Computergrafik

Vorlesung gehalten im SS 2002

Olaf Müller
Ralf Kunze

Fachbereich Mathematik/Informatik
Universität Osnabrück

Literatur

- J. Blinn:
  ”Jim Blinn’s Corner: A Trip Down the Graphics Pipeline”, Morgan Kaufmann Publishers, 1996.
- J. Encarnaçaõ, W. Straßer, R. Klein:
  “Graphische Datenverarbeitung 1 + 2”, Oldenbourg Verlag, 1996
- W. D. Fellner:
  “Computergrafik”, BI Wissenschaftsverlag, 1992.
- J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, R. L. Phillips:
  “Grundlagen der Computergraphik”, Addison Wesley, 1994.
- J. D. Foley, A. van Dam, S. K. Feiner et al:
  “Computer Graphics: Principle and Practice”, Addison Wesley, 2000.
- Thomas Möller, Erik Haines:
  ”Real-Time Rendering”, A K Peters, 1999.
- M. Woo, J. Neider, T. Davis, et al:
  “OpenGL Programming Guide: The Official Guide to Learning OpenGL 1.2”, Addison Wesley, 1999
- David Flanagan:
  ”Java in a nutshell”, O’Reilly, 1999.
- Ken Arnold, James Gosling:
  ”Die Programmiersprache Java”, Addison-Wesley, 2001.

Danksagung

Wir danken ...

... Gerda Holmann und Astrid Heinze für sorgfältiges Erfassen des Textes und Erstellen der Grafiken,

... Frank M. Thiesing für intensive Mitwirkung an der inhaltlichen Gestaltung der Vorlesung,

... Frank Lohmeyer für die Erstellung umfangreicher Java-Software zur Implementation von Beispielapplikationen,

... Viktor Herzog für die Konvertierung des Skripts nach HTML.

HTML-Version
Der Inhalt dieser Vorlesung kann online betrachtet werden unter
`http://www-lehre.inf.uos.de/~cg`

Osnabrück, im Juli 2002

(Olaf Müller) (Ralf Kunze)

Inhaltsverzeichnis

1 Einführung **13**

1.1 Motivation 13

1.2 Definition 13

1.3 Anwendungen 14

1.4 Kurze Geschichte der Computergrafik 14

1.5 Ausgabegeräte 15

1.5.1 Soft Copy 15

1.5.2 Hardcopy 15

1.6 Eingabegeräte 16

1.6.1 Technische Klassifizierung der Eingabegeräte 16

1.6.2 Logische Klassifizierung der Eingabegeräte 17

1.6.3 Eingabe-Modi 17

2 Grafische Oberflächen und ihre Programmierung **19**

2.1 Der Window-Manager 19

2.2 Swing 20

2.2.1 Swing-Übersicht 20

2.3 Swing-Beispiel 24

2.3.1 Applet 25

2.3.2 Applikation 26

3 2D-Grundlagen **29**

3.1 Koordinatensysteme 29

3.2 Punkt 30

3.3 Linie 31

3.3.1 Parametrisierte Geradengleichung 31

3.3.2 Geradengleichung als Funktion 32

3.3.3 Bresenham-Algorithmus . . . 33
3.3.4 Antialiasing . . . 38
3.4 Polygon . . . 39
3.4.1 Konvexität . . . 39
3.4.2 Schwerpunkt . . . 40
3.5 Kreis . . . 41
3.5.1 Trigonometrische Funktionen . . . 42
3.5.2 Bresenham-Algorithmus . . . 43

4 2D-Füllen **49**
4.1 Universelle Füll-Verfahren . . . 49
4.2 Scan-Line-Verfahren für Polygone . . . 53
4.3 Dithering . . . 54
4.4 Punkt in Polygon . . . 57
4.4.1 Kreuzungszahl . . . 57
4.4.2 Umlaufszahl . . . 60

5 Clipping **63**
5.1 Clipping von Linien . . . 63
5.2 Clipping von Polygonen . . . 65
5.3 Beispiel-Applet zu 2D-Operationen . . . 67

6 2D-Transformationen **69**
6.1 Translation . . . 69
6.2 Skalierung . . . 69
6.3 Rotation . . . 71
6.4 Matrixdarstellung . . . 73
6.5 Homogene Koordinaten . . . 73
6.6 Allgemeine Transformationen . . . 76
6.7 Raster-Transformationen . . . 76

7 Kurven **77**
7.1 Algebraischer Ansatz . . . 77
7.2 Kubische Splines . . . 78
7.3 Bézier-Kurven . . . 82
7.4 B-Splines . . . 85

7.4.1 Eigenschaften 86
7.5 NURBS 88

8 Farbe **91**
8.1 Physik 91
8.2 Dominante Wellenlänge 92
8.3 Grundfarben 92
8.4 RGB-Modell (Rot, Grün, Blau), (additiv) 94
8.5 CMY-Modell (Cyan, Magenta, Yellow), (subtraktiv) 94
8.6 YUV-Modell 95
8.7 YIQ-Modell 96
8.8 HSV-Modell 96
8.9 Color Table (Farbtabelle) 100
8.10 Erzeugung einer bildbezogenen Farbtabelle 104
8.11 Alpha-Kanal 108

9 Pixeldateiformate **111**
9.1 PBM, PGM und PPM 111
9.2 Kompression nach JPEG 114

10 3D-Grundlagen **123**
10.1 3D-Koordinatensystem 123
10.2 Länge und Kreuzprodukt 123
10.3 Skalarprodukt 126
10.4 Matrixinversion 126

11 3D-Transformationen **129**
11.1 Translation 129
11.2 Skalierung 129
11.3 Rotation 130
11.4 Transformation von Koordinatensystemen 133
11.5 Transformation der Normalenvektoren 135

12 Projektion **137**
12.1 Bildebene 137
12.2 Perspektivische Projektion 138

12.3 Parallelprojektion 140
12.3.1 Normalprojektionen 140
12.3.2 Schiefe Projektionen 140

13 Betrachtungstransformationen 143
13.1 Die synthetische Kamera 143
13.2 Viewing Pipeline 144
13.2.1 Modeling-Transformationen 145
13.2.2 View Orientation 145
13.2.3 View Volume 147
13.2.4 View Mapping 148
13.2.5 Device Mapping 151
13.2.6 Zusammenfassung 153
13.3 Clipping 153
13.3.1 Clipping im WC 154
13.3.2 Clipping im NPC 154
13.3.3 Vergleich der beiden Vorgehensweisen 154

14 3D-Repräsentation 157
14.1 Übersicht 157
14.2 Repräsentation 157
14.2.1 Elementarobjekte 158
14.2.2 Drahtmodell 158
14.2.3 Flächenmodell 158
14.2.4 CSG (constructive solid geometry) 159

15 3D-Objekte 161
15.1 Polyeder 161
15.1.1 Würfel 162
15.2 Gekrümmte Flächen 162
15.2.1 Zylinder 162
15.2.2 Kugel 163
15.2.3 NURBS-Flächen 164

16 Culling Techniken 167
16.1 Back-Face Removal/Culling 167

16.2 Hidden-Surface Removal 168
16.2.1 z-Buffer-Algorithmus 169
16.2.2 Span-Buffer 170
16.2.3 Binary Space Partitioning 171

17 Rasterung von Flächen **175**
17.1 Beleuchtung 177
17.1.1 Lichtquellen 177
17.1.2 Reflexionseigenschaften 178
17.1.3 Oberflächeneigenschaften 179
17.1.4 Materialeigenschaften 181
17.2 Schattierungsalgorithmen 181
17.2.1 Flat-Shading 181
17.2.2 Gouraud-Shading 181
17.2.3 Phong-Shading 182
17.3 Schatten 183

18 Texturing **185**
18.1 Texture Mapping 185
18.2 Mip Mapping 187
18.3 Light, Gloss und Shadow Mapping 187
18.4 Alpha Mapping 187
18.5 Environment oder Reflection Mapping 188
18.6 Bump Mapping 188
18.7 Multitexturing 188
18.8 Displacement Mapping 188

19 Hardwarebeschleunigung **191**

20 Radiosity **195**
20.1 Globale Beleuchtung 195
20.2 Physikalische Ausgangslage 196
20.3 Die Radiosity-Gleichung (Beleuchtungsgleichung) 196
20.4 Berechnung der Formfaktoren 198
20.5 Interpolation der Pixelfarben 200
20.6 Schrittweise Verfeinerung 200

20.7 Screenshots 203

21 Ray Tracing **205**

21.1 Grundlagen 205

21.2 Ermittlung sichtbarer Flächen durch Ray Tracing 206

21.3 Berechnung von Schnittpunkten 207

21.4 Effizienzsteigerung zur Ermittlung sichtbarer Flächen 207

21.5 Rekursives Ray Tracing 209

21.6 Public Domain Ray Tracer Povray 213

Kapitel 1

Einführung

1.1 Motivation

- "Ein Bild sagt mehr als 1000 Worte."
- Das Auge erfaßt 40 Millionen Bit pro Sekunde.
- Lesegeschwindigkeit = 10 Worte mit duchschnittlich 5 Buchstaben im ASCII-Zeichensatz pro Sekunde $= (10 \times 5 \times 8) = 400$ Bit pro Sekunde.
  $\Rightarrow$ Faktor 100.000.

1.2 Definition

Der Begriff *Grafische Datenverarbeitung* umfaßt

generative Computergrafik
: Erzeugung künstlicher Bilder aus einer Beschreibung:
Eingabe der Beschreibung
Manipulation der Beschreibung
Ausgabe des zur Beschreibung gehörigen Bildes

Bildverarbeitung
: Bildveränderung, so daß der Informationsgehalt leichter erkennbar wird:
Verfänderung von Bilddaten (z.B. Drehung)
Verbesserung von Bilddaten (z.B. Kontrasterhöhung)
Vereinfachung von Bilddaten (z.B. Farbreduzierung)

Mustererkennung
: Analyse von Bilddaten durch Zerlegung in bekannte graphische Objekte:
z.B. wo verläuft die Straße?
z.B. um welchen Buchstaben handelt es sich?

In dieser Vorlesung geht es ausschließlich um generative Computergrafik!

1.3 Anwendungen

Typische Anwendungsbereiche von Computergrafik sind:

- **Grafische Benutzungsoberflächen:** Point & Click statt Kommandos
- **Business-Grafik:** Balken- & Tortendiagramme statt Zahlenfriedhof
- **Kartografie:** Landkarten, Flächennutzungsplan
- **CAD (Computer aided design):** Entwurf von Autos, Häusern, Maschinen, VLSI-Chips
- **Echtzeitsimulation und -animation:** Flug- und Fahrsimulator
- **Überwachungs- und Steuerungssysteme:** Anlagen- und Kraftwerkssteuerung und Produktionsüberwachung
- **Medizin:** 3D-Darstellung einer Computer-Tomographie
- **Visualisierung in Bildung und Wissenschaft:** Darstellung von Funktionen, Molekülen, Kristallen oder Gensequenzen, interaktive Animation von komplexen Objekten (z.B. Viertaktmotor)
- **Unterhaltung:** Computerspiele, Spielfilme, Virtual Reality, Cyberspace

1.4 Kurze Geschichte der Computergrafik

1950	grobe Grafikdarstellung mit Matrixdrucker
1950	computergesteuerte Kathodenstrahlröhre
1963	Dissertation von Ivan Sutherland: *Sketchpad - A Man Machine Graphical Communication System*
1964	Automobilentwurf bei GM; Anfänge von CAD/CAM
1965	Doug Engelbart: Maus ersetzt Lichtgriffel
1980	Apple Macintosh und IBM PC mit Bitmap-Grafik und Maus
1985	GKS: Grafisches Kernsystem
1986	X-Windows vom MIT
1987	GKS-3D
1988	PHIGS: Programmer's Hierarchical Interactive Graphics System
1989	PEX: PHIGS Extension to X
1991	CGI: Computer Graphics Interface
1992	ISO PHIGS PLUS: (+ Rendering)
1993	OpenGL
1995	DirectX 1.0
1996	3D-Grafikkarte für den Consumerbereich (VooDoo 1)
1998	nVidia Riva TNT (2D/3D-Kombigrafikkarte)
2002	OpenGL 2.0 und DirectX 9.0?

1.5 Ausgabegeräte

Die generierte Grafik kann mithilfe von zwei Sorten Geräten ausgegeben werden:

1.5.1 Soft Copy

Die erzeugten Grafiken sind ”flüchtig”, da sie nicht zu Papier gebracht werden und deshalb spätestens beim Ausschalten des Computers/Ausgabegerätes verloren sind.

- Speicherbildschirm
  leuchtet extrem lang nach
  Vorteil: flimmerfrei
  Nachteil: kein selektives Löschen
- Refresh-Bildschirm
  Kathode emittiert Elektronen zur Anode (Phosphorschirm). Ablenkung erfolgt durch Magnetfeld. Auffrischung mit > 50 Hz.
- Farbbildröhre
  Schirm ist mit Tripeln von rotem, grünem, blauem Phosphor beschichtet, davor Lochplatte, 3 Kathoden
- Flach-Bildschirme
  - Plasma-Schirm
    2 Glasplatten mit horizontalen und vertikalen Leiterbahnen, dazwischen, an den Kreuzungspunkten der Leiterbahnen, Zellen mit Neongas.
  - LCD (Liquid Crystal Display)
  - TFT-Schirm (Thin Film Transistor)

  Vorteil: flach, leicht, flimmerfrei, hoher Kontrast (TFT)
  Nachteil: noch relativ teuer

1.5.2 Hardcopy

Die Grafik wird zu Papier gebracht und man kann sie ohne die Hilfe des Ausgabegerätes jederzeit wieder betrachten.

- Plotter (zeichnet mit Stiften aus Magazin)
- Matrixdrucker (setzt Nadeln auf Farbband)
- Tintenstrahldrucker (spritzt Farbtropfen aus Düse)
- Laserdrucker (überträgt aufgeladene Tonerpartikel)
- Thermosublimationsdrucker (löst Farbpigmente aus Folie)

Random Scan versus Raster Scan

Die Geräte beider Kategorien funktionieren nach einem der folgenden Prinzipien:

- Random Scan = vektororientiert
  Im Display-File liegen Vektorinformationen, die permanent angezeigt werden (Lasershow).
  Vorteil: hohe Auflösung
  Nachteil: Füllung schwierig, nur Linien

- Raster Scan
  Elektronenstrahl tastet Schirm zeilenweise ab. Er wird dabei in der Intensität variiert, je nachdem, wie hell der überstrichene Bildpunkt erscheinen soll. Die Information für jeden Bildpunkt (**pic**ture **el**ement = Pixel) wird aus dem Bildwiederholspeicher (frame buffer) = VIDEO-RAM geholt. Der Grafikprozessor legt dort die Bildinformationen ab.
  Vorteil: beliebig komplexer Bildaufbau
  Nachteil: diskrete Darstellung der Pixel: Linien müssen durch Rasterung (Scan Conversion) dargestellt werden. Antialiasing erforderlich.

1.6 Eingabegeräte

1.6.1 Technische Klassifizierung der Eingabegeräte

- Keyboard (Tastatur)
- Paddles (Drehknopf), z.B. bei CAD-Arbeitsplätzen zum Transformieren
- Joy-Stick (relative Koordinatenerfassung durch Stift-Auslenkung)
- Maus (relative Koordinatenerfassung durch Rollrad-Bewegung)
- Track-Ball (versenkte Kugel)
- Tablett (absolute Koordinatenerfassung durch Induktion im Drahtgitter)
- Lichtgriffel (Stift mit Photodiode)
- Touch Panel (Bildschirm zum Anfassen)
- Frame Grabber (Übernahme des Signals einer Videokamera)
- Scanner (optisches Abtasten und Digitalisieren einer Bildvorlage)
- 3D-Scanner (optisches Erfassen von Form und Aussehen des Vorlageobjektes)

1.6.2 Logische Klassifizierung der Eingabegeräte

Locator:	liefert Position	z.B. Joy-Stick, Trackball, Maus, Griffel, Tablett
Stroke:	liefert Folge von Positionen	z.B. Maus, Griffel, Tablett
Valuator:	liefert skalaren Wert	z.B. Keyboard, Paddles
Choice:	liefert Auswahl	z.B. Maus, Griffel, Touch Panel
Pick:	wählt angezeigtes Objekt	z.B. Maus, Griffel, Touch Panel
String:	liefert Zeichenfolge	z.B. Keyboard

Simulation möglich, z.B. Maus simuliert Valuator durch Scrollbar.

Probleme bei Pick

- Benutzer tippt nur in die Nähe.
- Benutzer muß gefüllte Objekte picken können.
- Benutzer muß verdeckte Objekte picken können.

1.6.3 Eingabe-Modi

Kommunikation zwischen Anwendungsprogramm und Eingabegeräten:

- Anforderungsmodus (Request Mode)
  Programm wartet auf Eingabe, z.B. Tastendruck, Buttondruck.
  Eingabegerät ist nur aktiv, wenn Daten benötigt werden.
- Abfragemodus (Sample Mode)
  Eingabegerät ist parallel zur Applikation aktiv und aktualisiert laufend die gemessenen Daten, z.B., Tablett sendet permanent (x, y)-Werte.
- Ereignismodus (Event Mode)
  Eingabegerät entkoppelt von Programm, parallel aktiv.
  Eingabegerät füttert Event-Queue, die von der Applikation geleert wird, z.B., Benutzer klickt mehrere Bildschirm-Buttons an.
  Programm kann spezielle Events entnehmen.

Kapitel 2

Grafische Oberflächen und ihre Programmierung

Das erste Window-System wurde in den 70er Jahren von Xerox PARC entwickelt. Ende der 70er Jahre traten die grafischen Oberflächen mit den Apple Computern Lisa und Macintosh ihren Siegeszug an.

Grafische Benutzungsoberflächen haben sich heute zum Standard für anwenderfreundliche Applikationsprogramme entwickelt und rein textuelle Mensch-Maschine-Kommunikation weitgehend abgelöst. Was dem Benutzer im allgemeinen das Leben erleichtern soll, stellt für den Programmierer eine echte Herausforderung dar.

Auf dem zweidimensionalen Bildschirm müssen eine Vielzahl von Fenstern, Icons und Infoboxen mit unterschiedlichster Funktionalität dargestellt werden. Diese müssen durch Eingabegeräte wie Maus oder Tastatur vom Benutzer arrangiert, aktiviert oder mit Eingaben versehen werden können.

Neben der angestrebten intuitiven Bedienbarkeit von grafischen Benutzungsoberflächen erlaubt ein Window-System dem Benutzer auch die Ausgaben mehrerer Programme gleichzeitig zu beobachten und so quasi parallel zu arbeiten.

2.1 Der Window-Manager

Für die geometrische Verwaltung der einzelnen Anwendungen auf einem Bildschirm steht dem Anwender ein Window-Manager zur Verfügung. Er legt die Fenster-Layout-Politik fest.

Der Window-Manager manipuliert Umrandung, Größe und Position der Fenster sowie die Reihenfolge, in der die Anwendungen übereinanderliegen. So kann er verdeckte Fenster in den Vordergrund holen, Fenster über den Schirm bewegen und deren Größe verändern.

Für den Inhalt ihrer Fenster ist die Applikation selbst verantwortlich. Muß ein Fenster nachgezeichnet werden, weil es z.B. durch eine Verschiebe-Aktion nun nicht mehr verdeckt ist, sendet der Window-Manager lediglich eine Redraw-Nachricht an die Applikation.

Außerdem kann der Window-Manager neue Applikationen starten.

Für (fast) jedes Betriebssystem gibt es ein zugehöriges Oberflächensystem, das mit Hilfe eines API (Application Programmer's Interface) manipuliert werden kann. Die Programmierung einer grafischen

Applikation erfolgt mit Hilfe einer Graphical User Interface (GUI)-Sprache, die in einer Hochsprache alle notwendigen API-Vokabeln zur Beschreibung des Aufbaus und Ablaufs einer interaktiven grafischen Anwendung bereitstellt. Die Programmierung erfolgt zumeist in einer gängigen Programmiersprache, die GUI-Bibliotheken verwendet.

Die Beispiele in dieser Vorlesung werden in Java 1.3 unter Zuhilfenahme der Swing-API implementiert.

2.2 Swing

Betrachtet man die verschiedenen Systeme und ihre GUIs, so stellt man fest, daß die Funktionalität bei allen relativ ähnlich ist, die Programmierumgebungen aber völlig verschieden sind. Will man für eine Applikation eine grafische Benutzungsschnittstelle entwickeln, die auf allen verwendeten Systemen eine identische Funktionalität aufweist, so benötigt man eine portable Programmierumgebung. Eine solche Umgebung sind die *Swing-Komponenten* der Programmiersprache Java. Sie sind eine Teilmenge der *Java Foundation Classes* (JFC), basieren auf dem *Abstract Window Toolkit* (AWT) und erweitern dieses um eine Reihe mächtiger GUI-Elemente.

Das AWT umfaßt einige der Java-Standard-Klassen, die für die portable Programmierung von GUI-Applikationen entwickelt wurden. Es heißt *abstract*, weil es die GUI-Elemente nicht selber rendert, sondern sich auf existierende Window-Systeme abstützt.

Im Gegensatz zum AWT kann mit den Swing-Komponenten eine Applikation so implementiert werden, daß alle GUI-Elemente unabhängig von der Plattform, auf der das Programm ausgeführt wird, das gleiche Aussehen und die gleiche Funktionalität haben (*pluggable look and feel* (plaf)). Unter Java2 beinhaltet dies auch den Austausch von Daten zwischen den GUI-Elementen durch den User (*drag and drop*). Dadurch, daß die Swing-Komponenten **keinen** architekturspezifischen Code enthalten, sind sie wesentlich flexibler als die AWT-Klassen. Diese Unabhängigkeit vom Betriebs- bzw. Window-System ist allgemein Teil der Java-Philosophie.

