der Holzmaserung große homogene Bereiche.

In Bild 27.2-d wurde nur für diejenigen Gradienten die Richtung codiert, deren Betrag größer als ein vorgegebener Schwellwert ist. An den Kantenverläufen sind die unterschiedlichen Gradientenrichtungen gut zu erkennen. Der abrupte Übergang von Hellgelb nach Dunkelblau entspricht Stellen, an denen die Gradientenrichtung von 360° auf 0° springt.

27.5 Kantendichte

Aufbauend auf dem Betrag des Gradienten wird in [Erns91] die *Kantendichte* kd_U im Texturfenster U als weiteres Texturmaß vorgeschlagen. Dazu wird das Gradientenbetragsbild binarisiert und die Kantendichte wie folgt berechnet:

$$kd_U(x,y) = \frac{1}{(M-1)} \sum_{(i,j)\in U} e(x-i, y-j), \tag{27.2}$$

$$\text{mit } e(x-i,y-j) = \begin{cases} 1, & \text{Kante,} \\ 0, & \text{keine Kante.} \end{cases}$$

Das Texturfenster U kann hier durch entsprechende Wahl an bestimmte Vorzugsrichtungen der zu untersuchenden Texturen angepasst werden. Der Wert von kd_U liegt zwischen 0 und 1. Durch entsprechende Skalierung kann dieser Wert bei Bedarf in die Grauwertmenge G abgebildet werden (Bild 27.3-a).

Die Berechnung des Parameters kd_U kann auch mit anderen Vorverarbeitungsschritten kombiniert werden. Statt Gradientenbeträge z.B. mit Hilfe eines Sobeloperators zu bilden, können auch die Ergebnisse eines Laplace-Operators (Abschnitt 18.4) als Grundlage für die Berechnung von kd_U verwendet werden (Bild 27.3-b).

Eine andere Möglichkeit besteht darin, das Originalbild $\mathbf{S}_e$ durch ein Medianfilter (Kapitel 19) in ein Bild $\mathbf{S}_{med}$ zu transformieren und dann die Differenz $\mathbf{S}_{diff} = 127 + \mathbf{S}_e - \mathbf{S}_{med}$ zu berechnen. Durch diese Operation bleibt die Struktur der Texturen erhalten, während homogene Bereiche (Gleichanteil) verschwinden (Bild 27.3-c). Dieses Bild $\mathbf{S}_{diff}$ kann jetzt als Ausgangspunkt für die Berechnung des Texturparameters kd_U dienen. Allerdings ist es sinnvoll, auch die Grauwerte, die kleiner als 127 sind, als Kanten zu interpretieren. Das erreicht man z.B. durch

$$\mathbf{S}_{norm} = |\mathbf{S}_{diff} - 127| \cdot 2. \tag{27.3}$$

Bild 27.3-d zeigt das Ergebnis.

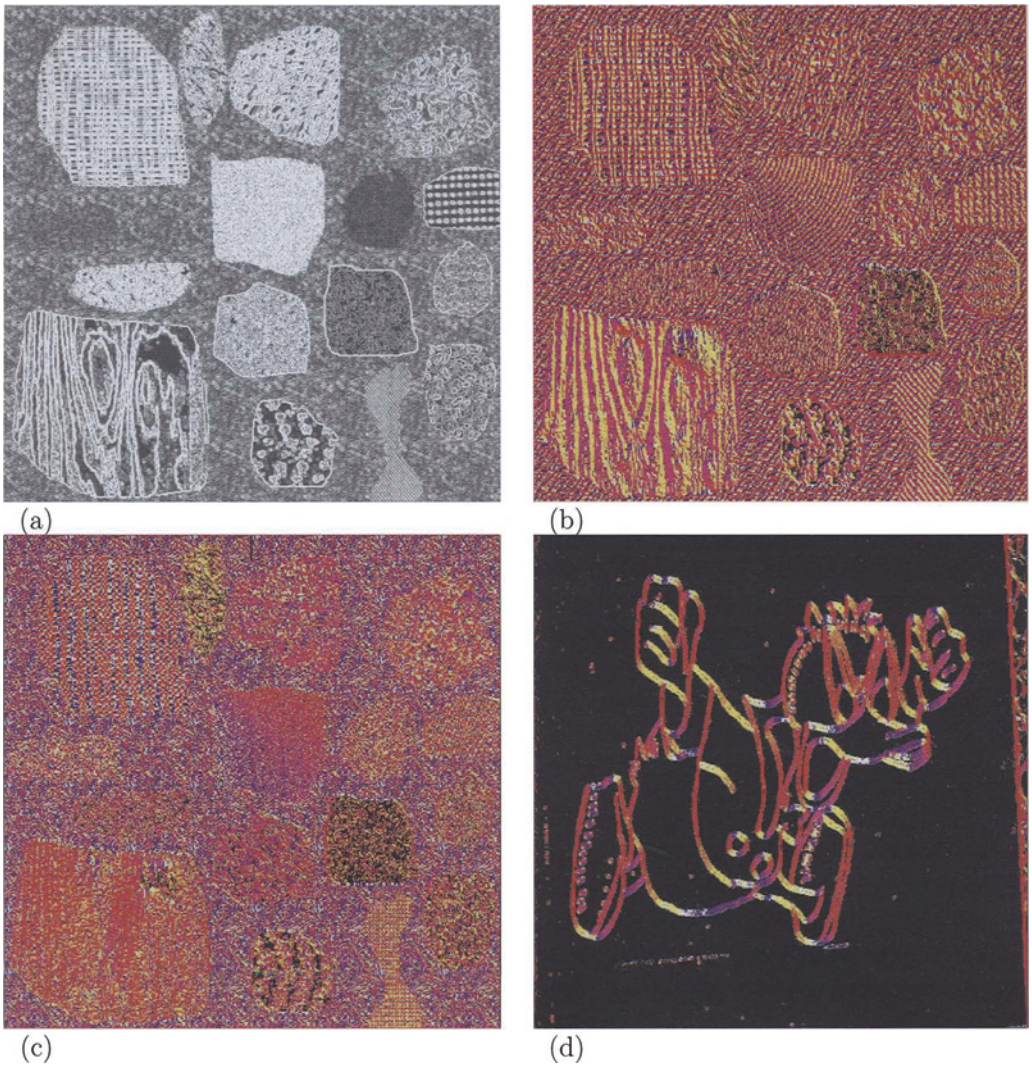


Bild 27.2: Der Gradient als Texturmerkmal. (a) Die Gradientenbeträge zum Testbild 27.1-a. Der Gradient wurde aus $3 \cdot 3$–Umgebungen berechnet. (b) Die Gradientenrichtungen, abgebildet in die Grauwertmenge G. Gleichen Farbtönen entsprechen gleiche Richtungen. (c) Hier wurden um 180° gegenüberliegende Gradientenrichtungen zu einem Grauwert zusammengefasst. (d) Hier wurden nur für diejenigen Bildpunkte die Gradientenrichtungen dargestellt, deren Gradientenbeträge über einem Schwellwert liegen. Man sieht deutlich, wie sich entlang der Umrandungen der Segmente die Gradientenrichtungen ändern.

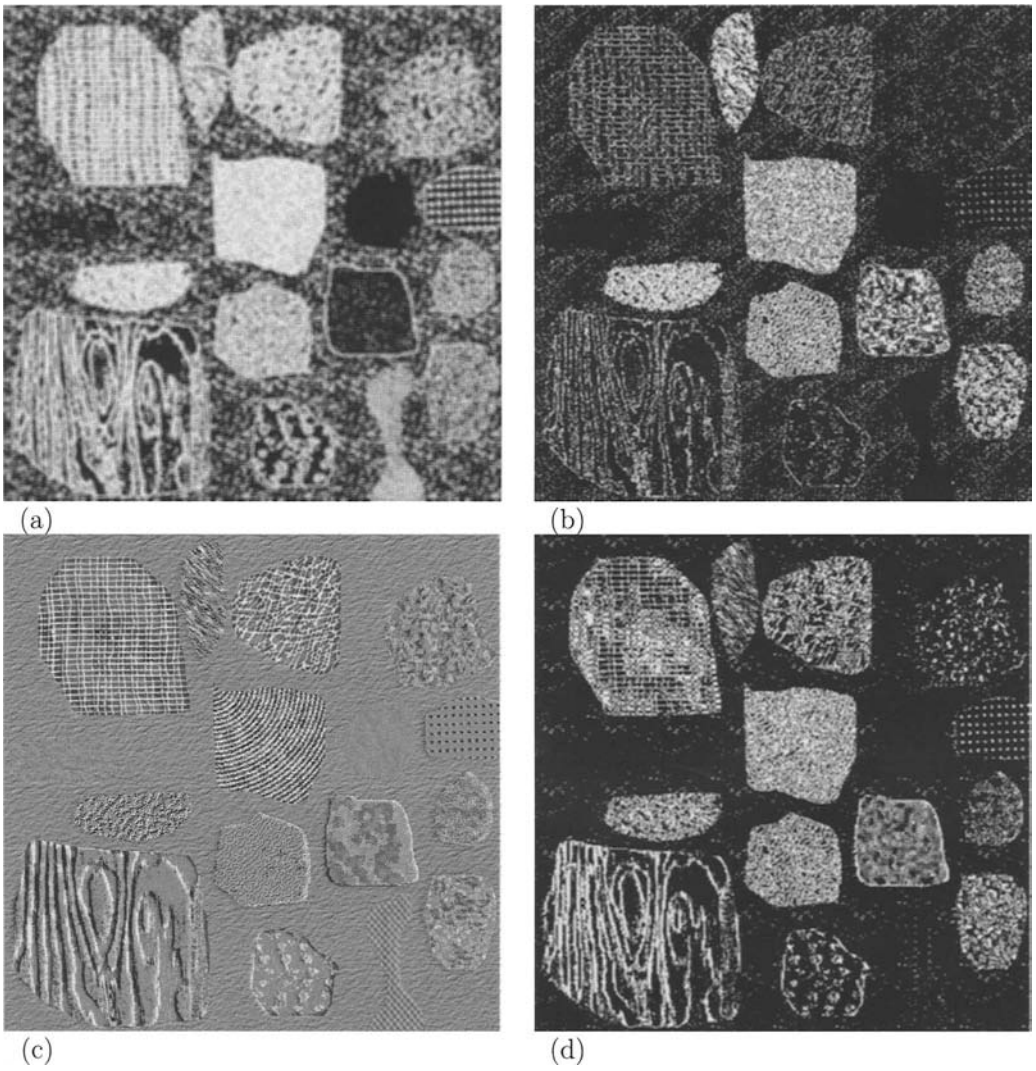


Bild 27.3: (a) Kantendichte der Gradientenbeträge nach einem Sobeloperator mit einem Texturfenster von $5 \cdot 5$ Bildpunkten. (b) Kantendichte nach einem Laplace-Operator. (c) Differenzbild: Original-Medianbild+127. (d) Kantendichte des Differenzbildes.

27.6 Autokorrelation

Die *Autokorrelation* ist ein weiteres Texturmerkmal, das vor allem bei Texturen mit einer Vorzugsrichtung erfolgversprechend ist. Dabei wird die Korrelation der Grauwerte im Texturfenster $U_{(x,y)}$ für den Bildpunkt in der Position (x, y) mit den Grauwerten in einem um einen Vektor $(\Delta x, \Delta y)$ verschobenen Fenster $U' = U_{(x+\Delta x, y+\Delta y)}$ berechnet. Die Korrelation kann man mit folgender Formel berechnen:

$$r_U^{(\Delta x,\Delta y)}(x,y) = \frac{\sum\limits_{(i,j)\in U} \Big(s(x-i,y-j) - m_U\Big)\Big(s(x+\Delta x-i, y+\Delta y-j) - m_{U'}\Big)}{\sqrt{\sum\limits_{(i,j)\in U} \Big(s(x-i,y-j) - m_U\Big)^2 \sum\limits_{(i,j)\in U} \Big(s(x+\Delta x-i, y+\Delta y-j) - m_{U'}\Big)^2}}. \tag{27.4}$$

Dabei ist m_U der Mittelwert der Umgebung U und $m_{U'}$ der Mittelwert der Umgebung U'. Der Korrelationskoeffizient r_U liegt zwischen -1 und $+1$ und kann z.B. durch $255 \cdot |r_U|$ oder $127 \cdot (r_U + 1)$ in die Grauwertmenge G abgebildet werden. Die Bilder 27.4 zeigen Beispiele zur Autokorrelation. Die Bilder wurden mit unterschiedlich großen Texturfenstern und verschiedenen Verschiebungsvektoren berechnet. Durch den Vergleich mit den Originaltexturen von Bild 27.1-a sind die Ergebnisse gut zu interpretieren. In der Praxis muss man in vielen Testläufen die optimalen Parametereinstellungen für die Größe des Texturfensters und den Verschiebungsvektor finden.

27.7 Abschlussbemerkung zu den einfachen Texturmaßen

Die Aufzählung von Texturmerkmalen könnte noch lange fortgesetzt werden. So werden z.B. in [Abma94] weitere Maßzahlen vorgeschlagen, z.B.:

- Merkmale aus der *co-occurrence*-Matrix (Grauwertübergangsmatrix, Abschnitt 16.11).
- Merkmale auf der Basis des Lauflängencodes (Kapitel 34). Hier liegt der Gedanke zugrunde, dass in Bereichen mit ausgeprägter Struktur viele kurze Ketten und in homogenen Bereichen wenige lange Ketten erzeugt werden.
- Merkmale aus der Entropie (Definition in Abschnitt 16.11).
- Merkmale aus dem Frequenzspektrum.

Betrachtet man die Ergebnisse der dargestellten Beispiele, so ist zu sehen, dass nahezu überall das gewählte Texturfenster zu klein ist, um die wesentlichen Eigenschaften der Texturen zu erfassen. Erfolgversprechender sind daher Verfahren, bei denen die Wesensart

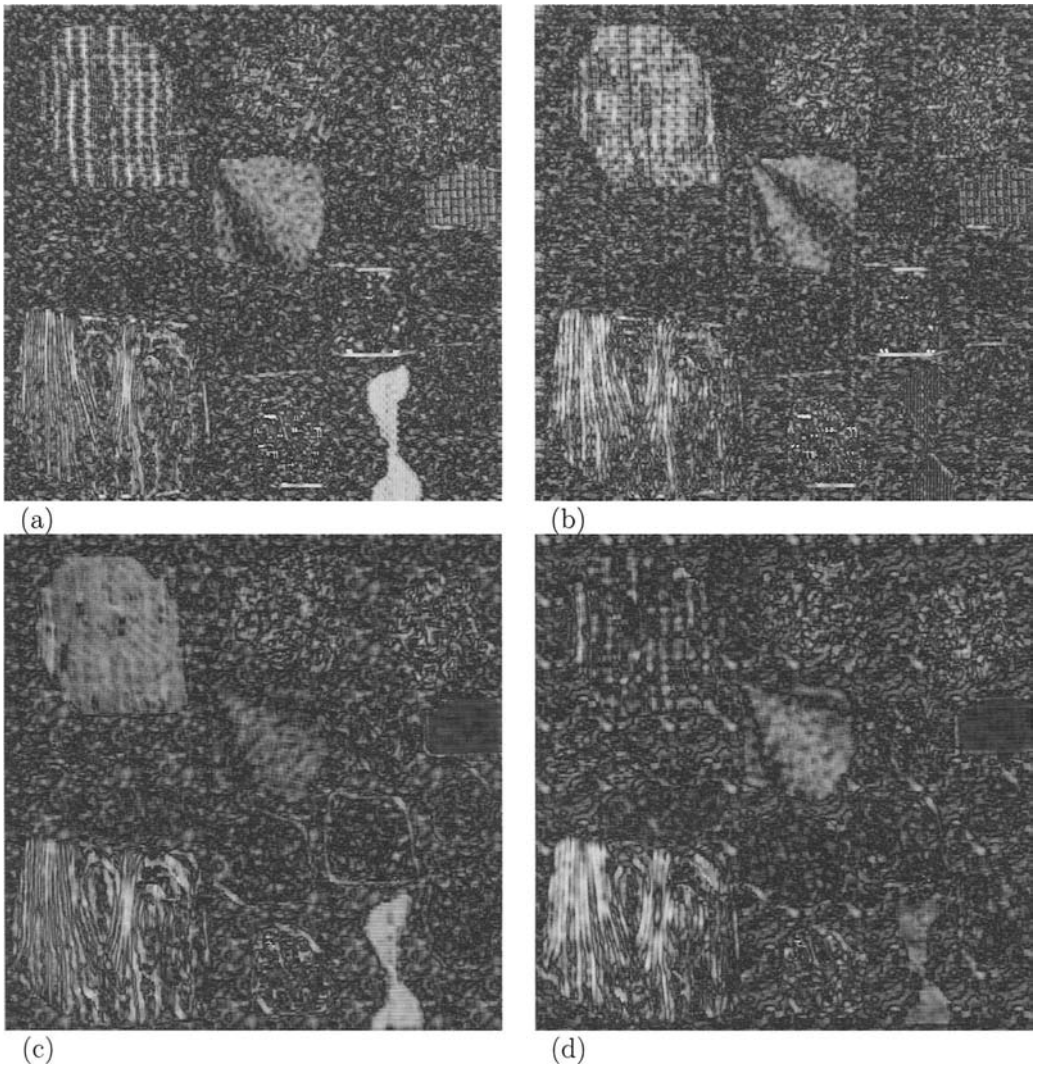


Bild 27.4: Autokorrelation $r_U^{(\Delta x, \Delta y)}$ mit unterschiedlich großen Texturfenstern und verschiedenen Verschiebungsvektoren: (a) $5 \cdot 5$–Umgebung, $(\Delta x, \Delta y) = (0, 3)$. (b) $5 \cdot 5$–Umgebung, $(\Delta x, \Delta y) = (0, -5)$. (c) $7 \cdot 7$–Umgebung, $(\Delta x, \Delta y) = (3, 3)$. (d) $7 \cdot 7$–Umgebung, $(\Delta x, \Delta y) = (5, 5)$.

von Texturen, dass in einer Textur unterschiedliche Frequenzen auftreten, berücksichtigt wird. Wenn eine Textur hierarchisch in Bereiche mit unterschiedlichen Wellenzahlindizes zerlegt wird, können die Eigenschaften der Textur besser erfasst werden.

Beispiele dazu werden im nächsten Kapitel gebracht, wo Texturparameter mittels Gauß- und Laplace-Pyramiden berechnet werden. Im übernächsten Kapitel wird eine weitere Betrachtungsweise erläutert, die auf der fraktalen Geometrie aufbaut.

Kapitel 28

Gauß- und Laplace-Pyramiden

28.1 Anwendungen

In diesem Kapitel wird eine interessante Kombination aus Unschärfe und Schärfe erläutert, die bei einigen Anwendungen sinnvoll eingesetzt werden kann. Soll z.B. ein Bild durch eine Filteroperation (Glättungsoperator) sehr stark unscharf gemacht werden, so kann dies durch einen lokalen Summenoperator erzielt werden (Kapitel 18). Allerdings muss dazu der Filterkern entsprechend groß gewählt oder mehrfach wiederholt werden, was einen erhöhten Rechenaufwand bedeutet (er wächst hier quadratisch mit der Größe des Filterkerns). Eine andere Möglichkeit wäre eine Filterung im Ortsfrequenzbereich. Das bedeutet aber unter anderem eine zweidimensionale Fourier-Transformation und eine zweidimensionale, inverse Fourier-Transformation, beides Operationen, die im Rechenaufwand nicht zu unterschätzen sind. In diesem Kapitel wird eine Technik erläutert, die es erlaubt, Bilder mit geringerem Aufwand zu filtern.

Eine andere Anwendung ist die Möglichkeit des *image mosaicing*, also des Zusammensetzens eines Ausgabebildes aus mehreren Eingabebildern. Hier werden, nach Maßgabe einer binären Bildmaske, Ausschnitte zweier Eingabebilder verwendet und zu einem Mosaikbild zusammengesetzt. Würde man nur die Bildausschnitte der beiden Eingabebilder zusammenkopieren, so würde man im Mosaikbild deutliche Ränder sehen. Mit der hier vorgeschlagenen Technik wird dieser Effekt vermieden. Die Bildausschnitte werden wechselseitig interpoliert, so dass in den Übergangsbereichen keine störenden Strukturunterschiede auftreten.

Mit den Gauß- und Laplace-Pyramiden können die Strukturen von Oberflächen (Texturen) untersucht werden. Beispielsweise können einzelne Informationsschichten als Masken bei der Segmentierung dienen. Anwendungsbeispiele dazu sind die Qualitäts- und Vollständigkeitskontrolle. Hier ergibt sich ein Zusammenhang mit der fraktalen Geometrie und dem *scale space filtering* (Kapitel 29).

Schließlich finden Gauß-Pyramiden eine breite Anwendung in der 3D-Computergrafik. Beim Textur Mapping werden häufig verkleinerte Varianten einer Textur benötigt, die in Form einer vorab berechneten Gauß-Pyramide zur Verfügung gestellt werden. In der

3D-Computergrafik werden Gauß-Pyramiden als *MipMaps* bezeichnet (Abschnitt 13.1.3). Eine weitere Anwendung ist die sogenannte z-Pyramide beim *Occlusion Culling* (Abschnitt 15.2.2).

28.2 Begriffe aus der Signaltheorie

Zum besseren Verständnis der Gauß- und Laplace-Pyramiden sind in diesem Abschnitt in kurzer Form einige Grundlagen der Signaltheorie zusammengestellt. Wenn Strukturen untersucht werden, die sich periodisch wiederholen, so charakterisiert man sie durch ihr Schwingungsverhalten, nämlich durch die Länge einer Periode oder die Anzahl der Schwingungen pro Einheitslänge. In zeitabhängigen Systemen bezeichnet man die Länge einer Periode als *Periodendauer* mit der Einheit *Sekunden* und die Anzahl der Schwingungen pro Sekunde als *Frequenz* mit der Einheit *Sekunden*$^{-1}$. Bei Bildern verwendet man die Begriffe *Wellenlänge* λ (Einheit: *Länge in Pixel*) und *Ortsfrequenz* f oder *Wellenzahl* k (Einheit: *Anzahl der Wellen pro Pixel*). Naturgemäß ist f bzw. k immer kleiner als eins. Es gilt

$$f = k = \frac{1}{\lambda}. \tag{28.1}$$

Die größtmögliche Wellenlänge wird durch die Bildgröße bestimmt: Bei einem Bild mit z.B. 512 Bildpunkten pro Zeile und Spalte sind die maximale Wellenlänge und die minimale Wellenzahl pro Zeile (Spalte) vorgegeben durch

$$\lambda_{max} = 512 Pixel \text{ und } f_{min} = \frac{1}{\lambda_{max}} = \frac{1}{512} Wellen\ pro\ Pixel. \tag{28.2}$$

Die Größe des Bildes bestimmt somit auch, wie oft sich eine periodische Struktur innerhalb eines Bildes wiederholen kann. Diese Anzahl wird als *Wellenzahlindex* u bezeichnet.

Das Abtasttheorem ([Abma94]) besagt, dass eine periodische Struktur aus den Abtastwerten nur dann richtig rekonstruiert werden kann, wenn die kleinste Wellenlänge mindestens zweimal abgetastet wird. Die maximale Wellenzahl, die bei einer gegebenen Abtastrate fehlerfrei rekonstruiert werden kann, heißt *Grenzwellenzahl*.

Die minimale Wellenlänge und die korrespondierende maximale Wellenzahl in einem digitalen Bild werden somit nicht durch die Bildgröße, sondern durch das Abtasttheorem beeinflusst. Für sie gilt immer:

$$\lambda_{min} = 2 Pixel \text{ und } f_{max} = \frac{1}{\lambda_{min}} = \frac{1}{2} Wellen\ pro\ Pixel. \tag{28.3}$$

In einem $512 \cdot 512$–Bild kann sich somit eine Struktur mit der Wellenzahl $k_{max} = 1/2$ höchstens 256 mal wiederholen, d.h. dass für den maximalen Wellenzahlindex gilt $u_{max} = 256$.

Der kleinstmögliche Wellenzahlindex ist immer $u_{min} = 1$, da die größtmögliche Struktur innerhalb einer Zeile (Spalte) nur einmal auftreten kann.

In einem Bild werden nun, begrenzt durch die minimale und die maximale Wellenlänge, viele verschiedene Wellenlängen auftreten. Die Menge der in einem Bild auftretenden Wellenlängen und den damit verbundenen Wellenzahlen (Ortsfrequenzen) bezeichnet man als *Spektrum* des Bildes. Es kann durch eine Fourier-Transformation berechnet werden (Kapitel 21).

28.3 Motivation für Gauß- und Laplace-Pyramiden

Da die Gauß- und Laplace-Pyramiden Kenntnisse aus der Signaltheorie voraussetzen, werden einige relevante Begriffe kurz zusammengefasst.

Bei vielen Anwendungen der digitalen Bildverarbeitung und Mustererkennung ist es notwendig, Bilder oder Bildausschnitte anhand ihres Frequenzverhaltens zu analysieren und dazu in ihre Frequenzanteile zu zerlegen. Der Begriff *Frequenz* wird in diesem Zusammenhang gleichbedeutend mit *Ortsfrequenz* oder *Wellenzahl* (siehe vorhergehender Abschnitt) verwendet.

Die Fourier-Transformation, mit der eine Frequenzzerlegung berechnet werden kann, ist aber mit einigen Nachteilen verbunden. Hier ist zunächst der beachtliche Rechenaufwand zu erwähnen, der jedoch bei der Verwendung von schneller Hardware bewältigt werden kann. Der zweite Nachteil ist, dass im Frequenzraum die einzelnen Frequenzanteile zwar zu erkennen sind, jedoch ihre direkte Zuordnung zu Strukturen im Ortsbereich fehlt.

Man benötigt somit eine Darstellungsform im Ortsbereich, die ein Bild in mehrere Frequenzbereiche aufspaltet. Die Gauß- und Laplace-Pyramiden führen eine Zerlegung eines Bildes in dieser Form durch.

Die grundlegende Idee ist im Folgenden anhand eines einfachen, eindimensionalen Beispiels erläutert: Angenommen, eine Bildzeile eines Bildes bestehe aus 32 Bildpunkten. Dann können maximal 16 Wellen der Wellenlänge $\lambda_{min} = 2Pixel$ in dieser Zeile auftreten. Das entspricht einer maximalen Wellenzahl von $k_{max} = \frac{1}{2}$ *Wellen pro Pixel.* Die maximale Wellenlänge ist $\lambda_{max} = 32Pixel$, was einer minimalen Wellenzahl von $k_{min} = \frac{1}{32}$ *Wellen pro Pixel* entspricht.

Wenn nun in dieser Bildzeile nur Wellenlängen auftreten, die kleiner sind als $\frac{1}{2}\lambda_{max}$, im gewählten Beispiel also kleiner als 16, so kann diese Zeile auch ohne Informationsverlust mit einem doppelt so großen Raster dargestellt werden. Das Abtasttheorem besagt außerdem, dass die ursprüngliche Bildzeile mit 32 Pixel aus dem gröberen Raster ohne Informationsverlust rekonstruiert werden kann. Ein eindimensionaler Glättungsoperator, etwa der Länge drei, hat aber auf dem gröberen Raster eine wesentlich stärkere Auswirkung als auf dem ursprünglichen Raster mit 32 Bildpunkten. Um Rechenaufwand bei einer Glättung zu sparen, könnte man somit das Raster vergröbern, die Glättung durchführen und anschließend wieder zum ursprünglichen Raster zurückkehren.

Auf diesem Sachverhalt baut die Idee der Gauß- und Laplace-Pyramiden auf: Auf ein Grauwertbild $\mathbf{S}_e = (s_e(x, y))$ wird ein geeigneter Glättungsoperator angewendet, und das so entstehende Bild wird in Zeilen- und Spaltenrichtung um die Hälfte verkleinert. Dieser Vorgang (Tiefpassfilterung und Verkleinerung) wird iterativ fortgesetzt, bis ein Bild mit

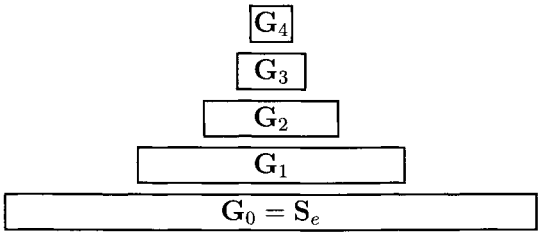


Bild 28.1: Gauß-Pyramide: Ausgehend vom Originalbild in der Schicht (Stufe), die am weitesten unten liegt, entsteht die darüber liegende Schicht durch eine Tiefpassfilterung und eine Halbierung der Zeilen- und Spaltenanzahl.

minimalem Flächeninhalt ($2 \cdot 2$ oder $3 \cdot 3$ Pixel) erreicht ist. Die so entstandene Folge von geglätteten und flächenmäßig jeweils um den Faktor vier reduzierten Bildern bezeichnet man als *Gauß-Pyramide*. Stellt man sich in einer grafischen Darstellung die einzelnen Bilder übereinander geschichtet vor, so entsteht die Form einer Pyramide, woher der Name kommt (Bild 28.1).

Die einzelnen Gauß-Pyramidenstufen unterscheiden sich durch ihren Frequenzanteil. Die unterste Stufe enthält alle Wellenzahlen des Originalbildes, also sein gesamtes Spektrum. Wählt man die Tiefpassfilterung so, dass die Frequenzen des halben Spektrums herausgefiltert werden, so enthält jede Stufe die Frequenzen des halben Spektrums der darunterliegenden Stufe.

Das Abtasttheorem besagt nun, dass ein Signal, bei dem die maximale Frequenz herausgefiltert wurde, durch weniger Abtastwerte dargestellt werden kann, ohne dass ein Informationsverlust auftritt. Da bei der Tiefpassfilterung die Breite des Spektrums halbiert wurde, kann man in der darüberliegenden Schicht die Zeilen- und Spaltenanzahl halbieren.

Die *Laplace-Pyramide* wird durch die Differenzbildung der übereinander liegenden Schichten der Gauß-Pyramide gewonnen. Sie ist somit eine Folge von hochpassgefilterten Bildern.

28.4 Der REDUCE-Operator

In diesem und den folgenden Abschnitten wird der Aufbau von Gauß- und Laplace-Pyramiden näher beschrieben. Dazu werden einige Annahmen gemacht und verschiedene Bezeichnungen eingeführt: Das Originalbild $\mathbf{S}_e = (s_e(x, y))$ sei quadratisch und besitze die Zeilen- und Spaltenlänge $R + 1 = 2^r + 1$. Bei $R = 512$ hat $\mathbf{S}_e$ also die Zeilen und Spalten $0, 1, ..., 512$ $(r = 9)$.

Die unterste Stufe der *Gauß-Pyramide* wird mit $\mathbf{G}_0$ bezeichnet. Sie ist mit dem Origi-

nalbild $\mathbf{S}$ identisch: $\mathbf{G}_0 = \mathbf{S}_e$. In der Praxis treten häufig Bilder der Größe $512 \cdot 512$ oder $256 \cdot 256$ auf. Für das hier beschriebene Verfahren werden aber Bilder der Größe $513 \cdot 513$ oder $257 \cdot 257$ benötigt. Man kann dies erreichen, indem man beim Aufbau der untersten Schicht die erste Bildzeile und die erste Bildspalte des Ausgangsbildes $\mathbf{S}_e$ verdoppelt. In der weiteren Darstellung wird dieser Sachverhalt nicht mehr berücksichtigt. Vielmehr wird abkürzend z.B. von $512 \cdot 512$–Bildern gesprochen.

Die weiteren Schichten sind $\mathbf{G}_1$, $\mathbf{G}_2$ bis $\mathbf{G}_r$. $\mathbf{G}_r$ ist die Spitze der Pyramide mit $2 \cdot 2$ Bildpunkten. In manchen Implementierungen wird als Spitze die darunter liegende Schicht mit $3 \cdot 3$ Bildpunkten verwendet. Die Zählung der Schichten geht dann von $\mathbf{G}_0 = \mathbf{S}_e$ bis $\mathbf{G}_{r-1}$. Wenn die Transformation von $\mathbf{G}_i$ zu $\mathbf{G}_{i+1}$ mit einer Reduktionsfunktion REDUCE bezeichnet wird, so kann der Aufbau einer Gauß-Pyramide formal wie folgt beschrieben werden:

$$\begin{aligned} &\mathbf{G}_0 = \mathbf{S}_e; \\ &\mathbf{G}_{i+1} = \text{REDUCE}(\mathbf{G}_i), \quad i = 0, 1, ..., r-1. \end{aligned} \tag{28.4}$$

In Bild 28.2 ist dieser Formalismus zum Aufbau einer Gauß-Pyramide als Blockdiagramm dargestellt. Bild 28.3 zeigt die Schichten $\mathbf{G}_0$ bis $\mathbf{G}_6$ eines Testbildes.

Aufgrund des Konstruktionsverfahrens der Gauß-Pyramide kann man Aussagen machen über den maximalen Wellenzahlindex und damit über das Spektrum der einzelnen Stufen. Für ein Bild mit z.B. $512 \cdot 512$ Zeilen und Spalten ist das in folgender Tabelle zusammengestellt:

Stufe	Zeilen/Spalten	max. Wellenzahlindex
$\mathbf{G}_0$	$2^9 + 1 = 513$	256
$\mathbf{G}_1$	$2^8 + 1 = 257$	128
$\mathbf{G}_2$	$2^7 + 1 = 129$	64
$\mathbf{G}_3$	$2^6 + 1 = 65$	32
$\mathbf{G}_4$	$2^5 + 1 = 33$	16
$\mathbf{G}_5$	$2^4 + 1 = 17$	8
$\mathbf{G}_6$	$2^3 + 1 = 9$	4
$\mathbf{G}_7$	$2^2 + 1 = 5$	2
$\mathbf{G}_8$	$2^1 + 1 = 3$	1
$\mathbf{G}_9$	$2^0 + 1 = 2$	-

28.5 Der EXPAND-Operator

Um die einzelnen Schichten bei weiteren Verarbeitungsschritten miteinander vergleichen zu können, ist es notwendig, mit einer EXPAND-Operation eine Schicht auf die Größe der darunterliegenden Schicht expandieren zu können. Die fehlenden Zeilen und Spalten werden dabei interpoliert. Die EXPAND-Operation wird mehrmals nacheinander ausgeführt, so

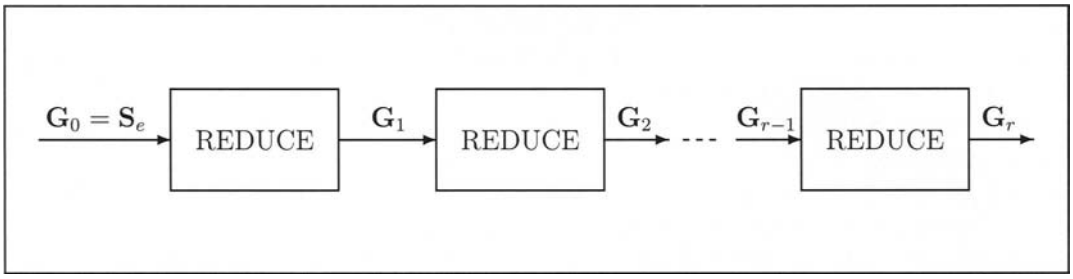


Bild 28.2: Blockdiagramm zum Aufbau einer Gauß-Pyramide.

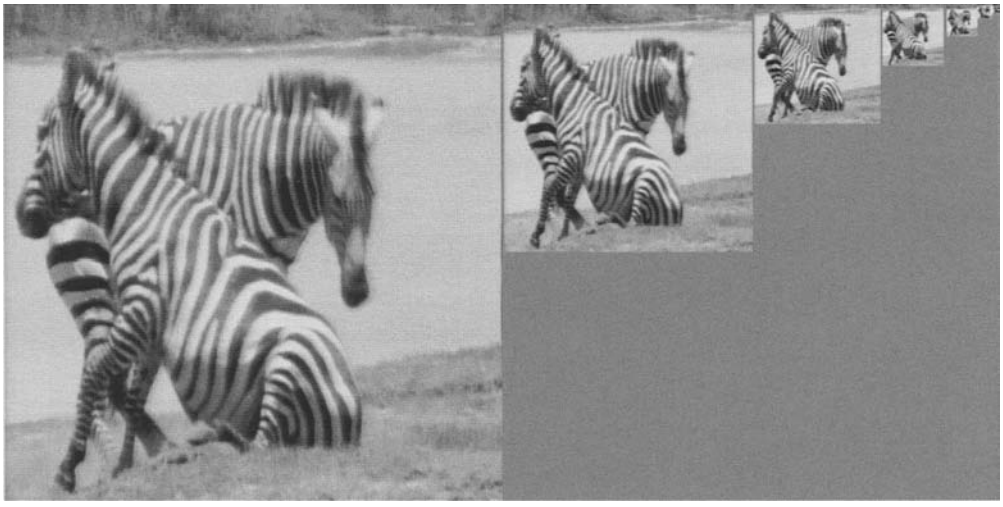


Bild 28.3: Gauß-Pyramide: Die Schichten $\mathbf{G}_0$ bis $\mathbf{G}_6$ eines Testbildes mit einer Größe von $512 \cdot 512$ Bildpunkten.

dass jede Stufe auf die Größe des Originals gebracht werden kann. Dieser Formalismus kann folgendermaßen beschrieben werden:

$$\begin{aligned}
&\mathbf{G}_{i,1} = \text{EXPAND}(\mathbf{G}_{i,0});\ \text{mit}\ \mathbf{G}_{i,0} = \mathbf{G}_i, \quad i = r, r-1, ..., 1; \\
&\mathbf{G}_{i,2} = \text{EXPAND}(\mathbf{G}_{i,1}); \\
&... \\
&\mathbf{G}_{i,i} = \text{EXPAND}(\mathbf{G}_{i,i-1}).
\end{aligned} \tag{28.5}$$

Die Bilder mit den Bezeichnungen $\mathbf{G}_{1,1}$, $\mathbf{G}_{2,2}$ bis $\mathbf{G}_{r,r}$ sind dann auf die Größe des Originalbildes expandiert. Wie der EXPAND-Operator genau funktioniert, wird im nächsten Abschnitt erläutert. In der Bildfolge 28.4 sind die Schichten $\mathbf{G}_0$, $\mathbf{G}_1$, $\mathbf{G}_2$, $\mathbf{G}_3$, $\mathbf{G}_4$ und $\mathbf{G}_5$ jeweils auf die Originalgröße expandiert dargestellt. Man sieht, dass in der Schicht $\mathbf{G}_3$ das Streifenmuster bereits undeutlich wird.

In den Schichten der *Laplace-Pyramide* werden nun die Bildanteile gespeichert, die durch die REDUCE-Operation bei der Bildung der Gauß-Pyramide herausgefiltert wurden. Der Schritt von der Gauß-Pyramide zur Laplace-Pyramide besteht somit in einer Differenzbildung der einzelnen Schichten $\mathbf{G}_0$ bis $\mathbf{G}_r$. Dabei tritt allerdings das Problem auf, dass z.B. die Schichten $\mathbf{G}_0$ und $\mathbf{G}_1$ in der Größe nicht zusammenpassen, da ja $\mathbf{G}_1$ flächenmäßig viermal kleiner ist als $\mathbf{G}_0$. $\mathbf{G}_1$ muss somit zuerst durch die bereits oben verwendete EXPAND-Operation auf die Größe von $\mathbf{G}_0$ gebracht werden. Wenn die unterste Schicht der Laplace-Pyramide mit $\mathbf{L}_0$ bezeichnet wird, kann dieser Verarbeitungsschritt durch $\mathbf{L}_0 = \mathbf{G}_0 - \text{EXPAND}(\mathbf{G}_1)$ beschrieben werden. Da $\mathbf{G}_1$ aus $\mathbf{G}_0$ u.a. durch die Anwendung eines Glättungsoperators hervorgegangen ist, wird das Differenzbild $\mathbf{L}_0$ die Information enthalten, die durch die Glättung verloren ging, also die Bildkanten. Die gesamte Laplace-Pyramide mit den Schichten $\mathbf{L}_0$ bis $\mathbf{L}_r$ wird folgendermaßen berechnet:

$$\begin{aligned}
&\mathbf{L}_i = \mathbf{G}_i - \text{EXPAND}(\mathbf{G}_{i+1}) = \mathbf{G}_{i,0} - \mathbf{G}_{i+1,1}; i = 0, 1, ..., r-1; \\
&\mathbf{L}_r = \mathbf{G}_r.
\end{aligned} \tag{28.6}$$

Die Spitze der Laplace-Pyramide $\mathbf{L}_r$ wurde in (28.6) mit der Spitze der Gauß-Pyramide $\mathbf{G}_r$ gleichgesetzt. Damit ist formal der Aufbau der Laplace-Pyramiden beschrieben. Ein Blockdiagramm, das den Aufbau einer Laplace-Pyramide grafisch verdeutlicht, zeigt Bild 28.5.

Wie die Gauß-Pyramide ist die Laplace-Pyramide eine Darstellung von verschiedenen Bereichen der Wellenzahlindizes. Als Beispiel zeigt die Bildfolge 28.6 die zu den Bildern 28.3 und 28.4 gehörige expandierte Laplace-Pyramide. Eine Schicht der Laplace-Pyramide enthält gerade die Frequenzen, die bei der Bildung der Gauß-Pyramide herausgefiltert wurden. In der folgenden Tabelle sind die Bereiche der Wellenzahlindizes für ein $512 \cdot 512$–Bild angegeben:

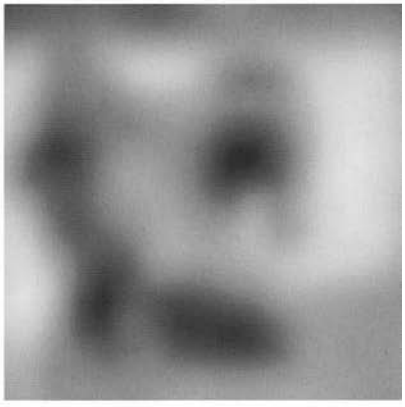
(a) Schicht $\mathbf{G}_0$

(b) Schicht $\mathbf{G}_{1,1}$

(c) Schicht $\mathbf{G}_{2,2}$

(d) Schicht $\mathbf{G}_{3,3}$

(e) Schicht $\mathbf{G}_{4,4}$



(f) Schicht $\mathbf{G}_{5,5}$

Bild 28.4: Expandierte Schichten einer Gauß-Pyramide. (a) bis (f): Schichten $\mathbf{G}_0$ (Original) bis $\mathbf{G}_{5,5}$. In der Schicht $\mathbf{G}_{4,4}$ sind die Streifen der Zebras nicht mehr zu sehen, während grobe Bildstrukturen noch deutlich hervortreten.

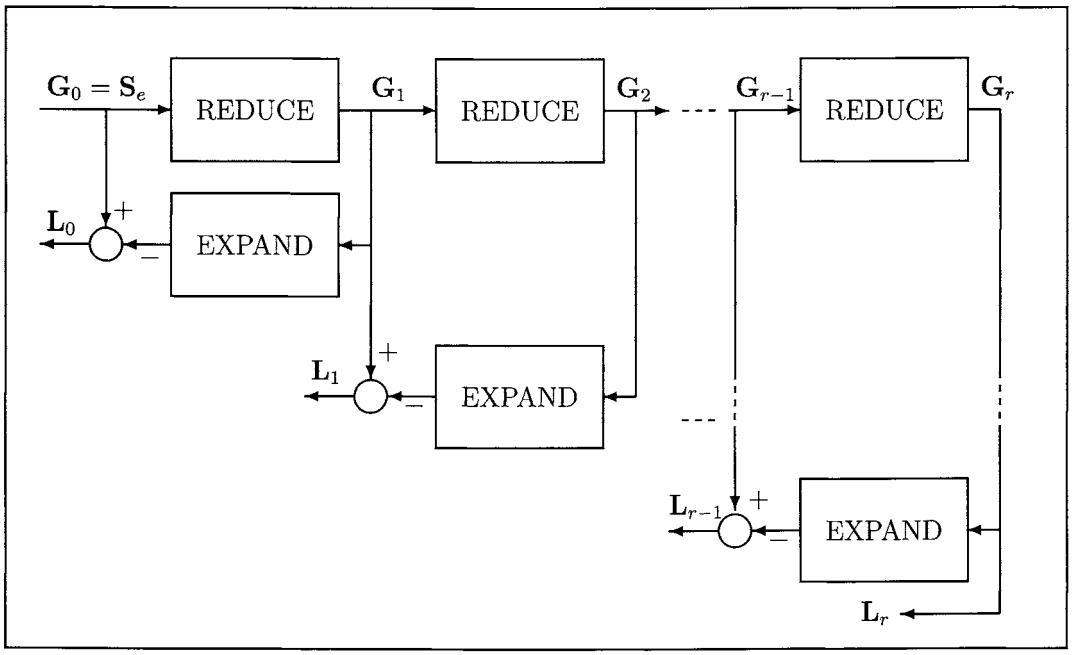
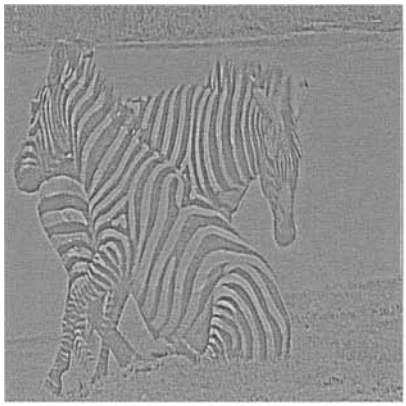


Bild 28.5: Blockdiagramm zum Aufbau einer Laplace-Pyramide.

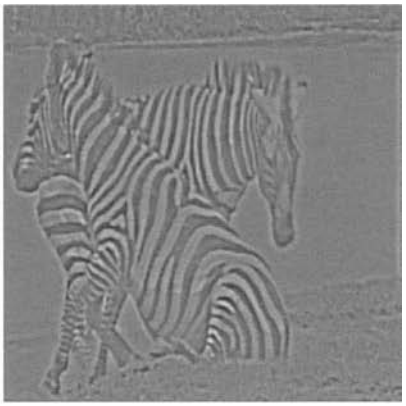
Stufe	Zeilen/Spalten	Wellenzahlindizes
L_0	$2^9 + 1 = 513$	129-256
L_1	$2^8 + 1 = 257$	65-128
L_2	$2^7 + 1 = 129$	33-64
L_3	$2^6 + 1 = 65$	17-32
L_4	$2^5 + 1 = 33$	9-16
L_5	$2^4 + 1 = 17$	5-8
L_6	$2^3 + 1 = 9$	3-4
L_7	$2^2 + 1 = 5$	1-2
L_8	$2^1 + 1 = 3$	1
L_9	$2^0 + 1 = 2$	-

28.6 Rekonstruktion des Originalbildes

Aus der Laplace-Pyramide kann das Originalbild $\mathbf{S}_e$ wieder rekonstruiert werden. Dies ist bei Anwendungen von Bedeutung, bei denen die einzelnen Schichten der Laplace-Pyramide bestimmten Verarbeitungsschritten unterzogen werden. Das Ergebnis dieser Verarbeitung ist dann das rücktransformierte Bild der möglicherweise veränderten Laplace-Pyramide.



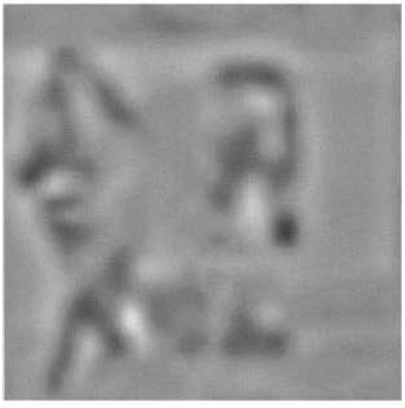
(a) Schicht $\mathbf{L}_0$



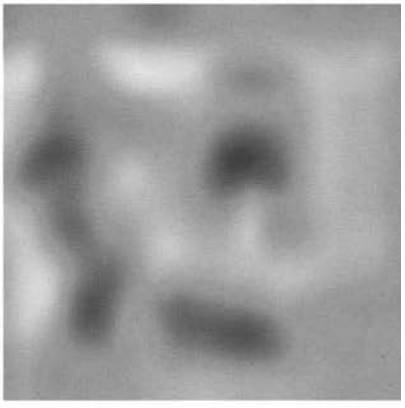
(b) Schicht $\mathbf{L}_{1,1}$

(c) Schicht $\mathbf{L}_{2,2}$

(d) Schicht $\mathbf{L}_{3,3}$



(e) Schicht $\mathbf{L}_{4,4}$



(f) Schicht $\mathbf{L}_{5,5}$

Bild 28.6: Expandierte Schichten einer Laplace-Pyramide. Die Bilder wurden um den Grauwert 127 angehoben und skaliert. (a) - (f) Schichten $\mathbf{L}_0$ bis $\mathbf{L}_{5,5}$. Die Streifen haben einen Wellenzahlindex im Bereich von 36. In Teilbild (c) treten sie am deutlichsten hervor. Die Streifen mit geringerem Wellenzahlindex sind in Teilbild (d) deutlich zu sehen.

Beispiele dazu werden z.B. in Abschnitt 28.10.1 gegeben. Im Folgenden wird nur die Rücktransformation einer Laplace-Pyramide zum originalen Grauwertbild beschrieben.

Zur Rücktransformation wird die Spitze der Laplace-Pyramide auf die Größe der darunterliegenden Schicht expandiert und zu dieser Schicht addiert. Dann wird diese Schicht expandiert und zur nächsten Schicht addiert. Das wird fortgesetzt, bis die Größe des Originals erreicht ist. Formal sieht die Rücktransformation folgendermaßen aus:

$$\begin{aligned} &\mathbf{S}_r = \mathbf{L}_r; \\ &\mathbf{S}_{i-1} = \mathbf{L}_{i-1} + \text{EXPAND}(\mathbf{S}_i); \quad i = r, r-1, ..., 1; \\ &\mathbf{S}_e = \mathbf{S}_0. \end{aligned} \tag{28.7}$$

Dieser formale Sachverhalt ist in Bild 28.7 als Blockdiagramm dargestellt.

Bei der Rücktransformation werden alle Bildinformationen, die beim Aufbau der Gauß-Pyramide herausgefiltert wurden, wieder zum Gesamtbild zusammengesetzt. Dies wird am Beispiel einer Laplace-Pyramide mit drei Stufen erläutert. Zunächst die Konstruktion der Laplace-Pyramide:

$$\begin{aligned} &\mathbf{L}_0 = \mathbf{S}_0 - \text{EXPAND}(\mathbf{S}_1); \\ &\mathbf{L}_1 = \mathbf{S}_1 - \text{EXPAND}(\mathbf{S}_2); \\ &\mathbf{L}_2 = \mathbf{S}_2. \end{aligned} \tag{28.8}$$

Die Rücktransformation:

$$\begin{aligned} &\mathbf{S}_e = \mathbf{S}_0; \\ &\mathbf{S}_0 = \mathbf{L}_0 + \text{EXPAND}\big(\mathbf{L}_1 + \text{EXPAND}(\mathbf{L}_2)\big). \end{aligned} \tag{28.9}$$

Substitution der einzelnen Ausdrücke:

$$\begin{aligned} &\mathbf{S}_0 = \mathbf{L}_0 + \text{EXPAND}\big(\mathbf{L}_1 + \text{EXPAND}(\mathbf{S}_2)\big); \\ &\mathbf{S}_0 = \mathbf{L}_0 + \text{EXPAND}\Big(\big(\mathbf{S}_1 - \text{EXPAND}(\mathbf{S}_2)\big) + \text{EXPAND}(\mathbf{S}_2)\Big); \\ &\mathbf{S}_0 = \mathbf{L}_0 + \text{EXPAND}(\mathbf{S}_1); \\ &\mathbf{S}_0 = \big(\mathbf{S}_0 - \text{EXPAND}(\mathbf{S}_1)\big) + \text{EXPAND}(\mathbf{S}_1); \\ &\mathbf{S}_0 = \mathbf{S}_0. \end{aligned} \tag{28.10}$$

Mit der Struktur einer Laplace-Pyramide liegt ein Bild in Schichten vor, die jeweils ganz bestimmte Ausschnitte des Spektrums des Originals enthalten. Die Verarbeitung von Bilddatenstrukturen dieser Art wird deshalb im englischen Sprachgebrauch als *multiresolution image processing* bezeichnet. Bildverarbeitungstechniken können gut in der richtigen

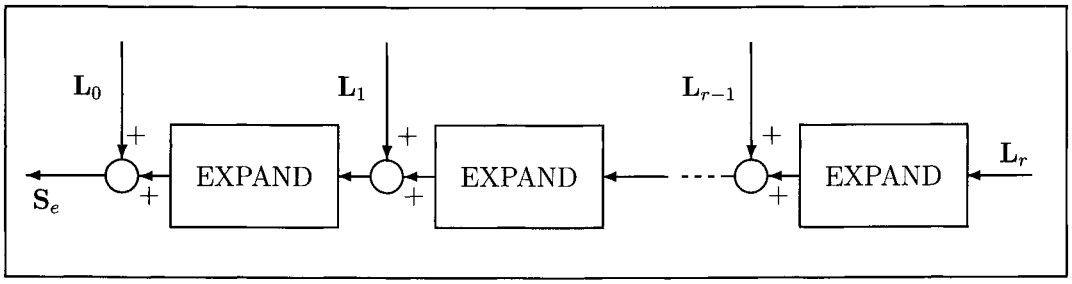


Bild 28.7: Blockdiagramm zur Rücktransformation einer Laplace-Pyramide.

Auflösungsstufe angewendet werden. Das ist diejenige Schicht der Pyramide, in der die interessierende Struktur gerade noch aufgelöst wird. Da die Bildgröße der Schichten in Richtung Spitze der Pyramide stark abnimmt, benötigt die gesamte Pyramide nur etwa $\frac{1}{3}$ mehr Speicherplatz als das Original in herkömmlicher Rasterdarstellung.

Zu bemerken wäre noch, dass bei praktischen Anwendungen nicht immer alle Pyramidenschichten von $\mathbf{L}_0$ bis $\mathbf{L}_r$ berechnet werden müssen. Wenn man z.B. weiß, dass sich die interessierenden Bildstrukturen am stärksten in den Schichten $\mathbf{L}_0$ bis $\mathbf{L}_3$ ausprägen, so kann man auf die weiteren Schichten verzichten. Die Schicht $\mathbf{L}_4$ ist dann identisch mit der Schicht $\mathbf{G}_4$ der Gauß-Pyramide.

28.7 Implementierung des REDUCE-Operators

Nachdem im vorhergehenden Abschnitt die grundlegenden Fragen zum Aufbau der Gauß- und Laplace-Pyramiden erläutert wurden, folgen hier einige Hinweise, die für die Implementierung wichtig sind.

Zunächst wird der in Abschnitt 28.4 verwendete REDUCE-Operator genauer untersucht. Beim Aufbau der Gauß-Pyramide wird eine höhere Schicht aus der darunterliegenden Schicht durch eine Glättung der Grauwerte und eine Verkleinerung gebildet. Der Glättungsoperator verwendet einen Filterkern $\mathbf{H} = (h(u,v))$ mit z.B. $5 \cdot 5$ Elementen. Die Reduzierung der Bildgröße wird dadurch erreicht, dass nur jeder zweite Bildpunkt der unteren Schicht zum Mittelpunkt der Glättungsoperation wird. Dadurch wird die darüberliegende Schicht in Zeilen- und Spaltenrichtung nahezu halbiert.

Mit diesen Voraussetzungen kann die REDUCE-Funktion mit folgendem Formalismus beschrieben werden:

$$\begin{aligned}
(g_0(x,y)) &= \mathbf{G}_0 = \mathbf{S}_e = (s_e(x,y)); \\
g_{i+1}(x,y) &= \sum_{u=-2}^{+2} \sum_{v=-2}^{+2} h(2+u, 2+v) g_i(2x+u, 2y+v); \qquad (28.11) \\
&\quad x = 0, ..., 2^{r-(i+1)}; \quad y = 0, ..., 2^{r-(i+1)};
\end{aligned}$$

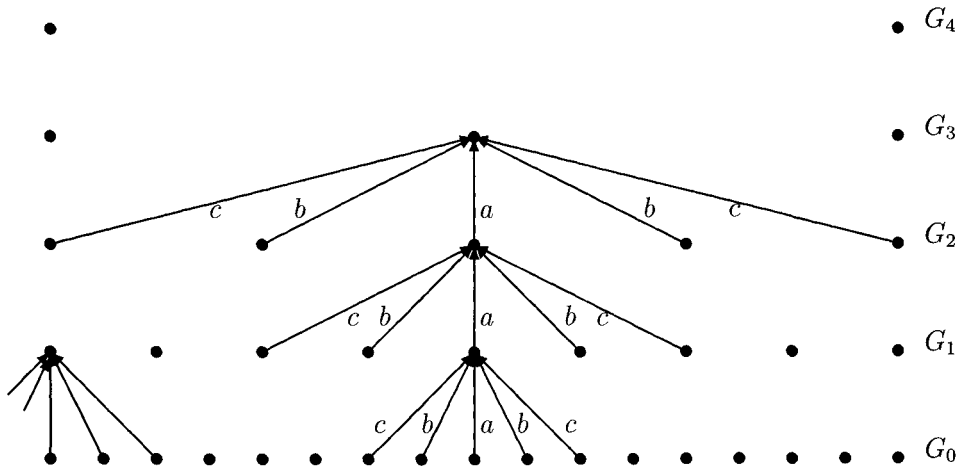


Bild 28.8: Glättungs- und Größenreduktionsoperationen beim Aufbau einer Gauß-Pyramide am Beispiel einer Bildzeile eines Bildes $\mathbf{G}_0 = \mathbf{S}_e$ mit der Zeilen- und Spaltenlänge $17 = (2^4 + 1)$. Am Rand liegende Bildpunkte müssen gesondert berechnet werden.

$$i = 0, 1, ..., r - 1.$$

Der Filterkern $\mathbf{H}$ soll einige Bedingungen erfüllen:

- $(h(u,v))$ ist *separabel*: $(h(u,v)) = (\hat{h}(u)) \cdot (\hat{h}(v))$. Diese Forderung wird vor allem aus programmiertechnischen Gründen aufgestellt, da die Bilder dann zeilen- und spaltenweise getrennt verarbeitet werden können.
- $(h(u,v))$ (und damit auch $(\hat{h})$) ist *symmetrisch*: $(\hat{h}(u)) = (\hat{h}(v)) = (c \quad b \quad a \quad b \quad c)$.
- $(\hat{h})$ ist *normiert*: $a + 2b + 2c = 1$.
- Jeder Bildpunkt einer Schicht i leistet zu den Bildpunkten der Schicht $i + 1$ denselben Beitrag, obwohl nur jeder zweite Bildpunkt zentraler Bildpunkt der Glättungsoperation wird. Diese Forderung hängt mit der Reduzierung der Auflösung bei der REDUCE-Operation zusammen. Da jeder gerade Bildpunkt einmal zentraler Punkt (Gewicht a) und zweimal Randpunkt (Gewicht c) ist und jeder ungerade Punkt zweimal mit dem Gewicht b eingeht, muss gelten: $a + 2c = 2b$.

Die letzte Forderung ist in Bild 28.8 schematisch dargestellt. Hier sind die Glättungs- und Größenreduktionsoperationen beim Aufbau einer Gauß-Pyramide am Beispiel einer Bildzeile eines Bildes $\mathbf{G}_0 = \mathbf{S}_e$ mit der Zeilen- und Spaltenlänge $17 = (2^4 + 1)$ verdeutlicht.

Die Zeilen- und Spaltenlängen der Schichten $\mathbf{G}_1$, $\mathbf{G}_2$, $\mathbf{G}_3$ und $\mathbf{G}_4$ sind 9, 5, 3 und 2. Am Rand liegende Bildpunkte müssen gesondert berechnet werden. Dies wird weiter unten erläutert.

Aus den Forderungen an $\hat{h}$ können die Beziehungen zwischen den Parametern a, b und c leicht berechnet werden:

$$a = \text{freie Variable}; \quad b = \frac{1}{4}; \quad c = \frac{1}{4} - \frac{1}{2}a. \tag{28.12}$$

Beispiele für Filterkerne sind das *Binomialfilter* mit $a = \frac{3}{8}$

$$(\hat{h}) = \frac{1}{16}\begin{pmatrix} 1 & 4 & 6 & 4 & 1 \end{pmatrix}, \tag{28.13}$$

und das *Gauß-Filter* mit $a = \frac{2}{5}$

$$(\hat{h}) = \frac{1}{20}\begin{pmatrix} 1 & 5 & 8 & 5 & 1 \end{pmatrix}. \tag{28.14}$$

Weitere Bemerkungen zur Wahl des freien Parameters a sind in Abschnitt 28.9 zusammengefasst.

Unter Verwendung der Separabilität von $(h(u,v)) = (\hat{h}(u)) \cdot (\hat{h}(v))$ werden die Glättung und die Reduktion zunächst über alle Bildzeilen und dann über alle Bildspalten durchgeführt. Es ergeben sich folgende Formeln:

$$\mathbf{G}_0 = \mathbf{S}_e; \tag{28.15}$$

$$\begin{aligned} \tilde{g}_{i+1}(x,y) &= \sum_{v=-2}^{+2} \hat{h}(2+v) g_i(x, 2y+v); \\ &\quad y = 0, ..., 2^{r-(i+1)}; \\ &\quad x = 0, ..., 2^{r-i}; \\ g_{i+1}(x,y) &= \sum_{u=-2}^{+2} \hat{h}(2+u) \tilde{g}_{i+1}(2x+u, y); \\ &\quad x = 0, ..., 2^{r-(i+1)}; \\ &\quad y = 0, ..., 2^{r-(i+1)}; \\ &\quad i = 0, 1, ..., r-1. \end{aligned}$$

Wie bereits oben erwähnt und in Bild 28.8 angedeutet, müssen die Randpunkte der einzelnen Schichten gesondert behandelt werden. Da in der hier beschriebenen Implementierung $5 \cdot 5$–Filterkerne verwendet werden, ist der Rand jeweils 2 Pixel breit. Die außerhalb des Bildes liegenden Pixel werden extrapoliert und zwar für die Zeilen

$$\begin{aligned} g_i(-1,y) &= 2g_i(0,y) - g_i(1,y), \\ g_i(-2,y) &= 2g_i(0,y) - g_i(2,y) \end{aligned} \tag{28.16}$$

und für die Spalten

$$\begin{aligned} g_i(x,-1) &= 2g_i(x,0) - g_i(x,1), \\ g_i(x,-2) &= 2g_i(x,0) - g_i(x,2). \end{aligned} \tag{28.17}$$

Bei dieser Extrapolation bleibt die erste Ableitung am Bildrand konstant und die zweite Ableitung wird null. Dadurch werden aber Veränderungen im Grauwertverlauf an den Rändern bei höheren Pyramidenstufen verstärkt, was zu Randverfälschungen führt. Dazu wird eine Extrapolation vorgeschlagen, bei der auch die erste Ableitung null ist. Für die Spalten erhält man dann:

$$\begin{aligned} g_i(x,-1) &= g_i(x,0); \\ g_i(x,-2) &= g_i(x,0). \end{aligned} \tag{28.18}$$

Die Extrapolationen für den rechten und den unteren Bildrand lauten sinngemäß.

28.8 Implementierung des EXPAND-Operators

Mit der EXPAND-Operation wird eine Schicht auf die Größe der darunterliegenden Schicht ausgedehnt. Dazu müssen die Zeilen- und die Spaltenanzahl verdoppelt werden, was zur Folge hat, dass Bildpunkte interpoliert werden müssen. Da die EXPAND-Operation als inverse Operation von REDUCE aufgefasst werden kann, wird ebenfalls die Filtermaske $\mathbf{H} = (h(u,v))$, hier allerdings zur Interpolation, verwendet. Da bei der REDUCE-Operation nur die Bildpunkte mit geradzahligem Zeilen-/Spaltenindex zentrale Punkte bei der Filterung werden (siehe Bild 28.8), werden sie in zwei Teilmengen aufgeteilt: Einmal die Bildpunkte, die mit dem Gewicht b in die Glättung eingehen, und zum anderen diejenigen, die mit den Gewichten a und c berücksichtigt werden.

Diesem Sachverhalt wird bei der EXPAND-Operation dadurch Rechnung getragen, dass bei der Interpolation die geradzahligen Positionen einer vergrößerten Zeile (Spalte) aus drei Bildpunkten der Ausgangszeile (-spalte) mit den Gewichten a und c interpoliert werden und die ungeraden Positionen nur aus zwei Bildpunkten mit dem Gewicht b. Dies ist in Bild 28.9 veranschaulicht. Hier wurde bereits berücksichtigt, dass auch die EXPAND-Operation separabel ist.

Durch diese Vorgehensweise wird allerdings die Forderung nicht aufrechterhalten, dass auch die EXPAND-Operation normiert ist, denn es gilt nach Abschnitt 28.4

$$a + 2b + 2c = 1. \tag{28.19}$$

Außerdem gilt wegen der vierten Filterbedingung:

$$a + 2c = 2b. \tag{28.20}$$

Daraus lässt sich leicht ableiten:

$$a + 2c = \frac{1}{2}, \tag{28.21}$$

$$2b = \frac{1}{2}.$$

Das heißt aber, dass man die Gewichtung bei der Interpolation in Zeilen- und Spaltenrichtung verdoppeln muss, was bei dem nicht separierten Filterkern einen Faktor vier und bei der separierten Form jeweils den Faktor zwei bedeutet. Für die folgenden Formeln wird die Notation von (28.5) verwendet:

$$\begin{aligned} g_{i,k+1}(x,y) &= 4 \sum_{u=-2}^{+2} \sum_{v=-2}^{+2} h(2+u,2+v) g_{i,k}\Big(\frac{x+u}{2}, \frac{y+v}{2}\Big); \\ & x = 0, ..., 2^{r-(i-1)+k}; \\ & y = 0, ..., 2^{r-(i-1)+k}; \\ & i = 1, ..., r; \\ & k = 0, 1, ..., i. \end{aligned} \tag{28.22}$$

Die beiden separierten Formeln lauten:

$$\begin{aligned} \tilde{g}_{i,k+1}(x,y) &= 2 \sum_{v=-2}^{+2} \hat{h}(2+v) g_{i,k}\Big(x, \frac{y+v}{2}\Big); \\ & y = 0, ..., 2^{r-(i-1)+k}; \\ & x = 0, ..., 2^{r-i+k}; \\ g_{i,k+1}(x,y) &= 2 \sum_{u=-2}^{+2} \hat{h}(2+u) \tilde{g}_{i,k+1}\Big(\frac{x+u}{2}, y\Big); \\ & x = 0, ..., 2^{r-(i-1)+k}; \\ & y = 0, ..., 2^{r-(i-1)+k}; \\ & i = 1, ..., r; \\ & k = 0, 1, ..., i. \end{aligned} \tag{28.23}$$

In (28.22) und (28.23) werden nur diejenigen Positionen verwendet, bei denen sich für den Zeilen- und Spaltenzähler ganzzahlige Werte ergeben. Dadurch wird die oben erwähnte Klasseneinteilung der Bildpunkte bei der REDUCE-Operation berücksichtigt.

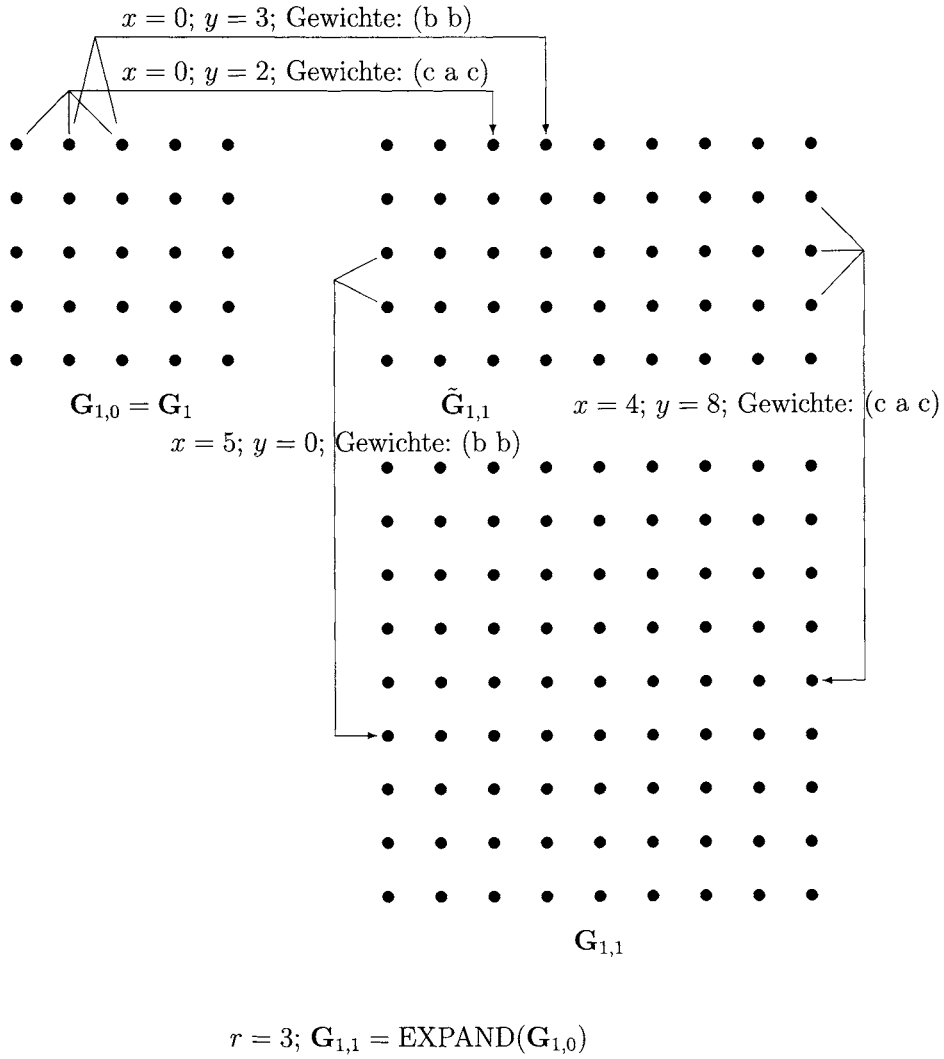


Bild 28.9: EXPAND-Schritt von $\mathbf{G}_1$, über das Hilfsbild $\tilde{\mathbf{G}}_{1,1}$ zur Schicht $\mathbf{G}_{1,1}$. Für das Beispiel wurde $r = 3$ gewählt.

Die Formeln (28.23) werden im Folgenden anhand eines Beispiels erläutert. Schematisch ist dieses Beispiel in Bild 28.9 dargestellt. Es wird ein Eingabebild $\mathbf{S}_e$ der Größe $9 \cdot 9$ verwendet. Also ist $r = 3$. Die Schichten der Gauß-Pyramide sind $\mathbf{G}_0$, $\mathbf{G}_1$, $\mathbf{G}_2$ und $\mathbf{G}_3$, mit den Zeilen- und Spaltenlängen 9, 5, 3 und 2.

Hier wird nun der EXPAND-Schritt von $\mathbf{G}_{1,0} = \mathbf{G}_1$, über das Hilfsbild $\tilde{\mathbf{G}}_{1,1}$ zum Bild $\mathbf{G}_{1,1}$ erläutert. Gemäß Formel (28.23) wird zur Berechnung der $\tilde{g}_{1,1}(x, y)$-Werte für eine Bildzeile x über die Spalten y interpoliert. So werden beispielsweise für $x = 0$ und $y = 2$ folgende Rasterpositionen berechnet:

$$(0,0),\ (0,\tfrac{1}{2}),\ (0,1),\ (0,\tfrac{3}{2}),\ (0,2)$$

Für $x = 0$ und $y = 3$ ergeben sich:

$$(0,\tfrac{1}{2}),\ (0,1),\ (0,\tfrac{3}{2}),\ (0,2),\ (0,\tfrac{5}{2})$$

Zur Interpolation werden nur die Rasterpositionen mit ganzzahligen Werten verwendet. Somit gehen in die Summation für $x = 0$ und $y = 2$ die ersten drei Pixel von $\mathbf{G}_{1,0}$ mit den Gewichten $(c \quad a \quad c)$ ein. Für $x = 0$ und $y = 3$ werden die Grauwerte in den Positionen $(0,1)$ und $(0,2)$ mit den Gewichten $(b \quad b)$ verwendet. Zum Ausgleich der fehlenden Positionen und zur Einhaltung der Normierungsforderung werden die gewichteten Summen mit 2 multipliziert. Sinngemäß wird im zweiten Teil der Formel (28.23) die Interpolation über die Zeilen durchgeführt. Das Ergebnis ist in diesem Beispiel ein $9 \cdot 9$–Bild $\mathbf{G}_{1,1}$, das die Größe von $\mathbf{G}_0$ besitzt.

28.9 Frequenzverhalten und Wahl des freien Parameters a

Die oben angegebenen Frequenzbereiche für die Gauß- und Laplace-Pyramiden beruhen auf theoretischen Überlegungen, basierend auf der Vorgehensweise beim Aufbau der Pyramidenstrukturen. Der tatsächliche Frequenzgehalt der einzelnen Pyramidenstufen hängt jedoch von der Filtermaske $\mathbf{H} = (h(u, v))$ ab, die sowohl bei der REDUCE- als auch bei der EXPAND-Operation verwendet wird. Da dieses Filter in der diskutierten Form ausschließlich vom Parameter a bestimmt wird, ist die richtige Wahl dieses Parameters wesentlich.

Zur Untersuchung des Filters $\mathbf{H}$ berechnet man das Frequenzspektrum der Impulsantwort. Da die Filtermaske symmetrisch ist, ist das Filter nahezu richtungsunabhängig, und es genügt zur Untersuchung ein zweidimensionaler Schnitt durch das zweidimensionale Spektrum der Impulsantwort. In [Burt84] wird für den freien Parameter der Wert $a = \frac{2}{5} = 0.4$ vorgeschlagen. Damit erhält man mit (28.12) das Gauß-Filter:

$$\begin{aligned} a &= \frac{2}{5} = 0.4 \\ b &= \frac{1}{4}; \\ c &= \frac{1}{4} - \frac{1}{2}a = \frac{1}{20} \end{aligned} \tag{28.24}$$

und

$$(\hat{h}(u)) = (\hat{h}(v)) = \frac{1}{20}(1 \quad 5 \quad 8 \quad 5 \quad 1). \tag{28.25}$$

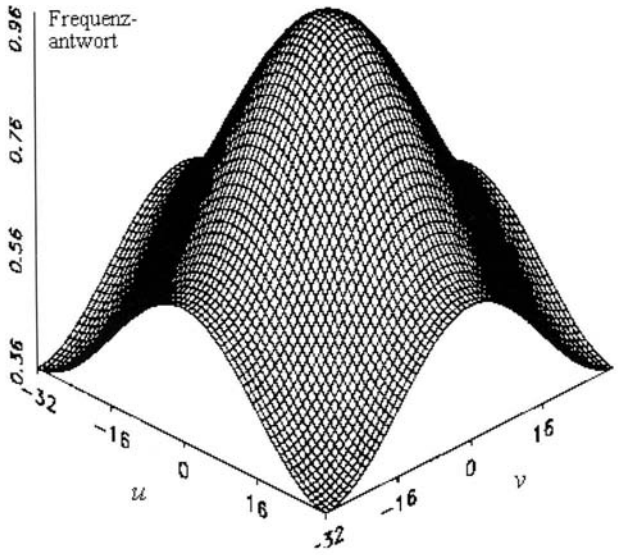
In Bild 28.10-a ist die Frequenzantwort der REDUCE-Funktion ($a = 0.4$, Gauß-Filter) gezeigt. Die Zahlenangaben für die Wellenzahlindizes beziehen sich auf eine Pyramidenstufe der Größe 129·129 Bildpunkte. In Bild 28.10-b ist ein Schnitt durch das dreidimensionale Spektrum der Impulsantwort des Gauß-Filters dargestellt. Man sieht, dass bei höheren Wellenzahlindizes die Signale stärker gedämpft werden. So ist z.B. für $u = 8$ die Durchlassrate ca. 95%, während sie für $u = 16$ nur mehr ca. 75% ist.

Da beim Aufbau der Gauß-Pyramide von Schicht zu Schicht die Zeilen- und Spaltenanzahl halbiert wird, müsste die REDUCE-Operation alle Strukturen mit Wellenzahlindizes $u > \frac{1}{4}M_{\mathbf{G}_i}$ herausfiltern, wobei $M_{\mathbf{G}_i}$ die Zeilen- (Spalten-)länge der zu reduzierenden Schicht ist. Beispielsweise müssten beim Übergang von einer Pyramidenschicht der Größe 129·129 zur nächsten Schicht der Größe 65·65 alle Wellenzahlindizes u im Bereich zwischen 33 und 64 herausgefiltert werden. Ein Vergleich mit der Frequenzantwort des Gauß-Filters zeigt, dass das reale Frequenzverhalten dem nicht entspricht: So ist z.B. bei $u = 32$ im obigen Beispiel eine Durchlassrate von immerhin noch 60% vorhanden. Dieser Sachverhalt kann bei periodischen Bildstrukturen dafür verantwortlich sein, dass bei der Rücktransformation Aliasingeffekte auftreten. Diese Untersuchungen des Frequenzverhaltens kann man nun auf die EXPAND-Funktion, die Gauß-Pyramide und die Laplace-Pyramide ausdehnen.

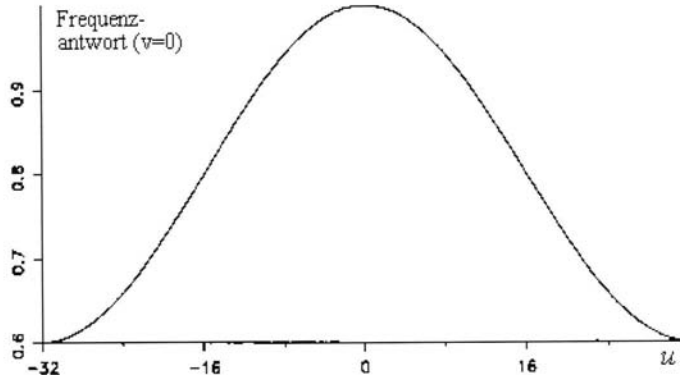
Für die Praxis ist Tabelle 28.1 sehr hilfreich. Sie beschreibt für unterschiedliche Werte von a das Filterverhalten der einzelnen Schichten der Laplace-Pyramiden. Die Tabelle ist für Bilder der Größe 512·512 ausgelegt. Sie kann jedoch leicht auf kleinere Bildgrößen übertragen werden. Die Spalte **max** enthält die maximalen Wellenzahlindizes der einzelnen Schichten. Die Spalte **75%** enthält Wellenzahlindizes, bei denen mindestens 75% der maximalen Durchlassrate erreicht werden, die Spalten **50%** und **25%** sind sinngemäß zu interpretieren.

Wie kann Tabelle 28.1 genutzt werden? Dazu ein Beispiel, das sich auf Bild 28.6 bezieht. Verwendet wurde ein Gauß-Filter ($a = 0.4$). Die Streifen der Zebras sind etwa 18 Pixel breit. Das entspricht bei einem $512 \cdot 512$–Bild einem Wellenzahlindex von etwa $u = 14$. Das bedeutet, dass die Streifen in der Schicht $\mathbf{L}_2$ deutlich hervortreten und in $\mathbf{L}_3$ bereits gedämpft sind. Ab der Schicht $\mathbf{L}_4$ sind sie nicht mehr zu sehen. Die Bilder bestätigen dies.

Bei Verwendung von Tabelle 28.1 muss man bei einer praktischen Anwendung wissen, in welchem Frequenzbereich sich die zu untersuchende Struktur ausprägt. Anhand der Tabelle kann man dann den passenden Parameter a wählen. Außerdem gestattet es die Tabelle, die richtige Schicht der Pyramide zu finden, in der die Struktur am deutlichsten zu finden ist.



(a)



(b)

Bild 28.10: (a) Frequenzantwort (Impulsantwort) der REDUCE-Operation für das Gauß-Filter ($a = 0.4$). Die Zahlenangaben der Wellenzahlindizes beziehen sich auf eine Pyramidenstufe mit $129 \cdot 129$ Bildpunkten. (b) Schnitt durch das zweidimensionale Spektrum der Impulsantwort des Gauß-Filters. Man sieht, dass z.B. Wellenzahlindizes der Größe $u = 16$ auf etwa 75% ihres ursprünglichen Wertes gedämpft werden. Am Rand sollten die Wellenzahlindizes ganz herausgefiltert werden. Das ist aber nicht der Fall, so dass bei bestimmten periodischen Bildstrukturen Aliasingeffekte zu erwarten sind.

Parameter a	Schicht	max	75%	50%	25%
0.30	$\mathbf{L}_0$	196	80-256	57-256	37-256
	$\mathbf{L}_1$	49	32- 74	23- 98	16-128
	$\mathbf{L}_2$	24	16- 35	12- 44	8- 56
	$\mathbf{L}_3$	12	8- 17	6- 22	4- 28
	$\mathbf{L}_4$	6	4- 8	3- 11	2- 14
	$\mathbf{L}_5$	3	2- 4	2- 5	1- 7
	$\mathbf{L}_6$	2	1- 2	1- 2	1- 4
0.35	$\mathbf{L}_0$	211	92-256	64-256	41-256
	$\mathbf{L}_1$	55	35- 94	26-128	17-128
	$\mathbf{L}_2$	26	17- 41	13- 55	9- 64
	$\mathbf{L}_3$	13	9- 20	7- 26	5- 32
	$\mathbf{L}_4$	7	5- 10	4- 13	2- 16
	$\mathbf{L}_5$	3	3- 5	2- 6	1- 8
	$\mathbf{L}_6$	2	2- 2	1- 3	1- 4
0.40	$\mathbf{L}_0$	256	112-256	76-256	48-256
	$\mathbf{L}_1$	68	40-128	29-128	19-128
	$\mathbf{L}_2$	31	19- 64	14- 64	9- 64
	$\mathbf{L}_3$	15	10- 30	7- 32	5- 32
	$\mathbf{L}_4$	7	5- 14	4- 16	3- 16
	$\mathbf{L}_5$	4	3- 7	2- 8	2- 8
	$\mathbf{L}_6$	2	2- 3	1- 4	1- 4
0.45	$\mathbf{L}_0$	256	144-256	97-256	61-256
	$\mathbf{L}_1$	128	63-128	42-128	26-128
	$\mathbf{L}_2$	64	29- 64	20- 64	13- 64
	$\mathbf{L}_3$	32	14- 32	10- 32	6- 32
	$\mathbf{L}_4$	16	7- 16	5- 16	3- 16
	$\mathbf{L}_5$	8	4- 8	3- 8	2- 8
	$\mathbf{L}_6$	4	2- 4	2- 4	1- 4

Tabelle 28.1: Filterverhalten der einzelnen Schichten der Laplace-Pyramide für unterschiedliche Werte des Parameters a.

28.10 Anwendungsbeispiele zu den Laplace-Pyramiden

28.10.1 Verwendung einzelner Schichten

Eine einfache Verwendung der Laplace-Pyramiden ist die Verarbeitung einzelner Schichten. Da jede Schicht bestimmte Wellenzahlindexbereiche enthält, kann man gezielt eine Schicht auswählen und versuchen, die interessierenden Strukturen in dieser Schicht zu entdecken und weiterzuverarbeiten. Dabei ist es möglich, entweder in der größenreduzierten Form die weiteren Verarbeitungsschritte anzuwenden oder die Schicht durch die EXPAND-Operation auf die Größe des Originals zu bringen.

Auch die Verwendung mehrerer Schichten kann sinnvoll sein. Dies wird anhand eines Beispiels erläutert: Bei einer konkreten Anwendung könnten die Wellenzahlindexintervalle 5 bis 16 und 33 bis 64 die interessierenden Bildstrukturen enthalten. Man wird dann mit der Schicht $\mathbf{L}_5$ beginnen und sie auf die Größe der darunter liegenden Schicht expandieren:

$$\mathbf{L}_{5,1} = \text{EXPAND}(\mathbf{L}_{5,0}), \text{ mit } \mathbf{L}_{5,0} = \mathbf{L}_5. \tag{28.26}$$

Dazu wird jetzt $\mathbf{L}_{4,0}$ addiert:

$$\tilde{\mathbf{L}}_{4,0} = \mathbf{L}_{4,0} + \mathbf{L}_{5,1}. \tag{28.27}$$

Dieses Ergebnis muss jetzt zweimal expandiert werden, dann wird die Schicht $\mathbf{L}_2$ dazu addiert:

$$\tilde{\mathbf{L}}_{2,0} = \tilde{\mathbf{L}}_{4,2} + \mathbf{L}_{2,0}. \tag{28.28}$$

Das so entstandene Bild enthält nun die gewünschten Wellenzahlindexbereiche. Es kann bei Bedarf abschließend auf die Größe des Originals expandiert werden. Allerdings ist zu beachten, dass durch das reale Frequenzverhalten beim Aufbau der Laplace-Pyramide die Wellenzahlindexbereiche nicht so exakt ausgewählt werden können, wie es in diesem Beispiel gemacht wurde. Hilfreich kann dabei Tabelle 28.1 sein.

28.10.2 Mosaicing

In manchen Anwendungsbereichen ist es notwendig, den Bildinhalt verschiedener Originalbilder in einem Ergebnisbild zusammenzufassen. Dieser Vorgang wird *mosaicing* genannt. Beispiele dazu sind Fotomontagen im grafischen Gewerbe, Phantombilder, bei denen Gesichter aus Teilstücken zusammengesetzt werden oder architektonische Aufnahmen, mit denen die Verträglichkeit von baulichen Veränderungen mit der Umgebung vor der Durchführung der Baumaßnahmen beurteilt werden soll.

Als beispielhafte Problemstellung soll zunächst das Zusammenfügen von zwei Bildhälften, linke Hälfte vom ersten Bild und rechte Hälfte vom zweiten Bild, dienen. Werden die beiden Hälften ohne weitere Verarbeitung nur zusammengefügt, so wird immer eine Nahtstelle zu sehen sein.

Eine Mittelung entlang der beiden Übergangsbereiche wäre ein nächster Lösungsversuch, der sich aber ebenfalls als nicht ausreichend herausstellt, da die Breite des Übergangsbereichs abhängig vom Bildinhalt ist: Bei sehr fein strukturierten Bildinformationen muss der Übergangsbereich kleiner gewählt werden als bei grob strukturierten. Diese Wahl mag bei Einzelproblemen möglich sein, es lässt sich daraus aber kaum ein allgemeines Verfahren ableiten. Außerdem wird bei dieser Vorgehensweise die Schnittkante nur unschärfer und fällt dadurch lediglich etwas weniger auf.

Mit den Laplace-Pyramiden hat man eine Datenstruktur, die hervorragend zur Lösung dieser Problemstellung geeignet ist. Der Lösungsweg sieht wie folgt aus: Zu den beiden Eingabebildern $\mathbf{S}_{e1}$ und $\mathbf{S}_{e2}$ werden die Laplace-Pyramiden $\mathbf{L}^{(e1)}$ und $\mathbf{L}^{(e2)}$ berechnet. Aus diesen beiden Laplace-Pyramiden wird nun ausgehend von der Spitze eine neue Laplace-Pyramide $\mathbf{L}^{(a)}$ aufgebaut, deren linke Hälfte aus den Schichten von $\mathbf{L}^{(e1)}$ gebildet wird und deren rechte Hälfte von $\mathbf{L}^{(e2)}$ stammt. Die mittlere Bildspalte nimmt eine Sonderstellung ein: Ihre Grauwerte ergeben sich durch Mittelung der entsprechenden Bildspalten der beiden Eingabepyramiden. Im Einzelnen sehen diese Operationen wie folgt aus:

$$l_i^{(a)}(x,y) = \begin{cases} l_i^{(e1)}(x,y), & \text{falls } (x,y) \text{ in der linken Bildhälfte liegt,} \\ \frac{l_i^{(e1)}(x,y)+l_i^{(e2)}(x,y)}{2}, & \text{falls } (x,y) \text{ im Übergangsbereich liegt,} \\ l_i^{(e2)}(x,y), & \text{falls } (x,y) \text{ in der rechten Bildhälfte liegt.} \end{cases} \tag{28.29}$$

Der Index i läuft hier über die Schichten der Pyramiden, also von r bis 0. Abschließend wird aus $\mathbf{L}^{(a)}$ mit dem in (28.7) angegebenen Verfahren das Ergebnisbild $\mathbf{S}_a$ erzeugt. Da bei der Umwandlung der Laplace-Pyramide $\mathbf{L}^{(a)}$ in das Ausgabebild $\mathbf{S}_a$ von der Spitze ausgehend die einzelnen Schichten expandiert und akkumuliert werden, werden dadurch hohe und niedere Ortsfrequenzen beeinflusst, sodass im zusammengefügten Bild keine Nahtstelle zu sehen ist.

Die Bildfolge 28.11 zeigt ein Beispiel dazu: Oben sind die beiden Originale abgebildet, und in den unteren Bildern wurde jeweils eine Hälfte des linken oberen Bildes mit einer Hälfte des rechten oberen Bildes zusammengefügt.

Das Zusammenfügen einer linken und rechten Bildhälfte wurde hier nur als einfaches Beispiel zur Erläuterung der Vorgehensweise verwendet. In praktischen Anwendungen tritt dagegen das Problem auf, dass von den beiden Eingabebildern beliebige Bildausschnitte zusammenzufügen sind. Auch diese allgemeinere Problemstellung, die das oben geschilderte, einfache Beispiel beinhaltet, lässt sich mit der Datenstruktur der Laplace-Pyramiden lösen.

Die Vorgehensweise ist wie folgt: Die beiden Eingabebilder werden wieder mit $\mathbf{S}_{e1}$ und $\mathbf{S}_{e2}$ und das Ausgabebild mit $\mathbf{S}_a$ bezeichnet. Zusätzlich wird ein Maskenbinärbild $\mathbf{S}_m$ benötigt, mit dem angegeben wird, welche Bildteile aus $\mathbf{S}_{e1}$ und welche aus $\mathbf{S}_{e2}$ in das Ausgabebild $\mathbf{S}_a$ zu übernehmen sind. Dieses Maskenbild muss vor dem Vorgang des Zusammenfügens von Bildteilen erstellt werden. Man kann dazu handgezeichnete Skizzen digitalisieren oder computerunterstützte Zeichenprogramme verwenden. Auch binarisierte Bilder sind als Masken geeignet. Das Ergebnis muss ein Binärbild sein, in dem z.B. die

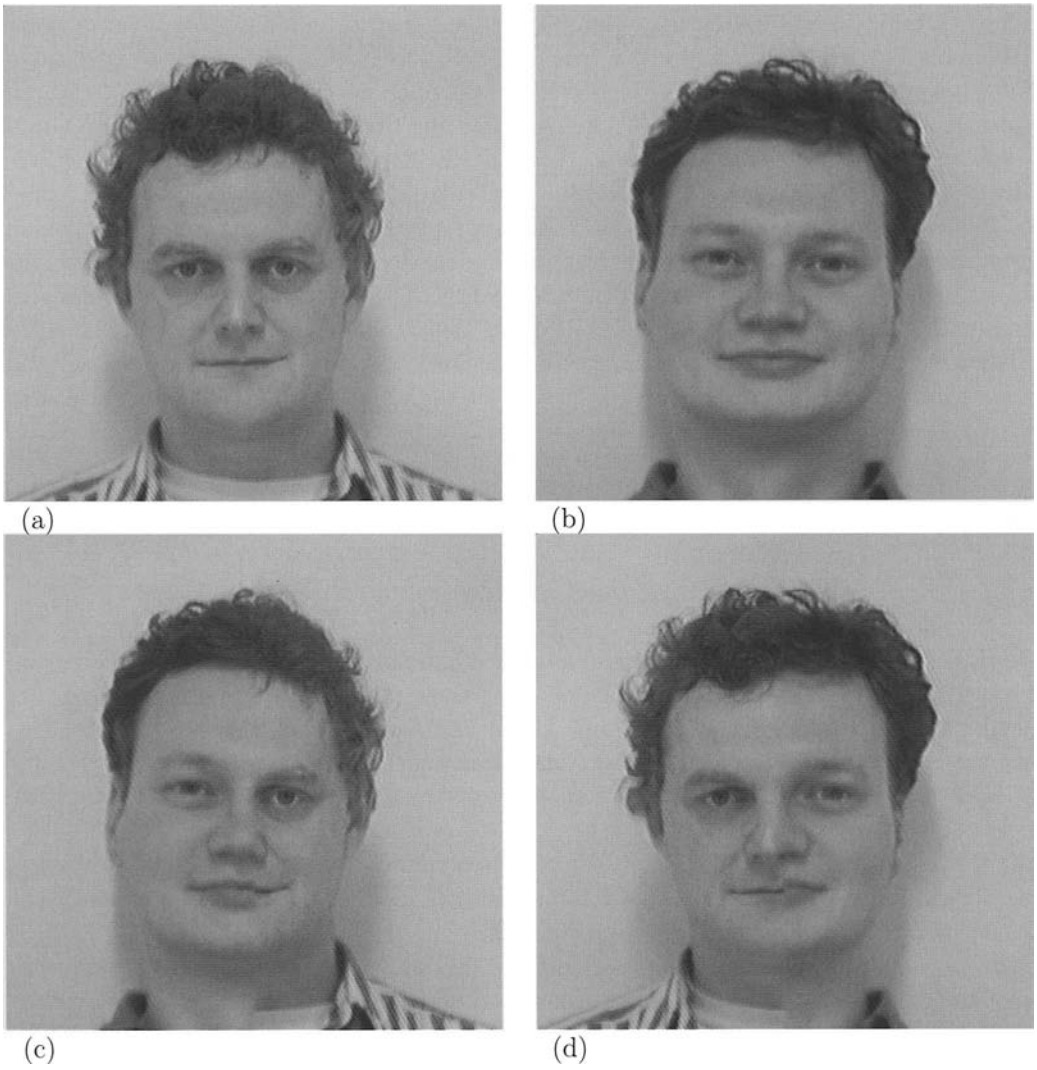


Bild 28.11: Zusammenfügen der linken und rechten Bildhälften mit Hilfe der Laplace-Pyramiden. (a) und (b) Originalbilder. (c) und (d): Über die Laplace-Pyramiden zusammengefügte Bildhälften. Die sichtbaren Ungenauigkeiten im Bereich des Mundes sind durch die unterschiedlichen Mundformen zu erklären. Hier wäre eine geometrische Korrektur vor dem Zusammenfügen sinnvoll.

Bildpunkte mit dem Grauwert 0 bzw. 1 signalisieren, dass an dieser Stelle im Ausgabebild die Information vom Eingabebild $\mathbf{S}_{e1}$ bzw. $\mathbf{S}_{e2}$ zu verwenden ist.

Zu den beiden Eingabebildern werden nun die Laplace-Pyramiden $\mathbf{L}^{(e1)}$ und $\mathbf{L}^{(e2)}$ erzeugt. Beim Aufbau der Pyramide $\mathbf{G}^{(m)}$ des Maskenbildes ist zu beachten, dass hier nur die Größenreduktion durchgeführt wird. Es entfallen im Reduktionsschritt die Tiefpassfilterung (Parameter für die Filtermaske $\mathbf{H} : a = 1, b = 0, c = 0$) und die anschließende Bildung der Laplace-Pyramide. Außerdem müssen in der Pyramide die Ränder zwischen den 0/1-Bereichen markiert werden, etwa mit dem Grauwert 2. Bei der Ermittlung der Ränder kann, je nach Problemstellung, mit 4- oder 8-Nachbarschaft gearbeitet werden.