2.2.1 Swing-Übersicht

In diesem Abschnitt wird eine kurze Übersicht der wesentlichen Konzepte der Swing-Komponenten und einiger AWT-Klassen gegeben. Die für die Programmierung notwendigen Details sind bei Java generell der HTML-Online-Dokumentation zu entnehmen.

Das Basis-Paket für Grafik- und Oberflächenprogrammierung ist `javax.swing`. Wie schon erwähnt, werden die dort zur Verfügung gestellten Klassen auf entsprechende Elemente eines schon vorhandenen Window-Systems abgebildet. Zu diesem Zweck gibt es das Paket `java.awt.peer`, das aber normalerweise vom Programmierer nicht direkt benutzt wird.

Weitere verwandte Klassen sind z.Zt. (d.h. in der Java-Version 1.1.x) `java.awt.image` mit Klassen für die direkte Manipulation von Bildern, `java.awt.event` für das neue Eventhandling-Konzept, sowie `java.awt.datatransfer` mit Möglichkeiten zum Datenaustausch zwischen Applikationen. Erwähnt werden sollte hier auch `javax.swing.applet`, dessen einzige Klasse `JApplet` verwendet wird, wenn man Java-Programme in einem HTML-Dokument benutzen will.

Die Klassen des swing-Pakets lassen sich grob in drei Gruppen einteilen: Grafikklassen, in denen grafische Objekte wie Farben, Fonts, Bilder beschrieben werden; Komponenten, d.h. Klassen, die GUI-

Komponenten wie z.B. Buttons, Menüs und Textfelder zur Verfügung stellen; sowie Layout-Manager, die die Anordnung von GUI-Komponenten in einer Applikation kontrollieren. Aus den verwandten Paketen sollen hier noch die Event-Listener besprochen werden, mit deren Hilfe die Interaktion zwischen User und Applikationen implementiert werden muß. Die folgende Übersicht erhebt keinen Anspruch auf Vollständigkeit, sondern beschreibt nur kurz die für Computergrafik-Programme wichtigen Klassen. Weitere Klassen sowie alle Details sind der entsprechenden On-Line-Dokumentation zu entnehmen.

Die Grafik-Klassen

Die Grafik-Klassen implementieren jeweils einen bestimmten Aspekt und sind weitgehend unabhängig voneinander. Wichtige Klassen des AWT sind `Color`, `Cursor`, `Font`, `FontMetrics`, `Image`, `MediaTracker`, deren Bedeutung aus dem jeweiligen Namen ersichtlich ist. Wichtige Hilfsklassen sind `Dimension`, `Point`, `Polygon`, `Rectangle`, die jeweils entsprechende Objekte verwalten. Dabei ist zu beachten, daß alle diese Objekte keine Methoden haben, mit denen sie sich z.B. auf dem Schirm darstellen können. Diese Funktionalität wird durch die Klasse `Graphcis` bereitgestellt, in der Methoden vorhanden sind, um Bilder und andere grafische Objekte zu zeichnen, zu füllen, Farben oder Fonts zu ändern, Ausschnitte zu kopieren oder zu clippen usw. Diese abstrakte Klasse wird von diversen Komponenten bereitgestellt und kann dann verwendet werden, um das Aussehen dieser Komponente zu manipulieren.

Die GUI-Komponenten

Die Komponenten, mit deren Hilfe der Benutzer mit einer Oberfläche interagieren kann, lassen sich in drei Gruppen einteilen:

Es gibt Komponenten, die andere Komponenten aufnehmen und anordnen können. Sie werden *Container* genannt und unterteilen sich wiederum in drei Gruppen:

- *Top-Level-Container*, wie `JFrame`, `JDialog` und `JApplet`, die jeweils ein eigenstaendiges "Fenster" erzeugen, das direkt in die grafische Benutztungsoberfläche eingeblendet und vom Window-Manager verwaltet wird.
- *General-Purpose Container*, wie `JPanel` und `JScrollPane`, die in erster Linie beliebige andere Componenten aufnehmen und deshalb nur eingeschränkte eigene Funktionalität bieten.
- *Special-Purpose Container*, wie `JInternalFrame` und `JLayeredPane`, die eine festgelegte Rolle in der Benutzerschnittstelle spielen und deshalb nur eingeschränkt konfgurierbar sind.

Weiterhin gibt es Komponenten, die eine spezielle Funktionalität bereitstellen und als *Basic Controls* bezeichnet werden.

- `JButton`: Es wird ein mit einem Titel oder Bild versehenes Feld erzeugt, das mit der Maus angeklickt werden kann, wodurch ein entsprechender Event erzeugt wird.
- `JCheckbox`: Ein Schalter mit zwei Zuständen, der bei einem Mausklick diesen wechselt und dabei einen Event auslöst.

- `JComboBox`: Eine Gruppe von Einträgen, von der immer nur einer sichtbar ist. Das Auswählen eines Eintrags erzeugt einen entsprechenden Event.
- `JList`: Es wird eine Liste von Strings angezeigt, aus der Einträge selektiert bzw. deselektiert werden können, was zur Erzeugung von entsprechenden Events führt.
- `JScrollBar`: Ein Rollbalken, der mit der Maus bewegt werden kann, wodurch entsprechende Events erzeugt werden.

Außerdem gibt es noch *Displays*, die auf unterschiedliche aber für die einzelne Klasse spezifische Weise Informationen anzeigen. Einige Klassen können editierbar (*editable*) erzeugt werden:

- `JTextArea`: Ein Feld, in dem ein längerer Text angezeigt und evtl. editiert werden kann.
- `JTextField`: Ein Text-Feld, in dem ein einzelner String angezeigt und evtl. editiert werden kann.

Andere sind nicht editierbar (*uneditable*). Hierzu gehören z.B.:

- `JProgressBar`: Ein Fortschrittsanzeiger, der einen prozentualen Wert grafisch darstellen kann.
- `JLabel`: Diese einfachste Komponente erzeugt ein Feld, in dem ein String oder ein Bild ausgegeben wird.

Da es die hier genannten Komponenten zum Teil auch in *java.awt* gibt, wurde den Swing-Komponentennamen zur besseren Unterscheidung jeweils ein "J" vorangestellt.

Da alle Komponenten von der Klasse `javax.swing.JComponent` abgeleitet wurden, die ihrerseits von `java.awt.Container` abstammt, kann jede Instanz einer dieser Klassen wiederum selber andere Komponenten aufnehmen und diese in ihrem Inneren frei anordnen. Dieses *Layout* regelt eine weitere wichtige Gruppe von Klassen:

Die Layout-Manager

Die wesentliche Aufgabe eines Containers ist die Darstellung der darin enthaltenen Komponenten. Zur Steuerung der Anordnung dieser Komponenten werden Klassen verwendet, die das Interface `Layout-Manager` adaptieren. Ein solcher Layout-Manager implementiert dazu eine spezielle Layout-Politik, die mehr oder weniger konfigurierbar ist. Es gibt die folgenden vordefinierten Layout-Manager, die alle aus `java.awt` stammen:

- `FlowLayout`: Dies ist das einfachste Layout, welches bei `JPanel` voreingestellt ist. Jede Komponente wird in seiner bevorzugten Größe rechts von der vorigen Komponente eingefügt. Ist der rechte Rand des Containers erreicht, wird eine weitere Zeile angehängt, in der weitere Komponenten eingefügt werden können.
- `BorderLayout`: Dies ist ebenfalls ein einfaches Layout, das als Default-Layout bei `JFrame` verwendet wird. Es besteht aus den fünf Gebieten, Nord, Süd, Ost, West und Zentrum, in

denen jeweils beliebige Komponenten plaziert werden können. Im Gegensatz zum `FlowLayout` werden hier die Komponenten an die Größe des Containers angepaßt: Die Nord- und Süd-Komponenten werden in X-Richtung gestreckt bzw. gestaucht; die Ost- und West-Komponenten in Y-Richtung und die Zentrums-Komponente in beiden Richtungen.

- `CardLayout`: Dies ist ein spezielles Layout, bei dem beliebig viele Komponenten eingefügt werden können, von denen aber immer nur eine zu sehen ist, da sie wie bei einem Kartenspiel übereinander angeordnet werden.

- `GridLayout`: Hier werden die Komponenten in einem zweidimensionalen Gitter angeordnet. Dabei werden alle Komponenten auf die gleiche Größe gebracht, wobei die größte Komponente die Gittergröße vorgibt.

- `GridBagLayout`: Dies ist der flexibelste Manager, mit dem beliebige Anordnungen möglich sind. Es wird wieder von einem zweidimensionalen Gitter ausgegangen, wobei hier jetzt aber eingestellt werden kann, wie viele Gitterzellen eine Komponente belegt und wie sie sich anpaßt.

Alle diese Layout-Manager bestimmen nur die Anordnung der GUI-Elemente, nicht aber deren Look-And-Feel. Diese Aufgabe übernimmt der *User Interface Manager* (`UIManager`). Mit dieser Klasse kann aus den mitgelieferten (z.Zt. Java, Mac, Motif und MS Windows) und evtl. selbstimplementierten Look-And-Feel's gewählt werden.

Event-Handling

Eine offene Frage ist nun noch, wie das Aktivieren einer GUI-Komponente eine entsprechende Aktion im Benutzer-Programm auslöst. Dazu gibt es in AWT folgendes Konzept: Eine Aktion des Benutzers — wie z.B. ein Mausklick — erzeugt in der Java-Maschine einen *Event*. Je nach Art der Aktion und des Kontextes, in dem sie stattfindet, ergeben sich verschiedene Event-Typen, die wiederum verschiedene Argumente haben können. Die Events, die in Zusammenhang mit einer Komponente ausgelöst werden, können im Anwender-Porgramm empfangen werden, indem ein sogenannter *Event-Listener* mit der Komponente verbunden wird. Für die verschiedenen Event-Typen gibt es verschiedene Listener, von denen einer oder mehrere bei einer Komponente eingetragen werden können. Ein Listener ist dabei ein Interface, in dem alle Methoden vorgegeben werden, die für eine Menge von zusammengehörigen Events notwendig sind. Tabelle 2.1 gibt eine Übersicht der verschiedenen Event-Listener.

Für einige der Swing-Komponenten wurden zusätzliche EventListener implementiert. Sie befinden sich in dem Paket `javax.swing.event`. Sie kommen nur bei spezialisierten Komponenten zum Einsatz und werden daher hier nicht ausführlich beschrieben.

Listener	Methoden	Komponenten
Action	actionPerformed	JButton, JList, JMenuItem, JTextField
Adjustment	adjustmentValueChanged	JScrollBar
Component	componentHidden	JComponent
	componentMoved	
	componentResized	
	componentShown	
Container	componentAdded	Container
	componentRemoved	
Focus	focusGained	JComponent
	focusLost	
Item	itemStateChanged	JCheckbox, JComboBox, JList
Key	keyPressed	JComponent
	keyReleased	
	keyTyped	
Mouse	mouseClicked	JComponent
	mouseEntered	
	mouseExited	
	mousePressed	
	mouseReleased	
MouseMotion	mouseDragged	JComponent
	mouseMoved	
Text	textValueChanged	JTextComponent
Window	windowActivated	JWindow
	windowClosed	
	windowClosing	
	windowDeactivated	
	windowDeiconified	
	windowIconified	
	windowOpened	

Tabelle 2.1: AWT-Event-Listener

2.3 Swing-Beispiel

In diesem Abschnitt wird ein einfaches Beispiel gezeigt, das die Worte “Hello World!” in einem eigenen Fenster darstellt.

Das Beispiel besteht aus zwei Klassen. Die Klasse `Hello` stammt von `JPanel` ab und besitzt nur einen Konstruktor. Bei der Instanziierung wird ein `JLabel` der Größe $160 \cdot 120$ Pixel (Breite mal Höhe) mit dem mittig plazierten String `Hello World!` erzeugt und zum Gebiet “Zentrum” des Containers (`JPanel`) hinzugefügt. Die Instanz der Klasse `Hello` enthält somit alle GUI-Komponenten, die die spezielle Funktionalität der Applikation ausmachen (hier die Darstellung eines

Strings):

```
package hello;

import java.awt.Dimension;                          // Importliste mit allen
import java.awt.BorderLayout;                       // benoetigten Klassen aus
import javax.swing.UIManager;                       // dem AWT und aus den
import javax.swing.JLabel;                          // Swing-Komponenten
import javax.swing.JPanel;

public class Hello extends JPanel {
  public Hello() {
    JLabel l = new JLabel("Hello World!");          // Label mit Text erzeugen
    l.setHorizontalAlignment(JLabel.CENTER);        // horizontal zentrieren
    l.setPreferredSize(new Dimension(160,120));     // Groesse festlegen

    add(l, BorderLayout.CENTER);                    // Label hinzufuegen
  }
}
```

Es fehlt noch der Top-Level Container, der den Rahmen für die Applikation bildet, indem er die Instanz der Klasse `Hello` aufnimmt und dadurch diese auf der grafischen Benutzungsoberfläche darstellbar macht. Diese Funktion übernimmt die Klasse `Main`. Sie besteht aus einem Konstruktor, der Methode `main` und der Methode `init` der Klasse `JApplet`.

Main ist so geschrieben, daß sie sowohl als Applikation als auch in Form eines Applets starten kann.

Die Trennung zwischen Darstellung des Fensters und der Verwaltung/Darstellung des Fensterinhaltes in zwei Klassen bietet die Möglichkeit den Code der Klasse `Main` in zukünftigen Beispielen wiederzuverwenden (lediglich der Titel müsste angepaßt werden).

2.3.1 Applet

Wenn die Klasse als Applet gestartet werden soll, wird sie in eine HTML-Seite eingebunden:

```
<HTML>
<HEAD>
<TITLE>HelloWorld</TITLE>
</HEAD>
<BODY>
Test fuer CG 2002: Swing Applet.
<P>
<APPLET width=160 height=120 codebase=".." code="hello.Main.class">
</APPLET>
</BODY>
</HTML>
```

Beim Start als Applet instanziiert der Appletwiever (oder Browser) eine Instanz der Klasse `hello.Main` und ruft dort die `init`-Methode auf. In dieser Methode wird eine neue Instanz der Klasse `Hello` erzeugt und der `ContentPane` des Applets hinzugefügt.

Es existiert kein Browser, der Java 1.3 ausführen kann. Deshalb kann ein Applet, das Swing-Klassen benutzt nur mit Hilfe eines Java-Plugins ausgeführt werden. Dieses ist ggf. zu installieren.

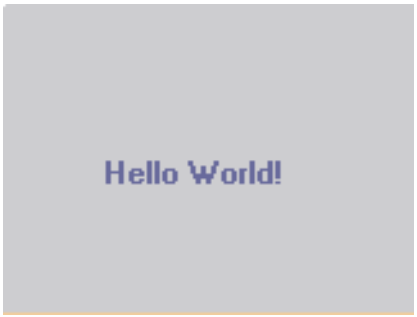


Abbildung 2.1: Beispiel als Applet

2.3.2 Applikation

Beim Start mit dem Kommando

```
java hello.Main
```

wird zunächst in der Methode `main` versucht, die Applikation im javatypischen Look-And-Feel darzustellen. Falls dieser Versuch mißlingt, wird das Look-And-Feel der zugrundeliegenden Benutzungsoberfläche realisiert. Anschließend wird ein Fensterobjekt `JFrame` erzeugt. Dann wird eine Instanz der Klasse `Main` erzeugt und deren Initialisierungs-Methode `init` aufgerufen (s. o.). Dieses `Main`-Objekt wird dem Fensterbereich, der für die Darstellung von GUI-Elementen zur Verfügung steht (*Content Pane*), hinzugefügt. Abschließend wird die Größe der Applikation mit der Methode `pack` berechnet und das Fenster mit der Methode `setVisible` auf der grafischen Benutzungsoberfläche sichtbar gemacht.

```
package hello;

import java.awt.event.WindowAdapter;              // Importliste mit allen
import java.awt.event.WindowEvent;                // benoetigten Klassen aus
import java.awt.BorderLayout;                     // dem AWT und aus den
import javax.swing.UIManager;                     // Swing-Komponenten
import javax.swing.JFrame;
import javax.swing.JApplet;

public class Main extends JApplet {
  public Main() {}

  public void init() {
                                                  // Instanz der Klasse Hello
                                                  // erzeugen und hinzufuegen
```

```
    getContentPane().add(new Hello(), BorderLayout.CENTER);
  }

  public static void main(String args[]) {
    try {                                          // Versuche Look & Feel so zu
      UIManager.setLookAndFeel(                    // setzen, dass es auf allen
                                                   // Systemen gleich aussieht
        UIManager.getCrossPlatformLookAndFeelClassName());
    } catch(Exception e) {};                       // sonst verwende default L&F

    JFrame frame = new JFrame("HelloWorld");       // FensterObjekt erzeugen

    frame.addWindowListener(                       // Listener hinzufuegen
      new WindowAdapter() {                        // Listener erzeugen
        public void windowClosing(WindowEvent e){// Methode ueberschreiben
          System.exit(0);                          // Programm sofort beenden
        }
      });

    Main m = new Main();                           // Instanz erzeugen
    m.init();                                      // initialisieren
                                                   // und zum Fenster hinzufuegen
    frame.getContentPane().add(m, BorderLayout.CENTER);

    frame.pack();                                  // anzeigbar machen und
                                                   // Fenstergroesse berechnen
    frame.setVisible(true);                        // Fenster sichtbar machen
  }
}
```

Beispiel als Applikation

Kapitel 3

2D-Grundlagen

In diesem Kapitel geht es um die Frage, wie man zweidimensionale mathematische Elementarobjekte wie Punkte, Linien und Kreise im Computer repräsentieren und auf dem Bildschirm darstellen kann.

Da die zu beschreibenden Objekte dem Bereich der täglichen Erfahrung entstammen, liegt es nahe, die folgenden Betrachtungen alle im Vektorraum $\mathbb{R}^2$ mit euklidischer Norm durchzuführen.

Die Elemente des $\mathbb{R}^2$ werden *Vektoren* genannt. Sie geben eine Richtung und eine Länge an. Vektoren werden wir mit einem Kleinbuchstaben und einem Pfeil darüber kennzeichnen: $\vec{x}$.

Zum Rechnen mit Vektoren, ist es notwendig, die *Koordinaten* des Vektors bzgl. eines bestimmten Koordinatensystems zu betrachten.

3.1 Koordinatensysteme

Das geläufigste Koordinatensystem für die oben genannte euklidische Ebene ($\mathbb{R}$) ist das *kartesische Koordinatensystem*. Seine beiden Koordinatenachsen sind senkrecht zueinander und schneiden sich im (willkürlich festgelegten) Ursprung O. Die beiden Einheitsvektoren $\vec{e}_x$ und $\vec{e}_y$ sind parallel zu den Achsen und führen, wenn sie von O abgetragen werden, zu den Punkten mit Abstand 1 von O. Als Spaltenvektoren geschrieben, haben sie folgendes Aussehen:

$$\vec{e}_x = \begin{pmatrix} 1 \\ 0 \end{pmatrix};\ \vec{e}_y = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

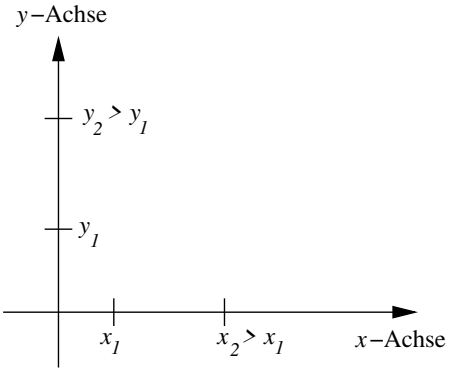
Aus Platzgründen wird stellenweise die Schreibweise als Zeilenvektor verwendet:

$$\vec{e}_x = (1\ \ 0)^T;\ \vec{e}_y = (0\ \ 1)^T$$

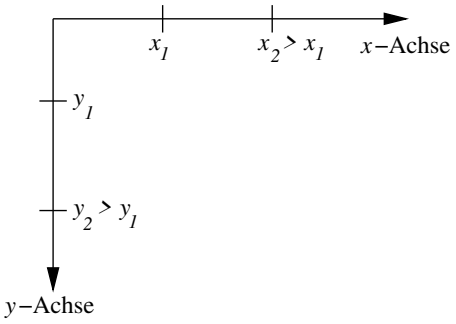
Die erste Koordinatenachse (x-Achse) wird immer von links nach rechts gezeichnet; d.h. die größeren Koordinatenwerte befinden sich weiter rechts.

An der Tafel bzw. in der Vorlesung wird für gewöhnlich die 2. Koordinatenachse (y-Achse) so gezeichnet, daß die größeren Werte weiter oben sind.

Auf dem Bildschirm hingegen befindet sich der Ursprung O oben links; d.h. die y-Achse liegt so, daß sich die größeren Koordinatenwerte weiter unten befinden. Insbesondere hat kein Bildschirmpunkt negative Koordinaten.



Kartesisches Koordinatensystem an der Tafel



Kartesisches Koordinatensystem am Bildschirm

Andere Koordinatensysteme und der Wechsel zwischen diesen werden später ausführlich behandelt.

3.2 Punkt

Mit Hilfe dieses Koordinatensystems läßt sich jeder Punkt P der euklidischen Ebene durch Angabe einer x- und einer y-Koordinate beschreiben:

$$P = (p_x, p_y)$$

Deutlich vom Punkt P zu unterscheiden ist der Vektor $\vec{p} = (p_x \;\; p_y)^T$, welcher von O abgetragen zu P führt. $\vec{p}$ kann als Linearkombination von $\vec{e}_x$ und $\vec{e}_y$ aufgefaßt werden:

$$\vec{p} = p_x \cdot \vec{e}_x + p_y \cdot \vec{e}_y$$

Die Koordinaten p_x und p_y sind Elemente von $\mathbb{R}$. Die Bildschirmpunkte hingegen sind ganzzahlig. Wir können einen bestimmten Bildschirmpunkt "anschalten", indem wir (bei dem Objekt, das den grafischen Kontext darstellt) die Methode

```
setPixel(int x, int y);
```

mit den entsprechenden ganzzahligen Koordinaten aufrufen.

Sei $P = (2.0, 2.0)$ gegeben. In diesem Fall tut

```
setPixel(2, 2);
```

genau das, was wir erwarten.

Wenn aber $P = (2.3, 3.7)$ gegeben ist, dann müssen die Koordinaten auf diejenigen ganzen Zahlen gerundet werden, die die gewünschten Koordinaten am besten repräsentieren:

```
x = 2.3; y = 3.7;
setPixel((int)(x+0.5), (int)(y+0.5));
```

3.3 Linie

- Gegeben sind Anfangspunkt $P_1 = (x_1, y_1)$ und Endpunkt $P_2 = (x_2, y_2)$ einer Linie l:
- Zu berechnen sind wieder die "anzuschaltenden" Pixel.

3.3.1 Parametrisierte Geradengleichung

Da P_1 und P_2 zwei Punkte auf einer Geraden sind, bietet es sich an, die ganze Linie als Teilstück einer Geraden aufzufassen und durch die Parametrisierung der Geraden mit einem Parameter r die Pixel zu bestimmen.

Gesucht ist der Vektor $\vec{v}$, der von P_1 nach P_2 führt.

Es gilt

$$\begin{aligned} & \vec{p}_1 + \vec{v} &&= \vec{p}_2 \\ \Leftrightarrow \quad & \vec{v} &&= \vec{p}_2 - \vec{p}_1 \\ \Leftrightarrow \quad & \begin{pmatrix} v_x \\ v_y \end{pmatrix} &&= \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} \end{aligned}$$

Die gesamte Gerade g ergibt sich, wenn man von O zu einem Punkt der Geraden geht (z.B. P_1) und von dort ein beliebiges Vielfaches des Richtungsvektors $\vec{v}$ abträgt (Punkt-Richtungsform):

$$g : \vec{u} = \vec{p}_1 + r \cdot \vec{v}; r \in \mathbb{R}$$

Die gesuchte Linie l erhält man, wenn man r auf das Intervall $[0;1]$ beschränkt:

$$l : \vec{u} = \vec{p}_1 + r \cdot \vec{v}; r \in [0;1]$$

Jetzt muß man nur noch entscheiden, in wievielen Schritten r das Intervall $[0;1]$ durchlaufen soll; d.h. wieviele Pixel man "anschalten" will, um die ganze Linie zu repräsentieren.

Eine von P_1 und P_2 unabhänigige Anzahl (z.B. 100 Schritte) würde bei kurzen Linien dazu führen, daß manche Pixel aufgrund der Rundung mehrfach gesetzt würden. Bei langen Linien hingegen wären die Pixel evtl. nicht benachbart.

Sinnvoll wäre es, soviele Pixel zu setzen, wie die Linie Einheiten lang ist. In der euklidischen Ebene ist die Länge d einer Strecke $\overline{P_1P_2}$ als Abstand von Anfangs- und Endpunkt definiert. Die Abstandsberechnung geschieht mit Hilfe der euklidischen Norm (vgl. Pythagoras):

$$d = \|\overline{P_1P_2}\| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Hier die `draw`-Methode der Klasse `line.VectorLine`. x_1, x_2, y_1 und y_2 sind `int`-Variablen, die bei der Instanziierung der Linie übergeben wurden. D.h. die Linie weiß, wo sie beginnt und wo sie aufhört und kann sich selber zeichnen.

```
public void draw(CGCanvas cgc) {
  int x, y, dx, dy;
  double r, step;

  dy = y2 - y1;                                  // Hoehenzuwachs
  dx = x2 - x1;                                  // Schrittweite

  step = 1.0 / Math.sqrt(dx*dx + dy*dy);         // Parameterschritt berechnen

  for(r=0.0; r < 1; r+=step) {                   // fuer jeden Parameterwert
    x = (int)(x1 + r*dx +0.5);                   // berechne neue x-Koordinate
    y = (int)(y1 + r*dy +0.5);                   // berechne neue y-Koordinate
    cgc.setPixel(x,y);                           // setze Pixel
  }
  cgc.setPixel(x2, y2);                          // letztes Pixel am Endpunkt
}
```

Diese Implementation hat den Nachteil, daß sehr viel Gleitkommaarithmetik durchgeführt werden muß. Gleitkommaarithmetik ist (in Java) im Gegensatz zu Integerarithmetik sehr zeitintensiv. Wir werden im Folgenden diesen Nachteil schrittweise beseitigen.

3.3.2 Geradengleichung als Funktion

Eigentlich ist es unnötig für jedes Pixel die neue x-Koordinate **und** die neue y-Koordinate auszurechnen, da man ja der Reihe nach alle Pixel der Linie setzen will. Wenn man also für jedes Pixel z.B. die x-Koordinate um 1 erhöht, braucht man nur noch die zugehörige y-Koordinate zu berechnen. Dazu muß die Geradengleichung aber in Form einer Funktion vorliegen:

$$y(x) = s \cdot x + c$$

Die Steigung s errechnet sich mit dem Steigungsdreieck wie folgt:

$$s = \frac{\text{Höhenzuwachs}}{\text{Schrittweite}} = \frac{y_2 - y_1}{x_2 - x_1}$$

das gilt sowohl für das Dreieck mit P_1 und P_2 als auch für das Dreieck mit P_1 und dem Punkt C, an dem die Gerade die y-Achse schneidet:

$$\begin{aligned} \frac{y_1 - c}{x_2 - 0} &= \frac{y_2 - y_1}{x_2 - x_1} \\ \Leftrightarrow c &= \frac{y_1 \cdot x_2 - y_2 \cdot x_1}{x_2 - x_1} \end{aligned}$$

einsetzen in die Geradengleichung ergibt:

$$y = \frac{y_2 - y_1}{x_2 - x_1} \cdot x + \frac{y_1 \cdot x_2 - y_2 \cdot x_1}{x_2 - x_1}$$

Hier die `draw`-Methode aus der Klasse `line.StraightLine`:

```
public void draw(CGCanvas cgc) {
  int x, y;
  double s, c;

  s = (double)(y2 - y1) / (double)(x2 - x1);   // Steigung berechnen
  c = (double)(y1*x2 - y2*x1) /                // y-Achsenabschnitt
      (double)(x2 - x1);

  x = x1;                                      // Koordinaten retten
  y = y1;

  if(x < x2)  {                                // Linie links -> rechts
    while(x <= x2) {                           // fuer jede x-Koordinate
      y = (int)(s*x + c + 0.5);                // berechne y-Koordinate
      cgc.setPixel(x,y);                       // setze Pixel
      x++;                                     // naechste x-Koordinate
    }
  }
  else {                                       // Linie rechts -> links
    while(x >= x2) {                           // fuer jede x-Koordinate
      y = (int)(s*x + c + 0.5);                // berechne y-Koordinate
      cgc.setPixel(x,y);                       // setze Pixel
      x--;                                     // naechste x-Koordinate
    }
  }
}
```

Diese Version kommt mit ungefähr halb soviel Gleitkommaarithmetik aus, wie die letzte. Allerdings brauchen wir eine zusätzliche Fallunterscheidung, um zu klären, ob die Linie von links nach rechts verläuft oder andersherum. Ein weiterer schwerer Nachteil liegt in der Tatsache, daß die Pixel für steile Geraden nicht benachbart sind.

3.3.3 Bresenham-Algorithmus

Wir werden zunächst den Fall diskutieren, bei dem die Steigung in $[0;1]$ liegt und später verallgemeinern.

Wie oben erwähnt, weicht die gezeichnete Linie für gewöhnlich von der idealen Linie ab:

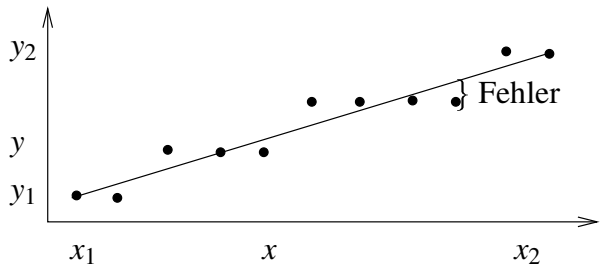


Abbildung 3.1: Fehler als Abweichung von der Ideal-Linie

Die Größe dieser Abweichung läßt sich ausnutzen, um zu entscheiden, ob man für die nächste *x*-Koordinate die aktuelle *y*-Koordinate beibehalten kann oder ob man die *y*-Koordinate um 1 erhöhen muß. Diese Art der Implementation ist aus dem Jahr 1965 und stammt von Jack Bresenham. Der Algorithmus berechnet den jeweils nächsten *y*-Wert aus dem vorherigen und hält dabei den momentanen Fehler nach.

```
public void drawBresenhamLine1(CGCanvas cgc) {
  int x, y, dx, dy;
  double s, error;

  dy    = y2 - y1;                                  // Hoehenzuwachs berechnen
  dx    = x2 - x1;                                  // Schrittweite

  x = x1;                                           // Koordinaten retten
  y = y1;

  error = 0.0;                                      // momentaner Fehler
  s     = (double) dy / (double) dx;                // Steigung

  while (x <= x2) {                                 // fuer jede x-Koordinate
    cgc.setPixel(x, y);                             // setze Pixel
    x++;                                            // naechste x-Koordinate
    error += s;                                     // Fehler aktualisieren
    if (error > 0.5) {                              // naechste Zeile erreicht?
      y++;                                          // neue y-Koordinate
      error-- ;                                     // Fehler anpassen
    }
  }
}
```

Man kann die verbleibende Gleitkommaarithmetik vermeiden, indem man zur Fehlerbestimmung und zur Entscheidung, ob die *y*-Koordinate angepasst werden muß, eine zweite (wesentlich steilere) Gerade verwendet. Die Steigung dieser neuen Geraden berechnet sich folgendermaßen:

$$s_{neu} = s_{alt} \cdot 2dx = \frac{dy}{dx} \cdot 2dx = 2dy$$

Für ein ganzzahliges *s* würde bereits die Multiplikation mit *dx* genügen. Da wir aber auch den Fahler ganzzahlig machen wollen, müssen wir zusätzlich mit 2 multiplizieren:

```
public void drawBresenhamLine2(CGCanvas cgc) {
  int x, y, dx, dy, error, delta;

  dy    = y2 - y1;                                  // Hoehenzuwachs berechnen
  dx    = x2 - x1;                                  // Schrittweite

  x = x1;                                           // Koordinaten retten
  y = y1;

  error = 0;                                        // momentaner Fehler
  delta = 2*dy;                                     // 'Steigung'

  while (x <= x2) {                                 // fuer jede x-Koordinate
    cgc.setPixel(x, y);                             // setze Pixel
    x++;                                            // naechste x-Koordinate
    error += delta;                                 // Fehler aktualisieren
    if (error > dx) {                               // naechste Zeile erreicht?
      y++;                                          // neue y-Koordinate
      error -= 2*dx;                                // Fehler anpassen
    }
  }
}
```

Um nochmals etwas Zeit zu sparen, vergleichen wir `error` mit 0 und verwenden die Abkürzung `schritt` für `-2*dx`:

```
public void drawBresenhamLine3(CGCanvas cgc) {
  int x, y, dx, dy, error, delta, schritt;

  dy    = y2 - y1;                                  // Hoehenzuwachs berechnen
  dx    = x2 - x1;                                  // Schrittweite

  x = x1;                                           // Koordinaten retten
  y = y1;

  error = -dx;                                      // momentaner Fehler
  delta = 2*dy;                                     // 'Steigung'
  schritt = -2*dx;                                  // Fehlerschrittweite

  while (x <= x2) {                                 // fuer jede x-Koordinate
    cgc.setPixel(x, y);                             // setze Pixel
    x++;                                            // naechste x-Koordinate
    error += delta;                                 // Fehler aktualisieren
    if (error > 0) {                                // naechste Zeile erreicht?
      y++;                                          // neue y-Koordinate
      error += schritt;                             // Fehler anpassen
    }
  }
}
```

Geraden in den anderen 7 Oktanten (Steigung $\notin [0;1]$) müssen durch Spiegelung und/oder Vertauschen von x und y auf den 1. Oktanten zurückgeführt werden (siehe folgendes Listing).

Abbildung 3.2: Vom Bresenham-Algorithmus erzeugte Linien

```
public void draw(CGCanvas cgc) {
  int x, y, error, delta, schritt, dx, dy, inc_x, inc_y;

  x = x1;                                          // Koordinaten retten
  y = y1;

  dy = y2 - y1;                                    // Hoehenzuwachs
  dx = x2 - x1;                                    // Schrittweite

  if(dx > 0)                                       // Linie nach rechts?
    inc_x = 1;                                     // x inkrementieren
  else                                             // Linie nach links
    inc_x = -1;                                    // x dekrementieren

  if(dy > 0)                                       // Linie nach unten?
    inc_y = 1;                                     // y inkrementieren
  else                                             // Linie nach oben
    inc_y = -1;                                    // y dekrementieren

  if(Math.abs(dy) < Math.abs(dx)) {                // flach nach oben oder unten
    error = -Math.abs(dx);                         // Fehler bestimmen
    delta = 2*Math.abs(dy);                        // Delta bestimmen
    schritt = 2*error;                             // Schwelle bestimmen
    while(x != x2) {                               // Fuer jede x-Koordinate
      cgc.setPixel(x,y);                           // setze Pixel
      x += inc_x;                                  // naechste x-Koordinate
      error = error + delta;                       // Fehler aktualisieren
      if (error > 0) {                             // neue Spalte erreicht?
        y += inc_y;                                // y-Koord. aktualisieren
        error += schritt;                          // Fehler aktualisieren
      }
    }
  }
  else {                                           // steil nach oben oder unten
    error = -Math.abs(dy);                         // Fehler bestimmen
    delta = 2*Math.abs(dx);                        // Delta bestimmen
    schritt = 2*error;                             // Schwelle bestimmen
    while(y != y2) {                               // fuer jede y-Koordinate
      cgc.setPixel(x,y);                           // setze Pixel
      y += inc_y;                                  // naechste y-Koordinate
      error = error + delta;                       // Fehler aktualisieren
      if (error > 0) {                             // neue Zeile erreicht?
        x += inc_x;                                // x-Koord. aktualisieren
        error += schritt;                          // Fehler aktualisieren
      }
    }
  }
  cgc.setPixel(x2, y2);                            // letztes Pixel hier setzen,
}                                                  // falls (x1==x2) & (y1==y2)
```

Bresenham-Algorithmus

3.3.4 Antialiasing

Eine diagonale Linie verwendet dieselbe Anzahl von Pixeln wie eine horizontale Linie für eine bis zu $\sqrt{2}$mal so lange Strecke. Daher erscheinen horizontale und vertikale Linien kräftiger als diagonale. Dies kann durch Antialiasing-Techniken behoben werden.

Bei gleichbleibender Auflösung (d.h. Pixel pro Zeile/Spalte) kann bei Schirmen mit mehreren Grauwerten pro Pixel die Qualität der Linie gesteigert werden. Hierzu werden die Pixel proportional zur Überlappung mit der Ideallinie geschwärzt.

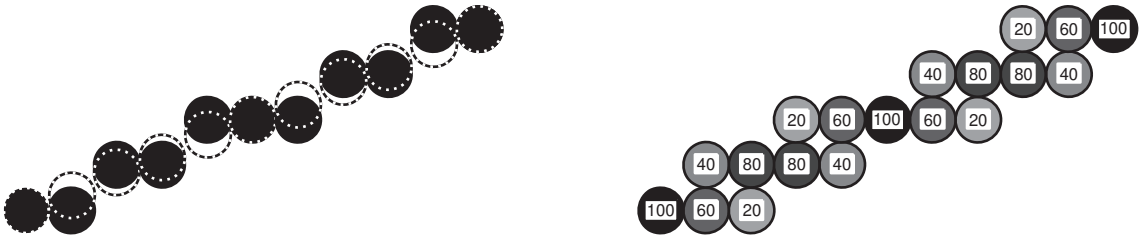


Abbildung 3.3: (a) Bresenham-Linie und Ideal-Linie (b) Resultierende Grauwerte

In Abbildung 3.3 liegt das zweite vom Bresenham-Algorithmus gesetzte Pixel unterhalb der Ideallinie und hat eine Überlappung mit der Ideallinie von 60 %. Daher wird das Bresenham-Pixel mit einem 60 % - Grauwert eingefärbt und das Pixel darüber mit einem 40 % - Grauwert. Das Auge integriert die verschiedenen Helligkeitswerte zu einer saubereren Linie als die reine Treppenform von schwarzen Pixeln.

3.4 Polygon

Ein Polygon wird spezifiziert durch eine Folge von Punkten, die jeweils durch Linien verbunden sind. Anfangs- und Endpunkt sind identisch.

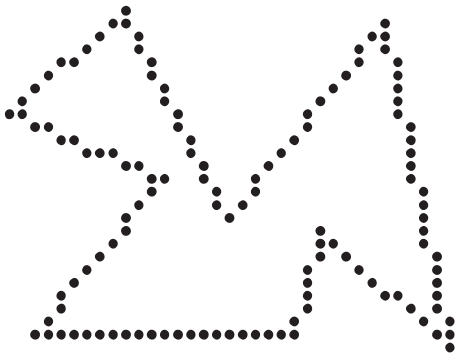
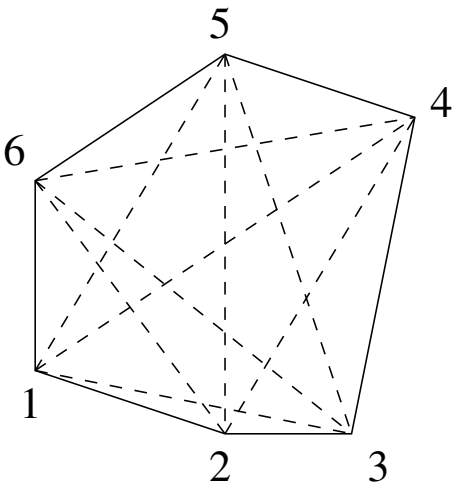


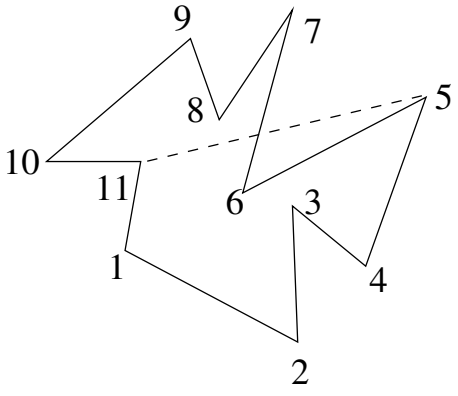
Abbildung 3.4: Polygon

3.4.1 Konvexität

Wir unterscheiden konvexe und konkave Polygone. Bei einem konvexen Polygon ist jeder Eckpunkt von jedem Eckpunkt aus ”sichtbar”. D.h. daß die Verbindungslinie zwischen zwei beliebigen Eckpunkten innerhalb des Polygons verläuft (bei benachbarten Eckpunkten ist diese Verbindungslinie natürlich identisch mit eine Polygonkante). Beim konkaven Polygon schneidet mindestens eine Verbindungslinie mindestens eine Polygonkante:



Konvexes Polygon; alle Verbindungen verlaufen innerhalb des Polygons oder auf dem Rand.



Konkaves Polygon; z.B. die Verbindung zwischen 5 und 11 schneidet die Polygonkanten.

Diese visuelle Bestimmung der Konvexität ist sehr aufwändig zu implementieren. Einfacher ist der Algorithmus von Paul Bourke, dessen Idee so veranschaulicht werden kann: "Wenn man mit dem Fahrrad die Aussenkanten des Polygons entlangfährt und dabei nur nach links oder nur nach rechts lenken muß, ist das Polygon konvex. Wenn man zwischendurch mal die Lenkrichtung wechseln muß, dann ist es konkav. Dabei ist es egal, ob man im Uhrzeigersinn oder gegen den Uhrzeigersinn die Kanten abfährt."

Um festzustellen, ob der nächste Punkt von der aktuellen Polygonkante aus gesehen links oder rechts liegt, benutzen wir die Geradengleichung in der Normalenform, die sich aus der parametrisierten Geradengleichung ergibt, wenn man den Parameter r eliminiert:

$$F(u) = u_x \cdot v_y - u_y \cdot v_x + (v_x \cdot p_y - v_y \cdot p_x)$$

Es gilt:

$$\begin{aligned} F(u) &= 0 \text{ für } u \text{ auf der Geraden.} \\ F(u) &< 0 \text{ für } u \text{ links der Geraden.} \\ F(u) &> 0 \text{ für } u \text{ rechts der Geraden.} \end{aligned}$$

"links" und "rechts" sind dabei in der Richtung von $\vec{v}$ gemeint; d.h. die Seite wechselt, wenn man den Umlaufsinn des Polygons tauscht.

Es muß nur für jedes Punktepaar die Geradengleichung aufgestellt und der darauffolgende Punkt eingesetzt werden. Sobald einmal ein Wert herauskommt, der im Vorzeichen von den bisherigen Werten abweicht, ist das Polygon als konkav identifiziert und es brauchen keine weiteren Punkte mehr getestet zu werden. Wenn sich fuer alle Punkte immer das gleiche Vorzeichen ergibt, dann ist das Polygon konvex.