Als Nächstes wird die Laplace-Pyramide $\mathbf{L}^{(a)}$ des Ausgabebildes berechnet. Wenn die Maskenpyramide an der gerade betrachteten Position den Wert 0 bzw. 1 aufweist, wird in die Ausgabepyramide der Grauwert der Laplace-Pyramide $\mathbf{L}^{(e1)}$ bzw. $\mathbf{L}^{(e2)}$ übernommen. Handelt es sich um einen Randpunkt (Wert 2 in der Maskenpyramide), so wird in die Ausgabepyramide der Mittelwert der beiden Grauwerte der Eingabepyramiden übernommen.

$$l_i^{(a)}(x,y) = \begin{cases} l_i^{(e1)}(x,y), & \text{falls } g_i^{(m)}(x,y) = 0, \\ \frac{l_i^{(e1)}(x,y) + l_i^{(e2)}(x,y)}{2}, & \text{falls } g_i^{(m)}(x,y) = 2, \\ l_i^{(e2)}(x,y), & \text{falls } g_i^{(m)}(x,y) = 1. \end{cases} \tag{28.30}$$

Nachdem die Ausgabepyramide $\mathbf{L}^{(a)}$ berechnet wurde, wird mit dem in (28.7) beschriebenen Verfahren das Ausgabebild $\mathbf{S}_a$ erzeugt. Die Bildfolge 28.12 zeigt Beispiele: Bild 28.12-a und 28.12-b sind die beiden Eingabebilder. Die Maskenbilder 28.12-c und 28.12-d wurden interaktiv mit einem Zeichenprogramm erzeugt. Die Bilder 28.12-e bis 28.12-h zeigen die Ergebnisse des *mosaicing* mit unterschiedlichen Kombinationen. Weitere Beispiele zum *mosaicing* enthält die Bildfolge 28.13.

Für die Behandlung der Übergangsbildpunkte im Maskenbild sind auch noch andere Varianten möglich: Falls sich die beiden Eingabebilder überlappen, kann auch ein Verfahren eingesetzt werden, bei dem die Übergangskanten im Maskenbild nicht mit einem starren Wert markiert sind, sondern stetig verlaufen. Dazu wird zum Maskenbild ebenfalls eine Laplace-Pyramide mit der üblichen Filtermaske $\mathbf{H}$ aufgebaut. Die Berechnungsvorschrift zur Bildung der Ergebnislaplace-Pyramide lautet dann:

$$l_i^{(a)}(x,y) = g_i^{(m)}(x,y) \cdot l_i^{(e1)}(x,y) + \left(1 - g_i^{(m)}(x,y)\right) \cdot l_i^{(e2)}(x,y). \tag{28.31}$$

28.10.3 Multifokus

Bei den bis jetzt beschriebenen *mosaicing*-Anwendungen wurde anhand eines Maskenbildes entschieden, welche Bildinformation in das Ausgabebild zu übernehmen ist. Bei der Multifokus-Anwendung wird anhand der Laplace-Pyramiden der beiden Eingabebilder entschieden, wie das Ausgabebild aufzubauen ist. Ein Beispiel dazu sind zwei Eingabebilder mit identischem Beobachtungsgebiet, die jedoch mit unterschiedlicher Schärfentiefe aufgezeichnet wurden. Im ersten Bild wurde z.B. auf den Vordergrund fokussiert, so dass der Hintergrund unscharf ist, während im zweiten Bild auf den Hintergrund fokussiert wurde.

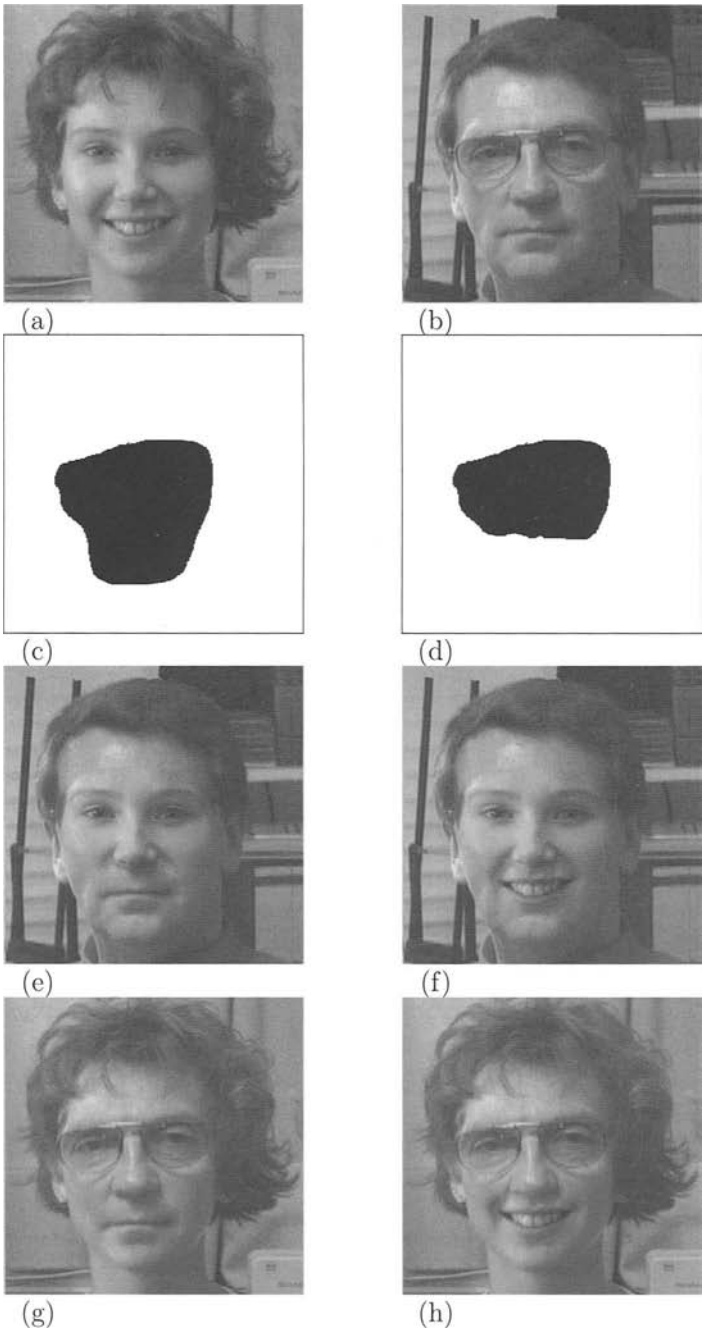


Bild 28.12: *mosaicing* von zwei Eingabebildern (a) und (b). Die Maskenbilder (c) und (d) geben an, welche Bildteile aus dem ersten bzw. zweiten Eingabebild zu übernehmen sind. Unterschiedliche Kombinationen zeigen die Bilder (e) bis (h).

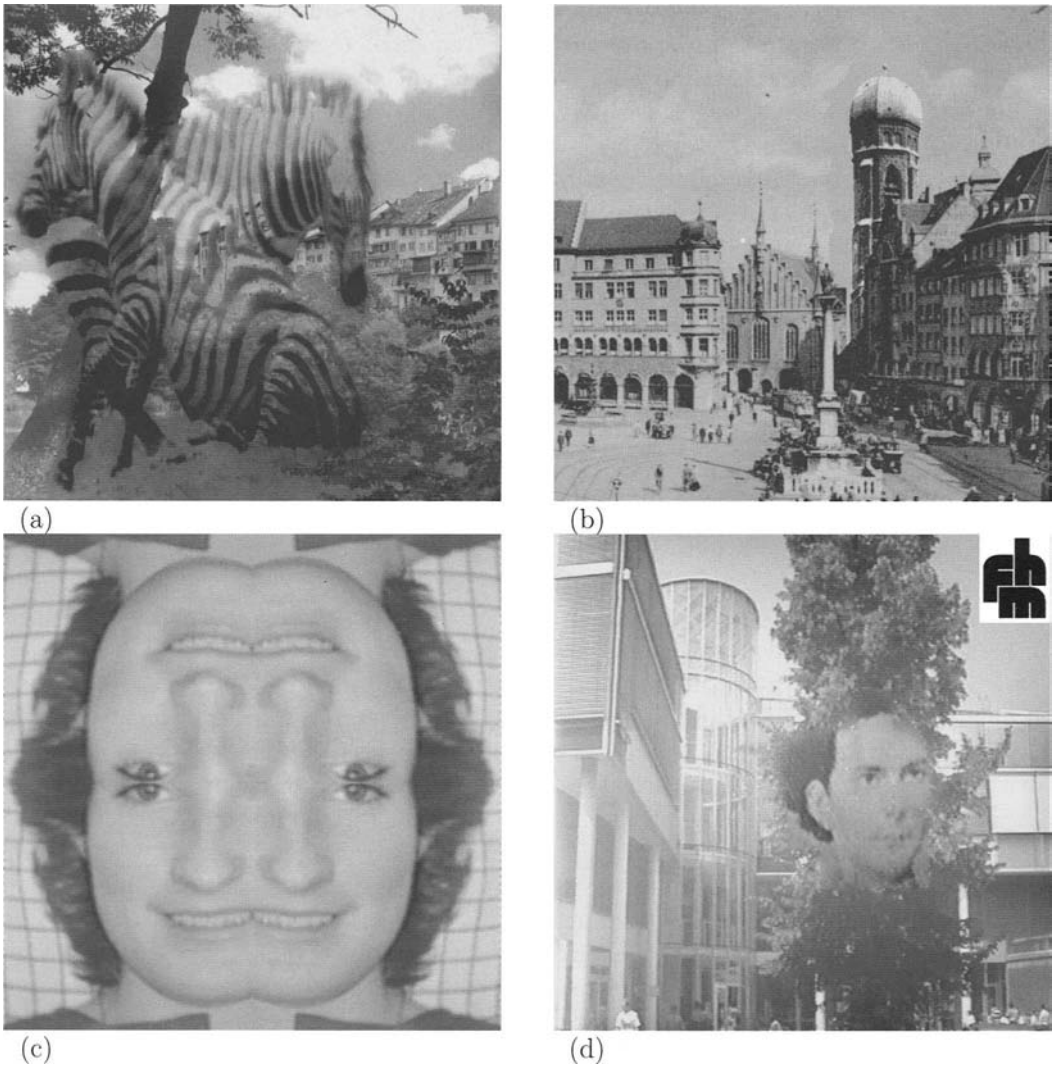


Bild 28.13: Beispiele zum *mosaicing*. (a) Zebras eingeblendet in ein Hintergrundbild. In den hellen Bereichen der Zebras erscheint das Hintergrundbild. (b) Gruß aus München? – Jeder, der München kennt, müsste bei dieser Ansicht skeptisch werden. (c) „Sabine hoch 4". (d) Fachhochschule München. Wer blickt durch den Baum? Als Maske wurde das FHM-Logo verwendet, das rechts oben eingeblendet ist.

Man kann nun ein Mosaikbild erzeugen, bei dem die Schärfentiefe so ausgedehnt ist, dass sowohl Vorder- als auch Hintergrund scharf erscheinen.

Dazu baut man wieder die beiden Eingabepyramiden $\mathbf{L}^{(e1)}$ und $\mathbf{L}^{(e2)}$ auf. Die Schichten der beiden Laplace-Pyramiden, die die niederen Frequenzbereiche abdecken, werden nur geringe Unterschiede aufweisen, da Fokusveränderungen in den niederen Frequenzbereichen nur wenige Auswirkungen zeigen. Dagegen werden Fokusveränderungen die Schichten beeinflussen, die die hohen Ortsfrequenzen enthalten. Korrespondierende Ausschnitte der beiden Eingabebilder werden nahezu dieselben Bereiche des Beobachtungsgebietes enthalten. Sie werden sich aber möglicherweise dadurch unterscheiden, dass sie unterschiedliche Bildschärfe aufweisen und somit unterschiedliche Amplituden in den hochfrequenten Schichten der Laplace-Pyramide besitzen. Für das zu erzeugende Ausgabebild heißt das, dass für die jeweilige Position derjenige Wert verwendet wird, der betragsgrößer ist:

$$l_i^{(a)}(x,y) = \begin{cases} l_i^{(e1)}(x,y), & \text{falls } |l_i^{(e1)}(x,y)| > |l_i^{(e2)}(x,y)|, \\ l_i^{(e2)}(x,y), & \text{falls } |l_i^{(e1)}(x,y)| \leq |l_i^{(e2)}(x,y)|. \end{cases} \tag{28.32}$$

28.10.4 Glättungsoperationen in Laplace-Pyramiden

In der Einleitung zu diesem Kapitel wurde bereits darauf hingewiesen, dass eine Motivation für den Einsatz der Pyramiden die Reduzierung des Rechenaufwandes bei Filteroperationen ist. Soll ein Bild auch im niederfrequenten Bereich geglättet werden, so muss ein großer Filterkern verwendet werden, was einen beachtlichen Rechenaufwand bedeutet.

Da ein Bild als Laplace-Pyramide in verschiedenen Auflösungsschichten vorliegt, haben auch kleine Filterkerne eine globale Auswirkung, je nachdem, in welcher Schicht der Pyramide sie angewendet werden. Bild 28.14 zeigt links oben ein Original. Rechts daneben ist das in der Pyramidenstruktur tiefpassgefilterte Bild zu sehen. Es wurde ein separabler $3 \cdot 3$–Binomialfilterkern (1 2 1) auf alle Schichten der Laplace-Pyramide angewendet. In der Schicht $\mathbf{L}_0$ der Pyramide wird dieser Kern nur geringe Auswirkungen zeigen. Bei Schichten, die näher an der Pyramidenspitze liegen, werden durch diesen Filterkern zunehmend niederfrequente Bildinformationen erfasst. Zur besseren Verdeutlichung wurde dieser Vorgang zweimal wiederholt. Die Ergebnisse sind in den beiden unteren Teilbildern von 28.14 abgebildet. Man sieht deutlich, dass die feinen Bildstrukturen unschärfer werden. Aber auch die niederfrequenten Anteile werden abgeschwächt, was hier am abnehmenden Kontrast zu sehen ist.

28.10.5 Texturen und Segmentierung

Zur Untersuchung von Texturen sind die Laplace-Pyramiden ebenfalls gut geeignet. Da sich eine Textur in verschiedenen Frequenzbereichen ausprägen kann, ist zu erwarten, dass sich diese Eigenschaft auch in den Schichten der Laplace-Pyramide zeigt. Ohne hier im Detail auf die Problematik der Texturanalyse eingehen zu wollen, werden im Folgenden einige Beispiele gebracht, die den Einsatz von Laplace-Pyramiden in der Texturanalyse und bei der Segmentierung von strukturierten Bildbereichen verdeutlichen.

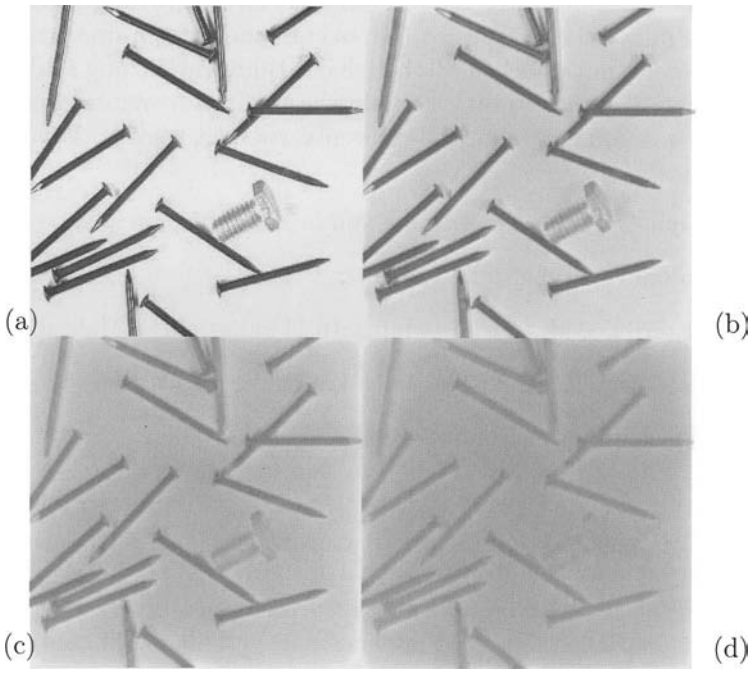


Bild 28.14: Filterung in Laplace-Pyramiden. (a) Original. (b) Das mit einem separablen $3 \cdot 3$–Binomialfilterkern bearbeitete Bild. Der Filterkern wurde auf alle Schichten der Laplace-Pyramide angewendet. In der Schicht $\mathbf{L}_0$ der Pyramide zeigen sich nur geringe Auswirkungen. Bei Schichten, die näher an der Pyramidenspitze liegen, werden durch diesen Filterkern zunehmend niederfrequente Bildinformationen erfasst. (c) Zweimal gefiltertes Bild. (d) Dreimal gefiltertes Bild. Man erkennt, dass feine Bildsturkturen unschärfer werden. Niederfrequente Anteile werden abgeschwächt, was hier am abnehmenden Kontrast zu sehen ist.

Weitere Bemerkungen zur Thematik „Textur" sind in Kapitel 29 zusammengestellt. Dort ergibt sich ein interessanter Querbezug zur *fraktalen Geometrie* und zum *scale space filtering*.

Bei allen Verfahren zur Texturanalyse werden *Texturmerkmale* berechnet. Die sich dabei ergebenden Maßzahlen sollen so beschaffen sein, dass sie für bestimmte Texturen „homogene Bereiche" im Merkmalsraum der Texturparameter ergeben. Bei nur einem Texturmerkmal heißt das, dass die Maßzahl, als Grauwert dargestellt, für die Textur einen homogenen Grauwertbereich ergibt. Die so entstandenen Grauwertbilder aus Texturmaßzahlen können dann mit üblichen Methoden der Bildverarbeitung (z.B. Schwellwertbildung, Klassifizierung, Kantenextraktion, Binarisierung, usw.) weiter verarbeitet und z.B. als Masken zur Extraktion bestimmter Texturen verwendet werden. Beispiele für Texturparameter sind:

- Die mittlere quadratische Abweichung von lokalen Umgebungen,
- der Betrag in die Richtung des Gradienten,
- aus der *co-occurrence*-Matrix (Abschnitt 16.11) abgeleitete Maßzahlen,
- die Kettenanzahl und Kettenlänge eines Lauflängencodes,
- Parameter aus den Fourier-Koeffizienten,
- usw.

Die Eignung der Laplace-Pyramiden zur Texturuntersuchung soll zunächst anhand eines Testbildes (Bild 28.15) untersucht werden.

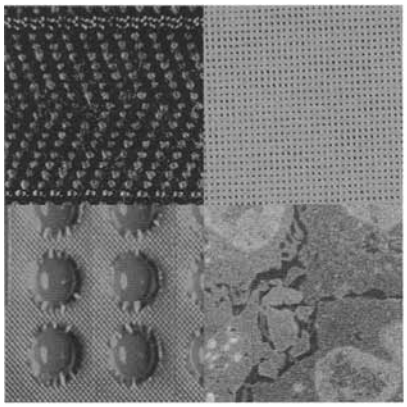
Für die erste Textur, ein Stoffgewebe, liegen die wesentlichen Wellenzahlindizes im Bereich $20 < u < 100$. Dieses breite Frequenzband verteilt sich hauptsächlich über die Schichten $\mathbf{L}_1$, $\mathbf{L}_2$ und $\mathbf{L}_3$. In $\mathbf{L}_0$ sind nur geringe und ab $\mathbf{L}_4$ fast keine Signale mehr zu erkennen.

Die zweite Textur (Gittergewebe) besitzt ebenfalls ein breites Frequenzspektrum: $50 < u < 170$. Diese Frequenzanteile sind im Wesentlichen in $\mathbf{L}_1$ enthalten. Wegen der Überschneidung der Frequenzbereiche sind auch in $\mathbf{L}_0$ und $\mathbf{L}_2$ Schwingungsanteile zu erkennen.

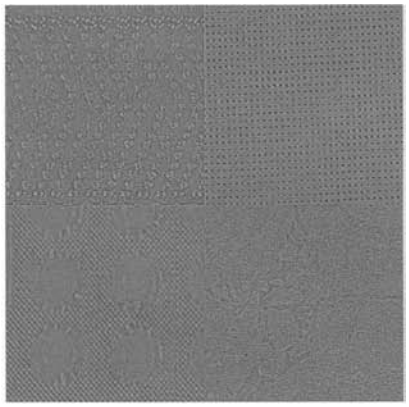
Die markanten Schwingungsanteile der dritten Textur (Blisterpackung) liegen im Bereich von $4 < u < 15$. Das bedeutet, dass sich diese Frequenzanteile hauptsächlich in den Schichten $\mathbf{L}_4$ und $\mathbf{L}_5$ zeigen. Die deutlich ausgeprägten Ränder um die Tablettenmulden sind hochfrequenter und daher in den Schichten $\mathbf{L}_1$ und $\mathbf{L}_2$ zu sehen.

Bei dem mikroskopischen Schnitt durch Melanomzellen (Bild 28.15-a, unten rechts) ist keine regelmäßige Struktur wie bei den anderen Beispielen zu erkennen. Das heißt, dass es auch keine markant hervortretenden Schwingungsanteile gibt und somit auch in keiner Schicht der Laplace-Pyramide diese Oberflächenstruktur signifikant hervortritt.

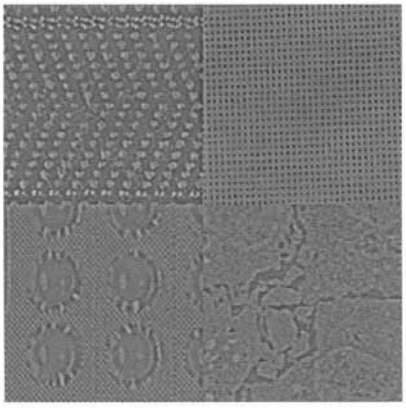
Wenn nun zu einer Laplace-Pyramide Texturmerkmale berechnet werden, so kann dies in jeder Schicht durchgeführt werden. Dabei sind die verschiedensten Vorgehensweisen möglich. Einige Beispiele sind im Folgenden aufgeführt:



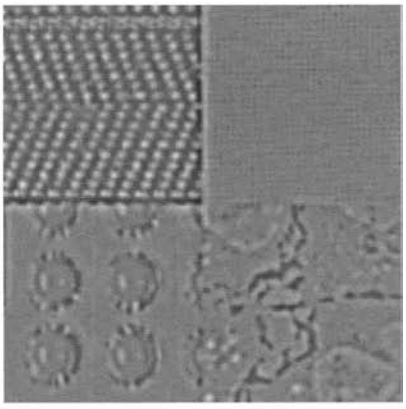
(a) Original



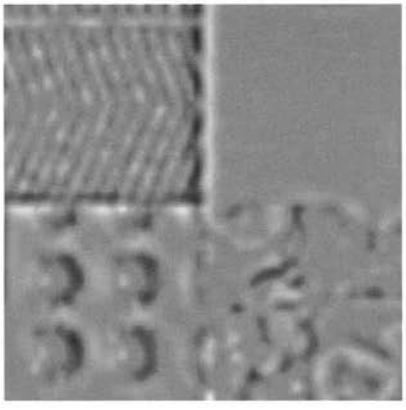
(b) Schicht $\mathbf{L}_0$



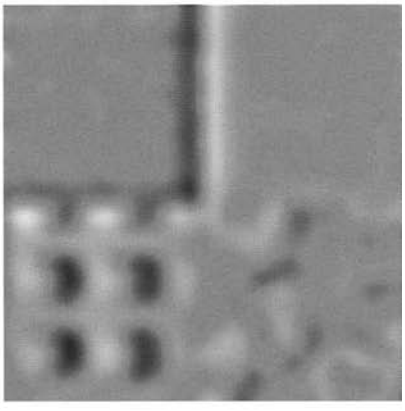
(c) Schicht $\mathbf{L}_{1,1}$



(d) Schicht $\mathbf{L}_{2,2}$



(e) Schicht $\mathbf{L}_{3,3}$



(f) Schicht $\mathbf{L}_{4,4}$

Bild 28.15: Laplace-Pyramiden und Texturanalyse. (a) Originalbild mit vier unterschiedlichen Texturen: Stoffgewebe, Gittergewebe, Blisterpackung, Melanom. (b) - (f) Die Schichten $\mathbf{L}_0$, $\mathbf{L}_{1,1}$ bis $\mathbf{L}_{4,4}$.

- Aufgrund von Voruntersuchungen kann man feststellen, dass die zu interessierende Struktur in einer bestimmten Schicht am deutlichsten hervortritt. Für diese Schicht wird ein Texturmaß berechnet und das daraus resultierende Ergebnis wird weiterverarbeitet.

- Ein Texturmaß zeigt für verschiedene Texturen in mehreren Schichten unterschiedliches Verhalten. Die Texturmaßzahlen der einzelnen Schichten werden zu mehrdimensionalen Merkmalsvektoren zusammengefasst und klassifiziert.

- Es werden verschiedene Texturmaße für die unterschiedlichen Schichten berechnet. Diese werden wie im vorhergehenden Fall als mehrdimensionale Merkmalsvektoren klassifiziert.

In den beiden folgenden Abschnitten werden zwei unterschiedliche Texturansätze in Laplace-Pyramiden vorgestellt und anhand verschiedener Bildbeispiele erläutert.

28.10.5.1 Mittlere quadratische Abweichung

Bei der Verwendung der mittleren quadratischen Abweichung q als lokales Texturmerkmal werden zu begrenzten Umgebungen (z.B. $3 \cdot 3$, $5 \cdot 5$,...) die Quadratsummen der Abweichungen vom Mittelwert aufsummiert. Die Resultate sind proportional zu einem Schätzwert für die Streuung der Umgebung. Oft wird deshalb q abkürzend als „Streuung der Umgebung" bezeichnet. Da q in allen Schichten für dieselbe Umgebung berechnet wird, werden die Streuungen für die verschiedenen Frequenzbereiche abgeschätzt.

Der Formalismus zur Berechnung von q wird im folgenden für eine $3 \cdot 3$–Umgebung wiedergegeben:

$$\begin{aligned} q_i(x,y) &= \frac{1}{9} \sum_{u=-1}^{+1} \sum_{v=-1}^{+1} \Big(l_i(x+u, y+v) - m_i(x,y)\Big)^2; \\ & x = 0, 1, ..., 2^{r-i}; \\ & y = 0, 1, ..., 2^{r-i}; \\ & i = 0, 1, ..., r. \end{aligned} \tag{28.33}$$

In dieser Formel ist $m_i(x,y)$ der Mittelwert der $3 \cdot 3$–Umgebung:

$$m_i(x,y) = \frac{1}{9} \sum_{u=-1}^{+1} \sum_{v=-1}^{+1} l_i(x+u, y+v). \tag{28.34}$$

Über die binomische Formel lässt sich (28.33) umformen:

$$
\begin{aligned}
q_i(x,y) &= \frac{1}{9}\sum_{u=-1}^{+1}\sum_{v=-1}^{+1} l_i^2(x+u,y+v) \quad - \quad m_i^2(x,y); \qquad (28.35)\\
&\quad x = 0,1,...,2^{r-i};\\
&\quad y = 0,1,...,2^{r-i};\\
&\quad i = 0,1,...,r.
\end{aligned}
$$

Ähnlich wie bei der REDUCE- und der EXPAND-Operation kann (28.35) in Zeilen- und Spaltenoperationen separiert werden:

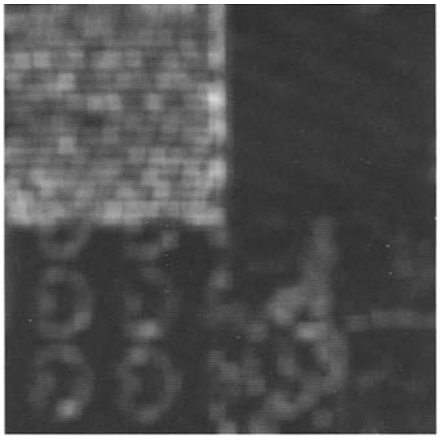
$$
\begin{aligned}
\tilde{q}_i(x,y) &= \sum_{v=-1}^{+1} l_i^2(x,y+v); \qquad (28.36)\\
&\quad x = 0,1,...,2^{r-i};\\
&\quad y = 0,1,...,2^{r-i};\\
&\quad i = 0,1,...,r;\\
q_i(x,y) &= \frac{1}{9}\sum_{u=-1}^{+1} \tilde{q}_i(x+u,y) \quad - \quad m_i^2(x,y);\\
&\quad x = 0,1,...,2^{r-i};\\
&\quad y = 0,1,...,2^{r-i};\\
&\quad i = 0,1,...,r.
\end{aligned}
$$

Für den Mittelwert der Umgebungen wird in den Laplace-Pyramiden null angenommen. Deshalb kann auch auf die Berechnung dieses Terms verzichtet werden, so dass man als Schätzwert für die Streuung der $3 \cdot 3$–Umgebungen erhält:

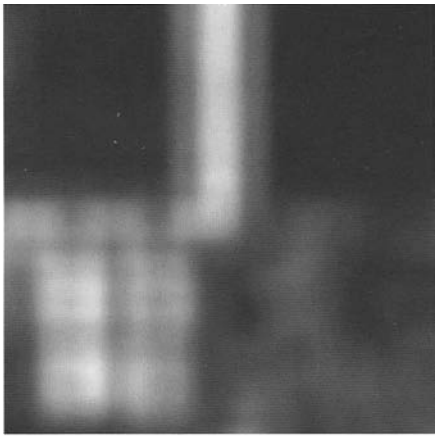
$$
\begin{aligned}
q_i(x,y) &= \frac{1}{9}\sum_{u=-1}^{+1}\sum_{v=-1}^{+1} l_i^2(x+u,y+v); \qquad (28.37)\\
&\quad x = 0,1,...,2^{r-i};\\
&\quad y = 0,1,...,2^{r-i};\\
&\quad i = 0,1,...,r.
\end{aligned}
$$

Auch diese Formel kann wie oben in Zeilen- und Spaltenoperationen separiert werden.

Häufig tritt ferner folgende Problemstellung auf: Oft ist es notwendig, in einem Bild Bereiche mit Objekten vom Bildhintergrund abzugrenzen. Die hier zugrundeliegende Idee ist folgende: Wenn der Bildhintergrund ein charakteristisches Frequenzverhalten zeigt, dann verursachen Objekte vor diesem Hintergrund Störungen im Frequenzverhalten. Diese Störungen müssen sich in den Schichten der Laplace-Pyramide, zu der ein passendes



(a) Schicht $\mathbf{Q}_{2,2}$



(b) Schicht $\mathbf{Q}_{4,4}$

Bild 28.16: Mittlere quadratische Abweichung als Schätzwert für die Streuung der Texturen von Bild 6.15. (a) Die Schicht $\mathbf{Q}_2$ als mittlere quadratische Abweichung der Schicht $\mathbf{L}_2$. (b) Die Schicht $\mathbf{Q}_4$ als mittlere quadratische Abweichung der Schicht $\mathbf{L}_4$. In $\mathbf{Q}_2$ ergibt die erste Textur die größten Werte, die anderen haben Werte, die sehr klein oder sogar null sind. Man könnte diese Schicht zur Extraktion der ersten Textur verwenden. In $\mathbf{Q}_4$ dagegen ist die dritte Textur deutlich markiert. Die breiten Balken in der Mitte sind Störungen im Bereich von Texturkanten.

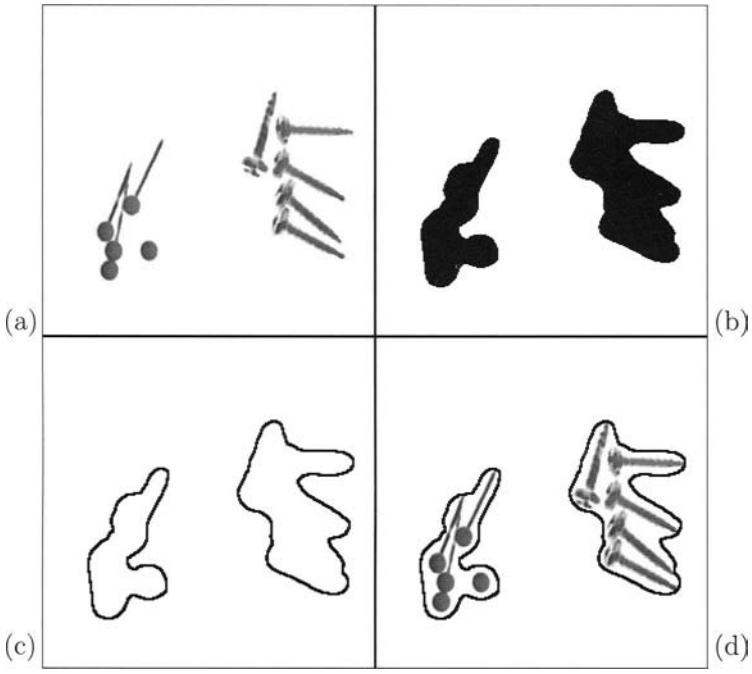


Bild 28.17: Abgrenzung von Bildbereichen mit Objekten vom Hintergrund. Links oben das Original. Durch Segmentierung geeigneter Schichten der Laplace-Pyramide können Masken für die Extraktion der Bildbereiche berechnet werden.

Texturmaß berechnet wurde, ausdrücken. Mit apriori-Wissen kann eine Schicht ermittelt werden, in der sich die Störungen am besten ausprägen. Diese Schicht wird dann zur Segmentierung verwendet.

Bild 28.17 zeigt ein einfaches Beispiel dazu. Das Original links oben enthält zwei Bereiche mit unterschiedlichen Objekten, nämlich Pin-Nadeln und Schrauben. Der Hintergrund ist hier trivialerweise homogen, so dass die Objekte sicherlich deutliche Störungen im Frequenzverhalten des Hintergrundes verursachen werden. Zur Segmentierung der beiden Bereiche wurde hier wie oben das Texturmaß berechnet und die Schicht $\mathbf{L}_2$ auf Originalgröße expandiert. Dieses Bild wurde mit einem Schwellwert (nahe bei 255) segmentiert und invertiert (rechts oben). Die Extraktion des Randes und die Überlagerung mit dem Original zeigen die beiden unteren Teilbilder. Man sieht, dass die beiden Bildbereiche gut erfasst werden. Mit den Informationen der Schichten $\mathbf{L}_0$ und $\mathbf{L}_1$ kann man jetzt versuchen zu unterscheiden, ob es sich um Nadeln oder um Schrauben handelt. Die Berechnung kann dabei auf die segmentierten Bereiche beschränkt werden.

Bild 28.16 zeigt dieses Texturmaß für die Bildbeispiele von Bild 28.15. Das linke Bild enthält die expandierte Schicht $\mathbf{Q}_2$ als mittlere quadratische Abweichung der Schicht $\mathbf{L}_2$ und das rechte Bild die expandierte Schicht $\mathbf{Q}_4$ als mittlere quadratische Abweichung der Schicht $\mathbf{L}_4$. In $\mathbf{Q}_2$ ergibt die erste Textur die größten Werte, die anderen haben Werte, die sehr klein oder sogar null sind. Man könnte diese Schicht zur Extraktion der ersten Textur verwenden. In $\mathbf{Q}_4$ dagegen ist die dritte Textur deutlich markiert. Die breiten Balken in der Mitte sind Störungen im Bereich von Texturkanten.

Schwieriger wird das Problem, wenn der Hintergrund texturiert ist (Bild 28.18). Im Original links oben liegen auf einer Plastikverpackungsfolie eine Schraube und eine Mutter. Zur Segmentierung wurden hier die Texturwerte der Schicht $\mathbf{L}_3$ verwendet (rechts oben). Um die Maske sinnvoll anfertigen zu können, wurde Zusatzinformation herangezogen: Zunächst wurde eine *area-of-interest* definiert, die die Störungen am Bildrand ausblendet. Alternativ dazu können die Störungen am Bildrand auch mit morphologischen Operationen (z.B. *grassfire*, Grundlagen in Kapitel 19) entfernt werden. Sodann wurde die apriori-Information verwendet, dass Störungen eine Fläche von mindestens c Bildpunkten besitzen. Der Bereich der Schraube wurde zufriedenstellend extrahiert. Für die Mutter ist die Maske etwas zu klein geraten. Das liegt u.a. daran, dass die Mutter etwa dieselbe Größe hat wie die Luftblasen der Plastikfolie des Hintergrundes und damit die Störung des charakteristischen Frequenzverhaltens des Hintergrundes nicht so ausgeprägt ist.

28.10.5.2 Spektrale Signalenergie

Aufgrund der Frequenzzerlegung der Laplace-Pyramide können die einzelnen Schichten als spektrale Zerlegung eines Bildes im Ortsbereich aufgefasst werden. Verschiedene Texturen prägen sich in den einzelnen Schichten unterschiedlich aus. Wenn sich nun eine Textur in einer bestimmten Schicht der Laplace-Pyramide weniger stark zeigt, so kann man davon ausgehen, dass die Signalenergie für diese Textur in dieser Schicht geringer ist. Ein Maß für die Signalenergie ist die mittlere quadratische Abweichung, wie sie im letzten Abschnitt verwendet wurde. Allerdings berücksichtigen die Berechnungen dort nicht, dass eine Schicht von verschiedenen Texturen beeinflußt wird.

Um in einzelnen Schichten unterschiedliche Texturen untersuchen zu können, muss man ein Verfahren anwenden, bei dem die Signalenergie in lokalen Bereichen für unterschiedliche Umgebungsgrößen abgeschätzt wird. Dazu wird eine *Energiehierarchie* aufgebaut, die als *hierarchische diskrete Korrelation* (HDC) bezeichnet wird. Begonnen wird mit der untersten Schicht $\mathbf{L}_0$. Hier werden die Signalwerte quadriert, dann wird r-mal die HDC-Operation (genaue Erläuterung der HDC-Operation weiter unten) darauf angewendet. Im nächsten Schritt wird die Schicht $\mathbf{L}_1$ quadriert und dann $(r-1)$-mal die HDC-Operation angewendet. Dies wird bis zur Spitze $\mathbf{L}_r$ fortgesetzt, deren Signalwerte nur quadriert werden. Der Aufbau dieser Energiehierarchie ist im Blockdiagramm in Bild 28.19 dargestellt.

Die HDC-Operation ist der REDUCE-Operation ähnlich. Zur Glättung wird die gleiche Filtermaske $\mathbf{H} = (h(u,v))$ verwendet. Beim Übergang zur nächsten Stufe wird jedoch nicht die Größe um die Hälfte reduziert, sondern die Breite des Ausschnittes vergrößert, in dem die Signalwerte geglättet werden. Durch diese Vorgehensweise kann man Bilder über einen

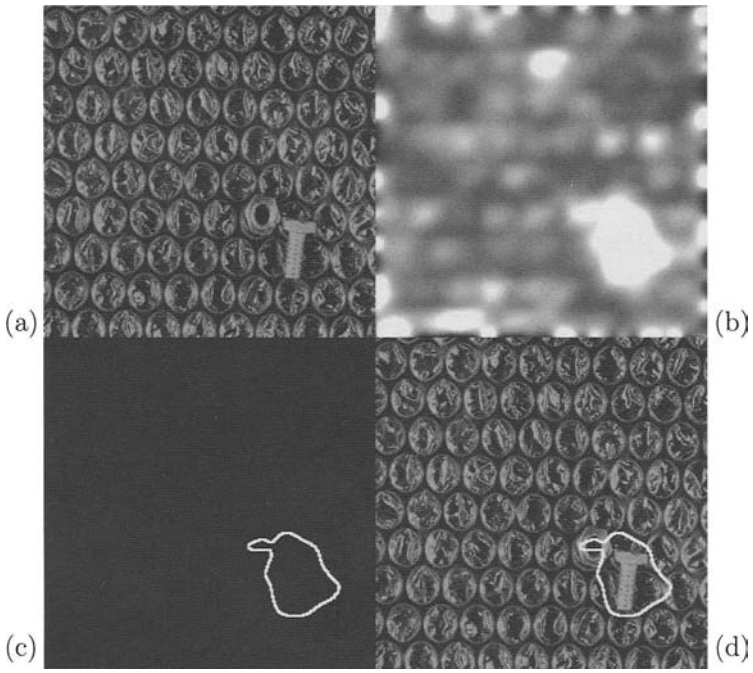


Bild 28.18: Abgrenzung von Bildbereichen mit Objekten vom texturierten Hintergrund. Links oben das Original. Durch Segmentierung geeigneter Schichten der Laplace-Pyramide können auch hier Masken für die Extraktion der Bildbereiche berechnet werden.

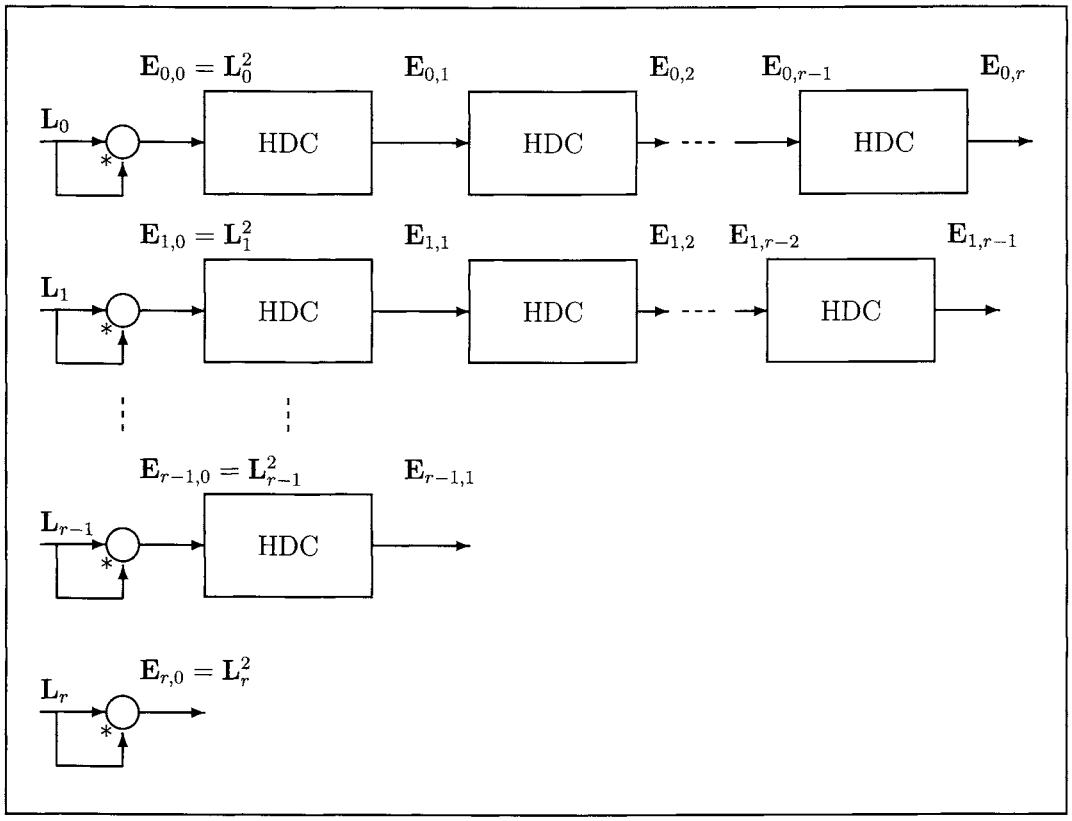


Bild 28.19: Blockdiagramm zum Aufbau der Energiehierarchie.

großen Bereich von Wellenzahlindizes glätten. Die HDC-Operation wird für jede Schicht $\mathbf{L}_i$ der Laplace-Pyramide durchgeführt. Es werden folgende Bezeichnungen verwendet:

$$\begin{aligned} \mathbf{E}_{i,0} &= \mathbf{L}_i^2; \\ \mathbf{E}_{i,k+1} &= \mathrm{HDC}(\mathbf{E}_{i,k}); \\ & i = 0, 1, ..., r; \\ & k = 0, 1, ..., r - (i+1). \end{aligned} \tag{28.38}$$

Die einzelnen Elemente der Energiehierarchiestufen berechnen sich dann gemäß:

$$\begin{aligned} e_{i,k+1}(x,y) &= \sum_{u=-2}^{+2} \sum_{v=-2}^{+2} e_{i,k}(x + u \cdot 2^k, y + v \cdot 2^k) \cdot h(u+2, v+2); \\ & x = 0, 1, ..., r - i; \end{aligned} \tag{28.39}$$

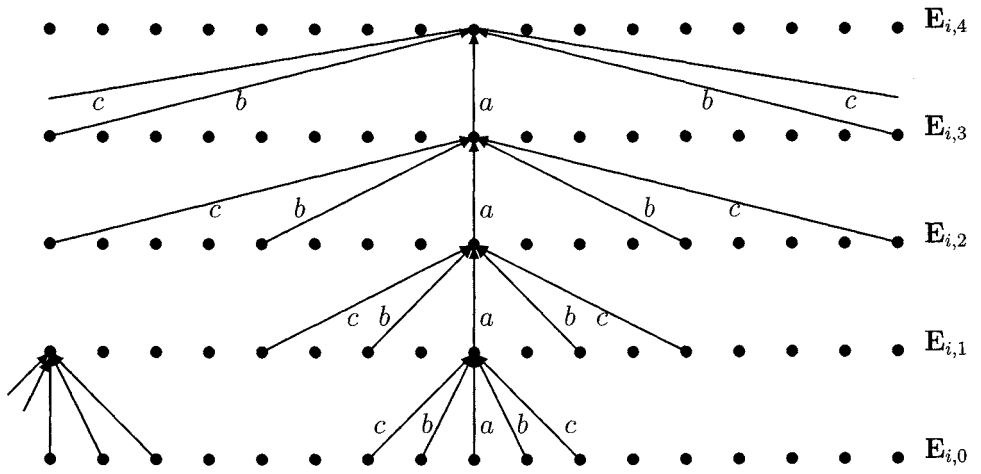


Bild 28.20: Berechnung der Energiehierarchiestufen (HDC-Verfahren). Am Rand liegende Bildpunkte müssen gesondert berechnet werden.

$$y = 0, 1, ..., r - i;$$
$$k = 0, 1, ..., r - (i + 1).$$

Wie bei der REDUCE-Operation kann man auch die HDC-Operation in Zeilen- und Spaltenoperationen auftrennen:

$$\tilde{e}_{i,k+1}(x, y) = \sum_{v=-2}^{+2} e_{i,k}(x, y + v \cdot 2^k) \cdot \hat{h}(v + 2); \tag{28.40}$$
$$x = 0, 1, ..., r - i;$$
$$y = 0, 1, ..., r - i;$$
$$e_{i,k+1}(x, y) = \sum_{u=-2}^{+2} \tilde{e}_{i,k+1}(x + u \cdot 2^k, y) \cdot \hat{h}(u + 2); \tag{28.41}$$
$$k = 0, 1, ..., r - (i + 1).$$

In Bild 28.20 ist dieses Verfahren für den jeweils mittleren Bildpunkt pro Zeile schematisch dargestellt. Man sieht hier auch, dass die Behandlung der Randpunkte etwas anders zu handhaben ist als bei der REDUCE-Operation: Beim Übergang von $\mathbf{E}_{i,0}$ zu $\mathbf{E}_{i,1}$ müssen zwei Bildpunkte am Rand extrapoliert werden, bei $\mathbf{E}_{i,1}$ zu $\mathbf{E}_{i,2}$ sind es vier und in der k-ten Iteration sind es 2^{k+1} Bildpunkte. Dadurch ergibt sich eine Begrenzung der maximalen

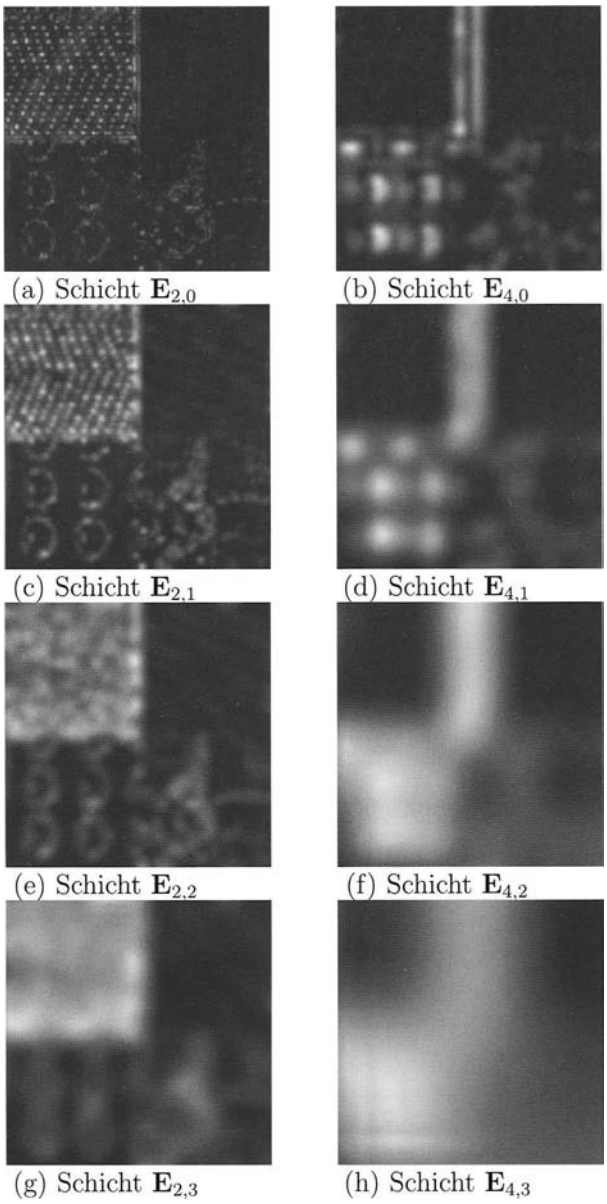
Anzahl von Energiehierarchiestufen, denn es müssen pro Zeilen- und Spaltenoperation von den fünf Bildpunkten, die zur Glättung verwendet werden, mindestens drei innerhalb des Bildes liegen.

Nun einige Anwendungsbeispiele zur Oberflächenuntersuchung mit dem HDC-Verfahren: Zunächst werden die Texturen von Bild 28.15 untersucht. In der Bildfolge 28.21 werden in der linken Spalte die Energiehierarchiestufen $\mathbf{E}_{2,0}$ bis $\mathbf{E}_{2,3}$ und in der rechten Spalte die Energiehierarchiestufen $\mathbf{E}_{4,0}$ bis $\mathbf{E}_{4,3}$ abgebildet. In der Laplace-Pyramide ist für die Textur 1 (Stoffgewebe) in der Schicht $\mathbf{L}_2$ die größte Signaldichte vorhanden. In den darunterliegenden Schichten der Energiehierarchie dehnen sich die signalstärksten Strukturen mehr und mehr aus, so dass die Textur 1 mit $\mathbf{E}_{2,3}$ am besten segmentiert werden kann.

Bei Textur 3 (Blisterpackung) ist die größte Energie in der Schicht $\mathbf{L}_4$ der Laplace-Pyramide vorhanden. In der Energierhierarchie ist deshalb diese Textur am besten mit $\mathbf{E}_{4,2}$ oder $\mathbf{E}_{4,3}$ zu segmentieren.

Ein weiteres Beispiel kommt aus dem Bereich der Vollständigkeitskontrolle. Bild 28.22 zeigt eine vollständig gefüllte Blisterpackung, daneben ist eine Packung mit einer fehlenden Tablette abgebildet. In der Schicht $\mathbf{E}_{4,1}$ ist diese „Texturstörung“ deutlich zu sehen. Nach einer Binarisierung dieser Schicht wurde damit die fehlende Tablette im Original markiert.

Abschließend zu den Gauß- und Laplace-Pyramiden sei noch bemerkt, dass in [Burt83] noch weitere interessante Anwendungen, wie z.B. das Multiresolution Spline, die Korrelation oder grafische Anwendungen, auch in Verbindung mit der fraktalen Geometrie (Kapitel 29) auf der Basis dieser Technik untersucht wurden.



(a) Schicht $\mathbf{E}_{2,0}$ (b) Schicht $\mathbf{E}_{4,0}$

(c) Schicht $\mathbf{E}_{2,1}$ (d) Schicht $\mathbf{E}_{4,1}$

(e) Schicht $\mathbf{E}_{2,2}$ (f) Schicht $\mathbf{E}_{4,2}$

(g) Schicht $\mathbf{E}_{2,3}$ (h) Schicht $\mathbf{E}_{4,3}$

Bild 28.21: Energiehierarchiestufen nach dem HDC-Verfahren. Linke Spalte (Bilder (a), (c), (e), (g)): Stufen $\mathbf{E}_{2,0}$, $\mathbf{E}_{2,1}$, $\mathbf{E}_{2,2}$ und $\mathbf{E}_{2,3}$. In der Schicht $\mathbf{E}_{2,0}$ treten die Bildstrukturen des Stoffgewebes mit der stärksten Signalenergie deutlich hervor. Durch das HDC-Verfahren dehnen sich diese Signale in den weiteren Schicht aus. Die Schicht $\mathbf{E}_{2,3}$ ist damit zur Segmentierung der Textur 1 geeignet. In der rechten Spalte ist derselbe Sachverhalt für die Textur 3 (Blisterpackung) dargestellt.

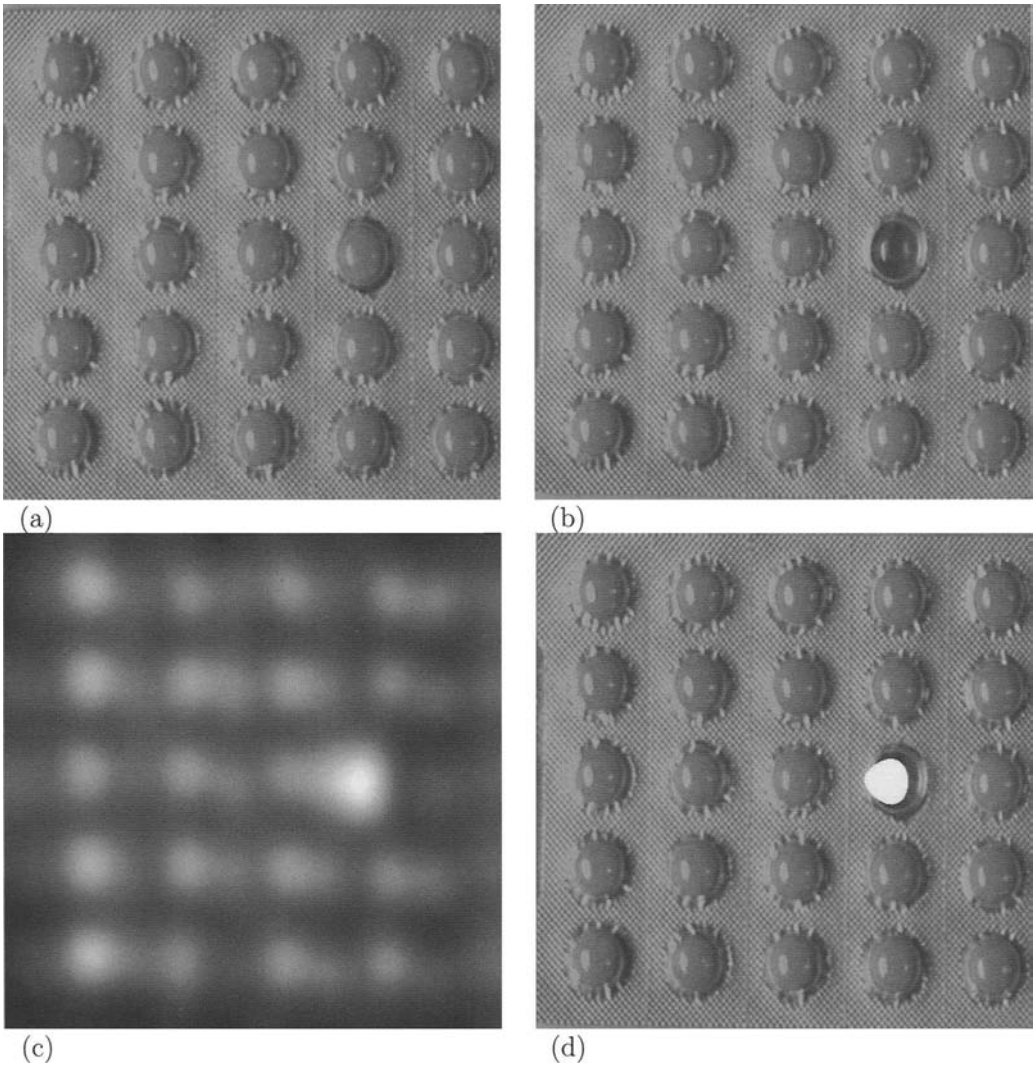


Bild 28.22: Vollständigkeitskontrolle mit dem HDC-Verfahren. (a) Vollständige Blisterpackung. (b) Blisterpackung mit einer fehlenden Tablette. (c) Die Schicht $\mathbf{E}_{4,1}$ der Energiehierarchie. (d) Die binarisierte Schicht $\mathbf{E}_{4,1}$ wurde in das Original eingeblendet und dadurch die fehlende Tablette markiert.

Kapitel 29

Scale Space Filtering

29.1 Anwendungen

Vor dem Hintergrund der fraktalen Geometrie wird in diesem Kapitel das *scale space filtering* erläutert. Dabei wird eine Oberfläche in unterschiedlichen „Vergrößerungen" betrachtet und untersucht, ob dabei sich wiederholende Strukturen auftreten. Als praktische Anwendung hat diese Technik gute Ergebnisse bei der Qualitätskontrolle von Oberflächen, z.B. von Stoffen, geliefert.

29.2 Grundlagen: Fraktale Geometrie

Dieser Abschnitt enthält eine kurze Übersicht über die fraktale Geometrie. Es werden wichtige Begriffe, wie z.B. die gebrochene (fraktale) Dimension vorgestellt und anhand von Beispielen erläutert. Zur Vertiefung dieser Thematik sei auf die umfangreiche Literatur verwiesen (z.B. [Ekel92], [Mand87]).

Erste Ansätze in Richtung fraktaler Geometrie wurden bereits im letzten Jahrhundert beschrieben, wenn auch aus einem ganz anderen Blickwinkel: Der Mathematiker Karl Weierstrass stellte 1872 eine Funktion vor, die überall stetig, aber nirgends differenzierbar ist. Der Zusammenhang zwischen Stetigkeit und Differenzierbarkeit fällt in den Bereich der klassischen Infinitesimalrechnung. Lange Zeit ging man davon aus, dass stetige Funktionen bis auf „einige" Ausnahmen auch differenzierbar sind. Weierstrass widerlegte diese Ansicht mit folgender Funktion: Zunächst konstruierte er eine Funktion, deren Graph wie ein Sägeblatt aussieht. Es ist offensichtlich, dass diese Funktion an den Spitzen der Sägezähne zwar stetig, aber nicht differenzierbar ist. An allen anderen Stellen ist sie differenzierbar. Nun definierte er eine neue Funktion, die durch den Grenzübergang entsteht, wenn man die Sägezähne unendlich nahe nebeneinander liegen lässt. Die so definierte Funktion ist dann zwar überall stetig aber an keiner Stelle mehr differenzierbar.

Sehr zögernd akzeptierten die Mathematiker die Existenz derartiger Funktionen. Sie wurden allgemein zunächst nicht sehr ernst genommen, da „Monster-Funktionen" dieser Art sicher nichts mit dem mathematischen Alltag zu tun haben.

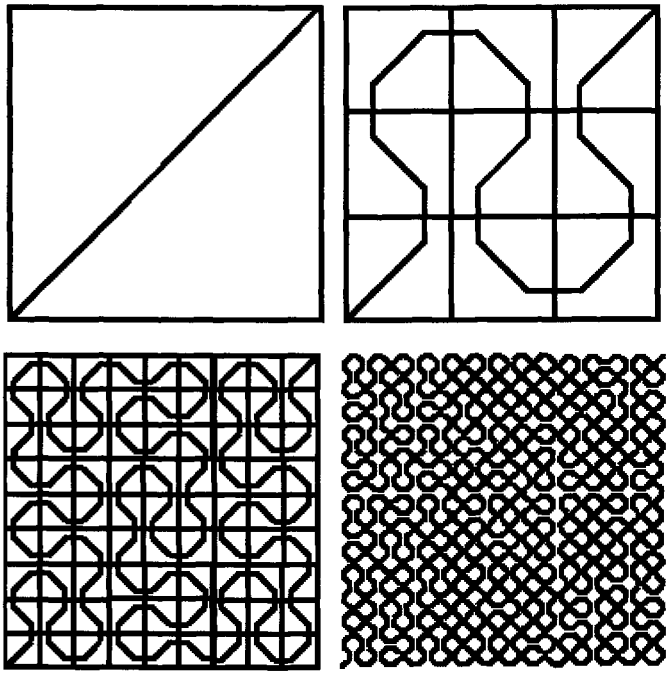


Bild 29.1: Die Peano-Kurve als Beispiel einer „raumfüllenden" Kurve. Initiator ist die Diagonale eines Quadrats. Der Generator ist daneben abgebildet. Setzt man die Generierungsschritte bis ins Unendliche fort, so werden alle Punkte des umgebenden Quadrats erfasst.

Für weitere Diskussion sorgte 1890 Giuseppe Peano, als er eine „raumfüllende" Kurve beschrieb (Bild 29.1). Um die Entstehung derartiger Kurven zu beschreiben, geht man von einem *Initiator* aus. Im Fall der Peano-Kurve ist der Initiator die Diagonale eines Quadrats. Mit dem *Generator* wird beschrieben, wie die Seiten iterativ verändert werden. Der Generator der Peano-Kurve ist in Bild 29.1 neben dem Initiator dargestellt. Setzt man die Generierungsschritte bis ins Unendliche fort, so werden alle Punkte des (zweidimensionalen) Quadrats erfasst. Damit kam der Begriff der Dimension ins Wanken: Wie kann eine eindimensionale Kurve alle Punkte einer zweidimensionalen Ebene erfassen?

An dieser Stelle sei folgende Betrachtung aus [Mand87] eingefügt. Der Begriff der Dimension ist im (mathematischen) Alltag einleuchtend: Punktmengen haben die Dimension null, Kurven die Dimension eins, Flächen die Dimension zwei, Körper die Dimension drei, usw. Ab der Dimension vier hat der Mensch zwar Schwierigkeiten bei der Vorstellung dieser Gebilde, aber der Mathematik macht es keine Probleme, mit n-dimensionalen Räumen zu arbeiten.

Man stelle sich nun einen Wollknäuel vor: Sieht man ihn aus weiter Entfernung, so ist

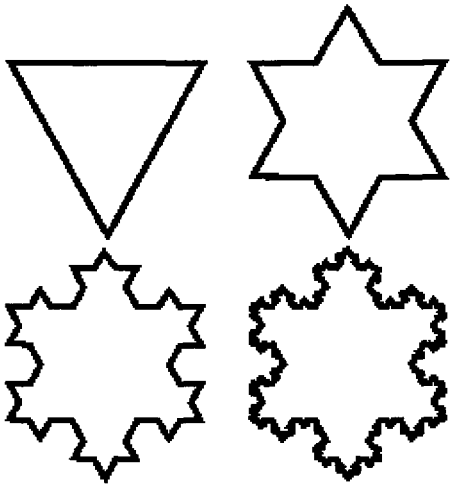


Bild 29.2: Die Koch'sche Insel (Schneeflockenkurve). Hier kann man sich fragen: Wie lang ist der Rand dieser Figur?

lediglich ein Punkt der Dimension null zu erkennen. Etwas näher betrachtet erscheint der Wollknäuel als Scheibe mit der Dimension zwei. Bei noch näherer Betrachtung erkennt man, dass es sich um ein dreidimensionales Gebilde handelt. Aber ist der Wollknäuel wirklich dreidimensional? Der aufgewickelte Faden ist ja eine Kurve, die eindimensional ist. Oder ist der Faden ein Zylinder, der dreidimensional ist? Bei mikroskopischer Betrachtung zerfällt der Faden in nahzu unendlich viele Härchen: Also doch eindimensional?

Der übliche Begriff der Dimension passt scheinbar nicht auf natürliche Problemstellungen. Man könnte entgegnen, dass die Mathematik als abstrahierte Wissenschaft nicht dazu gedacht ist, Problemstellungen wie einen Wollknäuel zu beschreiben. Jedoch ist letztlich die Mathematik ein Hilfsmittel zur Beschreibung unserer realen Umwelt, daher muss sie auch in der Lage sein, reale Sachverhalte so gut wie möglich zu beschreiben.

Weitere Gedankenexperimente wie bei der Peano-Kurve erfolgten: Helge von Koch stellte 1904 seine „Schneeflockenkurve" (Koch'sche Kurve) vor (Bild 29.2). 1915 konstruierte Sierpinski seine „Pfeilspitze" (Sierpinski-Dreieck , Bild 29.3).

Die Peano-Kurve, die Koch'sche Kurve und das Sierpinski-Dreieck sind Beispiele für *selbstähnliche Strukturen*, d.h. dass bei richtiger Skalierung jeder noch so kleine, geeignet gewählte Teil wieder das Ganze bilden kann.

Mit diesen Kurven wird ein weiterer Begriff der Geometrie in Frage gestellt, nämlich die Länge des Randes einer Figur. Wenn man das Bildungsgesetz z.B. der Koch'schen Kurve vor Augen hat, müsste sich doch ein unendlich langer Rand ergeben, da bei jedem Durchgang ein Teilstück um den Faktor 4/3 länger wird.

Benoit Mandelbrot griff um 1975 Ideen von Hausdorff und Julia auf, die zum Teil schon

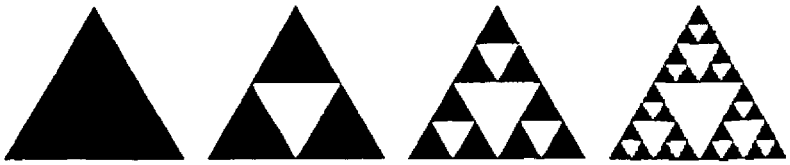


Bild 29.3: Das Sierpinski-Dreieck.

um 1900 publiziert waren. Er prägte den Begriff der *fraktalen Geometrie*. Ein wesentliches Merkmal der fraktalen Geometrie ist die *Selbstähnlichkeit*. Damit ist gemeint, dass sich die prinzipielle Struktur eines Fraktals immer wieder wiederholt, je „näher" man sie betrachtet, oder etwas mathematischer ausgedrückt, je kleiner der Maßstab ist, den man für die Betrachtung wählt. Dieser Sachverhalt wird auch als *Skaleninvarianz* bezeichnet. Schöne Beispiele für einfache, selbstähnliche Fraktale sind die bereits oben dargestellten Kurven.

Mandelbrot erkannte die Selbstähnlichkeit als ein „mächtiges Mittel zum Hervorbringen von Gestalten" und gab viele Beispiele, die mehr oder weniger skaleninvariant sind: Aktienkurse, Küstenlinien, Pflanzenblätter, Wolken, bestimmte komplexe Zahlen, usw.

Bekannt ist auch das Beispiel der Längenvermessung der Küstenlinien von Großbritannien, das 1961 zuerst von L.F. Richardson vorgestellt wurde. Wenn man die Länge der Küstenlinie vermessen will, so könnte man z.B. einen Meterstab als Skaleneinheit ($s = 1m$) nehmen und die Messung durchführen. Für die Länge wird sich dann ein bestimmter Wert ergeben. Möglicherweise wird sich bei der Durchführung der Messung die Frage erheben, ob denn die zugrunde gelegte Skala $s = 1m$ genügend genau ist, und man könnte die Messung mit einer Skala $s = 0.1m$ wiederholen. Es wird sich ein größerer Wert für die Länge ergeben, da man jetzt feinere Details der Küstenlinie erfasst hat. In einem nächsten Schritt könnte man $s = 0.01m$ verwenden. Die Länge der Kurve sollte sich als Grenzwert $L(s)$ der Längen der Polygonzüge ergeben. Kurven, deren Länge auf diese Weise ermittelt werden kann, heißen *rektifizierbar*. [1] Wenn man von einer Struktur sagt, sie zeige *Skalenverhalten*, so meint man damit, dass sie bei allen Skalen denselben Grad an Irregularität aufweist. Eine derartige Struktur bezeichnet man auch als *skaleninvariante Struktur*. Der Parameter s heißt *Skalenparameter*.

[1] Peitgen wollte in seinem Buch [Peit86] über fraktale Bilder ein ähnliches, „eingedeutschtes" Beispiel bringen: Wie lang ist die Grenze der Bundesrepublik Deutschland? Damit kam er aber in Schwierigkeiten: Die Grenze ist nach gesetzlicher Definition ein Polygonzug und damit endlich!

Macht man bei der Küstenlinie den Skalenwert s immer kleiner, so wird man immer mehr Details erfassen: Felsen, Steine, Staub, Moleküle... Diesem Gedankenexperiment folgend erkennt man, dass die wahre Länge der Küste Großbritanniens, wie auch die der Koch'schen Kurve, unendlich ist.

Aus den Experimenten, die Richardson durchführte, leitete er folgende Gesetzmäßigkeit ab:

$$L(s) \sim s^{1-D}. \tag{29.1}$$

Wenn man in ein Koordinatensystem auf der Abszisse $\log s$ und auf der Ordinate $\log L(s)$ aufträgt, so erhält man für die Küste Großbritanniens näherungsweise eine Gerade (Bild 29.4). Mandelbrot erkannte, dass die Steigung dieser Geraden eine Schätzung für $1 - D$ ist und dass D als eine *fraktale Dimension* betrachtet werden kann.

Fraktale Strukturen besitzen also eine fraktale Dimension. Nach Mandelbrot ist die fraktale Dimension die Hausdorff-Besicovitch-Dimension [Haus19]. Sie ist in der Praxis schwer zu bestimmen. Aus diesem Grund verwendet man oft einfachere Definitionen für fraktale Dimensionen, die jedoch, bei richtiger Wahl der Skalenparameter, der Hausdorff-Besicovitch-Dimension nahe kommen.

Dazu einige Beispiele: Teilt man eine Strecke in drei gleiche Teile, so ist die ganze Strecke dreimal so lang wie jede Teilstrecke. Die Anzahl a der Teilstrukturen ist also $a = 3^1 = 3$. Bei einem Quadrat, bei dem jede Seite dreigeteilt wird, erhält man $a = 3^2 = 9$ Teilstrukturen (Teilquadrate) und beim Würfel $a = 3^3 = 27$ Teilstrukturen (Teilwürfel). Man erkennt ein Potenzgesetz:

$$a = s^D, \tag{29.2}$$

wobei a die Anzahl der sich ergebenden Teilstrukturen, s die Anzahl der Unterteilungen der Strecken und D die Dimension der betrachteten Struktur ist. Löst man (29.2) nach D auf, so erhält man:

$$D = \frac{\log a}{\log s}. \tag{29.3}$$

Diese Betrachtungsweise kann man jetzt z.B. auf die Koch'sche Kurve anwenden: Der Generator besteht aus vier Teilstücken. Der Initiator ist dreimal so lang wie ein Teilstück. Somit ergibt sich: $D = \frac{\log 4}{\log 3} = 1.262$. Durch sinngemäße Betrachtungen erhält man für die Peano-Kurve $D = 2$ und für das Sierpinski-Dreieck $D = 1.585$.

Die so berechnete Größe D ist aber nicht die fraktale Dimension, da sie ja nur für streng selbstähnliche Strukturen - bei richtiger Wahl der Skalierung - zutrifft. In [Mand87] wird sie als *Ähnlichkeitsdimension* bezeichnet.

Die Beziehung (29.3) ist aber eine Motivation für die ziemlich willkürlich anmutende Definition der fraktalen Dimension, die im Folgenden anhand eines Beispiels erläutert wird. Dabei wird dargestellt, wie die fraktale Dimension einer Struktur definiert ist und wie sie ermittelt werden kann. Als Beispiel dient die Koch'sche-Kurve, die man sich auf

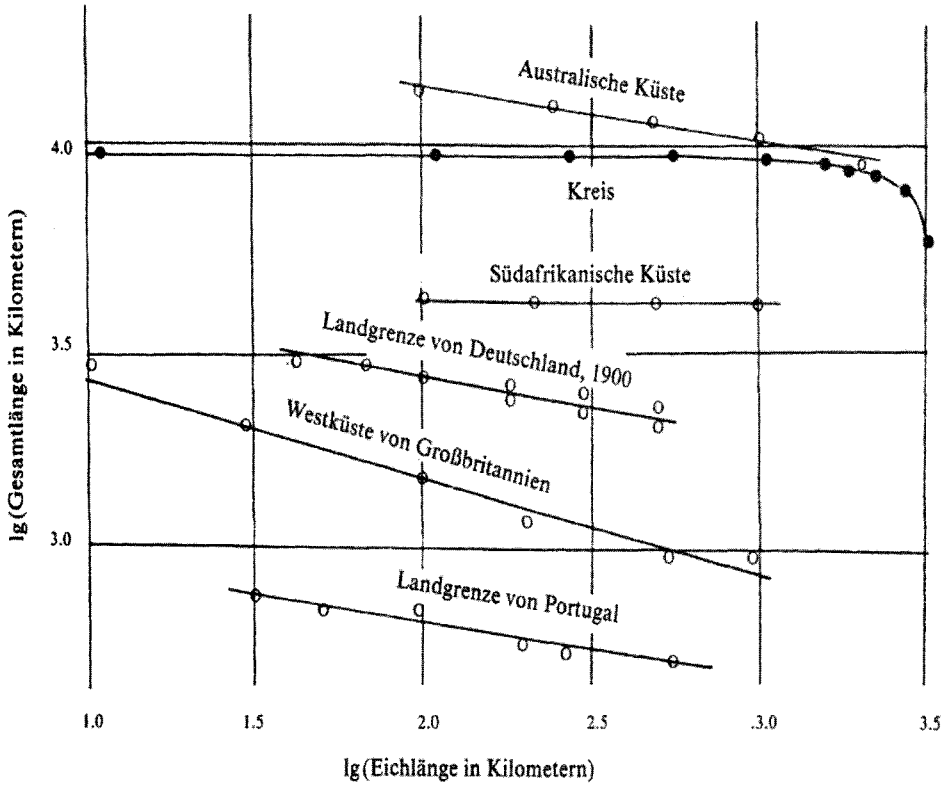


Bild 29.4: Zuwachsrate der Längen verschiedener Grenzlinien. Beim Auftragen von $\log L(s)$ über $\log s$ ergeben sich bei fraktalen Kurven Geraden. Die Untersuchung des Kreises ergibt eine Kurve, die sich schnell bei einem bestimmten Wert stabilisiert (aus [Mand87], siehe auch Fußnote auf der vorherigen Seite).

einem quadratisch gerasterten Papier gezeichnet vorstellen möge. Man zählt nun, wieviele Rasterquadrate von der Kurve geschnitten werden. Dann wird das Raster verfeinert, etwa durch Halbierung der Rasterquadratseiten, und es wird wieder gezählt. Dieser Vorgang wird einige Male bis zu einer noch sinnvollen Rasterflächengröße wiederholt. Es lässt sich zeigen, dass man als Beziehung ein Potenzgesetz ansetzen kann:

$$N = \frac{C}{\epsilon^D}, \tag{29.4}$$

wobei N die Überdeckungszahl, ϵ die Maschenweite der Rasterflächen, C eine Konstante und D die fraktale Dimension ist. Durch Logarithmieren erhält man:

$$\log N = D \log \frac{1}{\epsilon} + \log C. \tag{29.5}$$

Trägt man die Logarithmen von N und $\frac{1}{\epsilon}$ in ein Koordinatensystem ein, so erhält man eine Gerade, deren Steigung die fraktale Dimension D ist.

In der Praxis wird man nicht exakt eine Gerade erhalten, da sich z.B. beim Auszählen der Rasterquadrate Fehler einschleichen. Man wird daher einige „Messungen" mit verschiedenen Skalen durchführen, die Punkte $(\log N, \log \frac{1}{\epsilon})$ in das Koordinatensystem eintragen und mit Hilfe der linearen Regression die Steigung der Regressionsgeraden als fraktale Dimension ermitteln.

Diese Vorgehensweise kann auch bei natürlichen Strukturen, wie z.B. Wolken, Küstenlinien, Bäumen oder Flussystemen, sinngemäß angewendet werden. Als Beispiel wird angegeben, dass der Raum der Arterien des Menschen die fraktale Dimension $D = 2.7$ besitzt.

In [Mand87] werden ausführlich die verschiedensten Fraktalarten diskutiert und in ansprechenden Grafiken dargestellt. Außerdem werden die Bildungsgesetze mit Zufälligkeiten kombiniert, so dass natürlich wirkende Strukturen entstehen. Dies ist auch eine der Motivationen für die Untersuchung fraktaler Gebilde: Man möchte dadurch die in unserer natürlichen Umgebung auftretenden Fraktale besser verstehen und sie möglicherweise, zumindest innerhalb gewisser Schranken, durch bekannte, künstliche Fraktale annähern.

Damit schließt die Darstellung der Grundlagen der fraktalen Geometrie. Die fraktale Betrachtungsweise hat bereits verschiedene Anwendungen in der digitalen Bildverarbeitung und Mustererkennung gefunden. Ein interessanter Aspekt ist die Verwendung im Zusammenhang mit der Bilddatenkompression. Zu dieser Thematik, die hier nicht näher behandelt wird, sei auf [Fish92] verwiesen. Im folgenden Abschnitt werden Anwendungen der fraktalen Betrachtungsweise zur Texturerkennung untersucht.

29.3 Implementierung des Scale Space Filtering

Die in der Natur auftretenden Formen lassen sich oft, in gewissen Grenzen, der Klasse der fraktalen Strukturen zuordnen. Da diese Objekte auch in der digitalen Bildverarbeitung und Mustererkennung untersucht werden, liegt es nahe, die Erkenntnisse aus der fraktalen Geometrie auch in diesen Gebieten anzuwenden. Ursprünglich wurde die Idee der

Längenvermessung von Kurven, wie sie in Abschnitt 29.2 dargestellt wurde, sinngemäß auf digitalisierte Bilder übertragen. Das Problem ist dabei jedoch, dass natürliche Kurven nur in einem eingeschränkten Skalenbereich Skalenverhalten zeigen und die Ermittlung dieses Skalenbereichs wichtig und nicht trivial ist.

Von Witkin [Witk83] wird das *scale space filtering* vorgeschlagen, bei dem eine gegebene Struktur durch sukzessive Faltungen geglättet wird. Der Faltungskern ist von einem stetig veränderbaren Skalierungsparameter abhängig, der bei den verschiedenen Faltungen variiert wird. Nach der Glättung werden die Merkmale der Folge der geglätteten Strukturen in Abhängigkeit des Skalierungsparameters untersucht. Witkin hat in der zitierten Arbeit gezeigt, dass als Glättungsfunktion, auf Grund der notwendigen Voraussetzungen, nur die Gauß-Funktion in Frage kommt. In (29.6) ist die Faltung eines Signals $s(x)$ mit der Gauß-Funktion $g(x, \sigma)$ dargestellt. Das Ergebnis ist eine geglättete Kurve $c(x, \sigma)$:

$$c(x,\sigma) = \int_{-\infty}^{+\infty} s(u) \frac{1}{\sqrt{2\pi}\sigma} e^{\frac{-(x-u)^2}{2\sigma^2}} du. \tag{29.6}$$

Die in Abschnitt 29.2 dargestellte Vorgehensweise von Richardson zur Ermittlung der Länge von Grenzlinien lässt sich auf das *scale space filtering* übertragen: Die Länge einer Kurve wird in Abhängigkeit des Skalierungsparameters σ der Gauß'schen Glättungskurve untersucht. Die experimentellen Ergebnisse lassen auch hier ein Potenzgesetz vermuten

$$L(\sigma) \sim \sigma^p, \tag{29.7}$$

wobei p ein „charakteristischer Skalenexponent" ist, der linear mit einer fraktalen Dimension zusammenhängt.

Dazu ein Beispiel (Bild 29.5): Die Koch'sche Kurve wird mit unterschiedlichen σ-Werten geglättet. Die Logarithmen der Längen der geglätteten Kurven werden über die Logarithmen der unterschiedlichen Skalierungsparameter σ aufgetragen. In dieser Darstellung ergibt sich eine Gerade, deren Steigung der Parameter p ist.

Diese Vorgehensweise wird zur Erkennung von unterschiedlichen Texturen übertragen. Die grundlegende Idee ist, Umgebungen von Bildpunken (Texturfenster) mit zweidimensionalen Gauß-Funktionen mit unterschiedlichen σ-Werten zu glätten. Hier wird aber nicht wie oben die Länge, sondern die Oberfläche der Grauwertfunktion im Texturfenster verwendet. Es zeigt sich, dass auch hier wieder ein Potenzgesetz mit einem charakteristischen Skalenparameter gilt. Dieser Skalenparameter wird als Merkmalswert für die Textur dem zentralen Bildpunkt des Texturfensters zugewiesen. Diese Berechnungen werden für alle Bildpunkte durchgeführt. Es zeigt sich, dass gleiche Texturen auch gleiche Skalenparameter besitzen. Das so erzeugte Merkmalsbild kann in einem letzten Schritt in Bereiche gleicher Textur segmentiert werden.

Der Algorithmus dazu ist im Folgenden dargestellt, die einzelnen Teilprobleme werden in weiteren Abschnitten untersucht.

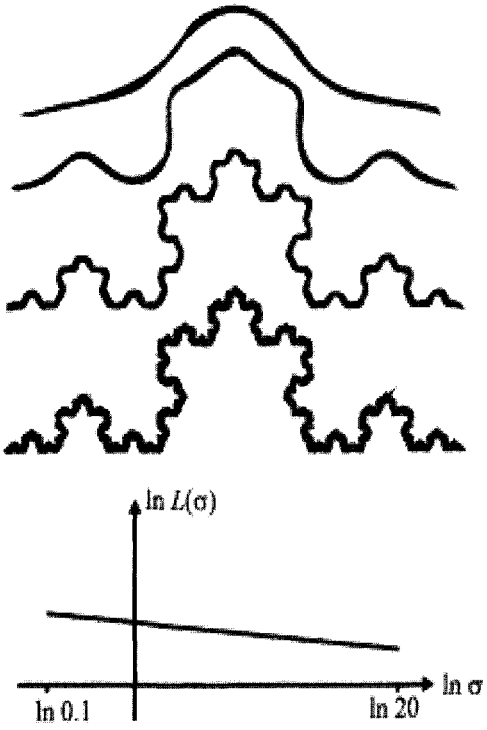


Bild 29.5: Analyse der Länge L einer Koch'schen Kurve in Abhängigkeit der Streuung σ einer Gauß'schen Glättungskurve.

A29.1: Scale Space Filtering.

Voraussetzungen und Bemerkungen:

◇ Es wird vorausgesetzt, dass das zu bearbeitende Bild als Grauwertbild $\mathbf{S}_e = (s_e(x,y))$ vorliegt. Weitere Vorverarbeitungsschritte sind nicht notwendig.

Algorithmus:

(a) Ermittlung der Größe der Grundtextur. Damit ist die Größe der Umgebung gemeint, die notwendig ist, um die Textureigenschaften von speziellen Texturen zu erfassen (Grundtexturfläche).

(b) Ermittlung der Gauß-Filterkerne für die Glättung.

(c) Berechnung der Oberflächen $O(\sigma)$ der Grauwertfunktion. Es gilt ein Potenzgesetz: $O(\sigma) \sim \sigma^p$.

Der Parameter p ist ein Skalenparameter, der für die jeweilige Textur charakteristisch ist.

(d) Berechnung der Skalenparameter p durch lineare Regression.

(e) Segmentierung.

Ende des Algorithmus

29.3.1 Ermittlung der Größe der Grundtextur

Unter der *Grundtexturfläche* versteht man die Umgebung eines Bildpunktes (x,y), die gerade hinreichend groß ist, um die charakteristischen Eigenschaften einer bestimmten Textur zu enthalten (Abschnitt 27.2). Diese Umgebung ist meistens nicht bekannt und wird auch keine regelmäßige Figur sein.

In der Praxis werden dazu rechteckige oder quadratische Umgebungen (Texturfenster) des Punktes (x,y) verwendet. Diese Umgebungen sollten jedoch die Grundtexturfläche so gut wie möglich annähern. Bei einer zu kleinen Umgebung werden nicht alle Textureigenschaften erfaßt und eine zu große Umgebung führt zu erhöhten Rechenzeiten.

Wenn der zu untersuchende Bildausschnitt verschiedene Texturen enthält, müsste man eigentlich mit unterschiedlich großen Umgebungen arbeiten, jeweils angepasst an die gerade untersuchte Textur. Dies ist in der Praxis aber nicht möglich. Wenn die Grundtexturflächen der verschiedenen Texturen sehr unterschiedlich groß sind, wird man in den Übergangsbereichen der verschiedenen Texturen breite Bereiche mit Mischtexturen erhalten, die bei der weiteren Verarbeitung, z.B. bei einer Segmentierung, störend sein können. Man könnte diese Bereiche nach der Segmentierung durch Dilatationen oder Erosionen (Kapitel 19) den benachbarten Klassen zuordnen.

In speziellen Anwendungsfällen wird man die sinnvoll zu verwendende Umgebung zur Erfassung der Grundtexturfläche am besten durch „Augenschein“ und durch Probieren ermitteln.

29.3.2 Ermittlung der Gauß-Filterkerne

Die Glättung des Eingabebildes wird im Ortsbereich mit Filterkernen der Größe $m \cdot m$ durchgeführt, die aus einer diskretisierten, zweidimensionalen Gauß-Funktion mit unterschiedlichen σ-Werten abgeleitet sind. Die Dichte einer zweidimensionalen Normalverteilung mit den Erwartungswerten $\xi_x = \xi_y = 0$ und den Standardabweichungen $\sigma_x = \sigma_y = \sigma$ ist gegeben durch:

$$g(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}. \tag{29.8}$$

Diese Funktion muss in einem $m \cdot m$–Bereich um den Nullpunkt mit insgesamt m^2 Werten diskretisiert werden. Die einfachste Möglichkeit ist, an m^2 diskreten Stützstellen die Funktionswerte von $g(x, y, \sigma)$ zu ermitteln. Beispiele für Filterkerne der Größe $7 \cdot 7$ und $\sigma = 0.5$, $\sigma = 1.0$ und $\sigma = 2.0$ sind:

$$\sigma = 0.5: \quad \begin{pmatrix} 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.011 & 0.084 & 0.011 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.084 & 0.619 & 0.084 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.011 & 0.084 & 0.011 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 & 0.000 \end{pmatrix}, \tag{29.9}$$

$$\sigma = 1.0: \quad \begin{pmatrix} 0.000 & 0.000 & 0.001 & 0.002 & 0.001 & 0.000 & 0.000 \\ 0.000 & 0.003 & 0.013 & 0.022 & 0.013 & 0.003 & 0.000 \\ 0.001 & 0.013 & 0.059 & 0.097 & 0.059 & 0.013 & 0.001 \\ 0.002 & 0.022 & 0.097 & 0.159 & 0.097 & 0.022 & 0.002 \\ 0.001 & 0.013 & 0.059 & 0.097 & 0.059 & 0.013 & 0.001 \\ 0.000 & 0.003 & 0.013 & 0.022 & 0.013 & 0.003 & 0.000 \\ 0.000 & 0.000 & 0.001 & 0.002 & 0.001 & 0.000 & 0.000 \end{pmatrix}, \tag{29.10}$$

und

$$\sigma = 2.0: \quad \begin{pmatrix} 0.005 & 0.009 & 0.013 & 0.015 & 0.013 & 0.009 & 0.005 \\ 0.009 & 0.017 & 0.025 & 0.028 & 0.025 & 0.017 & 0.009 \\ 0.013 & 0.025 & 0.036 & 0.041 & 0.036 & 0.025 & 0.013 \\ 0.015 & 0.028 & 0.041 & 0.047 & 0.041 & 0.028 & 0.015 \\ 0.013 & 0.025 & 0.036 & 0.041 & 0.036 & 0.025 & 0.013 \\ 0.009 & 0.017 & 0.025 & 0.028 & 0.025 & 0.017 & 0.009 \\ 0.005 & 0.009 & 0.013 & 0.015 & 0.013 & 0.009 & 0.005 \end{pmatrix}. \tag{29.11}$$

Eine bessere Diskretisierung erhält man, wenn man die Umgebungen der m^2 Stützstellen berücksichtigt und die diskreten Werte durch eine numerische Integration bestimmt. Beispiele für Filterkerne der Größe $11 \cdot 11$ und $\sigma = 0.3$, $\sigma = 0.6$ und $\sigma = 0.9$ sind:

$$\sigma = 0.3: \quad \frac{1}{2^8} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 4 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 14 & 18 & 14 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 28 & 53 & 28 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 14 & 18 & 14 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 4 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \tag{29.12}$$

$$\sigma = 0.6: \quad \frac{1}{2^8} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 3 & 2 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 & 6 & 7 & 6 & 3 & 1 & 0 & 0 \\ 0 & 1 & 2 & 6 & 10 & 12 & 10 & 6 & 2 & 1 & 0 \\ 0 & 1 & 3 & 7 & 12 & 15 & 12 & 7 & 3 & 1 & 0 \\ 0 & 1 & 2 & 6 & 10 & 12 & 10 & 6 & 2 & 1 & 0 \\ 0 & 0 & 1 & 3 & 6 & 7 & 6 & 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 2 & 3 & 2 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \tag{29.13}$$

und

$$\sigma = 0.9: \quad \frac{1}{2^8} \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 & 2 & 3 & 1 & 1 & 1 & 0 \\ 0 & 1 & 2 & 2 & 3 & 3 & 3 & 2 & 2 & 1 & 0 \\ 1 & 1 & 2 & 4 & 5 & 5 & 5 & 4 & 2 & 1 & 1 \\ 1 & 2 & 3 & 5 & 6 & 6 & 6 & 5 & 3 & 2 & 1 \\ 1 & 2 & 3 & 5 & 6 & 7 & 6 & 5 & 3 & 2 & 1 \\ 1 & 2 & 3 & 5 & 6 & 6 & 6 & 5 & 3 & 2 & 1 \\ 1 & 1 & 2 & 4 & 5 & 5 & 5 & 4 & 2 & 1 & 1 \\ 0 & 1 & 2 & 2 & 3 & 3 & 3 & 2 & 2 & 1 & 0 \\ 0 & 1 & 1 & 1 & 2 & 2 & 3 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}. \tag{29.14}$$

Welche Filtergröße verwendet wird, hängt von der Größe der $m \cdot m$–Umgebung zur Erfassung der Grundtexturfläche ab (Abschnitt 27.2). Natürlich müssen die Filterkerne

nicht jedesmal neu berechnet werden, sondern können in einem vom *scale space filtering* unabhängigen Arbeitsgang einmal für verschiedene Umgebungen und mit verschiedenen σ-Werten berechnet und in einer Datenbank abspeichert werden.

Wenn ein Bild mit der Technik des *scale space filtering* verarbeitet werden soll, wird nun im ersten Schritt eine Folge von Gauß-gefilterten Bildern $\mathbf{C}(\sigma) = (c(x, y; \sigma))$ mit unterschiedlichen σ-Werten erzeugt.

Eine andere Möglichkeit, um zu einer Folge von tiefpassgefilterten Bildern zu kommen, ist der Einsatz der Gauß-Pyramiden (Kapitel 28). Wenn die einzelnen Schichten der Gauß-Pyramiden wieder auf die Originalgröße expandiert werden, so sind sie ebenfalls für die weitere Verarbeitung im Sinn des *scale space filtering* geeignet.

29.3.3 Berechnung der Oberflächen der Grauwertfunktion

Den nächsten Schritt bildet die Berechnung der Oberflächen der $m \cdot m$–Umgebungen. Die Oberfläche einer Struktur kann wie folgt berechnet werden:

$$\mathbf{O}(\sigma) = \int \sqrt{1 + \mathbf{C}_x(\sigma)^2 + \mathbf{C}_y(\sigma)^2} dx dy, \tag{29.15}$$

wobei $\mathbf{C}_x(\sigma)$ und $\mathbf{C}_y(\sigma)$ die partiellen Ableitungen der geglätteten Strukturen sind. In der praktischen Umsetzung sind dazu folgende Schritte notwendig:

- Berechnung der Ableitungsbilder $\mathbf{C}_x(\sigma)$ und $\mathbf{C}_y(\sigma)$ durch Differenzenbildung in Zeilen- und Spaltenrichtung.
- Berechnung der „Wurzelbilder“ $\mathbf{W}(\sigma) = (w(x, y; \sigma))$ gemäß

$$w(x, y; \sigma) = \sqrt{1 + c_x(x, y; \sigma)^2 + c_y(x, y; \sigma)^2}.$$

- Berechnung der Oberfläche einer $m \cdot m$–Umgebung durch Aufsummierung der Werte $w(x, y; \sigma)$ in der $m \cdot m$–Umgebung des Bildpunktes (x, y). Dieser Wert wird dem Bildpunkt in der Position (x, y) des Ergebnisbildes $\mathbf{O}(\sigma) = (o(x, y; \sigma))$ zugewiesen.

29.3.4 Berechnung des Skalenparameters

Nach Berechnung der Oberflächenbilder von Abschnitt 29.3.3 liegt eine Folge von Bildern $\mathbf{O}(\sigma)$ vor, die durch die Wahl unterschiedlicher σ-Werte bei der Gauß-Filterung zustande kamen und deren Grauwerte ein Maß für die Oberfläche der Grauwertfunktion von jeweils $m \cdot m$ Bildpunkten sind.

Für jeden Bildpunkt (x, y) wird jetzt der Skalenparameter p berechnet, indem zu der Folge von Punktepaaren $(\log(o(x, y; \sigma)), \log(\sigma)), \sigma = ...$ die Regressionsgerade (Ausgleichsgerade) berechnet wird. Der Skalenparameter p ist dann die Steigung dieser Geraden. Im Ergebnisbild $\mathbf{S}_a$ hat jeder Bildpunkt den Wert des Skalenparameters p seiner $m \cdot m$–Umgebung.

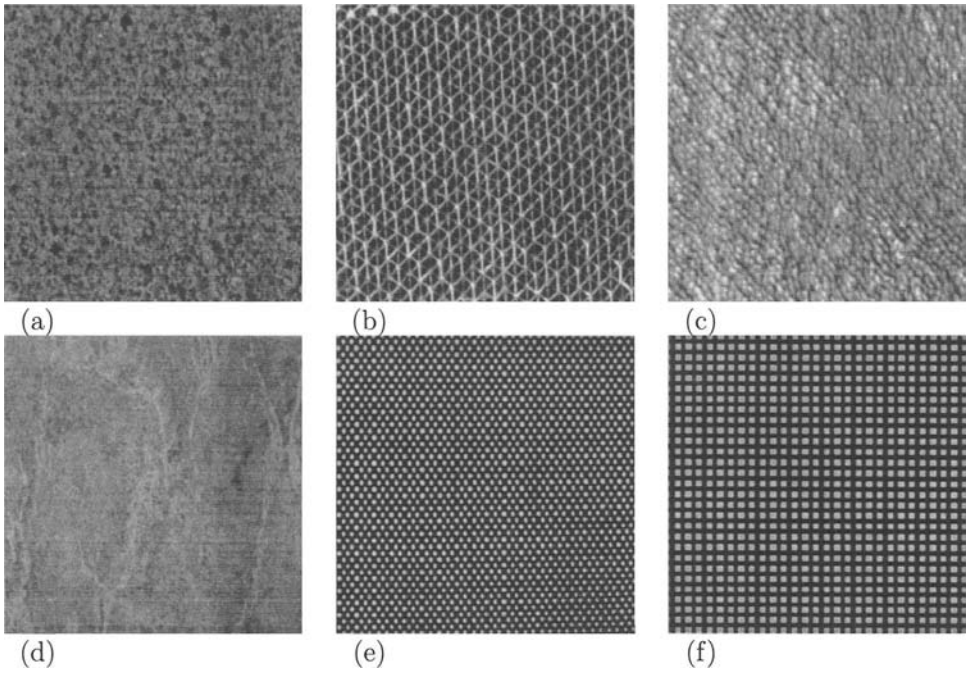


Bild 29.6: Sechs Texturen, deren Skalenverhalten untersucht wurde: (a) Schaumstoff, (b) Fahrradrückstrahler, (c) Frotteestoff, (d) Marmortapete, (e) Geschenkpapier, (f) synthetische Textur.