3.4.2 Schwerpunkt

Der Schwerpunkt (Baryzentrum) S eines Polygons berechnet sich als *baryzentrische Kombination* aus den Eckpunkten P_i des Polygons, versehen mit Gewichten m_i:

$$S = \sum_{i=0}^{n-1} m_i \cdot P_i$$

Für ein Dreieck gilt z.B.:

$$S_D = \frac{1}{3} \cdot p_0 + \frac{1}{3} \cdot p_1 + \frac{1}{3} \cdot p_2$$

Und für ein Viereck gilt:

$$S_V = \frac{1}{4} \cdot p_0 + \frac{1}{4} \cdot p_1 + \frac{1}{4} \cdot p_2 + \frac{1}{4} \cdot p_3$$

Bei baryzentrischen Kombinationen von Punkten gilt immer:

$$\sum_{i=0}^{n-1} m_i = 1$$

Die m_i sind im Allgemeinen verschieden und repräsentieren in der Physik die Massen der beteiligten Massenpunkte P_i.

Baryzentrische Kombinationen sind die einzige Möglichkeit **Punkte** sinnvoll additiv miteinander zu verknüpfen.

3.5 Kreis

- Gegeben seien der Mittelpunkt (x, y) und der Radius r eines Kreises.
- Bestimme die "anzuschaltenden" Pixel für einen Kreis mit Radius r um den Punkt (x, y).

Betrachte zunächst den Verlauf eines Kreises um $(0,0)$ im 2. Oktanten (Abbildung 3.5).

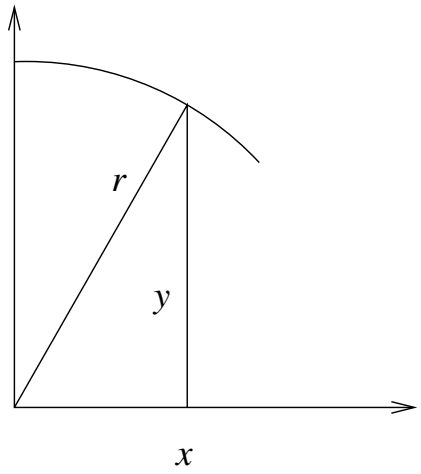


Abbildung 3.5: Verlauf des Kreises im 2. Oktanden

Jim Blinn nennt in "A Trip Down the Graphics Pipeline" 15 verschiedene Arten dies Problem zu lösen. Wir wollen uns 2 etwas genauer ansehen.

3.5.1 Trigonometrische Funktionen

Die *x*- und *y*-Koordinaten ergeben sich sofort, wenn man die trigonometrischen Funktionen Sinus und Cosinus benutzt:

$$x = r \cdot \cos(\alpha); \qquad y = r \cdot \sin(\alpha) \;\; \alpha \in [0; 2\pi]$$

Als Zahl der Schritte für α bietet sich die Zahl der Längeneinheiten des Kreisumfanges an:

$$Winkelschritt = \frac{2\pi}{2\pi \cdot r} = \frac{1}{r}$$

Entsprechend sieht die `draw`-Methode der Klasse `TriCalcCircle.java` aus:

```
public void draw(CGCanvas cgc) {
  double step = 1.0 / (double)r;              // Soviele Winkelschritte
                                              // machen, wie der Umfang
                                              // Einheiten lang ist

  for(double winkel = 0.0; winkel < 2*Math.PI; winkel+=step) {
    cgc.setPixel((int)(x + r*Math.sin(winkel) + 0.5),
                 (int)(y + r*Math.cos(winkel) + 0.5));
  }
}
```

Der ständige Aufruf der trigonometrischen Funktionen kostet sehr viel Zeit. Eine Möglichkeit diesen Aufwand zu reduzieren, besteht darin, eine Tabelle von Sinus- und Cosinus-Werten anzulegen. Dies wird in der Klasse `TriTableCircle` getan:

```
package circle;

public class TriTableCircle extends Circle {
                                                  // Arrays fuer Sinus- und
                                                  // Cosinus-Werte.
  protected final static double[] sin = new double[360];
  protected final static double[] cos = new double[360];
  protected static boolean initialized = false;  // Flagge fuer 1. Instanz

  public TriTableCircle(int x, int y, int r) {
    super(x, y, r);
    if(!initialized) {                            // Falls dies 1. Instanz
      double step = Math.PI / 180;                // Arrays fuellen
      double winkel;

      for(int i=0; i<360; i++) {
        winkel = step * (double)i;
        sin[i] = Math.sin(winkel);
        cos[i] = Math.cos(winkel);
      }
      initialized = true;                         // Fuellung vermerken
    }
  }

  public void draw(CGCanvas cgc) {
    double step = Math.PI / 180;                  // Immer 360 Schritte

    for(int winkel = 0; winkel < 360; winkel++) {
      cgc.setPixel((int)(x + r*sin[winkel] + 0.5),
                   (int)(y + r*cos[winkel] + 0.5));
    }
  }
}
```

Durch die feste Zahl von 360 Schritten für jeden (noch so kleinen/großen) Kreis, geht ein Teil der gewonnenen Zeit wieder verloren bzw. entstehen Pixel, die nicht benachbart sind.

3.5.2 Bresenham-Algorithmus

Für einen Punkt (x,y) sei $F(x,y) = x^2 + y^2 - r^2$

Es gilt:

$$\begin{aligned} F(x,y) &= 0 \text{ für } (x,y) \text{ auf dem Kreis,} \\ F(x,y) &< 0 \text{ für } (x,y) \text{ innerhalb des Kreises.} \\ F(x,y) &> 0 \text{ für } (x,y) \text{ außerhalb des Kreises,} \end{aligned}$$

Verwende F als Entscheidungsvariable dafür, ob y erniedrigt werden muß, wenn x erhöht wird. F wird angewendet auf die Mitte M zwischen Zeile y und $y-1$ (siehe Abbildung 3.6).

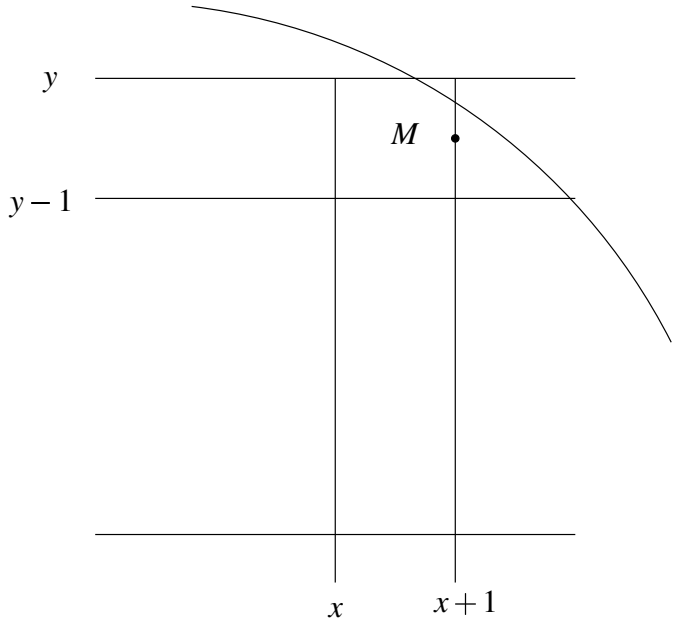


Abbildung 3.6: Testpunkt M für Entscheidungsvariable

$\Delta = F(x+1, y-\frac{1}{2})$

Falls $\Delta < 0 \Rightarrow M$ liegt innerhalb $\Rightarrow (x+1, y)$ ist ideal
$\Delta \geq 0 \Rightarrow M$ liegt außerhalb $\Rightarrow (x+1, y-1)$ ist ideal

Idee: Entscheide anhand von Δ, ob y erniedrigt werden muß oder nicht, und rechne neues Δ aus.

$$
\begin{aligned}
\text{Sei “altes” } \Delta &= F(x+1, y-\tfrac{1}{2}) &&= (x+1)^2 + (y-\frac{1}{2})^2 - r^2 \text{ gegeben.} \\
\text{falls } \Delta &< 0 \Rightarrow \\
\text{“neues” } \Delta' &= F(x+2, y-\tfrac{1}{2}) &&= (x+2)^2 + (y-\frac{1}{2})^2 - r^2 = \Delta + 2x + 3 \\
\text{falls } \Delta &\geq 0 \Rightarrow \\
\text{“neues”} \Delta' &= F(x+2, y-\tfrac{3}{2}) &&= (x+2)^2 + (y-\frac{3}{2})^2 - r^2 = \Delta + 2x - 2y + 5 \\
\text{Startwert für } \Delta &= F(1, r-\tfrac{1}{2}) &&= 1^2 + (r-\frac{1}{2})^2 - r^2 = \frac{5}{4} - r
\end{aligned}
$$

Also ergibt sich:

```
public void drawBresenhamCircle1(CGCanvas cgc) {
  int xh, yh;
  double delta;

  xh = 0;                                          // Koordinaten retten
  yh = r;
  delta = 5.0/4.0 - r;

  while(yh >= xh) {                               // Fuer jede x-Koordinate
    cgc.setPixel(xh, yh);                         // Pixel im 2. Oktanden setzen

    if (delta < 0.0) {                            // Falls noch im Kreis
      delta+=2*xh + 3.0;                          // Abweichung aktualisieren
      xh++;                                       // naechste x-Koordinate
    }
    else {                                        // aus dem Kreis gelaufen
      delta+=2*xh - 2*yh + 5.0;                   // Abweichung aktualisieren
      xh++;                                       // naechste x-Koordinate
      yh--;                                       // naechste y-Koordinate
    }
  }
}
```

Substituiere `delta` durch `d := delta - 1/4`. Dann ergibt sich
als neue Startbedingung: `d := 5/4 - r - 1/4 = 1 - r`
als neue if-Bedingung: `if (d < 0)`
Da `d` nur ganzzahlige Werte annimmt, reicht der Vergleich mit 0.

Weitere Verbesserung:
Ersetze `2xh + 3` durch `dx` mit Initialwert `dx = 3;`
Ersetze `2xh - 2yh + 5` durch `dxy` mit Initialwert `dxy = -2*r + 5`
Es ergibt sich:

```
public void drawBresenhamCircle2(CGCanvas cgc) {
  int xh, yh, d, dx, dxy;

  xh = 0;                                    // Koordinaten retten
  yh = r;
  d = 1 - r;                                 // Startbedingung einstellen
  dx = 3;
  dxy = -2*r + 5;

  while(yh >= xh) {                          // Fuer jede x-Koordinate
    cgc.setPixel(xh, yh);                    // Pixel im 2. Oktanden setzen

    if (d < 0) {                             // Falls noch im Kreis
      d += dx; dx += 2; dxy += 2; xh++;      // Entscheidungsvariablen setzen
    }
    else {
      d +=dxy; dx += 2; dxy += 4; xh++; yh--;// Entscheidungsvariablen setzen
    }
  }
}
```

Um den ganzen Kreis zu zeichnen, wird die Symmetrie zum Mittelpunkt ausgenutzt.

Die Anzahl der erzeugten Punkte des Bresenham-Algorithmus für den vollen Kreis beträgt $4 \cdot \sqrt{2} \cdot r$ Punkte. Verglichen mit dem Kreisumfang von $2 \cdot \pi \cdot r$ liegt dieser Wert um 10% zu tief.

```
public void draw(CGCanvas cgc) {
  int xh, yh, d, dx, dxy;

  xh = 0;                                          // Koordinaten retten
  yh = r;
  d = 1-r;
  dx = 3;
  dxy = -2*r + 5;

  while(yh >= xh) {                               // Fuer jede x-Koordinate
    cgc.setPixel(x+xh, y+yh);                     // alle 8 Oktanden werden
    cgc.setPixel(x+yh, y+xh);                     // gleichzeitig gesetzt
    cgc.setPixel(x+yh, y-xh);
    cgc.setPixel(x+xh, y-yh);
    cgc.setPixel(x-xh, y-yh);
    cgc.setPixel(x-yh, y-xh);
    cgc.setPixel(x-yh, y+xh);
    cgc.setPixel(x-xh, y+yh);

    if (d < 0) {                                    // Falls noch im Kreis
      d+=dx; dx+=2; dxy+=2; xh++;                   // passend aktualisieren
    }
    else {                                          // Aus dem Kreis gelaufen
      d+=dxy; dx+=2; dxy+=4; xh++; yh--;            // passend aktualisieren
    }
  }
}
```

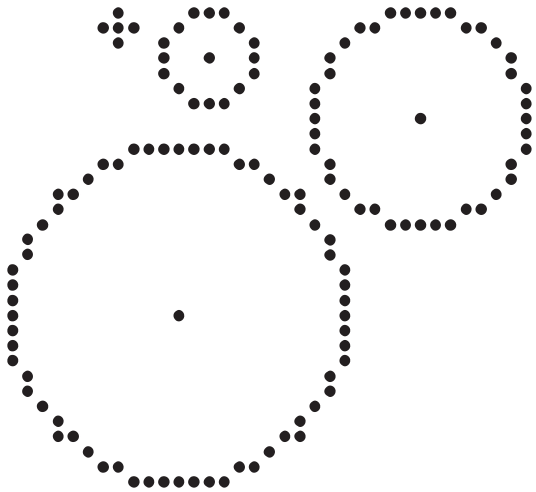


Abbildung 3.7: Vom Bresenham-Algorithmus erzeugte Kreise

Kapitel 4

2D-Füllen

Zum Füllen eines Objekts mit einer einzigen Farbe oder mit einem Muster bieten sich zwei Ansätze an:

- Universelle Verfahren, die die Zusammenhangseigenschaften der Pixel im Inneren der Objekte ausnutzen.
- Scan-Line-Verfahren, die eine geometrische Beschreibung der Begrenzungskurven voraussetzen.

4.1 Universelle Füll-Verfahren

Universelle Füllverfahren stützen sich auf die Nachbarschaft eines Pixels. Abbildung 4.1 zeigt zwei Varianten.

Nachbarn zu • im 4-way-stepping Nachbarn zu • im 8-way-stepping

Abbildung 4.1: Nachbarschaften bei universellen Füllverfahren

Ausgehend vom Startpunkt (x, y) werden so lange die 4-way-stepping-Nachbarn gefärbt, bis die Umgrenzung erreicht ist.

Vorteil: beliebige Umrandung möglich

Nachteil: hoher Rechen- und Speicherbedarf

Obacht:

Gebiete, die nur durch 8-way-stepping erreicht werden können, werden beim Füllen mit 4-way-stepping “vergessen”, wird hingegen die Nachbarschaft über 8-way-stepping definiert, so “läuft die Farbe aus”.

Bild 4.2 zeigt beide Effekte.

Unerreichtes Gebiet beim 4-way-stepping Farbauslauf beim 8-way-stepping

Abbildung 4.2: Probleme bei universellen Füllverfahren

Benötigt werden

```
/** Liefert true, wenn Pixel (x,y) die Vordergrundfarbe hat. */
public boolean getPixel(int x, int y)
```

und

```
/** Liefert true, wenn Pixel (x,y) innerhalb des Bildbereichs */
public boolean rangeOk(int x, int y)
```

```
/** fuellt eine durch Vordergrundfarbe umrandete Flaeche */
public void boundaryFill(int x, int y, CGCanvas cgc) {

  if (cgc.rangeOk(x,y) &&                          // falls Pixel im Bild
      !cgc.getPixel(x,y)) {                        // und bisher nicht gesetzt
    cgc.setPixel(x,y);                             // setze Vordergrundfarbe

    boundary_fill(x+1, y  , cgc);                  // rekursive Aufrufe
    boundary_fill(x,   y+1, cgc);                  // nach Schema des
    boundary_fill(x-1, y  , cgc);                  // 4-Way-Stepping
    boundary_fill(x,   y-1, cgc);
  }
}
```

```
/** leert eine durch Vordergrundfarbe definierten Flaeche */

public void boundaryEmpty(int x, int y, CGCanvas cgc) { // Saatpixel (x,y)

  if(cgc.rangeOk(x, y) &&                          // falls Pixel im Bild und
     cgc.getPixel(x, y)) {                         // in Vordergrundfarbe

    cgc.del_pixel(x, y);                           // setze Hintergrundfarbe

    boundaryEmpty(x+1, y  , cgc);                  // loesche Nachbarpunkte
    boundaryEmpty(x+1, y+1, cgc);                  // nach Schema des
    boundaryEmpty(x,   y+1, cgc);                  // 8-Way-Stepping
    boundaryEmpty(x-1, y+1, cgc);
    boundaryEmpty(x-1, y  , cgc);
    boundaryEmpty(x-1, y-1, cgc);
    boundaryEmpty(x,   y-1, cgc);
    boundaryEmpty(x+1, y-1, cgc);
  }
}
```

Der Nachteil der sehr ineffizienten Methode `boundaryFill` liegt darin, daß für jedes Pixel innerhalb der Begrenzungskurve(n) (mit Ausnahme der Randpixel des inneren Gebiets) der Algorithmus viermal aufgerufen wird. Dadurch werden die Pixel mehrfach auf dem Stapel abgelegt.

Eine Beschleunigung des Verfahrens läßt sich dadurch erreichen, daß auf dem Stapel jeweils Repräsentanten für noch zu füllende Zeilen abgelegt werden, d.h. nach dem Einfärben einer kompletten horizontalen Linie werden von den unmittelbar angrenzenden Zeilen solche Punkte auf den Stapel gelegt, die noch nicht gefüllt worden sind und die unmittelbar links von einer Begrenzung liegen.

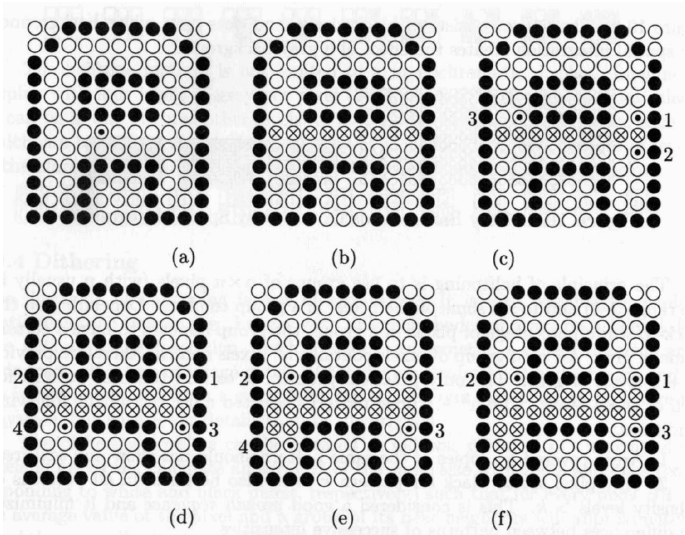


Abbildung 4.3: Beschleunigtes universelles Füllen (⊙: Saatpixel, ○: Pixel in Hintergundfarbe, ●: Randpixel, ⊗: vom Algorithmus gesetztes Pixel).

```
/** fuelle durch Vordergrundfarbe umrandete Fl"ache linienweise */
private void fillRowByRow(int x, int y, CGCanvas cgc) {
  int lg;                                        // nicht gesetzte Pixel ganz
  int rg;                                        // links/rechts in dieser Z.
  Punkt hilf;                                    // Hilfpunkt
  int px = x;                                    // lokale Kopie

  while(!cgc.getPixel(x, y)) {                   // Solange Pixel ungesetzt
    cgc.setPixel(x, y);                          // Pixel setzen
    x--;                                         // naechstes Pixel links
  }
  lg = x+1;                                      // 1 zu weit gelaufen

  x = px + 1;                                    // da (px,y) schon gesetzt

  while(!cgc.getPixel(x, y)) {                   // Solange Pixel ungesetzt
    cgc.setPixel(x, y);                          // Pixel merken und setzen
    x++;                                         // naechstes Pixel rechts
  }
  rg = x-1;                                      // 1 zu weit gelaufen

  for(int pos = rg; pos >= lg; pos--) {          // von rechts nach links
    if(!cgc.getPixel(pos, y-1) &&                // Pixel in Reihe ueber
                                                 // dieser nicht gesetzt und
       cgc.getPixel(pos+1, y-1)) {               // dessen rechter Nachbar
                                                 // ist gesetzt

      fillRowByRow(pos,y-1, cgc);                // Repraesentant! Neuaufruf
    }
    if(!cgc.getPixel(pos, y+1)) &&               // Pixel in Reihe unter
                                                 // dieser nicht gesetzt:
       cgc.getPixel(pos+1, y+1)) {               // dessen rechter Nachbar
                                                 // ist gesetzt
      fillRowByRow(pos,y+1, cgc);                // Repraesentant! Neuaufruf
    }
  }
}
```

4.2 Scan-Line-Verfahren für Polygone

Idee: Bewege eine waagerechte Scan-Line schrittweise von oben nach unten über das Polygon, und berechne die Schnittpunkte der Scan-Line mit dem Polygon.

1. Sortiere alle Kanten nach ihrem größten y-Wert.
2. Bewege die Scan-Line vom größten y-Wert bis zum kleinsten y-Wert.
3. Für jede Position der Scan-Line
   - wird die Liste der aktiven Polygonkanten ermittelt,
   - werden die Schnittpunkte berechnet und nach x-Werten sortiert,
   - werden jene Scan-Line-Segmente, die im Inneren des Polygons liegen, angezeigt.

Abbildung 4.4 zeigt eine Scanline beim Durchqueren eines Polygons. Die Sortierung der Kanten nach ihrem größten y-Wert ergibt die Folge $ABCDEFGHIJ$. Die zur Zeit aktiven Kanten sind $BEFD$. Die sortierten x-Werte der Schnittpunkte $x_1, x_2, \ldots, x_n$ ergeben die zu zeichnenden Segmente $(x_1, x_2), (x_3, x_4), \ldots$

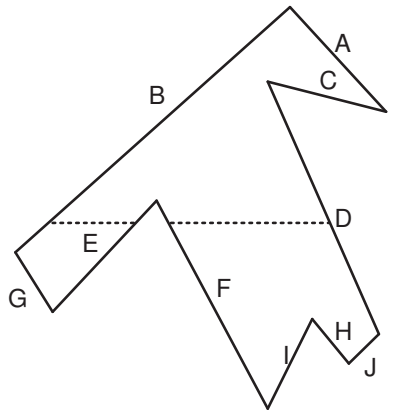


Abbildung 4.4: Polygon mit Scanline

Die Sortierung der Kanten nach ihren größten y-Werten ermöglicht den einfachen Aufbau und die effiziente Aktualisierung einer Liste von aktiven Kanten. Eine Kante wird in diese Liste aufgenommen, wenn der Endpunkt mit dem größeren y-Wert von der Scan-Line überstrichen wird, und wird wieder entfernt, wenn die Scan-Line den anderen Endpunkt überstreicht.

Horizontale Kanten werden nicht in die Kantenliste aufgenommen. Für sie wird eine Linie gezeichnet.

Trifft die Scan-Line auf einen Polygoneckpunkt, dessen Kanten beide oberhalb oder beide unterhalb liegen, so zählt der Schnittpunkt doppelt. Trifft die Scan-Line auf einen Polygoneckpunkt, dessen Kanten oberhalb und unterhalb liegen, so zählt der Schnittpunkt nur einfach (siehe Abbildung 4.5).

Dadurch wird sichergestellt, daß die Paare $(x_1, x_2), (x_3, x_4), \ldots$ der sortierten x-Werte der Schnittpunkte die zu zeichnenden Segmente im Inneren korrekt darstellen.

Kohärenz-Eigenschaft

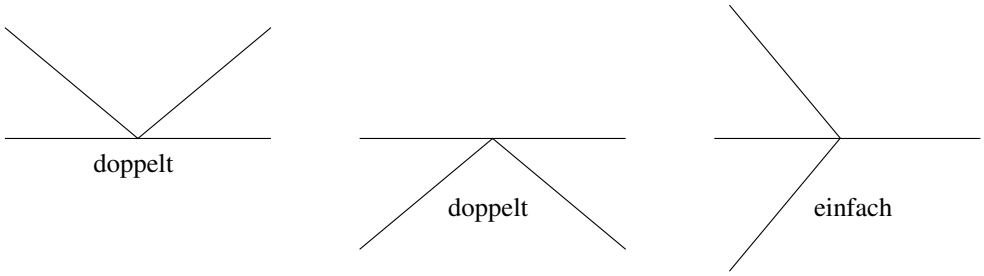


Abbildung 4.5: Fallunterscheidungen beim Berechnen der Schnittpunkte

Abbildung 4.6 zeigt, wie die Schnittpunkte für Scan-Line y_{i+1} sich mit Hilfe der Schnittpunkte von Scan-Line y_i bestimmen lassen.

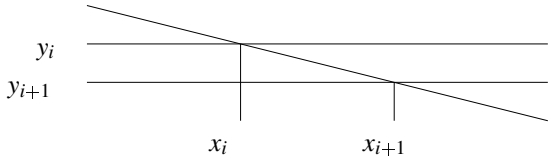


Abbildung 4.6: Fortschreiben der errechneten Schnittpunkte

Es gilt: Die Steigung der Geraden lautet $s = (y_i - y_{i+1})/(x_i - x_{i+1})$.
Wegen $y_i - y_{i+1} = 1$ ergibt sich $x_{i+1} = x_i - 1/s$.

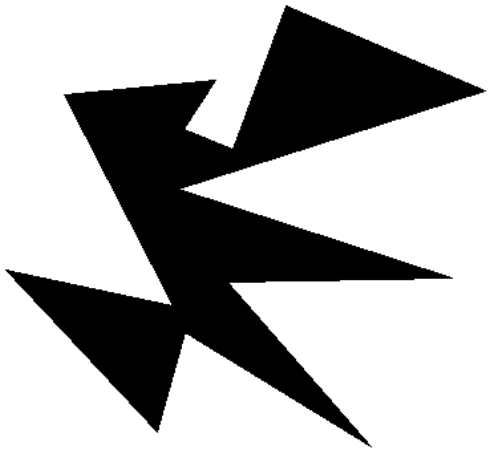


Abbildung 4.7: Vom Scanline-Algorithmus gefülltes Polygon

4.3 Dithering

Wenn einerseits die beiden ”Farben” Schwarz und Weiß zur Färbung der Objekte nicht ausreichen, andererseits der Bildschirm aber nur schwarze Pixel darstellen kann, wird wieder zu einem Trick ge-

griffen, bei dem das menschliche Auge betrogen wird.
Das Auge kann bei normalen Lichtverhältnissen Details unterscheiden, die mindestens eine Bogenminute ($1/60$ Grad) auseinander liegen. Wenn man in eine weiße Fläche eine Anzahl a von nicht benachbarten schwarzen Punkten hineinzeichnet und diese Fläche dann aus grösserer Entfernung betrachtet, dann ist das Auge nicht mehr in der Lage, die einzelnen Punkte voneinander zu unterscheiden. Die schwarzen und weißen Bereiche werden vom Auge integriert und dadurch entsteht der Eindruck einer grauen Fläche. Je größer a ist, desto dunkler wirkt die Fläche. Damit die Fläche gleichmäßig gefärbt erscheint, wird versucht die Punkte möglichst gleichmäßig zu verteilen. Eine Möglichkeit dies zu tun, ist das *Dithering*.

Eine $n \times n$-Dithermatrix D ist mit den n^2 Zahlen zwischen 0 und n^2-1 besetzt. Zum Färben einer Fläche mit Grauwert $k, 0 \leq k \leq n^2$ werden alle Pixel (i,j) gesetzt mit $D[i \mod n, j \mod n] < k$. Abbildung 4.8 zeigt das Füllmuster zum Schwellwert 7.

Die n^2 Zahlen können auf verschiedene Weisen auf die Matrix verteilt werden. Wir werden die Matrix nach dem Schema des *ordered dithering* füllen. Dabei wird die Matrix D_n rekursiv aus der Matrix D_{n-1} gebildet:

$$D_n = \begin{pmatrix} 4 \cdot D_{n-1} & 4 \cdot D_{n-1} + 2 \cdot U_{n-1} \\ 4 \cdot D_{n-1} + 3 \cdot U_{n-1} & 4 \cdot D_{n-1} + U_{n-1} \end{pmatrix}$$

Dabei bezeichnet U_n eine $n \times n$-Einheitsmatrix; d.h. alle Elemente der Hauptdiagonalen enthalten eine 1 und die restlichen Elemente sind 0.

Es ergeben sich:

$$D_0 = (0); \quad D_1 = \begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}; \quad D_2 = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

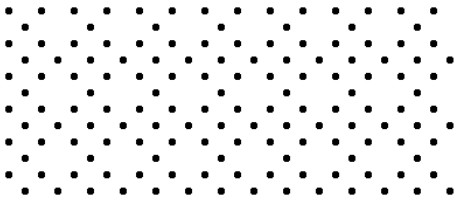


Abbildung 4.8: Zum Schwellwert 7 gehörender Grauwert

```
public class DitherMatrix extends NxNMatrix{

  /**
   * Gibt eine DitherMatrix zurueck, die mit Ditherwerten gefuellt ist.
   * @param n die Matrix ist (2 hoch n)x(2 hoch n) Elemente gross und
   * enthaelt die (2 hoch 2n) Ditherwerte von 0 bis (2 hoch 2n) - 1.
   */
  public DitherMatrix(int n) {
    super((int)Math.pow(2,n), 0.0);              // mit Nullen fuellen

    if(n != 0) {                                 // bei n==0 ist man fertig
      DitherMatrix dd = new DitherMatrix(n-1);   // 1/4 grosse DitherMatrix
                                                 // besorgen und die
      fill(0, 0, 0, dd);                         // Quadranten fuellen: ul
      fill(1, 1, 1, dd);                         // lr
      fill(1, 0, 2, dd);                         // ur
      fill(0, 1, 3, dd);                         // ll
    }
  }

  private void fill(int s, int z, int add, DitherMatrix dd) {
    int size = dd.size();                        // Ausmasse besorgen
    for(int i=0; i < size; i++) {                // passenden Quadranten
      for(int j=0; j < size; j++) {              // fuellen
        val[size*z+i][size*s+j]=4*dd.val[i][j] + add;
      }
    }
  }

  /**
   * Liefert zurueck, ob die Abbildung des uebergebenen Punktes in die
   * Matrix auf ein Matrixelement trifft, das kleiner als die uebergebene
   * Schwelle ist.
   * @param x, y  der zu testende Punkt
   * @param schwelle der Schwellwert
   */
  public boolean kleinerSchwelle(int x, int y, int schwelle) {
    if((x < 0) || (y < 0)) {                     // Wert(e) sinnvoll?
      return false;                              // Falls nicht: Abbruch
    }
    else {                                       // Sonst: Werte mit Modulo
      return (val[x%n][y%n] < schwelle);         // in Matrix abbilden und
    }                                            // Ergebnis zurueck
  }
}
```

4.4 Punkt in Polygon

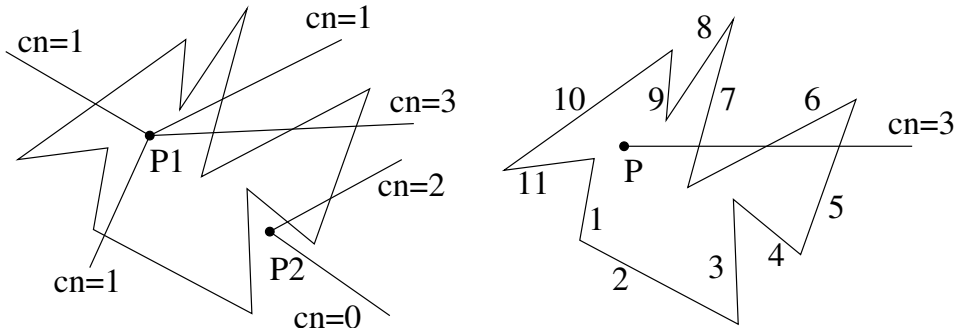
Wenn der Benutzer mit der Maus auf dem Bildschirm in ein Polygon klickt, damit es anschließend gefüllt wird, so muß zunächst festgestellt werden, in welches Polygon geklickt wurde, damit einer der oben beschriebenen Algorithmen mit dem Füllen beginnen kann.

Die Frage, ob ein Punkt in einem Polygon liegt kann auf zwei Arten beantwortet werden. Beide Algorithmen mit gleicher Effizienz implementieren und liefern bei **einfachen** Polygonen (d.h. Polygone, die sich nicht selbst schneiden) dieselben Ergebnisse. Bei Polygonen, die sich selbst teilweise überdecken, weichen die Ergebnisse leicht voneinander ab.

4.4.1 Kreuzungszahl

Nach dem *Jordanschen Kurvensatz* zerlegt jede einfache geschlossene Kurve die 2D-Ebene in genau zwei durch die Kurve verbundene Gebiete: Ein beschränktes Inneres und ein unbeschränktes Äußeres. Deshalb bedeutet eine Kreuzung mit der Kurve einen Wechsel von einem Gebiet in das andere.

Da ein einfaches Polygon eine Kurve im o.g. Sinne ist, gilt entsprechend, daß ein Punkt genau dann im Inneren des Polygons liegt, wenn ein von ihm in eine beliebige Richtung ausgehender Strahl ungeradzahlig viele Polygonkanten kreuzt, also die Kreuzungszahl (oder *Crossing Number*) ungerade ist (Abbildung 4.9 a)). Diese Idee ist auch als *Paritäts-* oder *Gerade-Ungerade-Test* bekannt.



a) Polygon mit zwei Testpunkten. P1 liegt im Polygon (cn gerade); P2 nicht (cn ungerade).

b) Zur effizienteren Berechnung wird der Teststrahl waagerecht nach rechts geschossen.

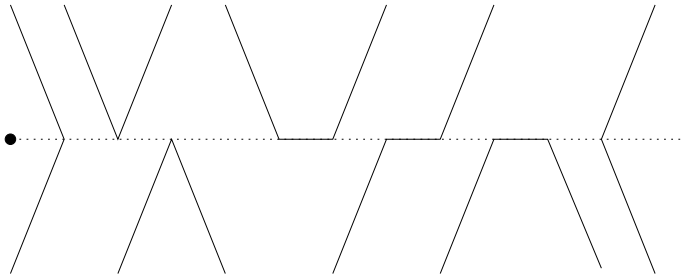
Abbildung 4.9: Ermittlung der Kreuzungszahl (cn) mit einem Teststrahl

Es muß für jede Polygonkante ermittelt werden, ob der Strahl sie schneidet. Wenn ja, wird die Kreuzungszahl um 1 erhöht, sonst nicht.
Dazu wird vom Testpunkt *P* der Strahl waagerecht nach rechts geschossen (Abbildung 4.9 b).

Um den Aufwand für die Schnittpunktberechnung auf diejenigen Kanten zu reduzieren, die den Strahl wirklich schneiden, werden zunächst einige Tests bezüglich der Koordinaten durchgeführt. Wenn Anfangs- und Endpunkt der aktuellen Polygonkante beide oberhalb oder beide unterhalb der Strahls liegen, kann die Kante den Strahl nicht schneiden (z.B. Kanten 1, 2, 3, 4, 8, 9 in Abbildung 4.9 b)).

Wenn ein Punkt oberhalb und der andere unterhalb liegt, kann ein Schnittpunkt vorliegen und die x-Koordinaten werden untersucht. Sind beide Punkte rechts von P, schneidet der Strahl die Kante irgendwo (diese Information ist ausreichend) und die Kreuzungszahl wird erhöht (Kanten 7, 8, 9). Sind beide links von P gibt es keinen Schnittpunkt. Ist einer links und einer rechts von P muß berechnet werden, wo die Kante den y-Wert von P erreicht; d.h. ob der Schnittpunkt rechts von P liegt (z.B. Kante 10). Wenn ja, wird die Kreuzungszahl erhöht.

Ein Problem stellen die bereits beim ScanLine-Verfahren erwähnten Sonderfälle dar, bei denen der Strahl genau einen Polygonpunkt trifft.



Ein Sonderfall liegt immer dann vor, wenn der Strahl einen oder mehrere Polygonpunkte trifft.

Das Problem wird gelöst, indem so getan wird als läge der Polygonpunkt infinitesimal üeber dem Strahl. Bei der Implementation wird dies dadurch erreicht, daß Polygonpunkte mit einem y-Wert $\geq p_y$ als über dem Strahl liegend interpretiert werden. Auf diese Weise wird gleichzeitig die Frage geklärt, zu welchem Polygon ein Pixel gehört, wenn zwei Polygone eine (oder mehrere) identische Kanten haben:

```
public boolean contains(int x, int y) {
  boolean inside = false;

  int x1 = xpoints[npoints-1];
  int y1 = ypoints[npoints-1];
  int x2 = xpoints[0];
  int y2 = ypoints[0];

  boolean startUeber = y1 >= y? true : false;
  for(int i = 1; i<npoints ; i++) {
    boolean endUeber = y2 >= y? true : false;
    if(startUeber != endUeber) {
      if((double)(y*(x2 - x1) -y1*x2 +y2*x1)/(double)(y2-y1) >= x) {
        inside = !inside;
      }
    }
    startUeber = endUeber;
    y1 = y2;
    x1 = x2;
    x2 = xpoints[i];
    y2 = ypoints[i];
  }

  return inside;
}
```

Die Division zur Berechnung des Schnittpunktes kann vermieden werden, wenn man die Steigung der Polygonkante mit der Steigung der Geraden durch P und den Endpunkt der Kante vergleicht und dabei berücksichtigt, ob der Endpunkt eine größere oder eine kleinere y-Koordinate hat.

```
public boolean contains(int x, int y) {
  boolean inside = false;

  int x1 = xpoints[npoints-1];
  int y1 = ypoints[npoints-1];
  int x2 = xpoints[0];
  int y2 = ypoints[0];

  boolean startUeber = y1 >= y? true : false;
  for(int i = 1; i<npoints ; i++) {
    boolean endUeber = y2 >= y? true : false;
    if(startUeber != endUeber) {
      if((y2 - y)*(x2 - x1) <= (y2 - y1)*(x2 - x)) {
        if(endUeber) {
          inside = !inside;
        }
      }
      else {
        if(!endUeber) {
          inside = !inside;
        }
      }
    }

    startUeber = endUeber;
    y1 = y2;
    x1 = x2;
    x2 = xpoints[i];
    y2 = ypoints[i];
  }
  return inside;
}
```

Diese Methode hat den Nachteil, daß sie sich auf Kurven stützt, die außer dem Anfangs- und dem Endpunkt keinen weiteren Punkt zweimal oder öfter berühren. Die Begrenzungskurve von Polygonen, die sich selbst überschneiden, hat mehr als einen Doppelpunkt. Deshalb entstehen mehrere beschränkte ”Innere” und ein unbeschränktes Äußeres. Bei einer Kreuzung mit einer Kante ist also nicht mehr sicher, ob von ”innen” nach ”außen” oder von einem ”Inneren” in ein anderes ”Inneres” gewechselt wird. Abhilfe schafft hier die *Umlaufszahl*.

4.4.2 Umlaufszahl

Bei dieser Methode wird gezählt, wie oft die Begrenzungskurve den Punkt P ganz umläuft. D.h. die Umlaufszahl (oder *Winding Number*) i liegt in $\mathbb{N}$ für Punkte, die im Polygon liegen und ist 0 für Punkte, die nicht im Polygon liegen.

Für jede Polygonkante von P_i nach P_{i+1} bildet man die Vektoren von P nach P_i und von P nach P_{i+1} und berechnet den Winkel Θ_i zwischen diesen beiden Vektoren. Die Umlaufszahl ergibt sich dann zu:

$$wn = \frac{1}{2\pi} \sum_{i=0}^{n-1} \Theta_i$$

Diese Methode ist jedoch mathematisch aufwändig. Es reicht, wenn der Algorithmus zählt, wie oft die Begrenzungskurve an einem bestimmten Punkt vorbeikommt. Es wird also wieder ein Strahl von P waagerecht nach rechts geschossen und wieder wird die Zahl der Kanten ermittelt, die diesen Strahl schneiden. Allerdings wird jetzt die Richtung der Kante berüecksichtigt. Wenn die Kante von unten nach oben läuft, wird die Umlaufszahl um 1 erhöht. Wenn die Kante hingegen von oben nach unten läuft, wird die Umlaufszahl um 1 erniedrigt. Dadurch erhalten Punkte außerhalb des Polygons die Umlaufszahl 0 und innere Punkte eine Umlaufszahl ungleich 0.

```
public boolean contains(int x, int y) {
  int wn = 0;

  int x1 = xpoints[npoints-1];
  int y1 = ypoints[npoints-1];
  int x2 = xpoints[0];
  int y2 = ypoints[0];

  boolean startUeber = y1 >= y? true : false;
  for(int i = 1; i<npoints ; i++) {
    boolean endUeber = y2 >= y? true : false;
    if(startUeber != endUeber) {
      if((y2 - y)*(x2 - x1) <= (y2 - y1)*(x2 - x)) {
        if(endUeber) {
          wn ++;
        }
      }
      else {
        if(!endUeber) {
          wn --;
        }
      }
    }

    startUeber = endUeber;
    y1 = y2;
    x1 = x2;
    x2 = xpoints[i];
    y2 = ypoints[i];
  }
  return wn == 0;
}
```

Kapitel 5

Clipping

Ziel: Nur den Teil eines Objekts darstellen, der innerhalb eines Fensters sichtbar ist.

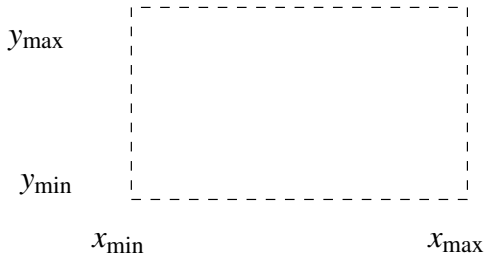


Abbildung 5.1: Clip-Fenster

5.1 Clipping von Linien

Zu einer Menge von Linien sind jeweils neue Anfangs- und Endpunkte zu bestimmen, welche komplett im Clip-Fenster liegen.

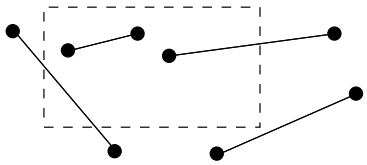


Abbildung 5.2: Ausgangslage beim Linien-Clipping

Idee von Cohen & Sutherland

Teile Ebene anhand des Clip-Fensters in 9 Bereiche ein, beschrieben durch 4-Bit-Bereichscode:

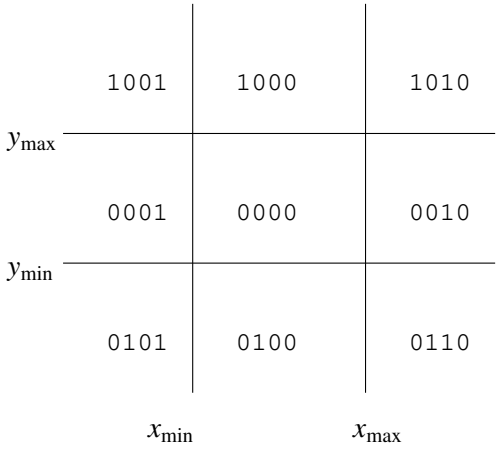


Abbildung 5.3: Bereichscodes nach Cohen & Sutherland

Bit 0: links vom Fenster
Bit 1: rechts vom Fenster
Bit 2: unter dem Fenster
Bit 3: über dem Fenster

Sei $\overline{P_1P_2}$ eine Linie. Dann gilt bei einer bitweisen Verknüpfung:

$code\ (P_1)$ AND $code\ (P_2) \neq 0 \Rightarrow \overline{P_1P_2}$ komplett außerhalb (beide Punkte auf derselben Seite)

$code\ (P_1)$ OR $code\ (P_2) = 0 \Rightarrow \overline{P_1P_2}$ komplett innerhalb (beide Punkte im Clip-Fenster)

In den anderen Fällen wird $\overline{P_1P_2}$ mit einer Fensterkante geschnitten und der Test mit der verkürzten Linie erneut ausgeführt.