Wenn dieses Bild in die Grauwertmenge $G = \{0, ..., 255\}$ abgebildet wird, so werden Bereiche mit gleicher Textur gleiche Grauwerte besitzen. Im einfachsten Fall können jetzt die Bereiche mit gleicher Textur durch eine Schwellwertbildung getrennt werden. Natürlich kann das so erhaltene Texturmerkmal auch im Rahmen aufwendigerer Segmentierungsverfahren verwendet werden.

29.3.5 Beispiele und Ergebnisse

Die Texturen in Bild 29.6 wurden mit σ-Werten von 0.1, 0.2,...,0.9 und 0.95 und Umgebungen mit $5 \cdot 5$, $7 \cdot 7$, ... , $15 \cdot 15$ Pixel verarbeitet. In Bild 29.7 ist das Skalenverhalten dieser Texturen dargestellt.

Im markierten Bereich zeigen diese Texturen mit Ausnahme der Texturen (e) und (f) Skalenverhalten, d.h. sie gehorchen dem Potenzgesetz $O(\sigma) \sim \sigma^p$ und sind dadurch im

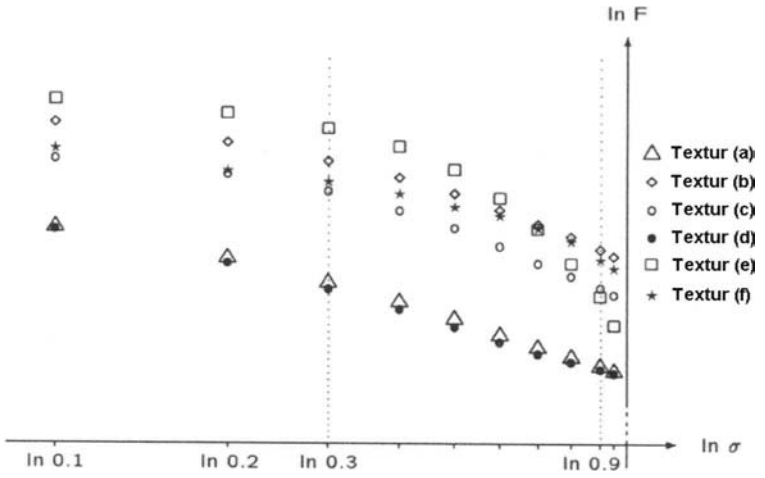


Bild 29.7: Skalenverhalten der sechs Texturen. Es zeigt sich, dass mit Ausnahme der Texturen (e) und (f) alle Texturen Skalenverhalten in einem markierten Bereich zeigen.

doppelt logarithmischen Papier durch eine Gerade repräsentiert. Die Texturen (e) und (f) sind offensichtlich keine fraktalen Texturen. Sie gehorchen deshalb auch nicht dem Potenzgesetz, was die dazu gehörigen Kurven bestätigen.

Ein Einflussparameter beim *scale space filtering* ist die Größe der $m \cdot m$–Umgebung, die zur Glättung und zur Berechnung der Oberflächen verwendet wird. Für die Texturen (a) und (b) sind die Ergebnisse der Filterung mit unterschiedlichen Umgebungen in Bild 29.8 gezeigt.

Bei Textur (a) ergeben sich Regressionsgeraden mit etwa gleicher Steigung. Bei Textur (b) ist das erst ab einer $13 \cdot 13$–Umgebung der Fall. Dies ist möglicherweise ein Hinweis, dass die Umgebung zur Erfassung der Grundtexturfläche erst ab dieser Größe ausreicht.

Die Technik des *scale space filtering* kann gut in der Qualitätskontrolle von Oberflächen eingesetzt werden: Wenn eine texturierte Oberfläche, z.B. ein Gewebe, Fehlerstellen (Löcher, Verschiebungen im Fadenlauf, ...) aufweist, so sind das Störungen in der Regelmäßigkeit der Textur. An diesen Stellen zeigt die Oberfläche ein anderes Skalenverhalten, was bedeutet, dass sich ein anderer Wert für den Skalenparameter p ergibt. Bild 29.9 zeigt Beispiele dazu. Die Bilder der oberen Reihe (29.9-a bis 29.9-c) zeigen ein Gewebe mit einer länglich ausgeprägten Fehlerstelle. Im mittleren Bild sind die Skalenparameter, abgebildet in die Grauwertmenge, dargestellt. Rechts ist das Ergebnis einer Binarisierung, in der die Fehlerstelle deutlich markiert ist. Die Teilbilder 29.9-d und 29.9-g zeigen eine Textur („Geschenkpapier") mit einer kleinen Störung und darunter das segmentierte Ergebnis. Die Teilbilder 29.9-e und 29.9-f sind Aufnahmen desselben Gewebes, das nur mit unterschiedlichem Abstand und mit einem unterschiedlichen Drehwinkel aufgenommen wurde.

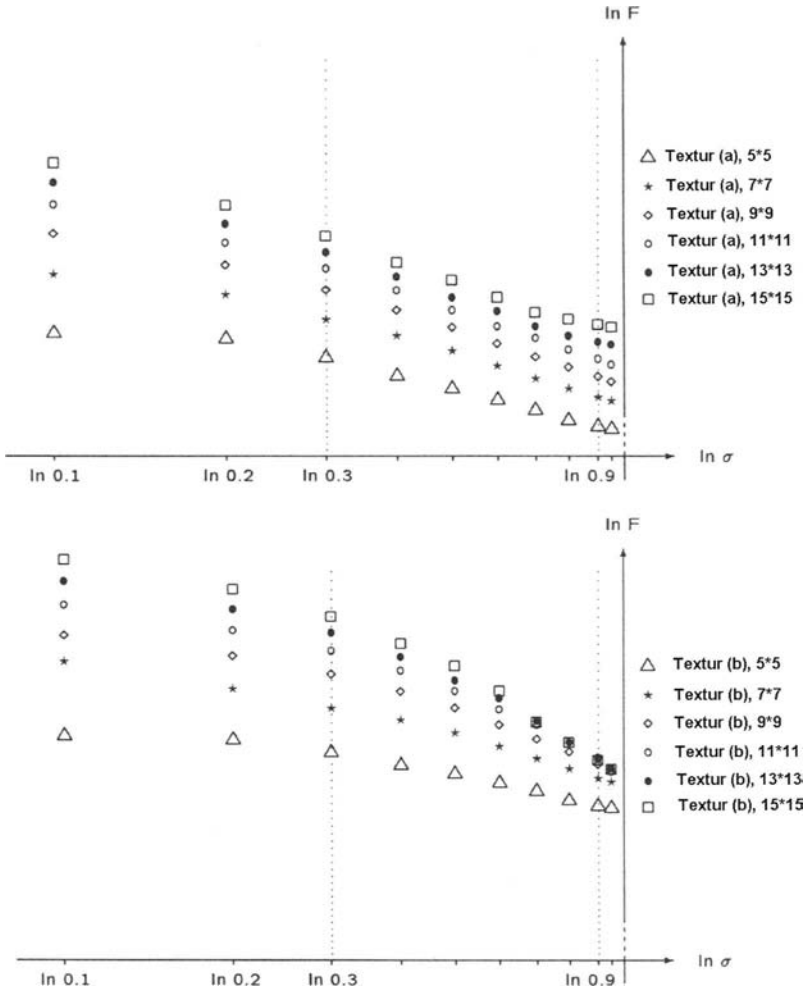


Bild 29.8: Skalenverhalten der Texturen (a) und (b) bei verschiedenen $m \cdot m$–Umgebungen. Bei Textur (a) ergeben sich Regressionsgeraden mit etwa gleicher Steigung. Bei Textur (b) ist das erst ab einer $13 \cdot 13$–Umgebung der Fall: Ein Hinweis, dass die Umgebung zur Erfassung der Grundtexturfläche erst ab dieser Größe ausreicht.

Darunter sind jeweils die Ergebnisse abgebildet. Diese beiden Beispiele zeigen, dass das Verfahren des *scale space filtering* größen- und rotationsinvariant ist.

Als Merkmal kann man im Ergebnisbild alternativ (oder zusätzlich) zum Skalenparameter auch die Verschiebung der Regressionsgeraden parallel zur Ordinate verwenden. In einigen der gezeigten Beispiele wurde dieses Merkmal verwendet.

Nun noch einige Bemerkungen zum Berechnungsaufwand. Wenn man die einzelnen notwendigen Schritte betrachtet, erkennt man, dass der Aufwand zur Berechnung der Ergebnisbilder mit dem Skalenparameter p als Merkmal beträchtlich ist. In der Praxis wird man jedoch den zu untersuchenden Bereich in Vorverarbeitungsschritten eingrenzen können, so dass sich die Größe deutlich reduziert. Außerdem kann man viele der notwendigen Operationen mit spezieller Faltungshardware erledigen, mit der sich die Rechenzeiten spürbar reduzieren.

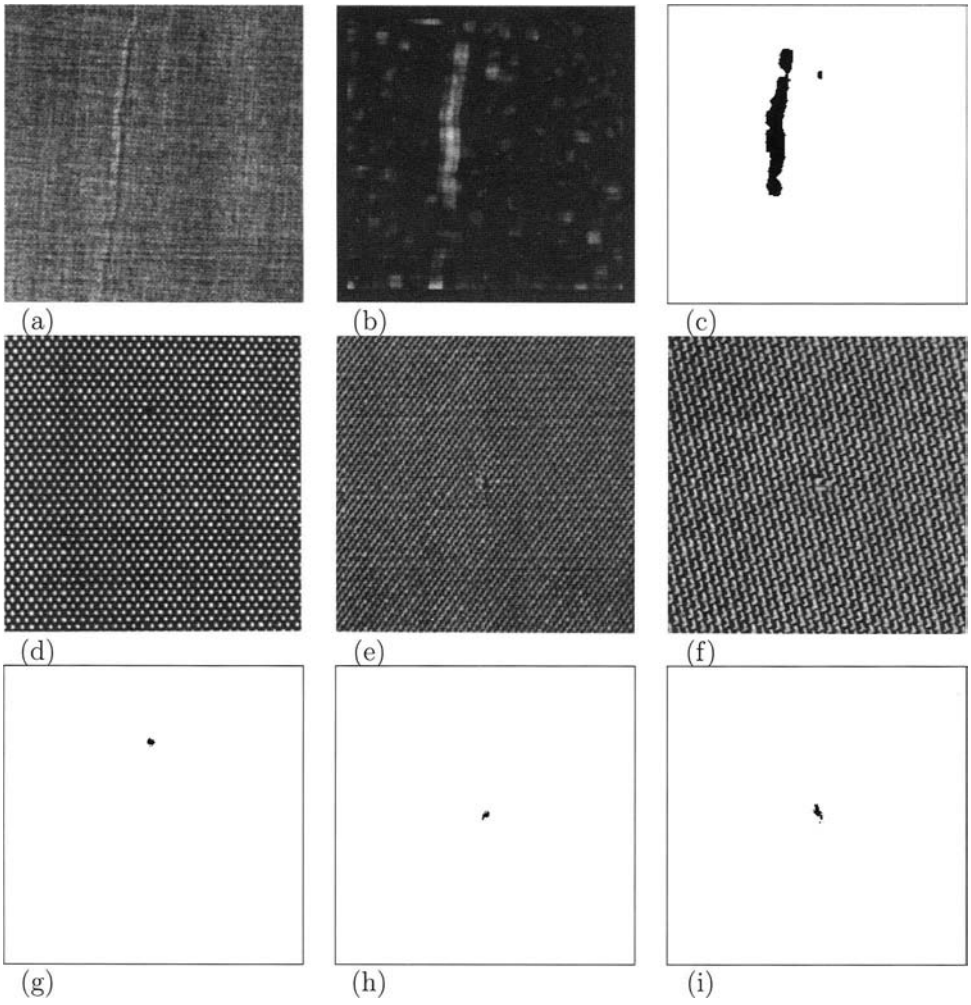


Bild 29.9: Beispiele zur Qualitätskontrolle von Oberflächen mit *scale space filtering*. (a) Gewebe mit einem länglich ausgeprägten Fehler. (b) Skalenparameter, abgebildet in die Grauwertmenge. (c) Binarisierung der Skalenparameter. Die Fehlerstelle ist deutlich markiert. (d) Textur „Geschenkpapier". (e) Textur „Stoff". (f) Textur „Stoff" aus einem anderen Abstand und einem anderen Drehwinkel aufgenommen. (g) Segmentiertes Bild zu (d). (h) Segmentiertes Bild zu (e). (i) Segmentiertes Bild zu (f). Es zeigt sich, dass das *scale space filtering* größen- und rotationsinvariant ist.

Kapitel 30

Baumstrukturen

30.1 Anwendungen

Neben den Gauß- und Laplac-Pyramiden (Kapitel 28) und der fraktalen Betrachtung beim *space scale filtering* (Kapitel 29) ist die Umwandlung eines Bildes in eine Baumstruktur eine weitere Variante der Betrachtung von Bilder in unterschiedlicher Feinheit. Es wird versucht, zusammenhängende Bildausschnitte der Größe $2 \cdot 2, 4 \cdot 4, 8 \cdot 8, \ldots$ mit homogenen Grauwerten in der Baumstruktur durch nur einen Knoten zu repräsentieren. Das bedeutet, dass Bilder mit großen zusammenhängenden, homogenen Bildbereichen sehr effektiv codiert werden können. Dieser Gesichtspunkt der *quad trees* wird vor allem in der Bilddatenreduktion und Bilddatenkompression verwendet.

Eine weitere Möglichkeit besteht in der Segmentierung von Bildern in homogene und inhomogene Bildbereiche. Auch diese Technik wird bei der Bilddatenreduktion und Bilddatenkompression verwendet, wenn homogene Bildbereiche feiner quantisiert werden sollen als inhomogene Bildbereiche.

Man kann Bilder, die als *quad trees* codiert sind, auch klassifizieren, wobei hier nicht einzelne Bildpunkte, sondern gesamte Bildbereiche, die durch einen Knoten des Baumes repräsentiert werden, einer Klasse zugewiesen werden.

Schließlich werden *quad trees* und ihre 3-dimensionale Verallgemeinerung *oct trees* auch in der 3D-Computergrafik verwendet (Abschnitt 15.6.2).

30.2 Grundlegende Vorgehensweise beim Aufbau von Baumstrukturen

Der grundlegende Algorithmus wird anhand eines Binärbildes $\mathbf{S}_e = (s_e(x, y))$ mit $G = \{0, 1\}$ erläutert. Es wird zunächst angenommen, dass das Binärbild quadratisch mit einer Seitenlänge von $L = R = 2^k$ ist. Bild 30.1-a zeigt diesen Sachverhalt für ein Binärbild mit $k = 3$, also $8 \cdot 8 = 64$ Bildpunkten. Bei der Konstruktion des Baumes, der letztlich das Binärbild $\mathbf{S}_e$ repräsentieren soll, wird den $2^k \cdot 2^k$ Bildpunkten die Wurzel des Baumes

zugeordnet. Die Wurzel ist ein Knoten der Stufe k. In Bild 30.1-a ist die Wurzel ein Knoten der Stufe 3.

Im nächsten Schritt werden die $2^k \cdot 2^k$ Bildpunkte des Bildes untersucht: Falls alle Bildpunkte den Grauwert 0 (schwarz) oder alle den Grauwert 1 (weiß) besitzen, ist das gesamte Bild homogen schwarz oder weiß. In diesem Fall besteht der Baum nur aus der Wurzel, in der vermerkt ist, ob alle Bildpunkte den Grauwert 0 oder 1 haben. Falls einige Bildpunkte den Grauwert 0 und einige den Grauwert 1 besitzen, was in der Regel der Fall ist, wird das Bild in seine vier Quadranten unterteilt. Diese vier Quadranten werden, beginnend mit dem nordwestlichen, wie folgt bezeichnet (Bild 30.1-a):

$$Q_0^{(k-1)}, Q_1^{(k-1)}, Q_2^{(k-1)}, Q_3^{(k-1)}.$$

Jeder der vier Quadranten, der $2^{(k-1)\cdot(k-1)}$ Bildpunkte umfasst, wird in der Baumstruktur durch einen Knoten der Stufe $l = k - 1$ repräsentiert. Die Wurzel des Baumes wird durch vier Kanten mit jedem ihrer „Söhne" (Nachfolger) verbunden. Nun wird für jeden der vier Quadranten sinngemäß dieselbe Untersuchung wie oben durchgeführt: Nur im Fall, dass in einem Quadranten die Grauwerte 0 und 1 auftreten, wird dieser wiederum in seine vier Quadranten unterteilt, denen im Baum Knoten der Stufe $l = k-2$ zugeordnet werden. Die Unterteilung in Quadranten endet spätestens nach k Schritten, wenn die Stufe der einzelnen Bildpunkte erreicht ist. Einem Bildpunkt ist in der Baumstruktur ein Knoten der Stufe $l = 0$ zugeordnet.

Knoten, die keine Nachfolger besitzen, werden als *Blätter des Baumes* oder *homogene Knoten* bezeichnet, da sie Bildausschnitte repräsentieren, die homogen schwarz oder weiß sind.

Die Adressierung der Knoten des Baumes erfolgt mit k-stelligen Zahlen zur Basis 4:

$$a_{k-1}, a_{k-2}, ..., a_1, a_0.$$

Die Ziffern $a_i \in (0, 1, 2, 3)$ stehen für die Position des zugehörigen Quadranten im Rahmen des ihn umgebenden Quadranten: 0-Nordwest, 1-Nordost, 2-Südwest und 3-Südost. Der Index i kennzeichnet die Stufe des Knotens. Durch die Stufennummer und die Knotenadresse ist jeder Knoten des Baumes eindeutig mit dem Bildausschnitt verbunden, den er repräsentiert. Die Wurzel in Bild 30.1-c hat die Stufennummer $l = 3$ und die Knotenadresse 000. Der nordöstliche Quadrant Q_1 hat die Stufennummer $l = 2$ und die Knotenadresse 100, und dem schraffiert markierten Bildpunkt entspricht ein Knoten der Stufe $l = 0$ mit der Knotenadresse 212.

Als Nächstes ist zu untersuchen, welche Informationen bei der Implementierung der Baumstruktur in den einzelnen Knoten abgespeichert werden müssen. Zunächst muss jeder Knoten seine Stufennummer und seine Knotenadresse enthalten, damit von der Baumstruktur aus der direkte Bezug zum Bild sichergestellt ist. Nur dadurch ist z.B. das Durchlaufen eines Pfades durch den Baum möglich. Weiter muss der Typ des Knotens festgehalten werden. Der Typ gibt an, ob es sich um einen homogenen weißen oder schwarzen Knoten handelt oder ob der Knoten inhomogen ist und Nachfolger besitzt. Die Kanten zwischen

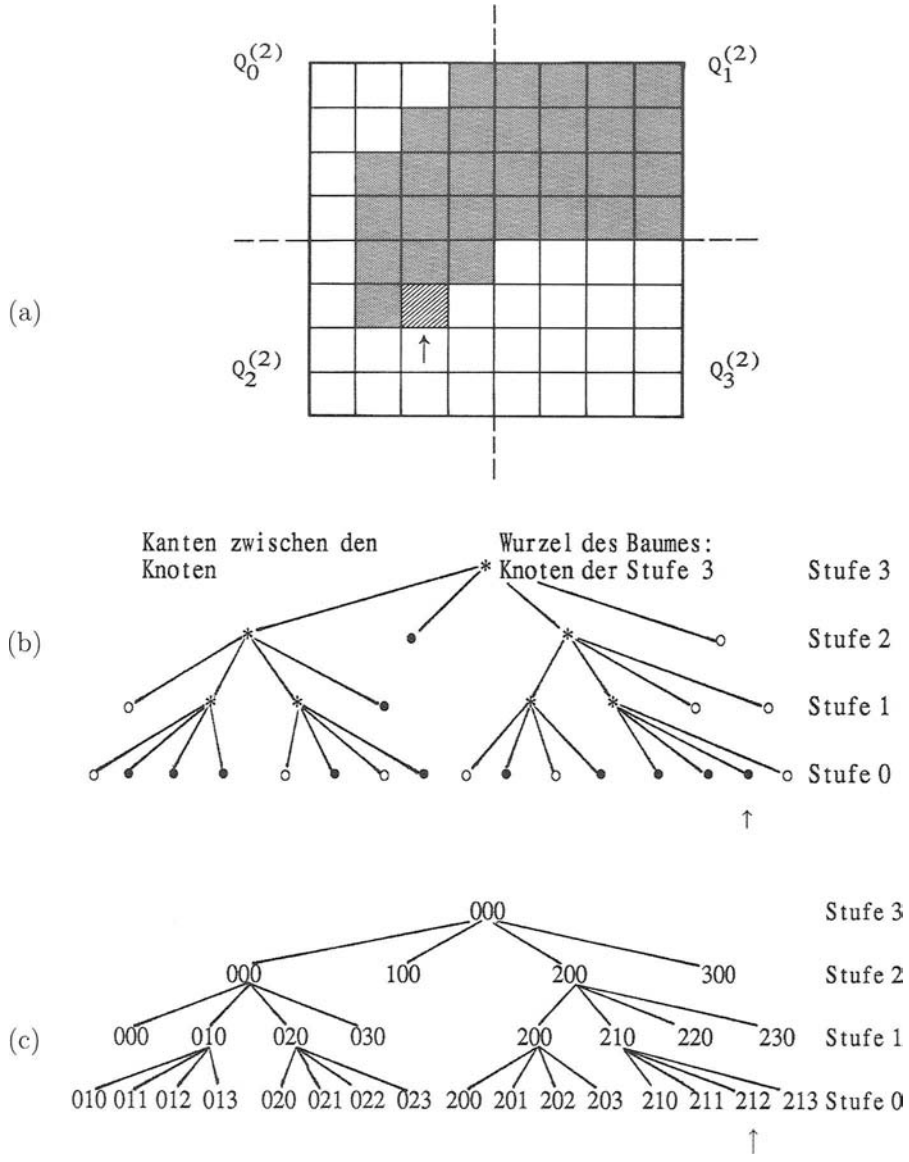


Bild 30.1: Beispiele zur Darstellung eines Bildes als Baumstruktur. (a) Originalbild mit $2^3 \cdot 2^3$ Bildpunkten. (b) Baumstruktur. (c) Adressierung der Knoten.

2	0	2	1
1	2	2	0
1	0	0	0
1	0	1	0
2	2	1	1
1	0	1	0
0	0	0	1

Tabelle 30.1: Baumstruktur zum Binärbild in Bild 30.1-a. Auf die Abspeicherung der Stufennummer, der Knotenadresse und der Adressverweise wurde in dieser Implementierungsvariante verzichtet. Die Rekonstruktion des Originals ist durch die spezielle, rekursive Speicherungstechnik möglich.

den Knoten werden durch Adressverweise realisiert. Jeder Knoten, mit Ausnahme der Blätter, hat vier Adressverweise auf seine vier Nachfolger („Söhne") und, mit Ausnahme der Wurzel, einen Adressverweis auf seinen Vorgänger („Vater"). Zusätzlich können in jedem Knoten noch weitere Informationen abgelegt werden, die jedoch abhängig vom jeweiligen Anwendungsfall sind.

Neben der hier beschriebenen Implementierung der Baumstruktur sind noch andere Varianten möglich. Wird in einer aktuellen Anwendung dem durch die Baumstruktur erzielten Kompressionsfaktor mehr Bedeutung beigemessen als der flexiblen Auswertungs- und Weiterverarbeitungsmöglichkeit, so kann z.B. auf die Stufennummer, die Knotenadresse und die Adressverweise der Kanten verzichtet werden. Allerdings muss dann auf Grund einer speziellen, rekursiven Speicherungstechnik diese Information bei der Rekonstruktion des Originals wieder abgeleitet werden können. Dazu als Beispiel der Sachverhalt von Bild 30.1. Zur Darstellung des Typs eines Knotens (schwarz, weiß oder schwarz/weiß) benötigt man 2 Bit. Man könnte also festlegen: 0 schwarzer Knoten, 1 weißer Knoten und 2 schwarz/weiß gemischter Knoten, der weiter unterteilt ist. Die vier Knoten der Stufe 2 können dann durch das Codewort 2021 beschrieben werden. Jetzt wird zu jeder 2 in diesem Codewort, die ja ein Hinweis darauf ist, dass der Knoten weiter unterteilt ist, der zugehörige Teilbaum abgespeichert. Dieses rekursive Verfahren endet für den jeweiligen Teilbaum, wenn in einem Codewort nur mehr Nullen und Einsen enthalten sind. Für den Baum von Bild 30.1-b ergibt sich dann die Struktur von Tabelle 30.1.

Als weitere Variante ist es möglich, von der Unterteilung in vier Quadranten abzugehen und z.B. eine Zerlegung in $4 \cdot 4$ Teilquadrate vorzunehmen, was einer Aufteilung in 16 Knoten entspricht. Ein Codewort, das diese 16 Knoten in der oben beschriebenen Weise darstellt, benötigt 32 Bit. Implementierungen dieser Art lassen sich leicht auf byteorientierten Rechenanlagen durchführen. Es sei aber darauf hingewiesen, dass durch das Weglassen der Stufennummer, der Knotenadresse und der Adressverweise zwar ein optimaler Kompressionsfaktor erreicht wurde, jedoch die Auswertungsmöglichkeiten der Baumstruktur eingeschränkt wurden. In den folgenden Betrachtungen wird immer vorausgesetzt, dass in

den Knoten die volle Information verfügbar ist.

Eine Möglichkeit der Weiterverarbeitung von Binärbildern, die als Baumstrukturen codiert sind, ist die logische Verknüpfung von zwei Bäumen. Dazu müssen zunächst die Boole'schen Operatoren NOT, AND und OR für Bäume als Verknüpfungsobjekte implementiert werden. Dies ist direkt in der Baumstruktur möglich, es ist also nicht notwendig, das als Baum gespeicherte Bild zuerst in die Rasterdarstellung zurückzuwandeln. Die logische Verknüpfung von Bäumen ist auch in der komprimierten Form gemäß Bild 30.1 möglich. Es seien $\mathbf{S}_1$ und $\mathbf{S}_2$ zwei Binärbilder in ihrer Darstellung als Baumstruktur. Dann wird z.B. durch die logische Operation NOT $\mathbf{S}_1$ der Baum erzeugt, der dem Negativbild von $\mathbf{S}_1$ entspricht. Durch $\mathbf{S}_1$ OR $\mathbf{S}_2$ wird der Baum erzeugt, der sowohl den Bildinhalt von $\mathbf{S}_1$ als auch von $\mathbf{S}_2$ enthält, also die Vereinigung der beiden Originale $\mathbf{S}_1$ und $\mathbf{S}_2$. Schließlich erzeugt $\mathbf{S}_1$ AND $\mathbf{S}_2$ einen Baum, in dem nur diejenigen Bildbereiche den Grauwert 1 haben, die sowohl in $\mathbf{S}_1$ als auch in $\mathbf{S}_2$ den Grauwert 1 haben, was dem Durchschnitt der beiden Originale entspricht.

Dazu ein praktisches Beispiel: $\mathbf{S}_1$ sei eine Strichzeichnung, z.B. eine technische Dokumentation, eine Explosionszeichnung oder ein Kartenausschnitt. Nun soll aus diesem als Baumstruktur gespeicherten Bild ein rechteckiges Fenster ausgeblendet werden. $\mathbf{S}_2$ ist dann die Baumstruktur zu einem Rechteckausschnitt, der innerhalb des auszublendenden Bereiches den Grauwert 1 und außerhalb den Grauwert 0 hat. $\mathbf{S}_1$ AND $\mathbf{S}_2$ liefert den Baum, der nur die Bildinformation von $\mathbf{S}_1$ innerhalb des mit $\mathbf{S}_1$ definierten Fensters hat. Anwendungen dieser Art sind bei der digitalen Verarbeitung von Strichzeichnungen sinnvoll, die auf Grund ihres Bildinhaltes nicht mit grafischen, vektoriellen Systemen verarbeitet werden können. Da diese Bilder in der Regel mit einem sehr feinen Raster digitalisiert werden müssen, fallen große Datenmengen an. Zur Kompression können hier die Baumstrukturen eingesetzt werden. Es besteht dann die Möglichkeit einer interaktiven Weiterverarbeitung, z.B. einer Modifikation der Zeichnung, wie oben geschildert, direkt in der komprimierten Datenstruktur.

Bei der Beschreibung des grundlegenden Algorithmus wurde angenommen, dass die Binärbildvorlagen quadratisch sind und ihre Seitenlänge eine Potenz von zwei ist. Dies kann bei vielen praktischen Anwendungen nicht vorausgesetzt werden. Um trotzdem einen Baum aufbauen zu können, muss das i. allg. rechteckige Bild mit einem definierten Hintergrundgrauwert wie in Bild 30.2 dargestellt, aufgefüllt werden, um zu einem quadratischen Bild mit einer 2-er Potenz als Seitenlänge zu kommen.

Abschließend zu diesem Abschnitt noch einige Bemerkungen, die jedoch im nächsten Abschnitt noch ausführlicher behandelt werden. Beim Aufbau des Baumes zu einem Binärbild wurde als Kriterium, ob ein Quadrant unterteilt wird oder nicht, die Homogenität „ganz weiß", „ganz schwarz" oder alternativ dazu „schwarz/weiß" verwendet. Soll der Baumalgorithmus auf Grautonbilder verallgemeinert werden, so muss ein anderes Homogenitätskriterium verwendet werden, z.B. ob alle Grauwerte des untersuchten Ausschnittes innerhalb eines bestimmten Grauwertintervalls liegen. Ist dies nicht der Fall, wird der Ausschnitt in seine vier Quadranten unterteilt. Repräsentierend für das Grauwertintervall wird in einem Feld „TYP" jedes Knotens der Mittelwert des Intervalls eingetragen. Es ist offensichtlich, dass ein so codiertes Grauwertbild nicht mehr eindeutig reproduziert werden

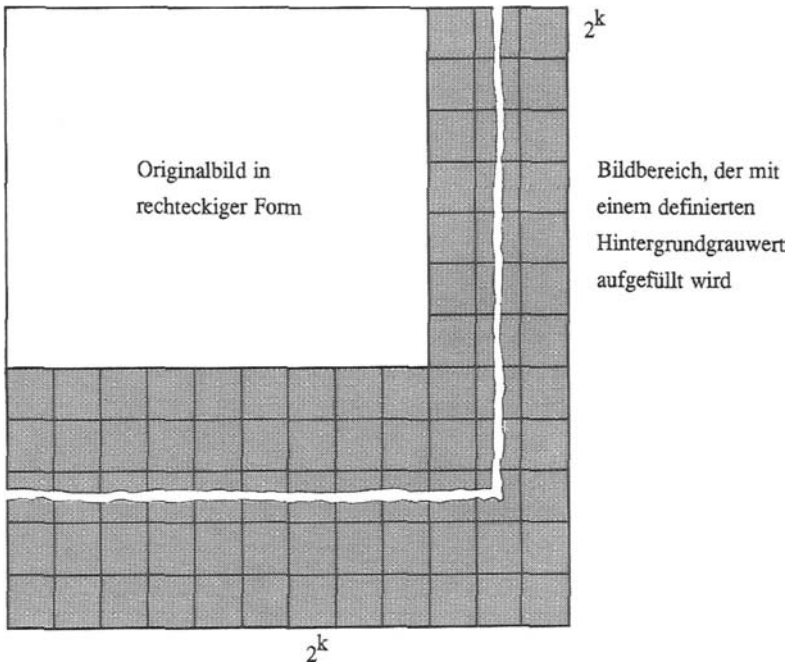


Bild 30.2: Rändelung eines rechteckigen Originalbildes mit einem definierten Hintergrundgrauwert, um zu einem quadratischen Bild mit einer 2-er Potenz als Seitenlänge zu kommen, das in eine Baumstruktur gewandelt werden kann.

kann. Ein Beispiel einer bildlichen Darstellung eines als Baum codierten Grauwertbildes wird im nächsten Abschnitt gegeben.

30.3 Regionenorientierte Bildsegmentierung mit *quad trees*

In Abschnitt 30.2 wurde auf die Darstellung von Bildern, besonders von Binärbildern, als Baumstrukturen eingegangen. Die Motivation des Einsatzes der Baumstrukturen war dort die Möglichkeit der Datenkompression. Es wurde aber bereits darauf hingewiesen, dass diese Technik auch gut zur Bildsegmentierung geeignet ist.

Beim Aufbau eines Baumes muss ein Homogenitätskriterium H angegeben werden, wann die Unterteilung in die vier Quadranten erfolgen soll. Im Fall von Binär- oder Zweipegelbildern wurde in Abschnitt 30.2 für H verwendet, ob alle Bildpunkte des betrachteten Quadranten weiß, schwarz oder schwarz und weiß sind. Dieses Kriterium kann folgendermaßen formuliert werden:

$$H_1 : min = max, \tag{30.1}$$

wobei min und max der minimale und der maximale Grauwert des Bildausschnittes sind.

Bei Grauwertbildern $\mathbf{S} = (s(x,y))$, $G = \{0,1,...,255\}$ wird die Aufteilung in die vier Quadranten nicht durchgeführt, wenn die Grauwerte des Bildausschnittes in bestimmten Grenzen liegen. Damit lautet das Homogenitätskriterium z.B.

$$H_2 : \max\{|mean - max|, |mean - min|\} < c \tag{30.2}$$

oder

$$H_3 : max - min < c, \tag{30.3}$$

wobei $mean$, max und min der Mittelwert, der maximale und der minimale Grauwert des Bildausschnittes sind. c ist ein Schwellwert, mit dem festgelegt werden kann, wie breit das Grauwertintervall ist, in dem die Grauwerte zu einem Knoten zusammengefasst werden.

Die Bildfolge 30.3 zeigt ein Beispiel eines als *quad tree* codierten Grauwertbildes. Es hat ein Originalgröße von $(256 = 2^8) \cdot (256 = 2^8)$ Bildpunkten. Somit hat die Baumstruktur die Stufen 0 bis 8: Die Wurzel ist ein Knoten der Stufe 8, die einzelnen Bildpunkte sind Knoten der Stufe 0. Von links oben nach rechts unten sind die Stufen 8 bis 0 des Bildes als Grauwertbilder dargestellt, wobei die Flächen, die ein Knoten im Originalbild repräsentiert, durch den mittleren Grauwert dargestellt sind.

Bei mehrkanaligen Bildern $\mathbf{S}_e = (s_e(x,y,n)), n = 0,1,...,N-1$ wird nicht aufgeteilt, wenn H für alle Kanäle zutrifft, also:

$$H_4 : \max\{|mean_i - max_i|, |maen_i - min_i|\} < c, i = 0,1,...,N-1. \tag{30.4}$$

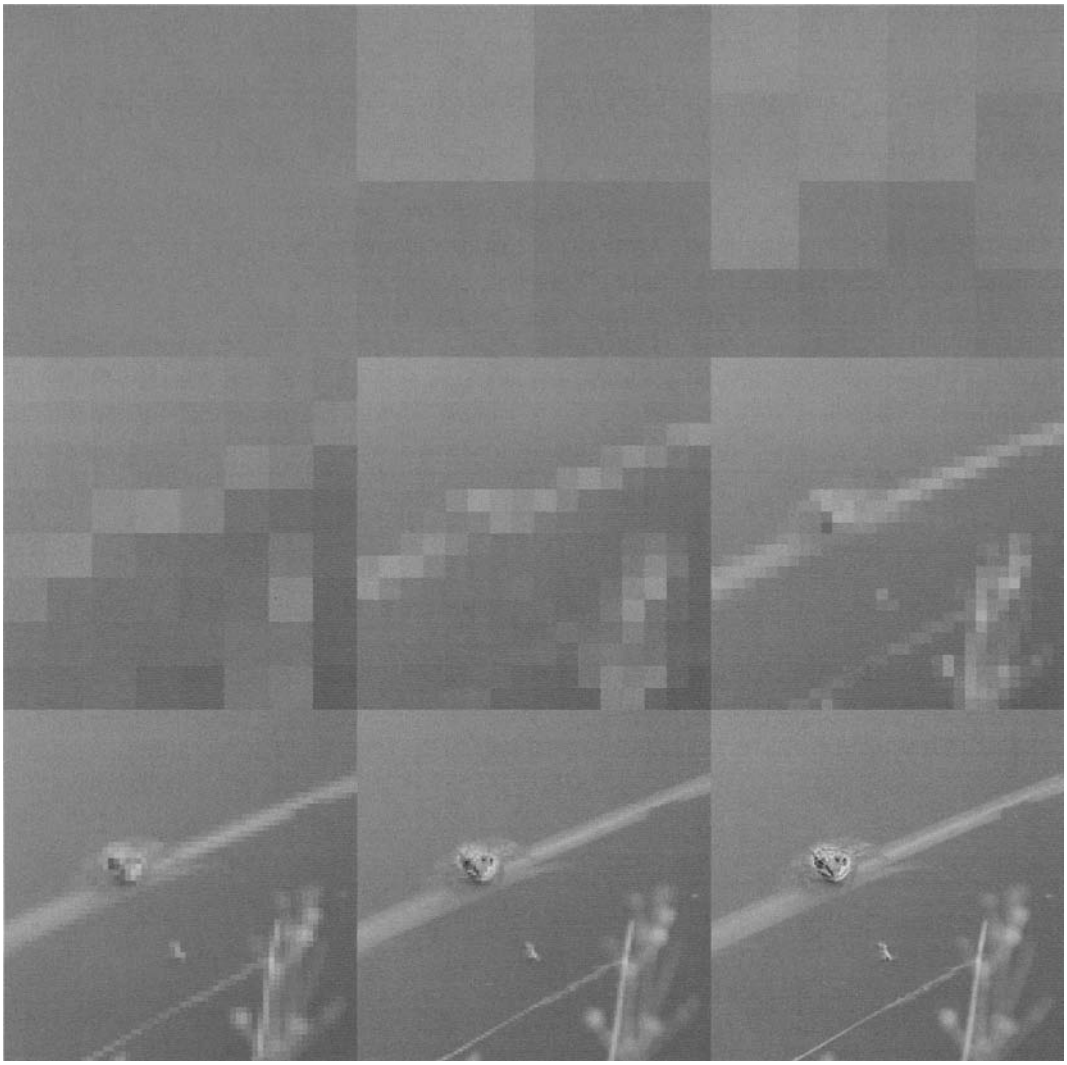


Bild 30.3: Als *quad tree* codiertes Grauwertbild der Größe $(256 = 2^8) \cdot (256 = 2^8)$ Bildpunkte. Die Baumstruktur hat die Stufen 0 bis 8: Die Wurzel ist ein Knoten der Stufe 8, die einzelnen Bildpunkte sind Knoten der Stufe 0. Von links oben nach rechts unten sind die Stufen 8 (Wurzel) bis 0 (einzelne Bildpunkte) des Bildes als Grauwertbilder dargestellt, wobei die Flächen die ein Knoten im Originalbild repräsentiert, durch den mittleren Grauwert dargestellt sind.

Dabei sind $mean_i, max_i$ und min_i wie oben die entsprechenden Werte für jeden der N Kanäle.

Die Wahl des Schwellwertes c beeinflusst den Baumaufbau wesentlich. Bei kleinem c ergibt sich eine große Knotenzahl, wodurch der Speicherbedarf und die Rechenzeit für nachfolgende Auswertungen steigt. Bei zu großem Schwellwert c geht Bildinformation verloren, da Bildinhalte mit geringen Grauwertunterschieden in homogenen Flächen aufgehen. Hier muss somit ein Kompromiss zwischen Speicher- und Rechenzeitaufwand und der Qualität des als Baum kodierten Bildes gefunden werden. Bei praktischen Anwendungen wird der Schwellwert c am besten empirisch ermittelt. So kann z.B. die Untersuchung der Grauwerte in strukturierten Bildbereichen, die von der Baumstruktur noch aufgelöst werden sollen, einen Hinweis auf den Schwellwert c geben.

Der Einfluss des Schwellwertes c wird mit den Bildern 30.4-a bis 30.4-d dokumentiert. Die Bilder wurden mit den Schwellwerten $c = 1, c = 4, c = 16$ und $c = 64$ und H_3 als Homogenitätskriterium codiert. Anschließend wurden sie in ein normales Rasterbild zurückgewandelt, um so die Auswirkung der unterschiedlichen Schwellwerte auf die Bildqualität darstellen zu können. Bei Bild 30.4-a ($c = 1$) wird unterteilt, wenn in dem jeweiligen Quadranten auch nur ein Bildpunkt ist, dessen Grauwert anders ist als die Grauwerte der restlichen Bildpunkte. Das bedeutet, dass die Baumstruktur das Grauwertbild exakt repräsentiert. Bei der Rückrasterung entsteht wieder das Original. Bei einem Schwellwert von $c = 4$ wird unterteilt, wenn die Abweichung in einem Quadranten größer oder gleich 4 ist (Bild 30.4-b). Hier ist im zurückgerasterten Bild noch kein wahrnehmbarer Unterschied zum Original festzustellen. Bei $c = 16$ (Bild 30.4-c) sieht man im ziemlich homogenen Hintergrund bereits eine Verschlechterung. Sehr deutlich wird es bei $c = 64$ (Bild 30.4-d). Allerdings werden die sehr fein strukturierten und kontrastreichen Bildbereiche (der Kopf des Frosches) auch hier noch bis auf Pixelebene aufgelöst. Dieser Sachverhalt ist in den drei Bildern Bild 30.5-a bis Bild 30.5-c dargestellt. Bild 30.5-c ist das Differenzbild zwischen dem Original (30.5-a) und der zurückgerasterten Baumstruktur mit $c = 64$. Zu den Differenzen wurde der Grauwert 128 addiert und das Ergebnis im Kontrast etwa angehoben, um die Unterschiede besser sichtbar zu machen. Man sieht hier deutlich, dass der stark strukturierte Kopf des Frosches noch fein aufgelöst wird.

Für jeden Knoten müssen Informationen gespeichert werden, die die Lage des Knotens im Baum beschreiben (Verweise zum Vater und zu den vier Söhnen), eine Zuordnung des durch den Knoten repräsentierten Ausschnittes im Originalbild gestatten (Knotenadresse und Stufe des Knotens) und eine möglichst gute Rekonstruktion der Grauwerte des Bildausschnittes erlauben. In Tabelle 30.2 sind die Knoteninformationen bei einer Anwendung der Baumstrukturen zur Bildsegmentierung von digitalisierten Luftbildern mit der Zielsetzung der Erstellung von thematischen Karten zusammengestellt ([Habe83], [Habe84]). Die digitalisierten Luftbilder lagen in diesem Fall als dreikanalige Bilder mit Rot-, Grün- und Blauauszug vor.

Der Algorithmus zum Aufbau der Baumstruktur wurde in Abschnitt 30.2 beginnend mit Wurzel des Baumes, also *top-down*, beschrieben. Diese Vorgehensweise hat den Nachteil, dass entweder große Bildausschnitte im Hauptspeicher des Datenverarbeitungssystems gehalten werden müssen oder viele Zugriffe auf den Hintergrundspeicher erfolgen. Die Baum-

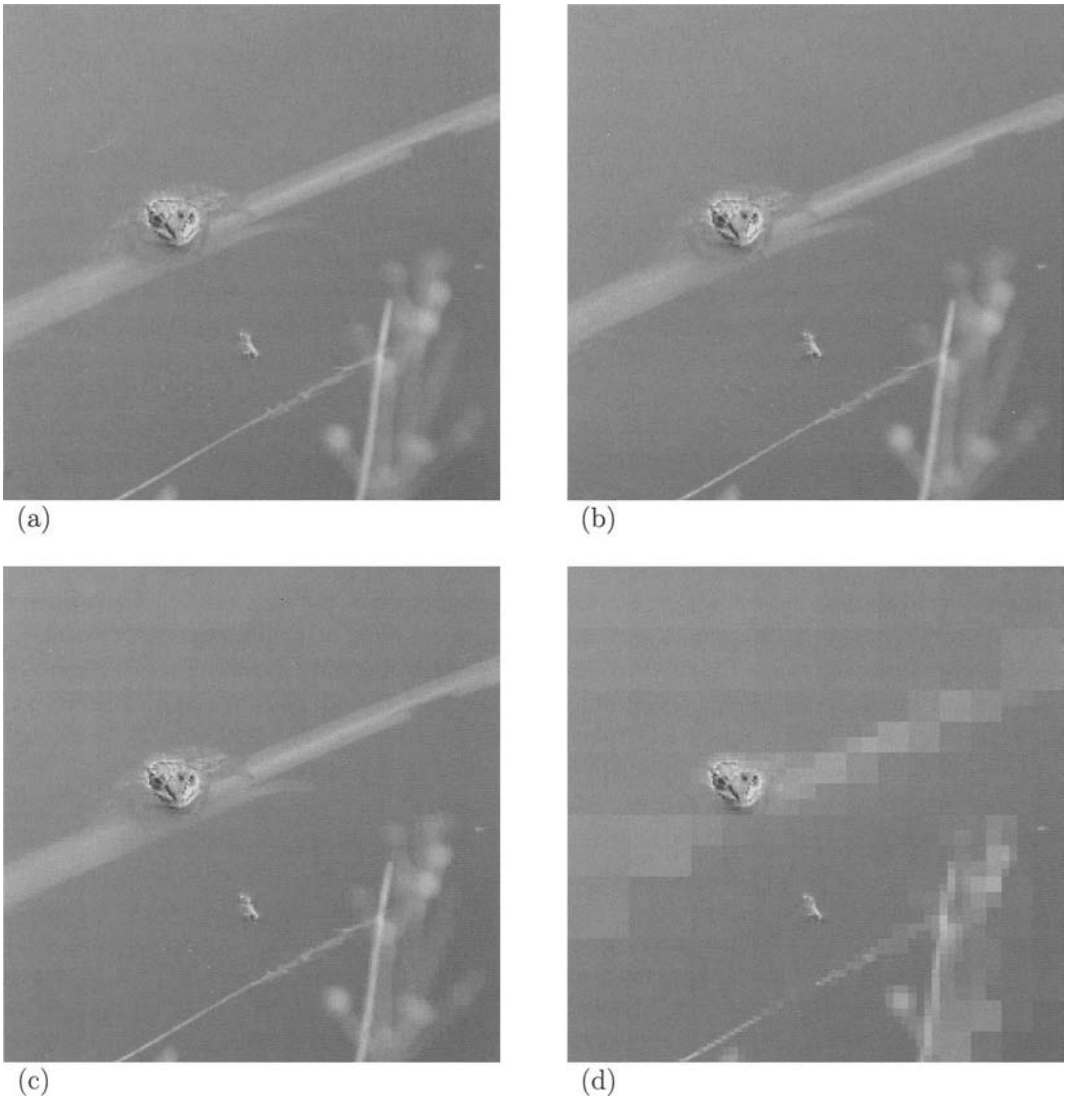


Bild 30.4: Der Einfluss des Schwellwertes c auf die Bildqualität bei der Rückrasterung. Je größer der Schwellwert, desto größere Grauwertintervalle werden zu einem Knoten zusammengefasst.

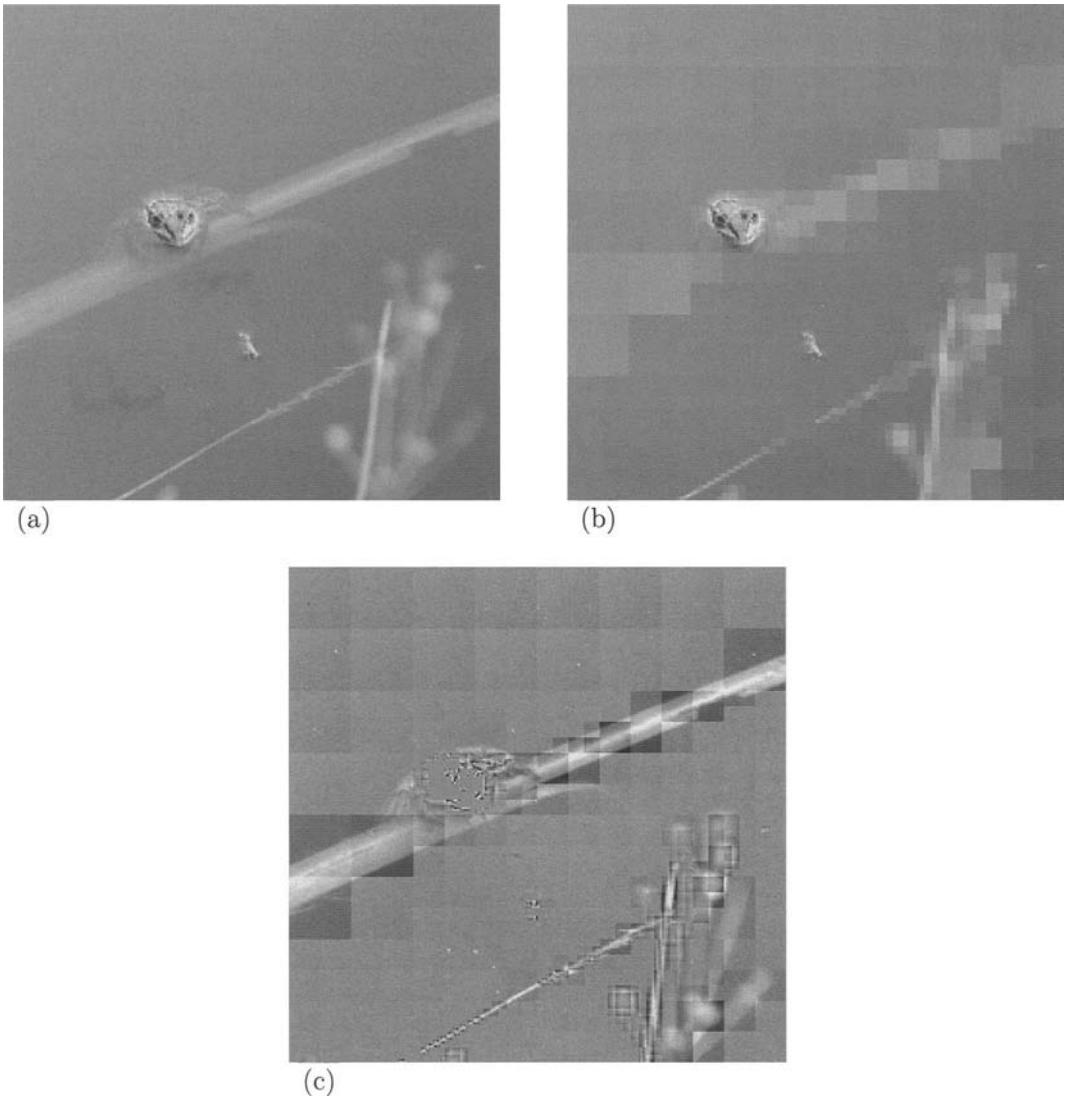


Bild 30.5: Differenzbild zwischen dem Original (a) und der mit einem Schwellwerte $c = 64$ erzeugten und zurückgerasterten Baumstruktur. Man sieht hier deutlich, dass der stark strukturierte Kopf des Frosches noch fein aufgelöst wird.

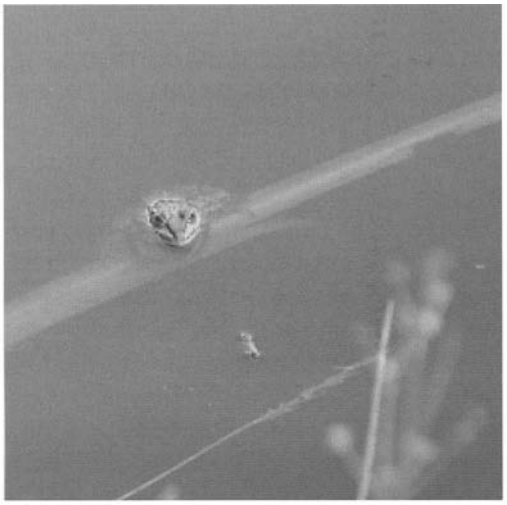
Zelle	Byte	Bezeichnung	Bedeutung
1	1	STUFE	Stufe l des Knotens
2	4	ADRESSE	Knotenadresse
3	4	VATER	Verweis auf den Vater
4	4	SOHN 0	Verweis auf den NW-Nachfolger
5	4	SOHN 1	Verweis auf den NO-Nachfolger
6	4	SOHN 2	Verweis auf den SW-Nachfolger
7	4	SOHN 3	Verweis auf den SO-Nachfolger
8	1	MITTEL	Mittelwert der $2^l \cdot 2^l$ Bildpunkte aus allen Kanälen
9	1	MINIMUM	Minimum der $2^l \cdot 2^l$ Bildpunkte aus allen Kanälen
10	1	MAXIMUM	Maximum der $2^l \cdot 2^l$ Bildpunkte aus allen Kanälen
11	1	MITTEL-GRÜN	Mittelwert der $2^l \cdot 2^l$ Bildpunkte im Grünkanal
12	1	MITTEL-ROT	Mittelwert der $2^l \cdot 2^l$ Bildpunkte im Rotkanal
13	1	MITTEL-BLAU	Mittelwert der $2^l \cdot 2^l$ Bildpunkte im Blaukanal
14	2	NACHFOLGER	Anzahl der Nachfolgeknoten des zugehörigen Teilbaumes
15	2	KNOTENCODE	(Zwischen-)Ergebniscode der Bildsegmentierung

Tabelle 30.2: Beispiel zur Struktur eines Baumknotens

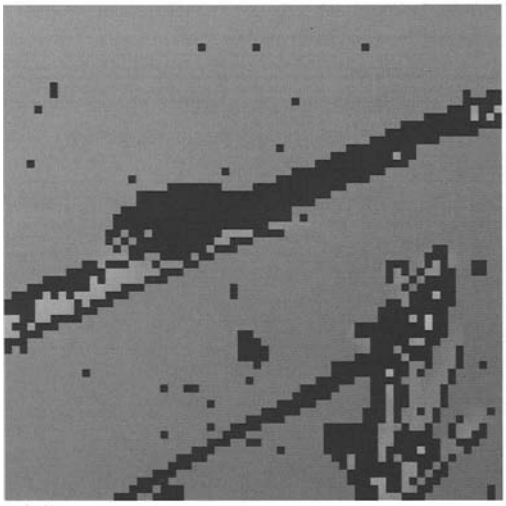
struktur kann auch während des sequentiellen Einlesens der Originalbilddaten von einem Hintergrundspeicher (oder einem Scanner) durchgeführt werden, wobei immer nur maximal zwei Bildzeilen im Hintergrundspeicher benötigt werden. Zu jeweils zwei Bildzeilen werden Teilbäume erzeugt, deren Wurzeln die Stufe 1 haben. Vier benachbarte Teilbäume einer Stufe $l, 1 \leq l \leq k-1$, werden dann zu einem Teilbaum der Stufe $l+1$ zusammengefasst, wobei durchaus der neue Teilbaum nur aus der Wurzel bestehen kann, wenn seine vier Söhne das Homogenitätskriterium erfüllen. Das Verfahren endet, wenn die letzten vier Knoten der Stufe $k-1$ zur Wurzel des Baumes zusammengefasst werden.

Um eine weitere Verarbeitung von Bildern, die als Baumstrukturen gespeichert sind, zu ermöglichen, müssen eine Reihe von Basisalgorithmen für Baumstrukturen implementiert sein. Ein Beispiel dazu ist ein Algorithmus, der ein als Baum gespeichertes Bild wieder in die normale Rasterbilddarstellung zurückwandelt. Ein derartiges Programm kann so implementiert werden, dass Ausgabebilder erzeugt werden, in denen homogene und inhomogene Bildbereiche getrennt dargestellt werden können, was ein erster Ansatz zur automatischen Aufteilung eines Bildes in homogene, also unstrukturierte, und inhomogene, also texturierte, Bildbereiche ist. In Bild 30.6-b und Bild 30.6-c wird dieser Sachverhalt am Beispiel eines Testbildes dargestellt. Bild 30.6-b zeigt alle homogenen Knoten bis zur minimalen Stufe l=3, während Bild 30.6-c alle inhomogenen Knoten bis zur maximalen Stufe 2 zeigt. Beim Aufbau des Baumes wurde hier das Homogenitätskriterium H_3 mit einem Schwellwert $c = 8$ verwendet.

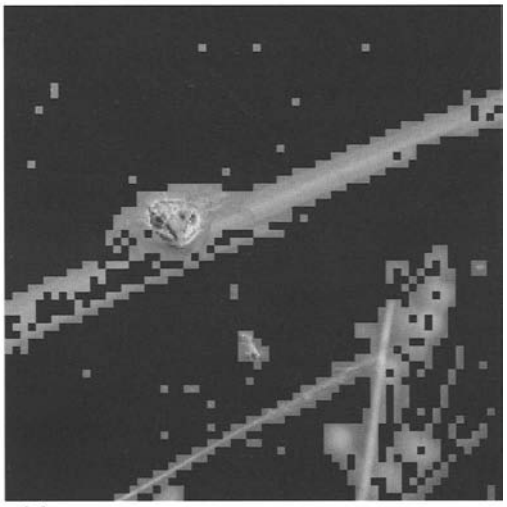
Bei vielen Anwendungen ist es notwendig, zur Inspektion von benachbarten Bildberei-



(a)



(b)



(c)

Bild 30.6: Rückrasterung des als Baumstruktur gespeicherten Testbildes. (a) Original (b) zeigt die homogenen Knoten bis zur minimalen Stufe $l = 3$ und (c) die inhomogenen Knoten bis zur maximalen Stufe $l = 2$.

chen in der Baumstruktur Pfade zu durchlaufen. Hierzu werden Nachbarschaftsalgorithmen benötigt, die es erlauben, ausgehend von einem beliebigen Knoten die Nachbarknoten in den acht Hauptrichtungen zu ermitteln. Das Auffinden eines Nachbarknotens basiert auf dem Auffinden eines gemeinsamen Vorgängers. Wird der Pfad zum gemeinsamen Vorgänger anschließend gespiegelt durchlaufen, so führt dieser Pfad zum Nachbarknoten [Same82]. Da die Stufe des Nachbarknotens höher, gleich oder niedriger sein kann, müssen hier Fallunterscheidungen berücksichtigt werden. Zunächst wird der Nachbar mit derselben oder größeren Stufe in einer der vier Richtungen Nord, West, Süd oder Ost ermittelt. Ist der Nachbarknoten in der gewünschten Richtung nicht homogen, so werden im zweiten Schritt die direkt anliegenden Nachbarn gesucht. Soll ein diagonal liegender Nachbar, z.B. der nordöstliche Nachbar eines Knotens, ermittelt werden, so wird dies zweistufig durchgeführt: Zuerst wird der östliche Nachbar und dann von diesem der nördliche bestimmt. Wegen der schon erwähnten Möglichkeit der unterschiedlichen Knotenstufen sind auch hier viele Fallunterscheidungen zu beachten.

Die Baumstrukturen legen es nahe, bei der Segmentierung Strategien einzusetzen, die auf die jeweilige Objektklasse abgestimmt sind. Wie schon oben erwähnt, ist es einfach, mit Hilfe der Baumstrukturen homogene von inhomogenen Bildbereichen abzugrenzen. Auf die homogenen Bildbereiche (z.B. Wasserflächen in einem Luftbild) werden dann spezielle Objektklassifikatoren angewendet. Ein derartiger Objektklassifikator wird im ersten Schritt alle homogenen Knoten des Baumes anhand bildpunktbezogener Merkmale klassifizieren und feststellen, ob es sich dabei um die gesuchte Klasse handelt. Dabei ist anzumerken, dass für größere homogene Flächen, d.h. für Ausschnitte, die durch Knoten mit einer höheren Stufennummer repräsentiert werden, nur eine Klassifizierungsentscheidung notwendig ist, was wesentlich zur Einsparung von Rechenzeit beiträgt.

Um im nächsten Schritt größere homogene Flächen zu extrahieren, werden alle Knoten, die im ersten Schritt als zur gesuchten Klasse gehörig erkannt wurden und eine bestimmte Minimalgröße erfüllen, darauf hin untersucht, ob sie ganz von Knoten derselben Art umgeben sind (8-Nachbarschaft). Diejenigen Knoten, für die diese Bedingungen erfüllt sind, werden gesondert markiert, da sie keiner weiteren Verarbeitung mehr unterzogen werden müssen. Zu den verbleibenden Knoten der gesuchten Klasse wird eine Randsuche durchgeführt, bei der schalenartig Knoten bis zur Stufe 0 der bereits erkannten Fläche zugewiesen werden.

Als Nächstes ein Beispiel für einen Objektklassifikator, der flächig ausgedehnte, inhomogene Bildbereiche segmentiert (z.B. Waldflächen in einem Luftbild). Ausgehend von dem Grundgedanken, dass die gesuchte Klasse großflächig ausgeprägt ist, wird zunächst ein Baumstrukturmaß ausgewertet, das bereits beim Baumaufbau berechnet werden kann. Dieses Maß beschreibt die Anzahl der Nachfolgeknoten eines bestimmten Knotens. Hat z.B. die Wurzel eines (Teil-)Baumes nur wenige Nachfolger, so repräsentiert sie ein Bild, das weitgehend aus homogenen Bereichen aufgebaut ist. Folgt auf die Wurzel des (Teil-) Baumes dagegen die maximal mögliche Anzahl von Nachfolgeknoten, die sich gemäß

$$N = \sum_{l=0}^{k} 4^l \tag{30.5}$$

berechnet, so beschreibt sie einen Bildausschnitt, der eine maximale Strukturiertheit aufweist. Inhomogene Knoten, die texturierte Gebiete repräsentieren, zeichnen sich also durch eine große Anzahl von Nachfolgern aus. Legt man nun für Knoten einer bestimmten Stufe eine Schwelle für die Anzahl der Nachfolgeknoten fest, so kann der Baum in Knoten, die texturierte und untexturierte Bildbereiche beschreiben, eingeteilt werden. Die so erkannten inhomogenen Knoten der festgelegten Stufe werden nun einer weiteren Texturanalyse in Richtung der gesuchten Klasse unterzogen, z.B. mit Hilfe der in Abschnitt 16.11 erläuterten Co-occurrence-Matrizen.

Inhomogene Knoten können auch von linienhaft ausgeprägten Objekten (z.B. Straßen) erzeugt worden sein. Zur Segmentierung derartiger Objekte ist eine Linienverfolgung (Kapitel 20) in der Baumstruktur möglich.

Kapitel 31

Segmentierung und numerische Klassifikation

31.1 Grundlegende Problemstellung

Die grundlegende Problemstellung bei der Segmentierung sei anhand eines Beispiels erläutert. Es wird eine zweikanalige Szene $\mathbf{S}_e = (s_e(x, y, n))$ angenommen, bei der $n = 0$ einen Rotauszug und $n = 1$ einen Blauauszug darstellt. In diesem Bild seien drei verschiedene Objekte abgebildet, die sich in den beiden Farbauszügen deutlich unterscheiden (etwa zwei verschiedenfarbige Oberflächen von Bauteilen vor einem dunklen Hintergrund). Es wird nun das zweidimensionale Histogramm (Abschnitt 16.11) dieses Bildes ermittelt und grafisch dargestellt. Auf den Koordinatenachsen werden dazu die Merkmale „Rotanteil" (Kanal 0) und „Blauanteil" (Kanal 1) aufgetragen. Gemäß der gewählten Definition sind die diskreten Werte für die Farbanteile aus der Grauwertmenge $G = \{0, 1, ..., 255\}$. Das bedeutet, dass im zweidimensionalen Histogramm alle Wertekombinationen $(i, k), i, k \in G$, mit bestimmten (relativen) Häufigkeiten auftreten können. Man sagt: Der *Merkmalsraum* ist zweidimensional. In der grafischen Darstellung des Histogramms von $\mathbf{S}_e$ werden sich zu den einzelnen Objekten im Merkmalsraum *Punktwolken* (*cluster*) ausbilden, da Bildpunkte von $\mathbf{S}_e$, die zum selben Objekt gehören, ähnliche Farbkombinationen aufweisen (Bild 31.1).

Aus diesem Beispiel ergeben sich drei wichtige Fragestellungen, die kennzeichnend für die numerische Klassifikation sind:

- Wie können die Punktwolken der abgebildeten Objekte ermittelt werden?
- Wie können die Punktwolken mit mathematischen Modellen beschrieben werden?
- Wie kann ein Bildpunkt, dessen Zugehörigkeit zu einem der abgebildeten Objekte nicht bekannt ist, mit einem Klassifizierungsverfahren einer der Klassen zugeordnet werden?

In den folgenden Abschnitten werden diese Fragestellungen näher untersucht. Zunächst aber einige Begriffsklärungen.

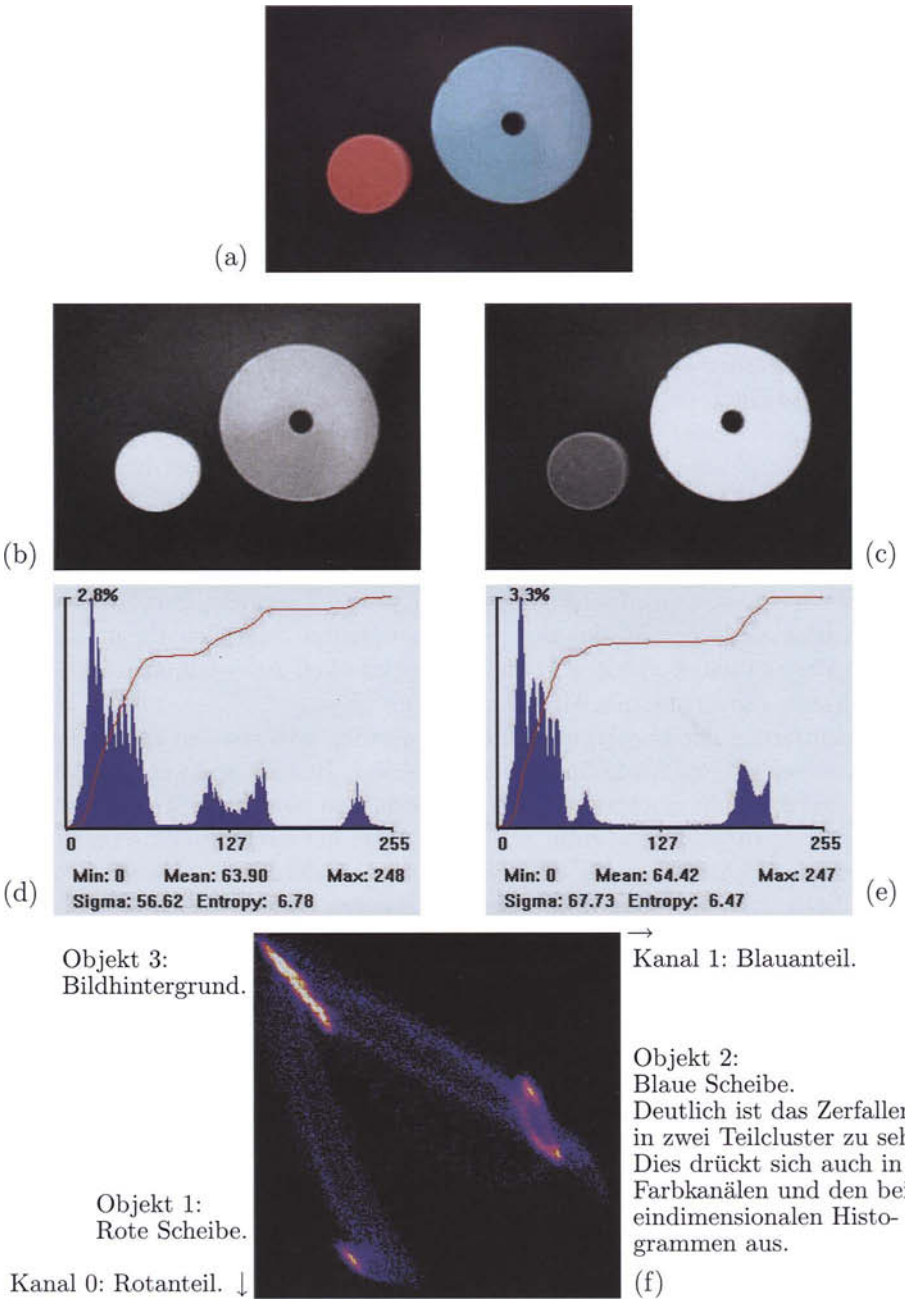


Bild 31.1: Beispiel eines zweidimensionalen Merkmalsraums. (a) Farbbild mit drei Objekten: Rote Scheibe, blaue Scheibe und Hintergrund. (b) Rotanteil (Kanal 0). (c) Blauanteil (Kanal 1). (d) Histogramm des Rotanteils. (e) Histogramm des Blauanteils. (f) Zweidimensionales Histogramm mit drei Clustern.

Da ein Klassifizierungs- oder Mustererkennungssystem in der Regel für einen ganz bestimmten Anwendungsbereich oder *Problemkreis* geschaffen wird, ist es auch nur mit bestimmten Objekten konfrontiert. Manchmal lassen sich die möglichen Objekte aufzählen, z.B. alle Zeichen einer genormten Schrift. In anderen Fällen hat man nur eine intuitive Vorstellung: Wiesen, Wälder, Gewässer, Siedlungen usw. sind z.B. Objekte, die in einer Luftaufnahme abgebildet sein können. Man spricht dann auch von *Objektklassen O*. So ist z.B. ein Bauteil, das in einer Aufnahme abgebildet ist, die von einem sichtgesteuerten Roboter gemacht wird, ein Objekt der Objektklasse „Bauteil vom Typ XYZ". Durch die Verwendung der spezifischen Sensoren des Problemkreises werden bestimmte *Merkmale* der Objekte oder der Objektklassen erfasst. Von optischen Sensoren, die bildliche Informationen liefern, werden von den Objekten Merkmale erfasst, wie der Grauton, die Farbe, die multispektrale Signatur, die Oberflächenstruktur (Textur), die Form oder das zeitliche Verhalten. Der Duft eines Objektes wird von einem optischen Sensor z.B. nicht erfasst.

Als *Aufzeichnung* wird die Abbildung des Beobachtungsgebietes in die digitalisierte Form der Szene $\mathbf{S}_e = (s_e(x, y, n, t))$ bezeichnet. Bei einem dynamischen Vorgang wird die resultierende Szene eine Bildfolge sein. Im Fall einer statischen Graubildaufnahme genügt das Modell $\mathbf{S}_e = (s_e(x, y))$. Nach der Aufzeichnung wird eine Objektklasse $O_i, i = 0, 1, ..., t-1$ durch eine *Musterklasse* (ein *Muster*) $K_i, i = 0, 1, ..., t-1$ repräsentiert. Das Muster kann man sich aus allen möglichen Merkmalen zusammengesetzt vorstellen, die man aus den aufgezeichneten Originaldaten ableiten kann. Bei praktischen Anwendungen ist es allerdings nicht möglich, alle nur denkbaren Merkmale zu berechnen.

Zur Segmentierung der abgebildeten Objekte werden nun aus den aufgezeichneten Originaldaten weitere *beschreibende Merkmale* extrahiert. Besteht zu zwei Objekten nicht die Möglichkeit, aus den aufgezeichneten Daten Merkmale zu berechnen, anhand derer die Segmentierung durchgeführt werden kann, so ist die Trennung dieser Objekte nicht möglich. In diesem Fall muss die Aufzeichnung des Beobachtungsgebietes mit anderen Sensoren (z.B. mit anderen Spektralausschnitten) durchgeführt werden.

Die Merkmale, anhand derer ein Muster beschrieben wird, können recht unterschiedlich sein. Angenommen, die beiden Bauteile des einführenden Beispiels von Bild 31.1 seien durch folgende Merkmale zu unterscheiden: unterschiedliche Farbanteile im Rot- und im Blauauszug, verschiedene Oberflächenstruktur (z.B. glatt und mattiert) und verschiedene Form (z.B. kreisförmig und rechteckig). Die Farbunterschiede, die sich bereits in den aufgezeichneten Originaldaten ausdrücken, sind rein *bildpunktbezogen*. Sie würden hier unter Umständen bereits zur Segmentierung ausreichen.

Man könnte aber auch noch die unterschiedliche Oberflächenstruktur als beschreibendes Merkmal verwenden. Dazu muss zu einer bestimmten Umgebung jedes Bildpunktes eine Analyse der Oberflächenstruktur durchgeführt werden, die dann für diesen Bildpunkt und seine Umgebung eine oder mehrere Maßzahlen zur Oberflächenbeschreibung liefert. Beschreibende Merkmale dieser Art sind also *umgebungsabhängig*.

Auch die *Form* kann als beschreibendes Merkmal verwendet werden. Hier wird dann die flächige oder räumliche Anordnung der Bildpunkte eines abgebildeten Objektes verwendet. Allerdings werden kompliziertere Merkmale dieser Art erst in einer weiteren Klassifizierungs- oder Interpretationsstufe eingesetzt werden können, da ja zunächst Bildpunkte gefunden

werden müssen, die zu den Objekten gehören, und erst dann ihre Anordunung untersucht werden kann. Für die Darstellung der numerischen Klassifikation sei zunächst auf Merkmale dieser Art verzichtet, obwohl zu bemerken ist, dass die Segmentierung auch hier letztlich wieder auf ein Klassifizierungsproblem hinausläuft.

In der weiteren Betrachtung wird eine N-kanalige Szene angenommen, deren Kanäle die verschiedenen beschreibenden Merkmale (originale oder auch abgeleitete) repräsentieren. Die N Kanäle spannen einen *N-dimensionalen Merkmalsraum* auf.

Ein Bildpunkt wird daher im Weiteren auch als ein *N-dimensionaler Merkmalsvektor* $\mathbf{s}(x, y)$ bezeichnet. Wenn keine bildliche Darstellung der einzelnen Merkmalskanäle gewünscht ist, wird man hier die Forderung nach diskreten Grauwerten fallen lassen. Die Maßzahlen für die einzelnen beschreibenden Merkmale sind dann rational, reell oder komplex.

Da bei den folgenden Betrachtungen häufig nicht die Lage der Bildpunkte $\mathbf{s}(x, y)$ im Ortsbereich der (x, y)-Koordinaten, sondern nur die Lage der Bildpunkte im Merkmalsraum von Bedeutung ist, wird eine Szene $\mathbf{S}$ oft auch als eine Menge S von *Merkmalsvektoren*

$$S = \{\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{M-1}\}, \tag{31.1}$$

geschrieben, wobei M die Anzahl der Merkmalsvektoren ist. Treten in S identische Merkmalsvektoren auf, so sollen sie alle dasselbe Element der Menge bedeuten. Hier kann eventuell aber auch die Häufigkeit identischer Elemente mit berücksichtigt werden.

Zusammenfassend kann man also sagen: Eine Objektklasse wird vermöge der Aufzeichnung durch eine bestimmte Musterklasse repräsentiert, die anhand der gewählten beschreibenden Merkmale charakterisiert wird. Die Merkmale erzeugen im Merkmalsraum eine für die jeweilige Musterklasse *charakteristische Population*, die sich, bei geschickter Wahl der Merkmale, als zusammenhängende *Punktwolke* (*cluster*) ausprägt. Welche Punktwolke einem bestimmten Objekt zugeordnet ist und welche Bereiche des N-dimensionalen Merkmalsraumes das Muster einnimmt, ist nicht bekannt. Die wesentliche Aufgabe einer Segmentierung ist es somit, den N-dimensionalen Merkmalsraum durch geeignete Trennungsgrenzen in Bereiche mit ähnlichen Merkmalsvektoren aufzuteilen.

Um Aufschluß über die Population der Objekte im Merkmalsraum zu erhalten, sind unterschiedliche Vorgehensweisen möglich, die als

- *fest dimensionierte überwachte*,
- *fest dimensionierte unüberwachte*,
- *überwacht lernende* oder
- *unüberwacht lernende*

Klassifizierungsstrategien bezeichnet werden.

31.2 Fest dimensionierte überwachte Klassifizierungsstrategien

Bei vielen Problemstellungen besteht die Möglichkeit, die Population der Muster im Merkmalsraum (abkürzend dafür wird im Folgenden oft auch nur von „Muster" gesprochen) durch eine Stichprobe mit bekannten Objekten zu ermitteln. Soll z.B. im Rahmen der Auswertung von Blutbildern die Zählung der unterschiedlichen Zelltypen automatisch durchgeführt werden, so besteht die Möglichkeit, die Merkmale anhand ausgesuchter Testbeispiele zu ermitteln. Werden Bilddaten aus der Fernerkundung verarbeitet, so werden die einzelnen Objektklassen anhand von Bildausschnitten (*Trainingsgebieten*) festgelegt.

Die Stichproben sollten dabei natürlich die Eigenschaften des zugehörigen Musters möglichst vollständig wiedergeben. Allerdings kann durch die Stichprobe die tatsächliche Lage der Muster im Merkmalsraum nur näherungsweise ermittelt werden. Die durch die Stichprobe erfassten Eigenschaften der Objekte werden im Merkmalsraum ebenfalls durch Punktwolken repräsentiert, die im Folgenden als *Realisationen* $k_i, i = 0, 1, 2...$ der Musterklassen bezeichnet werden.

Die Stichprobe ergibt zu jeder Klasse K_i eine *Realisation* k_i. Man kann sich das wie eine statistische Zufallsvariable X vorstellen, die bei der Durchführung eines Zufallsversuchs einen Wert x als Realisation annimmt. Wird eine andere Stichprobe verwendet, so ergibt sich auch eine andere Realisation k_i. Die k_i sollten die tatsächlichen K_i im Merkmalsraum so gut wie möglich annähern.

Es stellt sich die Frage, wie eine Realisation mathematisch beschrieben werden kann. Zunächst ist sie eine Menge von Merkmalsvektoren. Da zu jedem Merkmalsvektor noch wichtig ist, wie häufig er in der Realisation der Musterklasse auftritt, wird folgendes Modell gewählt:

$$k_i = \left\{ \left(\mathbf{g}_0^{(i)}, h_0^{(i)} \right), \left(\mathbf{g}_1^{(i)}, h_1^{(i)} \right), ..., \left(\mathbf{g}_{u_i-1}^{(i)}, h_{u_i-1}^{(i)} \right) \right\}, \tag{31.2}$$

wobei

k_i Realisation der Musterklasse,

$\mathbf{g}_j^{(i)}$ N-dimensionaler Merkmalsvektor,

$h_j^{(i)}$ Häufigkeit des Merkmalsvektors in der Realisation der Musterklasse,

u_i Anzahl der verschiedenen Merkmalsvektoren in der Realisation der Musterklasse.

Der Idealfall wäre nun, wenn die durch die Stichprobe gewonnenen Realisationen der Musterklassen mit der tatsächlichen Population deckungsgleich wären. Das wird in der Praxis aber nicht zu erreichen sein. Vielmehr werden sie nur einen gemeinsamen Durchschnitt haben. Bild 31.2 soll diesen Sachverhalt erläutern.

Werden durch die Stichprobe Eigenschaften erfasst, die die zugehörige Musterklasse nicht aufweist, so werden von den anschließenden Klassifizierungsverfahren Merkmalsvek-

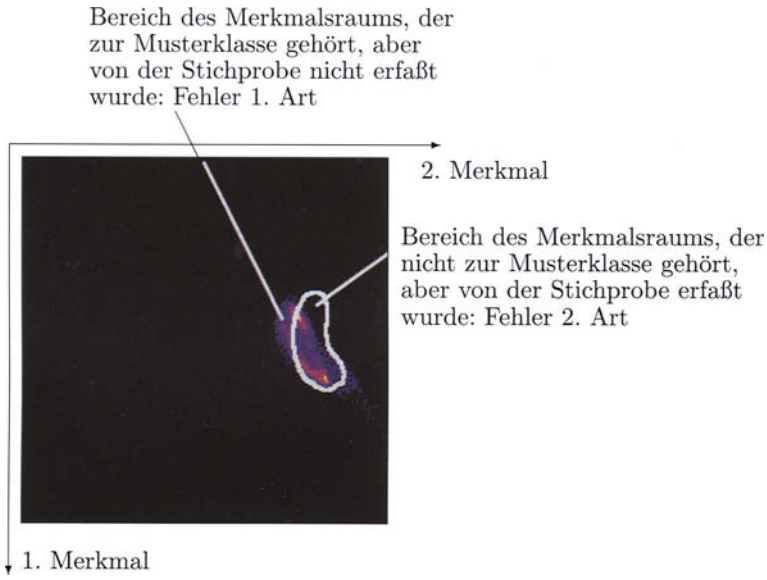


Bild 31.2: Fehler 1. und 2. Art bei der Approximation eines Musters mit Hilfe einer Stichprobe. In Farbe: Punktwolke der Musterklasse, deren Ausprägung in der Praxis aber in der Regel nicht bekannt ist. Der weiße Rand stellt eine Realisation einer Musterklasse nach Maßgabe eine Stichprobe dar.

toren, die eigentlich nicht zum Objekt gehören, fälschlich diesem zugeordnet (Fehler 2. Art). Der Fehler 1. Art tritt auf, wenn die Stichprobe nicht alle Eigenschaften der Klasse beinhaltet, so daß Bildpunkte, die eigentlich dazugehören würden, nicht oder falsch klassifiziert werden.

Die Realisationen der Musterklassen werden jetzt als Näherung der tatsächlichen Population der Musterklassen verwendet. Ein Merkmalsvektor, dessen Zugehörigkeit zu einem bestimmten Objekt nicht bekannt ist, wird demjenigen zugeordnet, zu dessen Realisation der Musterklasse er „am besten passt". Was bei den einzelnen Klassifikatoren unter „am besten passt" zu verstehen ist, wird bei der Darstellung der jeweiligen Verfahren in den folgenden Abschnitten erläutert. Diese Verfahren lassen sich, grob gesprochen, in zwei Bereiche aufteilen.

Bei der *statistischen Vorgehensweise* werden die Musterklassen im Merkmalsraum mit Hilfe von *Verteilungsfunktionen* oder *Verteilungsdichten* erfasst. Die Zuweisung eines unbekannten Bildpunktes erfolgt hier nach Gesichtspunkten der maximalen Wahrscheinlichkeit.

Bei der *geometrischen Vorgehensweise* werden *Trennungsfunktionen* zwischen den ein-

zelnen Musterklassen berechnet. Die Zuweisung der Bildpunkte erfolgt hier nach Maßgabe dieser Trennungsfunktionen.

Ein Klassifizierungssystem, das auf der Basis von bekannten Stichproben für die Realisationen der Musterklassen ermittelt, heißt ein fest *dimensioniertes überwachtes Klassifizierungssystem.* Zusammenstellend ist dabei Folgendes gegeben:

- $\mathbf{S} = (s(x, y, n))$ eine N-kanalige Szene, die einen N-dimensionalen Merkmalsraum aufspannt.
- t Objektklassen, denen bestimmte Musterklassen zugeordnet sind.
- t Realisationen $k_0, k_1, ..., k_{t-1}$, die das Ergebnis der Auswertung der Stichprobe sind.
- Zielsetzung ist die Zuweisung eines Merkmalsvektors $\mathbf{g} = \mathbf{s}(x, y)$, über dessen Zugehörigkeit zu einer der Objektklassen nichts bekannt ist, zu einer der Musterklassen. Algorithmen, die diese Zuordnung durchführen, heißen *Klassifikatoren.*

31.3 Fest dimensionierte unüberwachte Klassifizierungsstrategien

Bei einer fest dimensionierten überwachten Vorgehensweise werden die Realisationen der Musterklassen anhand von Stichproben ermittelt. Das heißt aber, dass damit die Anzahl der Klassen vorweg bekannt sein muss. Manchmal weiß man nicht, wieviele verschiedene Objekte abgebildet sind und wieviele Cluster sich demnach im Merkmalsraum ausprägen. Mit einer *unüberwachten Klassifizierungsstrategie* wird dann versucht, die Gruppierungstendenzen der Objekte im Merkmalsraum, also die Realisationen der Musterklassen, ohne bekannte Stichprobe zu ermitteln.

Hierzu werden Verfahren angewendet, die, ausgehend von einem beliebigen Merkmalsvektor als Zentrum einer ersten Punktwolke, versuchen, die Merkmalsvektoren dieser Musterklasse mit Hilfe geeigneter Kriterien zuzuordnen. Ist für einen Merkmalsvektor das Zuordnungskriterium nicht erfüllt, so wird eine neue Punktwolke mit einem neuen Zentrum eröffnet. Die Verfahren laufen (oft iterativ) so lange, bis alle Merkmalsvektoren einer Musterklasse zugeordnet sind. Abschließend werden Cluster, die auf Grund geeigneter Bewertungskriterien als „zu groß" oder „zu klein" erkannt wurden, aufgeteilt oder zusammengelegt.

Eine derartige *Clusteranalyse* liefert schließlich das Ergebnis, dass die Merkmalsvektoren dazu tendieren, t Realisationen $k_0, k_1, ..., k_{t-1}$ von Musterklassen mit den Zentren $\mathbf{z}_0, \mathbf{z}_1, ..., \mathbf{z}_{t-1}$ zu bilden. Da alle Bildpunkte einer der Musterklassen zugewiesen werden, kann das Eingabebild $\mathbf{S}_e$ als klassifiziert angesehen werden. Es besteht aber auch die Möglichkeit, ein derartiges Clusterverfahren einzusetzen, um sich einen ersten Überblick über die Aufteilung des Merkmalsraumes zu verschaffen und mit diesen Vorkenntnissen z.B. einen fest dimensionierten Klassifikator einzusetzen.

Diese Vorgehensweise wird am Beispiel des *unüberwachten Minimum-Distance-Cluster-Algorithmus* erläutert. Es sei

$$S_e = \left\{\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{M-1}\right\} \tag{31.3}$$

die Menge der Bildpunkte (Merkmalsvektoren) einer N-kanaligen Szene $\mathbf{S}_e$. Da die Lage der Bildpunkte im Ortsbereich hier nicht von Interesse ist, kann, wie schon in Abschnitt 31.1 erläutert, auf die Ortskoordinaten (x, y) bei der Darstellung des Verfahrens verzichtet werden. Der Wert $c > 0$ sei ein vorgegebener Schwellwert für die Distanz zweier Merkmalsvektoren im Merkmalsraum. Zu Beginn wird ein beliebiger Merkmalsvektor als Zentrum eines ersten Clusters (der ersten Musterklasse) definiert. Ohne Beschränkung der Allgemeinheit wird

$$\mathbf{z}_0 = \mathbf{g}_0 \tag{31.4}$$

angenommen. Für die folgenden Bildpunkte $\mathbf{g}_i, i = 1, 2, ...$ wird jetzt die Distanz

$$d_i^{(0)} = d(\mathbf{z}_0, \mathbf{g}_i) \tag{31.5}$$

berechnet. Als Distanzfunktion kann etwa die Euklidische Distanz im Merkmalsraum verwendet werden. Falls nun bis zu einem bestimmten Wert i gilt:

$$d_i^{(0)} \leq c, \quad i = 1, 2, ..., i_1 - 1, \tag{31.6}$$

so werden die dazugehörigen Bildpunkte der Musterklasse K_0 zugeordnet. Für den Bildpunkt $\mathbf{g}_{i_1}$ sei die Bedingung (31.6) nicht mehr erfüllt. Es wird dann

$$\mathbf{z}_1 = \mathbf{g}_{i_1} \tag{31.7}$$

als Zentrum eines zweiten Clusters festgelegt. Für die folgenden Bildpunkte

$$\mathbf{g}_{i_1}, \mathbf{g}_{i_1+1}, \cdots$$

werden jetzt die Distanzen

$$d_k^{(0)} \text{ und } d_k^{(1)}, \quad k = i_1, i_1 + 1, ... \tag{31.8}$$

berechnet. Die Bildpunkte werden den entsprechenden Musterklassen zugeordnet, falls die Schwellwertüberprüfung dies zulässt. Gilt für einen Bildpunkt mit dem Index j

$$d_j^{(0)} \leq c \text{ und } d_j^{(1)} \leq c, \tag{31.9}$$

so wird er demjenigen Cluster zugewiesen, zu dem er die kleinere Distanz hat. Für einen Index i wird der Fall eintreten, dass keine der beiden Distanzen kleiner oder gleich c ist. In diesem Fall wird ein weiteres Clusterzentrum festgelegt. Aus dieser Darstellung ist leicht

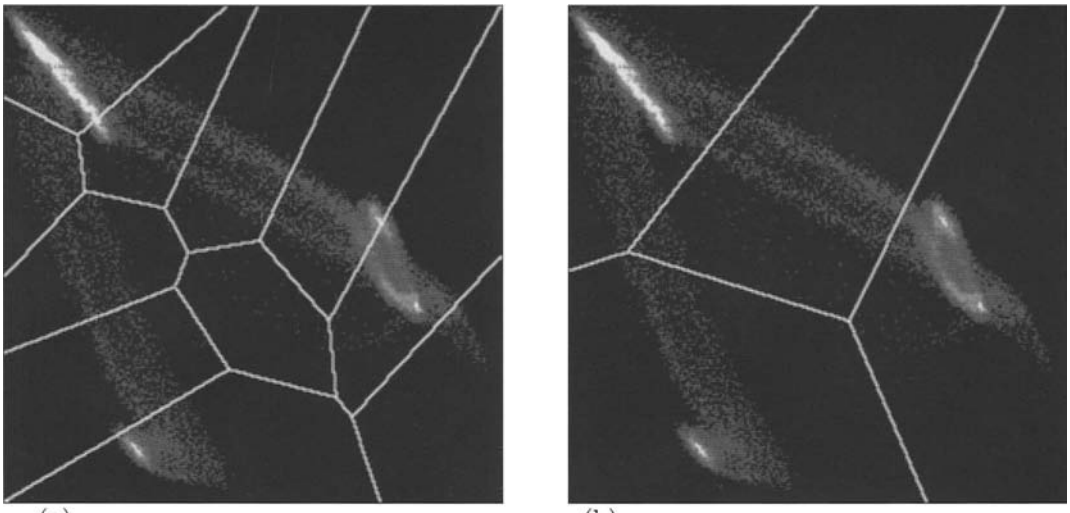


Bild 31.3: Die Wahl des Schwellwertes c beeinflusst das Ergebnis des Minimum-Distance-Clustering-Algorithmus wesentlich. (a) Bei kleinerer Wahl von c (hier: c=50) wird der Merkmalsraum feiner aufgeteilt als bei (b) größerer Wahl von c (hier: c=100).

zu ersehen, wie der Minimum-Distance-Cluster-Algorithmus abläuft, bis alle Merkmalsvektoren verarbeitet sind.

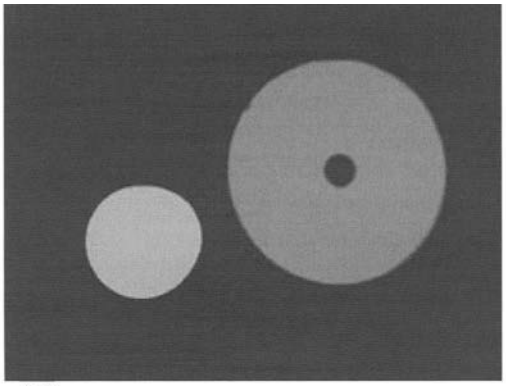
Die praktische Verwendung dieses Verfahrens ist allerdings nicht unproblematisch. Der verwendete Schwellwert c beeinflusst die Güte des Ergebnisses wesentlich, wie Bild 31.3 verdeutlicht.

Zum anderen hängt das Ergebnis von der Verarbeitungsreihenfolge der Bildpunkte ab, da z.B. die Möglichkeit besteht, dass Bildpunkte, die der ersten Musterklasse zugewiesen werden, nach Kreation des zweiten Clusters diesem zugeordnet würden.

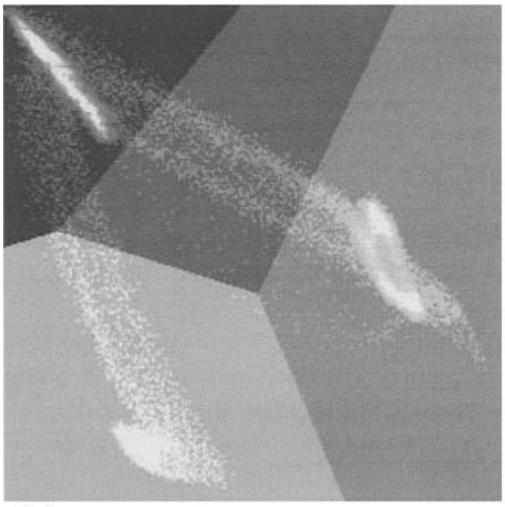
Als Vorteile sind zu nennen, dass das Verfahren einfach zu implementieren ist und meistens wenig Rechenzeit benötigt, da es nicht iterativ arbeitet.

Bild 31.4-a zeigt ein Klassifizierungsergebnis zu diesem Verfahren. Bild 31.4-b stellt die Aufteilung des zweidimensionalen Merkmalsraums dar. Die Lage der zu apprximierenden Cluster wurde eingeblendet. Dieses Verfahren kann auch noch erweitert werden, wenn eine *Zurückweisungsklasse* eingeführt wird. Dieser Klasse werden alle Merkmalsvektoren zugewiesen, die weiter als ein vorzugebender Schwellwert von den generierten Clusterzentren entfernt liegen (Bild 31.4-c).

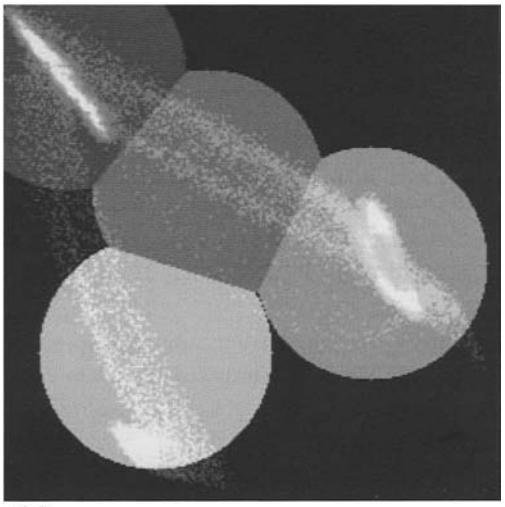
Es gibt eine Reihe von Clusterverfahren, die gegenüber dem dargestellten wesentliche Vorteile haben. Ein Beispiel eines Clusterverfahrens auf der Basis der *fuzzy logic* wird in Kapitel 33 gegeben.



(a)



(b)



(c)

Bild 31.4: Ergebnis des Clusterverfahrens angewendet auf den Rot- und den Blaukanal des Testbildes 31.1-a. (a) Ergebnis der Klassifizierung. Die drei Klassen „rote Scheibe", „blaue Schweibe" und „dunkler Hintergrund" konnten gut klassifiziert werden. Der Merkmalsraum wurde jedoch in vier Bereiche aufgeteilt. In den vierten Bereich wurden z.B. die Ränder um die beiden Ringe klassifiziert. (b) Aufteilung des Merkmalsraums durch das Clusterverfahren bei einem Schwellwert $c = 90$. Die zu approximierenden Cluster wurden eingeblendet. (c) Hier wurde zusätzlich eine Zurückweisungsklasse eingefügt, der alle Merkmalsvektoren zugewiesen werden, die zu weit von den generierten Clusterzentren entfernt liegen (Schwellwert 60).

31.4 Überwachtes und unüberwachtes Lernen

Bei vielen Problemstellungen sind die Eigenschaften der Objekte, die mit Hilfe der numerischen Klassifikation segmentiert werden sollen, gewissen Änderungen unterworfen.

Ein Beispiel: Im Rahmen einer automatischen Qualitätskontrolle muss geprüft werden, ob verschiedenfarbige Flachbandkabel in einer bestimmten Reihenfolge an einen Bauteil angeschlossen sind. Dazu werden die Kabel mit einer Videokamera oder mit einem CCD-Sensor aufgezeichnet und anhand ihrer Farbgebung segmentiert. Nach der Segmentierung kann geprüft werden, ob die richtige Reihenfolge vorliegt. Die Merkmale zur Segmentierung sind hier Farbkombinationen der einzelnen Flachbandkabel, die mit Hilfe einer anfänglichen Stichprobe ermittelt werden können. Es ist nun durchaus denkbar, dass die Farbtöne geringfügigen Nuanzierungen, bedingt durch den Herstellungsprozess, unterworfen sind. Bei einem festdimensionierten Klassifizierungssystem würden diese Trends nicht mit berücksichtigt.

Ein *überwacht lernendes Klassifizierungssystem* versucht diesem Sachverhalt wie folgt gerecht zu werden (Bild 31.5): Anhand einer anfänglichen Stichprobe werden die Merkmale der einzelnen Klassen „gelernt". Das Ergebnis besteht wie oben aus den t Realisationen der Musterklassen. Man bezeichnet diese Phase auch als die *Dimensionierung des Klassifizierungssystems*. Nach der Klassifizierung von Merkmalsvektoren werden jetzt aber ihre Merkmale rückgekoppelt und so die zugehörigen Realisationen der Musterklassen dem Trend angepasst.

Unüberwachtes Lernen liegt vor, wenn die Dimensionierung nicht anhand einer bekannten Stichprobe, sondern durch einen unüberwachten Clusteralgorithmus durchgeführt wird.

Damit soll die Untersuchung der ersten Fragestellung, wie die Populationen der Musterklassen werden können, beendet sein. In den folgenden Abschnitten werden nun einige Klassifikatoren erläutert und dabei die zweite und die dritte Fragestellung behandelt.

31.5 Der Minimum-Distance-Klassifikator

Die Voraussetzungen für die numerische Klassifikation werden einführend zu diesem Klassifikator nochmals zusammengestellt:

- $\mathbf{S}_e = (s_e(x, y, n))$ N-kanalige Szene, die einen N-dimensionalen Merkmalsraum aufspannt.

- t Objekte (Objektklassen), die in $\mathbf{S}_e$ abgebildet sind und denen vermöge der gewählten Aufzeichnung die Muster $K_i, i = 0, 1, ..., t-1$ zugeordnet sind.

- t Realisationen k_i der Musterklassen K_i. Sie sind das Ergebnis einer festdimensionierten Vorgehensweise (Stichproben) oder eines unüberwachten Cluster-Verfahrens.

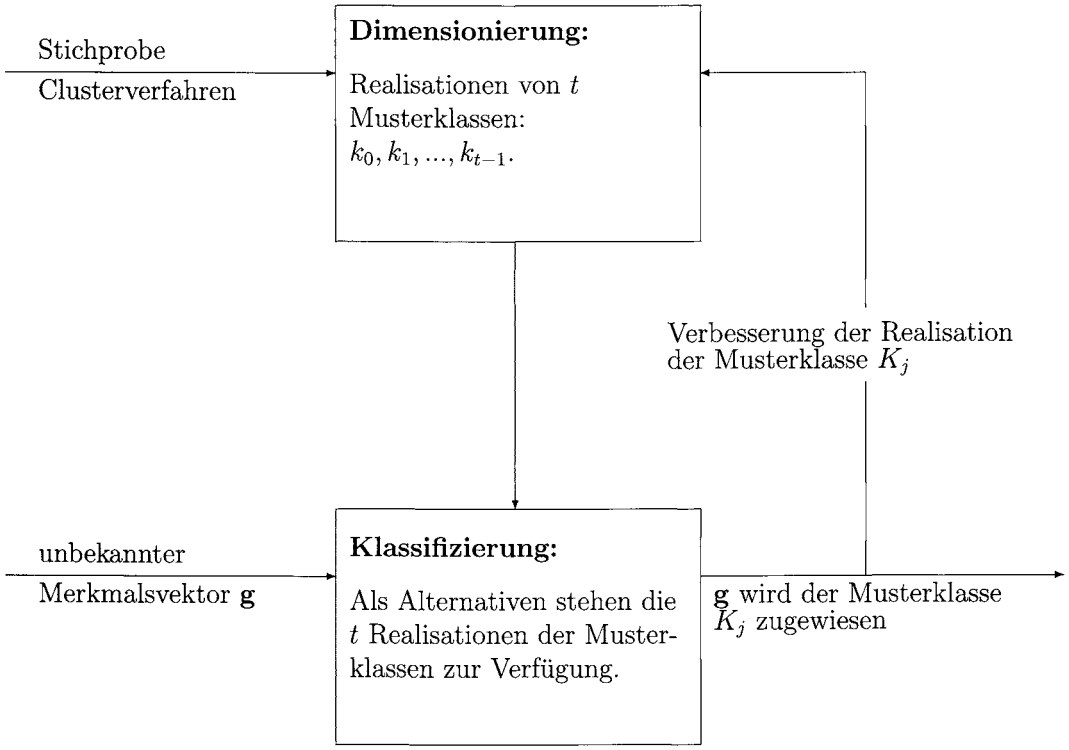


Bild 31.5: Überwacht oder unüberwacht lernendes Klassifizierungssystem. Die Musterklassen werden durch eine Rückkopplung der Merkmale von schon klassifizierten Bildpunkten dem Trend angepasst.

Die Aufgabe besteht nun darin, einen Merkmalsvektor (Bildpunkt) $\mathbf{g} = \mathbf{s}(x, y)$, über dessen Zugehörigkeit zu einer der Musterklassen nichts bekannt ist, mit Hilfe der Realisationen einer Klasse zuzuordnen.

Es sind somit noch die Fragen zu untersuchen, wie die Realisationen der Musterklassen im N-dimensionalen Merkmalsraum mathematisch beschrieben werden können und mit welchen Entscheidungskriterien ein unbekannter Merkmalsvektor klassifiziert werden kann.

Bei einem Klassifikator, der nach dem *Minimum-Distance-Verfahren* arbeitet, ist der Grundgedanke der Klassifizierung, dass Merkmalsvektoren, die im Merkmalsraum nahe beisammen liegen, wahrscheinlich zur selben Musterklasse gehören.

Als Abstandsmaß zweier Punkte $\mathbf{g}_1$ und $\mathbf{g}_2$ im Merkmalsraum wird z.B. die Euklidische Distanz verwendet:

$$d(\mathbf{g}_1, \mathbf{g}_2) = \sqrt{(\mathbf{g}_1 - \mathbf{g}_2)^T(\mathbf{g}_1 - \mathbf{g}_2)} = \sqrt{(g_{1,0} - g_{2,0})^2 + ... + (g_{1,N-1} - g_{2,N-1})^2} \quad (31.10)$$

Aufbauend auf diesem Distanzmaß wird ein Merkmalsvektor $\mathbf{g}$ derjenigen Klasse zugeordnet, zu deren Realisation er den geringsten Abstand hat.

Um den Abstand eines Merkmalsvektors zu einer Realisation berechnen zu können, wird zu jeder Musterklasse K_i ein Zentrum $\mathbf{z}_i, i = 0, 1, ..., t-1$ festgelegt. Als Zentrum können z.B. verwendet werden:

- derjenige Bildpunkt, der in der Realisation der Musterklasse am häufigsten auftritt,
- der Mittelwertvektor der Realisation der Musterklasse oder
- ein Zentrum, das durch einen unüberwachten Clusteralgorithmus ermittelt wurde.

Damit ergibt sich für den Minumum-Distance-Klassifikator folgender Algorithmus:

- Berechnung der t Zentren $\mathbf{z}_i$ zu den Realisationen k_i der Musterklassen.
- Berechnung der Abstände eines zu klassifizierenden Bildpunktes $\mathbf{g}$ zu den Zentren $\mathbf{z}_i$:

$$d_i = d(\mathbf{z}_i, \mathbf{g}), \quad i = 0, 1, ..., t-1. \quad (31.11)$$

- Zuweisung von $\mathbf{g}$ zur Musterklasse K_j, falls gilt: $d_j < d_i$ für alle $i \neq j$.

Im Falle eines zweidimensionalen Merkmalsraumes sind die Trennungsfunktion zweier Musterklassen bei diesem Klassifikator die Mittelsenkrechte zwischen den Zentrumsvektoren. Bei einem mehrdimensionalen Merkmalsraum sind die Trennungsfunktionen sinngemäß (Hyper-)Ebenen.

Die Berechnung des Abstandes eines Bildpunktes von einer Musterklasse nach (31.11) kann noch vereinfacht werden. Statt d_i wird d_i^2 verwendet und wie folgt umgeformt:

$$d_i^2 = (\mathbf{z}_i - \mathbf{g})^T(\mathbf{z}_i - \mathbf{g}) = \mathbf{z}_i^T\mathbf{z}_i - 2\mathbf{z}_i^T\mathbf{g}^T + \mathbf{g}\mathbf{g} = \mathbf{g}^T\mathbf{g} - 2\Big(\mathbf{g}^T\mathbf{z}_i - \frac{1}{2}\mathbf{z}_i^T\mathbf{z}_i\Big). \tag{31.12}$$

Der Term $\mathbf{g}^T\mathbf{g}$ ist für jeden Bildpunkt konstant und muss deshalb für die Berechnung der Entscheidung, welcher Musterklasse $\mathbf{g}$ zugewiesen wird, nicht berechnet werden. Der Term $1/2\mathbf{z}_i^T\mathbf{z}_i$ ist für jede Realisation k_i konstant und kann daher vorab berechnet werden. Der vereinfachte Algorithmus ist damit:

- Bestimmung der Zentren $\mathbf{z}_i$ der Realisationen k_i der Musterklassen K_i.
- Berechnung der *Klassenkonstanten* $c_i = -1/2\mathbf{z}_i^T\mathbf{z}_i$.
- Berechnung der Größen $d_i' = \mathbf{g}^T\mathbf{z}_i + c_i$ für alle Realisationen k_i und für den zu klassifizierenden Merkmalsvektor $\mathbf{g}$.
- Klassifizierung von $\mathbf{g}$ zur Klasse K_j, falls gilt: $d_j' > d_i'$, für alle $i \neq j$.

Nun einige Beispiele. Die Bildfolgen 31.6-a bis 31.6-d und 31.7-a bis 31.7-d zeigen Klassifizierungsergebnisse und die durch die Trainingsgebiete und den gewählten Minimum-Distance-Klassifikator erzeugte Aufteilung des Merkmalsraums, der hier zweidimensional gewählt wurde.

Dieses grundlegende Verfahren lässt eine Reihe von Modifikationen zu. Wie bereits oben erwähnt, können statt der Euklidischen Distanz auch andere Distanzmaße verwendet werden.

Zum anderen treten manchmal Problemstellungen auf, bei denen die Muster der einzelnen Objekte in mehrere Cluster zerfallen. In diesem Fall müssen zu den Musterklassen dementsprechend auch mehrere Zentrumsvektoren berechnet werden. Dieser Sachverhalt kann auch von einem Minimum-Distance-Klassifikator berücksichtigt werden. Man bezeichnet ihn dann als *Nächster-Nachbar-Klassifikator.*

Schließlich ist es in den meisten Anwendungsfällen sinnvoll, eine *Zurückweisungsklasse* einzuführen, in die alle Bildpunkte klassifiziert werden, deren Abstand von der Musterklasse, an der sie am nächsten liegen, größer als ein Schwellwert (*Zurückweisungsradius*) r ist. Unter Verwendung des originalen Abstandes von (31.11) lautet dann die Zuordnungsvorschrift:

- Der Merkmalsvektor $\mathbf{g}$ wird der Musterklasse K_j zugewiesen, falls $d_j < d_i$ für alle $i \neq j$ und falls $d_j < r$ ist. Andernfalls wird er einer Zurückweisungsklasse zugewiesen.

Durch diesen Minimum-Distance-Klassifikator mit *festem Zurückweisungsradius* werden die Musterklassen im N-dimensionalen Merkmalsraum durch N-dimensionale Kugeln angenähert, im Zweidimensionalen also durch Kreise.

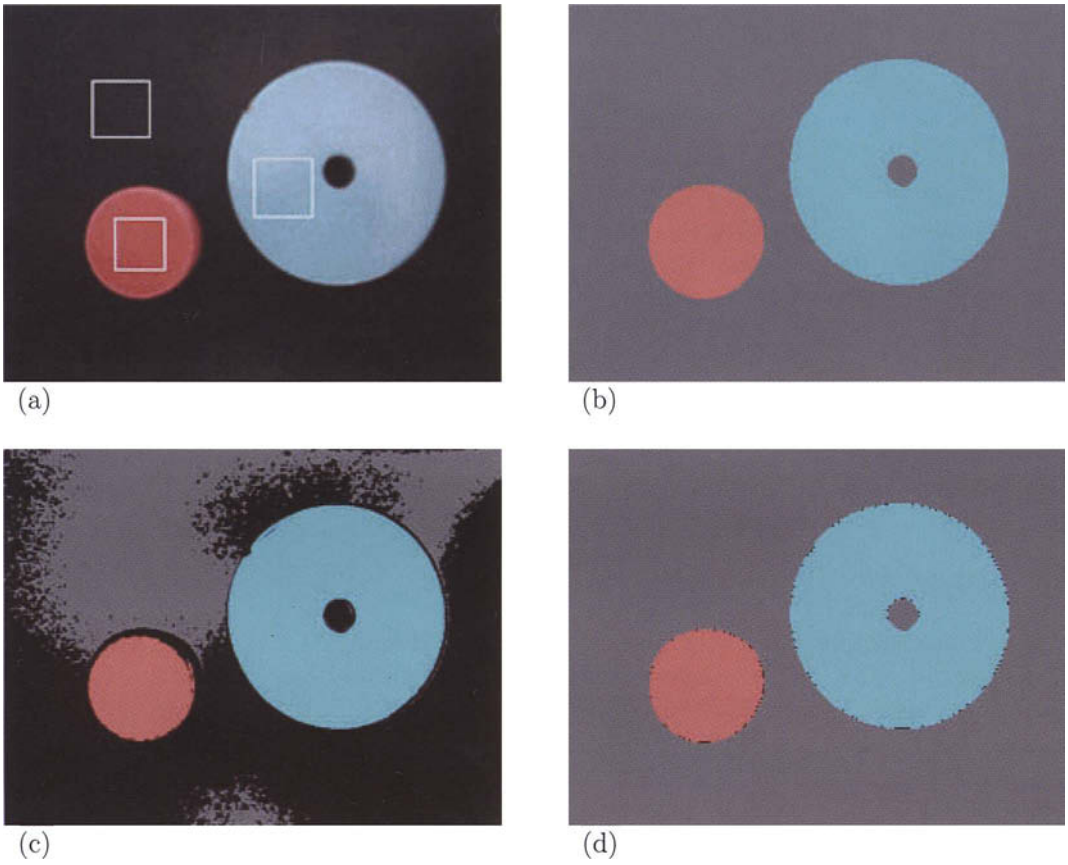


Bild 31.6: (a) Original mit drei Trainingsgebieten zu den drei Klassen „Hintergrund", „rote Bildpunkte" und „blaue Bildpunkte". Um den Minimum-Distance-Klassifikator zweidimensional darstellen zu können, wurden zur Klassifizierung nur der rote und der blaue Kanal verwendet. (b) Ergebnis der Klassifizierung. Die Bildpunkte des klassifizierten Bildes (logisches Bild: die Grauwerte sind Codes für die Zugehörigkeit zu einer Klasse) wurden in der jeweiligen Farbe ihrer Klasse dargestellt. Man sieht, dass die Bildpunkte (Merkmalsvektoren) richtig klassifiziert wurden. Korrespondierend zu diesem Ergebnis ist die Aufteilung des Merkmalsraums gemäß Bild 31.7-b. (c) Hier wurde ein an die Clustergröße angepasster Zurückweisungsradius verwendet. Da das gewählte Trainingsgebiet die Klasse „Hintergrund" nicht ausreichend erfasst (Vergleich: Bild 31.7-a und Bild 31.7-c) werden die Hintergrundbildpunkte zu einem großen Teil als „nicht klassifizierbar" der Zurückweisungsklasse zugewiesen. Die Aufteilung des Merkmalsraums wird hier durch Bild 31.7-c dargestellt. (d) Hier wurde der Zurückweisungsradius vergrößert. Nur mehr in den Übergangsbereichen sind einige Bildpunkte nicht klassifizierbar. Die Aufteilung des Merkmalsraums entspricht Bild 31.7-d.

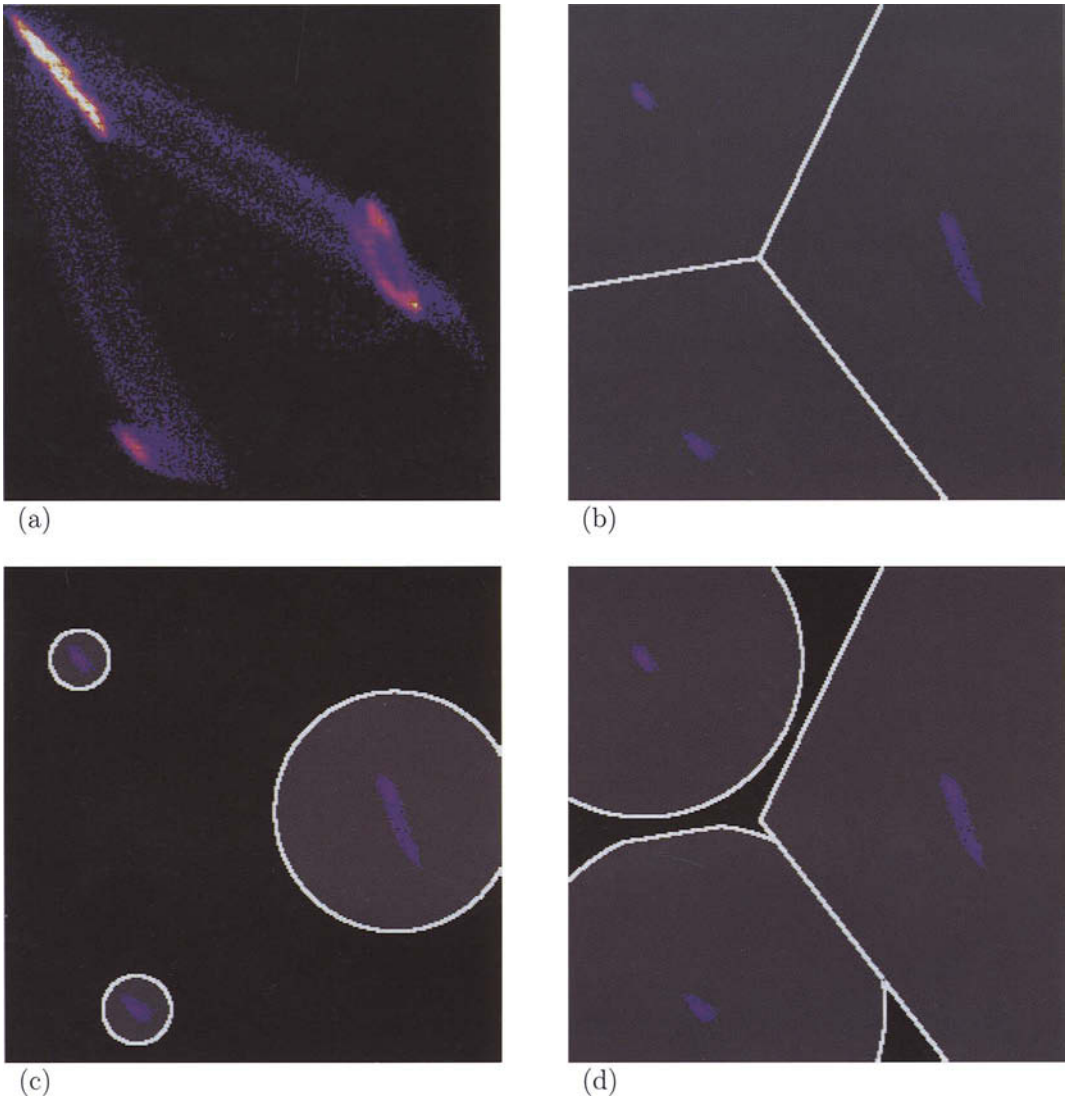


Bild 31.7: (a) Zweidimensionales Histogramm zweidimensionaler Merkmalsvektoren des Bildes 31.6-a unter Verwendung des roten und des blauen Kanals. (b) Durch die in Bild 31.6-a eingezeichneten Trainingsgebiete (Stichproben) werden im Merkmalsraum die dargestellten Realisationen der Klassen erzeugt. Man sieht durch einen Vergleich mit Bild a, dass die Klasse „Hintergrund" nicht ausreichend erfasst wurde. Die weißen Linien zeigen die Aufteilung des Merkmalsraums durch den Minimum-Distance-Klassifikator ohne Zurückweisung. Die Trennungsgrenzen sind hier die Mittelsenkrechten zwischen den Clusterzentren. (c) Verwendung eines an die Clusterzentren angepassten Zurückweisungsradius. Alle Merkmalsvektoren, die außerhalb der Zurückweisungsbereiche liegen, werden als „nicht klassifizierbar" ausgewiesen. (d) Vergrößerung des Zurückweisungsradius.

Eine weitere Möglichkeit besteht darin, für jede Musterklasse einen eigenen, an die Ausdehnung von k_i angepassten Zurückweisungsradius r_i zu verwenden, der z.B. aus der Streuung der Realisation der Musterklasse in den einzelnen Kanälen abgeleitet werden kann. Die Klassifizierungsvorschrift lautet dann:

- Der Merkmalsvektor $\mathbf{g}$ wird K_j zugewiesen, falls $d_j < d_i$ für alle $i \neq j$ und falls $d_j < r_j$ ist. Andernfalls wird er einer Zurückweisungsklasse zugewiesen.

Der Minimum-Distance-Klassifikator wurde bis jetzt als fest dimensionierter Klassifikator beschrieben, da nach der anfänglichen Stichprobe, aus der die Realisation der Musterklassen abgeleitet wurden, diese nicht durch eine Rückkopplung von bereits klassifizierten Bildpunkten verändert wurden. Um einen überwacht lernenden Klassifikator zu erhalten, ist bei der Zuweisung eines neuen Bildpunktes zu einer Musterklasse nur jeweils der Zentrumsvektor und evtl. der Zurückweisungsradius anzupassen. Wird als Zentrum der Mittelwertvektor verwendet, so wird er folgendermaßen modifiziert:

$k_j = \left\{\left(\mathbf{g}_0^{(j)}, h_0^{(j)}\right), ..., \left(\mathbf{g}_{u_j-1}^{(j)}, h_{u_j-1}^{(j)}\right)\right\}$
Realisation der Musterklasse vor der Zuweisung des Merkmalsvektors $\mathbf{g}$.

$$anz_j = \sum_{\mu=0}^{u_j-1} h_\mu^{(j)}$$
Anzahl der Bildpunkte, die k_j bis jetzt zugewiesen wurden.

$$\mathbf{z}_j = \frac{1}{anz_j} \sum_{\mu=0}^{u_j-1} h_\mu^{(j)} \mathbf{g}_\mu^{(j)} \text{ Mittelwertvektor von } k_j. \tag{31.13}$$

Anstatt $\mathbf{z}_j$ wird nach der Zuweisung von $\mathbf{g}$ zu K_j

$$\frac{1}{anz_j - 1}\left(anz_j \cdot \mathbf{z}_j + \mathbf{g}\right)$$

Auch der an die Musterklasse angepasste Zurückweisungsradius r_j kann bei Hinzunahme des Bildpunktes $\mathbf{g}$ zu K_j verbessert werden, wenn analog zu (31.13) die Streuung von k_j modifiziert wird.

Der Algorithmus lautet:

A31.1: Minimum-Distance-Klassifikator.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y,n))$ ein N-kanaliges Bild, dessen Kanäle einen N-dimensionalen Merkmalsraum aufspannen. Als Menge von Merkmalsvektoren geschrieben: $S = \{\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{M-1}\}$.

◇ $\mathbf{S}_a = (s_a(x,y))$ ein einkanaliges Grauwertbild mit der Grauwertmenge $G = \{0,1,...,255\}$ (Ausgabebild). Die Grauwerte sind Codes für die Zugehörigkeit zu einer Klasse.

Algorithmus:

(a) Im Training werden aus den repräsentativen Stichproben, aus denen die Realisationen k_i der Klassen K_i abgeleitet wurden, die Zentrumsvektoren $\mathbf{z}_i$ und die Vektoren der Streuungen $\mathbf{q}_i$ berechnet.

(b) Für alle Klassen K_i werden berechnet:

(ba) Die Klassenkonstanten:
$c_i = -\frac{1}{2}\mathbf{z}_i^T\mathbf{z}_i$.

(bb) Die Zurückweisungsradien (hier angepasst an die Klasse):
$r_i = c \cdot \max_n \{\sqrt{q_{i,n}}\}, n = 0,1,...,N-1$. Falls die einzelnen Merkmalskanäle sehr unterschiedliche Streuungen besitzen, kann eine Normierung der Merkmale vor der Dimensionierung des Systems sinnvoll sein.

(c) Für alle Merkmalsvektoren $\mathbf{g}$ des Bildes:

(ca) Für alle Klassen $K_i, i = 0,1,...,t-1$:

Berechnung der Größen
$d_i' = \mathbf{g}^T\mathbf{z}_i + c_i$.

(cb) Zuweisung von $\mathbf{g}$ zur Klasse K_j, falls gilt:
$d_j' > d_i', \quad i = 0,1,...,t-1, \quad i \neq j$ und
$d_j' > \frac{1}{2}(\mathbf{g}^T\mathbf{g} - r_i^2)$.

Ende des Algorithmus

Nach dieser Beschreibung eines einfachen geometrischen Klassifikators mit seinen verschiedenen Varianten wird im folgenden Kapitel ein statistischer Klassifikator untersucht.

31.6 Maximum-Likelihood-Klassifikator

Dem Maximum-Likelihood-Klassifikator liegt ein statistischer Ansatz zugrunde. Die Musterklassen werden durch N-dimensionale Verteilungs- oder Dichtefunktionen beschrieben. Dazu müssen einige Wahrscheinlichkeiten definiert werden:

$p(K_i)$	apriori-Wahrscheinlichkeit der Musterklasse K_i.	(31.14)
$f(\mathbf{g})$	Verteilungsdichte der Merkmalsvektoren $\mathbf{g}$.	(31.15)
$f(\mathbf{g}\|K_i)$	Verteilungsdichte der Musterklasse K_i.	(31.16)
R_i	Bereich des Merkmalsraumes, in welchem die Merkmalsvektoren $\mathbf{g}$ zur Musterklasse K_i klassifiziert werden.	(31.17)
$\mathbf{L} = (l_{ij})$	Verlustmatrix: Es tritt der Verlust l_{ij} ein, wenn $\mathbf{g}$ aus K_i ist, jedoch zu K_j klassifiziert wird.	(31.18)

Damit berechnet sich die Wahrscheinlichkeit, dass $\mathbf{g}$ zu K_i klassifiziert wird, wenn $\mathbf{g}$ auch tatsächlich zu K_i gehört, wie folgt:

$$p(\mathbf{g} \text{ im Bereich } R_i|K_i) = \int_{R_i} f(\mathbf{g}|K_i)\mathrm{d}\mathbf{g}. \tag{31.19}$$

Die Wahrscheinlichkeit, dass $\mathbf{g}$ fälschlicherweise zu K_j klassifiziert wird, wenn es tatsächlich zu K_i gehört, ist:

$$p(\mathbf{g} \text{ im Bereich } R_j|K_i) = \int_{R_j} f(\mathbf{g}|K_i)\mathrm{d}\mathbf{g}. \tag{31.20}$$

Die Wahrscheinlichkeit, dass nun ein $\mathbf{g}$ aus K_i auftritt und zu K_i klassifiziert wird, ist dann

$$p(K_i) \cdot p(\mathbf{g} \text{ im Bereich } R_i|K_i) \tag{31.21}$$

und die Wahrscheinlichkeit, dass ein $\mathbf{g}$ aus K_i auftritt und falsch zu K_j klassifiziert wird ist

$$p(K_i) \cdot p(\mathbf{g} \text{ im Bereich } R_j|K_i). \tag{31.22}$$

Der zu erwartende Verlust bei der Klassifizierung von $\mathbf{g}$ zu einer der Klassen ist dann gegeben durch:

$$\sum_{i=0}^{t-1} p(K_i) \sum_{\mu=0;\mu\neq i}^{t-1} l_{i\mu} \cdot p(\mathbf{g} \text{ im Bereich } R_\mu|K_i). \tag{31.23}$$

Die Bereiche R_i, $i = 0, 1, ..., t-1$, in die der Merkmalsraum aufgeteilt wird, sind so zu wählen, dass der zu erwartende Verlust minimiert wird. Dies ist der Fall, wenn $\mathbf{g}$ in R_j liegt, sofern gilt ([Ande69], [Niem74]):

$$\sum_{i=0;i\neq j}^{t-1} p(K_i) \cdot l_{ij} \cdot f(\mathbf{g}|K_i) < \sum_{i=0;i\neq \mu}^{t-1} p(K_i) \cdot l_{i\mu} \cdot f(\mathbf{g}|K_i) \qquad (31.24)$$
$$\mu = 0, ..., t-1; \quad \mu \neq j.$$

Ein Klassifikator, der nach dieser Regel entscheidet, heißt *Bayes'scher Klassifikator*. (31.24) ist die allgemeine Form eines Bayes'schen Klassifikators, der optimal ist, wenn alle statistischen Annahmen zutreffen und die Verlustmatrix bekannt ist. Um diese allgemeine Form für die Praxis verwenden zu können, werden eine Reihe von Vereinfachungen gemacht.

Die apriori-Wahrscheinlichkeiten $p(K_i)$ der Musterklassen sind in der Regel nicht bekannt. Eine grobe Näherung erhält man aus einer visuellen Beurteilung der zu klassifizierenden Szene, wenn man die Häufigkeiten der einzelnen Objekte abschätzt. Weiß man z.B. von einem medizinischen Präparat, dass etwa 70% des Bildausschnittes zum Hintergrund gehören, 10% von Zellen des Typs 1 und 20% von Zellen des Typs 2 eingenommen werden, so könnten die zugehörigen apriori-Wahrscheinlichkeiten mit 7/10, 1/10 und 2/10 angegeben werden.

Wenn der Klassifikation ein unüberwachter Cluster-Algorithmus vorausgegangen ist, so kann man die apriori-Wahrscheinlichkeiten aus der Häufigkeit der einzelnen Musterklassen ableiten.

In allen Fällen, in denen sich keine Information über die $p(K_i)$ gewinnen lässt, werden sie als gleichwahrscheinlich angenommen, d.h.:

$$p(K_i) = \frac{1}{t}. \qquad (31.25)$$

Für die Verlustmatrix $\mathbf{L}$ wird vereinfachend angenommen, dass bei richtiger Klassifizierung kein Verlust ($l_{ij} = 0$) und bei falscher Klassifizierung immer derselbe Verlust eintritt ($l_{ij} = 1, i \neq j$). Damit reduziert sich (31.24) zu:

$$f(\mathbf{g}) - p(K_j) \cdot f(\mathbf{g}|K_j) < f(\mathbf{g}) - p(K_\mu) \cdot f(\mathbf{g}|K_\mu), \quad \mu = 0, ..., t-1, \mu \neq j. \qquad (31.26)$$

Da sich für $f(\mathbf{g})$ hier immer derselbe Wert ergibt, wird es weggelassen. Die Größe, die zur Klassifizierungsentscheidung verwendet wird, ist dann:

$$d_i'(\mathbf{g}) = p(K_i) \cdot f(\mathbf{g}|K_i) \qquad (31.27)$$

und $\mathbf{g}$ wird zu K_j klassifiziert, falls gilt:

$$d_j'(\mathbf{g}) > d_\mu'(\mathbf{g}) \text{ für alle } \mu \neq j. \qquad (31.28)$$

Für die Verteilungsdichten $f(\mathbf{g}|K_i)$ wird angenommen, dass sie mit Hilfe von N-dimensionalen Gauß'schen Normalverteilungen angenähert werden können:

$$f(\mathbf{g}|K_i) = \frac{1}{(2\pi)^{\frac{N}{2}}} \cdot \frac{1}{\det(\mathbf{C}_i)^{\frac{1}{2}}} \cdot \exp\Big(-\frac{1}{2}(\mathbf{g}-\mathbf{z}_i)^T \cdot \mathbf{C}_i^{-1} \cdot (\mathbf{g}-\mathbf{z}_i)\Big). \tag{31.29}$$

Dabei ist $\mathbf{C}_i$ die Kovarianzmatrix der Musterklasse K_i und $\mathbf{z}_i$ der Mittelwertvektor. Wegen der Monotonie des Logarithmus ist es möglich, obige Formel zu logarithmieren, so dass sie schließlich folgende Form erhält:

$$d_i(\mathbf{g}) = \ln\big(p(K_i)\big) - \frac{1}{2}\ln\big(\det(\mathbf{C}_i)\big) - \frac{1}{2}(\mathbf{g}-\mathbf{z}_i)^T \cdot \mathbf{C}_i^{-1} \cdot (\mathbf{g}-\mathbf{z}_i)\Big). \tag{31.30}$$

Ein unbekannter Merkmalsvektor $\mathbf{g}$ wird von diesem *Maximum-Likelihood-Klassifikator* der Musterklasse K_j zugewiesen, falls gilt:

$$d_j(\mathbf{g}) > d_i(\mathbf{g}) \text{ für alle } i \neq j. \tag{31.31}$$

Ähnlich wie im Falle des Minimum-Distance-Klassifikators ist es sinnvoll, auch hier eine Zurückweisungklasse einzuführen. Die Größe

$$(\mathbf{g}-\mathbf{z}_i)^T \cdot \mathbf{C}_i^{-1} \cdot (\mathbf{g}-\mathbf{z}_i)\Big). \tag{31.32}$$

heißt *Mahalanobis-Abstand* des Bildpunktes $\mathbf{g}$ von der Musterklasse K_i. In die Zurückweisungsklasse werden alle Bildpunkte eingewiesen, deren Mahalanobis-Abstand größer als eine vorgegebene Schwelle ist.

Der geometrische Ort aller Bildpunkte, die von einer Musterklasse vermöge der Mahalanobis-Distanz denselben Abstand haben, ist ein N-dimensionales Ellipsoid, im Zweidimensionalen also eine Ellipse. So werden von diesem Maximum-Likelihood-Klassifikator mit Zurückweisung die Musterklassen auch wieder durch geometrische Gebilde approximiert. Die Bilder 31.8 zeigen die Aufteilung des Merkmalsraums durch den Maximum-Likelihood-Klassifikator mit verschiedenen Zurückweisungsradien. Als Grundlage wurde die Szene mit den Trainingsgebieten von Bild 31.6-a und und des Merkmalsraums von Bild 31.7-a verwendet.

Die Dimension des Merkmalsraumes, die sich in (31.30) u.a. in der Kovarianzmatrix $\mathbf{C}_i$ der Musterklasse K_i widerspiegelt, geht in die Berechnung quadratisch ein. Dies kann bei höheren Dimensionen zu beachtlichen Rechenzeiten führen. Es gibt verschiedene Ansätze diese Rechenzeiten zu reduzieren. Eine Möglichkeit besteht darin, den Maximum-Likelihood-Algorithmus statt in einer normalen Rechenanlage in eigens für diese Art der Klassifizierung konstruierter Spezialhardware ablaufen zu lassen. Der eigentliche Algorithmus wird dabei nicht geändert.

Bei einer anderen Möglichkeit wird zu jeder Musterklasse ein *Kern* berechnet. Ein Kern ist dabei ein N-dimensionales Ellipsoid, das so gewählt wird, dass, wenn ein Bildpunkt innerhalb des Kernes einer Musterklasse liegt, sofort entschieden werden kann, dass er zu dieser Musterklasse gehört.

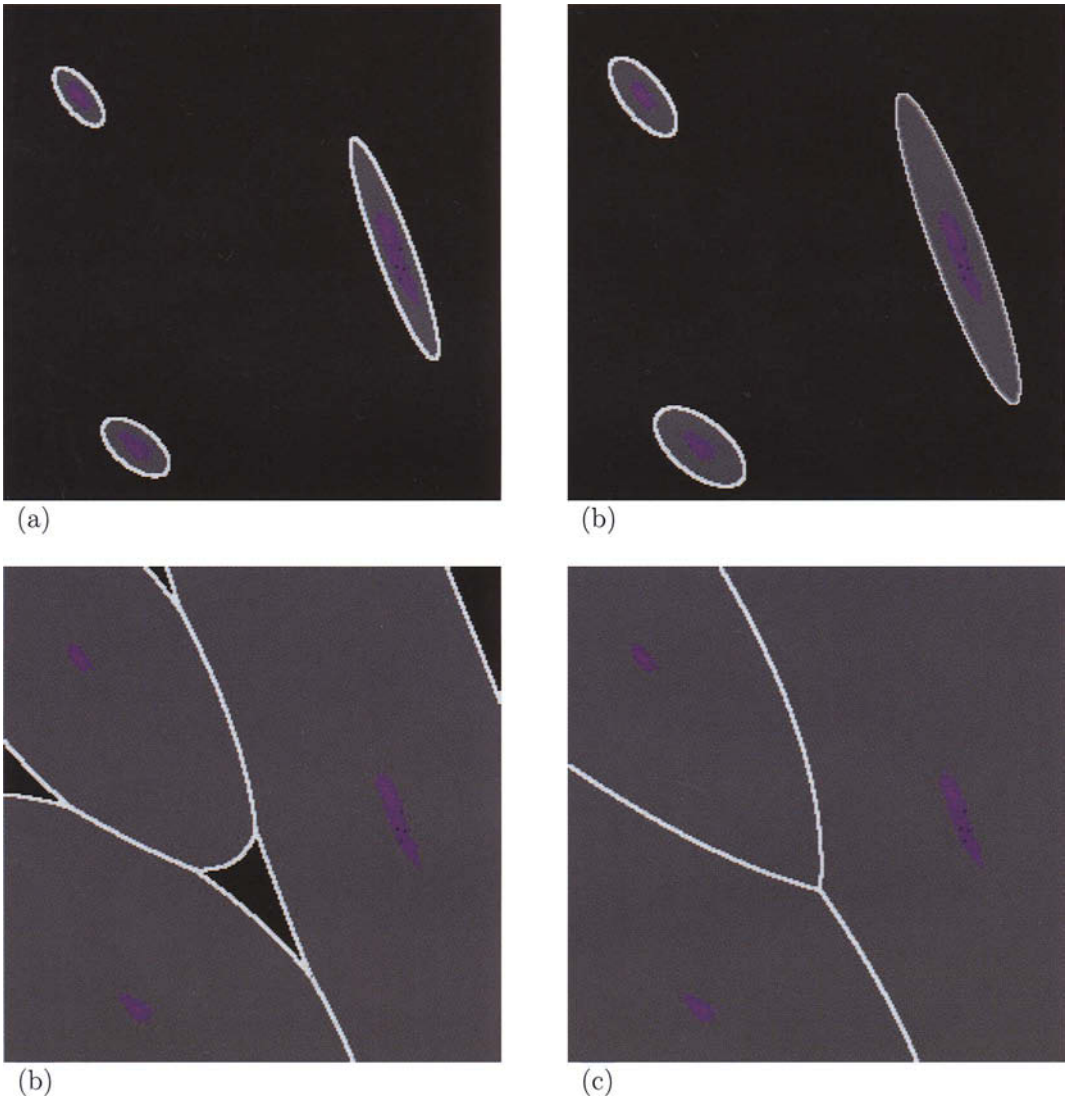


Bild 31.8: (a) - (c) Aufteilung des Merkmalsraums durch den Maximum-Likelihood-Klassifikator zum Klassifizierungstest von Abschnitt 31.5 mit unterschiedlichen Zurückweisungsradien. (d) Aufteilung ohne Zurückweisung.

Der Maximum-Likelihood-Klassifikator kann auch in einem überwacht lernenden Klassifizierungssystem verwendet werden. Dazu wird hier, wie in Abschnitt 31.5 beschrieben, bei der Zuweisung eines Bildpunktes $\mathbf{g}$ zur Musterklasse K_i der Mittelwertvektor $\mathbf{z}_i$ verbessert. Zusätzlich muss dann hier auch noch die Kovarianzmatrix $\mathbf{C}_i$ angepasst werden. Dies sei hier für eine Musterklasse und ein Kanalpaar (n_1, n_2) dargestellt. Der Index für die Musterklasse wird aus Gründen der Übersichtlichkeit weggelassen.

$\mathbf{z}_n^{(anz)}$ sei der Mittelwertvektor der Musterklasse im Kanal n, nach der Zuweisung von anz Merkmalsvektoren.

$\mathbf{z}_n^{(anz+1)}$ der verbesserte Mittelwertvektor nach der Zuweisung des $(anz+1)$-ten Merkmalsvektors $\mathbf{g}$.

Es gilt:

$$\mathbf{z}_n^{(anz+1)} = \frac{1}{anz+1}\Big(anz \cdot \mathbf{z}_n^{(anz)} + \mathbf{g}\Big). \tag{31.33}$$

$v_{n_1,n_2}^{(anz)}$ sei die Kovarianz der Musterklasse in den Kanälen n_1 und n_2 nach der Zuweisung von anz Merkmalsvektoren.

$v_{n_1,n_2}^{(anz+1)}$ sei die Kovarianz der Musterklasse in den Kanälen n_1 und n_2 nach der Zuweisung des (anz+1)-ten Merkmalsvektors $\mathbf{g}$.

Dann berechnet sich die verbesserte Kovarianz wie folgt:

$$v_{n_1,n_2}^{(anz+1)} = \frac{1}{anz+1}\Big(anz \cdot v_{n_1,n_2}^{(anz)} + g_{n_1} g_{n_2} + anz \cdot z_{n_1}^{(anz)} \cdot z_{n_2}^{(anz)}\Big) - z_{n_1}^{(anz)} \cdot z_{n_2}^{(anz)}. \tag{31.34}$$

Werden zu allen Paaren von Kanälen die Kovarianzen mit (31.34) neu berechnet, so wurde damit in der Kovarianzmatrix $\mathbf{C}_j$ die Zuweisung des Bildpunktes $\mathbf{g}$ zur Musterklasse K_j berücksichtigt.

31.7 Der Quader-Klassifikator

Die beiden bis jetzt dargestellten Klassifikatoren haben die Musterklassen durch N-dimensionale Kugeln (Minimum-Distance-Klassifikator, Abschnitt 31.5) oder durch N-dimensionale Ellipsoide (Maximum-Likelihood-Klassifikator, Abschnitt 31.6) approximiert. Es liegt nahe, die Klassifizierung auch noch mit anderen, mathematisch einfach zu handhabenden Gebilden, wie z.B. N-dimensionalen, achsenparallelen Quadern zu versuchen. Wie können diese Quader beschrieben werden? Es sei:

$$k_i = \left\{(\mathbf{g}_0^{(i)}, h_0^{(i)}), (\mathbf{g}_1^{(i)}, h_1^{(i)}), ..., (\mathbf{g}_{u_i-1}^{(i)}, h_{u_i-1}^{(i)})\right\} \tag{31.35}$$

die Realisation der i-ten Musterklasse.

Der Quader Q_i, der der Musterklasse K_i zugeordnet wird, ist ein N-Tupel von Zahlenpaaren, gemäß:

$$Q_i = \left\{(a_0^{(i)}, b_0^{(i)}), (a_1^{(i)}, b_1^{(i)}), ..., (a_{N-1}^{(i)}, b_{N-1}^{(i)})\right\}. \tag{31.36}$$

Dabei sind die Größen $a_n^{(i)}$ und $b_n^{(i)}$ die jeweils linke und rechte Begrenzung des Quaders im Kanal $n = 0, 1, ..., N-1$ (Bild 31.9). Sie können z.B. aus dem Mittelwertvektor $\mathbf{z}_i$ und dem Vektor der Streuungen q_i berechnet werden:

$$a_n^{(i)} = z_{i,n} - c \cdot \sqrt{q_{i,n}}; \quad b_n^{(i)} = z_{i,n} + c \cdot \sqrt{q_{i,n}}. \tag{31.37}$$

Auch hier ist wieder überwachtes Lernen möglich, wenn die Quaderbegrenzungen bei jeder Zuweisung eines neuen Bildpunktes angepasst werden.

Wird zunächst vorausgesetzt, dass die Quader paarweise disjunkt sind, also

$$Q_i \bigcap Q_j = \{\}, \text{für } i \neq j, \tag{31.38}$$

so kann folgende Zuordnungsvorschrift verwendet werden:

Ein unbekannter Merkmalsvektor $\mathbf{g}$ wird der Musterklasse K_j zugewiesen, falls er im Quader Q_j dieser Musterklasse liegt, d.h. falls $\mathbf{g}$ im Intervall $[a_n^{(j)}, b_n^{(j)}], n = 0, 1, ...N-1$ liegt. (31.39)

Dieser Quader-Klassifikator ist sehr einfach zu implementieren und rechenzeitsparend, da die Prüfung, ob ein Bildpunkt in einem Quader liegt, nur aus Vergleichsoperationen besteht. Er ist in solchen Fällen gut geeignet, in denen die verwendeten Merkmale unkorreliert sind, d.h. dass sich die Musterklassen nicht schmal und schräg-langgezogen ausprägen (Bild 31.10). So ist hier ein gewisser Querbezug zur Hauptkomponententransformation (Abschnitt 25.4) gegeben, die ja neue, unkorrelierte Merkmale berechnet.

Der Quader-Klassifikator wäre aber praktisch nicht verwendbar, wenn man die Forderung, dass sich die Quader der einzelnen Musterklassen nicht überdecken dürfen, aufrecht

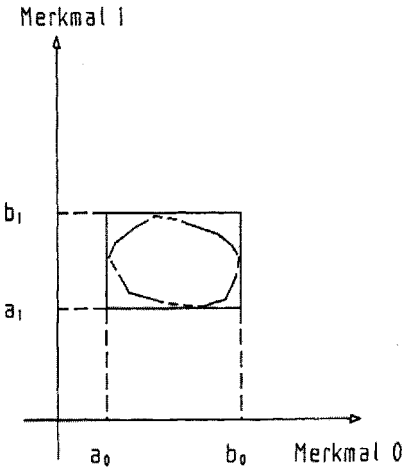


Bild 31.9: Beispiel eines „zweidimensionalen" Quaders (Rechtecks) zur Approximation einer Musterklasse.

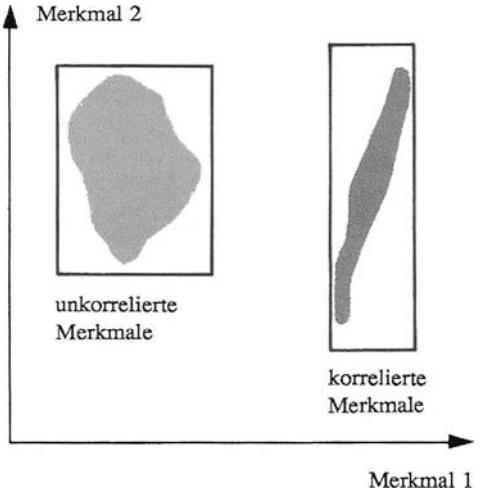


Bild 31.10: Die Approximation der Musterklassen bei der Quadermethode ist in der Regel besser, wenn die verwendeten Merkmale unkorreliert sind.

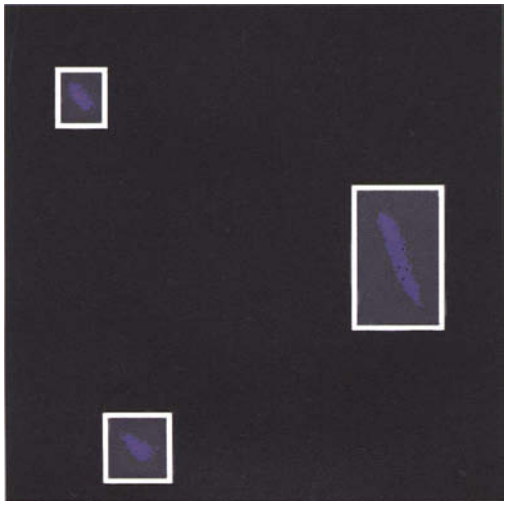


Bild 31.11: Beispiel zum Quader-Klassifikator (zum Testbeispiel in Bild 31.6-a und in Bild 31.7-a: Hier liegen die Quader deutlich voneinander getrennt.

halten würde. Was geschieht aber dann mit den Bildpunkten, die im Überdeckungsbereich von einem oder sogar von mehreren Quadern liegen?

Als eine Möglichkeit kann man diese Bildpunkte einer Sonderklasse „Überdeckungsbereich" zuweisen. Diese Vorgehensweise stellt sich in der Praxis aber auch als unbefriedigend heraus, denn was soll mit den Bildpunkten bei der weiteren Verarbeitung geschehen?

Die beste Möglichkeit ist es, die Bildpunkte in Überdeckungsbereichen mit anderen, etwa den in Abschnitt 31.5 und 31.6 besprochenen Verfahren, zu klassifizieren. Praktische Untersuchungen haben gezeigt, dass in der Regel weniger als 1/3 der Bildpunkte in Überdeckungsbereichen liegen, so dass die Rechenzeitvorteile immer noch gegeben sind. Bild 31.11 ist das Ergebnis des Quader-Klassifikators, angewendet auf das in den Abschnitten 31.5 und 31.6 verwendete Beispiel.

Der Algorithmus zu diesem Klassifikator lautet somit:

A31.2: Quader-Klassifikator.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S}_e = (s_e(x,y,n))$ ein N-kanaliges Bild, dessen Kanäle einen N-dimensionalen Merkmalsraum aufspannen. Als Menge von Merkmalsvektoren geschrieben: $S = \{\mathbf{g}_0, \mathbf{g}_1, ..., \mathbf{g}_{M-1}\}$.

◇ $\mathbf{S}_a = (s_a(x,y))$ ein einkanaliges Grauwertbild mit der Grauwertmenge $G = \{0,1,...,255\}$ (Ausgabebild). Die Grauwerte sind Codes für die Zugehörigkeit zu einer Klasse.

Algorithmus:

(a) Im Training werden aus der repräsentativen Stichprobe, aus denen die Realisationen k_i der Klassen K_i abgeleitet wurden, die Quadergrenzen $(a_n^{(i)}, b_n^{(i)})$ berechnet.

(b) Für alle Merkmalsvektoren $\mathbf{g}$ des Bildes:

(ba) Für alle Klassen $K_i, i = 0,1,...,t-1$: Überprüfen, ob $\mathbf{g}$ in Q_i liegt.

(bb) Falls $\mathbf{g}$ in nur einem Quader Q_j liegt: Zuordnung von $\mathbf{g}$ zur Klasse K_j.

(bc) Falls $\mathbf{g}$ in mehreren Quadern liegt:

(bca) Berechnung der Distanzen von $\mathbf{g}$ zu allen Zentrumsvektoren derjenigen Quader, in denen $\mathbf{g}$ liegt.

(bcb) Zuordnung von $\mathbf{g}$ zu der Klasse K_j, falls er zu dieser Klasse den kürzesten Abstand hat. Bei gleichen Abständen kann er zufällig einer Klasse zugewiesen werden.

(bd) Falls $\mathbf{g}$ in keinem Quader liegt, wird er der Zurückweisungsklasse zugeordnet.

Ende des Algorithmus

31.8 Beurteilung der Ergebnisse

Zur Beurteilung der Klassifizierungsergebnisse bieten sich mehrere Möglichkeiten an. Bei vielen praktischen Anwendungen ist es sinnvoll, vor der eigentlichen Klassifizierung eine apriori-Beurteilung durchzuführen. Dazu werden bei einem festdimensionierten Klassifikator zunächst, wie in den vorhergehenden Abschnitten beschrieben wurde, die benötigten Größen (Mittelwertvektor, Vektor der Streuungen, Kovarianzmatrix, usw.) berechnet. Bevor aber die eigentliche Szene klassifiziert wird, werden jetzt die einzelnen Stichproben klassifiziert. Im Idealfall sollte natürlich eine Stichprobe für ein bestimmtes Objekt auch zu 100% als zu diesem Objekt gehörig klassifiziert werden. Die Abweichungen von diesem Idealwert erlauben es, die zu erwartende Qualität zu beurteilen: Wenn nämlich die Klassifizierung der Stichproben bereits schlechte Ergebnisse liefert, so ist es unwahrscheinlich, dass die Klassifizierung der gesamten Szene besser ist. Diese Beurteilung kann auch automatisiert werden. Man kann etwa die Ergebnisse der Klassifizierung aller Stichproben zu einer Maßzahl zusammenziehen, die, wie die mathematische Wahrscheinlichkeit, Werte zwischen 0 und 1 annimmt und dann die zu erwartende Güte z.B. in Prozent angibt.

Diese Art der apriori-Beurteilung kann noch objektiviert werden, wenn man die Beurteilung nicht anhand der Stichprobe durchführt, aus der die Klassifizierungsparameter

abgeleitet wurden, sondern anhand einer „Beurteilungsstichprobe", die eigens für diesen Zweck festgelegt wurde.

Eine weitere Möglichkeit besteht aus der visuellen und manuellen Interpretation von speziell ausgewählten Beurteilungsgebieten und dem Vergleich mit dem Klassifizierungsergebnis. Hierbei ist allerdings zu bedenken, dass die visuelle und manuelle Interpretation auch Fehler beinhaltet und unter Umständen „etwas Falsches mit etwas Falschem" verglichen wird.

Handelt es sich bei den klassifizierten Szenen um Bilddaten aus der Fernerkundung, so bietet sich eine Möglichkeit an, die relativ objektiv ist. Mit Hilfe eines Zufallszahlengenerators werden Zeilen- und Spaltenkoordinaten im klassifizierten Bild erzeugt und deren Zuordnung zu einem der Objekte notiert. Anschließend werden die entsprechenden Positionen entweder in vergleichbaren Karten oder Luftbildern überprüft oder es wird im Rahmen einer Feldbegehung die tatsächliche Zugehörigkeit zu einer der Klassen bestimmt und mit dem Klassifizierungsergebnis verglichen. Diese Vorgehensweise ist zwar etwas zeitaufwändig, liefert aber ein Ergebnis, das auch nach dem Abschluss noch überprüfbar ist.

31.9 Ergänzungen

In den vorhergehenden Abschnitten wurden einige grundlegende Algorithmen zur Klassifizierung dargestellt. Bei praktischen Anwendungen wird man viele Varianten und Kombinationen mit anderen Lösungsansätzen finden.

So wurden bei den geschilderten Verfahren die Bildpunkte im N-dimensionalen Merkmalsraum parallel klassifiziert, was bedeutet, dass zur Entscheidung alle Merkmalskanäle gleichzeitig herangezogen werden. Bei einem sequentiellen Klassifikator werden nur diejenigen Komponenten $g_0, g_1, ..., g_{N-1}$ eines Bildpunktes $\mathbf{g}$ verwendet, die zu einer Klassifizierung mit ausreichender Sicherheit notwendig sind.

Bei den dargestellten Verfahren wurden überall implizit Trennungsfunktionen berechnet, nach deren Maßgabe die Bildpunkte zugewiesen wurden. Im Falle des Minimum-Distance-Klassifikators waren dies z.B. N-dimensionale Hyperebenen. Die explizite Berechnung der Trennungsfunktionen ist eine weitere Alternative.

Wenn die Dimension des Merkmalsraumes klein ist, etwa N=1, 2 oder 3, so kann die Klassifizierung auch über *table-look-up-Verfahren* durchgeführt werden. Dazu wird z.B. der zweidimensionale Merkmalsraum diskretisiert und durch eine Tabelle von $256 \cdot 256$ Positionen repräsentiert. Für jede Position der Tabelle wird dann angegeben, zu welcher Musterklasse sie gehört. Die dazu notwendigen Informationen können aus der Stichprobe gewonnen werden. Welcher Klassifikator dazu verwendet wird, spielt keine Rolle. Die eigentliche Klassifizierung eines Bildpunktes $\mathbf{g}$ besteht nur mehr darin,in der Tabelle nachzusehen, zu welcher Klasse er gehört. Auch überwachtes Lernen ist möglich, wenn die Tabelle nach jeder Zuweisung modifiziert wird. Dieses Verfahren, das sehr rechenzeitoptimal abläuft, findet aber bei höherdimensionalen Merkmalsräumen schnell seine Grenzen, da bei einem 4-dimensionalen Merkmalsraum die Tabelle bereits $256 \cdot 256 \cdot 256 \cdot 256$ Einträge umfassen müsste.

Kapitel 32

Klassifizierung mit neuronalen Netzen

32.1 Grundlagen: Künstliche neuronale Netze

In diesem Abschnitt wird eine kurze Übersicht zur Thematik *künstliche neuronale Netze* gegeben. Zu ausführlichen Darstellungen sei auf die vielfältige Literatur, z.B. [Koho88], [Wass89] oder [Krat90] verwiesen. Nach der Erläuterung des prinzipiellen Aufbaus neuronaler Netze werden einige Netztypen kurz vorgestellt. In Abschnitt 32.2 wird die Eignung der neuronalen Netze für die digitale Bildverarbeitung und Mustererkennung untersucht.

32.1.1 Prinzipieller Aufbau

Der prinzipielle Aufbau eines künstlichen Neurons ist in Bild 32.1 dargestellt.

Die Größen $x_i, i = 1, ..., m$ sind die Eingabewerte (die *Eingangsaktivierungen*) des Neurons. Sie werden häufig auch als Vektor $\mathbf{x} = (x_1, x_2, ..., x_m)^T$ geschrieben. Jedem Eingang ist ein *(Netz-)Gewicht* $w_i, i = 1, ..., m$ zugeordnet. Die *Netzaktivität net* wird wie folgt berechnet:

$$net = \sum_{i=1}^{m} x_i w_i. \tag{32.1}$$

Nachgeschaltet ist eine *Aktivierungsfunktion* $f(x)$, mit der die Ausgabe (*Ausgangsaktivierung*) *out* berechnet wird:

$$out = f(net). \tag{32.2}$$

Als Aktivierungsfunktion werden je nach Neuronentyp unterschiedliche Funktionen verwendet:

- $f(x)$ ist eine lineare Funktion (möglicherweise auch die Identität). Dieser Fall, der einem Klassifikator entspricht, welcher den Merkmalsraum mit Hyperebenen aufteilt,

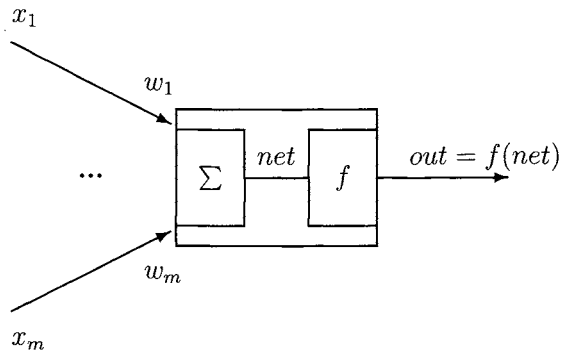


Bild 32.1: Grundlegende Struktur eines künstlichen Neurons.

wird als neuronales Netz in der Praxis selten verwendet, da die Einsatzmöglichkeiten eingeschränkt sind.

- $f(x)$ ist eine Schwellwert- oder Treppenfunktion. Hierbei handelt es sich um eine weit verbreitete Klasse von Aktivierungsfunktionen, da sie es erlauben, diskrete Ausgangsaktivierungen zu erzeugen.
- $f(x)$ ist die *Sigmoidfunktion*:

$$f(x) = \frac{1}{1 + e^{-\frac{x-a}{r}}}. \tag{32.3}$$

  Der Parameter a bestimmt die Lage des Durchgangs durch das Niveau 0.5 und der Parameter r die Steigung der Sigmoidfunktion. Die Sigmoidfunktion gehört zur Klasse der semilinearen Funktionen, die u.a. die Eigenschaft besitzen, überall differenzierbar zu sein.
- Manche Netze verwenden auch stochastische Aktivierungen (z.B. die Boltzmann-Maschine).

32.1.2 Adaline und Madaline

Zwei Spezialisierungen des allgemeinen Konzepts sind das Adaline (Adaptive Linear Neuron, Bild 32.2) und das Madaline (Multiple Adaline). Netze dieser Typen wurden in der Anfangszeit der Forschungen auf dem Gebiet der neuronalen Netze untersucht. Wegen ihrer eingeschränkten Fähigkeiten werden sie heute in der Praxis selten verwendet. Es lassen sich jedoch die grundlegenden Fragestellungen gut anhand dieser Netze erläutern.

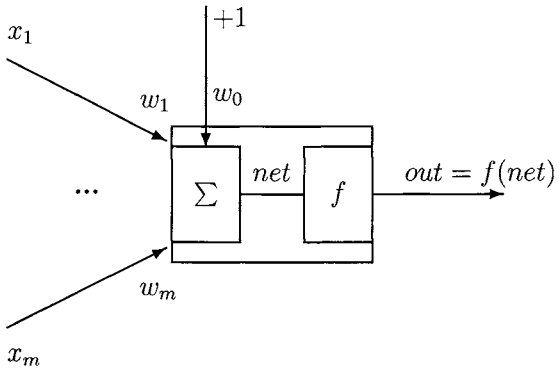


Bild 32.2: Struktur eines Adaline (Adaptive Linear Neuron).

Für die Größen $x_i, i = 1, ..., m$, also die Eingangsaktivierungen, gilt: $x_i \in \{-1, +1\}$. Das Gewicht w_0 übernimmt die Funktion eines Schwellwertes. Es ist die Aktivierung eines fiktiven Eingabeelements, das als Eingabeaktivierung immer mit $+1$ besetzt ist. Die Netzaktivität *net* und die Ausgabe *out* werden wie folgt berechnet:

$$net = \sum_{i=1}^{m} x_i w_i + w_0, \tag{32.4}$$

$$out = \begin{cases} +1, & \text{falls } net > 0, \text{ also } \sum\limits_{i=1}^{m} x_i w_i > -w_0 \\ -1, & \text{sonst.} \end{cases} \tag{32.5}$$

Es wird hier also eine Schwellwertfunktion als Aktivierungsfunktion verwendet.

Im Folgenden wird die Arbeitsweise eines Adaline am Beispiel des logischen AND gezeigt. Die Wahrheitswerte TRUE und FALSE werden durch $+1$ und -1 dargestellt. Als Gewichte werden verwendet: $w_1 = 0.1$ und $w_2 = 0.1$. Der Schwellwert ist $w_0 = -0.1$. Damit sehen die Werte wie folgt aus:

Eingabe x_1/x_2	*net*	*out*	Soll
$-1/-1$	-0.3	-1	-1
$-1/+1$	-0.1	-1	-1
$+1/-1$	-0.1	-1	-1
$+1/+1$	$+0.1$	$+1$	$+1$

Andere logische Operatoren lassen sich ähnlich realisieren. Schwierigkeiten gibt es jedoch bei XOR und bei NXOR. Eine Betrachtung des zweidimensionalen Raumes der Ein-

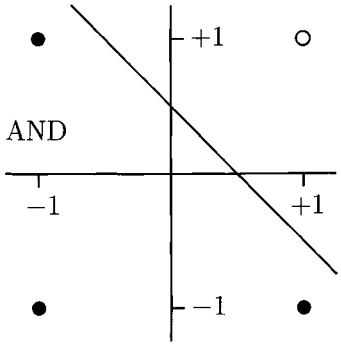


Bild 32.3: Zweidimensionaler Raum der Eingabevektoren für die zweistelligen logischen Operatoren der Boole'schen Algebra. Die Wahrheitswerte TRUE und FALSE werden durch $+1$ und -1 dargestellt.

gabevektoren für die zweistelligen logischen Operatoren der Boole'schen Algebra zeigt, warum das so ist (Bild 32.3).

Die Formel (32.4) sieht ausgeschrieben wie folgt aus:

$$x_1 w_1 + x_2 w_2 + w_0 = net. \tag{32.6}$$

Ein Adaline teilt den Raum der Eingabevektoren durch eine Gerade in zwei Teilmengen M^+ und M^-, so dass alle Eingabevektoren mit $out = +1$ in M^+ und alle Eingabevektoren mit $out = -1$ in M^- liegen. Im Fall des **XOR** ist das nicht möglich: Es müssten nämlich die Punkte $(x_1, x_2) = (-1, -1)$ und $(x_1, x_2) = (+1, +1)$ auf der einen Seite der Gerade und die Punkte $(x_1, x_2) = (+1, -1)$ und $(x_1, x_2) = (-1, +1)$ auf der anderen Seite liegen. Ein Adaline in dieser Form ist also nur bei Problemstellungen einsetzbar, bei denen die Eingabevektoren $\mathbf{x}$ linear separierbar sind. Bei hochdimensionalen Eingabevektoren $\mathbf{x}$ ist es nicht mehr trivial möglich zu entscheiden, ob der Merkmalsraum linear separierbar ist.

Ein anderes Problem ist die Wahl der Gewichte. Bei allgemeinen m-dimensionalen Eingabevektoren ist die Wahl der Gewichte $w_i, i = 0, 1, ..., m$ nicht durch geometrische Überlegungen möglich. Vielmehr müssen die Gewichte von anfänglichen Startwerten aus an die jeweilige Problemstellung adaptiert werden. Einen beispielhaften Algorithmus dazu beschreibt A32.1:

A32.1: Automatische Adaption der Gewichte eines Adaline.

Voraussetzungen und Bemerkungen:

◇ Die Gewichte werden mit $w_i, i = 0, 1, ..., m$ bezeichnet. w_0 ist der Schwellwert.

Algorithmus:

(a) Die Gewichte $w_i, i = 0, 1..., m$ werden mit zufälligen Werten besetzt.

(b) Nacheinander werden zufällige, gleichverteilte Eingabevektoren $\mathbf{x} = (x_1, ..., x_m)^T$ angelegt.

(c) Es wird eine Soll-Ausgabe out_{soll} ermittelt:

$$out_{soll} = \begin{cases} +1, & \text{falls } \mathbf{x} \in M^+ \\ -1, & \text{falls } \mathbf{x} \in M^- \end{cases}$$

(d) Die Abweichung von net vom Zielwert out_{soll} wird berechnet:

$$\delta_{ges} = out_{soll} - net.$$

(e) Um den ermittelten Fehler auszugleichen, werden alle Gewichte um einen Betrag δ verändert:

$$\delta = \frac{\delta_{ges}}{m+1}.$$

$$w_i \Leftarrow w_i + \eta x_i \delta; \qquad i = 1, ..., m;$$

$$w_0 \Leftarrow w_0 + \eta\delta.$$

Der Faktor η wird als *Lernfaktor* bezeichnet. Er bestimmt, in welchem Maß der ermittelte Fehler die Gewichtskorrektur beeinflusst.

Ende des Algorithmus

In dieser Form hat ein Adaline den entscheidenden Nachteil, dass der m-dimensionale Raum der Eingabevektoren nur linear separiert werden kann. Einfache Problemstellungen, wie z.B. die Realisierung eines logischen XOR, sind damit nicht möglich. Aus diesem Grund wurden Erweiterungen des Konzepts durchgeführt. Eine davon führt zum *Madaline* (Multiple Adaline), das in Bild 32.4 abgebildet ist.

Die Eingabeaktivierungen werden auf alle Adalines gelegt. Diese arbeiten autonom nach dem oben beschriebenen Verfahren. Jedes Adaline liefert eine Ausgabeaktivierung out_i, die durch ein *Verdichtungsverfahren* im Madaline zusammengefasst werden. Beispiele für Verdichtungsverfahren sind:

- Das *Mehrheitsverfahren*.

$$out_{mada} = \begin{cases} +1, & \text{falls } \sum_{i=1}^{n} out_i > 0 \\ -1, & \text{sonst} \end{cases} \tag{32.7}$$

Als Endaktivierung wird die mehrheitlich erzielte Aktivierung der Adalines verwendet.

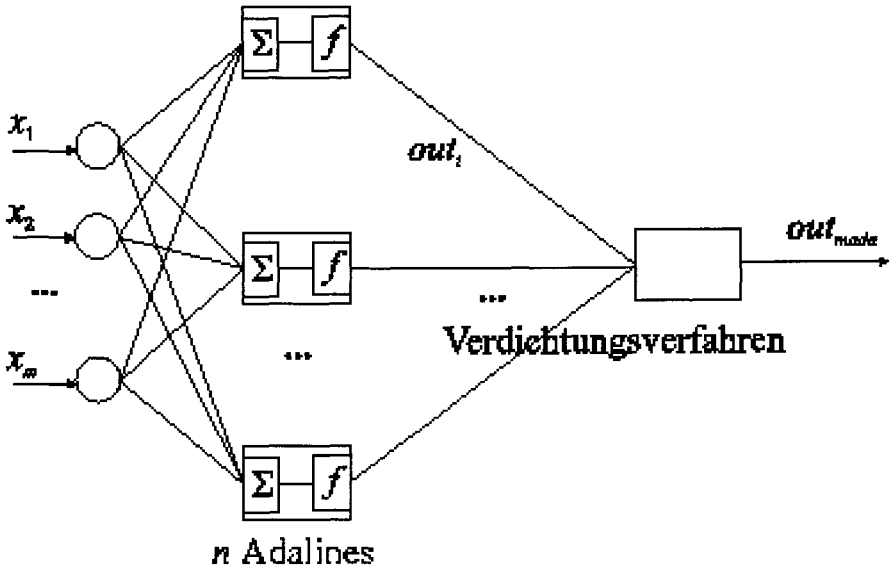


Bild 32.4: Architektur eines Madaline.

- Das *Einstimmigkeitsverfahren.*

$$out_{mada} = \min\{out_i, i = 1, ..., n\}. \tag{32.8}$$

Hier wird als Gesamtergebnis nur dann +1 ausgegeben, falls alle Adalines den Wert +1 liefern.

- Das *Singulärverfahren.*

$$out_{mada} = \max\{out_i, i = 1, ..., n\}. \tag{32.9}$$

Hier wird als Gesamtergebnis schon +1 ausgegeben, falls nur ein Adaline den Wert +1 liefert.

Damit können auch nicht linear separierbare Probleme behandelt werden. Ein Problem bei der Anwendung eines Madalines ist, dass man bei einer gegebenen Anwendung nicht weiß, welches Verdichtungsverfahren man verwenden soll. Im Fall des NXOR (Not-XOR, auch Äquivalenz genannt) ist das Einstimmigkeitsverfahren geeignet (Bild 32.5).

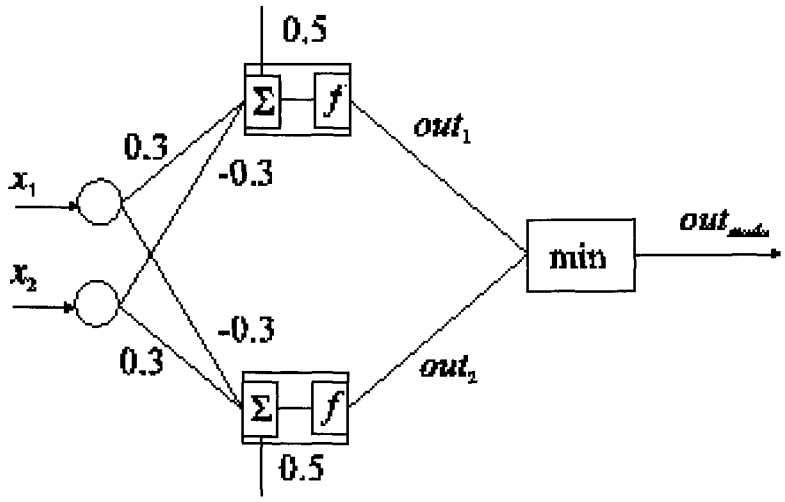


Bild 32.5: Realisierung des NXOR mit einem Madaline.

32.1.3 Das Perceptron

Das Perceptron ist ein künstliches neuronales Netzwerk, das ebenfalls in der Anfangszeit der Forschung auf dem Gebiet der neuronalen Netze ausführlich untersucht wurde. Es besitzt im Prinzip dieselbe Struktur wie das Adaline. Netze, die durch die Parallelschaltung mehrerer Elemente entstehen, werden auch als Perceptron bezeichnet. Der Eingabevektor $\mathbf{x} = (x_1, ..., x_m)^T$ wird auf die *Eingabeschicht* gegeben. Für die Eingabewerte gilt: $x_i \in \{0, 1\}$. Das i-te Neuron der Eingabeschicht ist mit dem j-ten Neuron der *verdeckten Schicht* (d.h. einer Schicht aus Neuronen zwischen Ein- und Ausgabeschicht) mit dem Gewicht w_{ji} verbunden. Die Netzaktivitäten net_j bilden mit der Aktivierungsfunktion f die Ausgangsaktivierungen out_j, wobei gilt:

$$out_j = \begin{cases} 1, & \text{falls } net_j \geq 0 \\ 0, & \text{sonst} \end{cases} \tag{32.10}$$

Die Beziehung zur Berechnung der Größen net_i lautet somit wie folgt:

$$(net_1, net_2, \cdots, net_n) = (x_1, \cdots, x_m) \begin{pmatrix} w_{11} & w_{21} & \cdots & w_{n1} \\ w_{12} & w_{22} & \cdots & w_{n2} \\ \cdots & \cdots & \cdots & \cdots \\ w_{1m} & w_{2m} & \cdots & w_{nm} \end{pmatrix} = \mathbf{x}^T \mathbf{W}. \tag{32.11}$$

Es wäre denkbar, statt nur einer Summationsschicht mehrere hintereinander zu schalten. Auf Grund der linearen Beziehung zwischen dem Eingabevektor und den Netzaktivie-

rungen lässt sich aber leicht zeigen, dass derartige mehrschichtige Netze mit einschichtigen äquivalent sind:

$$\Big(\Big(\cdots(\mathbf{x}^T\mathbf{W_1})\mathbf{W_2}\Big)\cdots\Big)\mathbf{W_k} = \mathbf{x}^T\mathbf{W}. \tag{32.12}$$

Dies gilt nicht, wenn mehrschichtige Netze mit kompletten Perceptronbausteinen und nichtlinearen Aktivierungsfunktionen verwendet werden, was weiter unten noch ausführlicher untersucht wird.

Wie das Training abläuft, ist in Algorithmus A32.2 beschrieben.

A32.2: Adaption der Gewichte bei einem Perceptron-Netz.

Voraussetzungen und Bemerkungen:

◇ Die Gewichte werden mit $w_{ji}, i = 1, ..., m; j = 1, .., n$ bezeichnet.

Algorithmus:

(a) Die Gewichte w_{ji} werden mit zufälligen Werten besetzt (z.B. mit 0 initialisiert).

(b) Eingabevektoren $\mathbf{x}$ werden zufällig ausgewählt. Zu jedem Eingabevektor sei die Soll-Ausgabe $\mathbf{out}_{soll}$ bekannt.

(c) Wenn für die Ausgabekomponente j ein korrektes Ergebnis erzielt wird, so wird keine Gewichtskorrektur durchgeführt.

(d) Falls das Ergebnis falsch war, wird ein Fehlermaß δ_j berechnet:

$\delta_j = out_{soll,j} - out_j$.

Die δ_j gehören zur Menge $\{-1, +1\}$, da die out_j entweder 0 oder 1 sind und die Gewichtskorrektur nur bei einem falschen Ergebnis durchgeführt wird.

(e) Die Gewichte w_{ij} werden jetzt nach folgender Regel korrigiert:

$w_{ij} \Leftarrow w_{ij} + \eta \cdot x_i \cdot \delta_j$.

η ist ein Lernfaktor. Da die x_i entweder 0 oder 1 sind, werden nur solche Gewichte korrigiert, die von einer aktiven Eingabe angesteuert wurden.

Ende des Algorithmus

32.1.4 Backpropagation

Es liegt die Vermutung nahe, dass unter Umständen Netze mit mehreren Schichten doch leistungsfähiger sind. Das ist aber nur der Fall, wenn die beteiligten Neuronen nichtlineare Aktivierungsfunktionen besitzen. Ein weiteres Problem ist das Training: Bei nur einer Schicht können mit Hilfe eines geeigneten Fehlermaßes die Gewichte, wie oben dargestellt, angepasst werden, da ja von der Ausgabeschicht her ein Sollwert vorliegt. Gibt es aber dazwischen verdeckte Schichten, so liegen für die dazu gehörigen Ausgaben keine Sollwerte vor. Bei vorwärts- und rückwärtsvermittelnden Netzen wird dieses Problem mit Hilfe des *Backpropagation*-Algorithmus gelöst.

Bei Netzen dieser Art wird ein Neuron zugrunde gelegt, wie es oben schon beschrieben wurde. Die Eingabeaktivierungen x_i werden hier meistens aus dem Intervall $[-1, +1]$ oder $[0, +1]$ gewählt. Die Größen *net* und *out* berechnen sich wie gewohnt. Als Aktivierungsfunktion wird die Sigmoidfunktion verwendet:

$$f(net) = \frac{1}{1 + e^{-net}}. \tag{32.13}$$

Neben anderen günstigen Eigenschaften der Sigmoidfunktion (z.B. Differenzierbarkeit, Verhalten im Unendlichen, usw.) gilt für ihre Ableitung:

$$f'(net) = f(net)(1 - f(net)). \tag{32.14}$$

Diese Eigenschaft wird später, bei der Adaption der Netzgewichte, verwendet (Gradientenabstiegsverfahren). Die Einführung eines Bias (zusätzliche Eingabe, die immer +1 ist und ein eigenes Gewicht w_0 besitzt) ist möglich, wird in dieser Darstellung aber weggelassen. Mit diesem Neuron als Grundelement werden nun mehrschichtige Netze aufgebaut (Bild 32.6).

In der Eingabeschicht geschieht keine Verarbeitung, sondern es werden nur die Eingabewerte x_i auf die Eingabewerte a_i der (ersten) verdeckten Schicht verteilt. Die letzte Schicht ist die Ausgabeschicht. Ihre Ausgabewerte sind zugleich die Ausgabewerte out_j des gesamten Netzes.

Das Training wird mit zufällig ausgewählten Eingabevektoren $\mathbf{x}$ und den dazu gehörigen Sollvektoren $\mathbf{out}_{soll}$ durchgeführt. Es läuft wie in Algorithmus A32.3 dargestellt ab:

A32.3: Backpropagation.

Algorithmus:

(a) Die Gewichte der Neuronen werden mit zufälligen Werten besetzt.

(b) Es wird zufällig ein Trainingspaar $(\mathbf{x}, \mathbf{out}_{soll})$ ausgewählt.

(c) Mit Hilfe der *Vorwärtsvermittlung* wird ein Ausgabevektor **out** berechnet.

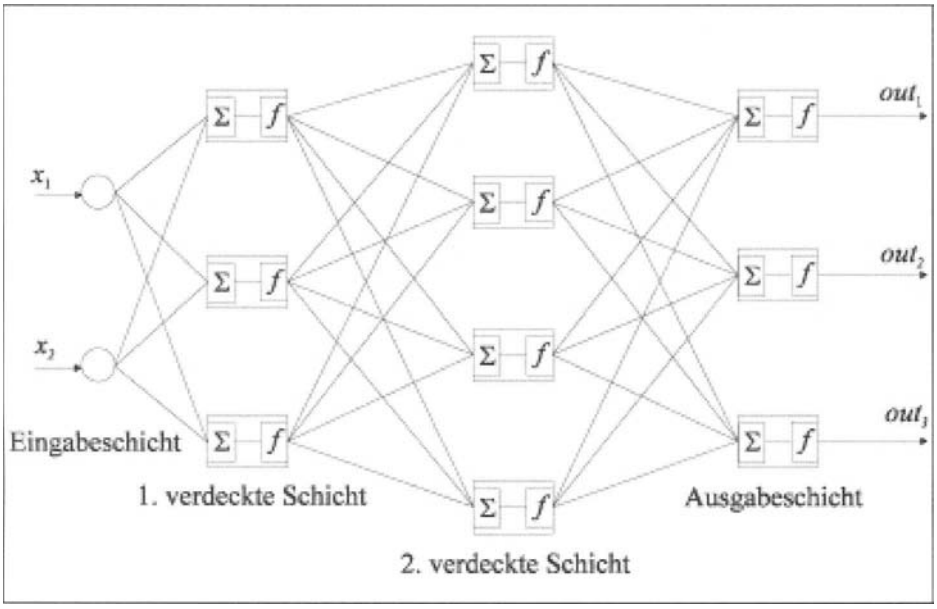


Bild 32.6: Mehrschichtiges Backpropagation-Netz.

(d) Es wird ein Fehlermaß zwischen **out** und $\mathbf{out}_{soll}$ berechnet.

(e) Die Gewichte werden so modifiziert, dass das Fehlermaß minimiert wird.

(f) Die Schritte (b)-(e) werden solange wiederholt, bis der Fehler akzeptabel klein ist.

Ende des Algorithmus

Die Positionen (b) und (c) von Algorithmus A32.3 werden als *Vorwärtsschritt* bezeichnet und die Positionen (d) und (e) als *Rückwärtsschritt*.

Zur genaueren Beschreibung werden folgende Annahmen gemacht (für ein Neuron sind die Bezeichnungen in Bild 32.7 zusammengestellt):

Das Netz habe die Schichten $k = 0, 1, ..., l$, wobei $l > 1$ sei. Mit $k = 0$ wird die Eingabeschicht bezeichnet. Hier wird keine Verarbeitung durchgeführt, sondern es werden die Elemente des Eingabevektors $\mathbf{x} = (x_1, x_2, ..., x_m)^T$ auf die Neuronen der ersten verdeckten Schicht ($k = 1$) verteilt.

Die Schicht mit dem Index $k = l$ ist die Ausgabeschicht. Die Schichten mit den Indizes $1 \leq k \leq l - 1$ sind die verdeckten Schichten. Jede Schicht besitzt n_k Neuronen. Für $k = 0$ gilt $n_0 = m$ und für $k = l$ gilt $n_l = n$. Die Eingabeschicht hat also m Neuronen und die Ausgabeschicht n Neuronen. Ein Trainingspaar $(\mathbf{x}, \mathbf{out}_{soll})$ besteht aus einem Eingabevektor und dem dazugehörigen Ausgabesollvektor.

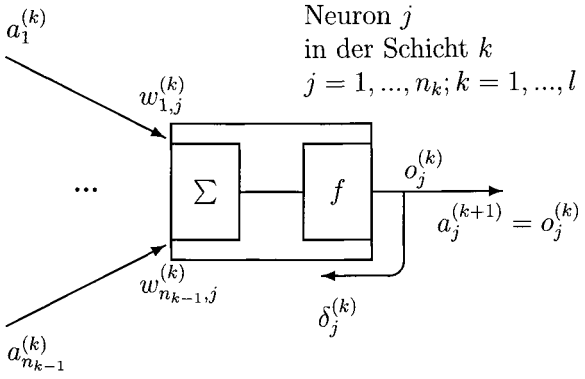


Bild 32.7: Bezeichnungen eines Neurons im Rahmen des Backpropagation-Algorithmus.

Die Eingangsaktivierungen eines Neurons j in der Schicht $k = 1, 2, ..., l$ seien $a_1^{(k)}, ..., a_{n_{k-1}}^{(k)}$. Die Ausgabe dieses Neurons wird mit $o_j^{(k)}$ bezeichnet. Die Eingaben der Neuronen der Schicht k sind, mit Ausnahme der ersten Schicht, die Ausgaben der Schicht $k-1$:

$$k = 1: \qquad a_i^{(k)} = x_i, i = 1, ..., n_0 = m. \tag{32.15}$$

$$k > 1: \qquad a_i^{(k)} = o_i^{(k-1)}, i = 1, ..., n_{k-1}; \tag{32.16}$$

Im Vorwärtsschritt wird zu jedem Neuron $j = 1, ..., n_k$ der Schicht $k = 1, 2, ...l$ die Ausgabe $o_j^{(k)}$ berechnet:

$$net_j^{(k)} = \sum_{i=1}^{n_{k-1}} a_i^{(k)} w_{ij}^{(k)}, \qquad o_j^{(k)} = f_j(net_j^{(k)}). \tag{32.17}$$

Dabei ist $w_{ij}^{(k)}$ das Gewicht der i-ten Eingangsaktivierung $(i = 1, ..., n_{k-1})$ des Neurons j, $j = 1, ..., n_k$ in der Schicht $k = 1, ...l$. Hier wurde berücksichtigt, dass die einzelnen Neuronen unterschiedliche Aktivierungsfunktionen haben können. Die Ausgabe einer Schicht k sind die Eingaben der nächsten Schicht $k+1$ $(k = 1, .., l-1)$:

$$a_j^{(k+1)} = o_j^{(k)}, j = 1, ..., n_k. \tag{32.18}$$

Die Ausgabewerte des gesamten Netzes sind die Ausgaben der letzten Schicht $k = l$:

$$out_j = o_j^{(l)}, j = 1, ..., n_l = n. \tag{32.19}$$

Nun der Rückwärtsschritt: Zunächst wird in der Ausgabeschicht mit Hilfe der Sollausgabe ein Korrekturwert $\delta_j^{(l)}$ für jedes Neuron berechnet. Dieses Maß wird nach der *Gradientenabstiegsmethode* ermittelt. Hier gehen die speziellen Eigenschaften der Ableitung der Sigmoidfunktion mit ein:

$$\delta_j^{(l)} = out_j(1 - out_j)(out_{soll,j} - out_j). \tag{32.20}$$

Für die verdeckten Schichten $k = l-1, ..., 1$ wird das Korrekturmaß wie folgt berechnet:

$$\delta_j^{(k)} = o_j^{(k)}(1 - o_j^{(k)}) \sum_{i=1}^{n_{k+1}} w_{ji}^{(k+1)} \delta_i^{(k+1)}. \tag{32.21}$$

Nun wird die Korrektur der Gewichte $w_{ij}^{(k)}$ vom Iterationsschritt t zum Iterationsschritt $t+1$ durchgeführt:

$$w_{ij}^{(k)}(t+1) = w_{ij}^{(k)}(t) + \eta \delta_j^{(k)} a_i^{(k)}, j = 1, ..., n_k, i = 1..., n_{k-1}. \tag{32.22}$$

Der Parameter η ist, wie schon oben, die Lernrate. Die Iteration wird beendet, wenn die Streuung der Abweichungen der Ausgabevektoren von den Ausgabesollvektoren kleiner als eine vorgegebene Schranke ist:

$$\frac{1}{2} \sum_{j=1}^{n_l} (out_{soll,j} - out_j)^2 < \epsilon. \tag{32.23}$$

Mit diesen Ausführungen soll die Darstellung der neuronalen Netze beendet werden. Es gibt noch viele weitere Netztypen, z.B. solche, die die Eingangsvektoren ohne Sollausgabe zu Punktwolken zusammenfassen und somit als unüberwachte Klassifikatoren verwendet werden können. Andere Netze arbeiten wie ein Signalverstärker: Ein Eingabevektor wird gelernt. Die Ausgabe ist derselbe Eingabevektor. Wenn nun der Eingabevektor leicht „verrauscht" ist, wird er durch diesen Netztyp wieder rekonstruiert (z.B. [Koho88]).

32.2 Neuronale Netze als Klassifikatoren

Nach der obigen Darstellung ist leicht zu sehen, wie neuronale Netze in der digitalen Bildverarbeitung und Mustererkennung eingesetzt werden können, nämlich überall dort, wo Merkmalsvektoren klassifiziert werden. Dabei kann es sich um Merkmalsvektoren handeln, wie sie in den vorhergehenden Abschnitten verwendet wurden, es können aber auch solche sein, die z.B. die Form eines Segments oder Objekts beschreiben. Es ist auch denkbar, die Bildpunkte ganzer Bilder oder von Bildausschnitten als Eingabevektoren eines neuronalen Netzes zu verwenden. Beispiele hierzu sind Bildausschnitte, die Objekte wie Buchstaben oder Symbole enthalten. Auch Bildausschnitte, die primitive Objekte enthalten (Grundbausteine für komplexere Formen, wie z.B. Linienstücke, Ecken, Verzweigungen, usw.), können als Eingabevektoren dienen.

In den folgenden Abschnitten werden einige einfache Beispiele dargestellt. Sie zeigen auch die Problematik bei der Verwendung neuronaler Netze auf. Ein Beispiel zur rotations- und größeninvarianten Segmenterkennung mit neuronalen Netzen wird in Kapitel 37 gegeben.

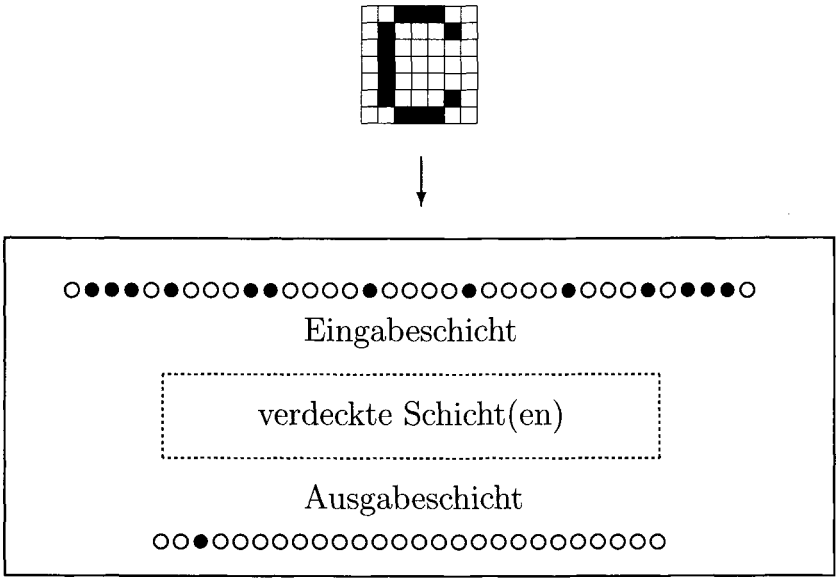


Bild 32.8: Klassifikation von Bildern (Bildausschnitten), die $7 \cdot 5$ Bildpunkte große Großbuchstaben enthalten. Die Bildpunkte werden hier direkt als Eingabe für das neuronale Netz verwendet. Da die linke und die rechte Randspalte bei allen Buchstaben nicht besetzt ist, werden diese Bildpunkte in der Eingabeschicht nicht verwendet. Die 26 Ausgabeneuronen geben die Positionen der Buchstaben im Alphabet an.

32.2.1 Verarbeitung von Binärbildern

Bei der Anwendung in diesem Abschnitt werden binäre Bilddaten vorausgesetzt. Es könnte sich dabei z.B. um Bilder (Bildausschnitte) handeln, die Großbuchstaben enthalten. Bild 32.8 zeigt den Buchstaben „C" mit einer Buchstabengröße von $7 \cdot 5$ Bildpunkten. Die linken und rechten Randspalten sind bei allen Buchstaben weiß. Sie werden deshalb nicht in der Eingabeschicht mitverwendet.

Die 35 Bildpunkte wurden in verschiedenen Testläufen ohne weitere Verarbeitung auf die 35 Eingangsneuronen eines Backpropagation-Netzes gegeben. Die Ausgabeschicht bestand aus 26 Neuronen. Bei einem bestimmten Großbuchstaben als Eingabe wurde als Sollausgabe die Position des Buchstaben im Alphabet (10...0 $\hat{=}$ A, 010...0 $\hat{=}$ B, ...) trainiert.

In der folgenden Tabelle ist zusammengefasst, wie die verdeckten Schichten bei unterschiedlichen Tests gewählt wurden:

verd.Schichten	Neuronen	Training	Systemfehler	0 Fehler	1 Fehler	2 Fehler
1	15	267	0.025	0	2	8
1	15	811	0.01	0	2	8
1	20	474	0.025	0	0	1
1	20	845	0.01	0	0	1
1	25	673	0.01	0	0	5
2	15 15	283	0.01	0	3	5
3	11 13 11	882	0.01	0	10	15

Der Test in der letzten Zeile der Tabelle wurde mit einer anderen Anzahl von Ausgabeneuronen durchgeführt. Auf diesen Test wird weiter unten näher eingegangen. Alle anderen Tests wurden, wie oben beschrieben, mit 35 Eingabe- und 26 Ausgabeneuronen durchgeführt.

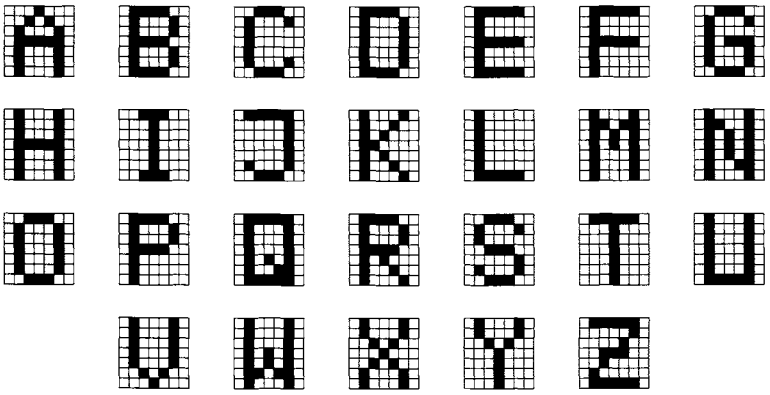
In der Spalte „Neuronen“ ist die jeweilige Anzahl der Neuronen der verdeckten Schicht(en) bei den verschiedenen Tests angegeben.

Die Spalte „Training“ gibt an, wieviele Trainingsdurchläufe mit der Trainingsmenge durchgeführt wurden, um ein zufriedenstellendes Ergebnis zu erreichen. Die Trainingsdaten bestanden aus den Buchstaben „A“ bis „Z“ und den entsprechenden Sollausgaben „10...0“ bis „0...01“. In der nächsten Spalte „Systemfehler“ ist der erzielte Systemfehler angegeben. Darunter versteht man die Streuung der Abweichungen der Soll-Werte von den trainierten Ist-Werten.

Die letzten drei Spalten der Tabelle enthalten die Ergebnisse von Erkennungsläufen. In der Spalte „0 Fehler“ ist das Ergebnis der Klassifizierung der Buchstaben „A“ bis „Z“ zusammengefasst, wobei die Buchstaben exakt so in das Netz eingegeben wurden, wie sie trainiert wurden (siehe Bild 32.9 oben). In der Ausgabeschicht wurde das Neuron mit dem maximalen Wert als „gesetzt“ und alle anderen als „nicht gesetzt“ interpretiert. In allen Fällen konnten die trainierten Buchstaben richtig erkannt werden. Das ist aber nicht weiter verwunderlich, da man ja in der Regel so lange trainiert, bis alle Eingabevektoren richtig erkannt werden. Wenn das nicht gelingt, so ist das ein Hinweis, dass die Struktur des Netzes ungeeignet ist.

In den Spalten „1 Fehler“ und „2 Fehler“ wurden Buchstaben verwendet, die jeweils einen (Bild 32.10 oben) bzw. zwei zufällig gestörte Pixel enthielten. Die Spalten geben an, wieviele Buchstaben falsch erkannt wurden. Man sieht hier deutliche Unterschiede zwischen den einzelnen Testläufen.