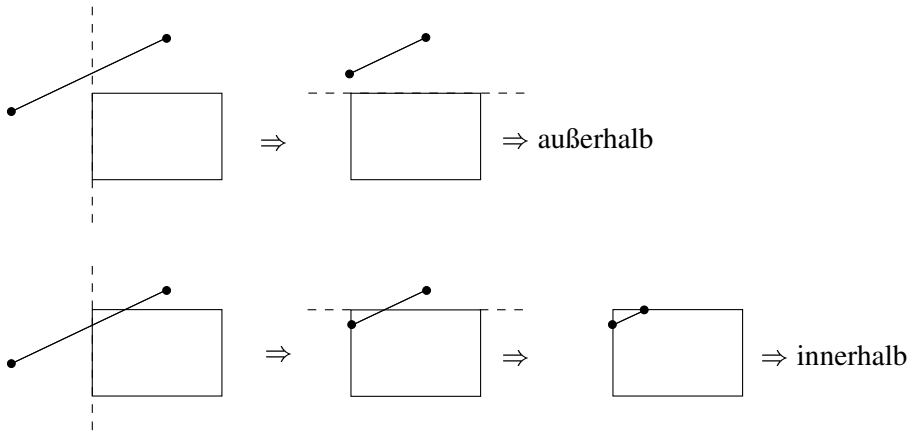


Abbildung 5.4: Möglichkeiten bei zwei außerhalb liegenden Punkten

```
private static final byte CENTER  = 0;
private static final byte LEFT    = 1;
private static final byte RIGHT   = 2;          // 4-Bit-Bereichscodes
private static final byte BOTTOM  = 4;
private static final byte TOP     = 8;

/**
 *  Liefert den region code fuer den uebergebenen Punkt
 *
 *  @param x Die x-Koordinate des zu testenden Punktes
 *  @param y Die y-Koordinate des zu testenden Punktes
 *
 *  @return Der region code in einem Byte kodiert
 */
public byte region_code(int x, int y) {
  byte c = CENTER;                              // zunaechst: im Zentrum

  if (x < xmin)                                 // falls links vom Fenster
    c = LEFT;                                   // links vermerken
  else                                          // falls nicht links
    if (x > xmax)                               // falls rechts vom Fenster
      c = RIGHT;                                // rechts vermerken

  if (y < ymin)                                 // falls oberhalb
    c |= TOP;                                   // oberhalb hinzufuegen
  else                                          // falls nicht oberhalb
    if (y > ymax)                               // falls unterhalb
      c |= BOTTOM;                              // unterhalb hinzufuegen

  return c;                                     // region code zurueck
}
```

5.2 Clipping von Polygonen

Für das Clippen von Polygonen reicht es nicht, jede beteiligte Kante zu clippen:

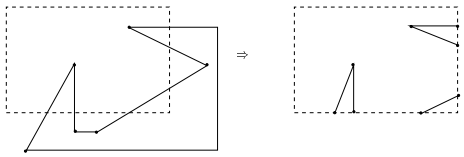


Abbildung 5.5: Zerfall eines Polygons bei reinem Linien-Clippen

Vielmehr müssen zusätzlich die Ein- und Austrittspunkte verbunden werden, um nach dem Clipping wieder ein Polygon zu erhalten:

Obacht: Bzgl. der Ecken des Clip-Windows ist eine Spezialbehandlung erforderlich:

Abbildung 5.6: Verbinden der Ein- und Austrittspunkte

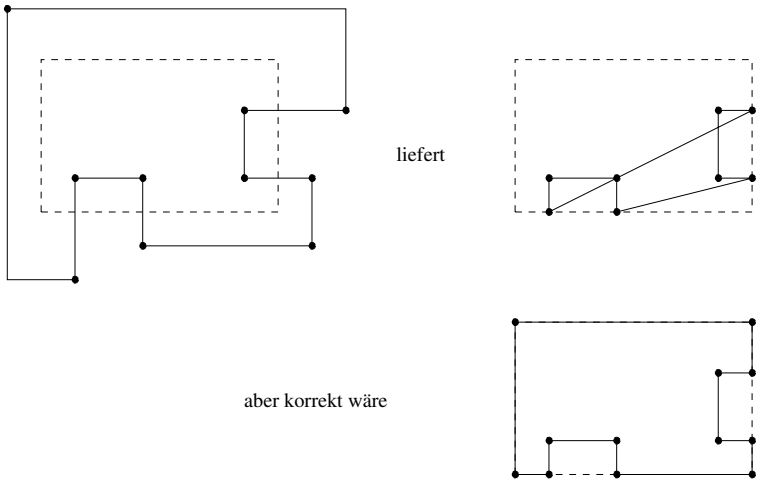


Abbildung 5.7: Spezialbehandlung an den Clipfensterecken

Der **Sutherland & Hodgman-Algorithmus** clippt an vier Fensterkanten nacheinander:

```
fuer jede Clipping-Gerade E tue:
  fuer jeden Polygonpunkt P tue:
    falls P sichtbar: uebernimm ihn
    falls Kante von P zu seinem Nachfolger E schneidet:
          uebernimm den Schnittpunkt
```

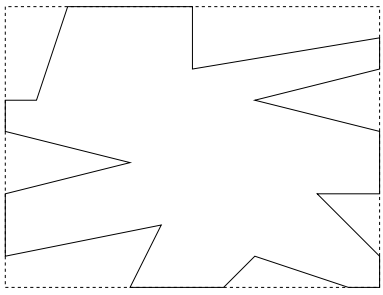


Abbildung 5.8: Vom Sutherland-Hodgman-Algorithmus geclipptes Polygon

5.3 Beispiel-Applet zu 2D-Operationen

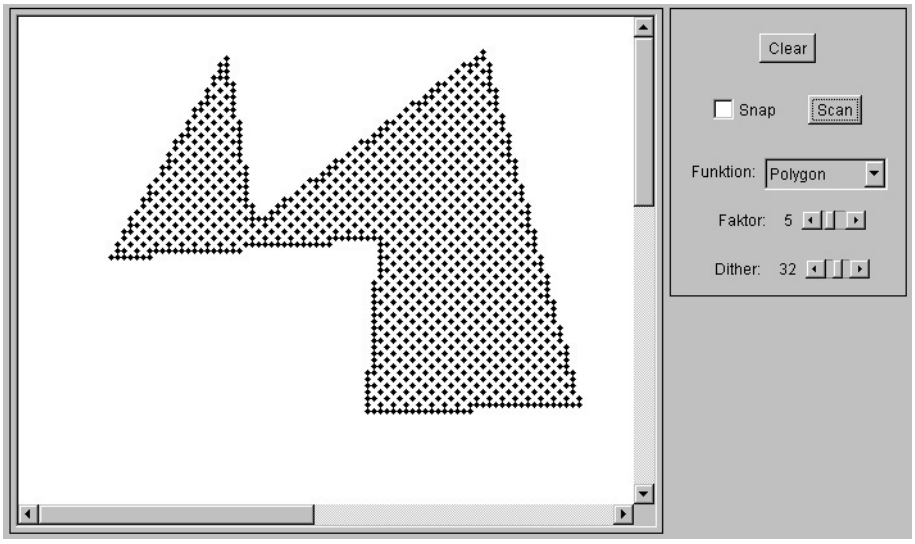


Abbildung 5.9: Screenshot vom 2D-Basic-Applet

Kapitel 6

2D-Transformationen

Mit Hilfe von Transformationen ist es möglich, die Position, die Orientierung, die Form und die Größe der grafischen Objekte zu bestimmen. Transformationen eines Objekts werden durch Operationen auf den Definitionspunkten realisiert. Durch 2 Punkte definierte Rechtecke müssen zunächst zu Polygonen gemacht werden.

6.1 Translation

Gradlinige Verschiebung um einen Translationsvektor $T = (t_x, t_y)$, d.h. $P' := P + T$

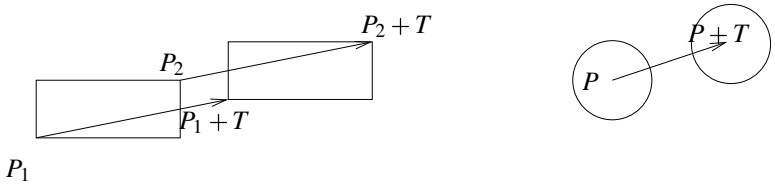


Abbildung 6.1: Translation von Rechtecken und Kreisen

6.2 Skalierung

Vergrößerung bzw. Verkleinerung bzgl. eines Fixpunktes. Zunächst liege der Fixpunkt im Ursprung $(0,0)$.

$$(x', y') \quad := \quad (s_x \cdot x, s_y \cdot y)$$

$s_x = s_y \Rightarrow$ uniforme Skalierung; $s_x \neq s_y \Rightarrow$ Verzerrung

Weder s_x noch s_y dürfen gleich 0 sein. Sonst würde das 2D-Objekt in einer Dimension (oder gar beiden Dimensionen) auf die Ausdehnung 0 zusammengestaucht. Die Objekte sind aber zweidimensional und sollen es im Laufe der Transformationen auch bleiben.

Abbildung 6.2 zeigt ein Beispiel für eine Skalierung bezüglich des Ursprungs mit den Faktoren $s_x = 4$, $s_y = 2$.

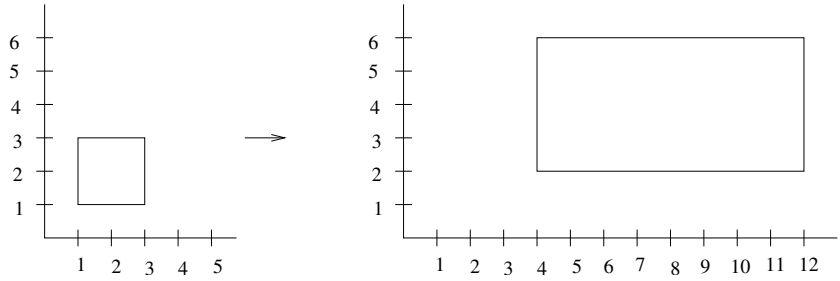


Abbildung 6.2: Skalierung mit $s_x = 4$ und $s_y = 2$

Bei Wahl eines beliebigen Fixpunktes (Z_x, Z_y) folgt für den Punkt P:

1. Translation um $(-Z_x, -Z_y)$ liefert P_1.
2. Skalierung mit (s_x, s_y) liefert P_2.
3. Translation um (Z_x, Z_y) liefert $P_3 = P'$.

Die neuen Koordiaten berechnen sich dann wie folgt:

$$\Rightarrow (x', y') := ((x - Z_x) \cdot s_x + Z_x, (y - Z_y) \cdot s_y + Z_y)$$

Vorteilhafter bei mehreren Objekten:

$$\begin{aligned} (x', y') &= (x \cdot s_x + d_x, y \cdot s_y + d_y) \\ &\quad \text{mit } d_x = Z_x \cdot (1 - s_x), d_y = Z_y \cdot (1 - s_y) \end{aligned}$$

Abbildung 6.3 zeigt ein Beispiel für eine Skalierung bezüglich des Punktes $Z = (1,3)$ mit den Faktoren $s_x = 3$, $s_y = 2$.

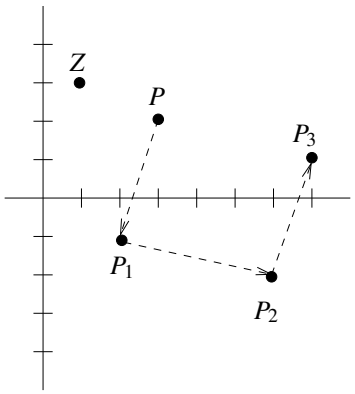
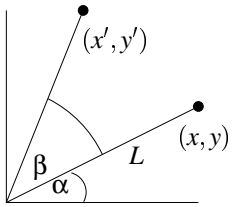


Abbildung 6.3: Skalierung bzgl. Punkt (1,3) mit Faktoren $s_x = 3$ und $s_y = 2$

6.3 Rotation

Drehung des Objekts bzgl. eines Fixpunktes um einen Winkel β. Der Fixpunkt liege im Ursprung.



$$L = \sqrt{x^2 + y^2}$$
$$\sin(\alpha) = y/L$$
$$\cos(\alpha) = x/L$$

Abbildung 6.4: Rotation um den Winkel β bzgl. des Ursprungs

$$\begin{aligned}
\sin(\alpha+\beta) &= \cos(\beta)\cdot\sin(\alpha) + \sin(\beta)\cdot\cos(\alpha) \\
\cos(\alpha+\beta) &= \cos(\beta)\cdot\cos(\alpha) - \sin(\beta)\cdot\sin(\alpha) \\
\sin(\alpha+\beta) = y'/L &= \cos(\beta)\cdot\sin(\alpha) + \sin(\beta)\cdot\cos(\alpha) \\
\Rightarrow y' &= L\cdot\cos(\beta)\cdot\frac{y}{L} + L\cdot\sin(\beta)\cdot\frac{x}{L} \\
\Rightarrow y' &= y\cdot\cos(\beta) + x\cdot\sin(\beta) \\
\cos(\alpha+\beta) = x'/L &= \cos(\beta)\cdot\cos(\alpha) - \sin(\beta)\cdot\sin(\alpha) \\
\Rightarrow x' &= L\cdot\cos(\beta)\cdot\frac{x}{L} - \sin(\beta)\cdot\frac{y}{L}\cdot L \\
\Rightarrow x' &= x\cdot\cos(\beta) - y\cdot\sin(\beta)
\end{aligned}$$

Positive Werte für den Drehwinkel β bewirken eine Rotation gegen den Uhrzeigersinn.
Vorsicht: Auf dem Bildschirm ist es andersherum, da die y-Achse nach unten zeigt.
Bei Wahl eines beliebigen Rotationszentrums (R_x, R_y) folgt für den Punkt P

1. Translation um $(-R_x, -R_y)$ liefert P_1
2. Rotation bzgl. Ursprung um Winkel β liefert P_2
3. Translation um (R_x, R_y) liefert $P_3 = P'$

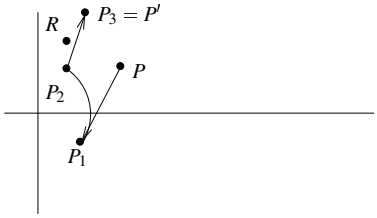


Abbildung 6.5: Rotation des Punktes P um 90 Grad bzgl Punkt R

```
/*****************************************************************************
/*
```

```
/*                      Skalieren und Rotieren eines Polygons
/*                      Obacht: Da die Polygon-Koordinaten ganzzahlig sind,
/*                      verstaerken sich die Rundungsfehler nach jeder Transformation
/*
/*****************************************************************************

private static final double DEG_TO_RAD = Math.PI/180.0; // Umrechnung Grad in Bogenmas


private void skalier_polygon(           // skaliert ein Polygon
                        Point[] points, // gespeichert im Array points
                        int n,          // mit n Eckpunkten
                        double sx,      // in x-Richtung um den Faktor sx
                        double sy,      // in y-Richtung um den Faktor sy
                        Point  z)       // bezueglich des Punktes z
{
        int i;
        for (i=0; i< n; i++) {

                points[i].x = (int)((points[i].x - z.x ) * sx + z.x + 0.5);
                points[i].y = (int)((points[i].y - z.y ) * sy + z.y + 0.5);

        }
}


private void rotate_polygon(            // rotiert ein Polygon
                        Point[] points, // gespeichert im Array points
                        int   n,        // mit n Eckpunkten
                        int   g,        // um einen Winkel von g Grad
                        Point z)        // bezueglich des Punktes z
{
        int i;
double beta = g * DEG_TO_RAD;   // Winkel im Bogenmass
        Point P = new Point();

        for (i=0; i < n; i++) {

                P.x = (int)((points[i].x - z.x ) * Math.cos( beta ) -
                      (points[i].y - z.y ) * Math.sin( beta ) + z.x + 0.5);

                P.y = (int)((points[i].y - z.y ) * Math.cos( beta ) +
                      (points[i].x - z.x ) * Math.sin( beta ) + z.y + 0.5);

                points[i].x = P.x;
                points[i].y = P.y;

        }
}
```

6.4 Matrixdarstellung

Häufig werden mehrere Transformationen hintereinander auf ein Objekt angewendet. Es entstehen Rundungsfehler, wenn nach jeder Einzel-Transformation die ganzzahligen Koordinaten bestimmt werden. Deshalb sollten mehrere Transformationen zu einer zusammengesetzt werden. Dies ist mit Hilfe von Matrizen, die die Transformationen repräsentieren, möglich. Durch Matrizenmultiplikation ($T \cdot P = C$) der Transformationsmatrix T und dem als einspaltige Matrix interpretierten Koordinatenvektor P wird die eigentliche Transformation durchgeführt. Die einzelnen Elemente c_{ik} der Ergebnismatrix C errechnen sich aus den Elementen der Eingangsmatrizen A und B wie folgt:

$$c_{ik} = \sum_{j=0}^{n} a_{ij} \cdot b_{jk}$$

Eine Skalierung sieht dann z.B. folgendermaßen aus:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} := \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \cdot s_x \\ y \cdot s_y \end{pmatrix}$$

Eine Rotation hat folgendes Aussehen:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} := \begin{pmatrix} \cos(\beta) & -\sin(\beta) \\ \sin(\beta) & \cos(\beta) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x \cdot \cos(\beta) - y \cdot \sin(\beta) \\ x \cdot \sin(\beta) + y \cdot \cos(\beta) \end{pmatrix}$$

Die Reihenfolge von Matrix und Vektor kann vertauscht werden. Allerdings müssen dann beide transponiert werden:

$$(P \cdot M)^T = M^T \cdot P^T$$

6.5 Homogene Koordinaten

Um auch die Translation durch eine Matrixmultiplikation ausdrücken zu können, muß das Konzept der *homogenen Koordinaten* eingeführt werden. Dabei wird unserem Objektraum eine Dimension hinzugefügt, es werden aber weiterhin 2D-Objekte repräsentiert und dargestellt.

Ein Punkt $P = (x, y)$ hat die homogenen Koordinaten $(x_h\ y_h\ w)^T$ mit $w \neq 0$ und

$$\begin{aligned} x_h &= x \cdot w \\ y_h &= y \cdot w \end{aligned}$$

Beispiel: Das homogene Koordinatentripel $(6\ 8\ 2)^T$ gehört zum Punkt $P = (3,4)$. Derselbe Punkt hat die homogenen Koordinaten $(3\ 4\ 1)^T$. D.h. jeder Punkt $P = (x, y)$ repräsentiert eine ganze Ursprungsgerade im 3D-Raum.

Der Richtungsvektor $\vec{r}$, der vom Ursprung zum Punkt $R = (x, y)$ führt, hat die homogenen Koordinaten $(x\ y\ 0)^T$.

Die Transformationen Translation, Skalierung und Rotation werden nun als 3×3-Matrizen realisiert. Zusammengesetzte Transformationen ergeben sich durch Matrix-Multiplikation.

Translation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} := \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ 1 \end{pmatrix}$$

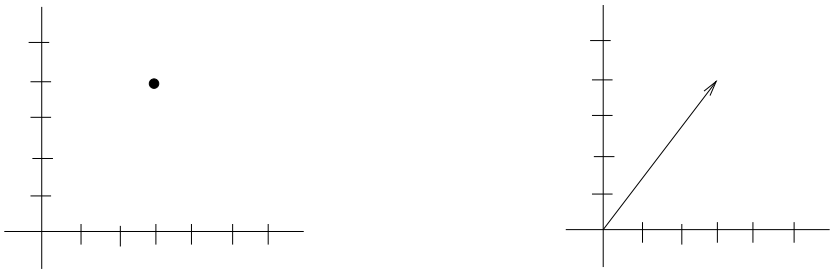


Abbildung 6.6: Punkt $(3,4)$ und Richtungsvektor $(3;4)^T$

Skalierung

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} := \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot s_x \\ y \cdot s_y \\ 1 \end{pmatrix}$$

Rotation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} := \begin{pmatrix} \cos(\beta) & -\sin(\beta) & 0 \\ \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x \cdot \cos(\beta) - y \cdot \sin(\beta) \\ x \cdot \sin(\beta) + y \cdot \cos(\beta) \\ 1 \end{pmatrix}$$

Für die Auswertung einer zusammengesetzten Transformation

- gilt das Assoziativgesetz, d.h. $A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$,
- gilt im allgemeinen nicht das Kommutativgesetz, d.h. $A \cdot B \neq B \cdot A$.

Beispiel: Translation um $(-3,-1)$ mit anschließender Rotation um 90° ist verschieden von Rotation um 90° mit anschließender Translation um $(-3,-1)$.

Beispiel: Rotation bzgl. $(3,1)$ um 60°

Matrix für Translation um $(-3,-1)$ lautet

$$A = \begin{pmatrix} 1 & 0 & -3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Matrix für Rotation um 60° lautet

$$B = \begin{pmatrix} 0.50 & -0.86 & 0.00 \\ 0.86 & 0.50 & 0.00 \\ 0.00 & 0.00 & 1.00 \end{pmatrix}$$

Matrix für Translation um $(3,1)$ lautet

$$C = \begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Matrix für gesamte Transformation lautet

$$D = C \cdot B \cdot A = \begin{pmatrix} 0.50 & -0.86 & 2.36 \\ -0.86 & 0.50 & -2.08 \\ 0 & 0 & 1 \end{pmatrix}$$

6.6 Allgemeine Transformationen

Weitere Transformationen, die sich durch Matrizen darstellen lassen, sind

- Spiegelung an einer beliebigen Geraden,
- Spiegelung an einem Punkt,
- Scherung.