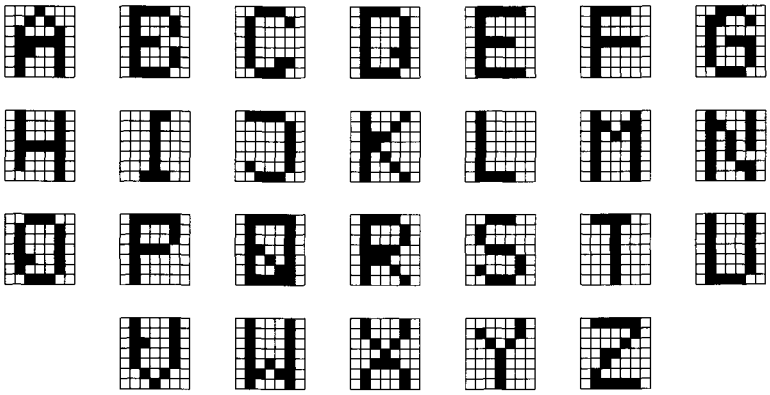
Die Ergebnisse der letzten Zeile der Tabelle beschreiben einen Test mit einer anderen Anzahl von Ausgabeneuronen und einem anderen Training. Die Eingabeschicht bestand wie vorher aus 35 Neuronen. Es wurde ein Netz mit drei verdeckten Schichten mit jeweils 11, 13 und 11 Neuronen verwendet. Als Ausgabe wurden nur fünf Neuronen verwendet, die die Position des jeweiligen Buchstaben im Alphabet als Dualzahl angeben (00001 $\hat{=}$ A, 00010 $\hat{=}$ B, 00011 $\hat{=}$ C,...). Die Ausgabe ist hier codiert, im Gegensatz zu den anderen Tests, wo für die jeweilige Position des Buchstaben im Alphabet ein Bit gesetzt wurde. Damit wird dem Netz mehr zugemutet, da es die Ausgaberepräsentation codieren muss.



Werte der Ausgabeneuronen:

```
0.020837  0.028464  0.017265  0.021185  0.982142 -> 00001 ->  1  A
0.020670  0.020221  0.014934  0.989956  0.019147 -> 00010 ->  2  B
0.027935  0.023546  0.024885  0.986592  0.999935 -> 00011 ->  3  C
0.000572  0.014370  0.999676  0.018239  0.018549 -> 00100 ->  4  D
0.002510  0.015419  0.999610  0.020248  0.981162 -> 00101 ->  5  E
0.022271  0.003551  0.998852  0.974118  0.005454 -> 00110 ->  6  F
0.001091  0.008356  0.990704  0.989253  0.999025 -> 00111 ->  7  G
0.018397  0.999732  0.023999  0.021194  0.019648 -> 01000 ->  8  H
0.015209  0.995435  0.015903  0.011473  0.997451 -> 01001 ->  9  I
0.017719  0.995189  0.000752  0.976232  0.001950 -> 01010 -> 10  J
0.015654  0.988395  0.004384  0.981284  0.999403 -> 01011 -> 11  K
0.000068  0.982192  0.999956  0.018638  0.018289 -> 01100 -> 12  L
0.005537  0.985710  0.998434  0.016890  0.984309 -> 01101 -> 13  M
0.015791  0.982911  0.999132  0.978255  0.000060 -> 01110 -> 14  N
0.000073  0.972075  0.979134  0.979410  0.984619 -> 01111 -> 15  O
0.991416  0.024989  0.024072  0.032943  0.000741 -> 10000 -> 16  P
0.986835  0.017315  0.013341  0.013476  0.991117 -> 10001 -> 17  Q
0.993490  0.013013  0.000066  0.976666  0.027819 -> 10010 -> 18  R
0.984228  0.013388  0.000033  0.999302  0.986272 -> 10011 -> 19  S
0.976467  0.018553  0.968682  0.026200  0.023099 -> 10100 -> 20  T
0.975068  0.003692  0.983546  0.022007  0.977489 -> 10101 -> 21  U
0.976925  0.000857  0.982606  0.972932  0.014265 -> 10110 -> 22  V
0.987626  0.000696  0.977273  0.978641  0.989696 -> 10111 -> 23  W
0.997931  0.990342  0.009320  0.001847  0.009105 -> 11000 -> 24  X
0.975234  0.976796  0.000583  0.016451  0.971005 -> 11001 -> 25  Y
0.969912  0.973222  0.000023  0.992793  0.014657 -> 11010 -> 26  Z
```

Bild 32.9: Ergebnisse der Großbuchstabenerkennung. Unter den Großbuchstaben sind die Werte der fünf Ausgabeneuronen aufgelistet. Mit einem Schwellwert von 0.5 können sie als Dualzahlen interpretiert werden, die die Nummer des Buchstaben im Alphabet angeben. Es ist zu sehen, dass hier ohne Schwierigkeiten die 26 Großbuchstaben trainiert und wiedererkannt werden konnten.



Werte der Ausgabeneuronen:

```
0.022017  0.026538  0.011898  0.039725  0.976535 -> 00001 ->  1  A
0.001897  0.061533  0.974516  0.902196  0.006236 -> 00110 ->  6  B -> F
0.045946  0.024845  0.011010  0.979124  0.999941 -> 00011 ->  3  C
0.003161  0.016500  0.999421  0.094493  0.004494 -> 00100 ->  4  D
0.654648  0.000696  0.755598  0.865997  0.992804 -> 10111 -> 23  E -> W
0.000370  0.013200  0.999785  0.369456  0.027534 -> 00100 ->  4  F -> D
0.000028  0.780247  0.983394  0.997842  0.973749 -> 01111 -> 15  G -> O
0.010467  0.999752  0.013863  0.049535  0.030799 -> 01000 ->  8  H
0.030367  0.996544  0.009684  0.005725  0.998188 -> 01001 ->  9  I
0.000111  0.985478  0.495491  0.615509  0.004989 -> 01010 -> 10  J
0.038400  0.996255  0.001789  0.832148  0.998877 -> 01011 -> 11  K
0.000060  0.981270  0.999959  0.069995  0.020088 -> 01100 -> 12  L
0.005423  0.986693  0.977855  0.034662  0.994876 -> 01101 -> 13  M
0.106422  0.871062  0.998939  0.986739  0.000044 -> 01110 -> 14  N
0.000059  0.880481  0.999446  0.959640  0.934473 -> 01111 -> 15  O
0.997912  0.017865  0.003772  0.075668  0.004611 -> 10000 -> 16  P
0.999167  0.010610  0.153588  0.023471  0.099921 -> 10000 -> 16  Q -> P
0.995054  0.018077  0.001841  0.336459  0.003432 -> 10000 -> 16  R -> P
0.004964  0.019297  0.001310  0.973724  0.990766 -> 00011 ->  3  S -> C
0.644564  0.043035  0.988861  0.030728  0.003821 -> 10100 -> 20  T
0.976208  0.008697  0.987211  0.162659  0.970364 -> 10101 -> 21  U
0.997931  0.007157  0.968808  0.136341  0.310576 -> 10100 -> 20  V -> T
0.996570  0.001039  0.541286  0.994480  0.986531 -> 10111 -> 23  W
0.979371  0.998029  0.001152  0.019924  0.651716 -> 11001 -> 25  X -> Y
0.681093  0.990846  0.000309  0.290778  0.994714 -> 11001 -> 25  Y
0.012139  0.999637  0.000950  0.523317  0.044018 -> 01010 -> 10  Z -> J
```

Bild 32.10: Ergebnisse der Großbuchstabenerkennung mit Buchstaben, die jeweils durch ein falsches Pixel gestört sind. Es ergeben sich beträchtliche Fehlklassifizierungen.

Es wird folglich eine größere Netzkomplexität notwendig.

Nach 882 Trainingszyklen wurde der Systemfehler auf einen Wert von 0.01 reduziert. Bild 32.9 zeigt die Ergebnisse zu dieser Problemstellung. Unter den Großbuchstaben sind die Werte der fünf Ausgabeneuronen aufgelistet. Mit einem Schwellwert von 0.5 können sie als Dualzahlen interpretiert werden, die die Nummer des Buchstaben im Alphabet angeben. Es ist zu sehen, dass hier ohne Schwierigkeiten die 26 Großbuchstaben trainiert und wiedererkannt werden konnten.

In einem zweiten Erkennungsdurchlauf wurden alle Buchstaben durch ein zufälliges fehlerhaftes Pixel gestört. Das Ergebnis dazu zeigt Bild 32.10. Hier treten erhebliche Fehlklassifizierungen auf. Für den menschlichen Betrachter ist z.B. das gestörte „E" ohne weiteres zu erkennen. Das neuronale Netz klassifiziert es aber als den Buchstaben „W". Im dritten Lauf wurden die Buchstaben durch zwei zufällige Pixel gestört, was zur Folge hat, dass mehr als die Hälfte aller Buchstaben falsch erkannt wurde (siehe Bild 32.10 unten).

Damit werden Fragen deutlich, denen man bei der Verwendung neuronaler Netze begegnet:

- Wieviele Ein- und Ausgabeneuronen sind zu verwenden?
- Wieviele verdeckte Schichten mit wie vielen Neuronen sollte man wählen?
- Welche Netztopologie soll überhaupt gewählt werden?
- Wie sollen Parameter wie z.B. die Lernrate η oder die Steigung der Sigmoidfunktion gewählt werden?
- Wie lange soll trainiert werden?
- Auf welchen Wert soll der Systemfehler reduziert werden?

Diese Fragen lassen sich nicht generell beantworten. Bei vielen praktischen Anwendungen ist man auf reines Probieren (oder „eine gute Nase") angewiesen. Trotzdem seien einige Hinweise gegeben, die man beachten sollte:

- Die Zahl der Eingabeneuronen ist in den Anwendungen des vorliegenden Bereichs oft durch die jeweilige Problemstellung vorgegeben. Durch eine geeignete Vorverarbeitung der Eingabedaten kann das neuronale Netz von der Notwendigkeit des Aufbaus interner Repräsentationen befreit werden. Eine Codierung der Eingabedaten kann die Anzahl der Eingabeneuronen reduzieren. Sie kann sich aber auch nachteilig auswirken, wenn das Netz die Repräsentation decodieren muss. Die Entfernung von Redundanzen in den Eingabedaten, z.B. mit der Hauptkomponententransformation, kann ebenfalls die Anzahl der notwendigen Eingabeneuronen reduzieren.
- Die Ausgabe sollte man so einfach wie möglich gestalten, was aber nicht immer bedeuten muss, dass das Netz mit weniger Ausgabeneuronen die besseren Ergebnisse liefert. Die Beispiele der obigen Tests bestätigen dies: Das eindeutig bessere Ergebnis

liefern die Tests, bei denen die Position des Buchstaben im Alphabet als Bitposition innerhalb der 26 Ausgabeneuronen codiert wurde. Bei der Codierung als Dualzahl benötigt man zwar weniger Ausgabeneuronen, aber dem Netz wird mehr Codierungsaufwand zugemutet. Es ist in den einzelnen Anwendungsfällen zu prüfen, ob es nicht besser ist, eine weitgehend uncodierte Ausgabe zu erzeugen und die gewünschten Ergebnisse in der Nachverarbeitung zu ermitteln.

- Je größer die Anzahl der zu lernenden Musterzuordnungen und je komplexer die Ein-/Ausgabebeziehungen sind, desto mehr verdeckte Neuronen werden notwendig. Die Anzahl der verdeckten Schichten und die Anzahl der Neuronen pro Schicht sollte aber trotzdem, problemangepasst, so gering wie möglich sein.

  Wird ein Netz zu groß dimensioniert, so schwindet der *Generalisierungseffekt*, d.h. die Fähigkeit, die Beziehungen zwischen nicht gelernten Daten korrekt wiederzugeben.

  Im obigen Beispiel liegt das Optimum anscheinend in der Nähe von 20 Neuronen bei einer verdeckten Schicht. Werden mehr oder weniger Neuronen verwendet, so schwindet der Generalisierungseffekt, d.h. es werden bei gestörten Eingabemustern mehr Fehler gemacht.

- Bei komplexen Beziehungen kann die Topologie des Netzes aufwändiger gestaltet werden als bei einem Standard-Backpropagationnetz. Hier sind Strukturen denkbar, bei denen z.B. die Ausgabeschicht zusätzlich direkt mit der Eingabeschicht verbunden ist oder bei denen zwei Systeme von verdeckten Schichten nebeneinander verwendet werden. Dadurch werden aber mehr Neuronen und Verbindungen notwendig, was zu einer schlechteren Generalisierung führen kann.

- Die Lernrate und die Steigung der Sigmoidfunktion beeinflussen die Konvergenz des Netzes in der Trainingsphase. Ein hoher Wert η kann zum Alternieren des Systemfehlers beitragen. Ein zu kleiner Wert kann eine sehr langsame Konvergenz zur Folge haben. In den vorliegenden Beispielen wurde $\eta = 0.7$ gewählt.

  Die Sigmoidfunktion ist für die Nicht-Linearität des Netzes verantwortlich. Wird die Steigung reduziert, so kann im Grenzfall ein nahezu lineares System erreicht werden. Das andere Extremum ist eine Schwellwertfunktion.

  Nichtlineare Systeme werden auch in der *Chaostheorie* untersucht. Möglicherweise sind Erkenntnisse aus der Chaostheorie auf neuronale Netze anwendbar, z.B. dass in einem nichtlinearen System geringfügige Änderungen an den Eingabedaten kolossale Änderungen der Ausgabedaten bewirken können. Um ein gutes Generalisierungverhalten zu erzielen, müsste man folglich möglichst „wenig nichtlineare" Netze verwenden. Dann ist aber das Konvergenzverhalten in der Regel schlechter.

- Wenn zu lange trainiert wird, sinkt das Generalisierungsverhalten. Bei verrauschten Eingabedaten kann es dann z.B. passieren, dass das Rauschen mit trainiert wird. Um ein generalisierendes Netz zu erhalten, trainiert man häufig gerade so lange, bis alle Eingabedaten richtig erkannt werden.

- Der erzielte Systemfehler kann als Maß für die Güte verwendet werden. Trainiert man aber solange, bis der Systemfehler sehr klein wird, so geht das wieder auf Kosten des Generalisierungsverhaltens. In den obigen Beispielen hat sich gezeigt, dass etwa ab einem Systemfehler, der kleiner ist als 0.05, alle unverfälschten Eingabemuster richtig klassifiziert wurden.

32.2.2 Verarbeitung von mehrkanaligen Bildern

In diesem Abschnitt werden die Bildpunkte $\mathbf{s}(x, y)$ eines mehrkanaligen Bildes $\mathbf{S}$ direkt als Eingabevektoren eines neuronalen Netzes verwendet. Die einzelnen Kanäle des Bildes können dabei das Ergebnis einer längeren Folge von Verarbeitungsschritten zur Merkmalsgewinnung (Abschnitt 23.1) sein.

Als Beispiel wird ein dreikanaliges RGB-Farbbild $\mathbf{S}_e = (s_e(x, y, n)), n = 0, 1, 2$ gewählt. Das Netz wird somit zur Farberkennung trainiert. Die Bildpunkte $\mathbf{s}_e(x, y) = (r, g, b)^T$ sind dreidimensionale Vektoren, die den Rot-, Grün- und Blauauszug repräsentieren. In der Trainingsphase werden ausgewählte Farb- (Merkmals-)Vektoren angeboten und dazu angegeben, um welchen Farbton es sich handelt. Die Eingabeschicht besteht hier somit aus drei Neuronen. Die Ausgabeschicht hängt von der Anzahl der zu lernenden Farbtöne ab: Für jede Klasse (Farbe) wird ein Ausgabeneuron verwendet.

In der Trainingsphase zu einem ersten Beispiel wurden die Farbvektoren zu den Farben Rot, Gelb, Grün, Zyan, Blau, Magenta, Weiß und Schwarz trainiert. In folgender Tabelle sind die Eingabevektoren (Spalte „Tr.Vekt."), die Sollausgabe für das Training (Spalte „Soll"), sowie die Eingabedaten für einen Bewertungstest (Spalte „Kl.Vekt.") zusammengefasst. Ein Vergleich mit der grafischen Darstellung des RGB-Farbraumes in Abschnitt 3.2.4 (Bild 3.5) zeigt, dass hier die acht Eckpunkte des Farbwürfels trainiert wurden.

Farbe	Tr.Vekt	Soll	Kl.Vekt.
Rot	1 0 0	1 0 0 0 0 0 0 0	0.80 0.20 0.10 und 0.80 0.20 0.20
Gelb	1 1 0	0 1 0 0 0 0 0 0	0.90 0.85 0.10 und 0.85 0.85 0.20
Grün	0 1 0	0 0 1 0 0 0 0 0	0.20 0.85 0.10 und 0.15 0.85 0.25
Zyan	0 1 1	0 0 0 1 0 0 0 0	0.15 0.85 0.75 und 0.25 0.90 0.75
Blau	0 0 1	0 0 0 0 1 0 0 0	0.10 0.20 0.95 und 0.20 0.35 0.90
Magenta	1 0 1	0 0 0 0 0 1 0 0	0.85 0.15 0.90 und 0.80 0.10 0.85
Schwarz	0 0 0	0 0 0 0 0 0 1 0	0.15 0.05 0.25 und 0.20 0.15 0.20
Weiß	1 1 1	0 0 0 0 0 0 0 1	0.75 0.85 0.85 und 0.80 0.80 0.85

In der nächsten Tabelle sind die Ergebnisse unterschiedlicher Testläufe zusammengestellt. Die Anzahl der Eingabeneuronen war immer drei und die Anzahl der Ausgabeneuronen immer acht. Spalten, die mit „-" gekennzeichnet sind, beziehen sich auf Tests, bei denen der Systemfehler nicht gegen den geforderten Wert von 0.05 konvergierte, sondern in einem lokalen Minimum hängenblieb. Bei den Ausgabeneuronen wurde das Neuron mit dem maximalen Wert als „gesetzt" interpretiert.

verd.Schichten	Neuronen	Training	Systemfehler	Tr.Vekt.	Kl.Vekt.
1	3	134	0.05	0	1
1	3	1790	0.01	0	1
1	5	79	0.05	0	0
1	5	990	0.01	0	0
1	7	-	0.05	-	-
1	11	-	0.05	-	-
2	3 3	184	0.05	0	0
2	3 3	1800	0.01	0	0
2	5 5	104	0.05	0	0
2	5 5	644	0.01	0	0
2	7 7	97	0.05	0	0
2	7 7	706	0.01	0	0
2	11 11	-	0.05	-	-
3	3 5 3	-	0.05	-	-
3	5 7 5	157	0.05	0	0
3	5 7 5	914	0.01	0	0
3	10 20 10	198	0.05	0	0
3	10 20 10	423	0.01	0	0

Um das Generalisierungsverhalten etwas genauer zu untersuchen, wurden in einem zweiten Beispiel aus dem RGB-Farbraum äquidistante Punkte ausgewählt: In der RG-Ebene (Blauanteil = 0.0) die $11 \cdot 11$ Punkte mit einem Rotanteil von 0.0, 0.1 bis 1.0 und und einem Grünanteil von 0.0, 0.1 bis 1.0, in der darüberliegenden Ebene (Blauanteil 0.1) sinngemäß wieder $11 \cdot 11$ Punkte, bis zur oberen Ebene mit einem Blauanteil von 1.0. Diese $11 \cdot 11 \cdot 11$ Punkte wurden klassifiziert. In den resultierenden Farbebenen wurden die Ergebnisvektoren in den trainierten Grundfarben (Rot, Gelb, ..., Weiß) dargestellt. Man sieht so, wie durch die verschiedenen Tests mit unterschiedlichen Netzen der Farbraum aufgeteilt wurde.

Die Ergebnisse sind in den Bildern 32.11, 32.12 und 32.13 zusammengestellt. Die Teilbilder zeigen von links oben nach rechts unten die einzelnen Ebenen des diskretisierten Farbraums. Links oben ist die unterste Ebene mit dem Blauanteil 0.0 (RG-Ebene). Daneben ist die Ebene mit einem Blauanteil von 0.1, usw. Die Ebene im Teilbild rechts unten hat einen Blauanteil von 1.0. Die weiteren Erläuterungen zu den Ergebnissen stehen in den Bildunterschriften.

Im letzten Beispiel zur Klassifizierung mit neuronalen Netzen wurden aus einem gescannten RGB-Farbbild mit 24 Bit pro Bildpunkt, also 8 Bit pro Farbauszug (Bild 32.14-a), zu verschiedenen Farben Trainingsgebiete ausgewählt und einem neuronalen Netz trainiert. Das Merkmal für die Segmentierung der (Farb-)Klassen ist somit der Farbton, d.h. ähnliche Farben werden in einer Klasse zusammengefasst. Es wurden fünf Trainingsgebiete definiert. Die Bildausschnitte wurden aus folgenden Bereichen gewählt: Weiß: Ei, Gelb: Zitrone, Hellblau: Schmetterling, Rot: Apfel, Grün: Hintergrund der Margerite. In der Tabelle sind die dazugehörigen Koordinaten angegeben:

Bild 32.11: Klassifizierung des diskretisierten Farbraums. Hier wurde ein neuronales Netz mit drei Eingabeneuronen, einer verdeckten Schicht mit fünf Neuronen und einer Ausgabeschicht mit acht Neuronen verwendet. Man sieht deutlich, dass in der untersten Ebene die Farben Schwarz, Rot, Gelb und Grün fast symmetrisch verteilt sind. In der vierten Ebene tritt Zyan und in der fünften Ebene treten Blau, Magenta und Weiß zum ersten Mal auf. In den weiteren Ebenen mit zunehmendem Blauanteil ist zu sehen, wie sich Blau, Magenta, Zyan und Weiß durchsetzen.

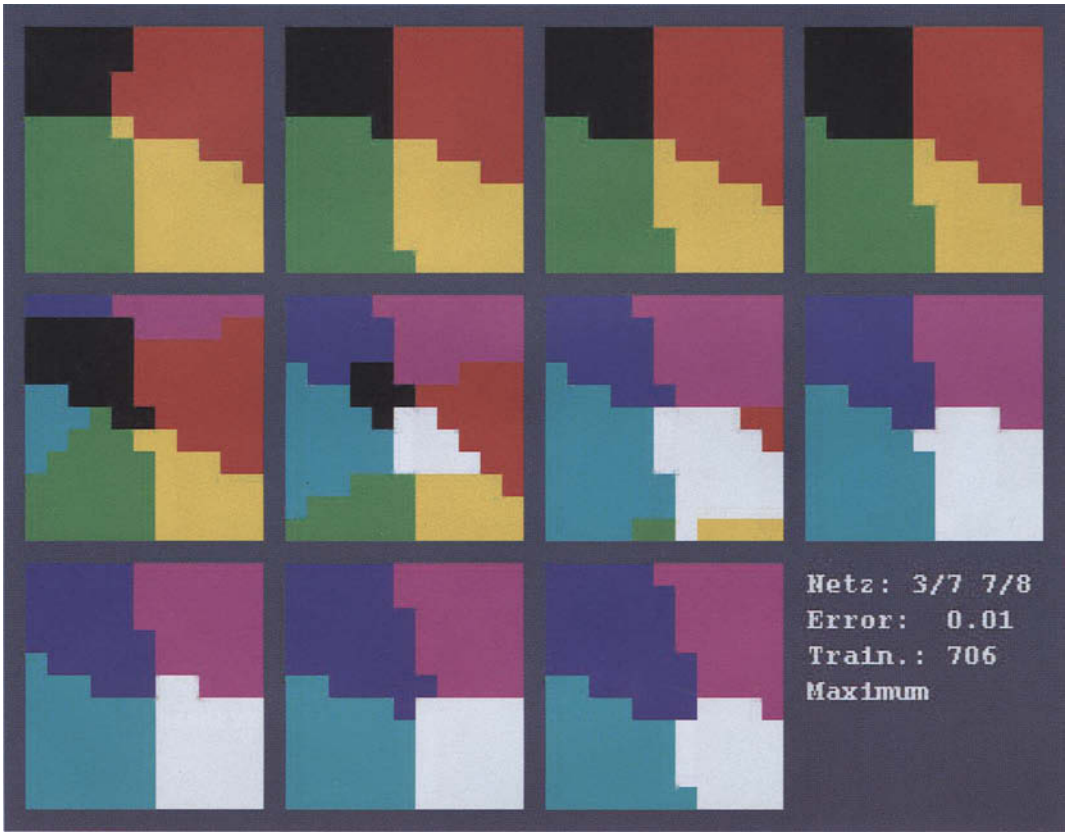


Bild 32.12: Klassifizierung des diskretisierten Farbraums. Hier wurde ein neuronales Netz mit drei Eingabeneuronen, zwei verdeckten Schichten mit jeweils sieben Neuronen und einer Ausgabeschicht mit acht Neuronen verwendet. In der untersten und obersten Ebene ist die Aufteilung auf die Farben Schwarz/Rot/Gelb/Grün und Blau/Magenta/Weiß/Zyan nicht mehr so symmetrisch wie im vorhergehenden Bild. Auch die Übergänge in den dazwischen liegenden Ebenen sind komplexer geworden.

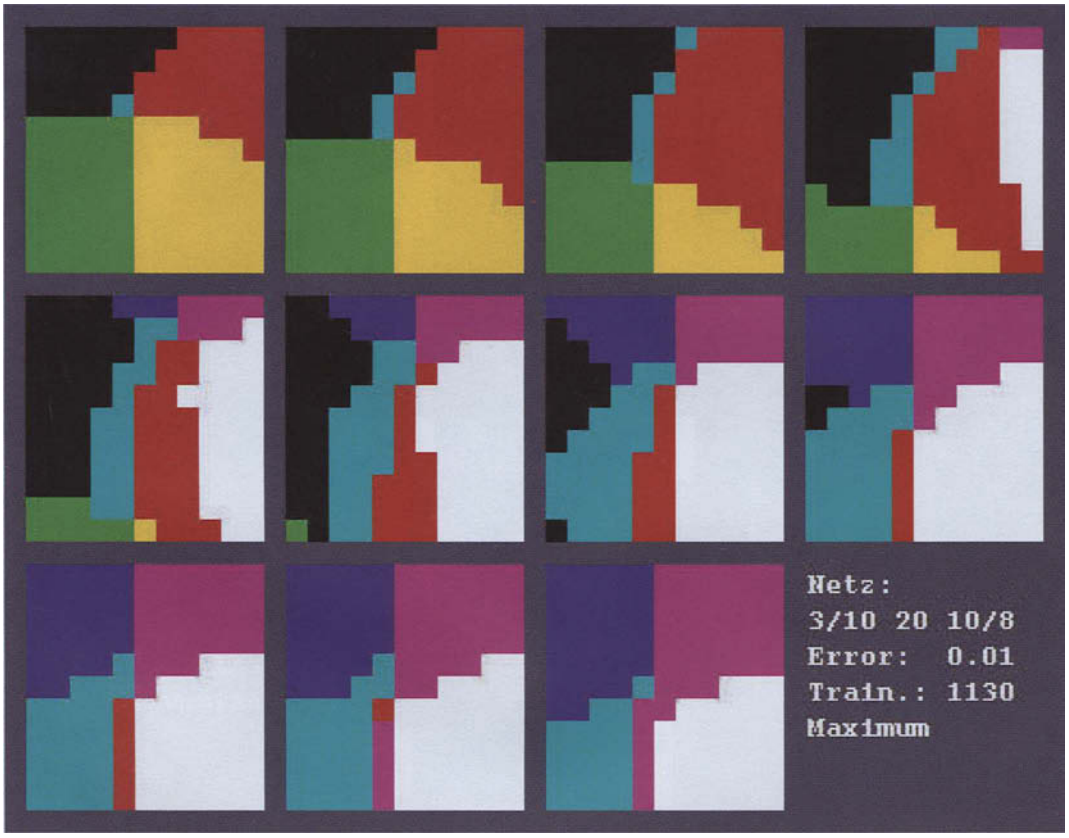


Bild 32.13: Klassifizierung des diskretisierten Farbraums. Hier wurde ein neuronales Netz mit drei Eingabeneuronen, drei verdeckten Schichten mit 10/20/10 Neuronen und einer Ausgabeschicht mit acht Neuronen verwendet. Hier zieht sich z.B. die Farbe Zyan bis zur untersten Ebene durch. Die Farbe Rot tendiert ab der vierten Ebene zum mittleren Bereich des Farbraums. Auch die Übergänge zwischen den anderen Farben entsprechen nicht der intuitiven Vorstellung, wie die Aufteilung sein sollte. Das liegt daran, dass ein Netz mit zu vielen Neuronen verwendet wurde.

Klasse	Anfangszeile/spalte	Endzeile/spalte	Anzahl der Vektoren
Weiss	254/387	301/419	1584
Gelb	89/244	127/279	1404
Hellblau	122/370	142/392	483
Rot	258/230	295/278	1862
Grün	293/ 48	328/ 65	648

Nach Maßgabe der dreidimensionalen Eingabevektoren und der fünf Ausgabeklassen wurde ein Netz mit drei Neuronen in der Eingabeschicht und fünf Neuronen in der Ausgabeschicht verwendet. Das Netz hatte eine verdeckte Schicht mit sieben Neuronen.

Zum Training des Netzes wurden aus den Trainingsgebieten jeweils 20 Farbvektoren pro Klasse, also insgesamt 100 Farbvektoren ausgewählt. Es wurde so lange trainiert, bis alle 100 Eingabevektoren richtig erkannt wurden.

Zur apriori-Beurteilung der Güte der Klassifizierung wurden die Trainingsgebiete, also die jeweils 20 ausgewählten und die restlichen, nicht für das Training des Netzes verwendeten Farbvektoren klassifiziert. Das Ergebnis dieses Schritts ist in folgender Tabelle zusammengefasst:

Klasse	Weiss	Gelb	Hellblau	Rot	Grün
Weiss	100.0%	0.0%	0.0%	0.0%	0.0%
Gelb	0.0%	100.0%	0.0%	0.0%	0.0%
Hellblau	0.0%	0.0%	100.0%	0.0%	0.0%
Rot	0.0%	0.1%	0.0%	99.9%	0.0%
Grün	0.0%	0.0%	0.0%	0.0%	100.0%

Diese Tabelle besagt z.B., dass bei der Klassifizierung der Farbvektoren der Klasse „Rot" 0.1% fälschlicherweise zur Klasse „Gelb" und die restlichen 99.9% richtig zur Klasse „Rot" klassifiziert wurden. Dieses Ergebnis zeigt einen guten Generalisierungseffekt, da ja die Farbvektoren eines Trainingsgebietes in den einzelnen Farbkanälen durchaus streuten.

Das Ergebnis zeigt Bild 32.14-b. Die Klassen wurden durch den mittleren Farbton ihrer Trainingsgebiete dargestellt. Da keine Zurückweisungsklasse eingeführt worden war, wurden alle Farbvektoren einer der fünf Farbklassen zugeordnet. Das bedeutet z.B., dass der dunkle Hintergrund des roten Apfels zur Klasse „Grün" (Eichenblatt) klassifiziert wurde.

Abschließend sei nochmals darauf hingewiesen, dass weitere Anwendungen mit neuronalen Netzen in Kapitel 14 zu finden sind.

(a)

(b)

Bild 32.14: (a) RGB-Farbbild als Testbeispiel für die Farbklassifizierung mit einem neuronalen Netz. (b) Ergebnis der Klassifizierung des Farbbildes. Die Farbklassen sind durch die Mittelwerte der Trainingsgebiete codiert. Es wurde keine Zurückweisungsklasse definiert.

Kapitel 33

Segmentierung mit Fuzzy Logic

33.1 Anwendungen

In diesem Kapitel wird ein unüberwachter Klassifikator vorgestellt, der eine Menge von Merkmalsvektoren einer vordefinierten Anzahl von Klassen zuordnet. Die Lage der *cluster*-Zentren im Merkmalsraum wird, ausgehend von einer initialen Verteilung, in mehreren Iterationsschritten so verändert, dass ein Minimalisierungskriterium erfüllt wird. Als Ergebnis wird zu jedem Merkmalsvektor angegeben, wie groß seine Zugehörigkeit zu den verschiedenen Klassen ist. Im Rahmen eines Systems, das mit der Modellierung von vagem Wissen operiert, kann diese Zugehörigkeitsaussage mit weiteren auf andere Weise gefundenen Aussagen unscharf verknüpft werden.

33.2 Grundlagen: Fuzzy Logic

33.2.1 Einführende Beispiele

In diesem Abschnitt werden die wesentlichen Grundlagen der *fuzzy logic* kurz zusammengefasst. Für den mathematisch interessierten Leser sei auf weitere einführende und vertiefende Literatur, etwa [Zimm92], [Boeh93] oder [Kosk92] verwiesen.

Was heißt „*fuzzy*"? Ein Englischlexikon gibt die folgenden Bedeutungen an: fusselig, faserig, kraus, verschwommen, trüb. Im Sinne der *fuzzy logic* wäre das Wort „*fuzzy*" eher in Richtung „ungenau", „unscharf" oder „unpräzise" zu interpretieren. Aus diesem Grund lautet die deutsche Übersetzung auch „Unscharfe Logik" oder „Unscharfe Mengenlehre". Aber auch die Bezeichnungen „Fuzzy Logik" oder *fuzzy logic* haben sich im deutschen Sprachgebrauch eingebürgert.

Im alltäglichen Leben werden häufig Begriffe verwendet, die sich nicht exakt abgrenzen lassen. Beispiele dazu sind:

- Das Wetter ist schön.
- Es ist warm.

- Ein Auto kostet viel und hat einen hohen Benzinverbrauch.
- Wenn ein Steak zu lange gebraten wird, ist es zäh.

Das sind alles vage Formulierungen, bei denen man sich nicht auf einen konkreten „Wert" festlegt. Außerdem kann sich je nach Standpunkt des Verwenders dieser Formulierungen eine unterschiedliche Interpretation verbergen: Der Benzinverbrauch eines Autos beispielsweise ist eine sehr subjektive Angelegenheit. Bei einem Mittelklassewagen würde man derzeit von einem angemessenen Verbrauch bei etwa 8 *Litern pro 100 km* sprechen. Für einen Wagen der gehobenen Klasse wäre das ein eher niederer Verbrauch. Dazu kommt noch eine weitere Unschärfe: In einigen Jahren könnte man bei einem Mittelklassewagen durchaus 5-6 *Liter pro 100 km* als angemessen ansehen. Man hat hier also zwei unscharfe Formulierungen, nämlich „angemessener Verbrauch" und „derzeit".

Diese sprachliche Ungenauigkeit ist jedoch kein Nachteil, sondern ermöglicht es uns, in Situationen, in denen nur unvollständige oder sogar teilweise widersprüchliche Informationen vorliegen, eine Entscheidung zu fällen.

Die *fuzzy logic* ist eine exakte mathematische Theorie, mit der versucht wird, nicht exakten Begriffen gerecht zu werden. Dabei stellt sie andere Logikkalküle nicht in Frage, sondern bildet eine Verallgemeinerung, aus der sich z.B. die klassische Aussagenlogik als Sonderfall ableiten lässt.

An dieser Stelle sei auch noch ein Hinweis auf die Wahrscheinlichkeitstheorie eingefügt: Das Ereignis A=„Würfeln einer Sechs" tritt mit einer mathematischen Wahrscheinlichkeit $p(A) = \frac{1}{6}$ ein. Wenn die Voraussetzung richtig ist, dass es sich um einen symmetrischen, idealisierten Würfel handelt, bei dem alle sechs möglichen Seiten gleichwahrscheinlich oben liegen, ist die obige Aussage exakt richtig. In der Praxis ist das aber nicht erfüllbar, da es keinen idealen Würfel gibt. Man könnte nun in vielen Versuchen näherungsweise die Wahrscheinlichkeiten ermitteln. Ein Ergebnis könnte dabei sein, dass z.B. die Zahl sechs mit einer Wahrscheinlichkeit von $\frac{13}{60}$ erscheint. Aber dies ist wiederum nur eine Näherung. Eine unscharfe Aussage wäre, dass bei einem Würfel die Zahl sechs „ziemlich sicher" mit der Wahrscheinlichkeit $\frac{13}{60}$ oben liegt. Hier ist „ziemlich sicher" eine unscharfe Aussage, die mit bestimmten Werten zu belegen ist, die angeben, was unter „ziemlich sicher" zu verstehen ist.

Unscharfen Begriffen der Art „ziemlich sicher", „eher klein", „nahe bei eins", „normal", „hohes Fieber", usw. müssen somit Bereiche der jeweiligen Skala zugeordnet werden, und es muss angegeben werden, wie stark die Zugehörigkeit der möglichen Werte zu dem unscharfen Begriff ist. Diese Zugehörigkeit nimmt Werte zwischen 0 und 1 an, wobei festgelegt wird, dass 0 als „sicher nicht dazu gehörig" und 1 als „sicher dazu gehörig" verstanden wird.

Dazu ein Beispiel: Der „angemessene Bezinverbrauch" für Mittelklassewagen könnte etwa so beschrieben werden:

Literverbrauch pro 100 *km*	<6.5	6.5	7.0	7.5	8.0	8.5	9.0	9.5	>9.5
Zugehörigkeit	0.0	0.2	0.5	0.9	1.0	0.9	0.5	0.2	0.0

Die Tabelle ist so zu lesen, dass z.B. ein konkreter Verbrauch von 7.5 *Litern pro 100 km* mit einer Zugehörigkeit von 0.9 als angemessen für Mittelklassewagen anzusehen ist, während ein Verbrauch, der größer als 9.5 *Liter pro 100 km* ist, sicher nicht als angemessen bezeichnet werden kann. Natürlich müssen die Werte nicht immer, wie in diesem Beispiel, diskret sein. Sie können auch kontinuierlich und die Zugehörigkeiten funktional angegeben werden.

Nun noch ein paar Worte zur Geschichte der *fuzzy logic*. Die ersten Veröffentlichungen erschienen 1965 von Lofti Asker Zadeh. Er gehörte dem Department of Electrical Engineering and Electronics Research Laboratory der Berkeley University an. Die Weiterentwicklung erfolgte auch in Europa und Japan. Heute ist die *fuzzy logic* ein Teilbereich der Mathematik mit einer großen Zahl von Publikationen. Praktische Anwendungen gibt es in der Regelungstechnik, bei Expertensystemen, in der digitalen Bildverarbeitung und Mustererkennung und, wenn der Reklame zu trauen ist, bei Waschmaschinen und Videokameras.

Nach dieser Einführung wird im folgenden die *fuzzy logic* etwas genauer definiert. Die Darstellung und Notation orientiert sich an [Boeh93].

33.2.2 Definitionen und Erläuterungen

Es sei G eine Grundmenge und μ_A eine Funktion von G in das Intervall $[0,1]$ der reellen Zahlenachse:

$$\mu_A : G \rightarrow [0,1]. \tag{33.1}$$

Die Menge

$$A = \{(x, \mu_A(x)) | x \in G\} \tag{33.2}$$

heißt *fuzzy-Menge* (*unscharfe Menge*, *fuzzy set*) über G.

Die Funktion μ_A heißt *Mitgliedsgrad-* oder *Zugehörigkeitsfunktion*. Die Angabe der Mitgliedsgradwerte von $\mu_A(x)$ ist nicht Aufgabe der *fuzzy logic*, sie werden vielmehr nach Erfahrung, sprachlicher Gewohnheit usw. festgelegt. Die Werte selbst können wieder unscharf sein, was zu *fuzzy*-Mengen höherer Ordnung führt. Dies ist auch schon in Beispielen von Abschnitt 33.2.1 angeklungen.

Eine klassische Menge M auf einer Grundmenge G kann als Sonderfall einer *fuzzy*-Menge abgeleitet werden: Die Mitgliedsgradwerte für alle $x \in M$ sind 1, während die Mitgliedsgradwerte für $x \notin M$ gleich 0 sind. Die Funktion μ_A ist dann gleich mit der charakteristischen Funktion μ_M der Menge M.

Fuzzy-Mengen dieser Art werden häufig auch als *linguistische Variable* bezeichnet. Ein Beispiel soll eine Möglichkeit der Notation verdeutlichen: Die *fuzzy*-Menge (linguistische Variable) HOHES FIEBER besitzt die Grundmenge $G = \{37, 38, 39, 40, 41, 42\}$ (Maßeinheit: *Grad Celsius*). Die *fuzzy*-Menge HOHES FIEBER ist:

$$\text{HOHES FIEBER} = \{(39, 0.5), (40, 0.8), (41, 1.0), (42, 1.0)\} \tag{33.3}$$

Die Zugehörigkeitsfunktion $\mu_{\text{HOHES FIEBER}}$ lautet tabellarisch:

x	37	38	39	40	41	42
$\mu_{\text{HOHES FIEBER}}(x)$	0.0	0.0	0.5	0.8	1.0	1.0

Es sei darauf hingewiesen, dass bei *fuzzy*-Mengen in der Notation von (33.3) Elemente, deren Mitgliedsgradwert 0 ist, weggelassen werden.

In diesem Beispiel wurden zwei übliche Notationen für endliche *fuzzy*-Mengen benützt: Die Aufzählung aller Paare $(x, \mu_A(x))$ mit $x \in G$ und Darstellung in Vektor- oder Tabellenform.

Eine andere Notation für eine endliche *fuzzy*-Menge A über der Grundmenge $G = \{x_1, x_2, ..., x_n\}$ ist:

$$A = \frac{\mu_A(x_1)}{x_1} + \frac{\mu_A(x_2)}{x_2} + ... + \frac{\mu_A(x_n)}{x_n} = \sum_{i=1}^{n} \frac{\mu_A(x_i)}{x_i} = \sum_{i=1}^{n} \mu_A(x_i)/x_i. \tag{33.4}$$

Es sei bemerkt, dass die Bruchstriche, Pluszeichen und Summenzeichen hier nicht die übliche Bedeutung wie in der Mathematik haben, sondern zur formalen Darstellung dienen.

Für unendliche *fuzzy*-Mengen wird, angelehnt an obige Notation, das Integralzeichen verwendet. Dazu ein weiteres Beispiel aus [Boeh93]:

Die Grundmenge G sei die Menge der reellen Zahlen. Die *fuzzy*-Menge A=NAHE NULL mit der Mitgliedsgradfunktion

$$\mu_{\text{NAHE NULL}}(x) = \frac{1}{1+x^2} \tag{33.5}$$

könnte dann so geschrieben werden:

$$A = \text{NAHE NULL} = \int_{-\infty}^{+\infty} \frac{1}{1+x^2}/x. \tag{33.6}$$

Häufig hat der Graph der Mitgliedsgradfunktion einen triangulären oder trapezförmigen Verlauf. Diese Mitgliedsgradfunktion und damit die *fuzzy*-Menge kann dann durch mehrere Terme definiert werden. Als Beispiel dient die Funktion von Bild 33.1.

Die Grundmenge ist das Intervall $G = [a, b]$ und es gelte $a \leq c_1 < m < c_2 \leq b$. Dann lässt sich die Mitgliedsgradfunktion wie folgt beschreiben:

$$\mu_A(x) = \begin{cases} \max\left(0, \frac{x-c_1}{m-c_1}\right) & \text{für} \quad x \in [a, m] \\ \max\left(0, \frac{c_2-x}{c_2-m}\right) & \text{für} \quad x \in [m, b] \end{cases} \tag{33.7}$$

Diese *fuzzy*-Menge könnte etwa als folgende linguistische Variable NORMALE TEMPERATUR interpretiert werden: „Der Sollwert ist m (*Grad Celsius*). Von a (z.B. $a = -273$

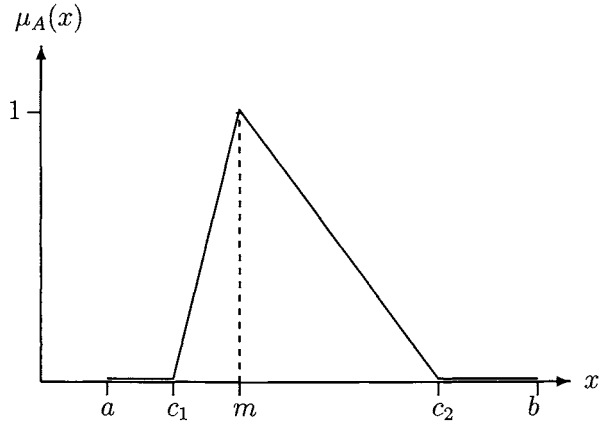


Bild 33.1: Beispiel für eine *fuzzy*-Menge mit triangulärer Mitgliedsgradfunktion.

Grad) bis c_1 ist die Zugehörigkeit 0, dann nimmt sie linear bis zum Wert m zu. Danach fällt die Zugehörigkeit linear bis zum Wert c_2 und ist von c_2 bis b wieder 0."

Oft müssen die Elemente einer unscharfen Menge bezüglich ihrer Zugehörigkeit bewertet werden: Es interessieren z.B. nur solche Elemente aus der Grundmenge G, deren Zugehörigkeit zur unscharfen Menge A größer ist als ein bestimmter Schwellwert α. Man spricht dann von der *α-Niveaumenge* (*α-level set*, *α-cut*, *α-Schnitt*):

$$A_\alpha = \{x \in G | \mu_A(x) \geq \alpha\} \tag{33.8}$$

Wird in (33.8) statt $\mu_A(x) \geq \alpha$ gefordert, dass gilt $\mu_A(x) > \alpha$, so spricht man von der *strengen α-Niveaumenge*. Die α-Niveaumengen sind klassische (scharfe) Mengen. Dazu ein Beispiel: Die linguistische Variable GS (geeigneter Schwellwert zur Binarisierung eines Bildes) sei:

$$\mathrm{GS} = \tfrac{0.1}{140} + \tfrac{0.4}{141} + \tfrac{0.7}{142} + \tfrac{0.9}{143} + \tfrac{1.0}{144} + \tfrac{0.9}{145} + \tfrac{0.7}{146} + \tfrac{0.4}{147} + \tfrac{0.1}{148}$$

Beispiele für α-Niveaumengen sind dann: $\mathrm{GS}_{0.4} = \{141, 142, 143, 144, 145, 146, 147\}$ oder $\mathrm{GS}_{1.0} = \{144\}$.

Zwei weitere Begriffe: Die *Höhe* und die *Normalisierung* einer *fuzzy*-Menge. Die Höhe ist

$$h(A) = \sup_{x \in G} \mu_A(x). \tag{33.9}$$

Dabei ist sup das Supremum (die kleinste obere Schranke) des Wertebereichs von μ_A. Falls es einen größten $\mu_A(x)$-Wert gibt (z.B. bei endlichem Wertevorrat), ist $\sup \mu_A(x) = \max \mu_A(x)$. Die *fuzzy*-Menge A heißt normalisiert, wenn gilt:

$$h(A) = 1. \tag{33.10}$$

Jetzt werden einige Beziehungen und Verknüpfungen von *fuzzy*-Mengen definiert:

Zunächst die *Gleichheit* von zwei *fuzzy*-Mengen und der Begriff der *Teilmenge*. Dazu werden zwei *fuzzy*-Mengen A und B über derselben Grundmenge G betrachtet:

$$A = B \Longleftrightarrow \mu_A(x) = \mu_B(x) \text{ für alle } x \in G. \tag{33.11}$$

und

$$A \subset B \Longleftrightarrow \mu_A(x) \leq \mu_B(x) \text{ für alle } x \in G. \tag{33.12}$$

Damit können auch Begriffe wie *Reflexivität* $(A = A)$, *Symmetrie* $(A = B \Longrightarrow B = A)$ und *Transitivität* $(A = B$ und $B = C \Longrightarrow A = C)$ für *fuzzy*-Mengen definiert werden.

Als Beispiel sei nochmals die *fuzzy*-Menge GS (geeigneter Schwellwert zur Binarisierung eines Bildes) von oben betrachtet. Wenn man eine *fuzzy*-Menge GGS „gut geeigneter Schwellwert ..."

$$\text{GGS} = \tfrac{0.9}{143} + \tfrac{1.0}{144} + \tfrac{0.9}{145}$$

festlegt, dann gilt GGS $\subset$ GS. Außerdem sind GS und GGS normalisiert, da $h(\text{GS}) = h(\text{GGS}) = 1$.

Sprachlich werden unscharfe Aussagen oft verstärkt (geeignet, gut geeignet, sehr gut geeignet) oder abgeschwächt (groß, nicht groß, eher klein). Diese Sprechweisen lassen sich auch auf *fuzzy*-Mengen übertragen. Dazu verwendet man Begriffe wie *Konzentration* und *Dilatation*. Da die Mitgliedsgradfunktion $\mu_A(x)$ nur Werte aus dem Intervall $[0, 1]$ annehmen darf, werden durch Potenzieren mit einem Exponenten $n > 0$ die Mitgliedsgradwerte verkleinert und durch Radizieren vergrößert.

Der *Konzentrations-Operator* kon_n und der *Dilatations-Operator* dil_n für eine *fuzzy*-Menge $A = \{(x, \mu_A(x)) | x \in G\}$ sind somit wie folgt definiert:

$$\text{kon}_n A = \{(x, \mu_A(x)^n) | x \in G\} \tag{33.13}$$

und

$$\text{dil}_n A = \{(x, \sqrt[n]{\mu_A(x)}) | x \in G\}. \tag{33.14}$$

Es gilt offenbar für $x \in G$ und $n > 1$:

$$\text{kon}_n A \subset A \subset \text{dil}_n A \tag{33.15}$$

Wie die Verstärkung und die Abschwächung von linguistischen Variablen durchzuführen ist, ist nicht die Aufgabe der *fuzzy logic*. Vielmehr muss eine passende Modellierung in jedem konkreten Anwendungsfall festgelegt werden.

Weitere wichtige Mengenoperationen sind der *Durchschnitt*, die *Vereinigung* und das *Komplement*. Auch hier werden wieder Definitionen verwendet, die mit der klassischen Mengenlehre verträglich sind. Betrachtet werden zwei *fuzzy*-Mengen A und B auf der Grundmenge G. Dann ist der Durchschnitt:

$$A \cap B : \mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x)), \tag{33.16}$$

die Vereinigung

$$A \cup B : \mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x)), \tag{33.17}$$

und das Komplement

$$\bar{A} : \mu_{\bar{A}}(x) = 1 - \mu_A(x). \tag{33.18}$$

Anhand der Definitionen ist leicht zu sehen, wie die Verträglichkeit mit der scharfen Mengenlehre gewährleistet ist. In der Literatur werden auch noch andere Definitionen für diese Operatoren diskutiert.

Die Wirkungsweise dieser Mengenperatoren ist nicht immer mit dem sprachlichen Verständnis im Einklang. Dazu ein Beispiel: Bei der Auswahl und Implementierung von Bildverarbeitungsoperatoren ist es wichtig, mit welchem Aufwand sie zu realisieren und wie ihre Ergebnisse zu beurteilen sind. Auf einer Menge zur Verfügung stehender Bildverarbeitungsoperatoren $G = \{O_1, O_2, ..., O_n\}$ zu einer bestimmten Problemstellung könnte man somit die linguistischen Variablen ANGEMESSEN in der Realisierung und GUT in der Qualität definieren. Für einen bestimmten Bildverarbeitungsoperator O_i gelte nun:

$$\mu_{\text{ANGEMESSEN}}(O_i) = 0.2 \text{ und } \mu_{\text{GUT}}(O_i) = 0.9,$$

was bedeutet, dass er zwar ziemlich gute Ergebnisse liefert, aber nicht einfach zu realisieren ist. In der *fuzzy*-Menge ANGEMESSEN $\wedge$ GUT, die man aus dem Durchschnitt der beiden *fuzzy*-Mengen bildet, erhält der Bildverarbeitungsoperator O_i den Wert

$$\mu_{\text{ANGEMESSEN} \cap \text{GUT}}(O_i) = \min(\mu_{\text{ANGEMESSEN}}, \mu_{\text{GUT}}) = \min(0.2, 0.9) = 0.2,$$

d.h. er wird insgesamt ziemlich schlecht bewertet. Legt man fest, dass ein Bildverarbeitungsoperator angemessen ODER gut sein soll, so erhält man in diesem Beispiel einen Wert von 0.9, was intuitiv einer zu guten Bewertung entspricht.

In der Praxis wird man häufig Kompromisse schließen müssen. Es kann sein, dass man einen hohen Implementierungsaufwand in Kauf nimmt, um gute Ergebnisse zu erhalten. Für die Mengenoperatoren auf *fuzzy*-Mengen heißt das, dass man Entscheidungen treffen möchte, die „nicht ganz UND" sind, also Werte liefert, die zwischen min und max liegen.

Mengenoperatoren dieser Art werden *kompensatorische* oder *kombinatorische* Operatoren genannt. Auch hierzu wird ein Beispiel gegeben und darauf hingewiesen, dass in der Literatur eine Vielzahl von verschiedenen Definitionen untersucht wird.

Der *min-max-Kompensations-Operator* $K_\gamma(x, y)$ wird folgendermaßen definiert:

$$K_\gamma(x, y) = \min(x, y)^{1-\gamma} \cdot \max(x, y)^\gamma. \tag{33.19}$$

Der Parameter $\gamma \in [0, 1]$ heißt *Kompensationsgrad* und die Argumente $x, y \in [0, 1]$ stehen für die Mitgliedsgrade. Für $\gamma = 0$ ist $K_0(x, y) = \min(x, y)$ und für $\gamma = 1$ ist $K_1(x, y) = \max(x, y)$.

Für das obige Beispiel sind die Zugehörigkeitswerte des Operators O_i zur *fuzzy*-Menge ANGEMESSEN $\wedge$ GUT für verschiedene Werte des Kompensationsgrads γ in einer Tabelle zusammengestellt:

γ	$K_\gamma(\mu_{\text{ANGEMESSEN}}(O_i), \mu_{\text{GUT}}(O_i))$
0.0	0.200
0.1	0.232
0.2	0.270
0.3	0.314
0.4	0.365
0.5	0.424
0.6	0.493
0.7	0.573
0.8	0.666
0.9	0.774
1.0	0.900

In [Boeh93] oder [Zimm92] werden weitere interessante Aspekte der *fuzzy*-Mengen untersucht, etwa *fuzzy*-Relationen, *fuzzy*-Zahlen oder die *fuzzy*-Aussagenlogik.

Nun noch einige Bemerkungen zum „unscharfen Schließen". Darunter versteht man das Herleiten von nichtpräzisen Folgerungen aus nichtpräzisen Voraussetzungen. Eine umfassende Darstellung dieser Problematik würde den Rahmen dieser kurzen Zusammenfassung bei weitem sprengen, so dass hier nur versucht wird darzustellen, welche Probleme auftreten. Zu Fragen, wie sie mathematisch modelliert werden können, sei auf die schon mehrfach zitierte Literatur verwiesen.

Nach [Boeh93], wo diesen Fragestellungen ein Kapitel am Ende gewidmet ist, müssen an eine Theorie zum unscharfen Schließen einige Forderungen gestellt werden:

- Die in *fuzzy*-Aussagen (z.B. „Das Auto fährt schnell", „Gallo ist intelligenter als Hugo", „Die Temperatur ist zu hoch") enthaltene Unschärfe muss zahlenmäßig darstellbar gemacht werden.

Dazu wird die Möglichkeitsverteilung (Possibilitätsverteilung) einer *fuzzy*-Aussage durch eine *fuzzy*-Menge charakterisiert (eingeschränkt, restringiert). Die Possibilität wird durch die Mitgliedsgradfunktion bestimmt.

- Wenn Schlussfolgerungen abgeleitet werden, so muss der Vorgang der Ableitung einem mathematischen Formalismus gehorchen und berechenbar sein. Typische Schlussfolgerungen sind: „Wenn X und Y dann Z" oder „Wenn X oder Y dann Z". Dabei sind X, Y und Z *fuzzy*-Aussagen, die durch *fuzzy*-Mengen A, B und C dargestellt werden. Es muss somit ein mathematischer Weg gefunden werden, wie aus den *fuzzy*-Mengen A und B die *fuzzy*-Menge C berechnet werden kann.

- Schlussfolgerungen, die auf diese Weise abgeleitet werden, müssen so gut wie möglich mit der Realität übereinstimmen, d.h. dass sie in einem bestimmten Anwendungsgebiet ein Ergebnis liefern, das mit „gesundem Menschenverstand" auch erzielt worden wäre.

Um diesen Forderungen gerecht zu werden, wird in [Boeh93] zunächst der in dieser Zusammenstellung etwas vage verwendete Begriff der *fuzzy*-Variablen oder linguistischen Variablen exakt definiert. Dazu wird eine Possibilitätsverteilung definiert, die auf den *fuzzy*-Mengen aufbaut. Dann werden das Projektionsprinzip und die zylindrische Erweiterung definiert, mit denen Possibilitätsverteilungen eingeschränkt bzw. erweitert werden können.

Die Partikularisation beschäftigt sich mit der Frage, wie man eine Possibilitätsverteilung zu ändern hat, wenn man als neues Wissen erfährt, dass sich die Possibilitätsverteilung einer Subvariablen geändert hat.

Regeln der maximalen und minimalen Restriktion beschäftigen sich mit Ableitungen der Form: Wenn X und Y *fuzzy*-Aussagen sind, wie können dann die *fuzzy*-Aussagen berechnet werden, die durch beide Aussagen beeinflusst werden? Ein Beispiel: X und Y werden durch die *fuzzy*-Mengen A und B dargestellt. Wie soll dazu die *fuzzy*-Menge berechnet werden, die aus dem kartesischen Produkt $A \times B$ besteht?

Aufbauend darauf werden „Wenn-dann-Inferenzregeln" (Wenn X dann Y) und der „generalisierte Modus ponens" (X ist ...; Wenn X dann Y; Was ist Y?) modelliert.

Beim Aufbau einer *fuzzy-logic*-Applikation ist somit praktisch wie folgt vorzugehen:

- Im ersten Schritt muss festgelegt werden, welcher Sachverhalt fuzzifiziert werden soll. Bei einer Bildverarbeitungsanwendung könnte man sich z.B. entscheiden, die Schwellwertbestimmung einer Binarisierung zu fuzzifizieren.

- Als Nächstes ist festzulegen, wie die Zugehörigkeitsfunktionen definiert werden. Hier spielt oft die persönliche Erfahrung oder Bewertung der Sachverhalte eine wichtige Rolle. Bei Anwendungen in der digitalen Bildverarbeitung und Mustererkennung können die Werte der Mitgliedsgradfunktionen häufig aus den zu verarbeitenden Daten abgeleitet werden. Bei einer dynamischen Schwellwertbestimmung könnte man z.B. den Schwellwert für einen Bildausschnitt durch eine *fuzzy*-Menge darstellen, deren Mitgliedsgradwerte aus einer bestimmten Bildumgebung berechnet werden.

- Des Weiteren ist festzulegen, wie die Verknüpfungen der *fuzzy*-Mengen realisiert werden sollen. Hier muss man z.B. definieren, welche Operatoren man für UND und ODER verwendet. Auch die Wahl bestimmter kompensatorischer Operatoren fällt in diesen Bereich.
- Der nächste Schritt ist das Aufstellen von problemrelevanten Regeln. Hier muss also das Wissen der jeweiligen Problemstellung modelliert und für die Anwendung in ein geeignetes Regelsystem gepackt werden.
- Der letzte Schritt ist die Defuzzifizierung: Hier muss festgelegt werden, wie die Abbildung der abgeleiteten *fuzzy*-Ergebnisse auf reale Werte geschehen soll, die dann z.B. zur Steuerung einer Anlage verwendet werden können.

33.3 Fuzzy Klassifikator

In diesem Abschnitt wird ein Klassifikator beschrieben, der die Aufteilung des N-dimensionalen Merkmalsraums mit einem *fuzzy*-Ansatz durchführt. Die Grundlagen der numerischen Klassifikation sind in den Kapiteln 23 und 31 zusammengestellt. Im Folgenden sind nochmals kurz die relevanten Größen mit ihren Bezeichnungen zusammengefasst:

t	Anzahl der *cluster*,
N	Dimension des Merkmalsraums,
M	Anzahl der Merkmalsvektoren,
$\mathbf{g}_i \in R^N$	i-ter Merkmalsvektor, i=0,...,M-1
$\mathbf{z}_j \in R^N$	j-tes *cluster*-Zentrum, j=0,...,t-1.

Zusätzlich werden noch eingeführt:

$w_{ij} = \mu_j(\mathbf{g}_i)$	Mitgliedsgradwert des Merkmalsvektors $\mathbf{g}_i$ zum *cluster* j,
m	ein Parameter, der den *fuzzy*-Grad steuert.

Gesucht sind t *cluster*-Zentren $\mathbf{z}_j$, die so im Merkmalsraum verteilt liegen, dass der Ausdruck

$$\sum_{i=0}^{M-1} \sum_{j=0}^{t-1} w_{ij}^m d_{ij} \tag{33.20}$$

minimal wird. Eine Randbedingung ist dabei

$$\sum_{j=0}^{t-1} w_{ij} = 1 \tag{33.21}$$

mit $w_{ij} > 0$, für $i = 0, ..., M-1$ und $j = 0, ..., t-1$. Die Größe d_{ij} ist der euklidische Abstand des Merkmalsvektors $\mathbf{g}_i$ zum *cluster*-Zentrum $\mathbf{z}_j$ im Merkmalsraum:

$$d_{ij} = \sqrt{(\mathbf{g}_i - \mathbf{z}_j)^T (\mathbf{g}_i - \mathbf{z}_j)}. \tag{33.22}$$

Die Optimierung läuft hier über die Mitgliedsgradwerte und die Lage der *cluster*-Zentren im $N-$dimensionalen Merkmalsraum, d.h. dass bei jedem Iterationsschritt die Mitgliedsgradwerte und die *cluster*-Zentren neu berechnet werden, bis ein Abbruchkriterium (siehe unten) erfüllt ist.

Wo liegt hier der *fuzzy*-Ansatz? Bei einem herkömmlichen („scharfen") Klassifikator wird als Ergebnis jeder Merkmalsvektor $\mathbf{g}_i$ genau einem *cluster*-Zentrum $\mathbf{z}_j$ und damit genau einer Klasse zugeordnet. Der unscharfe Klassifikator ordnet einen Merkmalsvektor allen *cluster*-Zentren zu, jedoch mit unterschiedlichen Mitgliedsgradwerten. Mit der Nomenklatur und der Notation der *fuzzy logic* heißt das, dass jedem Merkmalsvektor $\mathbf{g}_i$ eine unscharfe Menge MV_i zugeordnet wird:

$$MV_i = \frac{w_{i0}}{\vec{z_0}} + \frac{w_{i1}}{\vec{z_1}} + ... + \frac{w_{it-1}}{\vec{z_{t-1}}}, i = 0, ..., M-1. \tag{33.23}$$

Bei praktischen Anwendungen kann die Zugehörigkeit zu den *cluster*-Zentren noch durch weitere Restriktionen eingeschränkt werden. Dazu drei Beispiele:

- Jeder Merkmalsvektor $\mathbf{g}_i$ kann nur maximal c_1 *cluster*-Zentren mit einem Mitgliedsgradwert größer Null zugeordnet werden.
- Falls für die Distanz eines Merkmalsvektors $\mathbf{g}_i$ zu einem *cluster*-Zentrum $\mathbf{z}_j$ gilt $d_{ij} > c_2$, wird $w_{ij} = 0$ gesetzt.
- Der Mitgliedsgradwert eines Merkmalsvektors $\mathbf{g}_i$ zum *cluster*-Zentrum $\mathbf{z}_j$ wird zu $w_{ij} = 0$ gesetzt, falls $w_{ij} < c_3$ gilt.

Der folgende Algorithmus beschreibt den *fuzzy*-Klassifikator. Die zusätzliche Restriktion wird in diesem Algorithmus nach der dritten Variante implementiert.

A33.1: Fuzzy-Klassifikator.

Voraussetzungen und Bemerkungen:

◇ Die Voraussetzungen und Bezeichnungen sind in der Einführung zu diesem Abschnitt zusammengestellt.

Algorithmus:

(a) Festlegen von t anfänglichen *cluster*-Zentren $\mathbf{z}_j$, $j = 0, ..., t-1$. Diese *cluster*-Zentren könnten z.B. das Ergebnis eines Minimum-Distance-Cluster-Verfahrens sein (Abschnitt 31.3). Sie können aber auch „aus der Erfahrung" oder gleichverteilt im Merkmalsraum festgelegt werden.

(b) Iteration:

(ba) Berechnung der Distanzen d_{ij}^2 für $0 \leq i \leq M-1$ und $0 \leq j \leq t-1$.

$d_{ij}^2 = (\mathbf{g}_i - \mathbf{z}_j)^T(\mathbf{g}_i - \mathbf{z}_j)$.

(bb) Berechnung der nicht normierten Mitgliedsgradwerte v_{ij} für $0 \leq i \leq M-1$ und $0 \leq j \leq t-1$:

(bba) Falls $d_{ij}^2 = 0$: $v_{ij} = 1$ und $v_{ik} = 0$, $0 \leq k \leq t-1, k \neq j$.

(bbb) Falls $d_{ij}^2 \neq 0$:

$$v_{ij} = \frac{1}{\sum_{k=0}^{t-1} \left(\frac{d_{ij}}{d_{ik}}\right)^{\frac{2}{m-1}}}, 0 \leq k \leq t-1, k \neq j.$$

(bc) Berechnung der neuen *cluster*-Zentren:

$$\mathbf{z}_j = \sum_{i=0}^{M-1} v_{ij}^m \mathbf{g}_i \Big/ \sum_{i=0}^{M-1} v_{ij}^m, 0 \leq j \leq t-1.$$

(bd) Berechnung des Abbruchkriteriums. Hier sind zwei verschiedene Varianten möglich:

$$R_\gamma = \Big[\sum_{j=0}^{t-1} \sum_{n=0}^{N-1} |z_{jn}(s) - z_{jn}(s-1)|^\gamma\Big]^{\frac{1}{\gamma}} < \varepsilon,$$

oder

$$Q_\gamma = \Big[\sum_{i=0}^{M-1} \sum_{j=0}^{t-1} |v_{ij}(s) - v_{ij}(s-1)|^\gamma\Big]^{\frac{1}{\gamma}} < \varepsilon,$$

Dabei steht s für den s-ten Iterationsschritt. ε ist ein vorzugebender kleiner Wert. Mit γ können unterschiedliche Normen verwendet werden: $\gamma = 1$ ist die $l1$-Norm, $\gamma = 2$ die euklidische Norm und $\gamma = \infty$ die Tschebyscheff-Norm.

(c) Schwellwertbildung bei den Mitgliedsgradwerten: Falls $v_{ij} < c_3$: $v_{ij} = 0$, $0 \leq i \leq M-1$ und $0 \leq j \leq t-1$.

(d) Normalisierung der Mitgliedsgradwerte:

$$w_{ij} = \frac{v_{ij}}{\sum_{k=0}^{t-1} v_{ik}}$$

für $0 \leq i \leq M-1$ und $0 \leq j \leq t-1$.

Ende des Algorithmus

Wie oben bereits dargestellt, liegen nach Ablauf dieses Algorithmus M *fuzzy*-Mengen vor, und zwar zu jedem Merkmalsvektor eine. Die Möglichkeiten der Weiterverarbeitung sind unterschiedlich. Wenn z.B. mit dem Klassifizierungsschritt die Verarbeitung beendet ist, kann eine Defuzzifizierung vorgenommen werden, indem man etwa einen Merkmalsvektor derjenigen Klasse zuordnet, zu der er den größten Mitgliedsgradwert hat.

Der *fuzzy*-Ansatz kommt aber dann erst richtig zum Tragen, wenn die so erhaltenen unscharfen Mengen im weiteren Verlauf der Verarbeitung mit anderen unscharfen Mengen verknüpft werden. So könnten in einem zum *fuzzy*-Klassifikator parallel verlaufenden Arbeitsschritt, der jedoch ganz andere Merkmale zur Klassifizierung verwendet, auch *fuzzy*-Mengen als Ergebnis auftreten. Die letztliche Zuordnung erfolgt dann auf Grund einer UND-Verknüpfung der beiden *fuzzy*-Mengen.

Nach der Darstellung soll das Verfahren anhand von zwei Beispielen getestet werden. Das erste Beispiel kann bei der Implementierung als Vergleichsbeispiel verwendet werden. Es wird ein eindimensionaler Merkmalsraum verwendet. Die Eingabedaten und die Ergebnisse sind in folgender Tabelle zusammengestellt:

Eingabe-Merkmalsvekt.	-5.00	-4.00	1.00	2.00	5.00	6.00
Anf. *cluster*-Zentren	-4.00	5.00				
Iterierte *cluster*-Zentren	-4.35	3.76				
Mitgliedsgradwerte						
w_{i0}	0.99455	0.99795	0.21002	0.07126	0.01729	0.04475
w_{i1}	0.00545	0.00205	0.78998	0.92874	0.98271	0.95525

Der *fuzzy*-Parameter m wurde hier zu $m = 2$ gesetzt. Als Abbruchkriterium wurde Q_γ mit $\gamma = 2$, also die euklidische Norm, verwendet. Es waren neun Iterationen notwendig, um eine Fehlerschranke von $\varepsilon < 0.001$ zu erreichen. Eine Schwellwertbildung der Mitgliedsgradwerte wurde nicht durchgeführt.

Als zweites Beispiel wurden dreidimensionale Farbvektoren klassifiziert. Insgesamt wurden 16 Merkmalsvektoren verwendet:

Nummer	Rotanteil	Grünanteil	Blauanteil
0	220	20	14
1	218	18	23
2	244	6	8
3	223	28	35
4	22	220	35
5	65	238	12
6	14	228	4
7	12	251	18
8	12	28	235
9	23	14	220
10	2	31	242
11	18	22	249
12	180	85	22
13	93	141	31
14	12	185	242
15	164	143	156

Bei genauerer Betrachtung dieser Farbvektoren erkennt man, dass die ersten zwölf Vektoren der Reihe nach jeweils viermal die Farbe Rot, Grün und Blau, mit leicht unterschiedlichen Farbanteilen repräsentieren. Bei den letzten vier Vektoren sind die Farbanteile weiter verfälscht, der letzte repräsentiert einen Grauton, wobei der Rotanteil etwas überwiegt.

Sinnvollerweise wurden drei *cluster*-Zentren vorgegeben, die anfänglich mit den reinen additiven Grundfarben vorbesetzt waren. Die iterierten Zentren zeigt folgende Tabelle:

Anfängliche *cluster*-Zentren	Rot	Grün	Blau
0	255.00	0.00	0.00
1	0.00	255.00	0.00
2	0.00	0.00	255.00
Iterierte *cluster*-Zentren	Rot	Grün	Blau
0	218.40	27.52	23.78
1	31.71	226.59	23.49
2	15.33	31.40	235.24

In der folgenden Tabelle sind die Mitgliedsgradwerte für verschiedene Werte des *fuzzy*-Parameters m zusammengestellt. Eine Schwellwertbildung der Mitgliedsgradwerte wurde nicht durchgeführt. Für $m = 2$ waren sechs Iterationen, für $m = 3$ sieben Iterationen und für $m = 10$ vierzehn Iterationen notwendig (Fehlerschranke $\varepsilon = 0.001$ mit Q_2-Abbruchkriterium). Es ist klar, dass sich bei unterschiedlichen m-Werten auch geringfügig veränderte *cluster*-Zentren ergeben, die hier jedoch nicht aufgeführt sind.

	$m = 2$			$m = 3$			$m = 10$		
Nr	w_{i0}	w_{i1}	w_{i2}	w_{i0}	w_{i1}	w_{i2}	w_{i0}	w_{i1}	w_{i2}
0	0.9912	0.0048	0.0040	0.9211	0.0409	0.0380	0.5243	0.2398	0.2359
1	0.9929	0.0038	0.0033	0.9375	0.0320	0.0305	0.5790	0.2116	0.2094
2	0.9609	0.0210	0.0181	0.8098	0.0977	0.0926	0.4536	0.2747	0.2717
3	0.9948	0.0028	0.0024	0.9208	0.0405	0.0387	0.4894	0.2565	0.2541
4	0.0046	0.9908	0.0046	0.0533	0.8933	0.0534	0.2469	0.5057	0.2474
5	0.0181	0.9691	0.0128	0.1125	0.7923	0.0952	0.2886	0.4326	0.2788
6	0.0127	0.9760	0.0113	0.0779	0.8484	0.0737	0.2697	0.4634	0.2670
7	0.0167	0.9672	0.0161	0.0866	0.8280	0.0853	0.2725	0.4553	0.2722
8	0.0023	0.0024	0.9953	0.0157	0.0159	0.9685	0.1380	0.1390	0.7230
9	0.0126	0.0117	0.9757	0.0750	0.0718	0.8532	0.2682	0.2665	0.4653
10	0.0042	0.0045	0.9913	0.0443	0.0461	0.9096	0.2469	0.2502	0.5029
11	0.0063	0.0062	0.9874	0.0504	0.0498	0.8998	0.2553	0.2556	0.4891
12	0.8712	0.0840	0.0448	0.6293	0.2123	0.1584	0.3928	0.3133	0.2939
13	0.2409	0.6487	0.1105	0.3034	0.4866	0.2101	0.3290	0.3664	0.3045
14	0.1166	0.2633	0.6201	0.2117	0.3215	0.4668	0.3036	0.3352	0.3612
15	0.3813	0.3040	0.3147	0.3564	0.3194	0.3242	0.3376	0.3312	0.3312

Die Werte zeigen, wie der *fuzzy*-Parameter m die Mitgliedsgradwerte beeinflusst: Je größer m wird, desto mehr nähern sich die Werte an. Die Tendenz der Zugehörigkeit zu den

einzelnen *cluster*-Zentren bleibt aber immer erhalten. Sogar beim letzten Merkmalsvektor, einem Grauwert, bei dem der Rotanteil geringfügig überwiegt, ist bei allen m-Werten die Zugehörigkeit zum ersten *cluster*-Zentrum am größten.

Nachdem zu den Farbvektoren die *fuzzy*-Mengen berechnet sind, kann jetzt z.B. eine scharfe Zuordnung zu der Klasse (zu dem *cluster*-Zentrum) mit dem höchsten Mitgliedsgradwert erfolgen. Allerdings hat sich dann der Aufwand der Verwendung des *fuzzy*-Klassifikators eigentlich nicht gelohnt. Denn die Vorteile des Ansatzes über die *fuzzy logic* kommen erst zum Tragen, wenn weiter mit den unscharfen Mengen gearbeitet wird und diese im Rahmen einer umfangreicheren Problemstellung mit anderen unscharfen Aussagen verknüpft werden.

Bei praktischen Anwendungen dieses *fuzzy-clustering* sollte man darauf achten, dass die Menge der Merkmalsvektoren nicht zu groß wird. Denn wenn man den obigen Algorithmus betrachtet, erkennt man, dass ein beachtlicher Rechenaufwand notwendig ist. Man könnte z.B. mit anderen, einfacheren Verfahren eine Vorauswahl der Merkmalsvektoren durchführen und erst mit einer reduzierten Datenmenge den *fuzzy-cluster*-Algorithmus durchlaufen.

Kapitel 34

Speicherung von Segmenten mit Run-Length-Coding

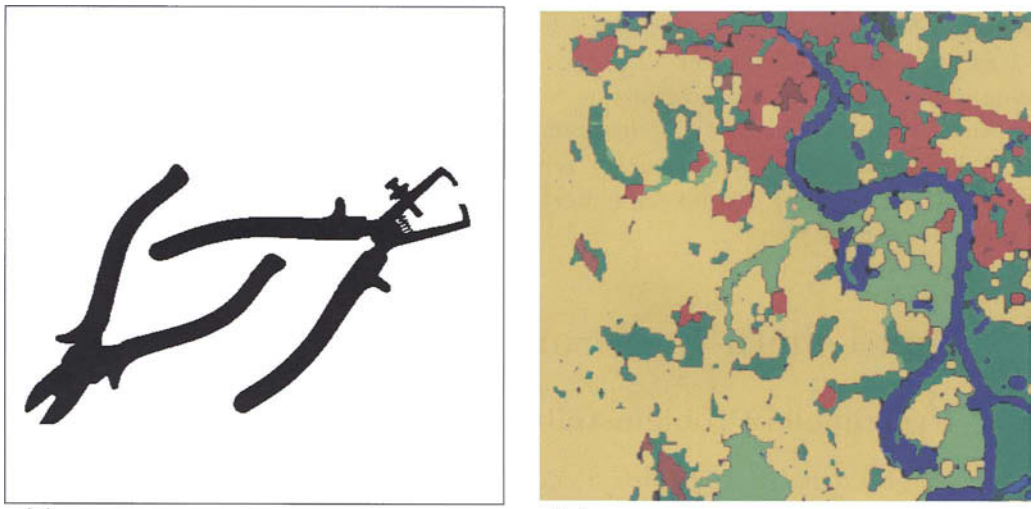
34.1 Anwendungen

Immer mehr Anwendungen erfordern, Objekte in Echtzeit zu lokalisieren, zu erkennen oder zu verfolgen, wobei die zu verwendende Bildverarbeitungshardware nur über eingeschränkte Ressourcen verfügt. Derartige Anforderungen treten oft in autonomen Systemen im Automotive-Bereich oder in der Robotik auf. Bekanntestes Beispiel dürfte die Bildverarbeitung auf eingebetteten Systemen im Roboter-Fußball sein. Die von der Kamera gelieferte Datenmenge muss hierbei möglichst frühzeitig und effizient verdichtet werden, um eine Ressourcen schonende (CPU, Speicher) Weiterverarbeitung zu ermöglichen.

Lauflängencodierung (engl.*run length encoding, RLE*) bietet hierzu die Möglichkeit, die Pixeldaten zu komprimieren und auf diesem reduzierten Datenvolumen effiziente Segmentierungs- und Merkmalsextraktionsalgorithmen aufzusetzen. Bild 34.5 zeigt die typische Zusammenarbeit von Lauflängencodierung und Union-Find-Algorithmen in diesem Umfeld effizienter Echtzeitbildverarbeitung, wie sie im Folgenden dargestellt wird.

In den vorhergehenden Kapiteln wurde erläutert, wie Bildpunkte (Merkmalsvektoren) eines, eventuell mehrkanaligen, Bildes zu Segmenten zusammengefasst werden. Das Eingabebild beinhaltet dabei die Merkmale, anhand derer die Segmentierung durchgeführt wird. Im einfachsten Fall kann das ein Grauwertbild sein, aus dem durch eine Binarisierung ein Binärbild mit Hintergrund- und Vordergrundbildpunkten erzeugt wird. Bei aufwändigeren Problemstellungen werden aus den aufgezeichneten Originaldaten zunächst weitere Merkmalsdaten berechnet (z.B. Kapitel 23, 28, 29), die einen mehrdimensionalen Merkmalsraum aufspannen. Zur Segmentierung werden dann aufwändigere Verfahren, wie multivariate Klassifikatoren, neuronale Netze oder Fuzzy-Logik (Kapitel 31, 32, 33) verwendet.

Als Eingabebild für dieses Kapitel wird ein segmentiertes Bild vorausgesetzt. Die Segmente können dabei alle den gleichen Grauwert besitzen, wie etwa bei einem Binär- oder Zweipegelbild mit dem Hintergrundgrauwert 0 und dem Vordergrundgrauwert 1 (oder 255) (Bild 34.1-a). Ein anderes Beispiel ist ein klassifiziertes Bild. Die Segmente besitzen hier



(a) (b)

Bild 34.1: (a) Beispiel für ein Binärbild. Die Hintergrundbildpunkte sind mit dem Grauwert 255 codiert. Die Vordergrundbildpunkte (Segmente) besitzen alle den Grauwert 0. (b) Beispiel für ein klassifiziertes Satellitenbild. Die Segmente besitzen unterschiedliche Grauwerte, die ein Code für die Zugehörigkeit zu einer Klasse sind. In diesem Beispiel wurden die Segmente mit einer Pseudofarbcodierung unterschiedlich eingefärbt. Die Farben der Segmente entsprechen hier folgenden Klassen: Gelb – landwirtschaftliche Nutzungsfläche, rot – Bebauung, blau – Gewässer, grün – Wald, schwarz – nicht klassifizierbar.

in der Regel unterschiedliche Grauwerte, die als Code für die Zugehörigkeit zu einer Klasse interpretiert werden können (Bild 34.1-b).

Eine Zielsetzung dieses Kapitels ist die Darstellung von Verfahren zur kompakten Speicherung von segmentierten Bildern. Dazu wird die *run-length*-Codierung (Lauflängencodierung) ausführlich dargestellt. In diesem Zusammenhang wird ein Verfahren benötigt, mit dem zusammengehörige Bildstrukturen, die lauflängencodiert sind, erkannt werden können. Auf der Basis von kompakt gespeicherten Segmenten bauen weitere Möglichkeiten der Beschreibung von Segmenten auf. In diesen Bereich fallen auch Verfahren, die Eigenschaften von Segmenten ermitteln. Beispiele dazu sind die Berechnung des Flächeninhalts, des Umfangs, des Schwerpunktes, der Orientierung, der Form, usw. Auf diese Thematik wird ab Kapitel 35 eingegangen.

Um einen Eindruck der Kompressionsrate dieses Verfahrens zu vermitteln, betrachten wir folgendes Beispiel: Das Binärbild einer Zange, Bild 34.1-a besteht aus 512x512=262144 Pixel. Durch Lauflängencodierung wird das Bild auf 806 Streifen reduziert. Das farbsegmentierte Bild, Abb. 34.1-b derselben Größe wird durch die Lauflängencodierung auf 6780 Streifen reduziert. Für die Laufzeitersparnis bedeutet dies einen Faktor 325 (a) bzw. 38 (b), der lediglich durch minimal komplexere Algorithmen zur Segmentierung bzw. Merkmalsextraktion abgeschwächt wird, für die Speicherplatzeffizienz steht der Repräsentation eines

Pixels (Binär oder farbsegmentiert: typisch 1 Byte/Pixel) durch ein Byte die Repräsentation eines Streifens durch weniger als 10 Byte gegenüber. Problematisch in der Theorie, aber von geringer praktischer Relevanz ist lediglich die Tatsache, dass keine Obergrenze für die Anzahl der Streifen existiert, die unter der Pixelanzahl des Originalbilds liegt.

Im Rahmen einer Verarbeitungsfolge kennzeichnet die Verwendung des *run-length*-Codes den Übergang von der pixelorientierten Verarbeitung zur listenorientierten Verarbeitung.

34.2 Run-Length-Codierung

34.2.1 Prinzipielle Problemstellung und Implementierung

Vorausgesetzt wird zunächst ein Binärbild mit den Grauwerten 0 (Hintergrundbildpunkte) und 1 (Vordergrundbildpunkte). Es werden nun die Längen der 0- und 1-Ketten angegeben. Der Anfang der Bildzeile

$$000000001110000000111110000 0...$$

ergibt folgenden *run-length*-Code:

$$(8 \cdot 0), (3 \cdot 1), (7 \cdot 0), (5 \cdot 1), (5 \cdot 0), \ldots .$$

Da die Hintergrundbildpunkte meistens nicht interessieren, werden nur die Positionen und die Längen der 1-Ketten codiert. Im obigen Beispiel ergibt sich dann folgender *run-length*-Code:

$$(8, 3), (18, 5), \ldots .$$

Der Code sagt also aus, dass in der betrachteten Zeile x ab der Spaltenposition $y = 8$ eine 1-Kette (*Sehne*) der Länge 3 und ab der Position $y = 18$ eine Sehne der Länge 5 beginnt. Ob in einer Implementierung für eine Sehne die Anfangsposition y_a und die Länge l angegeben wird oder die Anfangsposition y_a und die Endposition y_e, macht keinen prinzipiellen Unterschied. Die Entscheidung kann möglicherweise von spezieller Hardware abgenommen werden, die den *run-length*-Code in der einen oder anderen Form liefert.

Ein Beispiel zeigt Bild 34.2. Es entstand aus Bild 34.1-a durch Verkleinerung auf $100 \cdot 100$ Pixel. Es wurden die schwarzen Vordergrundpixel codiert. Ein Ausschnitt aus einem lesbaren Ausdruck des *run-length*-Codes ist im Folgenden wiedergegeben. Die Bedeutung des Wertes „`Code`“ wird später erläutert:

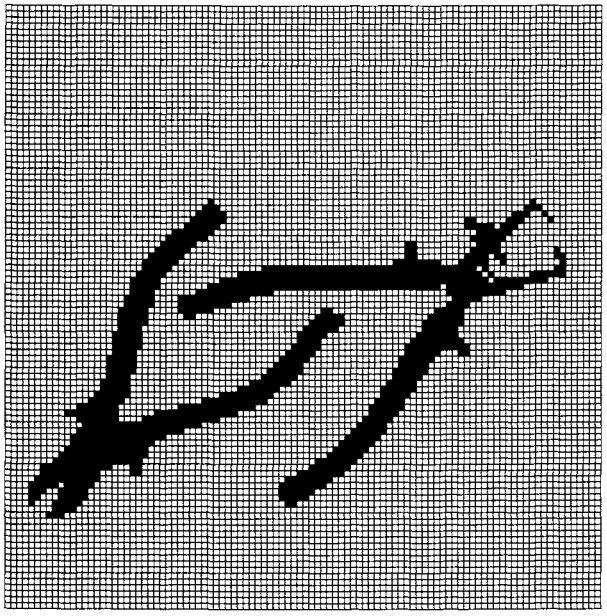


Bild 34.2: Beispiel zur *run-length*-Codierung. Das Testbild 34.1-a wurde auf eine Größe von $100 \cdot 100$ Pixel verkleinert. Dann wurden die scharzen Vordergrundbildpunkte codiert. Ein Auszug des zugehörigen *run-length*-Codes ist in den laufenden Text eingefügt.

```
Leerzeile: 1
Leerzeile: 2
...
Leerzeile: 32
In Zeile:  33 sind 2 Ketten
               Anfang:   34 Ende:   34 Code:   1
               Anfang:   88 Ende:   88 Code:   4
In Zeile:  34 sind 2 Ketten
               Anfang:   33 Ende:   35 Code:   1
               Anfang:   87 Ende:   89 Code:   4
In Zeile:  35 sind 3 Ketten
               Anfang:   32 Ende:   36 Code:   1
               Anfang:   85 Ende:   87 Code:   4
               Anfang:   90 Ende:   90 Code:   2
In Zeile:  36 sind 4 Ketten
               Anfang:   31 Ende:   36 Code:   1
               Anfang:   77 Ende:   78 Code:   4
               Anfang:   84 Ende:   86 Code:   4
               Anfang:   91 Ende:   91 Code:   3
In Zeile:  37 sind 3 Ketten
               Anfang:   30 Ende:   35 Code:   1
               Anfang:   77 Ende:   81 Code:   4
               Anfang:   83 Ende:   85 Code:   4
...
In Zeile:  83 sind 3 Ketten
               Anfang:    4 Ende:    4 Code:   1
               Anfang:    8 Ende:   12 Code:   1
               Anfang:   47 Ende:   48 Code:   4
In Zeile:  84 sind 1 Ketten
               Anfang:    8 Ende:   11 Code:   1
In Zeile:  85 sind 1 Ketten
               Anfang:    7 Ende:    9 Code:   1
Leerzeile: 86
Leerzeile: 87
...
Leerzeile: 100
```

Dieser Ausdruck zeigt auch, welche Informationen in der Datenstruktur eines *run-length*-codierten Bildes vorhanden sein müssen: Als erstes müssen die Bildgröße und die Zeilennummern verfügbar sein. Außerdem werden die Anzahl der Sehnen pro Zeile und die Anfangs- und Endangaben (oder Längenangaben) der Sehnen in der Zeile benötigt. Schließlich sollte noch ein Feld mitgeführt werden, in dem die Zugehörigkeit einer Sehne zu einem bestimmten Segment festgehalten wird. Auf diese Problematik wird im nächsten

Abschnitt eingegangen. Folgende Tabelle zeigt ein einfaches Beispiel einer Datenstruktur für den *run-length*-Code:

Globale Größen	
	Anzahl der Bildzeilen
	Anzahl der Bildspalten
run-length-Code	
	Zeilennummer
	Anzahl der Sehnen pro Zeile
	Sehnenanfang
	Sehnenende
	Code der Sehne
	...
	Zeilennummer
	Anzahl der Sehnen pro Zeile
	Sehnenanfang
	Sehnenende
	Code der Sehne
	...

34.2.2 Vereinzelung von Segmenten

Bei einem klassifizierten Bild sind die verschiedenen Klassen mit unterschiedlichen Grauwerten codiert. Wenn ein derartiges Bild lauflängencodiert wird, so können die Klassencodes sofort im Feld `Code der Sehne` eingetragen werden. Bei einem binarisierten Bild haben alle Segmente denselben (Vordergrund-)Grauwert. Wenn die zusammenhängenden Flächen zu einem Segment zusammengefasst werden sollen, so muss zu jeder Sehne des *run-length*-Codes im Feld `Code der Sehne` angegeben werden, zu welchem Segment sie gehört. Dieser Vorgang wird oft auch als *Vereinzelung der Segmente* bezeichnet.

Der *run-length*-Code wird dazu vom Anfang an abgearbeitet, was einer zeilenweisen Verarbeitung des Bildes von oben nach unten entspricht. Angenommen, es tritt die erste Sehne des korrespondierenden Binärbildes in Zeile x auf. Es ist zu prüfen, ob die Sehne mit Sehnen der Zeile $x + 1$ gemeinsame Nachbarn besitzen. Dabei wird entweder 4- oder 8-Nachbarschaft vorausgesetzt. Ist dies der Fall, so gehören diese Sehnen zum selben Segment. Es können aber auch zwei Sehnen einer Zeile x zum selben Segment gehören. Das ist der Fall, wenn das Segment nach oben oder unten geöffnet ist. Der Fall „nach unten geöffnet" ist einfacher zu entscheiden, da ja bei der Verarbeitung von oben nach unten dann Überdeckungen von Sehnen vorliegen. Der Fall „nach oben geöffnet" ist schwieriger: Hier kann erst in einem zweiten Durchlauf entschieden werden, welche Sehnen zusammen gehören.

In Bild 34.3 ist das Ergebnis der Vereinzelung von Bild 34.3-a abgebildet. Im verkleinerten Bild 34.2 treten, im Gegensatz zum Original, im rechten oberen Bereich der rechten Zange (Zeilen 35 und 36) zwei Bildpunkte auf, die schräg nach rechts/unten liegen. Da hier

Bild 34.3: Ergebnis der Vereinzelung des Testbildes 34.1-a

4-Nachbarschaft verwendet wurde, wurden diese Bildpunkte als eigene Segmente codiert. Im Ausdruck des *run-length*-Codes sind aus diesem Grund in den Zeilen 35 und 36 zwei Segmente mit den Codes 2 und 3 aufgeführt. Die linke Zange hat den Code 1 und die rechte den Code 4.

Wenn die Segmentvereinzelung abgeschlossen ist, kann der *run-length*-Code bezogen auf die gefundenen Segmente effektiver gespeichert werden. Im Folgenden wird ein Vorschlag für eine *run-length*-Code-Struktur gegeben, die sich in der Praxis bewährt hat. Es wurden zusätzliche Informationen mit aufgenommen. Das erleichtert die spätere Weiterverarbeitung, geht aber zu Lasten der Datenkompression.

Der *run-length*-Code der Segmente wird bezüglich eines frei wählbaren Referenzpunktes (x_r, y_r) erzeugt. Dieser Referenzpunkt kann für alle in einem Bild auftretenden Segmente identisch sein, etwa die linke obere Bildecke mit den Koordinaten $(0, 0)$. Es besteht auch die Möglichkeit, für jedes Segment i einen eigenen Referenzpunkt (x_{r_i}, y_{r_i}) festzulegen, z.B. einen beliebigen Punkt innerhalb des Segments. Die Koordinatenwerte aller Referenzpunkte (x_{r_i}, y_{r_i}) werden dann bezüglich eines „globalen" Referenzpunktes (x_r, y_r) (z.B. Position $(0, 0)$), angegeben. Als weitere globale Größen müssen noch Zeilen- und Spaltenanzahl des Bildes gespeichert werden. In folgender Tabelle ist die gesamte Datenstruktur wiedergegeben:

Globale Größen	
	Anzahl der Bildzeilen
	Anzahl der Bildspalten
	Referenzpunkt (x_r, y_r)
Lokale Größen des Segments i	
	Referenzpunkt (x_{r_i}, y_{r_i})
	Code des Segments
	Anzahl der Zeilen
	Zeilennummer bzgl. des Referenzpunktes
	Anzahl der Sehnen pro Zeile
	Sehnenanfang bzgl. des Referenzpunktes
	Sehnenende bzgl. des Referenzpunktes
	...
	Zeilennummer bzgl. des Referenzpunktes
	Anzahl der Sehnen pro Zeile
	Sehnenanfang bzgl. des Referenzpunktes
	Sehnenende bzgl. des Referenzpunktes
	...

In 34.4 ist ein einfaches Beispiel dazu gegeben. Das Bild ist $16 \cdot 16$ Pixel groß und enthält zwei Segmente. Als globaler Referenzpunkt (x_r, y_r) wird die linke obere Ecke $(0, 0)$ gewählt. Die folgenden Tabellen enthalten die Datenstruktur des dazugehörigen *run-length*-Codes.

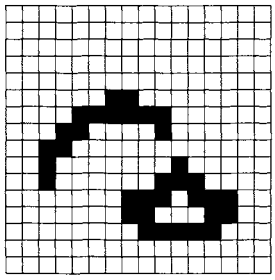


Bild 34.4: Beispiel zur *run-length*-Codierung. Das Bild ist $16 \cdot 16$ Pixel groß und enthält zwei Segmente. Als globaler Referenzpunkt (x_r, y_r) wird die linke obere Ecke $(0, 0)$ gewählt.

Globale Größen	
Anzahl der Bildzeilen	16
Anzahl der Bildspalten	16
Referenzpunkt	(0, 0)

Segment 1	
Referenzpunkt	(8, 5)
Code des Segments	1
Anzahl der Zeilen	6
Zeilennummer	-3
Anzahl der Sehnen pro Zeile	1
Sehnenanfang	1
Sehnenende	2
Zeilennummer	-2
Anzahl der Sehnen pro Zeile	1
Sehnenanfang	-1
Sehnenende	4
Zeilennummer	-1
Anzahl der Sehnen pro Zeile	2
Sehnenanfang	-2
Sehnenende	-1
Sehnenanfang	4
Sehnenende	4
Zeilennummer	0
Anzahl der Sehnen pro Zeile	1
Sehnenanfang	-3
Sehnenende	-2
Zeilennummer	1
Anzahl der Sehnen pro Zeile	1
Sehnenanfang	-3
Sehnenende	-3
Zeilennummer	2
Anzahl der Sehnen pro Zeile	1
Sehnenanfang	-3
Sehnenende	-3

Segment 2	
Referenzpunkt	(9, 7)
Code des Segments	1
Anzahl der Zeilen	5
Zeilennummer	0
Anzahl der Sehnen pro Zeile	1
Sehnenanfang	3
Sehnenende	3
Zeilennummer	1
Anzahl der Sehnen pro Zeile	1
Sehnenanfang	2
Sehnenende	4
Zeilennummer	2
Anzahl der Sehnen pro Zeile	2
Sehnenanfang	0
Sehnenende	2
Sehnenanfang	4
Sehnenende	6
Zeilennummer	3
Anzahl der Sehnen pro Zeile	2
Sehnenanfang	0
Sehnenende	1
Sehnenanfang	5
Sehnenende	6
Zeilennummer	4
Anzahl der Sehnen pro Zeile	1
Sehnenanfang	1
Sehnenende	5

Wenn Segmente als *run-length*-Code gespeichert sind, ist es möglich, eine Reihe von segmentbeschreibenden Merkmalen zu berechnen, ohne zuvor in die ursprüngliche Rasterdarstellung zurückgehen zu müssen. Beispiele dazu werden in den folgenden Kapiteln gegeben.

Eine zur *run-length*-Codierung verwandte Technik ist die Codierung von Bildern als *quadtrees* oder *octrees* (Kapitel 30).

34.2.3 Effiziente Vereinzelung mit Union-Find-Algorithmen

Eine Methode zur Bildung von Zusammenhangskomponenten, die wegen ihrer Laufzeit- und Speicherplatzeffizienz eine immer größere Verbreitung findet, stellt die Klasse der Union-Find-Algorithmen dar. Bild 34.5 verdeutlicht das Zusammenspiel zwischen der Lauflängencodierung und dem Union-Find-Algorithmus: Aus den unzusammenhängenden Streifen der Lauflängencodierung werden Zusammenhangskomponenten gebildet, für die dann - wie im folgenden Kapitel beschrieben - Merkmale bestimmt werden können.