Bei der Scherung in x bleiben die y-Werte konstant, und die x-Werte werden proportional zu den y-Werten horizontal verschoben, d.h. $x' = x + Sch_x \cdot y$.
Die Transformationsmatrix lautet:

$$\begin{pmatrix} 1 & 0 & 0 \\ Sch_x & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ bzw. } \begin{pmatrix} 1 & Sch_y & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

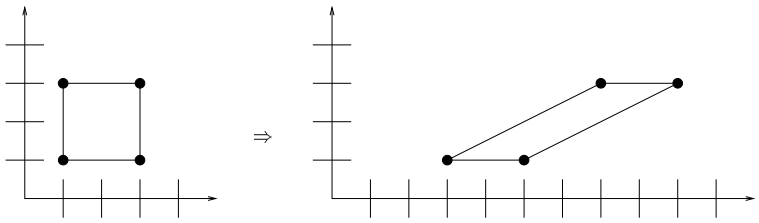


Abbildung 6.7: Scherung mit $Sch_x = 2$

6.7 Raster-Transformationen

Es werden nicht die Definitionspunkte, sondern die Pixel im Frame-Buffer modifiziert:

- Verschieben eines Fensterinhalts (Move),
- Kopieren eines Fensterinhalts (Copy),
- Vergrößern eines Fensterinhalts (Zoom).

Statt Pixelwerte zu setzen, können auch AND-, OR-, XOR-Verknüpfungen verwendet werden. Es gilt $(x \text{ XOR } y) \text{ XOR } y = x$. XOR ist daher geeignet, ein Objekt über den Schirm zu bewegen, ohne den Hintergrund zu zerstören.

```
zeichne Objekt mit XOR
repeat
   loesche Objekt mit XOR
   modifiziere Objekt_Koordinaten
   zeichne Objekt mit XOR
until Objekt am Ziel
```

Allerdings ändert das bewegte Objekt je nach Hintergrundfarbe sein Aussehen.

Kapitel 7

Kurven

Die bsher besprochenen 2D-Objekte haben – bis auf den Kreis – den Nachteil, daß sie im weitesten Sinne ”eckig” sind. Wenn ein Objekt mit ”runder” Form verlangt wird, z.B. ein Herz, ein Schiffsrumpf, ein Kotflügel oder ein Flugzeugflügel, kann entweder ein Polygon mit einer sehr großen Anzahl von Eckpunkten verwendet werden oder eine *Kurve*. Die Kurve hat dabei zwei Vorteile:

- Sie braucht weniger Speicherplatz als das Polygon und
- sie bleibt ”glatt” wenn man sie aus der Nähe betrachtet.

Kurven entstammen dem *Computer Aided Geometric Design (CAGD)*, einer Disziplin, die sich mit der computerinternen Darstellung von Objekten beschäftigt, die aus der Geometrie stammen.

Problem: Spezifiziere und zeichne eine beliebige “glatte” Kurve.

7.1 Algebraischer Ansatz

Gegeben n Stützpunkte.
Bestimme Polynom $(n-1)$-ten Grades

$$y = a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \ldots + a_1 \cdot x + a_0$$

welches durch alle Stützpunkte läuft.
Problem: Bei der Auswertung des Polynoms treten wegen der hohen Potenzen hohe Rechenzeiten und große Rundungsfehler auf. Außerdem oszilliert die Kurve und es kann pro x-Wert nur einen y-Wert geben.

7.2 Kubische Splines

Gegeben n Stützpunkte. Wähle $n-1$ Kurven (Polynome) kleiner Ordnung für den Verlauf innerhalb der $n-1$ Intervalle.

Kurven erster Ordnung (Geraden) ergeben nur den Linienzug. Ein solcher Linienzug ist C^0-stetig, da die einzelnen Linien an den Stützpunkten dieselben x-und y-Werte haben.

Kurven zweiter Ordnung (Parabeln) kann man so wählen, daß sie an den Intervallgrenzen einmal stetig differenzierbar (C^1-stetig) sind. Geometrisch bedeutet das, daß dort nicht nur die Orte übereinstimmen, sondern auch die Steigungen. Mit Parabeln lassen sich nur Kegelschnitte darstellen; die Einsatzmöglichkeiten sind also begrenzt.

Kurven dritter Ordnung mit stetigen ersten und zweiten Ableitungen (C^2-stetig) an den Intervallgrenzen (kubische Splines) können zu einer Gesamtkurve mit weichen Übergängen an den Nahtstellen zusammengesetzt werden. "weich" heißt, daß das Auge der gezeichneten Kurve nicht mehr ansehen kann, wo die Stützpunkte liegen. Neben der Steigung ist auch die Krümmung identisch. Dies sind die einfachsten Kurven, mit denen sich bereits komplexe Formen erzeugen lassen.

Gesucht: Approximation einer Kurve $f(t)$, die durch n vorgegebene Punkte laufen soll (z.B. für Pfadanimation).

Bemerkung: f ist in parametrisierter Form gegeben (d.h. $x = g(t), y = h(t)$), da die explizite Form $y = f(x)$ pro x-Wert nur einen y-Wert erlaubt.

Die n Stützpunkte seien $f(t_i)$, die resultierenden Intervalle seien $[t_i, t_{i+1}]$, die für das Intervall i zuständige Funktion sei $f_i(t)$, $1 \le i \le n-1$.

Jedes f_i hat die Form

$$\begin{aligned} f_i(t) &= f(t_i) + a_i \cdot (t - t_i) + b_i \cdot (t - t_i)^2 + c_i \cdot (t - t_i)^3 \\ & \quad t_i \le t \le t_{i+1},\ i = 1, \ldots, n-1 \end{aligned}$$

Es muß gelten

$$\begin{array}{lcll} f_i(t_{i+1}) & = & f_{i+1}(t_{i+1}) & \text{für } i = 1, \ldots, n-2 \quad \text{gleicher Wert} \\ f_i'(t_{i+1}) & = & f_{i+1}'(t_{i+1}) & \text{für } i = 1, \ldots, n-2 \quad \text{gleiche Steigung} \\ f_i''(t_{i+1}) & = & f_{i+1}''(t_{i+1}) & \text{für } i = 1, \ldots, n-2 \quad \text{gleiche Steigungsänderung} \end{array}$$

Nach Vorgabe von a_1 und a_{n-1} entsteht ein Gleichungssystem mit $n-2$ Gleichungen und $n-2$ Unbekannten $a_2, a_3, \ldots, a_{n-2}$.

Die Werte von a_1 und a_{n-1} können z.B. ermittelt werden durch Festlegung $f''(t_1) = f''(t_n) = 0$ (natürliche Spline-Interpolation).

Für die Folge t_i reicht jede monoton wachsende Folge. Geeignet ist z.B. die Folge der euklidischen Abstände zwischen den Stützpunkten. Statt den Abstand $d = \sqrt{dx^2 + dy^2}$ zu berechnen, verwendet man die Näherungsformel $\frac{1}{3} \cdot (|dx| + |dy| + 2 \cdot \max(|dx|, |dy|))$.

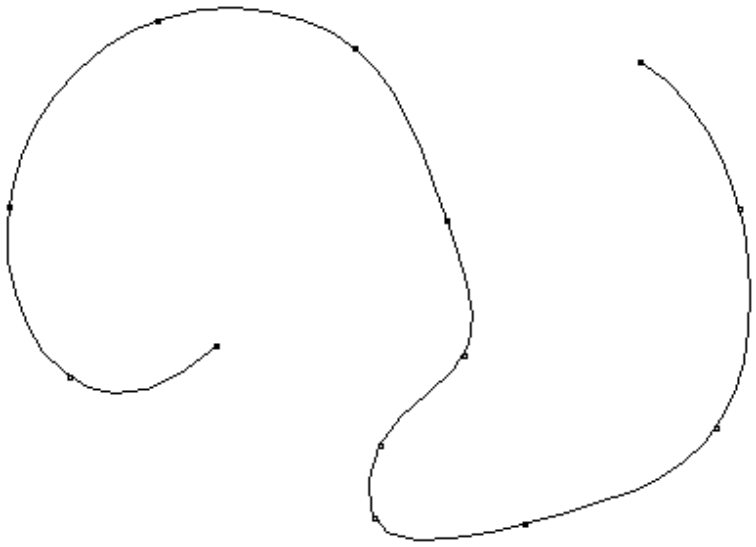


Abbildung 7.1: Vom Spline-Algorithmus erzeugte Kurveninterpolation

```
/*************************************************************************/
/*      Spline-Interpolation durch Polynome 3. Grades                    */
/*************************************************************************/

private static double[] calcfx = new double[MAX_POINTS];// x-Funktionswerte
private static double[] calcfy = new double[MAX_POINTS];// y-Funktionswerte
private static double[] x = new double[MAX_POINTS];  // x-Koord Stuetzpunkte
private static double[] y = new double[MAX_POINTS];  // y-Koord Stuetzpunkte
private static double[] t = new double[MAX_POINTS];  // t-Werte Stuetzpunkte

/** berechnet Intervallgrenzen */
void besetze_arrays(
  int point_cnt,                                  // Anzahl der Polygonpunkte
  Point punkt[])                                  // Punktliste
{
  int i;
  double ax,ay,dd;

  for (i=0; i<point_cnt; i++) {                   // besetze x- und y-Koords
      x[i] = (double) punkt[i+1].x;
      y[i] = (double) punkt[i+1].y;
  }

  t[0] = 0.0;                                     // besetze t-Werte mit dem
  for (i=1; i<point_cnt; i++) {                   // Euklidischen Abstand
      ax=Math.abs(x[i]-x[i-1]);                   // der Stuetzpunke
      ay=Math.abs(y[i]-y[i-1]);
      dd = ax + ay;
```

```
      if (ax > ay)
        dd = dd+2*ax;                              // Euklid-Naeherungsformel
      else
        dd = dd+2*ay;

      t[i] = t[i-1]+dd/3;
  }
}


/** berechnet die Approximation der Kurve */
void curve_fitting(
  int point_cnt,                                   // point_cnt Stuetzpunkte
  Point punkt[],                                   // uebergeben im Array punkt
  int spline_size)                                 // fuer spline_size
                                                   // Interpolationswerte
{
  int i,n=0;
  besetze_arrays(point_cnt, punkt);                // erzeuge Intervallgrenzen
                                                   // Gleitkommaversion von punkt
  // bestimme Interpolationswerte
  cubic_spline(point_cnt, t, x, spline_size, calcfx);
  cubic_spline(point_cnt, t, y, spline_size, calcfy);
  for (i=0; i<spline_size; i++)                    // und uebertrage sie
    spline_punkt[i+1] = new Point((int)calcfx[i],(int)calcfy[i]);
}


/** loest Gleichungssystem zur Bestimmung der Koeffizienten a,b,c */
void cubic_spline(
  int n,                                           // Anzahl der Stuetzpunkte
  double t[], double f[],                          // Stuetzpkte (t[i],f[t[i]])
  int spline_size,                                 // Anzahl Interpolationswerte
  double calcf[])                                  // Ausgabearray fuer die
{                                                  // Interpolationswerte
  int i,j;
  double a[] = new double[MAX_POINTS];
  double b[] = new double[MAX_POINTS];
  double c[] = new double[MAX_POINTS];
  double delta_t[] = new double[MAX_POINTS];
  double D[] = new double[MAX_POINTS];
  double m[] = new double[MAX_POINTS];
  double k[] = new double[MAX_POINTS];
  double bh,dh,e,h,wt,dt;

  for (i=1; i<n; i++) {
    delta_t[i] = t[i]-t[i-1];
    D[i] = (f[i]-f[i-1])/delta_t[i];
  }
  m[0] = delta_t[2];
  delta_t[0] = t[2]-t[0];
  h = delta_t[1];
  k[0] = ((h + 2*delta_t[0])*D[1]*delta_t[2]+h*h*D[2])/delta_t[0];
```

```
  for (i=1; i<(n-1); i++) {
    h = -delta_t[i+1]/m[i-1];
    k[i] = h*k[i-1]+3*(delta_t[i]*D[i+1]+delta_t[i+1]*D[i]);
    m[i] = h *delta_t[i-1] + 2* (delta_t[i] + delta_t[i+1]);
  }

  h = t[n-1]-t[n-3];
  dh = delta_t[n-1];
  k[n-1] = ((dh+h+h)*D[n-1]*delta_t[n-2] +  dh*dh*D[n-2])/h;
  h = -h/m[n-2];
  m[n-1] = delta_t[n-2];
  m[n-1] = h*delta_t[n-2] + m[n-1];
  a[n-1] = (h*k[n-2]+k[n-1])/m[n-1];
  for (i=n-2; i>=0; i--) a[i] = (k[i]-delta_t[i]*a[i+1])/m[i];
  for (i=1; i<n; i++) {
    dh = D[i]; bh = delta_t[i];
    e = a[i-1]+a[i]-dh-dh;
    b[i-1] = 2*(dh-a[i-1]-e)/bh;
    c[i-1] = 6*e/(bh*bh);
  }
  wt = 0; j=0;
  dt = t[n-1]/(double)(spline_size-1);
  for (i=0; i<spline_size; i++) {
    while ((t[j+1]<wt)&&(j<spline_size)) j++;
    h = wt - t[j];
    calcf[i] = f[j]+h*(a[j]+h*(b[j]+h*c[j]/3)/2);
    wt = wt+dt;
  }
  calcf[spline_size-1]= f[n-1];
}
```

7.3 Bézier-Kurven

Spezifiziere die Kurve durch $n+1$ Stützpunkte $P_0, P_1, \ldots, P_n$. Die Kurve läuft nicht durch alle Stützpunkte, sondern wird von ihnen beeinflußt. Zeitgleich in den 1960'ern entwickelt von P. Bézier bei Renault und von P. de Casteljau bei Citroen. Damals sollten Karosserien entworfen werden, die den ästhetischen Ansprüchen der Designer und den technischen Ansprüchen der Ingenieure genügten.

$$P(t) = \sum_{i=0}^{n} B_{i,n}(t) \cdot P_i, \;\; 0 \leq t \leq 1$$

Der Punkt P_i wird gewichtet mit Hilfe eines Bernstein-Polynoms.

$$B_{i,n}(t) = \frac{n!}{i! \cdot (n-i)!} \cdot t^i \cdot (1-t)^{n-i}, \; i = 0, ..., n$$

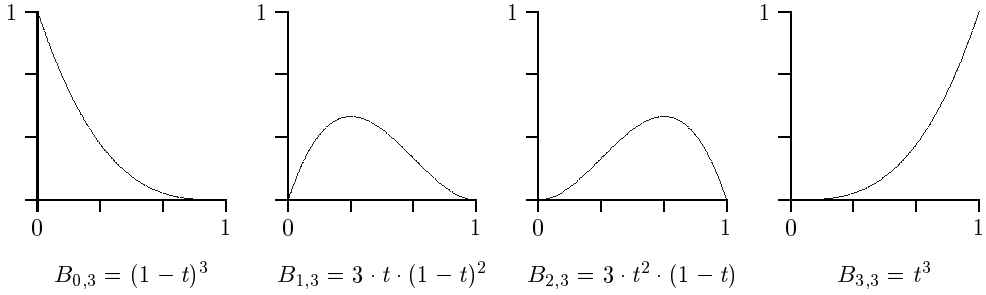


Abbildung 7.2: Verlauf der Bernsteinpolynome für n=3

Die wichtigsten Eigenschaften der Bernsteinpolynome:

- alle Bernsteinpolynome sind positiv auf $[0,1]$,
- für jedes feste t gilt $\sum_{i=0}^{n} B_{i,n}(t) = 1$,
- $B_{i,n}(t) = B_{n-i,n}(1-t)$,
- $\frac{dB_{i,n}(t)}{dt} = n \cdot (B_{i-1,n-1}(t) - B_{i,n-1}(t)), i > 0$,
- Maxima von $B_{i,n}(t)$ in $[0,1]$ an den Stellen $t = \frac{i}{n}, i = 0, \ldots, n$.

Da alle Bernsteinpolynome für $0 < t < 1$ ungleich Null sind, beeinflussen alle Stützpunkte in diesem Intervall den Kurvenverlauf.

Die Kurve beginnt im Stützpunkt P_0 tangential zur Geraden $\overline{P_0P_1}$, endet im Stützpunkt P_n tangential zur Geraden $\overline{P_{n-1}P_n}$ und verläuft innerhalb der konvexen Hülle der Stützpunkte. Somit können mehrere Bézier-Kurven aneinandergesetzt werden durch Identifikation von Endpunkt und Anfangspunkt aufeinanderfolgender Bézierkurven. Einen stetig differenzierbaren Übergang erreicht man bei Kollinearität der Punkte $P_{n-1}, P_n = Q_0, Q_1$.

Berechnung der Bézier-Kurve nach de Casteljau

Um die aufwendige Auswertung der Bernsteinpolynome zu vermeiden, wurde von de Casteljau ein Iterationsverfahren vorgeschlagen, welches durch fortgesetztes Zerteilen von Linien den gewünschten Kurvenpunkt approximieren kann.

Es gilt:

$$P(t)_{0,n} = (1-t) \cdot P(t)_{0,n-1} + t \cdot P(t)_{1,n}$$

wobei die tiefgestellten Indizes angeben, welche Stützpunkte in die Berechnung einfließen. D.h. eine Bézier-Kurve vom Grad n läßt sich durch zwei Bézier-Kurven vom Grad $n-1$ definieren, indem für ein festes t die Punkte beider Kurven berechnet werden, und die Verbindungsstrecke im Verhältnis t geteilt wird.

Beispiel für $t = 1/2$:

$$\begin{array}{cccc} \begin{pmatrix} 0 \\ 0 \end{pmatrix} & & & \\ \begin{pmatrix} 0 \\ 2 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} & & \\ \begin{pmatrix} 8 \\ 2 \end{pmatrix} & \begin{pmatrix} 4 \\ 2 \end{pmatrix} & \begin{pmatrix} 2 \\ \frac{3}{2} \end{pmatrix} & \\ \begin{pmatrix} 4 \\ 0 \end{pmatrix} & \begin{pmatrix} 6 \\ 1 \end{pmatrix} & \begin{pmatrix} 5 \\ \frac{3}{2} \end{pmatrix} & \begin{pmatrix} \frac{7}{2} \\ \frac{3}{2} \end{pmatrix} \end{array}$$

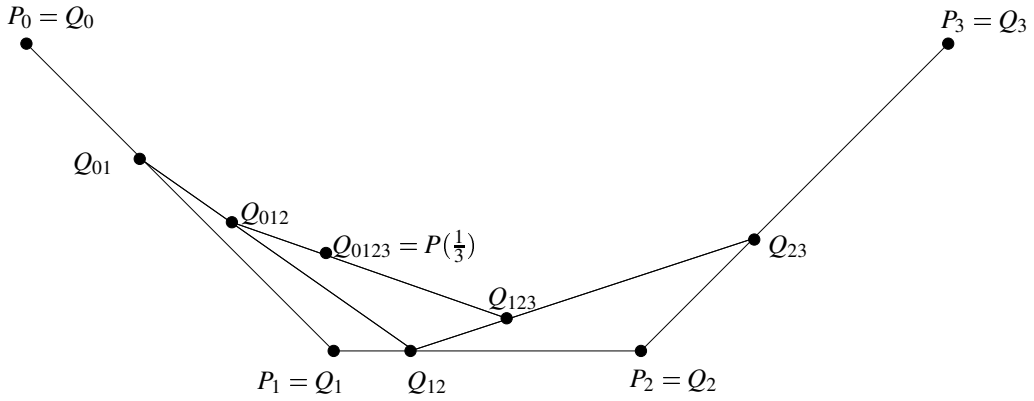


Abbildung 7.3: Approximation einer Bezier-Kurve nach de Casteljau

Abbildung 7.3 zeigt die Berechnung eines Kurvenpunktes für $t = \frac{1}{3}$. Die Indizes geben an, welche Stützpunkte beteiligt sind. Abbildung **??** zeigt das Ergebnis dieser Iteration für zwei aneinanderliegende kubische Bézierkurven (Rekursionstiefe 3).

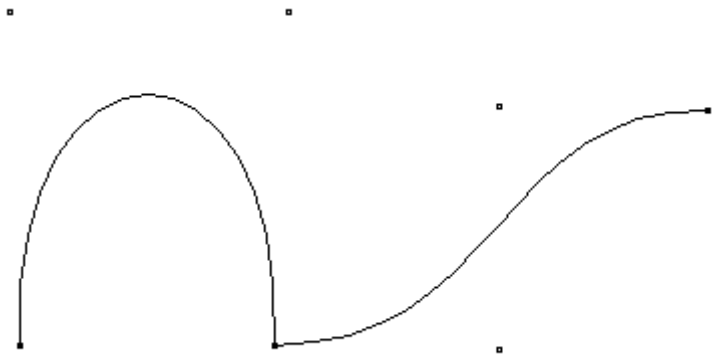


Abbildung 7.4: Vom De Casteljau-Algorithmus mit Rekursionstiefe 3 gezeichnete Bezierkurven

```
/*************************************************************************/
/*                                                                       */
/*        Zeichnen einer Bezierkurve 3. Grades nach De Casteljau         */
/*                                                                       */
/*************************************************************************/
/** Zeichne Bezierkurve 3. Grades */
private void bezier(
  double px0, double py0,                        // mit Stuetzpunkt p0
  double px1, double py1,                        // mit Stuetzpunkt p1
  double px2, double py2,                        // mit Stuetzpunkt p2
  double px3, double py3,                        // mit Stuetzpunkt p3
  int depth)                                     // aktuelle Rekursionstiefe
{
  double qx01,qy01,qx12,qy12,qx23,qy23,          // Hilfspunkte
         qx012,qy012,qx123,qy123,qx0123,qy0123;

  if (depth > iter)                              // Iterationstiefe erreicht
    drawLine(new Point( (int)px0, (int)py0),     // Linie zeichnen
             new Point( (int)px3, (int)py3));
    else {
      depth++;

      qx01   = (px0+px1)/2;     qy01   = (py0+py1)/2;
      qx12   = (px1+px2)/2;     qy12   = (py1+py2)/2;
      qx23   = (px2+px3)/2;     qy23   = (py2+py3)/2;
      qx012  = (qx01+qx12)/2;   qy012  = (qy01+qy12)/2;
      qx123  = (qx12+qx23)/2;   qy123  = (qy12+qy23)/2;
      qx0123 = (qx012+qx123)/2; qy0123 = (qy012+qy123)/2;

      bezier(px0,py0,qx01,qy01,qx012,qy012,qx0123,qy0123,depth);
      bezier(qx0123,qy0123,qx123,qy123,qx23,qy23,px3,py3,depth);
  }
}
```

7.4 B-Splines

Es sollen nun nicht alle Stützpunkte Einfluß auf den gesamten Kurvenverlauf haben und der Grad der Polynome soll unabhängig von der Zahl der Stützpunkte sein.