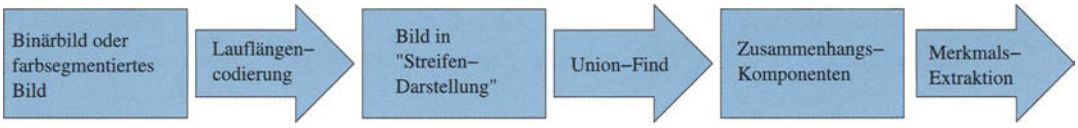


Bild 34.5: Zusammenspiel zwischen Lauflängencodierung und Union-Find-Algorithmus

Die unzusammenhängenden Streifen aus der Lauflängencodierung kann man sich nun als Graph denken, wobei die Streifen Knoten darstellen und zwischen zwei benachbarten, sich berührenden Streifen eine Kante besteht. Glücklicherweise stellt die Graphentheorie sehr einfache, effektive Algorithmen zur Verfügung, die Zusammenhangskomponenten sehr effizient – sowohl Speicherplatz als auch Rechenzeit betreffend – ermitteln können, die sogenannten Union-Find-Algorithmen. Der Name bezieht sich dabei auf die zwei Operationen, die für den Umgang mit Zusammenhangskomponenten erforderlich sind: `Union` für die Vereinigung zweier Zusammenhangskomponenten (beim dynamischen Erzeugen eines Graphen durch Einfügen einer Kante) sowie `Find` für die Abfrage, ob zwei Knoten des Graphen zur selben Zusammenhangskomponente gehören.

Als Datenstruktur zur Verwaltung der Zusammenhangskomponenten wird meist ein Wald von Bäumen verwendet, wobei die Bäume durch Rückwärtsverweise bis zur Wurzel realisiert werden (Bild 34.2.3). Die Wurzel eines Baumes ist dadurch gekennzeichnet, dass sie keinen weiterführenden Verweis enthält. Als Repräsentant für eine (durch einen Baum dargestellte) Zusammenhangskomponente wird die jeweilige Wurzel verwendet. Die Operation `Find` durchsucht iterativ die Kette der Vorgänger, bis ein Knoten ohne Vorgänger (der Repräsentant dieser Zusammenhangskomponente) gefunden wird:

```
Id Find ( Id x )          // findet den Repräsentanten der Zusammenhangs-
{                         // komponente, zu der x gehört
  Id rootX = x;           // Initialisierung, gleichzeitig Ergebnis
                          // bei einelementiger Komponente
  while hasParent(rootX) // iterativ rückwärts bis zur Wurzel laufen
    rootX = rootX.parent;
  return rootX;
}
```

Die Operation `Union` ermittelt die Repräsentanten der beiden Elemente, d.h. die Wurzeln der beiden Bäume. Sind die beiden Wurzeln nicht identisch, wird als Vorgänger der einen Wurzel die Wurzel der anderen Zusammenhangskomponente eingetragen:

```
Union ( Id x, Id y)          // vereinigt die Zusammenhangskomponenten,
{                            // denen x und y angehören
   Id rootX, rootY;
   rootX = Find(x);          // die beiden Wurzeln (Repräsentanten
   rootY = Find(y);          // der Komponenten) ermitteln...
   if ( rootX != rootY )
      rootX.parent = rootY; // ... und gemeinsamen Repräsentanten setzen
}
```

Die Grafiken in Bild 34.2.3 visualisieren die Verzeigerung in dieser Datenstruktur, die entsteht, wenn die lauflängencodierte Bildrepräsentation zu Zusammenhangskomponenten zusammengefügt wird, wobei jeweils nur die Streifen zweier aufeinanderfolgender Zeilen `prev` und `akt` betrachtet werden.

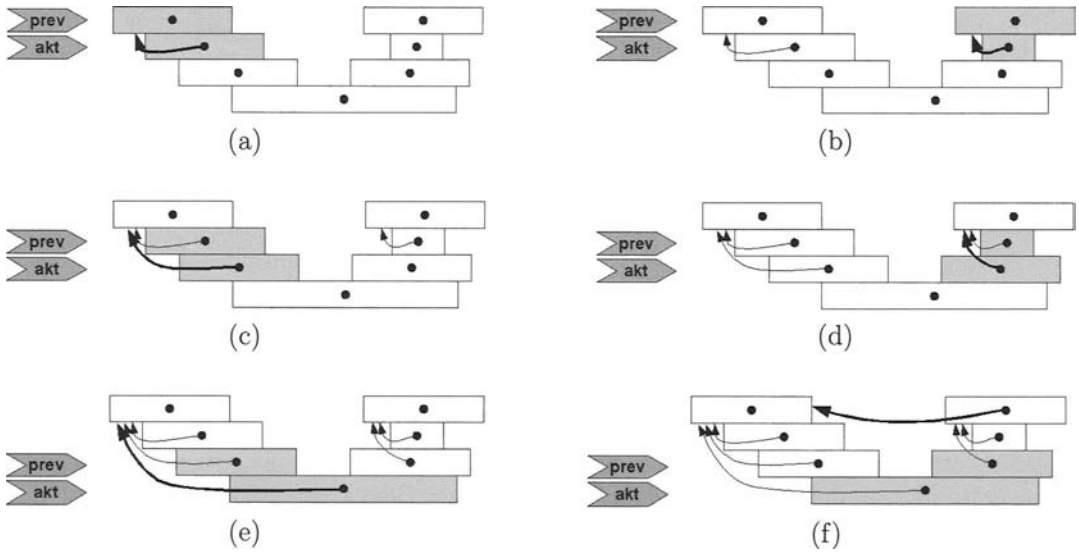


Bild 34.6: Visualisierung der Arbeitsweise des Union-Find-Algorithmus: Das Bild wird zeilenweise durchlaufen, wobei jeweils zwei benachbarte Zeilen (`akt` und `prev`) betrachtet werden. Berühren sich zwei Streifen, so werden diese (sowie ihre bereits bestehenden Zusammenhangskomponenten) zu einer Zusammenhangskomponente vereinigt.

Da bei obigem naiven Algorithmus entartete (z.B. links- oder rechtsgekämmte) Bäume entstehen können, werden in der Praxis meist Kombinationen der folgenden beiden Optimierungsverfahren verwendet:

Pfadkomprimierung (*path compression*): Wird die Wurzel eines Knotens x gesucht, kann dies bedeuten, dass ein mehr oder weniger langer Pfad über mehrere Vorgänger (s. gekämmter Baum) durchlaufen werden muss, bis die Wurzel gefunden wird. Ist die Wurzel gefunden, kann sie für den Knoten x als direkter Vorgänger gesetzt werden. Wird bei einem späteren Aufruf erneut die Wurzel von x gesucht, hat der Suchpfad die Länge 1. Der Zusatzaufwand für die Pfadkompression (z.B. das zusätzliche Statement `x.parent = rootX;` am Ende der Funktion `getRoot`) ist minimal. Manche Implementierungen machen dies für alle Knoten des durchlaufenen Pfades, was von Anwendungsfall zu Anwendungsfall aber auch einen Mehraufwand bedeuten kann.

Gewichtsausgleich (*weight balancing*): Man kann das Entstehen entarteter Bäume einfach dadurch verhindern, dass die Baumtiefe in der jeweiligen Baumwurzel mitcodiert wird und bei der Vereinigung zweier Bäume der jeweils flachere an die Wurzel des tieferen Baums gehängt wird, statt diesen Vorgang – wie in der naiven Darstellung oben – ohne Betrachtung der Baumtiefe durchzuführen.

Tarjan [Tarj75] hat nachgewiesen, dass durch diese Optimierungsverfahren die Baumtiefe für alle in der Realität denkbaren Kantenanzahlen nie größer als 4 wird, sodass die Zeit für die Operationen `union` und `find` nahezu konstant ist.

Zusammenfassend sei erwähnt,

- dass für die Repräsentation der Zusammenhangskomponenten lediglich ein Array mit den Indizes der Streifen (oder alternativ dazu je ein zusätzlicher Zeiger pro Streifen) erforderlich ist (Speicherplatzeffizienz),
- dass die Anzahl der für die Erzeugung der Zusammenhangskomponenten (`Union` ist nahezu konstant) erforderlichen Operationen nahezu linear in der Anzahl der Kanten (Nachbarschaftsbeziehungen zwischen Streifen) sind, sowie
- dass die Anzahl der für die Anfrage, ob zwei Streifen zur selben Zusammenhangskomponente gehören (`Find`), erforderlichen Operationen nahezu konstant ist.

Kapitel 35

Einfache segmentbeschreibende Parameter

35.1 Anwendungen

In diesem Kapitel werden einfache Verfahren angegeben, mit denen Segmente, die in einer Segmentierungsstufe erzeugt wurden, näher beschrieben werden. Dazu werden Parameter berechnet, die die Segmente charakterisieren. In dieser Verarbeitungsstufe findet in Verbindung mit der kompakten Speicherung von Segmenten (z.B. Kapitel 34) der Übergang von der pixelorientierten Verarbeitung zur listenorientierten Verarbeitung statt. Damit ist gemeint, dass die Segmente zusammen mit den sie beschreibenden Größen als Datenstrukturen gespeichert werden, die algorithmisch gut zu handhaben sind.

Häufig müssen die Segmente nach diesem Verarbeitungsschritt daraufhin untersucht werden, ob sie einem bestimmten vorgegebenen oder gesuchten Typ entsprechen. Man kann diese Problemstellung als eine Klassifizierung von Segmenten auffassen. Dabei wird man nach dem „Prinzip des minimalen Aufwands" vorgehen: Wenn z.B. zwei Segmente allein durch ihren Flächeninhalt zu unterscheiden und damit entsprechenden Klassen zuzuordnen sind, so ist es nicht notwendig, weitere Parameter zu berechnen. Hier wird deshalb oft von einer sequentiellen Segmentklassifizierung gesprochen, bei der der Reihe nach bestimmte Segmentparameter abgeprüft werden. Anhand ihrer Werte werden die Segmente den entsprechenden Klassen zugeordnet.

Bei einer parallelen Segmentklassifizierung werden zu den einzelnen Segmenten n-dimensionale Merkmalsvektoren berechnet, die dann mit einem Klassifikator weiterverarbeitet werden. Dazu können z.B. alle Verfahren verwendet werden, die in den Kapiteln 19, 20 und 27 erläutert wurden.

Zur Thematik der Segmentbeschreibung werden neben der Berechnung von einfachen Eigenschaften und der auf der Basis dieser Eigenschaften durchgeführten Klassifizierung auch komplexere Verfahren untersucht. Eine wichtige Zielsetzung dieser Verfahren ist die translations-, rotations- und größeninvariante Segmenterkennung. Beispiele dazu werden in diesem Kapitel und in den Kapiteln 20, 36 und 37 gegeben.

Am Ende einer derartigen Verarbeitung kann beispielsweise eine Aussage der folgenden Form stehen: „Das Segment ist die Abbildung eines Bauteils vom Typ X, das allerdings eine Fehlerstelle aufweist." Es kann sich dann eine geeignete Reaktion anschließen, z.B. das Markieren oder Aussteuern des Bauteils.

Im Folgenden wird vorausgesetzt, dass die Segmente vereinzelt, d.h. mit unterschiedlichen Grauwerten codiert, sind. Es besteht auch die Möglichkeit, dass sie *run-length*-codiert vorliegen und die Segmentvereinzelung gemäß Abschnitt 34.2 durchgeführt wurde.

35.2 Flächeninhalt

Der Flächeninhalt eines Segments ist einfach zu berechnen: Es müssen nur für jedes Segment die Pixel aufsummiert werden, die zum Segment gehören. Der Flächeninhalt wird in der Maßeinheit *Pixel* angegeben. Ist das Segment *run-length*-codiert (Kapitel 10), so werden alle Sehnenlängen addiert, die den Sehnencode des Segments haben. Der Algorithmus ist zusammen mit der Berechnung der Schwerpunkte der Segmente in Abschnitt 35.3 dargestellt.

35.3 Flächenschwerpunkt

Auch der Flächenschwerpunkt eines Segments ist einfach zu berechnen. Der Schwerpunkt (x_{cp}, y_{cp}) eines „Systems materieller Punkte" $M_j(x_j, y_j)$ mit den Massen m_j berechnet sich nach der Formel:

$$x_{cp} = \frac{\sum\limits_j m_j \cdot x_j}{\sum\limits_j m_j}, \quad y_{cp} = \frac{\sum\limits_j m_j \cdot y_j}{\sum\limits_j m_j}. \tag{35.1}$$

Übertragen auf die vorliegende Problemstellung sind die Punkte (x_{ij}, y_{ij}) die Pixel, die zum Segment mit der Nummer i gehören. Die „Massen" der Bildpunkte sind alle gleich eins, so dass im Nenner der Formeln (35.1) der Flächeninhalt des Segments i auftritt.

Die Segmentflächen und die Segmentschwerpunkte werden in Abschnitt 35.9 noch in einem anderen Zusammenhang, nämlich bei der Berechnung von Momenten, benötigt.

Im Folgenden ist der Algorithmus zur Berechnung der Segmentflächen und der Schwerpunkte angegeben, falls die Segmente *run-length*-codiert sind. Dieser Algorithmus kann auch als Vorlage für die in den weiteren Abschnitten nicht explizit angegebenen Algorithmen angesehen werden.

A35.1: Berechnung der Flächen und der Schwerpunkte.

Voraussetzungen und Bemerkungen:

◇ $\mathbf{S} = (s(x,y))$ segmentiertes Bild, das *run-length*-codiert ist. Der *run-length*-Code sei im Feld rl abgelegt. Die Struktur des *run-length*-Codes entspricht der Datenstruktur von Kapitel 10.

◇ Im Feld `Code der Sehne` ist die jeweilige Segmentnummer eingetragen.

◇ In $area[i]$ werden die Flächen der Segmente i gespeichert. $h1[i]$ und $h2[i]$ sind zwei Hilfsfelder.

◇ In $x_{cp}[i]$ und $y_{cp}[i]$ werden die Zeilen- und Spaltenkoordinaten des Schwerpunkts des Segments i gespeichert.

Algorithmus:

(a) Für alle Segmente i: $area[i] = 0; h1[i] = 0; h2[i] = 0;$

(b) akt_zeile = erste Zeile des RL-Codes;

(c) Solange akt_zeile nicht die letzte Zeile des RL-Codes ist:

(ca) Falls akt_zeile eine Leerzeile ist: akt_zeile = nächste Zeile;

(cb) Sonst:

(cba) Für alle Sehnen akt_sehne von akt_zeile:

$k = akt\_end - akt\_anf + 1;$

$area[akt\_code] = area[akt\_code] + k;$

$h1[akt\_code] = h1[akt\_code] + (akt\_anf + akt\_end) \cdot k;$

$h2[akt\_code] = h2[akt\_code] + akt\_zeile \cdot k;$

(d) Für alle Segmente i:

$x_{cp}[i] = h2[i]/area[i];$

$y_{cp}[i] = h1[i]/(2 \cdot area[i]);$

Ende des Algorithmus

Wenn die Koordinaten des Segmentschwerpunkts ermittelt sind, kann man als zusätzliches Merkmal verwenden, ob der Schwerpunkt innerhalb oder außerhalb des Segments liegt. Dieses Merkmal ist in der Praxis oft gut zur Unterscheidung von Segmenten geeignet. Ist von einem Segment bekannt, dass der Schwerpunkt immer innerhalb liegt, so können bei einer Klassifizierung alle Segmente, bei denen der Schwerpunkt außerhalb liegt, ignoriert werden.

35.4 Umfang

Bei vielen Anwendungen wird auch die Länge des Randes eines Segments benötigt. Soll für das Segment i dieser Wert ermittelt werden, so werden alle mit dem Wert i codierten Bildpunkte danach untersucht, ob die 4- (oder 8-) Nachbarn ebenfalls denselben Code besitzen. Bildpunkte, für die das nicht zutrifft, sind Randpunkte, die aufsummiert werden. Bei dieser Berechnung kann auch der Rand des Segments als Datenstruktur, z.B. in der Form eines *run-length*-Codes, codiert werden. Weitere Verfahren zur Randverarbeitung werden in Abschnitt 35.8 gegeben.

35.5 Kompaktheit

Aus dem Flächeninhalt und der Länge des Umfangs eines Segments lässt sich eine Maßzahl berechnen, die als *Kompaktheit* bezeichnet wird. Als Maßzahl verwendet man

$$V = \frac{Umfang^2}{Fläche}. \tag{35.2}$$

In der folgenden Tabelle ist für einige einfache geometrische Formen die Kompaktheit angegeben:

Form	Umfang	Fläche	Kompaktheit
Kreis	2π r	$r^2\pi$	$4\pi = 12.57$
Quadrat	$4a$	a^2	16
Rechteck	$2a + 2b$	ab	$8 + \frac{4a}{b} + \frac{4b}{a}$
gleichseitiges Dreieck	$3a$	$\frac{a^2}{4}\sqrt{3}$	20.78

Die kompakteste Form ist der Kreis. Man sieht, dass die Maßzahl für die Kompaktheit desto mehr vom Idealwert $V = 4\pi$ abweicht, je deutlicher sich die Form von einer Kreisform unterscheidet. Betrachtet man z.B. das Rechteck: Falls für die Seitenlängen $b = a$ gilt, ergibt sich ein Quadrat und damit der Wert $V = 16$. Bei $b = 2a$ erhält man $V = 18$ und bei $b = 4a$ ist $V = 25$, d.h. je länglicher eine Fläche ist, desto größer ist der Wert von V. Man kann somit dieses Maß verwenden, um längliche Segmente von nicht länglichen Segmenten zu unterscheiden. Eine weitere Methode dazu wird in Abschnitt 35.6 gegeben.

Praktische Anwendungen haben gezeigt, dass die Signifikanz dieses Maßes, vor allem bei Segmenten mit kleinem Flächeninhalt und in unterschiedlichen Drehlagen, nicht immer zufriedenstellend ist.

35.6 Orientierung

Zur Lagebestimmung von Segmenten wird häufig auch die *Orientierung* eines Segments benötigt. Im Folgenden wird ein einfaches Verfahren dargestellt, das bei deutlich länglichen Segmenten gut die beiden Hauptrichtungen ermittelt. Es beruht auf der Annäherung

eines Segments durch eine Ellipse, deren große und kleine Hauptachse die Orientierung des Segments bestimmen. Bei nicht länglichen Segmenten, z.B. bei nahezu kreisförmigen oder quadratischen Segmenten, liefert das Verfahren zwar auch Orientierungswerte, die aber meistens ohne Signifikanz sind.

Es wird ein bestimmtes Segment betrachtet, das aus der Menge der diskreten Punkte $\{(x,y)\}$ besteht. Alle Bildpunkte des Segments besitzen als Code den gleichen Grauwert g, für den ohne Beschränkung der Allgemeinheit $g = 1$ angenommen werden kann.

Zur Berechnung der Orientierung des Segments wird als Erstes die Kovarianzmatrix C des Segments berechnet:

$$C = \begin{pmatrix} v_{xx} & v_{xy} \\ v_{yx} & v_{yy} \end{pmatrix}, \tag{35.3}$$

wobei

$$v_{xx} = \frac{\sum_x \sum_y (x - x_{cp})^2}{A}, \quad v_{yy} = \frac{\sum_x \sum_y (y - y_{cp})^2}{A}, \tag{35.4}$$

$$v_{xy} = v_{yx} = \frac{\sum_x \sum_y (x - x_{cp})(y - y_{cp})}{A}.$$

Die Größe A in (35.4) ist die Fläche des Segments. x_{cp} und y_{cp} sind die Koordinaten des Schwerpunkts. Im nächsten Schritt werden die Eigenwerte λ_1 und λ_2 der Kovarianzmatrix C berechnet:

$$\lambda_1 = \frac{(v_{xx} + v_{yy}) + \sqrt{(v_{xx} - v_{yy})^2 + 4 v_{xy} v_{yx}}}{2}, \tag{35.5}$$

$$\lambda_2 = \frac{(v_{xx} + v_{yy}) - \sqrt{(v_{xx} - v_{yy})^2 + 4 v_{xy} v_{yx}}}{2}.$$

Die Orientierung des Segments lässt sich jetzt über die zu den Eigenwerten λ_1 und λ_2 gehörigen Eigenvektoren $\vec{ev}_1$ und $\vec{ev}_2$ berechnen. Die beiden Eigenvektoren werden auch als die beiden *Hauptrichtungen des Segments* bezeichnet. Die Orientierung wird mit Hilfe von λ_1, dem größeren der beiden Eigenwerte, berechnet. Die Berechnung über λ_2 würde einen dazu senkrechten Winkel ergeben.

Wegen der Beziehung

$$\begin{pmatrix} v_{xx} - \lambda_1 & v_{xy} \\ v_{yx} & v_{yy} - \lambda_1 \end{pmatrix} \cdot \begin{pmatrix} ev_{1x} \\ ev_{1y} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{35.6}$$

gilt zwischen den Komponenten des Eigenvektors folgende Gleichung:

$$ev_{1y} = \frac{\lambda_1 - v_{xx}}{v_{xy}} \cdot ev_{1x}. \tag{35.7}$$

Die Orientierung erhält man mit

$$\varphi = \arctan \frac{\lambda_1 - v_{xx}}{v_{xy}}. \tag{35.8}$$

Bei der numerischen Berechnung ist zu beachten, dass in der Nähe von $\varphi = \pm 90°$ der Nenner $v_{xy} = 0$ oder zumindest sehr klein sein kann. Diese Fälle sind gesondert zu behandeln, um Divisionen durch null zu vermeiden.

Ein weiteres segmentbeschreibendes Merkmal lässt sich aus den Eigenwerten ableiten: λ_1 und λ_2 geben die Ausdehnung des Segments entlang der 1. und 2. Hauptrichtung an. Der Quotient $\frac{\lambda_1}{\lambda_2}$ kann als Maß dafür verwendet werden, ob das Segment länglich ist oder nicht. Bei einem länglichen Segment ist λ_1 deutlich größer als λ_2, deswegen ist der Quotient deutlich größer als eins. Bei nicht länglichen Segmenten ist der Quotient ungefähr gleich eins.

Bild 35.1 ist ein Beispiel für ein *run-length*-codiertes Binärbild mit verschiedenen Segmenten. Die Parameter sind in folgender Tabelle zusammengefasst:

Bezeichnung	x_{cp}	y_{cp}	Fläche	λ_1	λ_2	$\frac{\lambda_1}{\lambda_2}$	φ
Viereck	64	92	4500	419.4	334.8	1.25	168°
Blechteil	69	285	3252	1639.0	63.2	25.93	61°
große Beilagscheibe	117	409	5971	685.4	575.3	1.19	4°
Teil mit Ring	174	65	3162	425.9	404.4	1.05	159°
Hacken	125	192	1195	3299.5	109.1	30.24	14°
Sechseck	186	286	3751	321.1	281.6	1.14	10°
kleine Beilagscheibe	298	51	497	83.3	72.3	1.15	4°
Kreisscheibe	270	151	3281	283.8	240.6	1.18	178°
Schlüssel	329	287	2267	532.0	123.4	4.31	125°
Verbindungsstück	279	418	2397	325.6	266.6	1.22	2°

35.7 Fourier-Transformation der Randpunkte

Um ein Segment bezüglich der Lage und Orientierung zu normieren, kann man die Randpunkte des Segments Fourier-transformieren. Die Grundlagen zur Fourier-Transformation sind in Kapitel 21 zusammengestellt. Um die Randpunkte Fourier-transformieren zu können, werden ihre (x, y)-Koordinaten als komplexe Zahlen aufgefasst: Die x-Werte sind der Realteil und die y-Werte der Imaginärteil. Einem Randpunkt mit den Koordinaten (x, y) ist durch diese Vorschrift die komplexe Zahl $x + iy$ zugeordnet. Wenn man die Fourier-Transformation auf diese Punkte anwendet, erhält man die Fourier-Komponenten der Randpunkte. Da die Fourier-Transformation umkehrbar ist, gehen bei dieser Transformation keine Informationen verloren.

Falls der Rand des Segments eine geschlossene Kurve ist und die Randpunkte im Uhrzeigersinn ermittelt wurden, wird die erste Fourier-Komponente die betragsmäßig größte

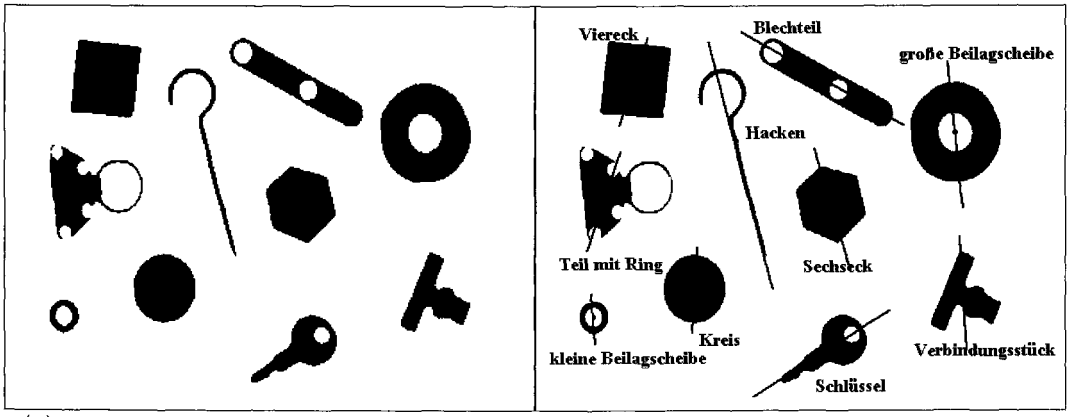


Bild 35.1: (a) Originalbild mit zehn Teilen. (b) *run-length*-codiertes Bild. Die Schwerpunkte und die Orientierungen sind eingezeichnet. Die weiteren Parameterwerte sind in der Tabelle zusammengestellt. Bei nicht länglichen Segmenten sind die berechneten Orientierungen meistens wenig bedeutsam. Die Drehwinkel in der zugehörigen Tabelle sind, ausgehend von der nach unten gehenden x-Achse, im mathematisch positiven Sinn zwischen 0° und 180° angegeben.

sein. Eine Größennormierung kann jetzt durch die Division aller Fourier-Komponenten durch den Betrag der ersten Komponente erhalten werden.

Einer Drehung der Randpunkte im Ortsbereich entspricht eine Multiplikation der Fourier-Komponenten mit dem Wert $e^{i\varphi}$, wobei φ der Drehwinkel ist. Diese Operation ändert somit den Betrag der Fourier-Komponenten nicht.

35.8 Chain-Codierung

Neben der Länge des Randes eines Segments (Abschnitt 35.4) ist oft auch der gesamte Umriss, die *Kontur*, von Interesse. Die *chain*-Codierung bietet eine Möglichkeit, zu einem Segment eine Datenstruktur zu erzeugen, mit der die Kontur kompakt gespeichert ist. Außerdem ist die Möglichkeit weiterer Verarbeitungsschritte gegeben.

Das Grundprinzip besteht darin, dass man ausgehend von einem Randpunkt des Segments als Startpunkt im Uhrzeigersinn entlang der Kontur wandert. Das Verfahren wird beendet, wenn man wieder am Startpunkt angelangt ist oder wenn kein neuer Randpunkt gefunden werden kann (z.B. wenn der Bildrand erreicht wird oder eine Linie endet).

Geht man vom Startpunkt aus, so kann der nächste Randpunkt nur in einer Richtung von $a \cdot 45$ liegen, mit $a = 0, 1, ..., 7$. Diese Werte werden als *Richtungscodes* bezeichnet. Üblicherweise sind sie wie in Bild 35.2 dargestellt festgelegt.

Bild 35.3 zeigt ein einfaches Beispiel für ein Segment. Der *chain*-Code der äußeren

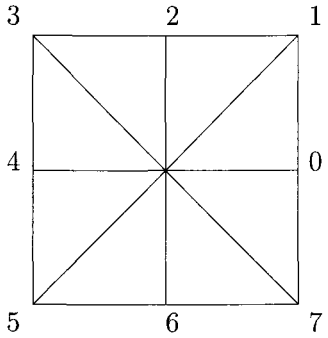


Bild 35.2: Richtungscodes der 8-Nachbarn.

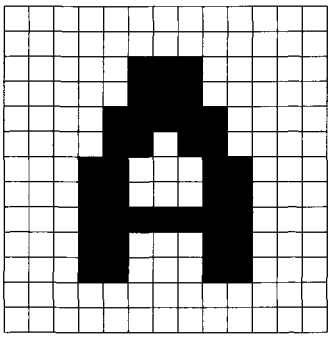


Bild 35.3: Beispiel für ein einfaches Segment.

Kontur dieses Segments sieht folgendermaßen aus:

040500200500676766664234456422221212

Diese Ziffernfolge muss noch etwas erläutert werden: Die Richtungscodekombination 04 wird als *Signalcode* verwendet, der anzeigt, dass danach „Sonderinformation" kommt. Eine tatsächlich auftretende Richtungscodekombination 04 wird dann mit 0404 codiert. In diesem Beispiel folgt ein weiterer Code 05, der anzeigt, dass anschließend die Koordinaten $(x, y) = (002, 005)$ des Startpunkts folgen. Die Richtungscodes der Kontur beginnen dann mit 00676...

Das Segment von Bild 35.3 enthält aber noch eine Aussparung innerhalb des Segments. Es wäre somit durch obigen *chain*-Code nicht zufriedenstellend wiedergegeben. Dieses Problem kann man wie folgt lösen: Es wird der Rand der Aussparung codiert und die beiden

Startpunkte der beiden Konturen werden durch eine „virtuelle Kette" verbunden (Signalcode 0421), die nur zur strukturellen Beschreibung des Segments dient. Damit könnte der *chain*-Code wie folgt aussehen:

040500200500676766664234456422221212	äußere Kontur
0421040500200576	virtuelle Kette
04050040067765443211	Kontur der Aussparung

Der *chain*-Code eignet sich nicht nur zur Codierung der Kontur eines Segments, sondern erlaubt auch die Berechnung von segmentbeschreibenden Parametern, z.B. die Länge der Kontur:

$$A = \Delta l(n_g + n_u\sqrt{2}). \tag{35.9}$$

Hier ist n_g die Anzahl der geraden und n_u die Anzahl der ungeraden Richtungscodes der Kette. Δl ist der Abstand der Gitterpunkte, die Länge der Kontur kann dann in der Einheit von Δl, z.B. in *mm* oder *cm*, angegeben werden.

Auch die vertikale und die horizontale Ausdehnung des Segments kann leicht aus der Kette ermittelt werden. Dazu ordnet man den Richtungscodes Richtungsvektoren zu, gemäß folgender Tabelle:

Richtungscode	x-Komponente	y-Komponente
0	0	1
1	-1	1
2	-1	0
3	-1	-1
4	0	-1
5	1	-1
6	1	0
7	1	1

Um die Ausdehnung in x- und y-Richtung zu berechnen, werden die zu den Richtungscodes gehörigen Komponenten der Richtungsvektoren getrennt für beide Richtungen akkumuliert und dabei das Minimum und das Maximum ermittelt.

35.9 Momente

In diesem Abschnitt wird eine Methode beschrieben, die es erlaubt, Segmente translations-, rotations- und größeninvariant zu beschreiben. Es wird ein Segment betrachtet, das aus der Menge der diskreten Punkte $\{(x, y)\}$ besteht. Hier muss das Segment nicht, wie in den vorhergehenden Abschnitten, einen homogenen Grauwert als Klassencode besitzen. Vielmehr wird angenommen, dass ein Segment die Grauwerte $s(x, y)$ besitzt. Ein zulässiger Sonderfall wäre, wenn alle $s(x, y)$ gleich sind (z.B. $s(x, y) = 1$).

Eine dazu passende Aufgabenstellung könnte etwa sein:

- Mit den Methoden der Klassifizierung wurde ein Bild segmentiert. Den verschiedenen Klassen wurden unterschiedliche Codes zugeordnet. Es kann natürlich vorkommen, dass verschiedene Segmente denselben Code besitzen, wenn ihre Punkte zur selben Klasse gehören.
- Die Segmente werden jetzt mit dem Original verknüpft, so dass für jedes Segment die Grauwerte (die Merkmalswerte) in einem bestimmten Kanal zur Verfügung stehen.
- Jetzt soll untersucht werden, ob zwei Segmente derselben Klasse ähnlich sind, also sich nur hinsichtlich der Position im Bild, der Skalierung (Größe) und der Drehlage unterscheiden.

Die *Momente der Ordnung* $p+q$ sind definiert durch:

$$m_{pq} = \sum_x \sum_y x^p y^q s(x,y) \tag{35.10}$$

In dieser Form hängen die Momente von der Position des Segments im Bild ab, so dass die oben angesprochene Translationsinvarianz nicht gegeben ist.
Aus diesem Grund definiert man die *zentrierten Momente der Ordnung* $p+q$:

$$\mu_{pq} = \sum_x \sum_y (x - x_{cp})^p (y - y_{cp})^q s(x,y). \tag{35.11}$$

Dabei sind x_{cp} und y_{cp} die Koordinaten des Schwerpunkts des Segments:

$$x_{cp} = \frac{\sum_x \sum_y x s(x,y)}{\sum_x \sum_y s(x,y)} = \frac{m_{10}}{m_{00}}, \tag{35.12}$$

$$y_{cp} = \frac{\sum_x \sum_y y s(x,y)}{\sum_x \sum_y s(x,y)} = \frac{m_{01}}{m_{00}}. \tag{35.13}$$

Falls das Segment homogen ist ($s(x,y) = 1$), tritt im Nenner von (35.12) und (35.13) die Fläche A des Segments auf. Nach der Koordinatentransformation gemäß

$$x' = x - x_{cp}, \qquad y' = y - y_{cp}, \tag{35.14}$$

liegt der Koordinatenursprung im Schwerpunkt, und es gilt: $\mu_{10} = \mu_{01} = 0$.
Nun zur Größennormierung: Die *normierten, zentrierten Momente der Ordnung* $p+q$ sind definiert durch:

$$\eta_{pq} = \frac{\mu_{pq}}{\mu_{00}^{\frac{p+q}{2}+1}} \tag{35.15}$$

Als Beispiel ist

$$\eta_{20} = \frac{\mu_{20}}{\mu_{00}^2} = \frac{\sum_x \sum_y (x - x_{cp})^2 s(x,y)}{(\sum_x \sum_y s(x,y))^2}, \tag{35.16}$$

und falls $s(x,y) = 1$ gilt, steht im Nenner das Quadrat A^2 der Fläche des Segments.

Die Momente η_{pq} sind invariant gegenüber Größenänderungen der Art $x' = ax, y' = ay$ des Segments, d.h. wenn in einem Bild an zwei unterschiedlichen Positionen zwei ähnliche Segmente auftreten, werden sich, abgesehen von Quantisierungsfehlern, gleiche Werte für η_{pq} ergeben.

Als Letztes wird die Rotationsinvarianz untersucht. Man erhält rotationsinvariante Momente, wenn das Koordinatensystem so gedreht wird, dass es mit den Hauptachsen des Segments übereinstimmt. Der Drehwinkel kann aus den Momenten berechnet werden:

$$\varphi = \frac{1}{2} \arctan \frac{2\eta_{11}}{\eta_{20} - \eta_{02}}. \tag{35.17}$$

Die Koordinatentransformation hat dann folgende Form:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}. \tag{35.18}$$

Aufbauend auf den Momenten wurden in [Hu62] sieben invariante Merkmale definiert, die im Folgenden dargestellt sind. Sie werden aus den normierten, zentrierten Momenten der Ordnung $p + q$ berechnet. Eine Transformation gemäß (35.17) ist nicht notwendig:

$$\begin{aligned}
c_1 &= \eta_{20} + \eta_{02} \\
c_2 &= (\eta_{20} - \eta_{02})^2 + 4\eta_{11}^2 \\
c_3 &= (\eta_{30} - 3\eta_{12})^2 + (3\eta_{21} - \eta_{03})^2 \\
c_4 &= (\eta_{30} + \eta_{12})^2 + (\eta_{21} + \eta_{03})^2 \\
c_5 &= (\eta_{30} - 3\eta_{12})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\
&\quad +(3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
c_6 &= (\eta_{20} - \eta_{02})[(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2] \\
&\quad +4\eta_{11}(\eta_{30} + \eta_{12})(\eta_{21} + \eta_{03}) \\
c_7 &= (3\eta_{21} - \eta_{30})(\eta_{30} + \eta_{12})[(\eta_{30} + \eta_{12})^2 - 3(\eta_{21} + \eta_{03})^2] \\
&\quad +(3\eta_{21} - \eta_{03})(\eta_{21} + \eta_{03})[3(\eta_{30} + \eta_{12})^2 - (\eta_{21} + \eta_{03})^2]
\end{aligned} \tag{35.19}$$

Da diese Merkmale sehr unterschiedliche Dynamikbereiche besitzen, wurden sie in den folgenden Bildbeispielen durch Logarithmierung normiert:

$$c_i \Leftarrow -lnc_i. \tag{35.20}$$

Im Folgenden werden einige Beispiele zu diesen sieben invarianten Parametern gegeben. Bild 35.4 zeigt Segmente, die sich durch die Position im Bild, die Größe und die Drehlage unterscheiden.

In der folgenden Tabelle sind für jedes Segment die sieben Parameter zusammengestellt. Man sieht, dass die Werte ungefähr gleich sind. Die Abweichungen sind durch die Fehler der diskreten Verarbeitung zu erklären. Zur weiteren Verarbeitung dieser Merkmalsvektoren können Klassifikatoren oder neuronale Netze verwendet werden. So könnte man z.B. ableiten, dass es sich hier immer um dasselbe Objekt in unterschiedlichen Lagen handelt.

Parameter	Original	andere Position	andere Größe	gedreht 5°	gedreht 123°
c_1	6.042	6.042	6.045	6.046	6.025
c_2	12.981	12.981	12.985	12.991	12.953
c_3	21.437	21.437	21.623	21.646	21.594
c_4	21.968	21.968	21.962	21.982	21.887
c_5	43.674	43.674	43.763	43.806	43.639
c_6	28.492	28.492	28.481	28.640	33.619
c_7	43.720	43.720	43.774	44.914	44.007

In Bild 35.5 sind sieben verschieden geformte Segmente abgebildet. Die folgende Tabelle enthält die Werte der c_i-Parameter. Zu einigen Parametern konnte der Logarithmus nicht berechnet werden, da der c_i-Wert nahezu null war. Diese Werte sind mit -1 markiert. Man sieht bei diesem Beispiel, dass sich durchaus unterschiedliche Werte ergeben und mit einer geeigneten Auswahl von Parametern die Segmente leicht unterschieden werden können.

Parameter	Niere	Kreis	Sichel	Blatt	Eck	Dreieck	Viereck
c_1	6.726	6.878	5.950	6.961	6.705	6.708	6.640
c_2	14.546	15.662	12.528	18.801	14.843	16.745	14.148
c_3	23.063	28.934	18.712	26.586	21.677	20.604	28.908
c_4	24.714	30.316	20.003	29.078	24.622	25.368	28.566
c_5	48.642	-1.000	39.854	-1.000	47.799	49.431	57.305
c_6	33.727	38.210	26.772	-1.000	32.165	34.886	38.063
c_7	48.984	-1.000	44.073	-1.000	49.056	48.581	57.142

Bild 35.6 behandelt eine weitere Problemstellung: Es sind von zwei verschiedenen Zangentypen jeweils vier Zangen in unterschiedlichen Drehlagen und Größen angeordnet. Die dazugehörige Tabelle zeigt, dass sich hier auf den ersten Blick keine deutlichen und verwendbaren Unterschiede ergeben. Sieht man sich jedoch die Parameter c_3 und c_4 genauer an, so ist zu erkennen, dass auch hier eine Klassifizierung in die Klassen „Typ 1“und „Typ 2“ möglich ist.

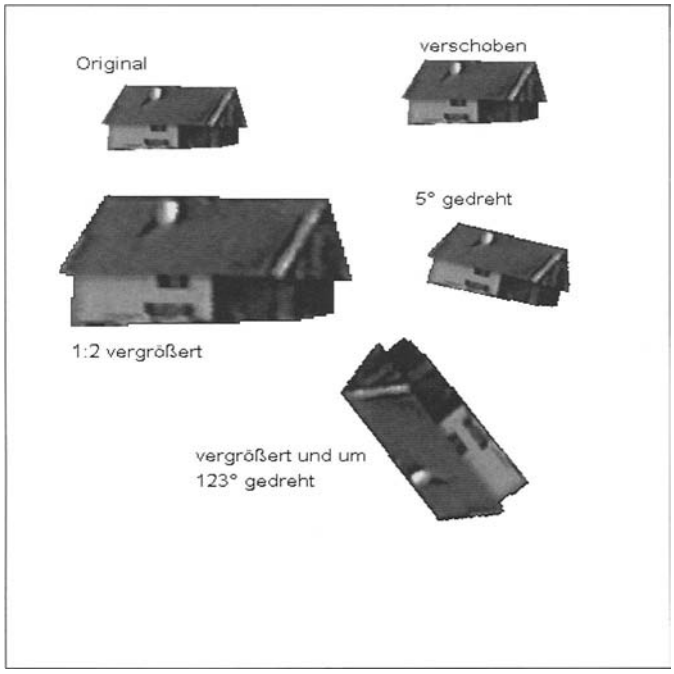


Bild 35.4: Beispiel für Segmente, die sich durch die Position im Bild, die Größe und die Drehlage unterscheiden. Die aus den Momenten abgeleiteten sieben Parameter sind ungefähr gleich (siehe Tabelle).

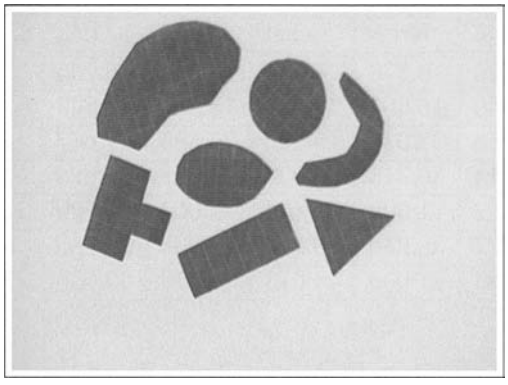


Bild 35.5: Beispiel für unterschiedlich geformte Segmente. Mit einer geeigneten Parameterauswahl können die Segmente leicht unterschieden werden (siehe Tabelle).

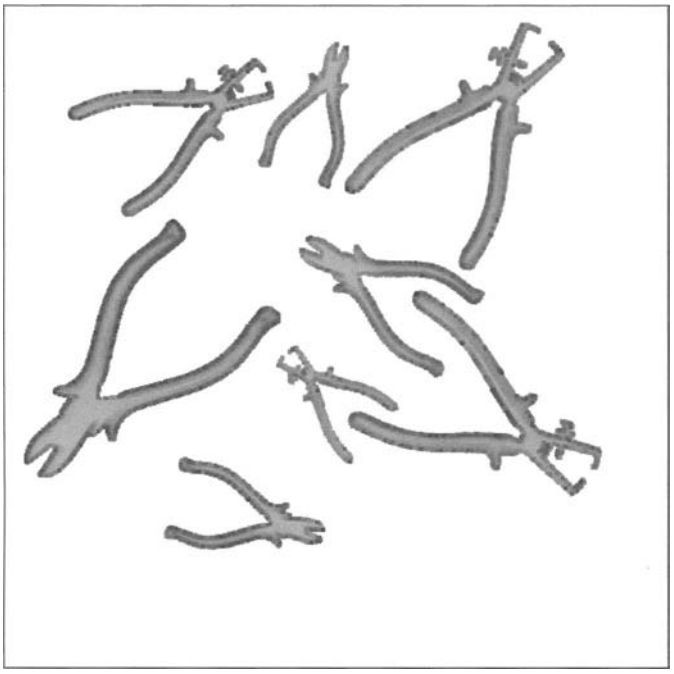


Bild 35.6: Das Bild enthält zwei Zangen in jeweils vier unterschiedlichen Drehlagen und Größen. Die Tabelle der c_i-Werte zeigt hier keine so deutlichen Unterschiede. Die Parameter c_3 und c_4 machen in diesem Beispiel aber eine Trennung der Klassen möglich.

Parameter	Typ 1				Typ 2			
c_1	4.654	4.736	4.703	4.761	4.798	4.706	4.747	4.869
c_2	11.175	11.285	11.263	11.339	11.103	10.913	11.025	11.255
c_3	14.870	15.147	15.036	15.255	15.613	15.372	15.456	15.867
c_4	17.070	17.203	17.134	17.279	18.600	17.993	18.330	18.515
c_5	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000	-1.000
c_6	25.105	23.755	24.261	-1.000	24.933	-1.000	26.213	24.661
c_7	-1.000	34.056	35.519	-1.000	-1.000	-1.000	36.899	-1.000

Die Momente wurden in der obigen Darstellung flächenhaft anhand aller Punkte eines Segments berechnet. Soll bei einer Anwendung nur der Rand des Segments verarbeitet werden, so können Momente auch nur für die Randpunkte berechnet werden. Dazu führt man folgende Schritte durch:

- Man berechnet den Schwerpunkt des Segments.
- Für jeden Punkt i der Kontur berechnet man den Abstand d_i zum Schwerpunkt.

- Man berechnet die relativen Häufigkeiten der Abstände d_i. Dazu müssen die Distanzen mit einer passenden Einheit (z.B. *Pixel*) diskretisiert werden. Angenommen, es gibt k verschiedene Distanzen, die mit a_1 bis a_k bezeichnet werden. $p(a_j)$ ist die relative Häufigkeit der Distanz a_j.

- Die zentrierten Momente $q-ter$ Ordnung sind dann:

$$\mu_p = \sum_{j=1}^{k} (a_j - m)^q p(a_j) \tag{35.21}$$

mit

$$m = \sum_{j=1}^{k} a_j p(a_j) \tag{35.22}$$

Die Größe m ist die mittlere Länge der Distanzen und μ_2 ein Schätzwert für die Streuung. Mit den so berechneten Momenten kann sinngemäß wie bei der flächenhaften Betrachtung vorgegangen werden.

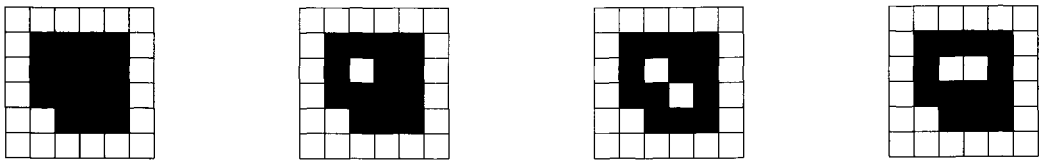
35.10 Euler'sche Charakteristik

Ein wichtiges Merkmal zur Beschreibung und Unterscheidung von Segmenten ist die Anzahl der „Löcher", die das Segment besitzt. Als „Loch" bezeichnet man eine Ansammlung von zusammenhängenden Bildpunkten (4-Nachbarschaft), die nicht zum Segment gehören, aber ganz von Bildpunkten des Segments umgeben werden. Zum Zählen der Löcher kann die Euler'sche Charakteristik verwendet werden.

In Abschnitt 20.9 wurde die Euler'sche Charakteristik $E = e - k + f$ in einem anderen Zusammenhang vorgestellt. In dieser Formel ist e die Anzahl der Ecken, k die Anzahl der Kanten und f die Anzahl der Flächen. Ecken oder Kanten, die zu mehreren Bildpunkten des Segments gehören, werden nur einmal gezählt. Für ein Polygonnetz gilt $E = 1$. Betrachtet man die Segmente in Bild 35.7, erhält man für das Segment in Bild 35.7-a $E = 1$, für Bild 35.7-b $E = 0$, für Bild 35.7-c $E = -1$ und für Bild 35.7-d $E = 0$. Die Anzahl L der Löcher ist dann

$$L = 1 - E. \tag{35.23}$$

Bei einer praktischen Anwendung kann man mit diesem Verfahren die Anzahl der Löcher der Segmente bestimmen. An einer unrichtigen Löcheranzahl könnte man z.B. fehlerhafte Teile erkennen. Eine andere Möglichkeit wäre, dass man nur die Segmente, die eine bestimmte Anzahl von Löchern haben, weiter verarbeitet und evtl. aufwändigere Verfahren darauf anwendet.



(a) (b) (c) (d)

Bild 35.7: Beispiele zur Euler'schen Charakteristik zum Zählen der Löcher eines Segments (schwarze Segmentbildpunkte): (a) e=24, k=38 und f=15, also E=1. (b) e=24, k=38 und f=14, also E=0. (c) e=24, k=38 und f=13, somit E=−1. (d) e=24, k=37 und f=13, somit$E$=0. Die Größe $L = 1 - E$ gibt die Anzahl der Löcher an.

35.11 Auswahl von Segmenten mit morphologischen Operationen

Wenn man bei einer aktuellen Problemstellung die Aufgabe hat, die Umgebung eines bestimmten Segments aus einem Bild auszublenden, so dass im Ergebnisbild nur mehr dieses Segment vorhanden ist, so kann man auch mit einfachen morphologischen Operationen arbeiten. Die Grundlagen der morphologischen Operationen sind in Kapitel 19 zusammengestellt.

Angenommen, es liegt ein Zweipegelbild mit verschiedenen Segmenten vor (Hintergrundgrauwert 0, Segmente mit Grauwert 255). Um ein bestimmtes Segment herauszugreifen, geht man wie folgt vor:

- Das Eingabebild wird in ein Eingabefeld eingelesen.
- Es wird ein beliebiger Punkt des Segments (z.B. der Schwerpunkt) als „Initialpunkt" definiert.
- In einem Ausgabefeld, das die Größe des Eingabefeldes hat, wird nur dieser Punkt auf den Grauwert 255 gesetzt. Alle anderen Punkte haben den Grauwert 0.
- Nun wird das Ausgabefeld dilatiert. Der ursprüngliche Initialpunkt wird um einen ein Pixel breiten Rand vergrößert.
- Jetzt wird die logische UND-Operation mit dem Eingabefeld durchgeführt. Falls durch die Dilatation Punkte dazu gekommen sind, die nicht zum gewünschten Segment gehören, werden diese durch diese Operation wieder eliminiert.
- Jetzt wird wieder das Ausgabefeld dilatiert, dann die UND-Operation ausgeführt.
- Diese Operationen werden so lange wiederholt, bis sich die Fläche der Bildpunkte mit dem Grauwert 255 vor und nach der UND-Verknüpfung nicht mehr ändert.

- Im Ausgabefeld ist nur mehr das gewünschte Segment.

Dieses Verfahren, das auch *grassfire*-Operation genannt wird [Abma94], kann auch verwendet werden, um Störungen am Bildrand zu eliminieren. Als Initialpunkte kann man die Randpunkte des Bildes verwenden. Das Ergebnisbild wird vom Ausgangsbild abgezogen, so dass nur mehr die Segmente übrig bleiben, die den Bildrand nicht berühren.

35.12 Segmentbeschreibung mit Fuzzy Logic

In diesem Abschnitt soll der Versuch kurz dargestellt werden, Segmente mit Methoden der *fuzzy logic* zu klassifizieren. Die Grundlagen dazu wurden in Kapitel 33 erläutert. Die Vorgehensweise ist etwa wie folgt:

- Transformation der Segmente in eine geeignete Datenstruktur, etwa *run-length*-Code.
- Berechnung von segmentbeschreibenden Merkmalen: Fläche, Umfang, Schwerpunkt, Orientierung, Momente, usw.
- Normierung der Segmente bezüglich Translation (Normierung über die Schwerpunktkoordinaten), Rotation und Größe (Normierung über Momente, siehe Abschnitt 35.9).
- Berechnung eines Merkmalsvektors für jedes Segment mit geeigneten Merkmalen.
- *fuzzy clustering* gemäß Kapitel 33.
- Zuordnung des Segments zur Klasse (zum *cluster*) mit dem höchsten Mitgliedsgradwert oder weitere Verwendung der *fuzzy*-Mengen der einzelnen Segmente bei anschließenden Verarbeitungsschritten.

Praktische Erfahrungen haben gezeigt, dass dieses Verfahren gut funktioniert. Es hat sich vor allem als sehr vorteilhaft erwiesen, dass für eine gewisse Anzahl von Segmenttypen, die die verschiedenen Klassen festlegen, nur ein einziger Trainingslauf benötigt wurde.

Kapitel 36

Segmentbeschreibung mit dem Strahlenverfahren

36.1 Anwendungen

In den vorangehenden Kapiteln wurden viele Beispiele gegeben, wie segmentbeschreibende Merkmale und Datenstrukturen berechnet werden können. Einige dieser Merkmale waren unabhängig von der Lage des Segments (z.B. der Flächeninhalt), andere unabhängig von der Größe des Segments (z.B. die Orientierung). Bei Datenstrukturen, die ebenfalls zur Charakterisierung von Segmenten verwendet werden können, wie z.B. der *run-length*-Code oder der *chain*-Code, hängt es von der Implementierung ab, ob sie lageinvariant sind oder nicht. Auf jeden Fall sind beide Verfahren nicht orientierungsunabhängig.

Wenn Segmente in Klassen eingeteilt werden sollen (z.B. „Bauteil vom Typ X/Y“ oder „defektes/nicht defektes Bauteil“), so wird man genau diejenigen charakteristischen Merkmale verwenden, die bei einer gegebenen Problemstellung gerade noch zur Unterscheidung ausreichen. Wenn ein einfaches Merkmal, wie der „Flächeninhalt“, zur Unterscheidung genügt, so wäre es unsinnig und würde den Rechenaufwand nur unnötig erhöhen, wenn man weitere, aufwändigere Segmentmerkmale mit verwenden würde.

Bei manchen Anwendungen, bei denen die Segmente komplexe Formen besitzen und durch einfache Merkmale nicht zu unterscheiden sind, muss man etwas mehr Aufwand betreiben. Eine zusätzliche Forderung kann noch sein, dass man die Segmente, unabhängig von ihrer Größe, ihrer Position im Bildausschnitt und ihrer Drehlage, immer erkennt. Dazu wurden Verfahren zur rotations-, translations- und größeninvarianten Segmenterkennung entwickelt.

In diesem Kapitel und im folgenden Kapitel 37 werden zwei Verfahren zur dieser Problemstellung erläutert, anhand von Beispielen dargestellt und auf ihre Leistungsfähigkeit untersucht. Beide Verfahren tasten den Rand eines Segments ab und verwenden den so entstehenden Merkmalsvektor der Distanzen zum Rand als charakterisierendes Merkmal. Das erste Verfahren, das *Strahlenverfahren*, verwendet einen multivariaten Klassifikator. Das zweite Verfahren arbeitet mit einem neuronalen Backpropagation-Netz.

36.2 Prinzipieller Ablauf des Strahlenverfahrens

Das Strahlenverfahren arbeitet nach dem Prinzip einer überwachten Klassifizierungsstrategie, die fest dimensioniert oder lernend implementiert werden kann (Kapitel 23 und 31). Im hier beschriebenen Fall wird die feste Dimensionierung gewählt. Der prinzipielle Ablauf der *Trainingsphase* des Strahlenverfahrens ist im folgenden Algorithmus zusammengestellt:

36.1: Trainingsphase des Strahlenverfahrens.

Algorithmus:

(a) Einzug des Bildes über einen Videosensor (oder für Testbeispiele aus einer Datei).

(b) Segmentierung des Bildes.

(c) Kompakte Speicherung der Segmente mit Hilfe spezieller Datenstrukturen (z.B. *run-length*-Code).

(d) Für alle zu trainierenden Segmente:

(da) Auswahl eines Segments für das Training.

(db) Aufbau der Trainingsdaten.

(dc) Eintragen der Trainingsdaten in die Trainingsdatei.

(e) Dimensionierung eines multivariaten Klassifikators.

(f) Abspeichern der Ergebnisse des Trainings in einer Datei.

Ende des Algorithmus

In der *Erkennungsphase* (*Klassifizierungsphase*, *Produktionsphase*, *recall*-Phase) werden Segmente verarbeitet, deren Zugehörigkeit zu einer der möglichen Klassen nicht bekannt ist. Mit Hilfe des trainierten Klassifikators kann eine Zuordnung zu einer Klasse erfolgen. Die Produktionsphase ist der Trainingsphase sehr ähnlich:

36.2: Klassifizierungsphase des Strahlenverfahrens.

Algorithmus:

(a) Einzug des Bildes über einen Videosensor.

(b) Segmentierung des Bildes.

(c) Kompakte Speicherung der Segmente mit Hilfe spezieller Datenstrukturen (z.B. *run-length*-Code).

(d) Einlesen der Dimensionierungsdaten des Klassifikators aus einer Datei.

(e) Für alle zu erkennenden Segmente:

(ea) Auswahl eines Segments.

(eb) Aufbau des Merkmalsvektors.

(ec) Klassifizierung des Merkmalsvektors.

Ende des Algorithmus

Der wichtigste Punkt in der Trainingsphase ist der Aufbau der Trainingsdaten. Dieser Problemkreis wird im folgenden Abschnitt behandelt.

36.3 Aufbau des Merkmalsvektors für ein Segment

Der Grundgedanke beim Strahlenverfahren ist die Tatsache, dass der Rand eines Segments zu seiner Charakterisierung verwendet werden kann. Es muss aber ein Verfahren verwendet werden, das der Forderung nach der Translations-, Rotations- und Größeninvarianz gerecht wird. Dazu werden, um ein Segment möglichst eindeutig zu beschreiben, Strahlen ermittelt, indem man die Distanzen von bestimmten Randpunkten zum Schwerpunkt berechnet und als Merkmale verwendet. Die Randpunkte eines Segments werden durch dessen Kontur festgelegt. In die Berechnung werden jedoch nicht alle Randpunkte einbezogen, sondern nur die Schnittpunkte der Kontur mit Strahlen, die in gleichen Winkelabständen vom Schwerpunkt ausgehen. Dazu werden zu einem zu trainierenden Segment

- der Schwerpunkt berechnet,
- die Orientierung ermittelt,
- die Randpunkte bestimmt und,
- ausgehend vom Schwerpunkt und nach Maßgabe der Orientierung, ein Merkmalsvektor mit Distanzen Schwerpunkt/Rand in vorgegebenen Winkelintervallen berechnet.

Die Berechnung des Segmentschwerpunkts und der Randpunkte ist einfach und wurde bereits in den Abschnitten 35.3 und 35.8 dargestellt.

Die Berechnung der Orientierung des Segments ist ein wesentliches Kernstück des Strahlenverfahrens, da ausgehend von der Orientierung der Längen der Strahlen Schwerpunkt/Rand berechnet werden.

Verfahren zur Orientierung wurden in den Abschnitten 35.6 und 35.9 erläutert: Es werden aus der Kovarianzmatrix des Segments die Eigenwerte berechnet und zur Ermittlung der Hauptrichtung verwendet. Praktische Tests haben gezeigt, dass dieses Verfahren nur verwendet werden kann, wenn die in Frage kommenden Segmente eine ausgeprägte

Hauptrichtung besitzen, also länglich sind. Bei Segmenten, bei denen das nicht der Fall ist, wird sich mehr oder weniger zufällig eine Hauptrichtung ergeben, die bei unterschiedlichen Drehlagen auch unterschiedlich sein kann. Dieses Verfahren kann dann nicht eingesetzt werden.

Eine andere Möglichkeit, die sich als verhältnismäßig stabil gezeigt hat, ist die Bestimmung der Orientierung durch die Ermittlung der kürzesten Distanz Rand/Schwerpunkt/Rand (Bild 36.1-a).

Bei diesem Verfahren werden viele Segmentformen berücksichtigt. Sicherlich finden sich Segmente, bei denen auch durch diese Methode keine eindeutige Drehlage ermittelt werden kann, d.h. wenn mehr als eine kürzeste Distanz vorkommt. Zu bemerken ist noch, dass das Verfahren mit der Berechnung der Orientierung über die Eigenwerte kompatibel ist: Bei einem ausgeprägt länglichen Segment stimmen die berechneten Orientierungen der beiden Verfahren überein.

Zur Bestimmung der Orientierung mit diesem Verfahren werden zu allen Randpunkten die Distanzen zum Schwerpunkt berechnet. Dann werden die Strahlen Schwerpunkt/Rand zu Strecken Rand/Schwerpunkt/Rand ergänzt, indem zu einem Strahl, wenn möglich, derjenige Strahl gesucht wird, der um 180° weiter liegt. Diese Ergänzung wird meistens nicht mit einem exakten Winkel von 180° möglich sein, so dass hier bei der Implementierung Toleranzen eingebaut werden müssen. Aus diesen Strecken wird dann die kürzeste gesucht und die zugehörige Richtung als Hauptrichtung verwendet.

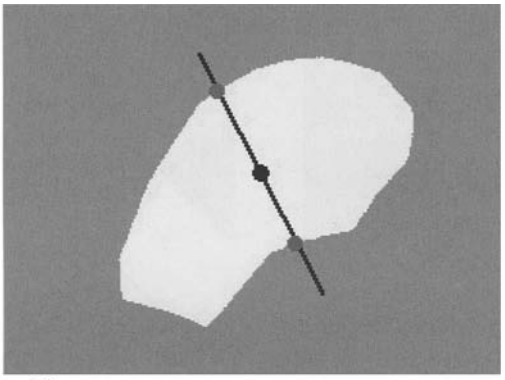
Wenn ein Segment theoretisch mehrere gleiche Hauptrichtungen besitzt (Bild 36.1-b), so werden sich bei einer praktischen Analyse doch geringfügige Längenunterschiede ergeben, so dass sich ein Minimum ermitteln lässt. Das Problem ist dann allerdings, dass bei einer anderen Drehlage eine andere Hauptrichtung zufällig den minimalen Wert annimmt und somit die Berechung der Orientierung nicht mehr eindeutig ist. In diesem Fall kann man versuchen, das Segment mehrmals zu trainieren. Falls auch das nicht möglich ist, ist das Verfahren zur Bestimmung der Hauptrichtung nicht verwendbar.

Wenn der Schwerpunkt außerhalb des Segments liegt, muss die Bestimmung der Orientierung etwas anders durchgeführt werden. Man kann hier z.B. die kürzeste Distanz Rand/Rand in Richtung des Schwerpunkts verwenden (Bild 36.1-c).

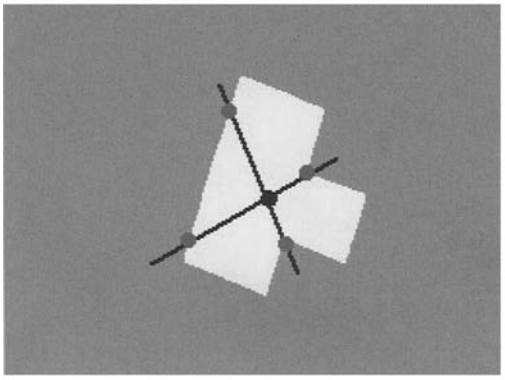
Nachdem die Orientierung des Segments bestimmt ist, wird im nächsten Schritt ein Merkmalsvektor mit Distanzen Schwerpunkt/Rand berechnet. Dazu können die bei der Bestimmung der Orientierung berechneten Distanzen aller Randpunkte zum Schwerpunkt verwendet werden. Der Vollkreis wird in vorgegebene Winkelintervalle eingeteilt, wobei die Richtung 0° durch die berechnete Hauptrichtung des Segments (oder senkrecht dazu) festgelegt ist (Bild 36.2-a).

Jetzt werden für jede Strahlenrichtung die Schnittpunkte mit dem Rand des Segments gesucht. Das wird in der Praxis nicht exakt möglich sein, da ja die Randpunkte nicht genau auf den gewählten Strahlenrichtungen liegen müssen. In diesem Fall werden die Randpunkte gewählt, die der jeweiligen Richtung am nächsten liegen (Bild 36.2-b).

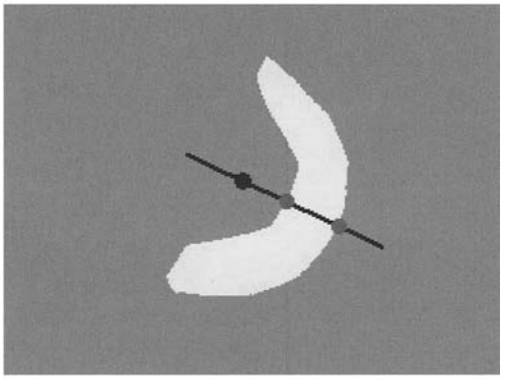
Es wird häufig der Fall sein, dass sich in bestimmten Richtungen mehrere Schnittpunkte mit dem Rand ergeben (Bild 36.2-c). Alle diese Schnittpunkte werden zunächst erfasst. Erst bei der Aufstellung des Merkmalsvektors wird entschieden, welche davon verwendet



(a)



(b)



(c)

Bild 36.1: (a) Ermittlung der Orientierung eines Segments durch das Finden der kürzesten Distanz Rand/Schwerpunkt/Rand. (b) Manche Segmente besitzen mehrere gleichgroße Distanzen dieser Art. (c) Liegt der Schwerpunkt außerhalb des Segments, so kann die kürzeste Distanz Rand/Rand in Richtung Schwerpunkt verwendet werden.

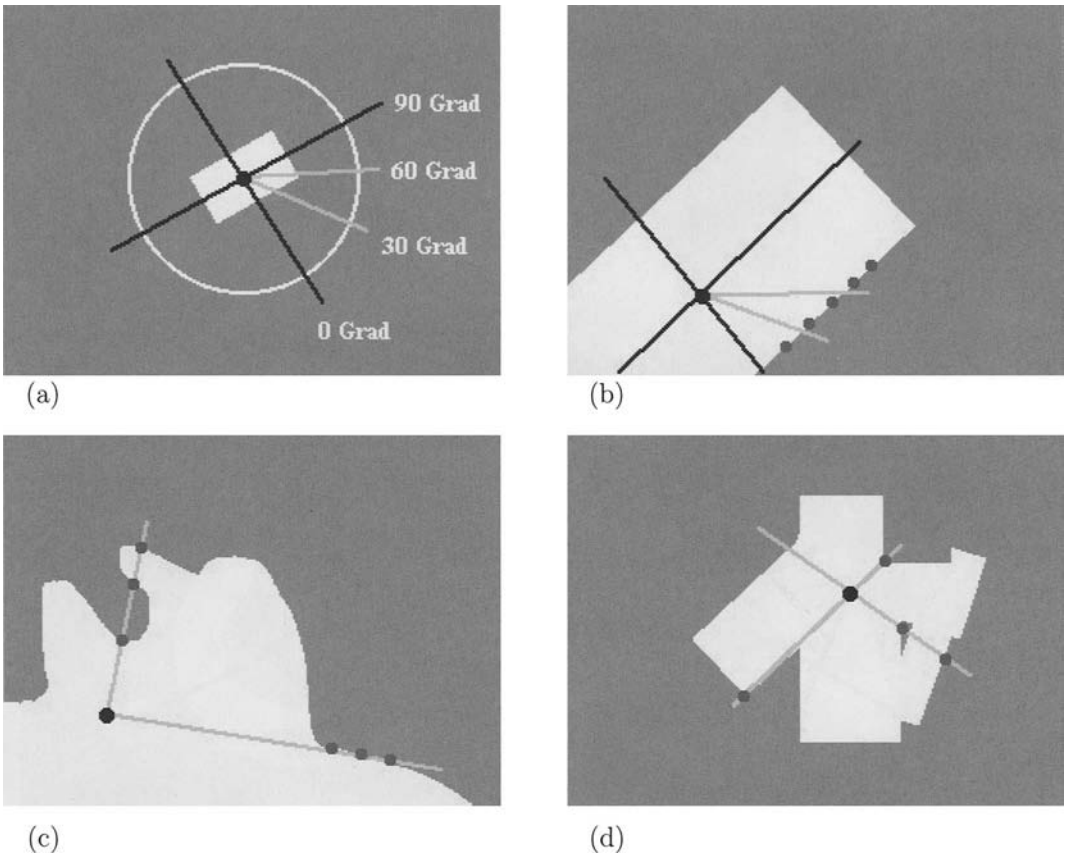


Bild 36.2: (a) Einteilung des Vollkreises in vorgegebene Winkelintervalle. Die Richtung 0° ist durch die berechnete Hauptrichtung des Segments (oder senkrecht dazu) festgelegt. (b) Die Randpunkte liegen nicht exakt auf den gewählten Strahlenrichtungen. Es werden die Randpunkte verwendet, die der jeweiligen Richtung am nächsten liegen. (c) Bei einigen Richtungen ergeben sich möglicherweise mehrere Schnittpunkte mit dem Rand des Segments. (d) Beim Aufbau des Merkmalsvektors, der das Segment charakterisieren soll, wird pro Richtungspaar die minimale und die maximale Distanz verwendet.

werden.

Nachdem zu den Richtungsstrahlen alle Distanzen zum Rand berechnet sind, kann ein Merkmalsvektor aufgebaut werden, der die Kontur des Segments charakterisieren soll. Hier muss man sich entscheiden, welche der Distanzen dazu verwendet werden sollen. Im vorliegenden Fall werden pro Richtungsstrahlenpaar (eine Richtung zusammen mit der um 180° weiter liegenden Richtung, z.B. 0° und 180°, 12° und 192°,...) jeweils die minimale und die maximale Distanz verwendet (Bild 36.2-d). Bei einem Winkelintervall von 12° werden somit 180/12=15 Richtungen verwendet. Der Merkmalsvektor ist dementsprechend 30-dimensional.

Mit dieser Vorgehensweise bei der Berechnung der Merkmalsvektoren wurde die erste Forderung, nämlich die Rotationsinvarianz der Merkmale, erfüllt, da ja die Distanzen nach Maßgabe der ermittelten Hauptrichtung berechnet wurden. Es muss an dieser Stelle nochmals darauf hingewiesen werden, dass das Verfahren voraussetzt, dass die Hauptrichtung mit genügender Genauigkeit bestimmt werden kann.

Die Größeninvarianz wird durch Normierung der Distanzen erreicht. Dazu folgende Überlegungen: Bei einem Quadrat Q_1 mit der Seitenlänge a ist die Fläche $A_1 = a^2$ und die Länge der Diagonalen $d_1 = a\sqrt{2}$. Bei einem Quadrat Q_2 mit der doppelten Seitenlänge $2a$ ergibt sich: $A_2 = 4a^2$ und $d_2 = 2a\sqrt{2}$. Wenn die Strecken d_1 und d_2 dadurch normiert werden, dass sie durch die Wurzel aus ihrem Flächeninhalt dividiert werden, ergibt sich in beiden Fällen der Wert $\sqrt{2}$, obwohl das zweite Quadrat eine viermal so große Fläche besitzt. Dieser Sachverhalt wird zur Normierung der Distanzen verwendet: Es wird jede Distanz durch die Wurzel aus dem Flächeninhalt des Segments dividiert.

Die folgende Tabelle zeigt die verwendeten normierten Distanzen für ein Segment, das eine ähnliche Lage und Form hat wie das Segment von Bild 36.2-b:

Richtung	maximale Distanz	minimale Distanz
0°/180°	38.41	34.94
12°/192°	39.88	35.50
24°/204°	43.26	37.74
36°/216°	49.66	40.81
48°/228°	58.23	48.91
60°/240°	72.49	61.17
72°/252°	93.94	86.62
84°/264°	89.85	88.33
96°/276°	89.82	88.36
108°/288°	94.15	90.02
120°/300°	95.61	79.61
132°/312°	76.28	56.43
144°/324°	54.72	45.81
156°/336°	44.43	40.04
168°/348°	38.83	36.15

Bild 36.3: Drei Verkehrszeichen als Testbeispiele für das Strahlenverfahren: „Viehtrieb, Tiere", „Wildwechsel", „Flugbetrieb".

Nachdem zu allen in Frage kommenden Segmenten die Merkmalsvektoren berechnet wurden, kann ein multivariater Klassifikator dimensioniert werden. Im vorliegenden Beispiel wird ein Minimum-Distance-Klassifikator verwendet (Abschnitt 31.5). Mit einem geeigneten Zurückweisungsradius kann zusätzlich eine Klasse „nicht klassifizierbar" eingeführt werden, in die Segmente eingeteilt werden, die zu weit von den *cluster*-Zentren entfernt liegen.

36.4 Klassifizierungs- / Produktionsphase

Wie der Algorithmus von Abschnitt 36.2 zeigt, ist die Klassifizierungs- oder Produktionsphase sehr ähnlich der Trainingsphase. Für ein zu untersuchendes Segment wird in der Weise wie in Abschnitt 36.3 beschrieben ein Merkmalsvektor berechnet. Nach dem Training ist der Klassifikator dimensioniert. Somit kann ein unbekannter Merkmalsvektor einer Klasse (oder der Zurückweisungsklasse) zugewiesen werden.

36.5 Strahlenverfahren: Ein Beispiel

Zum Abschluss ein Beispiel, das dieses Verfahren illustrieren soll. Im Bereich „intelligente Fahrzeuge" könnte man z.B. Systeme vorsehen, die Verkehrszeichen erkennen. Dabei ist zunächst die Aufgabe zu lösen, ein Verkehrszeichen aus seiner Bildumgebung zu isolieren. Im zweiten Schritt kann man versuchen, den Sinngehalt des Verkehrszeichens zu ermitteln. Für das Beispiel wurden die drei Verkehrszeichen „Viehtrieb, Tiere", „Wildwechsel" und „Flugbetrieb" gewählt (Bild 36.3).

In Vorverarbeitungsschritten wurden die drei Symbole segmentiert. Im nächsten Schritt wurden zu jedem Symbol in zwölf verschiedenen Drehlagen (0°, 30°, ..., 330°, Bilder 36.4-a

bis 36.4-c) wie oben beschrieben die Merkmalsvektoren berechnet und damit ein Minimum-Distance-Klassifikator trainiert. Dabei wurden pro Segment 15 Richtungen gesetzt, so dass die Merkmalsvektoren 30-dimensional waren. Für das Segment „Kuh, Lage 4“ (links oben) sind in der folgenden Tabelle sämtliche Distanzen aufgeführt. Die im Merkmalsvektor verwendeten Distanzen sind mit einem (*) markiert:

Richtung	Schnitte
0°/180°	33.94* 20.95*
12°/192°	67.87* 37.43 20.95*
24°/204°	67.59* 60.83 42.10 21.80*
36°/216°	61.86* 55.16 50.31 24.47*
48°/228°	102.83* 97.95 92.16 86.27 80.46 59.35 38.61 34.80 28.36*
60°/240°	111.65* 88.96 71.19 60.50*
72°/252°	130.79* 95.71 92.26 87.70 83.99*
84°/264°	98.09* 93.90 82.79 69.31*
96°/276°	100.46* 85.50 61.99*
108°/288°	103.84* 97.51 59.12*
120°/300°	72.50* 63.15*
132°/312°	88.17* 84.55 66.80 56.21 47.15*
144°/324°	51.48* 44.75 42.43 38.61 25.74*
156°/336°	68.47* 64.62 59.39 38.48 22.01*
168°/348°	36.03* 21.80*

In der Produktionsphase wurde jedes Symbol in vier unterschiedlichen Drehlagen dem Klassifikator angeboten (Bild 36.4-d). Das Ergebnis der Klassifizierung ist in folgender Tabelle zusammengefasst:

Eingabesymbol	klassifiziert zu Klasse	Soll
Segment „Kuh“, Lage 1	1	1
Segment „Kuh“, Lage 2	1	1
Segment „Kuh“, Lage 3	1	1
Segment „Kuh“, Lage 4	1	1
Segment „Hirsch“, Lage 1	2	2
Segment „Hirsch“, Lage 2	2	2
Segment „Hirsch“, Lage 3	2	2
Segment „Hirsch“, Lage 4	2	2
Segment „Flugzeug“, Lage 1	3	3
Segment „Flugzeug“, Lage 2	3	3
Segment „Flugzeug“, Lage 3	3	3
Segment „Flugzeug“, Lage 4	3	3

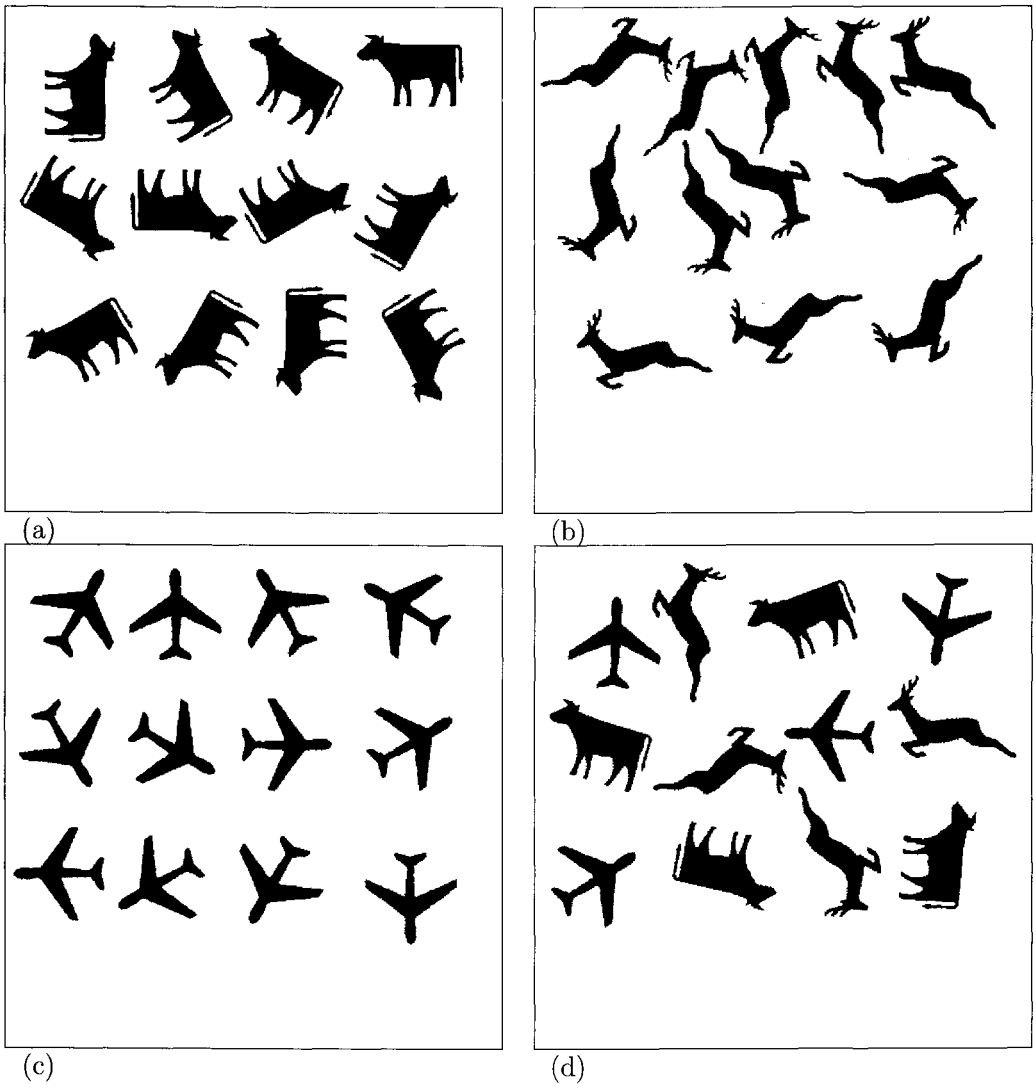


Bild 36.4: Jedes Symbol wurde in zwölf Drehlagen trainiert. (a) Kuh. (b) Hirsch. (c) Flugzeug. (d) Zur Klassifizierung wurde jedes Symbol viermal in unterschiedlichen Drehlagen, die nicht mit den trainierten Drehlagen identisch waren, verarbeitet. Es wurden alle Symbole richtig klassifiziert.

Die Ergebnisse zeigen, dass in diesem Fall alle Segmente richtig erkannt wurden. Weitere Tests mit unterschiedlich großen Symbolen ergaben ebenfalls gute Ergebnisse. Allerdings sind hier den erkennbaren Segmentgrößen nach unten und oben Grenzen gesetzt (etwa halber bzw. doppelter Flächeninhalt).

Kapitel 37

Neuronale Netze und Segmentbeschreibung

37.1 Anwendungen

Die Anwendungen sind hier dieselben wie im vorhergehenden Kapitel: Die translations-, rotations- und größeninvariante Erkennung von Segmenten. Es wird sich zeigen, dass hier das Training einfacher ist als im Fall des Strahlenverfahrens.

37.2 Prinzipieller Ablauf

Die theoretischen Grundlagen zu neuronalen Netzen sind in Kapitel 32 zusammengefasst. Im vorangehenden Kapitel 36 wurde auf die Thematik der translations-, rotations- und größeninvarianten Segmenterkennung ausführlich eingegangen. In diesem Abschnitt wird beschrieben, wie neuronale Netze in diesem Zusammenhang zur Erkennung von Segmenten eingesetzt werden können.

Der prinzipielle Ablauf ist ähnlich wie beim Strahlenverfahren in Kapitel 36. Auch hier kann eine Einteilung in die Trainingsphase und die Produktionsphase (*recall*-Phase) vorgenommen werden. Die Trainingsphase läuft wie folgt ab:

37.1: Training der Segmenterkennung mit einem neuronalen Netz.

Algorithmus:

(a) Einzug des Bildes über einen Videosensor.

(b) Segmentierung des Bildes.

(c) Kompakte Speicherung der Segmente mit Hilfe spezieller Datenstrukturen (z.B. *run-length*-Code).

(d) Für alle zu trainierenden Segmente:

(da) Auswahl eines Segments für das Training.

(db) Aufbau der Trainingsdaten.

(dc) Eintragen der Trainingsdaten in die Trainingsdatei.

(e) Training des neuronalen Netzes.

(f) Abspeichern des trainierten Netzes in einer Datei.

Ende des Algorithmus

Die Produktionsphase ist der Trainingsphase sehr ähnlich:

37.2: Segmenterkennung mit einem neuronalen Netz.

Algorithmus:

(a) Einzug des Bildes über einen Videosensor.

(b) Segmentierung des Bildes.

(c) Kompakte Speicherung der Segmente mit Hilfe spezieller Datenstrukturen (z.B. *run-length*-Code).

(d) Einlesen des trainierten Netzes aus einer Datei.

(e) Für alle zu erkennenden Segmente:

(ea) Auswahl eines Segments.

(eb) Aufbau des Merkmalsvektors .

(ec) Klassifizierung des Merkmalsvektors mit dem trainierten Netz.

Ende des Algorithmus

Der Punkt (db) im Algorithmus zum Training und der Punkt (eb) im Algorithmus zur Produktionsphase sind die wichtigsten Kernstücke des Verfahrens. Sie werden im Folgenden näher erläutert.

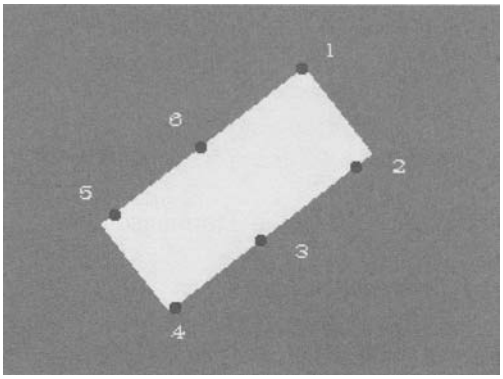


Bild 37.1: Gleichabständige Unterteilung des Randes eines Segments durch sechs Punkte.

37.3 Trainingsdaten und Training

Zu einem Segment, das trainiert werden soll, wird zunächst der Rand des Segments ermittelt. Hier sei darauf hingewiesen, dass in der vorliegenden Beschreibung nur Segmente mit einem zusammenhängenden Rand, also ohne Löcher innerhalb des Segments, betrachtet werden. Auf dem Rand des Segments werden dann $m = n + 1$ Punkte äquidistant verteilt. In der Praxis wird das nicht immer exakt möglich sein, da ja die Länge des Randes (in *Pixel*) meistens nicht durch m teilbar sein wird. Es sollen aber die m Punkte „so gut wie möglich" gleichabständig verteilt sein (Bild 37.1).

Nun werden für jeden der $m = n + 1$ Punkte die n Distanzen zu den restlichen n Punkten berechnet. Die so entstehenden Distanzvektoren werden als Merkmalsvektoren verwendet. Im Beispiel von Bild 37.1 wurden $m = 6 = 5 + 1$ Punkte verteilt. Die sechs Merkmalsvektoren $\vec{g_1}$ bis $\vec{g_6}$ sind hier fünfdimensional:

$$\vec{g_1} = \begin{pmatrix} d_{12} \\ d_{13} \\ d_{14} \\ d_{15} \\ d_{16} \end{pmatrix}, \vec{g_2} = \begin{pmatrix} d_{23} \\ d_{24} \\ d_{25} \\ d_{26} \\ d_{21} \end{pmatrix}, \vec{g_3} = \begin{pmatrix} d_{34} \\ d_{35} \\ d_{26} \\ d_{31} \\ d_{32} \end{pmatrix}, ..., \vec{g_6} = \begin{pmatrix} d_{61} \\ d_{62} \\ d_{63} \\ d_{64} \\ d_{65} \end{pmatrix}. \tag{37.1}$$

Die Distanz des Punktes i mit den Koordinaten (x_i, y_i) zum Punkt j mit den Koordinaten (x_j, y_j) wird mit der euklidischen Abstandsformel berechnet:

$$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}. \tag{37.2}$$

Es liegen somit zu einem Segment m Merkmalsvektoren $\vec{g}$ mit der Dimension n vor. Man sagt auch: Zum Rand des Segments werden m *Ansichten* berechnet. In der späteren

Trainingsphase werden diese m Ansichten des Segments als charakterisierende Merkmale mit einem neuronalen Netz trainiert.

Da die Eingangswerte eines Backpropagation-Netzwerks zwischen 0 und 1 liegen müssen, werden die Distanzen normiert. Dazu wird während der Berechnung eines Merkmalsvektors die minimale (min) und die maximale (max) Komponente bestimmt. Die Normierung der Distanz d_{ij} kann dann nach der Formel

$$d_{ij}^{norm} = \frac{d_{ij} - min}{max - min} \tag{37.3}$$

erfolgen.

In einer Trainingsdatei werden alle Merkmalsvektoren aller Segmente gesammelt, für die das Netz trainiert werden soll. Ein Beispiel einer Trainingsdatei wird in Abschnitt 37.5 gegeben.

Mit diesen Daten wird das Backpropagation-Netz trainiert. Für die Netztopologie, die Anzahl der Neuronen in verdeckten Schichten, die Lernrate und den zu erzielenden minimalen Systemfehler gelten die Aussagen in Kapitel 32.

37.4 Die Produktions- (Recall-) Phase

Der Algorithmus zur *recall*-Phase in Abschnitt 37.2 zeigt, dass hier fast derselbe Weg beschritten wird. Im Gegensatz zur Trainingsphase wird hier zu den m äquidistanten Punkten nur *ein* Merkmalsvektor berechnet, z.B. der Vektor, der dem ersten Punkt zugeordnet ist. Das Segment wird hier also nur durch *eine* Ansicht charakterisiert. Dieser Merkmalsvektor wird dem neuronalen Netz zugeführt, und die Ausgabe des Netzes gibt die zugehörige Klasse an.

Dabei ist zu beachten, dass das Segment in einer anderen Drehlage vorliegen kann. Die äquidistanten Punkte werden dann sicher nicht genau so liegen wie in der Trainingsphase. Der grundlegende Gedanke ist dabei jedoch, dass die Ansicht einer der gelernten Ansichten ähnlich sein wird und das Netz durch den Generalisierungseffekt für die richtige Klasse entscheidet.

Hier wird das Generalisierungsverhalten eines Netzes als wesentliche Eigenschaft der Zuordnung zu den Klassen verwendet. Aus diesem Grund ist es sinnvoll, keine übertrainierten Netze (zu viele Neuronen in den verdeckten Schichten und zu kleiner Systemfehler) zu verwenden (Kapitel 32).

37.5 Ein Beispiel: Unterscheidung von Schrauben

In diesem abschließenden Beispiel zur translations-, rotations- und größeninvarianten Segmenterkennung sollen zwei verschiedene Schraubentypen unterschieden werden (Bild 37.2-a). In der Vorverarbeitung wurde das Bild invertiert und zweimal dilatiert, um die

Konturen der Schrauben etwas ausgeglichener zu gestalten (Bild 37.2-b). Die Segmentierung konnte durch eine einfache Binarisierung erzeugt werden. Zur Bezeichnung wurden die Segmente von 1 bis 6 numeriert (Bild 37.2-c).

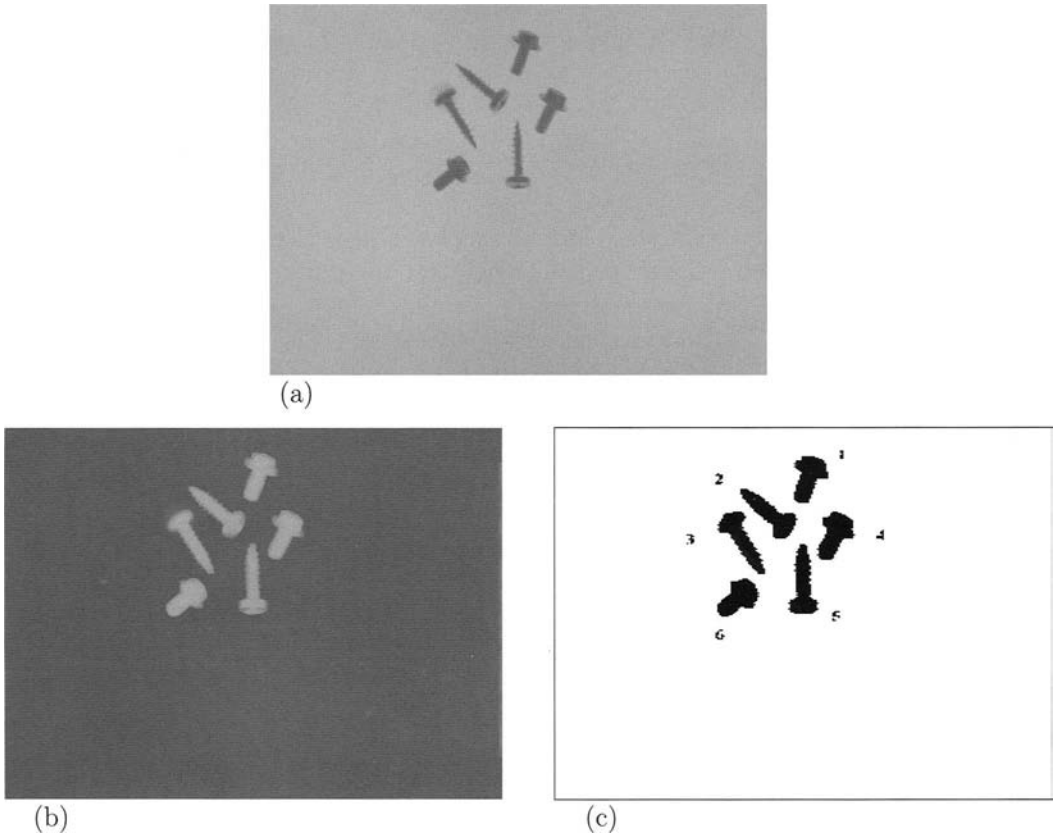


Bild 37.2: (a) Originalbild: Jeweils drei Schrauben eines bestimmten Schraubentyps. (b) Vorverarbeitung: Invertierung und einmalige Dilatation, um die Konturen etwas „runder" zu gestalten. (c) Segmentierung durch Binarisierung. Mit den Segmenten vier und fünf wurde das neuronale Netz für die zwei Klassen trainiert.

Im Training wurden nur die Segmente vier und fünf verwendet. Es wurden auf der Kontur der Segmente elf Punkte gleichmäßig verteilt, so dass die Merkmalsvektoren zehndimensional sind. Das neuronale Netz hatte bei diesem Test, abhängig von den Ein-/Ausgabedaten, folgende Struktur: Zehn Eingabeneuronen, zwei Ausgabeneuronen und eine verdeckte Schicht mit zehn Neuronen. Die Trainingsdatei ist in folgender Tabelle wiedergegeben:

```
train
inpn 10
outpn 2
data
0.022891 0.297219 0.501564 0.499727 0.748440
1.000000 0.865390 0.583158 0.297538 0.053051    0.95 0.05
0.022891 0.098389 0.349881 0.417831 0.701165
0.980041 0.894565 0.619416 0.340513 0.212547    0.95 0.05
0.297219 0.098389 0.053051 0.195373 0.487907
0.780699 0.758973 0.518783 0.303894 0.354507    0.95 0.05
0.501564 0.349881 0.053051 0.000000 0.257634
0.547774 0.577370 0.389987 0.279645 0.461229    0.95 0.05
0.499727 0.417831 0.195373 0.000000 0.074066
0.365914 0.362515 0.172850 0.123398 0.376825    0.95 0.05
0.748440 0.701165 0.487907 0.257634 0.074066
0.073503 0.137564 0.108117 0.269947 0.568188    0.95 0.05
1.000000 0.980041 0.780699 0.547774 0.365914
0.073503 0.038732 0.234922 0.490473 0.786469    0.95 0.05
0.865390 0.894565 0.758973 0.577370 0.362515
0.137564 0.038732 0.062584 0.349881 0.621778    0.95 0.05
0.583158 0.619416 0.518783 0.389987 0.172850
0.108117 0.234922 0.062584 0.067808 0.344337    0.95 0.05
0.297538 0.340513 0.303894 0.279645 0.123398
0.269947 0.490473 0.349881 0.067808 0.080194    0.95 0.05
0.053051 0.212547 0.354507 0.461229 0.376825
0.568188 0.786469 0.621778 0.344337 0.080194    0.95 0.05
0.012551 0.241898 0.471953 0.709719 0.935187
1.000000 0.805930 0.595665 0.345771 0.095366    0.05 0.95
0.012551 0.070258 0.307902 0.546835 0.767430
0.850464 0.672995 0.455651 0.215990 0.000000    0.05 0.95
0.241898 0.070258 0.069366 0.307752 0.527592
0.617138 0.455651 0.236322 0.035707 0.033615    0.05 0.95
0.471953 0.307902 0.069366 0.069366 0.293825
0.380524 0.241032 0.033615 0.026119 0.227095    0.05 0.95
0.709719 0.546835 0.307752 0.069366 0.060871
0.156343 0.098571 0.029008 0.225476 0.458492    0.05 0.95
0.935187 0.767430 0.527592 0.293825 0.060871
0.060871 0.185213 0.238597 0.455651 0.687740    0.05 0.95
1.000000 0.850464 0.617138 0.380524 0.156343
0.060871 0.065763 0.237723 0.487742 0.737700    0.05 0.95
...
0.805930 0.672995 0.455651 0.241032 0.098571
0.687740 0.737700 0.540947 0.331763 0.081286    0.05 0.95
```

In der Trainingsdatei ist gespeichert, dass das Backpropagation-Netz in diesem Beispiel zehn Eingabeneuronen und zwei Ausgabeneuronen besitzt, entsprechend den zehndimensionalen Merkmalsvektoren und den beiden Ausgabeklassen (Segmentnummer 1 und 2). Ab dem Kennzeichen `data` sind die Trainingspaare abgespeichert. Nach dem ersten Merkmalsvektor des ersten Segments folgt die Sollausgabe 0.95 und 0.5. Aus numerischen Gründen wird hier nicht 1 und 0 verwendet. Dann folgen die weiteren Merkmalsvektoren des Segments, jedes Mal mit der Sollausgabe 0.95 und 0.05. Die Trainingsdaten eines zweiten Segments schließen sich an.

Im Training waren 224 Durchläufe notwendig, um einen Systemfehler von 0.05 zu erreichen. Das Ergebnis ist in folgender Tabelle zusammengefasst:

Segmentnr.	Ausgabeneuron 1	Ausgabeneuron 2	Klasse
1	0.981386	0.019592	1
2	0.101804	0.892246	2
3	0.050654	0.945478	2
4	0.937410	0.064300	1
5	0.057105	0.938721	2
6	0.937877	0.063785	1

Man sieht, dass alle Segmente richtig erkannt wurden. Wie schon im einleitenden Abschnitt zu dieser Thematik erwähnt, ist die Trainingsphase einfacher als beim Strahlenverfahren, da hier ein Segment nur einmal mit seinen verschiedenen Ansichten trainiert werden muss.

Kapitel 38

Kalman-Filter

38.1 Grundidee

Gegeben sei ein physikalisches System: z.B. eine Kugel, die durch die Luft fliegt und von Luftwirbeln ein wenig aus der Normal-Bahn abgelenkt wird (Bild 38.1). Die Bewegung dieses Systems kann durch das Newton'sche Gesetz ($F_a = ma$) beschrieben werden ($F_a = F_g + F_r$, wobei F_g die Gewichtskraft ist und F_r die Reibungskraft). Die Kräfte F_r, die durch die Luftwirbel auf die Kugel wirken, werden einfach als eine zufällige, normalverteilte Komponente betrachtet. Weiterhin gegeben sei eine physikalische Messvorrichtung: z.B. eine feststehende Kamera, die in bestimmten zeitlichen Abständen Bilder der Situation aufnimmt. Messbar sind dadurch die relative Position und die relative Größe der Kugel im Bild. Die Messgenauigkeit der Kamera wird nicht nur durch die Auflösung limitiert, sondern sei zusätzlichen Schwankungen durch die Luftwirbel oder anderen Störungen (z.B. Verdeckung) ausgesetzt.

Die Aufgabe besteht nun darin, den besten Schätzwert für die Position und die Geschwindigkeit der Kugel zum Zeitpunkt k unter Verwendung aller bis zu diesem Zeitpunkt angefallenen Messungen zu finden.

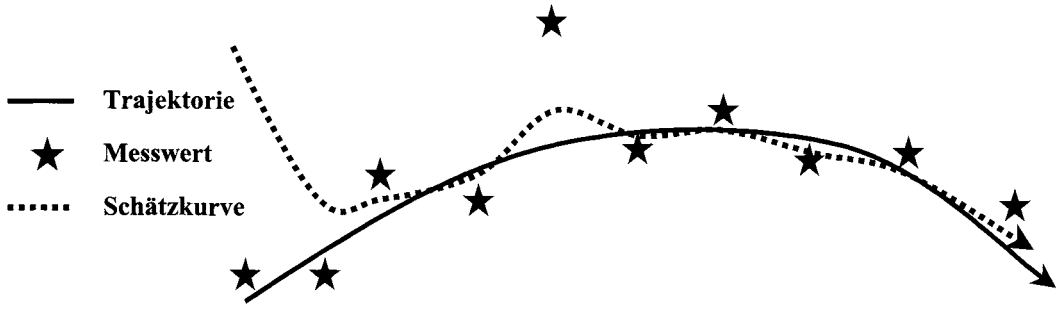


Bild 38.1: Die Grundidee des Kalman-Filters in der Bildverarbeitung: Prädiktion der Objekttrajektorie mit Hilfe von Bewegungsgleichungen und Korrektur der Trajektorie mit jeder neuen Messung. Die Messgenauigkeit bestimmt den Einfluss der jeweiligen Messung.

In der Lösungsidee werden alle verfügbaren Informationen sofort verarbeitet: Aus der bekannten Größe der Kugel und den Abbildungseigenschaften der Kamera kann die Entfernung Kugel – Kamera und somit auch die Position der Kugel (x, y, z) geschätzt werden. Aus zwei aufeinanderfolgenden Bildern kann auch die Geschwindigkeit (v_x, v_y, v_z) der Kugel geschätzt werden. Mit Hilfe der Systemgleichung (hier die Newton'sche Bewegungsgleichung) kann der Ort und die Geschwindigkeit der Kugel für zukünftige Zeitpunkte prädiziert werden. Die Abweichung zwischen prädiziertem und gemessenem Systemzustand (hier Ort und Geschwindigkeit) wird zur Verbesserung der Schätzung des Systemzustands verwendet. Messungen mit großer Genauigkeit sollen stark in die Berechnung einfließen, Messungen mit geringer Genauigkeit sollen nur wenig in die Berechnung einfließen. Das Kalman-Filter ist gerade so ausgelegt, dass es unter Berücksichtigung des Mess- und Systemrauschens (hier die Luftwirbel) aus den vorhandenen Messwerten den optimalen Schätzwert für den Systemzustand liefert.

38.2 Anwendungen

Seit der Originalarbeit von Kalman im Jahr 1960 [Kalm60] wurden zahlreiche Anwendungen für das gleichnamige stochastische Filter entwickelt. In der Regelungstechnik gehört das Kalman-Filter bereits seit längerem zu den Standardmethoden der Optimalen Regelung mit Zustandsbeobachter [Ludy95]. Eine wesentliche Rolle spielt das Kalman-Filter in modernen Navigationssystemen, in denen Messdaten verschiedener Navigations-Sensoren (z.B. Inertialsensor und GPS (Global Positioning System)) zur einer optimalen Positionsschätzung fusioniert werden [Jeke01]. Ganz allgemein ist das Kalman-Filter eines der am häufigsten eingesetzten Verfahren, mit denen verrauschte Messdaten verschiedener Sensoren zusammengeführt werden. Ein weiteres Einsatzgebiet des Kalman-Filters ist die Vorhersage ökonomischer Indikatoren. In diesem Fall hat man es mit sehr komplexen Systemen zu tun, bei denen systematische Zusammenhänge einzelner Größen von starken statistischen Einflüssen überlagert werden. Mit dem Kalman-Filter kann man die systematischen Zusammenhänge herausfiltern, falls sie richtig modelliert wurden.

Im Folgenden wird etwas detaillierter auf Anwendungen des Kalman-Filters in der Bildverarbeitung und Computergrafik eingegangen.

38.2.1 Tracking

Der häufigste Anwendungsfall des Kalman-Filters in der Bildverarbeitung ist die Objektverfolgung, auch *Tracking* genannt. In diesem Fall soll die Position und/oder die Geschwindigkeit von Objekten zu jedem Zeitpunkt optimal geschätzt werden. Als Eingabewerte liegen in der Regel Kamerabilder vor[1]. Mit Hilfe von Bildverarbeitungsverfahren wird eine Objekterkennung durchgeführt, so dass die Lage des Objektabbildes auf dem Kamerasensor bekannt ist. Aus der prädizierten Position und Lage des Objekts im 3-dimensionalen

[1] In speziellen Anwendungen können dies auch Positionsdaten anderer Sensoren sein, wie z.B. von einem Radar, Lidar oder Sonar.

Raum kann mit den vorab vermessenen Abbildungseigenschaften der Kamera die Lage des auf den Sensor projizierten Objekts vorhergesagt werden. Die Abweichung zwischen prädizierter und gemessener Lage des Objekts wird zur Verbesserung der Positions- und Geschwindigkeits-Schätzung des Objekts verwendet. Für eine erfolgreiche Filterung ist es erforderlich, sowohl die Genauigkeit der Positionsmessung als auch die Objekt- und Eigenbewegung zu berücksichtigen.

Für das Tracking gibt es eine große Bandbreite verschiedener Anwendungen, von denen hier nur einige Beispiele stellvertretend genannt werden. Ein vielversprechendes Gebiet sind Mensch-Maschine-Schnittstellen, wie z.B. die Gestenerkennung (z.B. Verfolgung einer bewegten Hand, Bild 38.3), die Verfolgung von Lippenbewegungen zur Spracherkennung, die Verfolgung von Augen-, Kopf- oder Körperbewegungen zur Steuerung von Automaten oder Virtual Reality Umgebungen [Dorf03]. In der Automobiltechnik werden neue Fahrerassistenzsysteme auf der Basis bildgebender Sensoren entwickelt, wie z.B. automatische Geschwindigkeits- und Abstandskontrolle, Warnung vor Hindernissen oder zu hoher Kurvengeschwindigkeit, automatische Fahrspurverfolgung bis hin zum automatischen Fahrer. In der Robotik geht es sowohl um die bildbasierte Navigation und Hinderniserkennung zur autonomen Bewegung, als auch um das Greifen oder Fangen von bewegten Objekten. Im Verteidigungsbereich wird das Kalman-Filter seit Langem zur automatischen Zielverfolgung eingesetzt.

Die prinzipiellen Ablaufschritte eines Kalman Trackers bei der Objektverfolgung sind:

- Aufschalten des Kalman-Filters (*Alignment*):
  Beim *Alignment* muss das gesamte Bild[2] abgescannt werden, da noch keine verlässliche Schätzung über die Lage des Objekts im Bild vorhanden ist.

  - Bildaufnahme:
    Mit dem Kamerasensor wird ein Bild eingezogen und digitalisiert.
  - Bildvorverarbeitung:
    Geometrische Entzerrung (z.B. wegen Kissenverzerrung der Optik), Kontrastoptimierung (z.B. Histogrammlinearisierung), Rauschunterdrückung (z.B. Median-Operator).
  - Merkmalsextraktion:
    Häufig sind dies Kantendetektion (z.B. Gradienten- oder Sobel-Operatoren) und Linienverfolgung (z.B. Bestimmung der Gradienten-Kammlinie). Ergebnis: Position und Orientierung der Kanten, sowie Geradheit, Ausgefranstheit von Linien.
  - Projektion:
    Die Kanten des gesuchten 3D-Objekts werden auf die 2D-Sensorbildebene projiziert.

[2] Bei einer beweglichen Kamera wird meist der gesamte Schwenkbereich der Kamera nach dem gesuchten Objekt abgescannt.

- Matching-Verfahren:
  Korrelation des gemessenen Linienmusters mit dem erwarteten Muster. Das Ergebnis bei einer ausreichenden Korrelation ist eine Startschätzung des Systemzustands (z.B. Position, Lage und Geschwindigkeit des Objekts relativ zur Kamera).

Nach dem *Alignment* laufen abwechselnd die beiden folgenden Schritte ab, bis alle Bilder verwertet sind:

- Prädiktionsschritt:
  Vorhersage der relativen Position und Lage des Objekts zum nächsten Abtastzeitpunkt auf der Basis der zugrunde liegenden Systemdynamik (Position, Lage, Geschwindigkeit, Beschleunigung der eigenen Kamera und des Objekts, Kardanwinkel der Kamera usw.)

- Korrekturschritt:
  Korrektur der Prädiktion mit der Messung. Dabei werden ähnliche Schritte durchgeführt wie beim *Alignment*, d.h. Bildaufnahme, Bildvorarbeitung, Merkmalsextraktion, Objekterkennung mit eingeschränktem Suchbereich (je nach Qualität der Prädiktion): Dabei wird die Messung (Detektion des Objekts) dadurch erheblich erleichert, dass vom vorhergehenden Prädiktionsschritt ein Schätzwert für die Objektposition vorhanden ist. Deshalb genügt es, das Objekt in einem kleinen Bildausschnitt zu suchen - man muss nicht mehr das gesamte Bild bzw. den gesamten Schwenkbereich der Kamera absuchen (wie beim *Alignment*). Dieser Bildausschnitt wird „track gate" genannt, und seine Größe wird bestimmt von der Genauigkeit der Prädiktion. Die Abweichung zwischen geschätzter und gemessener Objektposition auf der Sensorebene wird zur Verbesserung der Positions- und Geschwindigkeits-Schätzung für das Objekt benutzt.

Qualitätsmaße für die detektierten Kanten und Linien, wie z.B. der Kontrast der Kanten oder die Ausgefranstheit von Linien, werden als Maß für das Messrauschen benutzt und bestimmen somit, wie stark die Messung in die Schätzung des Systemzustands eingeht. Ist die Kantendetektion z.B. aufgrund eines sehr niedrigen Bildkontrasts sehr unsicher, geht die Messung kaum in die Zustandsschätzung ein. Auf diese Weise wird eine evtl. fehlerhafte Objekterkennung praktisch nicht berücksichtigt. Sind die Kanten dagegen klar zu erkennen, gehen sie mit höherem Gewicht in die Schätzung ein. In diesem Sinne werden die guten Messergebnisse herausgefiltert.

Die Wirkungsweise eines Kalman-Filters, bei dem die oben geschilderten Ablaufschritte durchgeführt werden, ist in Bild 38.2 dargestellt. In diesem Beispiel werden charakteristische Kanten eines Helicopterlandeplatzes verfolgt, so dass ein Helicopter mit Kamera automatisch an der vorgesehen Stelle landen kann. Die blau hervorgehobenen Kanten sind gemessen, d.h. durch Bildverarbeitungsalgorithmen aus dem aktuellen Bild extrahiert. Die rot eingezeichneten Kanten sind vom Kalman-Filter aus dem vorhergehenden Bild prädiziert. Wie man aus den Bildern sehen kann, liegen die prädizierten Kanten meistens relativ nahe an den gemessenen Kanten.

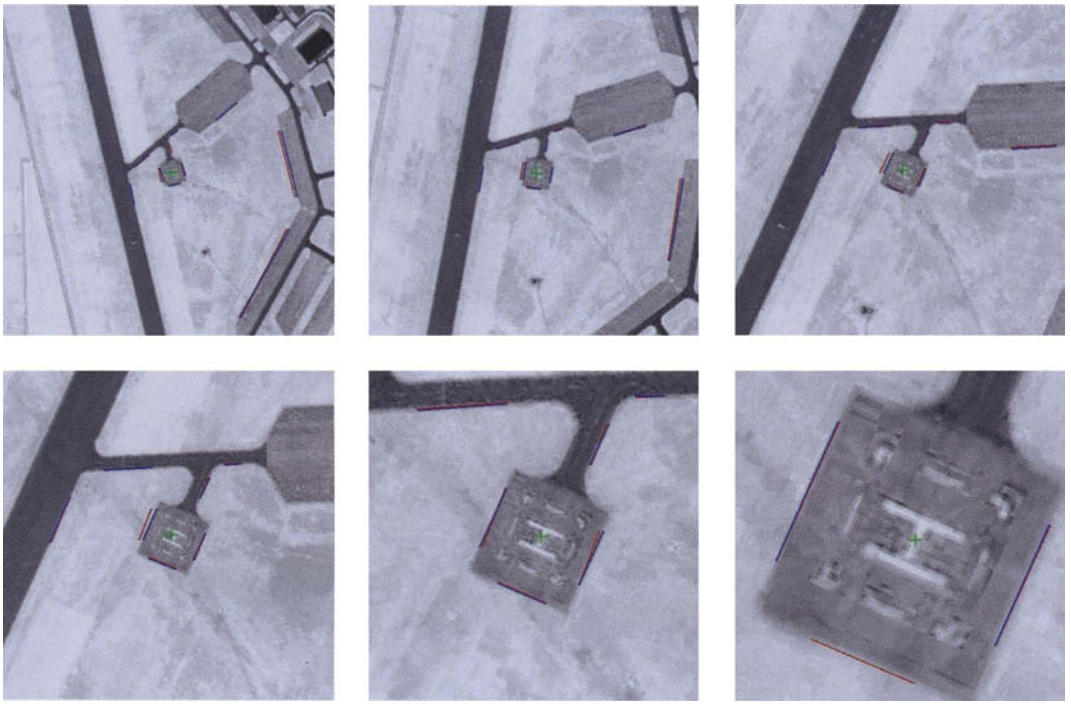


Bild 38.2: Kanten-basierter Kalman Tracker zur Detektion eines Hubschrauberlandeplatzes. In diesem Fall ruht das Objekt (der Landeplatz) und die Kamera bewegt sich. Die blauen Kanten sind gemessen, d.h. durch Bildverarbeitungsalgorithmen detektiert. Die roten Kanten sind vom Kalman-Filter prädiziert. Quelle: Lackner, EADS-LFK Lenkflugkörpersysteme GmbH.

Bei dem oben dargestellten Prinzip eines Kalman Trackers wird der Vergleich zwischen geschätzter und gemessener Objektposition auf der Basis von Kanten und Linien durchgeführt. Dies ist der häufigste Fall. Eine Alternative dazu sind segment-basierte Methoden, bei denen eine Szene nach bestimmten Kriterien in verschiedene Segmente unterteilt wird. Solche Kriterien können z.B. ähnliche Bewegungsvektoren oder Texturmerkmale sein. Das Kalman-Filter dient in diesem Fall dazu, die Verschiebung und evtl. die Verzerrung des Segments im nächsten Bild vorherzusagen. Das prädizierte Segment wird nun verglichen mit dem „gemessenen" Segment, das sich aus der Einzelbild-Segmentierung des aktuellen Bildes ergibt. Die sich nicht überlappenden Teile des Segments sind ein Maß für die Abweichung der Prädiktion von der Messung und dienen zur Verbesserung der Schätzung der Objektbewegung. Ein Beispiel für einen segment-basierten Kalman Tracker, der in diesem Fall zur Verfolgung einer Handbewegung dient, ist in Bild 38.3 gegeben [Huan02].

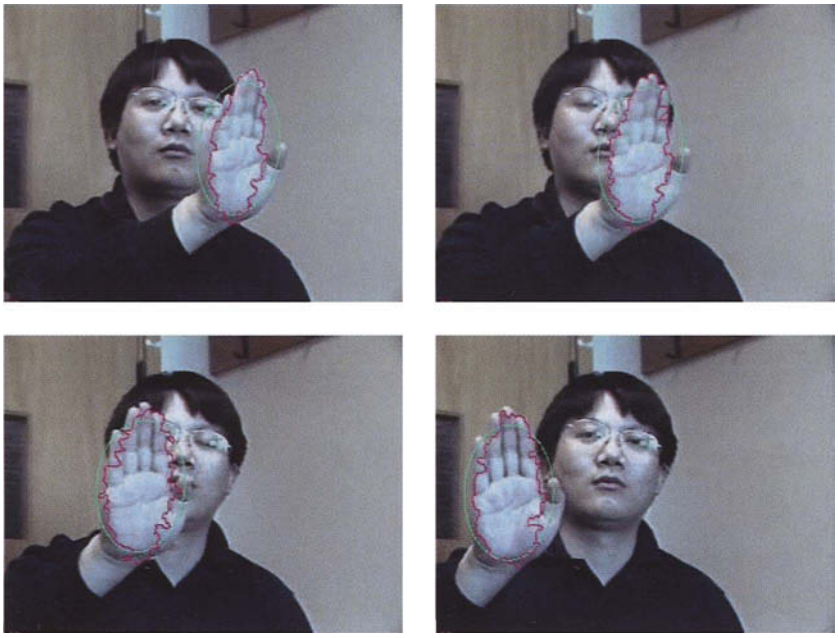


Bild 38.3: Segment-basierter Kalman Tracker zur Verfolgung von Handbewegungen. In diesem Fall bewegt sich das Objekt (die Hand) und die Kamera ruht. Die grüne Ellipse wird anfangs manuell um das zu verfolgende Objekt (die Hand) gelegt. Das *Alignment* wird in diesem Fall also manuell durchgeführt. Die rote Kurve stellt die detektierte Kontur der Hand dar. Quelle: Yu Huang.

In den meisten Tracking-Anwendungen geht es um die Verfolgung von bewegten Objekten. „Bewegung" bezieht sich in diesem Fall immer auf eine Relativbewegung zwischen Kamera und Objekt, d.h. es ist unerheblich, welches Koordinatensystem man wählt. Im Koordinatensystem der Kamera bewegt sich das Objekt (wie in Bild 38.3), im Koordinatensystem des Objekts bewegt sich die Kamera (wie in Bild 38.2) und in einem ungeschickt gewählten Koordinatensystem bewegen sich sowohl die Kamera als auch das Objekt. An dieser Stelle sei aber darauf hingewiesen, dass man auch andere Objekteigenschaften „verfolgen" und „vorhersagen" kann, wie z.B. die Orientierung (d.h. eine Rotationsbewegung), die Verformung oder Oberflächeneigenschaften (Textur, Farbe etc.) von Objekten.

Außerdem sollte an den bisher gezeigten Anwendungsbeispielen klar geworden sein, dass das Kalman-Filter kein Bildverarbeitungsfilter im herkömmlichen Sinn ist, wie z.B. ein Hoch- oder Tiefpassfilter, das direkt auf den pixel-basierten Grauwerten eines Bildes agiert, sondern ein Meta-Filter, das auf einer höheren Abstraktionsebene arbeitet, wie z.B. Kanten, Linien oder Segmenten. Das Kalman-Filter setzt also als Vorverarbeitungsschritte immer die klassischen Methoden der Bildverarbeitung und Mustererkennung voraus.

38.2.2 3D-Rekonstruktion aus Bildfolgen

Die Rekonstruktion der 3-dimensionalen Struktur einer Szene ausgehend von 2-dimensionalen Bildfolgen ist eines der zentralen Probleme der *Computer Vision.* Die 2-dimensionalen Bildfolgen können dabei sowohl von einer bewegten Videokamera als auch von einer Stereokamera (stehend oder bewegt) aufgezeichnet worden sein. Das wesentliche Problem dabei ist, korrespondierende Bildpunkte in aufeinander folgenden Bildern zu finden. Dieses sogenannte Korrespondenzproblem ist vor allem deshalb schwierig zu lösen, weil durch Bildrauschen und Verdeckungen eine Zuordnung von Bildpunkten zwischen zwei Bildern manchmal unmöglich ist, oder weil wegen Mehrdeutigkeiten falsche Zuordnungen erfolgen. Die Lösungansätze für das Korrespondenzproblem kann man einteilen in pixel-basierte Verfahren, bei denen Verschiebungsvektorfelder mit Hilfe des Blockmatchings (Abschnitt 26.5) oder differentieller Methoden (Abschnitt 26.4) bestimmt werden, sowie in merkmal-basierte Verfahren, bei denen versucht wird, Merkmale wie Kanten oder Linienzüge einander zuzuordnen. Für die weitere Verarbeitung dieser Informationen in einem Kalman-Filter wird davon ausgegangen, dass das Korrespondenzproblem gelöst ist. Fehlerhafte Korrespondenzen werden als Messrauschen aufgefasst.

Die 3D-Rekonstruktion aus Bildfolgen ist eine Erweiterung des Trackings, bei der von einer statischen Szene ausgegangen wird, durch die sich eine Kamera bewegt. Während beim Tracking nur die 3-dimensionale Bewegung der Kamera geschätzt wird, soll bei der 3D-Rekonstruktion sowohl die Bewegung der Kamera, als auch die 3-dimensionale Struktur der Szene optimal geschätzt werden. Dazu wird der Zustandsvektor, der beim Tracking nur die Bewegungsparameter enthält, um eine bestimmte Anzahl von Parametern erweitert, die die räumliche Tiefe der Merkmale beschreiben. Als Eingabewerte liegen wieder Bilder der Kamera aus verschiedenen Aufnahmepositionen vor. Die Aufgabe der Bildvorverarbeitung ist es, die Positionen einer bestimmten Anzahl von Merkmalen in jedem Bild der Sequenz zu berechnen. Aufgrund des oben geschilderten Korrespondenzproblems ist dies eine nichttriviale Aufgabe, deren Lösung in der Regel fehlerbehaftet ist. Aus der Position eines Merkmals in den Kamerabildern kann mit Hilfe der bekannten Projektionseigenschaften der Kamera auf die Position des Merkmals im 3-dimensionalen Raum geschlossen werden. Im Prinzip kann die räumliche Tiefe von Merkmalen bereits aus zwei Bildern (d.h. einem Stereobildpaar) durch Triangulation berechnet werden. Wegen der fehlerbehafteten Korrespondenzen ist es jedoch sinnvoll, Bilder aus weiteren Aufnahmepositionen zur Verbesserung der 3D-Struktur-Schätzung im Rahmen des Kalman-Filters heranzuziehen. Bei einer ausreichenden Anzahl an Bildern aus geeigneten Aufnahmepositionen kann bei nicht zu stark verwinkelten Szenen sowohl die 3D-Struktur der Szene als auch die Bewegung der Kamera rekonstruiert werden ([Ying04], [Alon00], [Azar95]). Falls bestimmte Merkmale nicht aus mehreren unterschiedlichen Kamerapositionen einzusehen sind, ist die räumliche Tiefe dieses Merkmals ein nicht beobachtbarer Zustand des Systems.

Das Kalman-Filter bietet sich in der 3D-Rekonstruktion von Szenen natürlich auch besonders gut an, wenn Daten von verschiedenen Sensoren gleichzeitig gemessen werden. Die 3D-Rekonstruktion ausschließlich aus Videobildsequenzen ist zwar, wie eben geschildert, durchaus möglich, aber verhältnismäßig schwierig und fehleranfällig. Wenn es die

Anwendung erlaubt, werden deshalb neben einer Videokamera häufig zusätzliche Sensoren, wie Laserentfernungsmesser, Radar oder Ultraschallentfernungsmesser eingesetzt, um direkt eine Tiefenkarte der Szene erzeugen zu können. Zur Fusion der verschiedenen Sensordaten kann ein Kalman-Filter eingesetzt werden, in dem die Beiträge der einzelnen Sensoren – gewichtet mit ihrer jeweiligen Messgenauigkeit – in eine optimale Schätzung der 3-dimensionalen Geometrie der Szene eingehen [Vanp93].

38.2.3 Bilddatencodierung

Die Grundidee moderner Standards zur Codierung von Bewegtbildern, wie MPEG-1/2 oder MPEG-4, besteht darin, die Ähnlichkeit zweier aufeinander folgender Bilder zur Datenreduktion auszunutzen. Häufig unterscheidet sich ein Bild zum Zeitpunkt t vom vorhergehenden Bild zum Zeitpunkt $t-1$ nur durch geringfügige Objekt- oder Kamerabewegungen. Aus den beiden Bildern lässt sich durch Blockmatching ein Bewegungsvektorfeld berechnen (Abschnitt 26.5), das angibt, wie sich die $8 \cdot 8$ Pixel großen Blöcke des Bildes von $t-1$ bis t bewegen. Folglich kann man aus dem Bild zum Zeitpunkt $t-1$ mit Hilfe der Bewegungsvektoren das Bild zum Zeitpunkt t bis auf gewisse Differenzen prädizieren. Dieses Bild bezeichnet man als *bewegungskompensiert*. Zur Codierung des Bildes t benötigt man also nur die Bewegungsvektoren, sowie das sehr gut komprimierbare Differenzbild zwischen dem bewegungskompensierten Bild und dem Originalbild. Auf diese Weise kann man häufig eine Datenkompression um den Faktor 10 oder besser erreichen. Zur Rekonstruktion des Originalbildes rechnet man zunächst aus dem Bild $t-1$ und den Bewegungsvektoren das bewegungskompensierte Bild aus. Durch die anschließende Addition des Differenzbildes erhält man wieder das Originalbild t (Bild 38.4).

Die erreichbare Kompressionsrate hängt bei dem geschilderten Verfahren von der Qualität der Bewegungsschätzung ab. Da das Kalman-Filter, wie in den vorigen Abschnitten dargestellt, hervorragend für die Bewegungsschätzung geeignet ist, können damit bessere Ergebnisse erzielt werden, als mit dem Blockmatching. Es gibt zahlreiche Vorschläge für den Einsatz des Kalman-Filters in der Bilddatencodierung, von denen einer stellvertretend kurz vorgestellt wird [Calv00]. Grundannahme bei diesem Verfahren, das sich insbesondere für Videokonferenzen sehr gut eignet, ist eine ruhende Kamera bzw. ein ruhender Hintergrund, vor dem sich einige starre Objekte (z.B. Kopf, Arm, Oberkörper) bewegen. Die Objektbewegungen besitzen sechs Freiheitsgrade, drei für die Translation und drei für die Rotation. Zusätzlich können noch einige spezielle Merkmale (wie z.B. Augen und Mund) als verformbare Objekte mit Hilfe von Formparametern beschrieben werden. Ein erweitertes Kalman-Filter (EKF, Abschnitt 38.3.6) wird zur Schätzung der Bewegungen und der Formparameter eingesetzt. Dieses Verfahren lässt sich auch sehr gut zur Animation von hybriden synthetisch/natürlichen Objekten einsetzen, die durch ein 3-dimensionales Polygonnetz beschrieben werden, auf das eine Foto-Textur gemappt wird (Kapitel 13). Durch das Kalman-Filter wird in diesem Fall die Bewegung der Eckpunkte des Polygonnetzes geschätzt. Für das erste Bild muss die Foto-Textur und das Polygonnetz übertragen werden, für alle weiteren Bilder nur noch die Bewegung der Eckpunkte des Polygonnetzes.

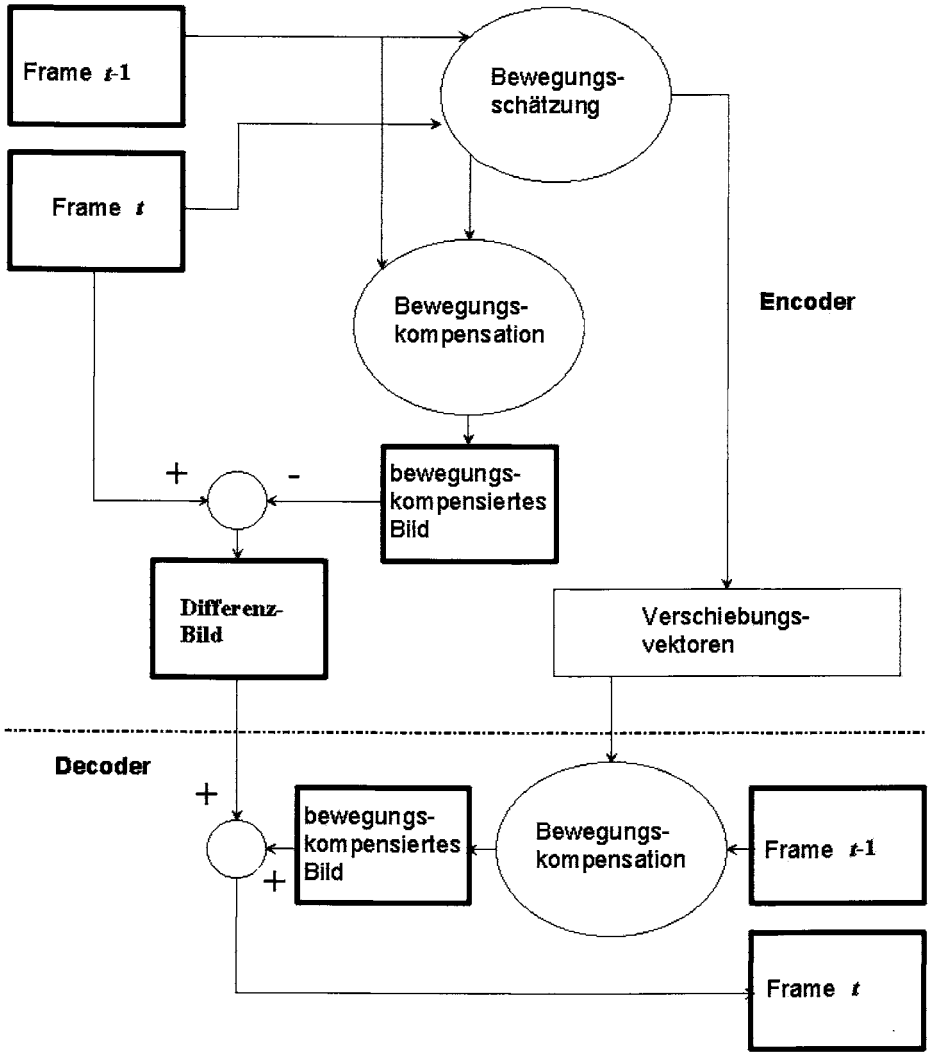


Bild 38.4: Das Blockschaltbild für einen Encoder/Decoder mit Bewegungsschätzung und Bewegungskompensation.

Auf der Empfängerseite werden die Bilder mit Hilfe einer schnellen Grafikhardware wieder erzeugt. Mit solchen hybriden Codierverfahren, wie sie seit dem Codierstandard MPEG-4 möglich sind, lassen sich sehr hohe Kompressionsfaktoren erzielen.

38.3 Theorie des diskreten Kalman-Filters

38.3.1 Das System

Das Kalman-Filter ist eine Lösung des Problems, einen optimalen Schätzwert für den Zustand $\mathbf{x}$ eines Systems zu liefern, welches durch folgende zeitdiskrete lineare stochastische Differenzengleichung beschreibbar ist:

$$\mathbf{x}_{k+1} = \mathbf{A}_k \cdot \mathbf{x}_k + \mathbf{B}_k \cdot \mathbf{u}_k + \mathbf{w}_k \tag{38.1}$$

Dabei ist $\mathbf{x}_k$ der n-dimensionale Zustands-Vektor des Systems und $\mathbf{u}_k$ der l-dimensionale Steuer-Vektor zum Zeitpunkt k. Die $(n \cdot n)$-Matrix $\mathbf{A}_k$ in (38.1), auch Systemmatrix genannt, bildet den Systemzustand vom vorherigen Zeitschritt k auf den aktuellen Zeitschritt $k+1$ ab, und zwar ohne eine externe Kraft $\mathbf{u}_k$, sowie ohne Systemrauschen $\mathbf{w}_k$. Die $(n \cdot l)$-Matrix $\mathbf{B}_k$, auch Eingabematrix genannt, bildet den optionalen Steuer-Vektor $\mathbf{u}_k$ auf den Systemzustand ab. Die n-dimensionale, $N(0, \mathbf{Q}_k)$-verteilte Zufallsvariable $\mathbf{w}_k$ repräsentiert das Systemrauschen ($N(0, \mathbf{Q}_k)$ steht für Normalverteilung mit Mittelwert 0 und Varianz $\mathbf{Q}_k$).

38.3.2 Die Messung

Beobachtbar seien nur die Messungen $\mathbf{y_k}$, die über folgende Messgleichung aus dem Zustand des Systems hervorgehen:

$$\mathbf{y}_k = \mathbf{H}_k \cdot \mathbf{x}_k + \mathbf{v}_k \tag{38.2}$$

$\mathbf{y}_k$ ist ein m-dimensionaler Mess-Vektor zum Zeitpunkt k. Die $(m \cdot n)$-Matrix $\mathbf{H}_k$ in (38.2), auch Messmatrix genannt, bildet den Systemzustand beim Zeitschritt k auf den Messwert $\mathbf{y}_k$ ab, und zwar ohne Messrauschen $\mathbf{v}_k$. Die m-dimensionale, $N(0, \mathbf{R}_k)$-verteilte Zufallsvariable $\mathbf{v}_k$ repräsentiert das Messrauschen ($N(0, \mathbf{R}_k)$ steht für Normalverteilung mit Mittelwert 0 und Varianz $\mathbf{R}_k$).
Es gelten folgende Annahmen:

- Systemrauschen $\mathbf{w}_k$ und Messrauschen $\mathbf{v}_k$ seien unabhängig voneinander.
- Es wird angenommen, dass die Größen $\mathbf{w}_k$, $\mathbf{v}_k$, $\mathbf{A}_k$, $\mathbf{B}_k$, $\mathbf{H}_k$ konstant sind. In der Praxis kann sich das Rauschen bzw. die System-, Eingabe- und Messmatrix mit jedem Zeitschritt ändern.

38.3.3 Die Schätzfehler-Gleichungen

Man definiert $\hat{\mathbf{x}}_k^-$ als den a priori Schätzwert für den Systemzustand zum Zeitschritt k, wobei alle Messungen bis zum Zeitschritt $k-1$ einfließen können. „A priori" bedeutet in diesem Zusammenhang „im Vorhinein", d.h. $\hat{\mathbf{x}}_k^-$ ist der prädizierte Schätzwert für den

Systemzustand. Man definiert $\hat{\mathbf{x}}_k^+$ als den a posteriori Schätzwert für den Systemzustand zum Zeitschritt k, wobei die aktuelle Messung zum Zeitschritt k eingeflossen ist. „A posteriori" bedeutet in diesem Zusammenhang „im Nachhinein", d.h. $\hat{\mathbf{x}}_k^+$ ist der mit der neuen Messung korrigierte Schätzwert für den Systemzustand.

Jetzt kann man die a priori und a posteriori Schätzfehler definieren als:

$$\mathbf{e}_k^- = \mathbf{x}_k - \hat{\mathbf{x}}_k^- \tag{38.3}$$

$$\mathbf{e}_k^+ = \mathbf{x}_k - \hat{\mathbf{x}}_k^+ \tag{38.4}$$

sowie die a priori und a posteriori Schätzfehlerkovarianzen als:

$$\mathbf{P}_k^- = E\left[\mathbf{e}_k^- \cdot \mathbf{e}_k^{-T}\right] \tag{38.5}$$

$$\mathbf{P}_k^+ = E\left[\mathbf{e}_k^+ \cdot \mathbf{e}_k^{+T}\right] \tag{38.6}$$

Hinweis: „E" steht für Erwartungswert. Bei einer diskreten Verteilung α_i bedeutet das: $E(\mathbf{x}) = \sum \alpha_i \cdot \mathbf{x}_i$. Bei einer Gleichverteilung entspricht der Erwartungswert dem Mittelwert.

38.3.4 Optimales Schätzfilter von Kalman

Zunächst muss eine Formel zur Berechnung des a posteriori Schätzwerts $\hat{\mathbf{x}}_k^+$ des Systemzustands hergeleitet werden, und zwar auf Basis des „a priori" Schätzwerts $\hat{\mathbf{x}}_k^-$ und der gewichteten Differenz zwischen dem aktuellen Messwert $\mathbf{y}_k$ und dem vorhergesagten Messwert $\mathbf{H}_k \cdot \hat{\mathbf{x}}_k^-$:

$$\hat{\mathbf{x}}_k^+ = \hat{\mathbf{x}}_k^- + \mathbf{K}_k\left(\mathbf{y}_k - \mathbf{H}_k \cdot \hat{\mathbf{x}}_k^-\right) \tag{38.7}$$

Die Differenz $\left(\mathbf{y}_k - \mathbf{H}_k \cdot \hat{\mathbf{x}}_k^-\right)$ ist ein Maß für die Güte der Vorhersage des Systemzustands. Die Differenz wird null, falls der vorhergesagte Messwert exakt mit dem aktuellen Messwert übereinstimmt.

Die $(n \cdot m)$-Matrix $\mathbf{K}_k$ in (38.7), auch Matrix der Kalman-Verstärkungsfaktoren genannt, wird so gewählt, dass die a posteriori Schätzfehlerkovarianz $\mathbf{P}_k^+$ minimal wird. Eine mögliche Form von $\mathbf{K}_k$, die $\mathbf{P}_k^+$ minimiert ist:

$$\mathbf{K}_k = \mathbf{P}_k^- \cdot \mathbf{H}_k^T\left(\mathbf{H}_k\mathbf{P}_k^-\mathbf{H}_k^T + \mathbf{R}_k\right)^{-1} = \frac{\mathbf{P}_k^- \cdot \mathbf{H}_k^T}{\mathbf{H}_k\mathbf{P}_k^-\mathbf{H}_k^T + \mathbf{R}_k} \tag{38.8}$$

Hinweis: Eine ausführliche Herleitung von Gleichung (38.8) ist in [Mayb79] zu finden.

Zur Vertiefung des Verständnisses betrachtet man die Grenzfälle der Gleichung (38.8):

- Falls die Varianz des Messfehlers $\mathbf{R}_k$ gegen null geht, wird die Matrix der Kalman-Verstärkungsfaktoren $\mathbf{K}_k$ maximal. Anders ausgedrückt: Falls die Varianz des Messfehlers $\mathbf{R}_k$ gegen Null geht, wird der aktuellen Messung $\mathbf{y}_k$ immer mehr „vertraut", der prädizierten Messung $\mathbf{H}_k \cdot \hat{\mathbf{x}}_k^-$ wird dagegen immer weniger „vertraut".

- Falls die a priori Schätzfehlerkovarianz $\mathbf{P}_k^-$ gegen null geht, wird die Matrix der Kalman-Verstärkungsfaktoren $\mathbf{K}_k$ ebenfalls null. Anders ausgedrückt: Falls die a priori Schätzfehlerkovarianz $\mathbf{P}_k^-$ gegen null geht, wird der aktuellen Messung $\mathbf{y}_k$ immer weniger „vertraut", der prädizierten Messung $\mathbf{H}_k \cdot \hat{\mathbf{x}}_k^-$ wird dagegen immer mehr „vertraut".

38.3.5 Gleichungen des Kalman-Filters

Das Kalman-Filter schätzt die Entwicklung eines Systemzustands in einer Art Regelschleife durch Rückkopplung über die Messwerte: das Filter schätzt den Systemzustand zu einem bestimmten Zeitpunkt ab und benützt anschließend die Rückkopplung in Form einer verrauschten Messung, um den Systemzustand im nächsten Zeitschritt etwas besser abzuschätzen. Danach kommt wieder die nächste Messung usw. in einer Rekursion. Die Gleichungen des Kalman-Filters kann man deshalb in zwei Gruppen einteilen:

- Die Gleichungen für die zeitliche Entwicklung des Systems (Prädiktion)
- Die Gleichungen für die Rückkopplung durch die Messung, d.h. die Berücksichtigung einer neuen Messung, um aus der a priori Schätzung eine verbesserte a posteriori Schätzung abzuleiten (Korrektur mit der Messung).

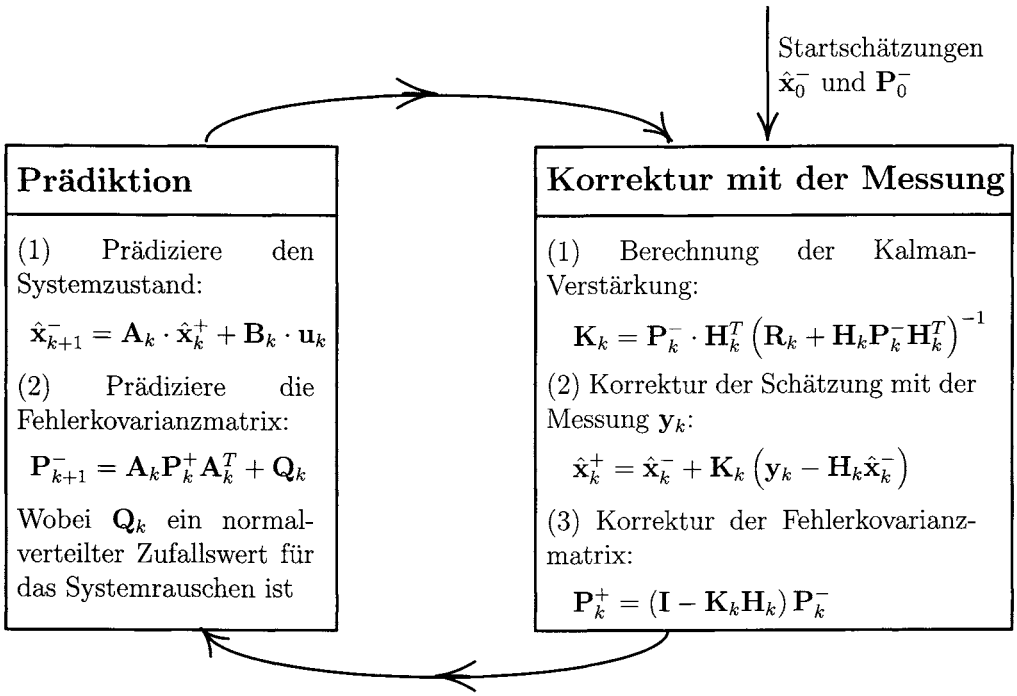


Bild 38.5: Die Regelschleife des Kalman-Filters: Prädiktion und Korrektur mit der Messung.

Eigenschaften des Kalman-Filters und Hinweise:

- Das Kalman-Filter ist rekursiv:
  nach jedem Prädiktions- und Messschritt wird der Vorgang wiederholt, wobei die vorhergehenden a posteriori Schätzungen benützt werden, um die nächsten a priori Schätzungen zu bestimmen.

- In der Praxis ist ein rekursives Filter sehr gut brauchbar, da es mit jeder neuen Messung auch einen neuen, d.h. aktuellen Schätzwert liefert (im Gegensatz z.B. zum Wiener Filter, das immer alle bisherigen Messungen gleichzeitig für die Schätzung braucht). Das Kalman-Filter ist daher prädestiniert für den Einsatz in Echtzeit-Anwendungen.

- Falls die Varianzen des Mess- bzw. Systemrauschens konstant sind, nähern sich sowohl die Matrix der Kalman-Verstärkungsfaktoren $\mathbf{K}_k$ als auch die (a priori und a posteriori) Kovarianzen des Schätzfehlers $\mathbf{P}_k$ sehr schnell konstanten Werten an (die auch vorab, d.h. offline berechnet werden können).

- In der Praxis sind die Messfehler $\mathbf{R}_k$ jedoch meist nicht konstant: falls sich z.B. der Kontrast in Videobildern vorübergehend abschwächt, wird die Objekterkennung entsprechend unsicherer, so dass die entsprechenden Messwerte im Kalman-Filter nicht so stark berücksichtigt werden sollten. Man kann also z.B. den (messbaren) Kontrast als Maß für die Varianz des Messfehlers benutzen und somit den gewünschten Effekt erzielen.

- Auch das Systemrauschen $\mathbf{Q}_k$ ändert sich in manchen Anwendungen dynamisch: z.B. kann beim Tracking einer Person die Größe von $\mathbf{Q}_k$ reduziert werden, falls die Person sich gerade langsam zu bewegen scheint, und falls sich die Bewegungen der Person schnell ändern, wird die Größe von $\mathbf{Q}_k$ erhöht.

- Eine wesentliche Voraussetzung für das Funktionieren eines Kalman-Filters ist die *Beobachtbarkeit* aller Systemzustände durch die Messung [Ludy95]. Falls versucht wird, nicht beobachtbare Systemzustände zu schätzen, divergiert das Kalman-Filter.

- Bei der praktischen Realisierung von Kalman-Filtern treten häufiger numerische Probleme auf. Ursachen dafür können z.B. Singularitäten bei der Matrix-Invertierung für die Berechnung der Kalman-Verstärkung $\mathbf{K}_k = \mathbf{P}_k^- \cdot \mathbf{H}_k^T \left(\mathbf{R}_k + \mathbf{H}_k \mathbf{P}_k^- \mathbf{H}_k^T\right)^{-1}$ sein, unzureichende Maschinengenauigkeit oder ungünstige Implementierung. Abhilfe schaffen hier Methoden zur numerischen Stabilisierung, wie sie in [Scho03] beschrieben sind.

38.3.6 Das erweiterte Kalman-Filter (EKF)

Das bisher hergeleitete Kalman-Filter basiert auf einer linearen stochastischen Differenzengleichung. In der Praxis sind die System- und Messgleichungen jedoch meist nicht-linear ! Die allgemeine Form einer nicht-linearen Systemgleichung lautet:

$$\mathbf{x}_{k+1} = \mathbf{f}\left(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w}_k\right) \tag{38.9}$$

Die allgemeine Form einer nicht-linearen Messgleichung lautet:

$$\mathbf{y}_k = \mathbf{h}\left(\mathbf{x}_k, \mathbf{v}_k\right) \tag{38.10}$$

wobei die nicht-lineare Übergangsfunktion $\mathbf{f}$ den Systemzustand vom vorherigen Zeitschritt k auf den aktuellen Zeitschritt $k+1$ abbildet, und zwar mit Systemrauschen $\mathbf{w}_k$. Die nicht-lineare Messfunktion $\mathbf{h}$ bildet den Systemzustand beim Zeitschritt k auf den Messwert $\mathbf{y}_k$ ab, und zwar mit Messrauschen $\mathbf{v}_k$.

In ähnlichem Sinne wie man eine nicht-lineare Funktion um einen bestimmten Punkt in eine Taylor-Reihe entwickeln und nach dem linearen Glied abbrechen kann, linearisiert man beim erweiterten Kalman-Filter (EKF) die nicht-lineare Übergangs- und Messfunktion jeweils um den aktuellen Schätzwert:

$\delta\mathbf{A}_k$ ist die Jakobi-Matrix der partiellen Ableitungen von $\mathbf{f}$ nach $\mathbf{x}$ zur Zeit k:

$$\delta\mathbf{A}_k = \frac{\partial}{\partial \mathbf{x}}\mathbf{f}\left(\hat{\mathbf{x}}_k^+, \mathbf{u}_k, 0\right) \quad \text{bzw.} \quad \mathbf{A}_{k,[i,j]} = \frac{\partial}{\partial \mathbf{x}_{[j]}}\mathbf{f}_{[i]}\left(\hat{\mathbf{x}}_k^+, \mathbf{u}_k, 0\right) \tag{38.11}$$

$\delta\mathbf{W}_k$ ist die Jakobi-Matrix der partiellen Ableitungen von $\mathbf{f}$ nach $\mathbf{w}$ zur Zeit k:

$$\delta\mathbf{W}_k = \frac{\partial}{\partial \mathbf{w}}\mathbf{f}\left(\hat{\mathbf{x}}_k^+, \mathbf{u}_k, 0\right) \quad \text{bzw.} \quad \mathbf{W}_{k,[i,j]} = \frac{\partial}{\partial \mathbf{w}_{[j]}}\mathbf{f}_{[i]}\left(\hat{\mathbf{x}}_k^+, \mathbf{u}_k, 0\right) \tag{38.12}$$

$\delta\mathbf{H}_k$ ist die Jakobi-Matrix der partiellen Ableitungen von $\mathbf{h}$ nach $\mathbf{x}$ zur Zeit k:

$$\delta\mathbf{H}_k = \frac{\partial}{\partial \mathbf{x}}\mathbf{h}\left(\hat{\mathbf{x}}_k^-, 0\right) \quad \text{bzw.} \quad \mathbf{H}_{k,[i,j]} = \frac{\partial}{\partial \mathbf{x}_{[j]}}\mathbf{h}_{[i]}\left(\hat{\mathbf{x}}_k^-, 0\right) \tag{38.13}$$

$\delta\mathbf{V}_k$ ist die Jakobi-Matrix der partiellen Ableitungen von $\mathbf{h}$ nach $\mathbf{v}$ zur Zeit k:

$$\delta\mathbf{V}_k = \frac{\partial}{\partial \mathbf{v}}\mathbf{h}\left(\hat{\mathbf{x}}_k^-, 0\right) \quad \text{bzw.} \quad \mathbf{V}_{k,[i,j]} = \frac{\partial}{\partial \mathbf{v}_{[j]}}\mathbf{h}_{[i]}\left(\hat{\mathbf{x}}_k^-, 0\right) \tag{38.14}$$

Eine exakte Herleitung des erweiterten Kalman-Filters ist z.B. in [Welc95] zu finden.

Das erweiterte Kalman-Filter läuft nach der Linearisierung im Prinzip genauso ab, wie das Standard-Kalman-Filter für lineare Probleme. Einziger Unterschied: Die Jacobi-Matrizen müssen für jeden Zeitschritt neu berechnet werden, so dass ein entsprechend größerer Rechenaufwand nötig ist.

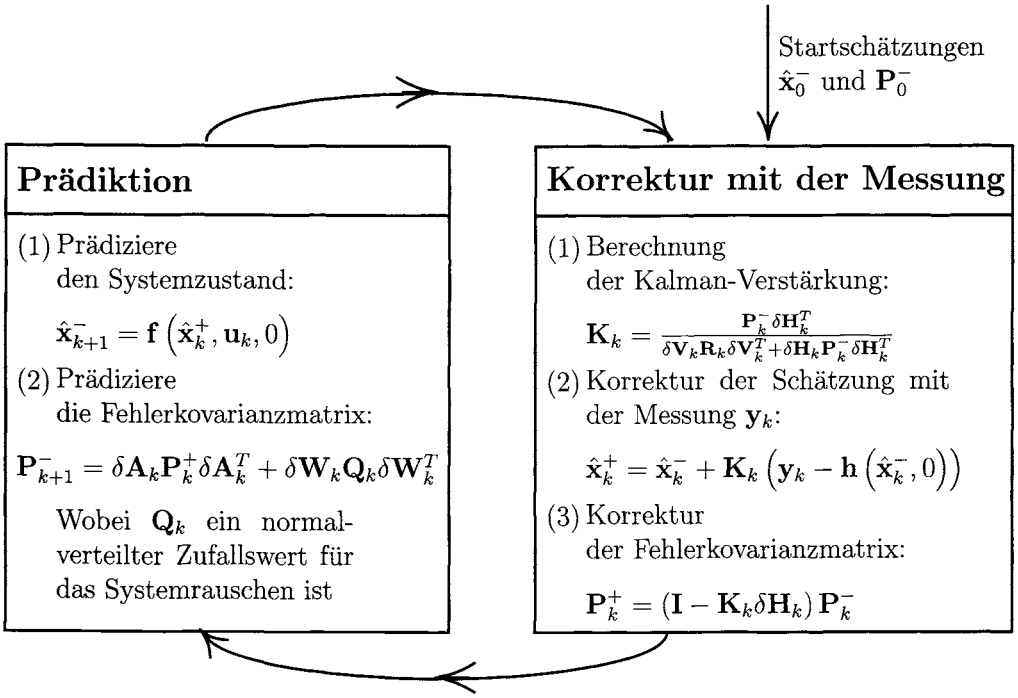


Bild 38.6: Die Regelschleife des erweiterten Kalman-Filters (EKF): Prädiktion und Korrektur mit der Messung. Die Jacobi-Matrizen müssen für jeden Zeitschritt neu berechnet werden.

38.4 Konkrete Beispiele

In diesem Abschnitt werden drei konkrete Beispiele für ein Kalman-Filter vorgestellt, um ein besseres Gefühl für die Auslegung, die Funktionsweise und die Leistungsfähigkeit des Konzepts zu vermitteln. Das erste Beispiel, die Schätzung eines konstanten Wertes auf der Basis verrauschter Messwerte, ist extrem einfach und dient dazu, ein Gefühl für das Grundprinzip des Kalman-Filters zu entwickeln. Es ist empfehlenswert, dieses Beispiel selber nachzurechnen. Das zweite Beispiel, die Positions- und Geschwindigkeitsschätzung einer konstant beschleunigten Bewegung, führt die im ersten Abschnitt dieses Kapitels anhand eines Ballwurfs vorgestellte Grundidee des Kalman-Filters aus. Solche Objektverfolgungsaufgaben, auch *Tracking* genannt, sind ein sehr häufiger Fall für die Anwendung des Kalman-Filters in der Bildverarbeitung. Außerdem wird an dem zweiten Beispiel noch die besondere Leistungsfähigkeit des Kalman-Filters im Hinblick auf Messstörungen dargestellt. Anhand des dritten Beispiels wird schließlich gezeigt, dass das Kalman-Filter auch in der umgekehrten Problemstellung hervorragend eingesetzt werden kann, nämlich bei der Bestimmung der Beobachterposition auf der Basis bekannter Objektpositionen, auch *Navigation* genannt.

38.4.1 Schätzung einer verrauschten Konstante

Der Systemzustand x wird durch einen einzigen Wert (z.B. eine eindimensionale Position) beschrieben, der zeitlich konstant sei. Die Systemmatrix A_k degeneriert in diesem Fall zu einem einzigen Wert, der wegen der zeitlichen Konstanz auch noch 1 ist, d.h. $A_k = 1$ für alle k. Ein Steuer-Vektor existiert ebenfalls nicht, d.h. $u_k = 0$. Damit vereinfacht sich die Systemzustandsgleichung zu:

$$x_{k+1} = A_k \cdot x_k + B_k \cdot u_k + w_k = x_k + w_k \tag{38.15}$$

Man kann die Konstante zu bestimmten Zeitpunkten direkt messen, allerdings sind die Messwerte verrauscht durch ein weißes Messrauschen mit einer Standardabweichung von 0,1. Die Messmatrix H_k degeneriert in diesem Fall zu einem einzigen Wert, der wegen der direkten Messbarkeit des Systemzustands auch noch 1 ist, d.h. $H_k = 1$ für alle k. Damit vereinfacht sich die Messgleichung zu:

$$y_k = H_k \cdot x_k + v_k = x_k + v_k \tag{38.16}$$

Die Kalman-Filter-Gleichungen für den Prädiktionsschritt und den Korrekturschritt durch die Messung sind in Bild 38.7 zusammengefasst:

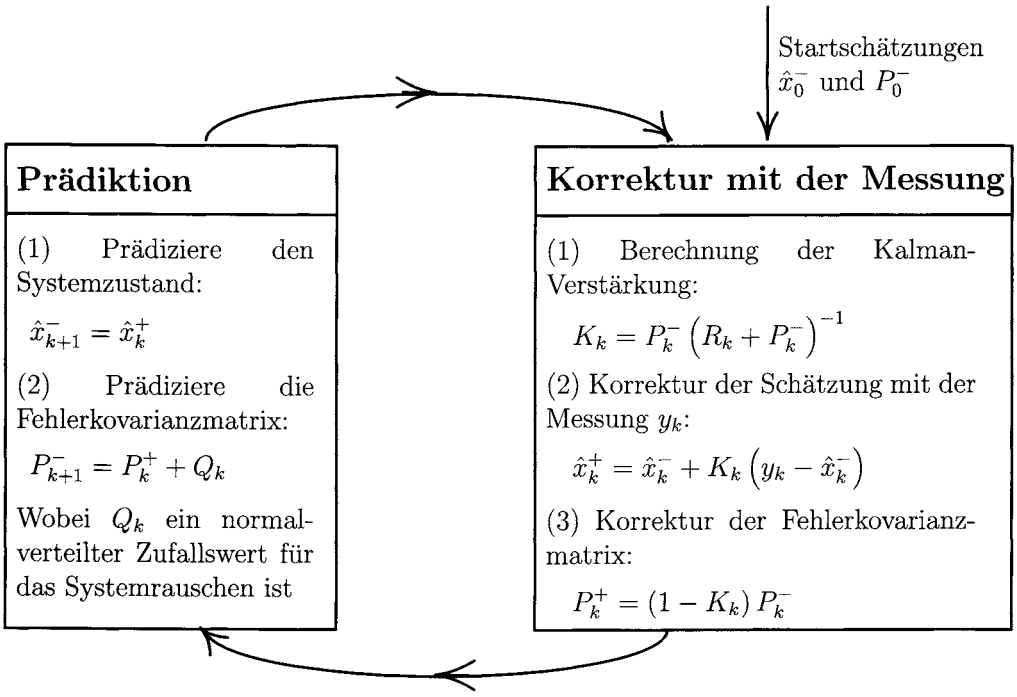


Bild 38.7: Die Regelschleife des Kalman-Filters für die Schätzung einer verrauschten Konstante. Man beachte, dass die Variablen nicht mehr fettgedruckt sind, d.h. es sind nur noch Skalare anstatt Vektoren oder Matrizen.

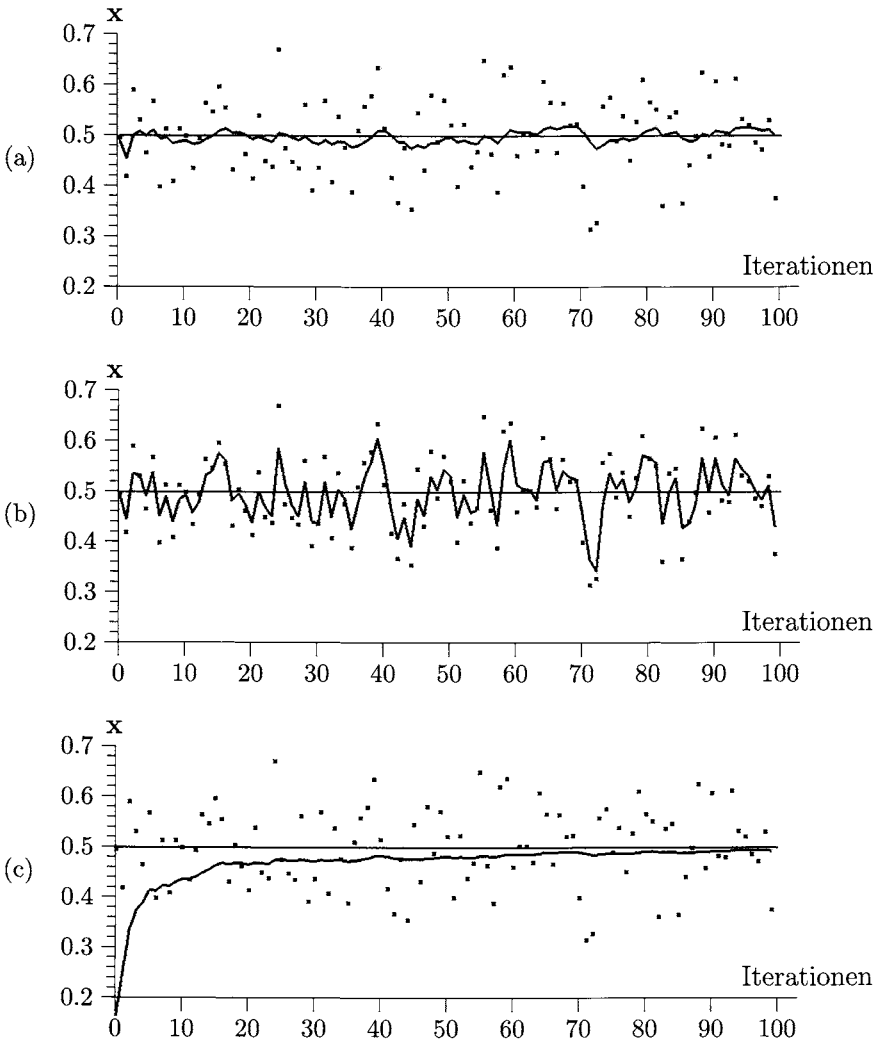


Bild 38.8: Schätzung einer verrauschten Konstante: (a) Simulation 1: $R_k = 0,1^2 = 0,01$. Der Systemzustand $\mathbf{x} = +0,5$ ist durch die gerade Linie dargestellt, die Punkte markieren die verrauschten Messwerte und der Schätzwert des Filters ist die verbleibende dicke Kurve. (b) Simulation 2: $R_k = 0,01^2 = 0,0001$. Das Filter reagiert sehr schnell auf die einzelnen Messwerte, woraus eine größere Varianz des Schätzwerts für den Systemzustand folgt. (c) Simulation 3: $R_k = 1$. Das Filter reagiert langsamer auf die einzelnen Messwerte, woraus eine geringere Varianz des Schätzwerts für den Systemzustand folgt.

Die Simulationsparameter:

Der Systemzustand x wird zufällig ausgewählt und beträgt: $x = +0,5$ (dies ist der „wahre" Wert, deshalb ist kein „Dach" auf dem x). Da man über den „wahren" Systemzustand nichts weiß, beginnt man mit einer beliebigen Startschätzung des Systemzustands

z.B. $\hat{x}_0^- = 0$. Aus dem selben Grund, nämlich der Unkenntnis des „wahren" Systemzustands, beginnt man mit einer beliebigen Fehlerkovarianz größer Null, z.B. $P_0^- = 1$. Da man weiß, dass es sich beim Systemzustand um eine Konstante handelt, könnte man für das Systemrauschen Null annehmen. Dies würde allerdings bei einem niedrigen Messrauschen zum „Einschlafen" des Kalman-Filters führen (d.h. neue Messwerte würden kaum mehr in den Schätzwert einfließen). Deshalb wählt man einen sehr kleinen Wert für das Systemrauschen, z.B. $Q_k = 10^{-5}$.

Jetzt werden 100 verschiedene „Messwerte" y_k simuliert, indem man zum wahren Systemzustand jeweils einen normalverteilten Zufallswert mit einer Standardabweichung von 0,1 addiert. Diese 100 „Messwerte" werden für die Simulationen 1,2,3 benutzt, um die Auswirkungen unterschiedlicher Parametereinstellungen des Kalman-Filters besser vergleichen zu können.

Simulation 1: Varianz des Messfehlers $R_k = (0,1)^2 = 0,01$ (Bild 38.8-a).
Dies ist das „wahre" Messrauschen, da der Messfehler eine Standardabweichung von 0,1 hat. Deshalb erwartet man für diese Parametereinstellung die beste Leistung des Kalman-Filters im Sinne eines Gleichgewichts zwischen Reaktionsvermögen auf neue Messwerte und Varianz des Schätzfehlers (dies wird offensichtlich bei den Simulationen 2 und 3, bei denen Abweichungen vom Gleichgewicht in beide Richtungen auftreten).

Simulation 2: Varianz des Messfehlers $R_k = (0,01)^2 = 0,0001$ (Bild 38.8-b).
d.h. die angenommene Varianz des Messfehlers ist um Faktor 100 kleiner als bei Simulation 1 (dem „wahren" Wert). Es wird also (zu Unrecht) angenommen, dass die Messungen sehr genau sind und man deshalb jeder einzelnen Messung relativ gut vertrauen kann. Folglich reagiert das Kalman-Filter sehr schnell auf die einzelnen Messwerte, die jetzt zu einem großen Anteil in den neuen Schätzwert für den Systemzustand einfließen.

Simulation 3: Varianz des Messfehlers $R_k = 1$ (Bild 38.8-c).
d.h. die angenommene Varianz des Messfehlers ist um Faktor 100 größer als bei Simulation 1 (dem „wahren" Wert). Es wird also (zu Unrecht) angenommen, dass die Messungen sehr stark verrauscht sind und man deshalb jeder einzelnen Messung relativ wenig Vertrauen entgegen bringen kann. Folglich lässt das Kalman-Filter die einzelnen Messwerte nur sehr langsam, also nur mit einem geringen Anteil in den neuen Schätzwert für den Systemzustand einfließen.

Die Wahl der anfänglichen Fehlerkovarianz P_0^- ist unkritisch (solange $P_0^- \neq 0$), da der Wert der Fehlerkovarianz sehr schnell von 1 auf ca. 0,0002 konvergiert.

38.4.2 Schätzung einer Wurfparabel

Im ersten Abschnitt dieses Kapitels wurde anhand eines Ballwurfs die Grundidee des Kalman-Filters vorgestellt. Jetzt wird für dieses Beispiel, in dem es um die Positions- und Geschwindigkeitsschätzung einer konstant beschleunigten Bewegung geht, ein Kalman-Filter ausgelegt. Als Input liegen Messdaten zur Position des Objekts zu diskreten Zeitpunkten vor. Die Aufgabe, die Bahn eines Objekts im Raum zu verfolgen, auch *Tracking* genannt, ist ein sehr häufiger Fall für die Anwendung des Kalman-Filters in der Bildverarbeitung.

Hier wird der Fall der Bewegung eines Massepunkts mit konstanter Beschleunigung (der Erdbeschleunigung g) behandelt. Der Massepunkt habe eine bestimmte Anfangsgeschwindigkeit $\mathbf{v}_0 > 0$, die Anfangsposition sei im Ursprung des Koordinatensystems. Die ideale Bahn des Massepunkts, eine Wurfparabel, ist aus der Physik bekannt. Die Bewegung lässt sich also in zwei Dimensionen x, y beschreiben, wobei x nach rechts und y nach oben bzgl. des Beobachters zeigen soll. Als Systemzustände werden die Position und die Geschwindigkeit des Massepunkts gewählt, d.h. $\mathbf{x} = (x, y, v_x, v_y)^T$, die treibende Kraft ist die Erdbeschleunigung, d.h. $\mathbf{u} = (0, -g)^T$. Die stochastische Systemzustandsgleichung lautet daher:

$$\begin{array}{rcccccccl} \mathbf{x}_{k+1} & = & \mathbf{A}_k & \cdot & \mathbf{x}_k & + & \mathbf{B}_k & \cdot & \mathbf{u}_k + \mathbf{w}_k \\ \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}_{k+1} & = & \begin{pmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \cdot & \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}_k & + & \begin{pmatrix} 0 & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ 0 & 0 \\ 0 & \Delta t \end{pmatrix} & \cdot & \begin{pmatrix} 0 \\ -g \end{pmatrix} + \mathbf{w}_k \end{array} \quad (38.17)$$

Man kann die Position des Objekts zu bestimmten Zeitpunkten direkt messen, allerdings sind die Messwerte in y-Richtung verrauscht durch ein weißes Messrauschen mit einer Standardabweichung von 0,1. Die Messgleichung lautet in diesem Fall:

$$\begin{array}{rcccl} \mathbf{y}_k & = & \mathbf{H}_k & \cdot & \mathbf{x}_k + \mathbf{v}_k \\ \begin{pmatrix} x_m \\ y_m \end{pmatrix}_k & = & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} & \cdot & \begin{pmatrix} x \\ y \\ v_x \\ v_y \end{pmatrix}_k + \mathbf{v}_k \end{array} \quad (38.18)$$

Die Kalman-Filter-Gleichungen für den Prädiktionsschritt und den Korrekturschritt durch die Messung lauten genau so, wie die in Bild 38.5 dargestellten.

Die Simulationsparameter:
Der Systemanfangszustand lautet: $\mathbf{x} = (x, y, v_x, v_y)^T = (0, 0, 0.5 \cdot v_0, 0.86 \cdot v_0)^T$ (dies ist der „wahre" Wert, deshalb ist kein „Dach" auf dem x). Da man über den „wahren" Systemzustand nichts weiß, beginnt man mit irgend einer Startschätzung des Systemzustands, z.B. $\hat{\mathbf{x}}_0^- = (0, 0, 1, 1)^T$. Aus dem selben Grund der Unkenntnis des „wahren" Systemzustands beginnt man mit einer Fehlerkovarianzmatrix $\mathbf{P}_0^-$, bei der alle Komponenten 1 sind. Kleine

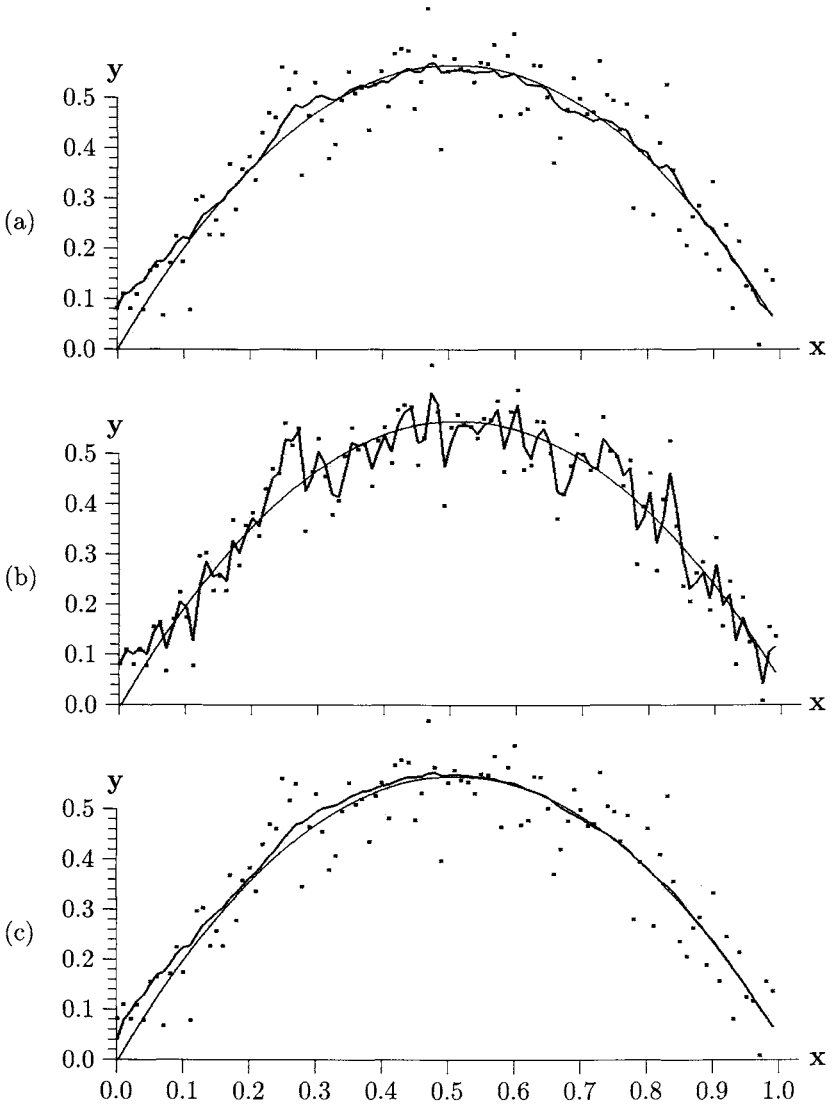


Bild 38.9: Schätzung einer Wurfparabel. (a) Simulation 1: $R_k = 0,1^2 = 0{,}01$. Die Position des Massepunkts ist durch die glatte Parabel dargestellt, die Punkte markieren die verrauschten Messwerte, und der Schätzwert des Filters ist die verbleibende dicke Kurve. (b) Simulation 2: $R_k = 0,01^2 = 0{,}0001$. Das Messrauschen wird um einen Faktor 100 zu klein angenommen, so dass den einzelnen Messwerten zu sehr „vertraut" wird. Das Filter reagiert daher sehr schnell auf die einzelnen Messwerte, woraus eine größere Varianz des Schätzwerts für den Systemzustand folgt. (c) Simulation 3: $R_k = 1$. Das Messrauschen wird um einen Faktor 100 zu groß angenommen, so dass den einzelnen Messwerten zu wenig „vertraut" wird. Das Filter reagiert zu langsam auf die einzelnen Messwerte, woraus zwar eine geringere Varianz des Schätzwerts für den Systemzustand folgt, aber auch eine große Trägheit des Filters.

Luftwirbel stören die Bewegung des Massepunkts etwas, deshalb wählt man einen sehr kleinen Wert für die Komponenten der Systemrauschmatrix, z.B. 10^{-5}.

Jetzt werden 100 verschiedene „Messwerte" $\mathbf{y}_k$ simuliert, indem man zur y-Koordinate des wahren Systemzustands jeweils einen normalverteilten Zufallswert mit einer Standardabweichung von 0,1 addiert. Diese 100 „Messwerte" werden für die Simulationen 1,2,3 benutzt, um die Auswirkungen unterschiedlicher Parametereinstellungen des Kalman-Filters besser vergleichen zu können.

Die Simulationen 1,2,3 laufen bis auf das geänderte Systemmodell genau so ab, wie im vorigen Abschnitt 38.4.1 beschrieben. Bei der Simulation 1 entspricht die angenommene Varianz des Messfehlers $R_k = (0,1)^2 = 0,01$ (Bild 38.9-a) dem „wahren" Messrauschen. Bei der Simulation 2 ist die angenommene Varianz des Messfehlers $R_k = 0,0001$ (Bild 38.9-b) um den Faktor 100 kleiner als das „wahre" Messrauschen. Bei der Simulation 3 ist die angenommene Varianz des Messfehlers $R_k = 1$ (Bild 38.9-c) um den Faktor 100 größer als das „wahre" Messrauschen.

Besonders interessant ist die Leistungsfähigkeit des Kalman-Filters im Hinblick auf Messstörungen. Eine typische Problematik beim Tracking ist die kurzzeitige Verdeckung der Objektbahn durch irgend welche Hindernisse, wie z.B. Häuser, Bäume oder Hügel. Das Objekt kann daher im Suchfenster, dessen Position und Größe durch das Kalman-Filter vorgegeben wird, nicht detektiert werden. Da das Kalman-Filter ein Modell der zeitlichen Entwicklung des Systems, sowie Daten über die letzte gemessene Position und Geschwindigkeit besitzt, kann die Objektbahn für einige Zeitschritte auch ohne Messwerte prädiziert werden. Nachdem das Objekt im prädizierten Suchfenster wieder detektierbar ist und somit neue (bzw. verlässliche) Messwerte zur Verfügung stehen, kann das Kalman-Filter wieder wie gewohnt abwechselnd Korrektur- und Prädiktionschritte durchführen.

Zur Simulation einer Messstörung wurden die simulierten Messwerte im Intervall $x \in [0.6, 0.7]$ stark verfälscht, d.h. die y-Komponente der Position wurde einfach auf den Wert 0.1 gesetzt und anschließend verrauscht. Bei dieser Messstörung liegen also zu jedem Zeitschritt Messwerte vor, was die Sache eher noch erschwert. In der Simulation 4 (Bild 38.10-a) ist die Varianz des Messfehlers $R_k = 0,01$ konstant, da kein Suchfenster verwendet wurde, in dem die Messwerte liegen müssen. Dies führt dazu, dass die geschätzte Position des Objekts den gestörten Messwerten relativ rasch folgt und anschließend mit einer gewissen Trägheit wieder auf die ungestörte Objektbahn einschwenkt. Ein sehr viel besseres Verhalten des Filters kann man erzielen, wenn man ein Suchfenster definiert, das umso kleiner wird, je verlässlicher die Messwerte sind (d.h. je kleiner die Fehlerkovarianz ist). In der Simulation 5 (Bild 38.10-b) liegen die gestörten Messwerte außerhalb des Suchfensters und werden deshalb nicht berücksichtigt. Das Kalman-Filter läuft deshalb im Prädiktionsmodus, Korrekturen durch die Messwerte gehen praktisch nicht in das Filter ein. Da vor der Störung bereits gute Schätzungen für die Position und die Geschwindigkeit des Objekts vorliegen, kann die Bahn des Objekts auch ohne Messwerte für einen gewissen Zeitraum gut prädiziert werden. Nachdem die Messstörung vorüber ist (Position $x = 0.7$), liegen wieder Messwerte innerhalb des (evtl. vergrößerten) Suchfensters vor, und die Schätzung der Objektbahn kann wieder durch Messwerte gestützt werden.

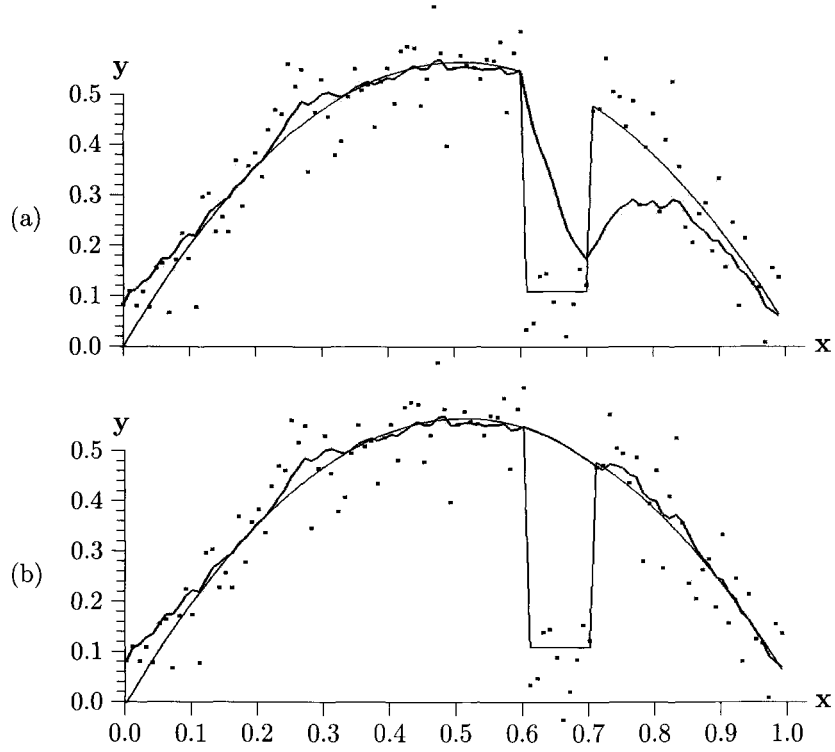


Bild 38.10: Schätzung einer Wurfparabel mit Störung (bei den simulierten Messwerten im Intervall $x \in [0.6, 0.7]$ wurden die y-Komponenten der Position auf den Wert 0.1 gesetzt). (a) Simulation 4: $R_k = 0,01$ konstant auch während der Störung. Die Schätzwerte für die Objektposition werden durch die Störung stark beinflusst. (b) Simulation 5: $R_k = 1$ während der Störung, $R_k = 0,01$ sonst. Die Störung wird detektiert und die Varianz des Messfehlers wird um einen Faktor 100 hochgesetzt, so dass die (falschen) Messwerte im Kalman-Filter praktisch nicht mehr berücksichtigt werden. Das Kalman-Filter läuft deshalb im Prädiktionsmodus und kann nach dem Ende der Störung wieder an der richtigen Stelle weiterarbeiten.

38.4.3 Bildbasierte Navigation

Bei dem vorherigen Beispiel ist man davon ausgegangen, dass der Beobachter bzw. die Kamera an einem bekannten Ort fest steht. Geschätzt wurde die Position des bewegten Objektes. Das Kalman-Filter kann man auch bei der umgekehrten Problemstellung hervorragend einsetzen: Der Schätzung der Beobachter- bzw. Kameraposition auf der Basis bekannter Positionen von Objekten bzw. Landmarken. Aufgaben dieser Art werden als *Navigation* bezeichnet, speziell als *Bildbasierte Navigation*, da als Sensordaten Kamerabilder dienen.

Navigation ist eine der grundlegenden Funktionen eines autonomen mobilen Roboters. Bestückt mit einer kalibrierten Kamera soll die Position der Kamera und damit auch des Roboters aus den abgebildeten Landmarken berechnet werden. Voraussetzung dafür ist,

dass die Positionen der Landmarken vorab bekannt sind. Dies ist in einem kontrollierbaren Umfeld, wie z.B. in einer Fabrikhalle durchwegs gegeben. Um das Beispiel möglichst einfach zu gestalten, geht man von einer statischen Situation aus, d.h. weder die Kamera noch die Landmarken bewegen sich. Damit wird die stochastische Systemzustandsgleichung trivial, da die Systemmatrix zur Einheitsmatrix wird ($\mathbf{A} = \mathbf{I}$) und die Steuermatrix Null ist ($\mathbf{B} = 0$):

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{w}_k \tag{38.19}$$

Im Gegensatz zu dem vorigen Beispiel, der Schätzung einer Wurfparabel, steckt die Komplexität bei diesem Beispiel allein in der Messung. Deshalb wurde es ausgewählt. Bei einer realen Kamera als Sensor hat man es mit einer perspektivischen Projektion zu tun, die eine nicht-lineare Abbildung darstellt (Abschnitt 7.6.2). Aufgrund dieser Nichtlinearität muss hier das erweiterte Kalman-Filter (EKF, Abschnitt 38.3.6) zum Einsatz kommen. Im Folgenden wird zunächst die Bildvorverarbeitung kurz dargestellt und anschließend die Herleitung der linearisierten Messmatrix beschrieben.

Bildvorverarbeitung

In der Arbeit von Lohmann [Lohm88], von der dieses Beispiel abgeleitet wurde, bedient man sich einiger einfacher Tricks, um die Landmarken vom Hintergrund und voneinander zu unterscheiden. Als Landmarken werden Kombinationen von jeweils sechs blinkenden Leuchtdioden eingesetzt, die äquidistant auf einer dunklen Box angebracht sind. Zieht man die Grauwerte zweier Bilder pixelweise von einander ab, so wird das entstehende Differenzbild überall dort dunkel sein, wo die beiden verwendeten Bilder übereinstimmen. Nimmt man bei ruhender Kamera und ruhender Umgebung eine Anordnung von blinkenden Leuchtdioden auf, so werden bei Differenzbildung der zeitlich nacheinander gewonnenen Bilder die Leuchtdioden sehr deutlich hervortreten (die Frequenz der blinkenden Leuchtdioden muss in sinnvoller Relation zur Bildrate der Kamera stehen). Die LED-Landmarken können dann mit einfachsten Mitteln der Bildverarbeitung, wie z.B. Schwellwertbildung, erkannt werden. Die LED 1 und 6 blinken im Gleichtakt bei gleicher Hell- wie Dunkelzeit. LED 2 bis LED 5 blinken dazu gleich- oder gegenphasig. Dadurch lassen sich $2^4 = 16$ verschiedene Codierungen einstellen, so dass eine entsprechende Anzahl an Landmarken identifiziert werden kann. Die Abstände der 6 LEDs untereinander, sowie die Position der LEDs im Raum sind vorab bekannt.

Herleitung der linearisierten Messmatrizen

Falls eine Landmarke im Bild (mit bekannter Position) erkannt wird, kann über die Größe der LED-Kette im Bild der Abstand zu dieser LED-Kette berechnet werden. Folglich kann sich die Kamera in einem Kreis[3] um die Landmarke befinden (Bild 38.11-a). Werden zwei

[3]Im dreidimensionalen Raum wäre es eine Kugeloberfläche. Hier wird aber nur das etwas einfachere zweidimensionale Navigationsproblem behandelt. Die Erweiterung auf drei Dimensionen erfordert vor allem mehr Schreibarbeit, das prinzipielle Vorgehen ist aber das Gleiche

Landmarken erkannt, ergeben sich normalerweise zwei Kreise, die sich schneiden (Bild 38.11-b). Einer der beiden Schnittpunkte ist der Standort der Kamera $\mathbf{x}_C = (x_C, y_C)^T$. Durch weitere Landmarken kann ein Schnittpunkt ausgeschlossen werden. Allerdings werden sich wegen der unvermeidlichen Messfehler nicht alle Kreise in exakt einem Punkt schneiden. Weitere Informationen über den Standort liefert der Peilwinkel ϑ zwischen zwei erkannten Landmarken, der aus dem Bild der Kamera ebenfalls ermittelt werden kann. Man hat es also bei einer ausreichenden Zahl erkannter Landmarken mit einem überbestimmten Problem zu tun. Das Kalman-Filter dient zur Fusion dieser Informationen und liefert neben der besten Schätzung für die Kameraposition $\mathbf{x}_C$ auch eine Kovarianzmatrix $\mathbf{P}$ des Schätzfehlers. Zunächst werden die für die Schätzung der Abstände l_i und Peilwinkel ϑ nötigen Gleichungen hergeleitet. Anschließend werden diese Gleichungen linearisiert, damit sie in den Messmatrizen verwendbar sind.

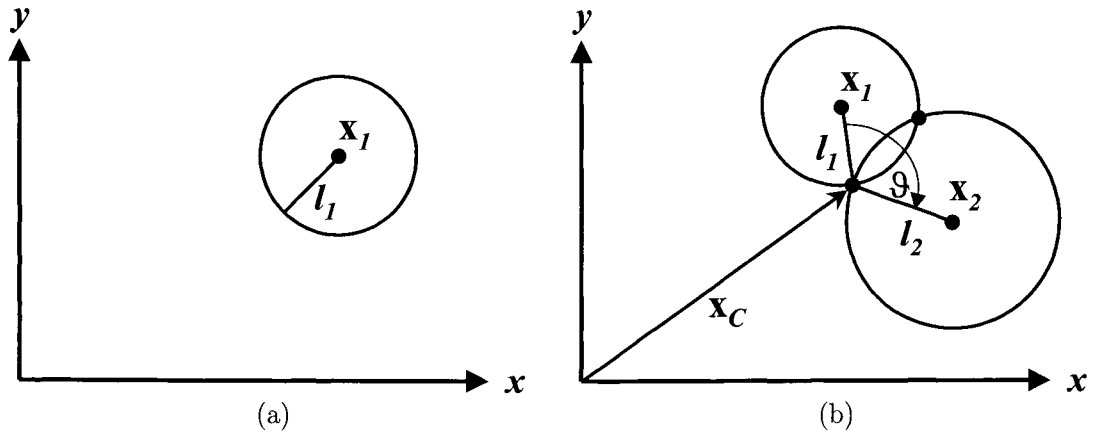


Bild 38.11: (a) Eine Landmarke: Die Kamera kann sich auf einem Kreis mit Radius l_1 um die Landmarke befinden. (b) Zwei Landmarken: Die Kamera kann sich an einem der beiden Kreis-Schnittpunkte befinden. Aus je zwei Landmarken kann ein Peilwinkel ϑ aus dem Bild berechnet werden.

Verwertung der Längenmessung

Zur Längenmessung mit der kalibrierten Kamera wird der Strahlensatz verwendet: Das Verhältnis zwischen der Bildweite f und dem Abstand l vom Brennpunkt der Kamera zu einem Objekt ist gleich dem Verhältnis zwischen der Länge des Objekts z und der Länge z', unter der das Objekt in der Bildebene erscheint (Bild 38.12):

$$\frac{f}{l} = \frac{z'}{z} \tag{38.20}$$

In (38.20) ist die Bedingung eingeflossen, dass z (d.h. die Verbindungsgerade der sechs Leuchtdioden) senkrecht auf der optischen Achse steht. Da z und f vorab bekannt sind,

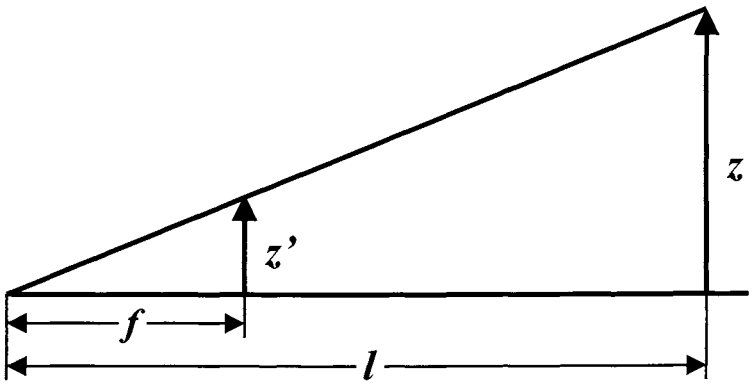


Bild 38.12: Abstandsmessung mit der kalibrierten Kamera: l = Abstand vom Brennpunkt der Kamera zu einer Landmarke. f = Bildweite in Pixel. z = Länge des Objekts (Abstand von der ersten zur sechsten Leuchtdiode) z' = Länge des Objekts in der Bildebene in Pixel.

kann aus der Messung von z' der Abstand der Kamera zu einer erkannten Landmarke aus (38.20) errechnet werden.

Die Abstandsmessungen werden nun zur Schätzung der Kameraposition im Rahmen des Kalman-Filters verwertet. Bei n erkannten Landmarken liegen n Größen z'_k, $k = 1, .., n$ vor, die die Höhen der LED-Ketten in der Bildebene beschreiben. Daraus resultieren nach (38.20) n indirekte Abstandsmessungen:

$$l_k = \frac{f}{z'_k} z \tag{38.21}$$

Die Kameraposition $\mathbf{x}_C = (x_C, y_C)^T$ und die Landmarkenpositionen $\mathbf{x}_k = (x_k, y_k)^T$ stehen über die Kreisgleichung in Beziehung zu den Abstandsmessungen:

$$l_k^2 = (x_k - x_C)^2 + (y_k - y_C)^2 \quad , \qquad k = 1, .., n \tag{38.22}$$

Mit Gleichung (38.21) folgt daraus die nicht-lineare Messgleichung für z'_k:

$$z'_k = \frac{z \cdot f}{\sqrt{(x_k - x_C)^2 + (y_k - y_C)^2}} + v_k = h_z(x_C, y_C) + v_k \tag{38.23}$$

Hierbei ist v_k eine Zufallsvariable, die das $N(0, R_k)$-verteilte Messrauschen repräsentiert. Die nicht-lineare Funktion h_z bildet den Systemzustand (hier die Kameraposition $\mathbf{x}_C$) auf den Messwert ab.

Voraussetzung für den Einsatz des Kalman-Filters ist eine lineare Messgleichung. Deshalb wird (38.23) um einen Schätzwert $\hat{\mathbf{x}}_C = (\hat{x}_C, \hat{y}_C)^T$ für die Kameraposition $\mathbf{x}_C$ linearisiert:

$$z'_k \approx h_z(\hat{\mathbf{x}}_C) + \left.\frac{\partial h_z}{\partial \mathbf{x}_C}\right|_{\hat{\mathbf{x}}_C} \cdot (\mathbf{x}_C - \hat{\mathbf{x}}_C) + v_k \tag{38.24}$$

Mit der Definition der Messmatrix $\mathbf{H}_k$:

$$\begin{aligned}\mathbf{H}_k &= \left.\frac{\partial h_z}{\partial \mathbf{x}_C}\right|_{\hat{\mathbf{x}}_C} = \left.\left(\frac{\partial h_z}{\partial x_C}, \frac{\partial h_z}{\partial y_C}\right)\right|_{\hat{\mathbf{x}}_C} \\ &= \left(\frac{z \cdot f \cdot (x_k - \hat{x}_C)}{\sqrt{(x_k - \hat{x}_C)^2 + (y_k - \hat{y}_C)^2}^3}, \frac{z \cdot f \cdot (y_k - \hat{y}_C)}{\sqrt{(x_k - \hat{x}_C)^2 + (y_k - \hat{y}_C)^2}^3}\right)\end{aligned} \tag{38.25}$$

und einer Umformung von Gleichung (38.24) folgt:

$$z'_k - h_z(\hat{\mathbf{x}}_C) + \mathbf{H}_k \cdot \hat{\mathbf{x}}_C = \mathbf{H}_k \cdot \mathbf{x}_C + v_k \tag{38.26}$$

Mit der Definition von

$$y_k = z'_k - h_z(\hat{\mathbf{x}}_C) + \mathbf{H}_k \cdot \hat{\mathbf{x}}_C \tag{38.27}$$

lässt sich (38.26) in der Form einer linearen Messgleichung schreiben, wie im normalen Kalman-Filter[4] üblich (38.2):

$$y_k = \mathbf{H}_k \cdot \mathbf{x}_C + v_k \tag{38.28}$$

Der Kalman-Filter Algorithmus zur Schätzung der Kameraposition $\mathbf{x}_C$ mit Hilfe der Abstandsmessungen ist in **A38.1** zusammengefasst.

[4]Bis auf den gesuchten Systemzustand $\mathbf{x}_C$ (Position der Kamera), der im Allgemeinen veränderlich ist und somit $\mathbf{x}_{C,k}$ heißen müsste, aber im Spezialfall einer ruhenden Kamera konstant ist, so dass $\mathbf{x}_{C,k} = \mathbf{x}_C$.

A38.1: Algorithmus zur Schätzung der Kameraposition mit Hilfe der Abstandsmessungen.

Voraussetzungen und Bemerkungen:

◇ Bestimmung einer Startschätzung für die Kameraposition $\hat{\mathbf{x}}_{C,0}$ und einer Fehlerkovarianzmatrix $\mathbf{P}_0^-$ dieser Schätzung.

Algorithmus:

(a) Für alle erkannten Landmarken:

(aa) Berechnung der Messmatrix $\mathbf{H}_k$ nach Gleichung (38.25)

(ab) Berechnung der Differenz $(y_k - \mathbf{H}_k \cdot \hat{\mathbf{x}}_{C,k}^-)$ nach Gleichung (38.27). Der aktuelle Messwert z_k' und der vorhergesagte Messwert $h_z(\hat{\mathbf{x}}_{C,k}^-)$ sind bekannt.

(ac) Durchführung des Korrekturschritts:

(aca) Berechnung der Kalman-Verstärkung:
$\mathbf{K}_k = \mathbf{P}_k^- \cdot \mathbf{H}_k^T \left(\mathbf{R}_k + \mathbf{H}_k \mathbf{P}_k^- \mathbf{H}_k^T\right)^{-1}$

(acb) Korrektur der Schätzung mit der Messung:
$\hat{\mathbf{x}}_{C,k}^+ = \hat{\mathbf{x}}_{C,k}^- + \mathbf{K}_k \left(\mathbf{y}_k - \mathbf{H}_k \hat{\mathbf{x}}_{C,k}^-\right)$

(acc) Korrektur der Fehlerkovarianzmatrix:
$\mathbf{P}_k^+ = (\mathbf{I} - \mathbf{K}_k \mathbf{H}_k)\, \mathbf{P}_k^-$

(ad) Durchführung des Prädiktionsschritts:

(ada) Prädiktion des Systemzustands nach (38.19):
$\hat{\mathbf{x}}_{C,k+1}^- = \hat{\mathbf{x}}_{C,k}^+$

(ada) Prädiktion der Fehlerkovarianzmatrix:
$\mathbf{P}_{k+1}^- = \mathbf{P}_k^+ + \mathbf{Q}_k$
Wobei $\mathbf{Q}_k$ ein normal-verteilter Zufallswert für das Systemrauschen ist.

Ende des Algorithmus

Verwertung der Peilwinkelmessung

In Bild 38.13 ist das Prinzip dargestellt, nach dem der horizontale Peilwinkel ϑ, unter dem zwei Landmarken erscheinen, berechnet wird.

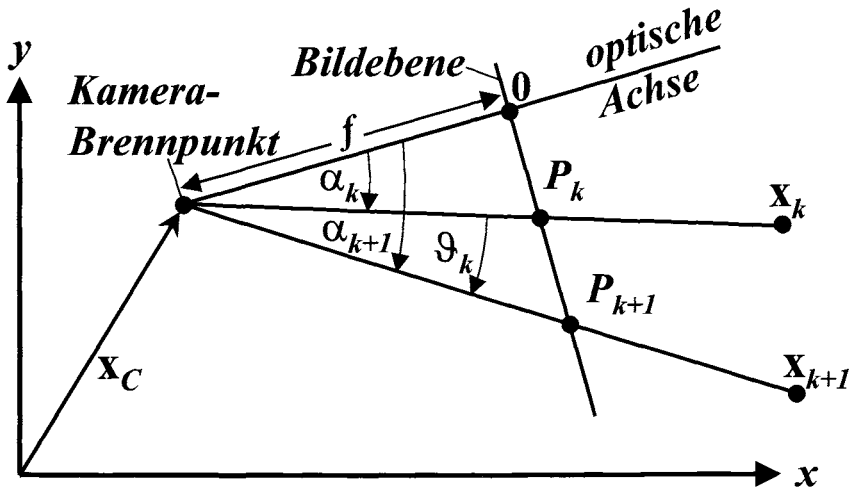


Bild 38.13: Peilwinkelmessung mit der kalibrierten Kamera: aus den Abständen P_k und P_{k+1} der Landmarken von der Bildmitte lässt sich der Peilwinkel ϑ_k bestimmen.

Aus den Abständen P_k und P_{k+1} der auf die Bildebene projizierten Landmarken von der Bildmitte lassen sich zunächst die Winkel α_k und α_{k+1} bestimmen:

$$\alpha_k = \arctan \frac{P_k}{f} \tag{38.29}$$

Wie aus Bild 38.13 ersichtlich, ist der Peilwinkel ϑ_k der Betrag der Differenz zwischen den Winkeln α_{k+1} und α_k:

$$\vartheta_k = |\alpha_{k+1} - \alpha_k| = \left| \arctan \frac{P_{k+1}}{f} - \arctan \frac{P_k}{f} \right| \tag{38.30}$$

Auf diese Weise kann man aus den n Landmarken $n-1$ Winkel ϑ_k, $k = 1, ..., n-1$ ermitteln, die zur Ortsbestimmung der Kamera beitragen sollen.

Als Nächstes wird der Zusammenhang zwischen dem horizontalen Peilwinkel ϑ und dem Systemzustand $\mathbf{x}_C$ (Position der Kamera) abgeleitet. Die Positionen der Landmarken $\mathbf{x}_k = (x_k, y_k)^T$ sind im Weltkoordinatensystem bekannt. $\mathbf{x}_C = (x_C, y_C)^T$ ist der unbekannte Ortsvektor des Kamerabrennpunktes.

Für das Skalarprodukt zweier Vektoren gilt per Definition:

$$\begin{aligned} \vartheta_k &= \arccos \frac{(\mathbf{x}_k - \mathbf{x}_C) \cdot (\mathbf{x}_{k+1} - \mathbf{x}_C)}{|\mathbf{x}_k - \mathbf{x}_C| \cdot |\mathbf{x}_{k+1} - \mathbf{x}_C|} \\ &= \arccos \frac{(x_k - x_C) \cdot (x_{k+1} - x_C) + (y_k - y_C) \cdot (y_{k+1} - y_C)}{\sqrt{(x_k - x_C)^2 + (y_k - y_C)^2} \cdot \sqrt{(x_{k+1} - x_C)^2 + (y_{k+1} - y_C)^2}} \\ &= h_\vartheta(\mathbf{x}_C) \end{aligned} \tag{38.31}$$

womit der Zusammenhang zwischen Messgröße ϑ_k und Zustandsgröße $\mathbf{x}_C$ hergestellt ist. Die nicht-lineare Funktion h_ϑ bildet den Systemzustand (hier die Kameraposition $\mathbf{x}_C$) auf den Messwert ab.

Das weitere Vorgehen ist das gleiche wie bei der Verwertung der Längenmessung. Zunächst erfolgt die Linearisierung der Messgleichung (38.31) um einen Schätzwert $\hat{\mathbf{x}}_C = (\hat{x}_C, \hat{y}_C)^T$ für die Kameraposition. Die Messmatrix $\mathbf{H}_k$ für die Peilwinkelmessung lautet:

$$\mathbf{H}_k = \left.\frac{\partial h_\vartheta}{\partial \mathbf{x}_C}\right|_{\hat{\mathbf{x}}_C} = \left.\left(\frac{\partial h_\vartheta}{\partial x_C}, \frac{\partial h_\vartheta}{\partial y_C}\right)\right|_{\hat{\mathbf{x}}_C} \tag{38.32}$$

Damit lässt sich (38.31) in eine lineare Messgleichung vom Typ (38.2) umformen. Nun kann der Kalman-Filter Algorithmus **A38.1** zur Verbesserung der Schätzung der Kameraposition $\mathbf{x}_C$ mit Hilfe der Peilwinkelmessungen gestartet werden. Als Startschätzung geht man hier vom Ergebnis $\hat{\mathbf{x}}_{C,n}^-$ der letzten Schätzung auf Basis der Längenmessungen aus. Als Messmatrix muss (38.32) anstatt (38.25) verwendet werden. Der Algorithmus **A38.1** wird dieses Mal für die $(n-1)$ Peilwinkelmessungen durchgeführt. Damit sind alle Informationen für die Schätzung der Kameraposition ausgewertet, die die ruhende Kamera liefert.

Das dargestellte Prinzip ist natürlich erweiterbar:

- Auf Situationen, bei denen der Roboter (bzw. die Kamera) bewegt wird.
- Auf Situationen, bei denen sich sowohl der Roboter also auch die Landmarken (bzw. Objekte) bewegen. Falls Bewegung im Spiel ist, erreicht die Zustandsübergangsgleichung wieder die volle Komplexität (38.1), da jetzt die Eigen- bzw. Objektdynamik in der Systemmatrix $\mathbf{A}_k$ berücksichtigt werden muss. Entsprechend komplexer wird auch die Gleichung für die Prädiktion der Fehlerkovarianzmatrix.
- Auf Situationen, bei denen eine Stereo-Kamera (d.h. zwei Kameras oder auch mehr) eingesetzt wird.
- Auf Situationen, bei denen andere Navigations-Sensoren (z.B. Beschleunigungsmesssysteme, Odometer, GPS (Global Positioning System) oder Höhenmesser) eingesetzt werden.
- Auf kombinierte Situationen, bei denen mehrere Sensoren gleichzeitig eingesetzt werden [Scho03].

Kapitel 39

Zusammenfassen von Segmenten zu Objekten

39.1 Bestandsaufnahme und Anwendungen

In diesem abschließenden Kapitel werden einige Verfahren erläutert, wie in einem segmentierten Bild die Segmente zu Objektstrukturen zusammengefasst werden können. Dies ist bei vielen Anwendungen notwendig, bei denen aus komplexen Bildinhalten eine Reaktion abgeleitet werden soll. Einige Beispiele dazu werden weiter unten gegeben.

Zunächst erfolgt eine Bestandsaufnahme der Informationen, die mit den bisher diskutierten Verfahren aus der Originalszene abgeleitet wurden. Aus den Originaldaten wurden, nach einer eventuellen Korrektur systematischer oder statistischer Fehler (z.B. Kapitel 17 bis 22), weitere Merkmale berechnet (Kapitel 24 bis 30). Mit Verfahren zur Segmentierung (Kapitel 31 bis 33) wurden Bildpunkte (Merkmalsvektoren), die einander ähnlich sind, d.h. die im N-dimensionalen Merkmalsraum benachbart liegen, zu Segmenten zusammengefasst. Hier spielt sich auch der Übergang von der bildpunktorientierten Bildverarbeitung zur listenorientierten Bildverarbeitung ab (Beispiele dazu in Kapitel 34). Im nächsten Schritt werden zu den Segmenten Parameter berechnet, die sie näher beschreiben. Diese Verarbeitungsstufe wurde in den Kapiteln 35 bis 37 eingehend behandelt.

Häufig ist diese Verarbeitungsstufe aber aus der Sicht der digitalen Bildverarbeitung und Mustererkennung nicht die letzte Stufe. Vielmehr müssen jetzt einzelne Segmente, die auf Grund bekannter Struktureigenschaften zusammengehören, zu Objekten zusammengefasst werden. Es interessiert nämlich letztlich nicht die Information, ob eine kreisförmige Anordnung von Bildpunkten vorliegt, sondern ob es sich dabei um eine Beilagscheibe oder ein Kugellager handelt.

Im Folgenden werden einige Beispiele gegeben, die die Bandbreite der möglichen Interessenlagen etwas illustrieren sollen.

Bei einem segmentierten Luft- oder Satellitenbild ist eine Zusammenfassung von Segmenten zu Objekten meistens nicht gefordert. Nach der Segmentierung liegt die Information vor, dass unterschiedlich codierte Segmente verschiedene Landnutzungsklassen repräsen-

tieren. Parameter wie z.B. zur Form oder Orientierung der Segmente werden hier nicht benötigt. Allenfalls ist noch die Größe der Segmente einer Klasse von Belang. Die weitere Verarbeitung kann die Bereinigung des segmentierten Bildes sein (z.B. die Zuweisung von sehr kleinen Segmenten zu den sie umgebenden Segmentklassen), die Überführung der Segmente in kompaktere Datenstrukturen und die Integration dieser Daten in Informationssysteme, um sie mit anderen Daten (z.B. politische Grenzen, statistische Erhebungen, digitale Geländemodelle, usw.) verknüpfen zu können.

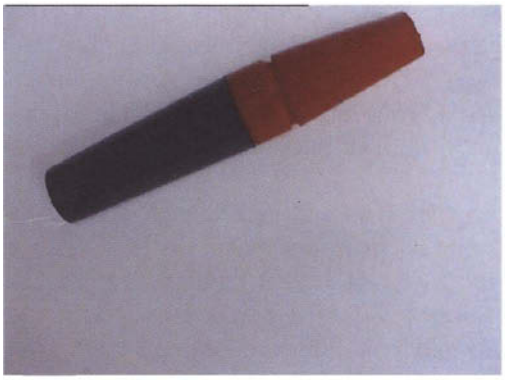
Wenn im Rahmen einer Qualitätskontrolle Oberflächen auf Fehlerstellen oder Störungen untersucht werden sollen, so kann der Segmentierungsschritt das Endergebnis, nämlich die Markierung der fehlerhaften Stelle, liefern. Beispiele dazu wurden in Kapitel 28, Bilder 28.18, 28.17 und 28.22 und in Kapitel 29, Bildfolge 29.9 gegeben. Die Segmentierung besteht hier aus einer Einteilung in die Klassen „fehlerhafter Bereich" und „nicht fehlerhafter Bereich". Meistens genügt hier eine Positionsangabe (z.B. Schwerpunktskoordinaten). Manchmal wird noch eine Auswahl der markierten Bereiche getroffen, etwa dadurch, dass alle Fehlerstellen am Rand ignoriert werden oder dass nur Störungen innerhalb einer festgelegten *area-of-interest* akzeptiert werden. Weitere Beispiele zu Anwendungen aus diesem Bereich sind die Inspektion von lackierten Oberflächen auf Unebenheiten und Kratzer oder die Untersuchung von Pressteilen an sicherheitsrelevanten Stellen auf Einschnürungen oder Risse.

Bei einer anderen Gruppe von Anwendungen wird segmentiert, und danach werden die Segmente (oder das Segment, da hier häufig nur ein relevantes Segment vorliegt) näher untersucht. Beispiele dazu wurden in der Einleitung zu Kapitel 23 (Bildfolge 23.1, Kleberaupen) gegeben. Bei der Anwendung in Bild 23.1-a und 23.1-b ist zu untersuchen, ob die Kleberaupe unterbrochen ist. Das Segmentierungsergebnis ist hier nur in der festgelegten *area-of-interest* relevant. Falls in diesem Bereich, nach der Elimination von Segmenten mit weniger als c_1 Pixel, nur ein Segment übrig bleibt, so kann daraus die Schlussfolgerung abgeleitet werden, dass die Kleberaube vorhanden und nicht unterbrochen ist. Falls z.B. zwei Segmente auftreten, ist noch zu prüfen, ob die Unterbrechung größer als c_2 Pixel ist und somit außerhalb der Toleranz liegt. Auf eine Untersuchung, ob die Orientierung der Segmente zusammenpasst, kann hier verzichtet werden, da nur die Segmente innerhalb der *area-of-interest* verarbeitet werden und stillschweigend die Annahme gemacht wird, dass segmentierte Bereiche automatisch zur Kleberaupe gehören.

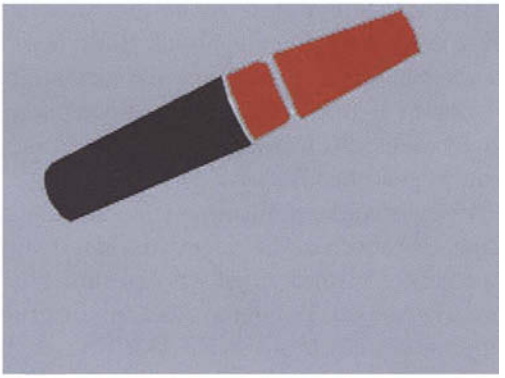
Etwas anders ist die Situation bei der Kleberaupenvermessung in den Bildbeispielen 23.1-c und 23.1-d. Hier dient die Segmentierung nur zur Ermittlung des Bildbereichs, in dem das Lichtband verläuft. Die eigentliche Auswertung, die die gewünschte Information liefert, ob die Kleberaupe die richtige Form hat, erfolgt erst anschließend. Dazu wird der segmentierte Bereich z.B. skelettiert und die Skelettlinie analysiert, um die markanten Punkte zu erhalten. Aus der relativen oder absoluten Lage dieser Punkte zueinander kann dann die Entscheidung abgeleitet werden.

Ähnlich sind Anwendungen, bei denen das Ergebnis der Segmentierung als Maske verwendet wird, um Bildbereiche festzulegen, in denen eine weitere Verarbeitung stattzufinden hat. Ein Beispiel hierzu wurde in Kapitel 20 , Bildfolge 20.34, gegeben. Hier wird durch die Segmentierung ein Bereich markiert, in dem eine Beschriftung vorliegt. In weite-

(a)



(b)



(c)

Bild 39.1: (a) Rotauszug des Originalbilds: Das Objekt „Filzstift" besteht aus den drei Komponenten „Griff" (schwarz), „Halterung" (rot) und „Verschlusskappe" (rot). (b) Vorverarbeitung: Zwei Erosionen, gefolgt von zwei Dilatationen (*opening*), um die Reflexionen zu entfernen. (c) Segmentiertes Bild. Zur Segmentierung wurden alle drei Farbkanäle verwendet. Es wurden die Klassen „schwarze Bildpunkte" (Griff), „rote Bildpunkte" (Halterung und Verschlusskappe) und „sehr helle Bildpunkte"(Hintergrund) klassifiziert.

ren Verarbeitungsschritten werden nur mehr die Bildpunkte in diesem markierten Bereich untersucht, um z.B. die Schriftzeichen zu erkennen und dadurch die Bezeichnung eines Teils zu erfassen. Dieser zweite Verarbeitungsschritt ist vom ersten Segmentierungsschritt weitgehend unabhängig. Er wird aber wieder aus einer Elimination von Störungen, einer Berechnung von Merkmalen und einer Segmentierung (möglicherweise mit ganz anderen Verfahren als im ersten Schritt) bestehen.

Komplexe Objektstrukturen können nach der Segmentierung in mehrere Segmente zergliedert sein. Es ist jetzt die Aufgabe eines weiteren Verarbeitungsschrittes, die zu einem Objekt gehörigen Segmente zu finden, zu untersuchen, ob sie „zusammenpassen" und letzt-

lich zu entscheiden, ob es sich um ein Objekt eines bestimmten Typs handelt. Ein einfaches Beispiel hierzu ist in der Bildfolge 39.1 zusammengestellt. Das Objekt ist hier ein Filzstift, der sich aus einem schwarzen Griff (links unten), einer Halterung für den Faserstift (in der Mitte) und der Verschlusskappe (rechts oben) zusammensetzt. Nach einer Vorverarbeitung, die hier aus der Elimination von Reflexionen bestehen könnte (z.B. zwei Erosionen gefolgt von zwei Dilatationen, *opening*), werden zur Segmentierung die Farbmerkmale verwendet: Griff - Schwarz, Halterung und Verschlusskappe - Rot, Hintergrund - Hellgrau. Die drei für die Segmentierung maßgeblichen Klassen (die Klasse der schwarzen und der roten Bildpunkte des Objekts und die Klasse der sehr hellen Bildpunkte des Hintergrundes) verteilt sich im Segmentierungsergebnis auf die vier Segmente mit den Bedeutungen „Hintergrund", „Griff", „Halterung" und „Verschlusskappe".

Aber erst eine genauere Untersuchung der Objektsegmente liefert die Information, ob es sich dabei tatsächlich um die entsprechende Struktur handelt: Man muss z.B. prüfen, ob die Flächeninhalte innerhalb der Toleranz liegen und ob die Formen stimmen. Wenn diese Untersuchungen ein positives Ergebnis liefern, kann versucht werden, die Segmente zusammenzufassen. Dazu wird man prüfen, ob die Lage der Schwerpunkte zusammenpasst, was hier bedeutet, dass sie auf einer Geraden liegen und dass die Abstände innerhalb gewisser Toleranzen liegen. Wenn dies der Fall ist, kann man entscheiden, dass es sich um einen Filzstift des Typs XYZ handelt.

Diese Beispiele haben gezeigt, dass die nach einer Segmentierung folgenden Schritte einfach, aber auch sehr komplex sein können. In den weiteren Abschnitten werden nun einige Vorgehensweisen für diese Verarbeitungsschritte diskutiert.

39.2 Einfache, heuristische Vorgehensweise

Im vorhergehenden Abschnitt ist bereits angeklungen, dass bei vielen Implementierungen heuristisch, pragmatisch vorgegangen wird. Wenn z.B. auf einer texturierten Oberfläche Störungen der Textur gefunden werden sollen, so könnte die Vorgehensweise wie folgt aussehen:

- Segmentierung nach Maßgabe bestimmter Texturmerkmale.
- Alle Segmente, die kleiner als c_1 Pixel sind, werden ignoriert.
- Alle Segmente, die außerhalb einer festgelegten *area-of-interest* liegen, werden ignoriert.
- Alle Segmente, die vom Rand der *area-of-interest* berührt werden, werden ignoriert.
- Aufgrund von apriori-Wissen sei bekannt, dass die Störungen immer länglich ausgeprägt sind. Deshalb werden Segmente, bei denen das Verhältnis der Eigenwerte der Kovarianzmatrix kleiner als c_2 ist, nicht berücksichtigt.

- Falls die Orientierung des Segments mit der (bekannten) Texturhauptrichtung übereinstimmt, wird das Segment als Markierung einer Texturstörung akzeptiert.

Man sieht an diesem Beispiel, dass eine sequentielle Vorgehensweise gewählt wurde: Es wird versucht, durch apriori-Wissen Schritt für Schritt Segmente auszuschließen. Dabei wird man, um Rechenzeit zu sparen, die Berechnung nur auf benötigte Segmentparameter beschränken. Wenn sich die abgebildeten Objekte bereits durch den Flächeninhalt unterscheiden, so kann man auf aufwändige Verfahren, wie z.B. das Strahlenverfahren (Kapitel 36) oder eine Segmentbeschreibung mit neuronalen Netzen (Kapitel 37) verzichten.

Wenn die interessierenden Objekte nach der Segmentierung in mehrere Segmente zergliedert sind, so muss anhand von apriori-Wissen geprüft werden, welche Segmente zusammengehören. Dies sei am Beispiel des Objekts „Filzstift" von Bild 39.1 erläutert. Dieses Objekt setzt sich aus drei Segmenten zusammen:

- Das erste Segment ist der „Griff". Es gehört zur Klasse der schwarzen Bildpunkte. Der Flächeninhalt dieses Segments muss bei etwa 13500 Bildpunkten liegen. Der Parameter für die Länglichkeit (λ_1/λ_2) muss bei etwa 10 liegen.

- Das zweite Segment ist die „Halterung". Es gehört zur Klasse der roten Bildpunkte. Flächeninhalt: Etwa 3600 Bildpunkte. Das Segment ist nicht ausgeprägt länglich $(\lambda_1/\lambda_2 = 2)$.

- Das dritte Segment ist die „Verschlusskappe". Dieses Segment gehört ebenfalls zur Klasse der roten Bildpunkte. Flächeninhalt: Etwa 9000 Bildpunkte. Das Segment ist ausgeprägt länglich $(\lambda_1/\lambda_2 = 5)$.

Auf eine genaue Untersuchung der Form der Segmente kann in diesem Beispiel verzichtet werden, da die Flächeninhalte, das Länglichkeitsmerkmal und die Klassenzugehörigkeit die Unterscheidung gewährleisten. Falls das Problem größeninvariant behandelt werden soll, muss eine Flächennormierung, etwa über eine Division aller Segmentflächen durch die größte Segmentfläche, erfolgen. Dann treten an die Stelle der absoluten Flächeninhalte die Verhältnisse der Flächeninhalte der Segmente.

Wenn nun drei Segmente extrahiert wurden, auf die die obigen Beschreibungen zutreffen, so kann man daraus noch nicht schließen, dass diese drei Segmente zusammengesetzt das abgebildete Objekt „Filzstift" ergeben. Um zu diesem Schluss zu kommen, muss man in diesem Beispiel weiter fordern:

- Die drei Schwerpunkte der Segmente müssen näherungsweise auf einer Geraden liegen.

- Die Orientierungen der Segmente müssen übereinstimmen. Das ist bei nicht länglichen Segmenten oft schwer zu bestimmen. Im vorliegenden Beispiel könnte man die Tatsache verwenden, dass das Segment „Halterung" rechteckig ist und die längere Rechteckseite senkrecht auf der Orientierungsachse des Objekts stehen muss.

- Die Reihenfolge der Segmente entlang der Orientierungsachse muss stimmen: Ausgehend vom Segment „Griff" muss das Segment „Halterung" folgen und dann das Segment „Verschlusskappe".

- Die Abstände der Segmentschwerpunkte müssen innerhalb bestimmter Toleranzen liegen.

Diese Forderungen werden in der Bildfolge 39.2 verdeutlicht.

Es ist offensichtlich, dass selbst bei diesem sehr einfachen Beispiel vielfältige Überprüfungen durchgeführt werden müssen, um das gewünschte Ergebnis ableiten zu können. Ungleich schwieriger wird es natürlich, wenn die Objekte die Struktur von komplexen Bauteilen, Fahrzeugen oder Personen haben. Hier muss im jeweiligen Anwendungsfall eine besondere Strategie festgelegt werden. Sehr wichtig ist hier die sorgfältige Zerlegung des komplexen Objekts in einfache Bestandteile, die mit den Methoden der Segmentierung erkannt werden können (siehe Kapitel 31). Bei der Synthese sollte man soweit wie möglich apriori-Wissen in die Entscheidungen mit einbeziehen.

39.3 Strukturelle Verfahren

Die Verfahren des vorhergehenden Abschnitts haben den Nachteil, dass sie zwangsläufig sehr stark auf eine bestimmte Anwendung zugeschnitten sind. Das bedeutet, dass bei Änderungen der Systemkonstellation oder beim Übergang zu einer anderen, ähnlichen Anwendung viele Teile modifiziert werden müssen. Da bei Implementierungen dieser Art das apriori-Wissen und die Auswertungsstrategie fest in den Programmcode eingebunden sind, wird die Umkonfiguration einer Anwendung nur mit einer Umprogrammierung zu lösen sein.

Wünschenswert wäre ein System, das allgemeingültige Inferenzmethoden zur Verfügung stellt, die in den verschiedensten Anwendungsfällen eingesetzt werden können. Ein derartiges System sollte folgende Komponenten enthalten:

- Eine (*Bildverarbeitungs-* und) *Mustererkennungskomponente* (MK), die die Methoden der digitalen Bildverarbeitung und Mustererkennung bereitstellt.

- Eine *statische Wissenbasis* (WB) für das gesamte Wissen, das der Anwendung zugrunde liegt.

- Eine *dynamische Wissensbasis* für das während einer Auswertung anfallende Wissen.

- Eine *Reaktionskomponente* (*Exekutive*, RK), hinter der sich die spezielle Anwendung verbirgt.

- Eine *Verwaltungskomponente* (VK), die die Ablaufsteuerung der Anwendung übernimmt.

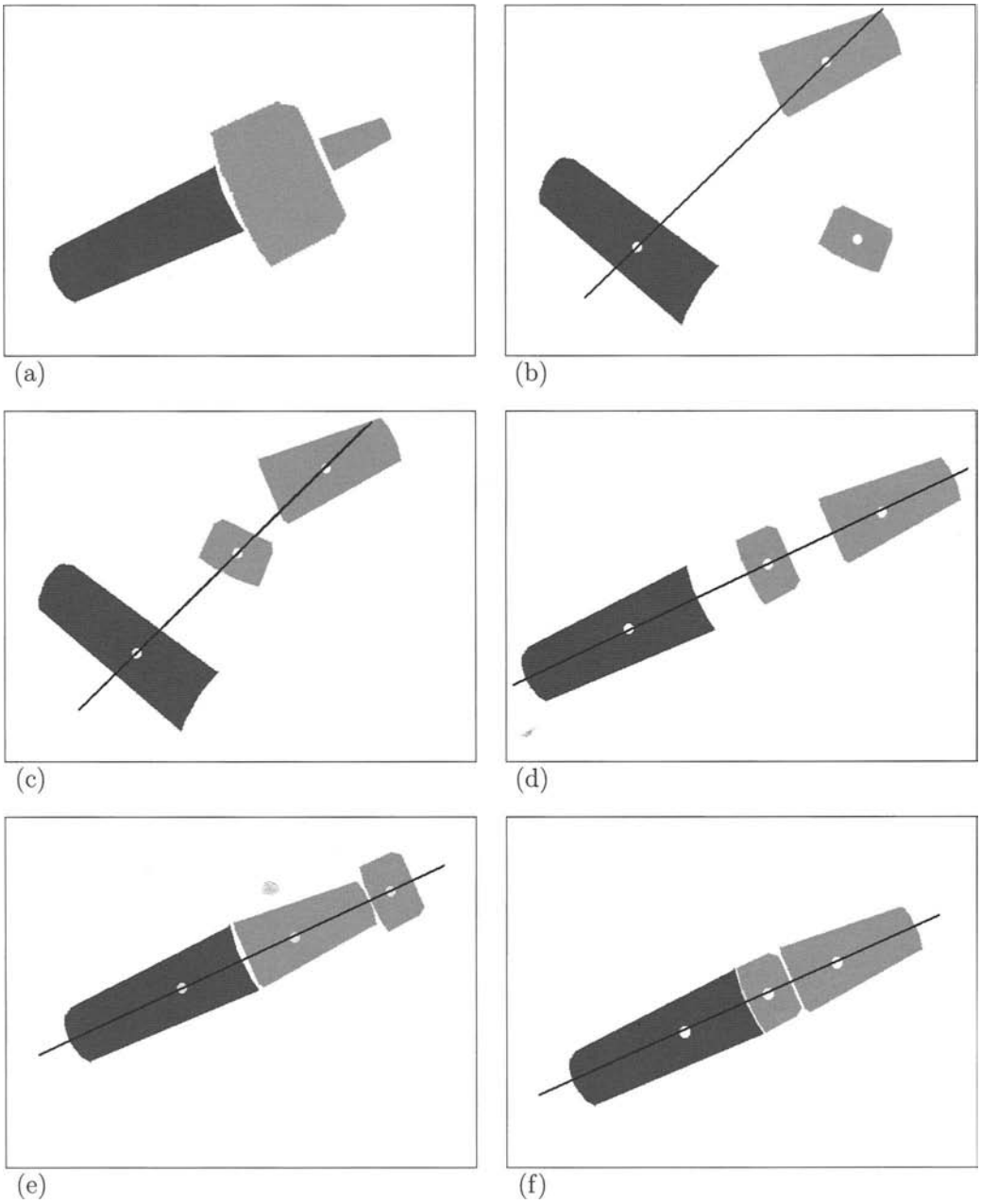


Bild 39.2: Zusammenfassen von Segmenten zu einem Objekt. (a) Bei den drei Segmenten stimmen die Größenverhältnisse nicht. (b) Die Segmentschwerpunkte liegen nicht auf einer Geraden. (c) Die Orientierungen der Segmente passen nicht zusammen. (d) Die Abstände der Segmentschwerpunkte stimmen nicht. (e) Die Reihenfolge der Segmente entlang der Orientierungsachse stimmt nicht. (f) Korrekte Konstellation: Hier kann auf das Objekt „Filzstift“ geschlossen werden.

- Eine *Interaktionskomponente* (IK), durch die der Benutzer mit dem System kommunizieren kann.

- Eine *Dokumentationskomponente* (DK), die die gesamte Beschreibung des Systems enthält.

In den folgenden Abschnitten werden einige dieser Komponenten ausführlicher erläutert.

39.3.1 Die Mustererkennungskomponente

Die Mustererkennungskomponente kann man sich als eine Methodenbasis mit Modulen (Prozeduren, Prozessen) vorstellen, die die Gebiete der digitalen Bildverarbeitung und Mustererkennung so weit wie möglich abdecken. Wenn eine neue Anwendung entworfen wird, so wählt der Designer des Systems aus der Methodenbasis die benötigten Module aus. Bei der Untersuchung der Kleberaupen (Bildbeispiel in Abschnitt 8.2, Bild 8.1-c und 8.1-d und Bemerkung in Abschnitt 15.1) werden etwa folgende Module benötigt:

- Ein Modul zum Einzug des Bildes mit einem Framegrabber.

- Ein Modul zur Festlegung einer *area-of-interest*, in der das Lichtband liegt. Alle weiteren Module arbeiten nur mehr in dieser *area-of-interest*.

- Ein Modul zur Entfernung von Bildstörungen, etwa durch einen bewegten Mittelwert.

- Ein Modul zur Binarisierung: Hintergrund / Lichtband.

- Ein Modul zur Skelettierung des segmentierten Lichtbandes.

- Ein Modul, das die Skelettlinie auswertet, die markanten Punkte ermittelt und die Koordinaten dieser Punkte in geeigneter Weise bereitstellt.

In diesem Beispiel könnte es vorkommen, dass die Methodenbasis kein spezielles Modul zur Auswertung der Skelettlinie enthält. Dann muss dieses Modul neu implementiert werden und steht ab diesem Zeitpunkt in der Methodenbasis auch für spätere Anwendungen zur Verfügung.

Die Module der Methodenbasis sollten so implementiert werden, dass ihre Ein-/Ausgabeschnittstellen, soweit sinnvoll, miteinander kompatibel sind. So sollte z.B. das Binarisierungsmodul direkt die Ergebnisse des Moduls für den bewegten Mittelwert übernehmen können. Anschaulich gesprochen sollte die Methodenbasis so implementiert werden, dass die Module, vergleichbar mit den Steinen eines Dominospiels, zu längeren Verarbeitungsfolgen kombiniert werden können.

Ob die Module als Prozeduren (Unterprogramme) oder als selbständige Prozesse ablaufen können, muss beim Design der Methodenbasis entschieden werden. Auf jeden Fall

ist eine saubere prozedurale Lösung, eventuell in Verbindung mit objektorientierter Programmierung, immer vorteilhaft. Die Implementierung einzelner Module als Prozesse ist dann mit geringem Aufwand zu erreichen.

Bei einer Konstellation, in der einzelne Module oder Kombinationen von Modulen zu Prozessen zusammengefasst werden, ist die Möglichkeit der Parallelisierung von Abläufen gegeben. Dieser Aspekt gewinnt bei komplexen Anwendungen immer mehr an Bedeutung und sollte auf jeden Fall bedacht werden.

39.3.2 Die statische und dynamische Wissensbasis

In diesen Komponenten wird das gesamte Wissen der Anwendung in einer strukturierten Form bereitgestellt. Für das Beispiel des Problems der Erkennung eines Objekts vom Typ „Filzstift" aus dem vorhergehenden Abschnitt könnten das etwa Informationen zu folgenden Fragen sein:

- Welche Module aus der Methodenbasis werden benötigt?
- In welcher Reihenfolge müssen die Module ablaufen?
- Welche äußeren Voraussetzungen erwarten bestimmte Module? Hier könnte z.B. die Position von Beleuchtungseinrichtungen festgelegt werden, die beim Bildeinzug notwendig sind.
- Mit welchen Parameterwerten müssen die Module ablaufen? Für das Klassifizierungsmodul sind das z.B. die Dimension des Merkmalsraums, die Anzahl der Klassen, die trainierte Datenbasis für den Klassifikator, usw.
- Mit welchen Parametern und welchen Toleranzen werden die Segmente beschrieben? In diesem Beispiel könnte das für das Segment „Griff" lauten: „Das Segment muss zur Klasse der schwarzen Bildpunkte gehören. Der Flächeninhalt muss zwischen 13000 *Pixel* und 14000 *Pixel* und die Maßzahl für die Länglichkeit muss zwischen 9.5 und 10.5 liegen."
- Welche Segmentkonstellationen müssen vorliegen, dass auf ein bestimmtes Objekt geschlossen werden kann? Die für das gewählte Beispiel notwendigen Informationen hierzu wurden bereits im Kapitel 23 und in der Bildfolge 39.2 zusammengestellt.
- Welche Reaktion hat zu erfolgen, falls ein bestimmtes Objekt erkannt wird?

Die Antworten auf diese Fragen werden in der statischen Wissensbasis gespeichert. Sie stehen während der gesamten „Lebenszeit" einer Anwendung zur Verfügung. Sollten Änderungen notwendig werden, so ist das meistens durch eine Modifikation der Wissensbasis und einen Neustart des Systems zu erreichen.

Die dynamische Datenbasis nimmt diejenigen Informationen auf, die während eines Ablaufs der Anwendung anfallen. Das könnte z.B. die Information sein, dass nach einer

Segmentierung 25 Segmente gefunden wurden, deren Positionen durch die Segmentschwerpunkte festgehalten wurden. Die Segmente sind jeweils als Listenstrukturen abgelegt, auf die über Pointer zugegriffen werden kann.

Die dynamische Datenbasis ist nur während eines Arbeitsvorgangs der Anwendung gültig. Nach dessen Abschluss wird sie wieder gelöscht.

Zu einzelnen Teilbereichen der hier angeschnittenen Datenbasenproblematik liegen brauchbare Lösungsmöglichkeiten vor. Ohne auf Details einzugehen seien hier erwähnt: Syntaktische Beschreibung, relationale Graphen, semantische Netze oder Expertensysteme.

39.3.3 Die Reaktionskomponente

Die Reaktionskomponente, die auch häufig Exekutive genannt wird, enthält die Module, die für die spezielle Anwendung notwendig sind. Dazu wieder Beispiele: Wenn ein sichtgesteuerter Industrieroboter Bauteile erkennen und manipulieren soll, so wird die Exekutive alle Programme enthalten, die zur Steuerung des Roboters notwendig sind. Wenn es sich dagegen um die Auswertung mikroskopischer Zellabstriche handelt, so sind in der Reaktionskomponente alle Module zusammengefasst, die benötigt werden, um z.B. das Mikroskop und die Ver- und Entsorgung des Mikroskops mit Messpräparaten zu steuern.

Die Exekutive ist immer problemabhängig. Wenn eine neue Anwendung implementiert wird, so werden die Programme für die Exekutive in der Regel neu zu erstellen sein. Die Module der anderen Komponenten sollten aber möglichst unabhängig von der Reaktionskomponente sein.

39.3.4 Die Verwaltungskomponente

Diese Komponente ist für die gesamte Ablaufsteuerung des Systems verantwortlich. Sie besorgt z.B. die Initialisierung des Systems (der einzelnen Komponenten). Auch die Gewährleistung der Betriebssicherheit, die Ablaufsteuerung und die korrekte Beendigung des Systems fällt in den Aufgabenbereich der Verwaltungskomponente. Wenn parallele Prozesse aktiv sind, so werden die Botschaften zwischen den einzelnen Komponenten über die Verwaltungskomponente abgewickelt.

39.3.5 Die Interaktionskomponente

Die Interaktionskomponente ist die Schnittstelle des Systems zum Benutzer. In diesen Bereich fällt die gesamte Benutzeroberfläche, aber auch die Steuerung der möglichen Interaktionen mit dem System. Bei einem Industrieroboter könnte z.B. eine natürlichsprachliche Anweisung lauten: „Bauteile des Typs X sind defekt, wenn sie keine Bohrung oder mehr als zwei Bohrungen haben. Defekte Bauteile sind an der Position Y abzulegen." Die Interaktionskomponente hat die Aufgabe, eine Kommandosprache bereitzustellen, die eingegebenen Befehle geeignet aufzubereiten und sie an die Verwaltungskomponente weiterzuleiten.

Zu Testzwecken und zur Fehlersuche sollte die Interaktionskomponente auch Möglichkeiten bereit stellen, mit denen man in das (laufende) System „hineinschauen" kann.

39.3.6 Die Dokumentationskomponente

Die Dokumentationskomponente enthält die gesamte Beschreibung des Systems sowie die *online*-Hilfe und möglicherweise auch die Entwicklungsdokumentation des Systems.

39.3.7 Ein Beispiel

In Bild 39.3 ist in Form eines einfachen Petri-Netzes ein System zur Robotersteuerung schematisch dargestellt. Es besteht aus Ereignis- und Zustandsknoten. Ein Ereignis kann stattfinden, wenn die für dieses Ereignis notwendigen Zustände eingetreten sind. Wenn ein Zustand eingetreten ist, wird das durch eine Markierung dieses Zustands dargestellt.

Im Beispiel sind die Zustände „IK: Bereit zur Entgegennahme von Aufträgen" und „RK: Roboter bereit zur Initialisierung" Anfangszustände, die nach dem Start des Systems markiert werden. Wenn ein Ereignis stattfindet, werden alle Markierungen der vorausgesetzten Zustände abgezogen und die nachfolgenden Zustände markiert. Ein Ablauf könnte etwa wie folgt aussehen:

- Da das System nach der Startphase zur Entgegennahme von Aufträgen bereit ist, wird an die Interaktionskomponente folgender Auftrag gegeben: „Alle Bauteile vom Typ X mit ungerader Lochanzahl sind defekt".
- Die Interaktionskomponente bereitet diesen Auftrag auf und leitet ihn an die Verwaltungskomponente weiter.
- Die Verwaltungskomponente extrahiert aus dem Auftrag die für die einzelnen Komponenten notwendigen Informationen und benachrichtigt diese Komponenten. Die weiteren Schritte können teilweise parallel ablaufen.
- Die Reaktionskomponente (der Roboter) wird beauftragt, eine bestimmte Grundposition einzunehmen.
- Parallel dazu führt die Mustererkennungskomponente den Bildeinzug durch, macht bestimmte Vorverarbeitungen, segmentiert das Bild und legt die Segmente in einer passenden Datenstruktur ab.
- Parallel dazu ermittelt die Wissenskomponente, anhand der Informationen der Verwaltungskomponente und der Wissensbasis, welche Aktionen durchzuführen sind. Das könnte z.B. sein: Flächenzählung (Bauteil Typ X liegt zwischen ... und ... Bildpunkten), Formerkennung (Bauteil vom Typ X besteht aus nur einem quadratischen Segment), Ermittlung der Lochanzahl (Bauteil vom Typ X muss zwei Bohrungen haben), Ermittlung der Lage (der Greifpunkt von Bauteil Typ X ist in der Position ... und der Orientierung ...), usw.

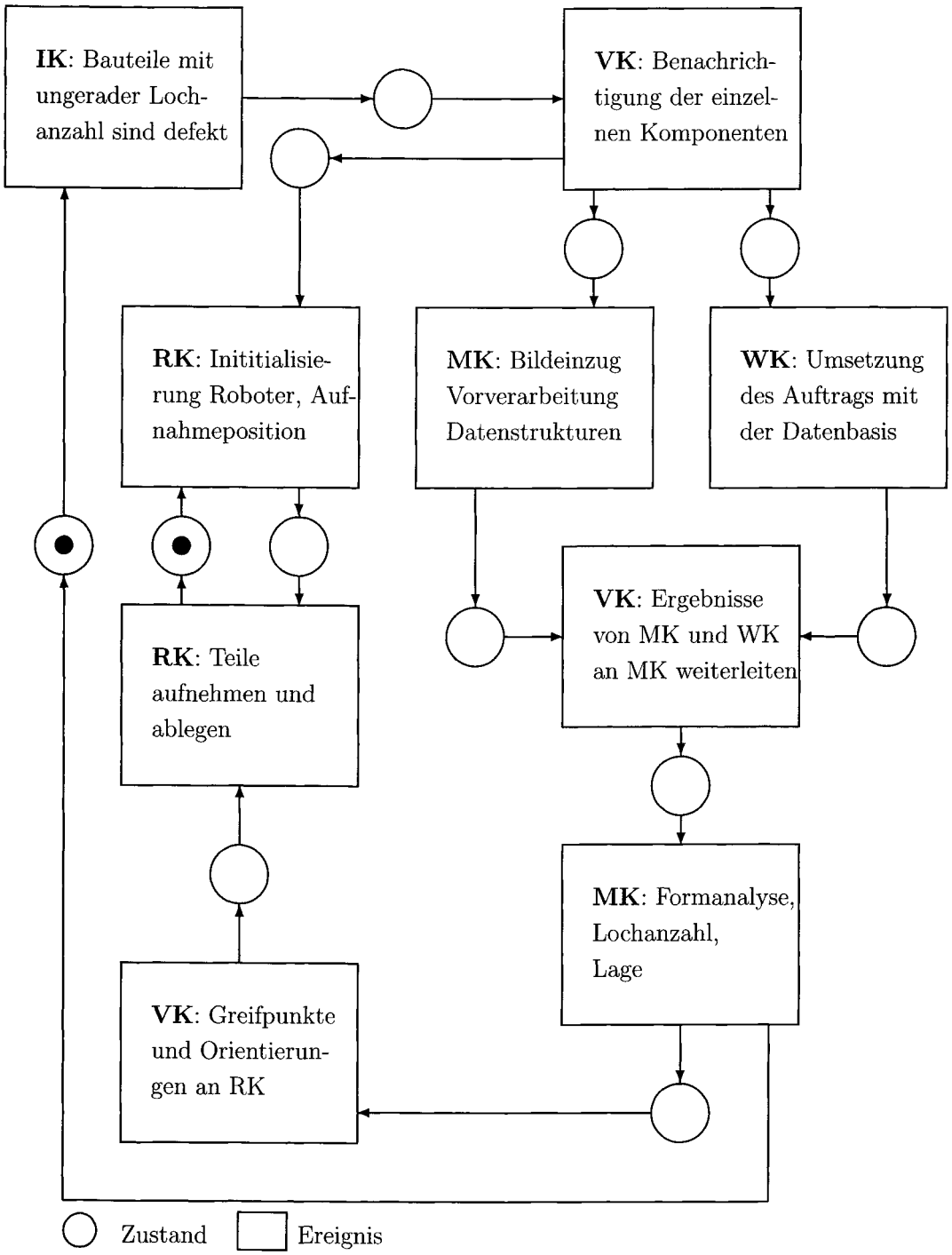


Bild 39.3: Beispiel eines einfachen Systems zum sichtgesteuerten Betrieb eines Roboters. Darstellung als Petri-Netz.

- Die beiden letzten Ereignisse müssen synchronisiert werden, da die weitere Verarbeitung erst fortgesetzt werden kann, wenn beide Ereignisse abgeschlossen sind.
- Nach einer Benachrichtigung durch die Verwaltungskomponente führt die Mustererkennungskomponente die von der Wissenskomponente erhaltenen Überprüfungen durch.
- Nach Abschluss dieses Ereignisses wird über die Verwaltungskomponente die Reaktionskomponente aktiviert, falls diese nach der obigen Initialisierungsphase bereit ist (Synchronisation). Die Reaktionskomponente nimmt nun die gefundenen Bauteile auf, legt sie entsprechend ab und ist dann wieder bereit für den nächsten Auftrag.
- Parallel dazu benachrichtigt die Verwaltungskomponente die Interaktionskomponente, dass das System zur Entgegennahme eines neuen Auftrags bereit ist.

Dieses Beispiel ist hier zwar weitgehend vereinfacht, es zeigt aber doch gut das Zusammenspiel der einzelnen Systemkomponenten.

39.4 Bildverarbeitungssysteme im Einsatz

Abschließend sind in diesem Abschnitt einige Bemerkungen zusammengestellt, die die manchmal leidvollen Erfahrungen bei der Implementierung von Systemen mit bildverarbeitenden Komponenten widerspiegeln.

Bei der Planung des Systems sollte der Auftraggeber soweit wie möglich mit einbezogen werden, um sicherzustellen, dass das gewünschte System entsteht. Oft weiß der Auftraggeber nicht genau, was er will.

Häufig werden derartige Systeme aus Gründen der Rationalisierung geplant. Wenn ein System mehrere Arbeiter ersetzt und vielleicht noch eine bessere Kontinuität liefert, so wird es schwerlich für einige tausend Mark zu realisieren sein. Sollte der Auftraggeber hier nicht einsichtig sein, so verzichtet man besser auf den Auftrag.

Der größere Kostenanteil liegt meistens nicht bei der Hardware, sondern bei der Softwareerstellung. Diese Seite kann etwas geringer gehalten werden, wenn der Auftragnehmer über eine modulare Methodenbasis (siehe Bemerkungen in Abschnitt 39.3.1) verfügt.

Die Erfahrung zeigt, dass der Implementierungsanteil für die Bildverarbeitungs- und Mustererkennungskomponenten oft nicht der umfangreichste Teil des gesamten Systems ist. Viel aufwändiger (und damit teurer) ist z.B. die gesamte Benutzeroberfläche, die zur einfachen Bedienung des Systems geschaffen werden muss. Ein System, das in einer Werkhalle nur von einem promovierten Ingenieur betrieben werden kann, ist wertlos.

Wenn man sich bei der Planung des Systems an einigen Stellen nicht ganz sicher ist, wie sie aus algorithmischer Sicht mit den Methoden der Bildverarbeitung und Mustererkennung zu lösen sind, sollte eine Durchführbarkeitsstudie vereinbart werden. Im Rahmen dieser Studie ist zu prüfen:

- Wie lassen sich die Teilprobleme prinzipiell lösen?

- Welche der gefundenen Verfahren lassen sich im gegebenen Zeitrahmen für die Systemreaktion realisieren?
- Lassen sich einzelne Algorithmen so vereinfachen, dass sie die Aufgabe noch lösen und in den Zeitrahmen passen?
- Sind die gefundenen Verfahren mit dem finanziellen Rahmen verträglich (Kosten der Hardware, Kosten der Implementierung)?

Wenn das System als Labormuster läuft, so ist es noch ein weiter Weg zum operationellen System in der Werkhalle. Das System sollte nach Möglichkeit von einem Team von Fachleuten aus verschiedenen Bereichen erstellt werden. Ein Informatiker, der Spezialist auf dem Sektor der Bildverarbeitung ist, wird in der Regel nicht in der Lage sein, ein System erfolgreich in eine Fertigungslinie zu integrieren.

Das System muss Komponenten zur Selbstkontrolle enthalten. Dazu ein Beispiel: Bei einem in eine Fertigungslinie integrierten System wird im Lauf der Zeit die Linse des Sensors verschmutzen oder die Lichtquellen werden an Leuchtkraft verlieren. Ab einer gewissen Stufe sind dann die Voraussetzungen für die Funktionsfähigkeit nicht mehr gegeben und das System wird fehlerhaft arbeiten. Wenn das System Kontrollen für diesen Fall selbst durchführt, kann rechtzeitig ein Alarm ausgelöst werden.

Da in der Industrie zunehmend flexible Fertigungssysteme zum Einsatz kommen, muss das Inspektionssystem ebenfalls flexibel auf den jeweils zu fertigenden Auftrag reagieren. Es wäre sehr behindernd, wenn für die unterschiedlichen Fertigungsaufträge z.B. die Lichtquellen für die Sensoren mit der Hand neu justiert werden müßten.

Es ist auch zu beachten, dass für das Inspektionssystem ein geeigneter Ort gewählt wird. Dafür kommen vor allem Positionen in Frage, an denen die Teile für gewisse Taktzeiten unbewegt liegen, was den Inspektionsvorgang wesentlich erleichtert. Wenn es möglich ist, sollte man auch darauf achten, dass bei einem Ausfall des Systems nicht die gesamte Produktionslinie zum Stehen kommt. Ein mehrstündiger Stillstand kann sehr teuer sein!

Sachverzeichnis

glAccum, 147
    GL_ACCUM, 148
    GL_ADD, 148
    GL_LOAD, 148
    GL_MULT, 148
    GL_RETURN, 148
glActiveTexture, 257
glAlphaFunc, 137
glBeginQueries, 304
glBegin, 57
    GL_LINES, 58, **61**
    GL_LINE_LOOP, 58, **64**
    GL_LINE_STRIP, 58, **64**
    GL_POINTS, 58, **59**
    GL_POLYGON, **65**
    GL_QUADS, 58, **73**
    GL_QUAD_STRIP, 58, **73**
    GL_TRIANGLES, 58, **70**
    GL_TRIANGLE_FAN, 58, **72**
    GL_TRIANGLE_STRIP, 58, **70**
glBlendColor, 130
glBlendEquation, 127
    GL_FUNC_ADD, 128
    GL_FUNC_REVERSE_SUBTRACT, 128
    GL_FUNC_SUBTRACT, 128
    GL_MAX, 128
    GL_MIN, 128
glBlendFuncSeparate, 129
glBlendFunc, 127, 144
    GL_CONSTANT_ALPHA, 129
    GL_CONSTANT_COLOR, 129
    GL_DST_ALPHA, 129
    GL_DST_COLOR, 129
    GL_ONE_MINUS_CONSTANT_ALPHA, 129
    GL_ONE_MINUS_CONSTANT_COLOR, 129
    GL_ONE_MINUS_DST_ALPHA, 129
    GL_ONE_MINUS_DST_COLOR, 129
    GL_ONE_MINUS_SRC_ALPHA, 129
    GL_ONE_MINUS_SRC_COLOR, 129
    GL_ONE, 129
    GL_SRC_ALPHA_SATURATE, 129
    GL_SRC_ALPHA, 129
    GL_SRC_COLOR, 129
    GL_ZERO, 129
glClearColor, 125
glClearDepth, 113
glClearIndex, 126
glClear
    GL_COLOR_BUFFER_BIT, 113, 125
    GL_DEPTH_BUFFER_BIT, 113, 125
glColor*, 123
glColorMaterial
    GL_COLOR_MATERIAL, 184
glCopyTexImage2D, 228
glCopyTexSubImage2D, 228
glCullFace, 68
    GL_BACK, 68
    GL_FRONT_AND_BACK, 68
    GL_FRONT, 68
glDepthFunc, 113
    GL_ALWAYS, 113
    GL_EQUAL, 113
    GL_GEQUAL, 113
    GL_GREATER, 113
    GL_LEQUAL, 113
    GL_LESS, 113
    GL_NEVER, 113
    GL_NOTEQUAL, 113
glDepthMask, 139
glDepthRange, 113
glDisable, 45
glDrawPixels, 115
glEnable, 45
    GL_ALPHA_TEST, 137
    GL_BLEND, 61, 127, 144
    GL_COLOR_MATERIAL, 184
    GL_CULL_FACE, 69
    GL_DEPTH_TEST, 112
    GL_FOG, 155
    GL_LIGHT0, 181
    GL_LIGHTING, 180
    GL_LINE_SMOOTH, 64, 144
    GL_LINE_STIPPLE, 62
    GL_NORMALIZE, 185
    GL_POINT_SMOOTH, 61, 144
    GL_POLYGON_SMOOTH, 144

GL_POLYGON_STIPPLE, 67
GL_RESCALE_NORMALS, 185
GL_TEXTURE_1D, 253
GL_TEXTURE_2D, 253
GL_TEXTURE_3D, 253
GL_TEXTURE_CUBE_MAP, 253
GL_TEXTURE_GEN_Q, 254
GL_TEXTURE_GEN_R, 254
GL_TEXTURE_GEN_S, 254
GL_TEXTURE_GEN_T, 254
glEndQueries, 304
glEnd, 57
glFog*, 154
GL_FOG_COLOR, 154
GL_FOG_COORDINATE _SOURCE, 154
GL_FOG_COORDINATE, 154
GL_FRAGMENT_DEPTH, 154
GL_FOG_DENSITY, 154
GL_FOG_END, 154
GL_FOG_MODE, 154
GL_EXP2, 154
GL_EXP, 154
GL_LINEAR, 154
GL_FOG_START, 154
glFogCoord, 157
glFrontFace, 68
GL_CCW, 68
GL_CW, 68
glFrustum 97
glGenQueries, 304
glGenTexture, 254
glGetIntegerv
GL_MAX_TEXTURE_UNITS, 257
glGetString
GL_EXTENSIONS, 257
glHint, 217
glIndex*, 126
glLight*, 179
GL_AMBIENT, 180
GL_CONSTANT_ATTENUATION, 189
GL_DIFFUSE, 180
GL_LINEAR_ATTENUATION, 189
GL_POSITION, 180
GL_QUADRATIC_ATTENUATION, 189
GL_SPECULAR, 180
GL_SPOT_CUTOFF, 189
GL_SPOT_DIRECTION, 189
GL_SPOT_EXPONENT, 189
glLightModel*, 186
GL_LIGHT_MODEL_AMBIENT, 187
GL_LIGHT_MODEL_COLOR_CONTROL, 187
GL_LIGHT_MODEL_LOCAL_VIEWER, 187
GL_LIGHT_MODEL_TWO_SIDE, 187
glLineStipple, 62
glLineWidth, 62
glLoadIdentity, 85
glLoadMatrix*, 85
glMaterial*, 182
GL_AMBIENT_AND_DIFFUSE, 182
GL_AMBIENT, 182
GL_BACK, 182
GL_DIFFUSE, 182
GL_EMISSION, 182
GL_FRONT_AND_BACK, 182
GL_FRONT, 182
GL_SHININESS, 182
GL_SPECULAR, 182
glMatrixMode, 104
GL_MODELVIEW, 104
GL_PROJECTION, 104
glMultMatrix*, 85
glMultiTexCoord*, 257
glNormal3*, 185
glOrtho, 95
glPixelStorei, 224
glPointSize, 59
glPolygonMode, 69
GL_BACK, 69
GL_FILL, 69
GL_FRONT_AND_BACK, 69
GL_FRONT, 69
GL_LINE, 69
GL_POINT, 69
glPolygonStipple, 67
glPopMatrix, 104, 180
glPushMatrix, 104, 180
glReadPixels, 115
glRotate*, 87
glScale*, 89
glShadeModel
GL_FLAT, 192
GL_SMOOTH, 197
glTexCoord*, 243
glTexEnv*, 239

GL_BLEND, 240
GL_COMBINE, 241
GL_DECAL, 240
GL_MODULATE, 240
GL_REPLACE, 239
GL_TEXTURE_ENV_COLOR, 239
GL_TEXTURE_ENV, 239
glTexGen*, 249
GL_EYE_LINEAR, 249
GL_NORMAL_MAP, 250
GL_OBJECT_LINEAR, 249
GL_Q, 249
GL_REFLECTION_MAP, 250
GL_R, 249
GL_SPHERE_MAP, 250
GL_S, 249
GL_TEXTURE_EYE_PLANE, 249
GL_TEXTURE_GEN_MODE, 249
GL_TEXTURE_OBJECT_PLANE, 249
GL_T, 249
glTexImage2D, 220
glTexParameter*, 230
GL_CLAMP_TO_BORDER, 238
GL_CLAMP_TO_EDGE, 238
GL_CLAMP, 238
GL_GENERATE_MIPMAP, 233
GL_LINEAR_MIPMAP_LINEAR, 235
GL_LINEAR_MIPMAP_NEAREST, 234
GL_LINEAR, 230
GL_MIRRORED_REPEAT, 238
GL_NEAREST_MIPMAP_LINEAR, 234
GL_NEAREST_MIPMAP_NEAREST, 234
GL_NEAREST, 230
GL_REPEAT, 238
GL_TEXTURE_1D, 230
GL_TEXTURE_2D, 230
GL_TEXTURE_3D, 230
GL_TEXTURE_MAG_FILTER, 230
GL_TEXTURE_MAX_ANISOTROPY_EXT, 236
GL_TEXTURE_MIN_FILTER, 230
GL_TEXTURE_WRAP_S, 238
GL_TEXTURE_WRAP_T, 238
glTexSubImage2D, 226
glTranslate*, 86
glVertex*, 56
glViewport, 102
gluBuild2DMipmapLevels, 233
gluBuild2DMipmaps, 233
gluPerspective, 98
gluScaleImage, 224
glutInitDisplayMode
GLUT_ACCUM, 147
GLUT_DEPTH, 112
GLUT_DOUBLE, 282
GLUT_INDEX, 126
GLUT_RGBA, 123
glutMainLoop, 282
glutPostRedisplay, 283
glutSetColor, 126
glutSwapBuffers, 282
glutTimerFunc, 282
3D-Modellierung, 79
3D-Rekonstruktion, 817
3D-Visualisierung, 37
4·4-Matrix, 84
4-Nachbarn, 350
8-Nachbarn, 350

Absorptionsgesetz, 151
Abstand zwischen zwei Bildpunkten, 351
Abtast-Artefakte, 141
Abtastfrequenz, 141, 346
Abtastung, 230, 344
Abtastwerte, 285, 286, 346
Accumulation Buffer, 146
Adaline, 725
additives Farbmodell, 132, 365
additive Verknüpfung von Kanälen, 585
Ähnlichkeitsdimension, 667
Äquidensiten, 400, 406, 412
1. Ordnung, 406
2. Ordnung, 406
affine Abbildungen, 551
affine Transformation der Ortskoordinaten, 551
Akkumulator, **506**, 513, 515
Aktivierungsfunktion, **724**, 726, 730
Alignment, 813
Alpha-Test, 137
Alpha-Wert, 119, 144, 313
Alpha Blending, 119, 313
Animation, 279, 298
Artikulation, 286
Morphing, 288

Partikelsysteme, 290
Pfad-, 284
Schwärme, 290
Techniken, 283
Anisotropiekoeffizient, 396
Ansichten eines Segments, 807
Anti-Aliasing, 140
Flächenabtastung, 144
Hardware, 149
Multi-Sample, 149
Post-Filterungs-Methode, 144
Pre-Filterungs-Methode, 144
stochastisches Multi-Sample, 149
Subpixel, 149
Zeitliches, 149
Applikations-Prozess (App), 319
Approximation im quadratischen Mittel, 528, 531
Approximationsfunktion, 528
Architecture Review Board (ARB), 41
Artikulation, 286
atmosphärische Effekte, 151, 171
Augenpunkt
-Vektor, 186
lokaler, 186
unendlich entfernt, 186
Augmented Reality, 38
Ausdrucken der Grauwerte, 381
Ausgabe von Grauwertbildern, 399
Ausgabe von logischen Bildern, 383
Ausgangsaktivierung, 724, 730
Ausgleichsgerade, 675
Ausgleichsrechnung, 555
Autokorrelation, **379**, 471, 618
Autokovarianz, 379

back buffer, 280
Backface-Culling, 67, 307
Backpropagation, **732**, 793, 807
Bandpassfiltern, 534
Bandsperre, 535
Basisfunktionen, 528
Baumstrukturen, 681
Bayes'scher Klassifikator, 715
Bedeckungswert (*coverage*), 144
Begriffe aus der Signaltheorie, 622
Behandlung des Randbereichs, 440
Beleuchtung, 159
ambiente Komponente, 172
diffuse Komponente, 172
emissive Komponente, 172
Formel, 160, 179
Globales Modell, 169
Lambert'sches Modell, 165
Lokales Modell, 169
Modell, 160, 215
Phong'sches Modell, 172, 176
spekulare Komponente, 172
Standard-Modell in OpenGL, 166, 171
vollständige Formel, 189
zweiseitig, 186
Beleuchtungsrechnung, 165, 277, 290
in Objekt-Koordinaten, 213
in Welt-Koordinaten, 213
Berechnung von Merkmalen, 342
Berechnung einer neuen Grauwertmenge, 432
Beschleunigungsverfahren, 293
Beschreibung von Segmenten, 343
betragsunabhängige Kantendetektion, 486
bewegter Mittelwert, 146, 231, 442
Bewegungsschätzer, 602
Bezier–Flächen, 52
Bezugspunkt, 458, 514
Bidirectional Reflectance Distribution Function (BRDF), 166
Bidirectional Transmission Distribution Function (BTDF), 168
Bild
als Funktion zweier diskreter Variablen, 376
als Funktion zweier reeller Variablen, 375
als Zufallsprozess, 376
Bildakkumulation, 440, 456
Bildbasiertes Rendering, 19
Bilddatencodierung, 14, 543, 818
Bilddatenkompression, 17, 389, 681
Bilddatenreduktion, 389, 681
Bildfolge, **373**, 456, 526
3D-Rekonstruktion aus einer, 817
Akkumulation, 598
Differenz, 598
Bildgenerierrate, 28, 279, 325
bildliche Reproduktion von digitalisierten Bildern, 380, 399

Bildmatrix, 219, 348, 351, 352
Bildpunkt, 348, 351
bildpunktbezogene Merkmale, 343
Bildschirm-Refresh-Rate, 29
Bildsegmentierung
    kantenorientiert, 472
    linienorientiert, 472
Bildspalte, 348, 351
Bildspeicher, 30, 48, 112, 121, 280
Bildverbesserung, 341
Bildzeile, 348, 351
bilineare Interpolation, 550
Billboard, 23, 136, 294, **315**
    animierte Texturen auf, 17, 319
    axiale, 317
    blickpunktabhängiges, 21, 317
bimodales Histogramm, 393, 413
Bimodalitätsprüfung, 415
Binärbild, 350, 490
Binärbildausgabe, 385
Binarisierung, 400, **412**, 474, 492, 562
    mit dynamischem Schwellwert, 414
    mit fixem Schwellwert, 412
Binarisierung eines Grauwertbildes, 412
Binomialfilter, 634, 648
Bitebene, 432
bounding box, 297
bounding sphere, 297
bounding volume, 297
bounding volume hierarchy, 297
Brechung, 161, 269
    Gesetz, 164
Bump Map, 216, 270
Busbandbreite, 325
Busbandbreiten-Limitierung, 329

Canny-Kantendetektor, 486
chain-Codierung, 782
Chaostheorie, 741
Charakterisierung digitalisierter Bilder, 389
charakteristischer Skalenparameter, 670
Chroma, 366
CIE-Farbdreieck, 360
CIE-Normfarben, 363
Clipping, 82, **300**
clipping planes, 46, 53, 94, 96
closing, 420, **461**, 492
cluster, 579, 696
    Algorithmus, 583
Clusterverfahren, 702
CMY-Farbmodell, 365
co-occurrence-Matrix, **396**, 471, 618, 650
color buffer, 110
Computergrafik, 1
    Fotorealistische, 215
Constructive Solid Geometry (CSG), 54
CPU-Limitierung, 327
Cube Map, 266
Cull-Prozess (Cull), 320
Culling, 294, 297, **299**
    Backface, 307
    Detail, 309
    Occlusion, 303
    Portal, 308
    Viewing Frustum, 301

database traversal, 299
Datenkompression, 388
Datenreduktion, 388
depth complexity, 325
Deskriptor, 526
Detektion kollinearer Bildpunkte, 506
diagonale Nachbarn, 350
Dichte, 376
differentielle Ermittelung von
        Verschiebungsvektoren, 603
Differenzbildung von Kanälen, 259, 587
Differenzenoperator, 272, **442**, 474, 475, 485,
        504
digitale Bildverarbeitung, 2
digitale Filterung im Ortsfrequenzbereich,
        534
Digitalisierung, 342, 344
    von Farbbildern, 357
    von Grautonbildern, 352
    von Schwarz/Weiß-Bilddaten, 348
Digitalkamera, 344
Dilatation, 419, 420, 457, **459**, 460, 466, 490,
        672
    für Fuzzy-Mengen, 754
Dimensionierung, 794
Direct Memory Access (DMA), 329
diskrete, zweidimensionale
        Cosinustransformation, 543

diskrete Verteilung, 376
diskrete zweidimensionale Fouriertransformation, 532
Dispersion, 161, 269
Display Listen, 46, 301
Distanzmaß, 351
Dither-Verfahren, 387
dithering, 582
Dokumentationskomponente, 847
Doppelbilder, 28
double buffer, 121, 279
Draw-Prozess (Draw), 320
Durchschnitt für fuzzy-Mengen, 755
dynamische Wissensbasis, 845

Ebnen der Grauwerte, **421**, 426
Echtfarbendarstellung, 373
Echtzeit, 160, 165, 216, 262, 290, 293
  Harte, 13, 28
  Weiche, 13, 31
Echtzeit-3D-Computergrafik, 27
Echtzeit-Anforderung, 13, 28
Echtzeit 3D-Computergrafik, 293
Eigenvektoren, 594
Eigenwerte, 594, 780, 795
Eingabematrix, 820
Eingabeschicht, 730
Eingangsaktivierung, 724
Einheitlichkeitsprädikat, 561
Einheitsvektor, **184**
Einstimmigkeitsverfahren, 729
Elektromagnetische Wellen, 160
Elimination gestörter Bildzeilen, 440
Elimination gestörter Bildpunkte, 440
Energiehierarchie, 656
Entfernungsmessung, 118
Entropie, 394, 618
Environment Map, 216, 262
Erkennungsphase, 794
Ermittelung von Verschiebungsvektoren mit Blockmatching, 605
Erosion, 419, 420, 457, **459**, 460, 466, 490, 672
Erwartungswert, 377, 379, 821
euklidische Distanz, 351, 703
Euler'sche Charakteristik, **499**, 790
Euler'scher Polygonsatz, 499

Exekutive, 344, **845**
EXPAND-Operator, **625**, 642, 653
exponentielle Skalierung, 419
Extraktion des Randes von Segmenten, 490

Färbung, 358, 366
Falschfarbencodierung, 587
Falschfarbendarstellung, 372
Faltungen, 670
Faltungsmaske, 440
Faltungssatz, 536
Farb-Index–Modus, 122
Farbauflösung, 121
Farbauszüge angleichen, 568
Farbbild, **357**, 368, 567
Farbe, 119, 159, 215
  Fragment-, 121
  Indexbilder, 574
  Indexbilder, 122
  Korrektur der Farbauszüge, 423
  normativer, technischer Aspekt, 360
  physikalischer Aspekt, 358
  physiologischer Aspekter, 358
  Pixel-, 121
  Reduktion der Farben, 573
  Reduktion durch Division, 573
  Separate spekulare, 190
  Textur-, 121
  Vertex-, 121, 172
  vier Komponenten (R,G,B,A), 119
  vierte Komponente A (Transparenz), 119
Farbempfinden, 358
Farbhäufigkeitsverteilung, 576
Farbhistogramm, 580
Farbkante, 470
Farbmerkmal, 490
Farbmischung, 119, **127**, 154, 171, 313
  3D-Probleme, 137
  Additive, 132
  Farbfilter, 134
  Farbmisch-Faktoren, 135
  Klassische, 130, 145
  Mehrfach-, 132
  Primitive, 130
  Subtraktive, 133
Farbmodelle, 357
Farbskala, 362

Farbspeicher, 110
far clipping plane, 96
Filterkern(-maske), 232, **440**, 474, 638
    normiert, 633
    symmetrisch, 633
Filteroperationen im Ortsbereich, 440
Flächeninhalt, 777
Flächenschwerpunkt, 777
Flaschenhals, 324
Flat-Shading, 192
Flicker-Problem, 280
Fotopigmente, 358, 360
Fourier-Koeffizienten, 650
Fourier-Komponenten der Randpunkte, 781
Fourier-Transformation der Randpunkte, 781
Fourierkoeffizienten, 530
Fourierreihe, 530
Fourierspektrum, 534
Fouriertransformation, 532
Fouriertransformierte, 533
Fragment, 160
Fragment-Operationen, 47
Fragment-Programm, 210, 275
fraktale Dimension, 667
fraktale Geometrie, 621, 650, **663**, 666
Frame, 30
Frame-Drop, 30
Framebuffer, 48, 112, 280
frame grabber, 344
Frequenz, 622, 623
Frequenzantwort der REDUCE-Funktion, 639
Fresnel'sche Gleichungen, 164, 269
front buffer, 280
fuzzy-Mengen
    Gleichheit von, 754
    Höhe von, 753
    Komplement von, 755
    Normalisierung von, 753
    Reflexivität, 754
    Symmetrie, 754
    Teilmenge von, 754
    Vereinigung von, 755
fuzzy Klassifikator, 514
fuzzy logic, 584, **749**
    Segmentbeschreibung mit, 792
fuzzy set, 751
Gauß-Filter, 634
Gauß-Filterkerne, 673
Gauß-Pyramide, 231, 305, 620, **623**, 675
Gauß-Tiefpass, 442
Generalisierungseffekt, 741, 807
Generator, 664
Geometrie-Limitierung, 328
Geometrie-Stufe, 320
Geometrische Grundobjekte, 51
Geradenbüschel, 506
Glättungsoperator, 412, **440**, 621, 623, 627
    in Laplace-Pyramiden, 648
Gleichheit von zwei fuzzy-Mengen, 754
gleichmäßige und nicht gleichmäßige Quantisierung, 346
Gradationskurve, 400
Gradient, 272, **451**, 469, 471, 474, 475, 480, 481, 485, 613, 650, 813
Grafik-Primitive, **54**
    Dreiecks-Fächer, 72
    Einzelne Dreiecke, 70
    Einzelne Vierecke, 73
    Linien, 61
    Polygone, 65
    Punkte, 59
    Verbundene Dreiecke, 70
    Verbundene Linien, 64
    Verbundene Vierecke, 73
Grafik-Programmierschnittstelle, 41
    Direct3D, 41
    GKS, 41
    GKS-3D, 41
    OpenGL, 41
    PHIGS, 41
Grafikhardware
    vierte Generation, 205, 216
grassfire, **461**, 656
Grautonbilder, 352
Grauwertematrix, 396
Grauwertkante, 477
Grauwertmanipulation, 400
Grauwertskalierung, 462
    mit einer Hochpassfilterung, 426
Grauwerttransformation, 400
Grauwertübergänge, 442
Grauwertübergangsmatrix, **396**, 618
gray level slicing, 406

Grenzwellenzahl, 622
größeninvariante Segmenterkennung, 776, 793, 804
Größenproportionale Codierung, 386
Grundfarben, 360
Grundtexturfläche, **612**, 672, 677

Hadamartransformation, 543
Häufigkeitsverteilung der Farben, 574
Halbtonverfahren, 385
Halfway-Vektor, 177
Hash-Adresse, 577
Hash-Codierung, 577
Hash-Tabelle, 577, 580
Hauptkomponente, 594
Hauptkomponententransformation, **589**, 740
Hauptrichtung, 795
  eines Segments, 780
Hausdorff-Besicovitch-Dimension, 667
HDC-Operation, 656
Helligkeit, 362, 366
Hesse'sche Normalform, 506
hierarchische diskrete Korrelation, 656
hierarchische Datenstruktur, 296
Histogramm, **392**, 421, 576–579
  bimodal, 393
Histogrammlinearisierung, 421
Hochpassfilterung, 421, 462, 534, 574, 624
Höhe einer fuzzy-Menge, 753
Höhenlinien, 114
Homogene Koordinaten, 56, **83**
homogener Zufallsprozess, 380
horizontale Ausdehnung eines Segments, 784
Houghraum, 512
Houghtransformation, **506**
  erweitert, 514
  standard, 512
  verallgemeinert, 512
HSI-Farbmodell, 366
hue, 358, 366
Huygen'sches Prinzip, 163

image mosaicing, 621, **642**
image sharpening, 427
Impostors, 23, 319
Indexbild, 122, 372, **574**
Inferenzmethoden, 845
Informationsextraktion, 341
Initiator, 664
Intensität, 358, 362, 366
Interaktionskomponente, 847
Interaktive 3D-Computergrafik, 16, **27**, 293
Interpolation mit Polynomen, 552
invariante Merkmale aus Momenten, 786
Inverse Dynamik, 288
inverse Filterung, 542
Inverse Kinematik, 288
inverser Streckungsfaktor, 56, **83**, 99, 180, 217

Jakobi-Matrix, 824

Kalibrierung, 587
Kalibrierung der Grauwerte, 428
Kalibrierungsbild, 416, 430
kalligrafische Lichtpunkte, 294
Kalman-Filter, 17, 18, **811**
  erweitertes (EKF), 824
Kalman-Verstärkungsfaktoren, 821
Kanten, 469
Kantendetektor, 480, 483, 486
Kantendichte, 615
Kantenextraktion, 457, **469**, 490
  parallel, 475
  sequentiell, 521
kantenorientierte Bildsegmentierung, 472
Kanten und Linien
  mit dem Canny-Kantendetektor, 486
  mit morphologischen Operationen, 490
Kantenverstärkung, 504
Key Frame–Technik, 286
Klassencode, 357
Klassifikatoren, 612
Klassifizierung
  Beurteilung der Ergebnisse, 722
Klassifizierungsphase, 794
Klassifizierungsstrategie
  fest dimensioniert überwacht, 700
  fest dimensioniert unüberwacht, 702
  überwacht lernend, 706
  unüberwacht lernend, 706
klassische Aussagenlogik, 750
Klothoide, 285
Knoten, 296

bounding volume, 297
End-, 296
Gruppen-, 296
Hülle, 297
interne, 296
LOD-, 296
Sequence-, 296
Switch-, 296
Transformations-, 296
Wurzel-, 296
Koch'sche Kurve, 665
Kollisionserkennung, 294, 297
kombinatorische Operatoren, 756
kompakte Speicherung von Segmenten, 343, **765**
Kompaktheit, 779
Kompatibilitätsfunktion, 504
kompensatorische Operatoren, 756
komplementäre Farben, 365
Komplement für fuzzy-Mengen, 755
Konstruktive Körpergeometrie, 54
Kontrast, 419, 421
Kontrastanreicherung, 474
Kontur, 799
eines Segments, 782
Konturverfolgungsverfahren, 496
Konvergenz, 741
Konzentration für fuzzy-Mengen, 754
Koordinatensystem, 80
Augenpunkt-, 81
Bildschirm-, 81, 205, 217
Definition des, 80
Euklidisches, 80
gespiegeltes, 90
Normiertes Projektions-, 81
Objekt-, 81, 205, 217
Projektions-, 81, 205
Textur-, 217
Welt-, 81, 118, 180
Korrektur der Farbauszüge, 423
Korrektur des Bildhintergrundes, 417
Korrekturschritt, 814
Korrelation, 814
Korrelationskoeffizient, 378
Korrespondenzproblem, 817
Kovarianz, 377
Kovarianzmatrix, **378**, 595, 780, 795
einer Musterklasse, 716
kubische Faltung, 550
künstliche Intelligenz, 2

längliche Segmente, 781
Laplace-Operator, 427, 474, 615
Laplace-Pyramide, 620, **623**
Rücktransformation, 631
Lauflängencode, 618, 650, **765**
Leistungsnachweis, 12, 31, 36
Lernfaktor, 728
Lernrate, 735, 741
Level-of-Detail, 294, **310**
Continuous, 315
Fade LOD, 313
Morph LOD, 314
Skalierungsfaktor, 332
Switch LOD, 312
Lichtfeld-Rendering, 22
Lichtquelle
Abschwächungsfaktor, 180, 187
ausgedehnte, 169
Eigenschaften, 179
Lokale, 180
punktförmige, 169
Richtungs-, 180
Spotlight, 187
Lichtvektor, 163
Lighting Model, 185
lineare digitale Filter im Ortsbereich, 442
lineare Approximation, 527
lineare Skalierung, 401
Line Loop, 64
Lines, 61
Line Strip, 64
linguistische Variable, 751
Linien, 469
linienorientierte Bildsegmentierung, 472
Linienverfolgung, 521
listenorientierte Verarbeitung, 343, 776
Lösungsmenge für den Bezugspunkt, 515
logarithmische Filterung, 542
logarithmische Skalierung, 419
Logikkalküle, 750
logisches Bild, 354, 466
look-up-Tabelle, **372**, 400, 573
Luminanz, 358, 362, 363

Madaline, 725
Mahalanobis-Abstand, 716
Maler–Algorithmus, 109, 139
markante Punkte, 526
Materialeigenschaften, 182
mathematische Modelle für Bilder, 352, 375
mathematische Morphologie, 457
Matrix
  Einheits-, 94
  Normierungs-, 100
  Projektions-, 95
  Rotations-, 89
  Skalierungs-, 91
  Translations-, 87
  Viewport-, 102
Matrizen Stapel, 103, 298
  Modelview-, 93, 103, 180, 205
  Projektions-, 103, 205
  Textur-, 253
Maximum-Likelihood-Klassifikator, 714
Media Filer, 461
Median-Cut-Verfahren, 579
Medianfilter, 427, **459**, 466, 615
mehrdimensionale Schwellwertverfahren, 416
Mehrheitsverfahren, 728
mehrkanaliges Bild, 370
mehrschichtige Netze, 732
Mengenoperationen für fuzzy-Mengen, 755
Merkmal
  -Extraktion, 813
  aus Bildfolgen, 597
  aus mehrkanaligen Bildern, 585
  Bewegung, 600
  Farbe, 490, 567, **570**
  Grauwert, 567
  invariantes aus Momenten, 786
  Textur, 490, 611
Merkmalsraum, **565**, 611, 696
  N-dimensional, 699
Merkmalsvektor, **699**, 795, 805
Messgleichung, 820
Messmatrix, 820
Messrauschen, 814, 820
min-max-Kompensations-Operator, 756
Minimum-Distance-Cluster-Algorithmus, 583
Minimum-Distance-Klassifikator, 706
MipMap, 231
  Erzeugung, 233
  Level, 233
Mitgliedsgradfunktion, 751
Mittelwert, 379, 389
Mittelwertvektor, 390, 719
mittlere quadratische Abweichung, **390**, 613, 650, 652, 656
mittlerer Grauwert, 389
Modifikation der Grauwerte, 399
Modifikation der Ortskoordinaten, 547
Möglichkeitsverteilung, 757
Moiré-Muster, 141
Momente, 784
  für die Randpunkte, 789
Monitor-Rot/Grün/Blau, 362
monochromatisches Licht, 358, 360
Morphing, 288, 314
  Level-Of-Detail-, 289
Morphologie im Grauwertbild, 465
morphologische Operationen, **457**, 656
  im Grauwertbild, 417
  Kanten und Linien, 490
morphologischer Gradient, 492
mosaicing, 621, **642**
motion-capturing, 288
Motion Blur, 150
MPEG-4, 15, 39, 295, 819
Multifokus, 645
Multipass Rendering, 226, 263
Multiprozessorsysteme, 294, 319
multiresolution image processing, 631
Multischwellwertverfahren, 387
Multispektralbild, 370, 586
Multispektralscanner, 344
Multitexturing, 256
multivariate Klassifikatoren, 583
Muster, **565**
Mustererkennung, 2
Mustererkennungskomponente, 845
Musterklasse, **565**, 699

N-dimensionaler Merkmalsraum, 699
Nachbarn, 350
natural video object, 16
Navigation, 812
  bildbasiert, 813, 832
near clipping plane, 96

Nebel, 151
Boden-, 158
Dichte, 155
Faktor, 155
Farbe, 153
Netz-Gewicht, 724
Netzaktivität, 724, 730
neuronale Netze, 612, **724**, 793
zur Segmentbeschreibung, 804
neuronale Netze als Klassifikatoren, 735
nichtlineare digitale Filter im Ortsbereich, 442
Niveaumenge, 753
nonuniform quantiser, 346
Normalenvektor, 163, **184**, 307
Flächen-, 192
Pixel-, 199
Vertex-, 194
Normalgleichungen, 528, 530
Normalisierung einer fuzzy-Menge, 753
Normalverteilung, 820
Normfarbtafel, 360
Normfarbwerte, 360
normierter Filterkern, 633
NTSC-System, 366
numerische Klassifikation, 696
NURBS, 52

Oberfläche einer Grauwertfunktion, 675
Oberflächenmessung, 117
Objekt, 561, 562
Objekte vom Bildhintergrund abgrenzen, 653
Objektklasse, 565, 699
Objektverfolgung, 812
Occlusion Query, 304
opak, 51
Opazität, 119
OpenGL, 41
ES (Embedded Systems), 42
Extension, 42, 158, 256, 304
Implementierung, 43
Kommando Syntax, 48
Kurzbeschreibung, 43
Library, 44
ML (Media Library), 42
Rendering Pipeline, 45
SL (Shading Language), 42
Utility Library, 44
Utility Toolkit (GLUT), 44, 112
Zustandsautomat, 45
opening, **461**
Operationen im Frequenzbereich, 527
Operationen im Ortsbereich, 438
optical flow, 526
Optik
Geometrische, 160
Physikalische, 160
Orientierung, 779, 795
Ortsauflösung, 121
Ortsfrequenz, 141, 622, 623
Ortskoordinaten, 352

p%-Methode, 414
PAL-System, 30, 366
Palette, 372
parallele Segmentklassifizierung, 776
parallele Kantenextraktion, 475
Partikelsysteme, 290
Partikularisation, 757
Passpunktmethode, 553
Peano-Kurve, 664
Perceptron, 730
Periodendauer, 622
Phong-Shading, 199, 216
picture element, 348
Pixel, 119, 160, 217, 348, 351
Pixel-Operationen, 46
Pixel-Shader, 165
Pixelfüll-Limitierung, 328
Pixelfüll-Rate, 325
pixelorientierte Verarbeitung, 343, 776
plenoptische Funktion, 20
Points, 59
Polygon-Netze, 19, 39, 51, 219
Polygon-Rate, 325
Polygone, **65**
gekrümmte, 52
Konkave, 66
Nicht-planare, 66
Orientierung, 67
planare, 51
Polygonfüllung, 69
Polygonmuster, 67
winding, 67

pop-up–Effekt, 289, 311, 312
Popularity-Verfahren, 579
Positionsrang, **480**, 492
Possibilitätsverteilung, 757
Prädiktionsschritt, 814
Primärfarben, 360
Problemkreis, 565
Produktionsphase, 794, 804
Pseudofarbdarstellung, **372**, 399, 411
Punktefinder, 526
Punktwolken, 696
Purpurfarben, 360

Quad-Strips, 73
Quader-Klassifikator, 719
Quads, 73
quad trees, 681
    Homogenitätskriterium, 685
    Knotenadresse, 682
    Nachbarschaftsalgorithmen, 694
    regionenorientierte Bildsegmentierung, 687
    rekursive Speicherungstechnik, 684
    Stufennummer, 682
Qualitätskontrolle, 621
    von Oberflächen, 663, 677
Quantentheorie, 160
Quantisierung, 20, 219, 278, 344, 348
    Ausgangs-, 120
    Eingangs-, 120
    interne, 120

Radiosity, 28, **170**, 216
Randabschattung, 492, 587
Rand eines Segments, 457, 469
random field, 379
Randpunkt, 795
Rangbild, **481**, 482, 485
Rangordnung, **458**, 480
Rangordnungsoperatoren, 457
Rasterisierung, 47, 217
Rasterisierungs-Stufe, 320
Rasterung, 346, 348
Ratio, 259, 587
Raytracing, 28, **169**, 216, 227, 262
Reaktionskomponente, 344, **845**
recall-Phase, 794, 804
REDUCE-Operator, 305, 625, **632**, 653, 656, 659
Reduktion der Grauwertmenge durch Differenzbildung, 434
Reduktionsfunktion, 625
Referenzbild, 553
Referenzkontur, 514
Referenzpunkt, 770
Referenzsegment, 514
Referenzstruktur, 512
Reflexion, 161
    ambiente, 182
    diffuse, 182
    Gesetz, 163
    ideal diffuse, 165
    ideal spiegelnde, 165
    real spiegelnde, 165
    spekulare, 182
Reflexivität für fuzzy-Mengen, 754
regionenorientierte Bildsegmentierung mit quad trees, 687
Regression, 669
Regressionsgerade, 675, 677
    und fraktale Dimension, 669
reine Farben, 358
Rekonstruktion des höchstwertigen Bit, 434
rektifizierbare Kurven, 666
relative Summenhäufigkeit, **393**, 406, 421
Relaxation, 502
Rendering, **28**, 51
    -Geschwindigkeit, 74
    Volume, 53
Rendering Pipeline, 45, 201, 259, 299, 319
resampling, 549
Restfehlervektor, 557
RGB-Farbmodell, 119, **363**
RGB-Farbraum, **363**, 579, 583
RGBA–Modus, 122, 179
Richtungscodes, 782
Richtungsquadrat, 481
Richtungssystematik, 481
Richtungstoleranz, 481
Rotation, 87, 296
rotationsinvariante Segmenterkennung, 776, 793, 804
Rücktransformation einer Laplace-Pyramide, 631

Rückwärtsschritt, 733
run-length-Code, **765**, 777

Sättigung, 358, 366
saturation, 358, 366
scale space filtering, 621, 650, 663, **669**
Scan Line-Algorithmus, **195**, 199, 217
Scanner, 344
Schachbrettdistanz, 351
Schärfentiefe, 645
Schätzfehler
    -kovarianz, 821
Schätzwert, 811
    a posteriori, 821
    a priori, 820
    optimaler, 812
Schattierung, 159, **191**
    Flat-Shading, 192
    Formwahrnehmung aus, 159
    Phong-Shading, 199
    shape from shading, 159
    Smooth-/Gouraud-Shading, 193
Schattierungsverfahren, 160
Schneeflockenkurve, 665
Schnittebenen, 46, 53, 94
Schwärme, 290
Schwellwertbildung, 474
Schwerpunkt, 795
Segementbeschreibung mit neuronalen Netzen, 844
Segment, 562
    kompakte Speicherung, 343, **765**
    zählen, 457
    zusammenfassen zu Objekten, 840
Segmentauswahl mit morphologischen Operationen, 791
Segmentbeschreibung, 776
    einfache Verfahren, 776
    mit dem Strahlenverfahren, 793
    mit fuzzy logic, 792
    mit neuronalen Netzen, 804
Segmentierung, 17, 343, 412, 457, 474, 611, 696, 840
    durch Binarisierung, 412
    mit fuzzy logic, 749
    p%-Methode, 414
Sehne, 766
selbstähnliche Strukturen, 665
Sensoren, 341
separabler Filterken, 633
sequentielle Kantenextraktion, 521
sequentielle Segmentklassifizierung, 776
Shader
    Fragment-, 201
    Hardware Profile, 204
    Pixel-, 201, 216, 259, 290
    Programmierbare, 200
    Vertex-, 201, 216, 290
    zur Bildverarbeitung, 7
Shading, **191**
Shading-Programmiersprachen, 202
    Cg, 200
    HLSL, 200
    OpenGL SL, 200
Shadow Map, 216, 277
Shannon'sches Abtasttheorem, 141, 232, 346, 354
sichtbares Volumen, 95, 293, 301
Sierpinski-Dreieck, 665
Sigmoidfunktion, 725, 732, 741
Signalcode, 783
Signatur, 343
Simulation
    Ausbildungs-, 28, 32
    closed loop, 11
    Entwicklungs-, 34
    Fahr-, 28, 32
    Flug-, 28, 32
    Hardware-In-The-Loop (HIL), 13, 30, 36
    LKW-, 28
    Mathematisch Digitale, (MDS), 11, 31, 35
    Medizin-, 34
    open loop, 11
    Unterhaltungs-, 31
    von kameragesteuerten Geräten, 10
Simulator
    Flug-, 293
Singulärverfahren, 729
Skaleninvarianz, 666
Skalenparameter, 666, 675
Skalenverhalten, 666
Skalierung, 89, 296
Skalierung der Grauwerte, 400

Skalierungsfunktion, 406, 412, **421**, 423
Skalierungsparameter, 405
Skelett, **492**, 502
Skelettierung, 457, 469, **492**
    mit der Euler'schen Charakteristik, 499
    mit morphologischen Operationen, 492
Smooth-/Gouraud-Shading, 193
Sobeloperator, 272, **450**, 474, 475, 485, 505, 615, 813
Spaltenindex, 351
Speicherung von Segmenten mit run-length-coding, 764
Spektral-Rot/Grün/Blau, 161, 362
spektrale Signalenergie, 656
Spektrum, 161, 624
Spektrum der Ortsfrequenzen, 534
Sphere Map, 263
Spiegelungsexponent, 182
Standard-Houghtransformation, 512
Standardabweichung, 377
statische Wissenbasis, 845
Statistik-Werkzeuge, 294
Stereoprojektion, 161
stetige Verteilung, 376
stochastische Aktivierung, 725
stochastischer Prozess, 379
stochastisches Filter, 812
Strahlenverfahren, 526, **793**, 844
Stressfaktor, 333
Streuung, **377**, 379, 471, 613, 652
strukturierendes Element, 457
stückweise lineare Skalierung, 403
subsampling, 344
subtraktives Farbmodell, 133, 365
Summenoperator, **442**, 621
Supremum, 754
Symmetrie für fuzzy-Mengen, 754
symmetrischer Filterkern, 633
Synthese von Objekten, 343
synthetic video object, 16
Systematik, 492
Systematikbild, 481, 482
Systemfehler, 737, 742
Systemgleichung, 812
Systemkorrekturen, 552
Systemmatrix, 820
Systemrauschen, 820
Systemzustand, 812, 820
Szenenanalyse, 561
Szenen Graph, 15, 294, **296**
Szenenkomplexität, 293

Teilmenge einer fuzzy-Menge, 754
Telepräsenz, 36
Tesselierung, 215, 217
Tessellierung, 77
Texel, 217
Textur, **215**, 343, 397, **611**, 621, 650
    -Abbildungen, 244
    -Environment, 239
    -Filter, 218, 228
    -Fortsetzungsmodus, 236
    -Koordinaten, 228
    -Koordinaten,
        Automatische Zuordnung von, 249
        Explizite Zuordnung von, 242
        Lineare Interpolation von, 217
        vier Komponenten (s, t, r, q), 242
        Zuordnung von, 242
    -Matrizen Stapel, 253
    -Objekte, 254
    -Vergrößerung, 229
    -Verkleinerung, 229
    -Wraps, 236
    1-dimensionale, 217
    2-dimensionale, 217
    3-dimensionale, 217
    Anisotrope MipMap-Filter, 236
    Beleuchtungs-, 216
    Foto-, 13, 215, 315
    Foto-,, **217**
    fraktale, 217
    Gauß-Pyramiden-, 231
    Glanz-, 216
    kubische, 21
    Kubische, 266
    Mehrfach-, 256
    MipMap-, 231
    MipMap-Filter, 234
    Projektive, 260
    Projektive Schatten-, 261
    prozedurale, 217
    Relief-, 13, 216, 270
    Schatten-, 13, 216, 277

Spezifikation, 219
Sphärische, 263
transparente, 18
transparente,, **136**
Umgebungs-, 216, 262
Textur-Speicher, 47
Texture-Mapping, 215
Einschalten des -, 253
Texturfenster, **612**, 615, 670, 672
Texturkanten, 470, **612**, 613
Texturmerkmal, 17, 490, 652
Texturparameter, 475, **612**
Tiefpassfilterung, 144, 230, 534, 550, 623
Tilgungskriterium, 496
Tönung, 358, 366
Tracking, 812
Training, 731, 732, 794, 804
Trainingspaar, 733
Trainingsphase, 741
Transformation, 80
Augenpunkt-, 93
inverse Projektions-, 118
Matrizen-, 84, 296
Modell-, 86
Normierungs-, 100
Orthografische, 94
Perspektivische, 95
Projektions-, 94, 217
Reihenfolge der, 91
Viewport-, 101, 217
Transformationsraum, 512
Transitivität für fuzzy-Mengen, 754
Translation, 86, 296
translationsinvariante Segmenterkennung, 776, 793, 804
Transmission, 167
Transparenz, 119, **127**, 157, 171
-Komponente A (Alpha), 119, 217, 315
Transport-Verzögerung, 30, 282, 323
traversal
application, 299, 319
cull, 299, 319
draw, 301
intersection, 299
Trennung von Objekten vom Hintergrund, 400
Triangle-Fan, 72
Triangle-Strips, 71
Triangles, 70
trigonometrische Approximationsfunktionen, 530
trigonometrisches Polynom, 530

Übertragungsfunktion, 534
Umfang, 779
umgebungsbezogene Merkmale, 343
Umrechnung der Bildkoordinaten
mit der direkten Methode, 548
mit der indirekten Methode, 549
Umriß eines Segments, 782
uniform quantiser, 346
Union-Find, 773
unkorreliert, 378
unscharfe Logik, 749
unscharfe Mengenlehre, 749
unscharfes Schließen, 756
Unterabtastung, 344
unüberwachte Klassifizierung
zur Reduktion der Farben, 583
unüberwachter Klassifikator, 749

Validation, 13, 36
Varianz, **377**, 379
Vektor der mittleren quadratischen Abweichungen, 390
Vektor der Streuungen, 719
Vektorisierung, 502
verallgemeinerte Houghtransformation, 512
Verarbeitung von Linien, 469, 492
Verarbeitung von mehrkanaligen Bildern, 742
Verbesserung verrauschter Einzelbilder, 440
verdeckte Schicht, 730
Verdeckung, 108, 171, 303
Verdeckungsgrad, 325
Verdichtungsverfahren, 728
Verdünnen von Linien, 492
Vereinigung für fuzzy-Mengen, 755
Vereinzelung von Segmenten, 769
Vergrößerung, 548
Verkleinerung, 548
Verlustmatrix, 715
Vermessung der Passpunkte, 559
Verschiebungsvektor, 18, 22, 40, 551
differentielle Ermittelung, 603

Ermittelung von mit Blockmatching, 605
Verschiebungsvektorfeld, 22, 526, **603**, 817
Verteilung
diskret, 376
stetig, 376
Vertex, Mz. Vertices, 53, **56**, 84, 160, 293
Vertex-Operationen, 46
Vertex-Programm, 209, 273
Vertex Arrays, 75
vertikale Ausdehnung des Segments, 784
Verwaltungskomponente, 845
Videobandbreite, 325
Videobandbreiten-Limitierung, 328
video capture card, 344
Videokamera, 344
Videosignalgenerier-Stufe, 321
Vierfarbendruck, 365
viewing volume, 95
Vignettierung, 492, 587
visuelle Datenbasis, 297
Vollständigkeitskontrolle, 621, 660
Volumenmessung, 116
Vorquantisierung der Farben, 573
Vorverarbeitung, 342, 813
vorwärts- und rückwärtsvermittelnde Netze, 732
Vorwärtsschritt, 733
Vorwärtsvermittlung, 732
Voxel, 53

Wahrscheinlichkeitsfunktion, 377
Wellenlänge, 161, 166, 622
Wellenzahl, 622
Wellenzahlindex, 622, 658
Wiener Filterung, 542

YIQ-Farbmodell, 366

z-Buffer, 110, 153, 278
Algorithmus, 109, 138, 303
Flimmern, 112
z-Pyramide, 305
z-Wert (Tiefeninformation), 109
Zählen von Segmenten, 457
Zeilenabtaster, 344
Zeilendruckerausgabe von Grauwertbildern, 384
Zeilenindex, 351
zeitbezogene Merkmale, 343
Zufallscodierung, 388
Zufallsprozess, 376
Autokorrelation, 379
Autokovarianz, 379
Erwartungswert, 379
homogen, 380
Mittelwert, 379
stochastischer, 379
Streuung, 379
Varianz, 379
zweidimensionaler, 379
Zugehörigkeitsfunktion, 751
Zurückweisungsklasse, 704, 716
Zusammenfassen von Segmenten zu Objekten, 840
Zusammensetzens eines Bildes aus mehreren Eingabebildern, 621
zweidimensionale Gauß-Funktion, 670
zweidimensionaler Zufallsprozess, 379
Zweipegelbild, 350, 412, 490