Spezifiziere die Kurve durch $n+1$ Stützpunkte $P_0,\ldots,P_n$ und einen Knotenvektor
$T=(t_0,t_1,\ldots,t_{n+k}), t_j \le t_{j+1}$.

$$P(t)=\sum_{i=0}^{n} N_{i,k}(t)\cdot P_i$$

Die Punkte P_i wirken sich nur auf maximal k Kurvenabschnitte aus und werden gewichtet durch die Basis des B-Splines, Polynome $N_{i,k}(t)$ vom Grad $k-1$ $(0 \le i \le n)$:

$$N_{i,1}(t) = \begin{cases} 1 & \text{falls } t_i \le t < t_{i+1} \\ 0 & \text{sonst} \end{cases} \tag{7.1}$$

$$N_{i,k}(t) = \frac{t-t_i}{t_{i+k-1}-t_i}\cdot N_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}}\cdot N_{i+1,k-1}(t),\ k>1 \tag{7.2}$$

Bei Division durch Null wird der Quotient gleich 0 gesetzt.
Durch die Wahl von k und T geht jeder Stützpunkt auf einzigartige Weise in die Kurve ein. T kann entweder *uniform* (die t_j sind äquidistant) oder *nicht uniform* gewählt werden. Jede der beiden Arten kann *offen* (Anfang und Ende von T bestehen jeweils aus k-mal dem kleinsten bzw. größten t_j) oder *periodsich* (es ergeben sich periodische Gewichtungspolynome, die durch einfaches Verschieben auseinander hervorgehen) sein.

Der Knotenvektor wird häufig wie folgt gewählt für $0 \le j \le n+k$:

$$t_j = \begin{cases} 0 & \text{falls } j<k \\ j-k+1 & \text{falls } k \le j \le n \\ n-k+2 & \text{falls } j>n \end{cases}$$

Hierdurch wird t im Intervall $[0, n-k+2]$ definiert.

Beispiel: $k=3$, $n=4$ ergibt einen offenen uniformen quadratischen B-Spline mit dem Knotenvektor $T=(0,0,0,1,2,3,3,3)$. Die Stützpunkte P_0,P_1,P_2,P_3,P_4 haben nur lokal Einfluß auf den Kurvenverlauf, und zwar in den Intervallen $t \in [0,1],[0,2],[0,3],[1,3],[2,3]$.

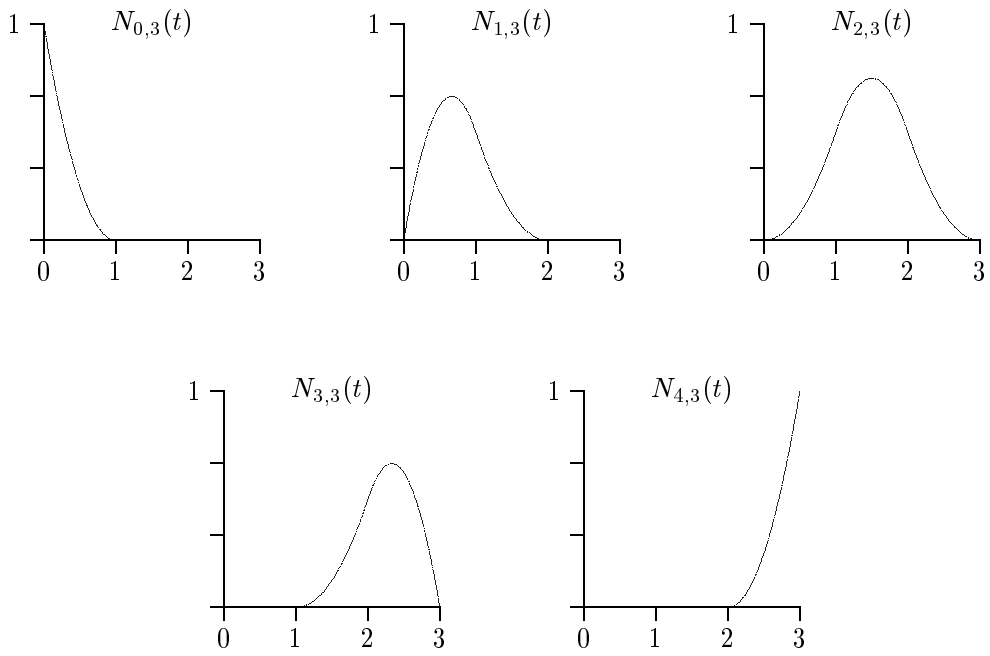


Abbildung 7.5: Verlauf der Gewichtungspolynome $N_{i,k}$ für $k = 3$

Sonderfall

Für $k = n+1$ ergibt sich für den Knotenvektor der Sonderfall

$$T = (\underbrace{0,...,0}_{k \text{ mal}}, \underbrace{1,...,1}_{k \text{ mal}}).$$

Die zugehörigen B-Splinefunktionen haben die schon bekannte Form

$$N_{i,k}(t) = \frac{(k-1)!}{i! \cdot (k-1-i)!} \cdot t^i \cdot (1-t)^{k-1-i},\ i = 0,...,k-1$$

Somit lassen sich also die B-Splinefunktionen als Verallgemeinerung der Bernsteinpolynome auffassen.

Wie bei den Bernsteinpolynomen gilt auch für die B-Splinefunktionen $N_{i,k}$, daß sie positiv sind, und $\sum_{i=0}^{n} N_{i,k} = 1$.

7.4.1 Eigenschaften

B-Splines haben zwei weitere Eigenschaften, die für den praktischen Einsatz von großem Interesse sind:

- Sie verlaufen innerhalb des aus den Stützpunkten gebildeten konvexen Polygons (*konvexe Hülle*.
- Sie sind invariant unter *affinen Abbildungen*.

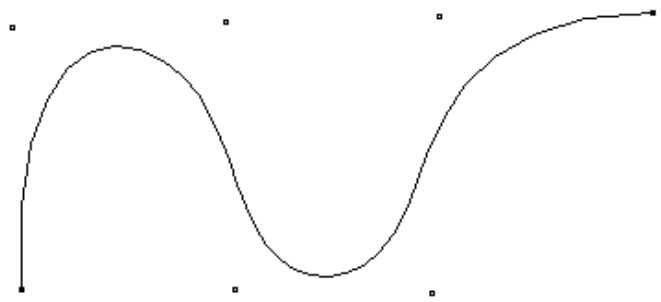


Abbildung 7.6: B-Spline mit $n = 6$, $k = 4$ und 35 Interpolationspunkten

Konvexe Hülle

Die Kurvenpunkte eines B-Splines ergeben sich durch eine baryzentrische Kombination der Stützpunkte. Da die Gewichtungspolynome für alle Parameterwerte t größer oder gleich Null sind, handelt es sich sogar um eine *konvexe Kombination*. Deren Werte liegen innerhalb der konvexen Hülle der Sützpunkte.

Affine Abbildungen und Invarianz

Affine Abbildungen bestehen aus einer linearen Abbildung und einem translativen Anteil:

$$B\vec{x} = A\vec{x} + \vec{t}$$

Es handelt sich um eineindeutig umkehrbare Abbildungen, bei denen

- Geradlinigkeit,
- Zusammengehörigkeit (Objekt wird nicht zerissen),
- Parallelität und
- Teilverhältnisse auf jeder Geraden

erhalten bleiben. Wogegen

- Positionen,
- Längen,
- Winkel,
- Flächeninhalte und
- Orientierungen (Umlaufsinn)

sich ändern können.

Alle bisher in der Vorlseung behandelten Transformationen sind lineare Abbildungen.

Invarianz unter linearen Abbildungen bedeutet in diesem Zusammenhang, daß es keinen Unterschied macht, ob erst die Kurve gezeichnet und dann jeder der gezeichneten Kurvenpunkte abgebildet wird oder ob die Stützpunkte abgebildet werden und die Kurve dann auf Basis diesen neuen Stützpunkte gezeichnet wird.

$$\begin{aligned} B\vec{x} = B(\sum_{i=0}^{n} m_i \vec{p}_i) &= A(\sum_{i=0}^{n} m_i \vec{p}_i) + \vec{t} \\ &= \sum_{i=0}^{n} A m_i \vec{p}_i + \vec{t} \\ &= \sum_{i=0}^{n} m_i A \vec{p}_i + \sum_{i=0}^{n} m_i \vec{t} \\ &= \sum_{i=0}^{n} m_i (A\vec{p}_i + \vec{t}) \\ &= \sum_{i=0}^{n} m_i B \vec{p}_i \end{aligned}$$

Natürlich ist die zweite Variante viel weniger rechenaufwändig. Darüberhinaus kann es (z.B. bei einer uniformen Skalierung mit dem Faktor 2) vorkommen, daß die transformierten Kurvenpunkte auf dem Bildschirm nicht mehr benachbart sind. Ein Effekt, der vermieden wird, wenn erst die Stützpunkte transformiert und dann die Kurvenpunkte berechnet werden.

7.5 NURBS

Über die Eigenschaften der B-Splines hinaus, wäre für den 3D-Fall auch eine Invarianz unter *projektiven Abbildungen* wünschenswert, da dort die 3D-Kurven auf den 2D-Bildschirm projeziert werden müssen.

Um möglichst große Freiheit bei den zu appxroximierenden Formen zu haben, sollten die Kurven beliebige Kegelschnitte (insbesondere Kreise) annähern können.

Beide Wünsche werden von der allgemeinsten Form zum Zeichnen von Kurven erfüllt. Es handelt sich dabei um *NURBS (nonuniform rational basis splines).*

Es handelt sich wieder um eine baryzentrische Kombination:

$$P^h(t) = \sum_{i=0}^{n} N_{i,k}(t) \cdot P_i^h,$$

wobei hier die P_i^h mit homogener Koordinate eingehen ($P_i^h = (x_i\ y_i\ h_i)^T$) und die Punkte der Kurve $P^h(t)$ ggf. eine von 1 verschiedene homogene Koordinate erhalten. Um die Kurve zeichnen zu können, müssen wir sie in die euklidische Ebene projizieren. Dies geschieht durch die Abbildung vom dreidimensionalen Raum der homogenen Koordinaten in die zweidimensionale euklidische Ebene. Da sich

für einen Punkt in homogenen Koordinaten seine Position in der euklidischen Ebene ergibt, wenn man mit der homogenen Koordinate h_i normiert, ergeben sich die Gewichtungsfunktionen als Quotient zweier Polynome. Deswegen heißen diese Kurven *rational*:

$$P(t) = \sum_{i=0}^{n} \frac{h_i \cdot N_{i,k}(t) \cdot P_i}{\sum_{i=0}^{n} h_i \cdot N_{i,k}(t)} = \sum_{i=0}^{n} R_{i,k}(t) \cdot P_i$$

mit $P_i = (x_i\ y_i)^T$ und P(t) zweidimensional und

$$R_{i,k}(t) = \frac{h_i \cdot N_{i,k}(t)}{\sum_{j=0}^{n} h_j \cdot N_{j,k}(t)}.$$

Die h_i werden auch *homogene Gewichte* oder nur *Gewichte* genannt, weil sie zusätzlichen Einfluß auf den Verlauf der Kurve ausüben. Für einen offenen uniformen Knotenvektor $T = [0\ 0\ 0\ 1\ 2\ 3\ 3\ 3]$ mit $n = 4$ und $k = 3$ und Gewichten $h_i = 1 \forall i \neq 3$ ergeben sich die folgenden Kurven, wenn h_3 die Werte 0, 0.25, 1 und 5 annimmt:

Je größer h_3 wird, umso stärker zihet der Punkt P_3 die Kurve an. Wenn h_3 sich Null nähert, wird der Einfluß von P_3 immer schwächer. Für $h_3 = 0$ hat P_3 gar keinen Einfluß mehr. Das bedeutet, daß P_3 mit Berücksichtigung der homogenen Koordinate im Unendlichen liegt.

- NURBS sind eine Verallgemeinerung der B-Splines. Für $h_i = 1 \forall i$ reduziert sich die NURBS-Kurve zur entsprechenden B-Spline-Kurve.
- NURBS sind (im Gegensatz zu nicht-rationalen *B-Splines*) in der Lage, Kreisstücke zu beschreiben.
- Sie sind invariant bzgl. perspektivischer Projektion. Dies bedeutet, daß zur Projektion einer Kurve nicht alle Kurvenpunkte projeziert werden müssen, sondern es reicht, die Stützpunkte zu projezieren und dann zu einer Kurve zu verbinden.

Kapitel 8

Farbe

8.1 Physik

Ein Teil des elektromagnetischen Spektrums wird vom Auge wahrgenommen:

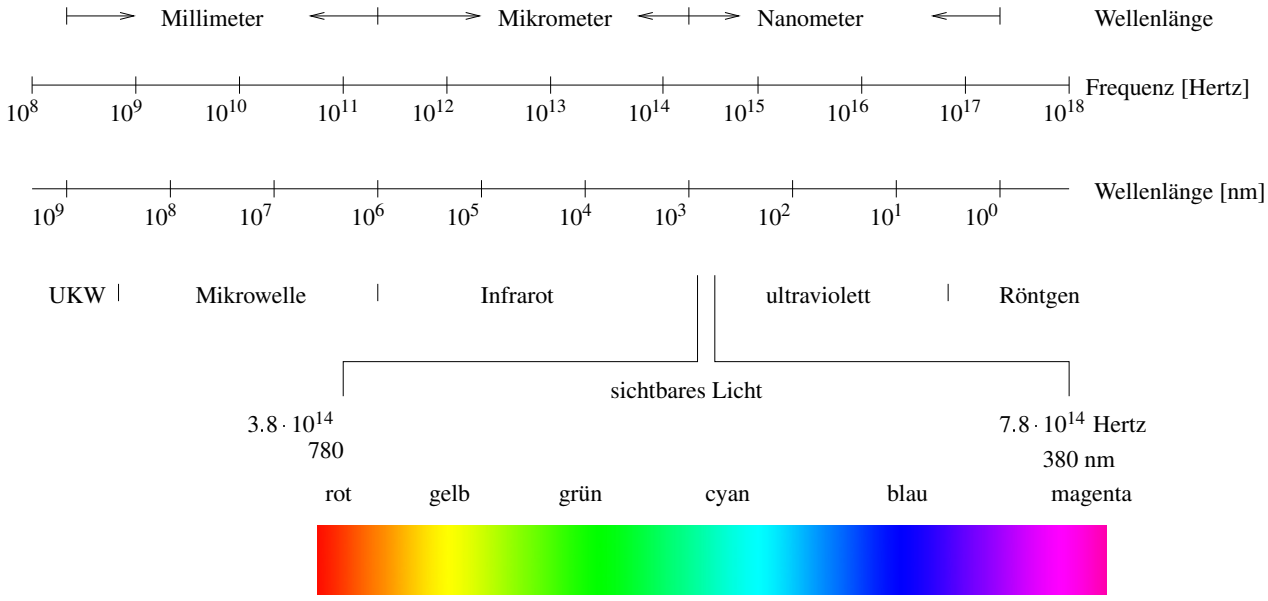


Abbildung 8.1: Elekromagnetisches Spektrum

Es gilt:

- Wellenlänge $\cdot$ Frequenz $=$ Lichtgeschwindigkeit $(= 2.998 \cdot 10^8$ m/s).
- Spektralfarben bestehen aus Licht einer einzigen Wellenlänge.
- In der Natur vorkommende Farben bestehen aus Licht, das aus verschiedenen Wellenlängen zusammengesetzt ist.
- Die Verteilung der Wellenlängen bezeichnet man als Spektrum.

8.2 Dominante Wellenlänge

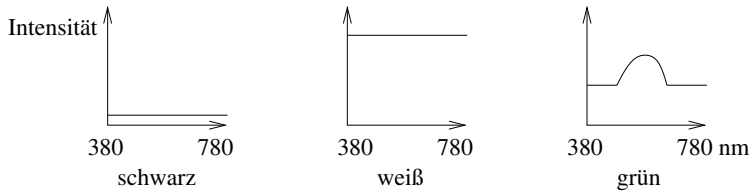


Abbildung 8.2: Spektren zu drei Farben

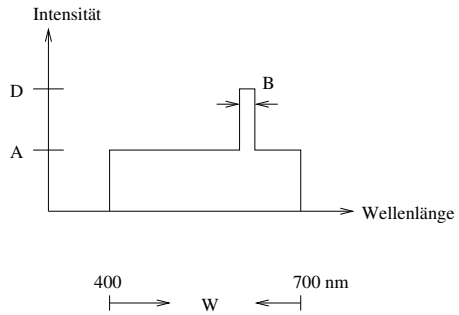


Abbildung 8.3: Dominante Wellenlänge

Eine mögliche intuitive Charakterisierung für den Farbeindruck lautet:

hue: Farbton, gegeben durch dominante Wellenlänge.

luminance: Helligkeit = $L = (D - A) \cdot B + A \cdot W$

saturation: Sättigung, Reinheit, gegeben durch das Verhältnis der Fläche im "Turm" zur Gesamtfläche $= \frac{(D-A) \cdot B}{L}$.

Je weiter A und D auseinanderliegen, desto größer ist die Sättigung, d.h. desto reiner ist die Farbe. Bei $A = 0$ liegt nur die dominante Wellenlänge vor. Bei $A = D$ liegt weißes Licht vor.

Der Mensch kann etwa 128 reine Farbtöne unterscheiden. Pro Farbton können etwa 20 Sättigungsgrade unterschieden werden.

8.3 Grundfarben

Beobachtung: Durch Mischen (= Addieren) von Farben entstehen neue Farben. Wähle 3 Grundfarben, z.B. Rot (R), Grün (G), Blau (B). Bei einer Normierung $R + G + B = 1$ läßt sich jede Kombination durch Angabe von zwei Parametern (siehe Abbildung 8.4) beschreiben (da z.B. B aus R und G folgt).

Abbildung 8.5 zeigt den im Jahre 1931 definierten CIE-Standard (*Commission Internationale l'Éclairage*), in dem drei (künstliche) Grundfarben festgelegt wurden, die alle sichtbaren Farben erzeugen können.

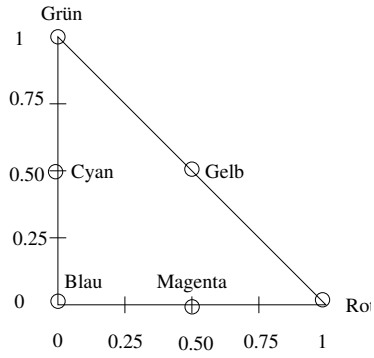


Abbildung 8.4: 2-dimensionale Beschreibung von Farbtönen in baryzentrischen Koordinaten.

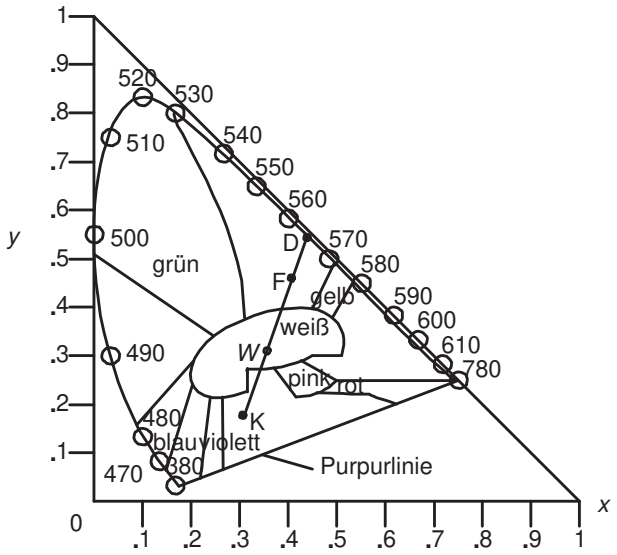


Abbildung 8.5: CIE-Farbdiagramm des sichtbaren Spektrum, Angaben zur Wellenlänge in nm

Die Grundfarben eines typischen Farbbildschirms haben die (x,y)-Koordinaten

$$\begin{aligned} \text{Rot} &= (0.628, 0.346), \\ \text{Grün} &= (0.268, 0.588), \\ \text{Blau} &= (0.150, 0.070). \end{aligned}$$

Die Reinheit der Farbe F ist der relative Abstand von F zu W, bezogen auf die Strecke $\overline{WD}$, wobei D den Schnittpunkt der Geraden durch W und F mit der Kurve bildet. D ist die dominante Wellenlänge in F. Das Komplement K der Farbe F ergibt sich durch Spiegelung von F an W mit entsprechender Skalierung.

8.4 RGB-Modell (Rot, Grün, Blau), (additiv)

Zur Ansteuerung der dreifarbigen Phosporschicht mit roten, grünen, blauen Phosphorpunkten bietet sich das RGB-Modell an. Es handelt sich um ein additives Farbmodell, da das von den Phosphorpunkten ausgehende Licht addiert wird.
Typische Darstellung durch Einheitswürfel:

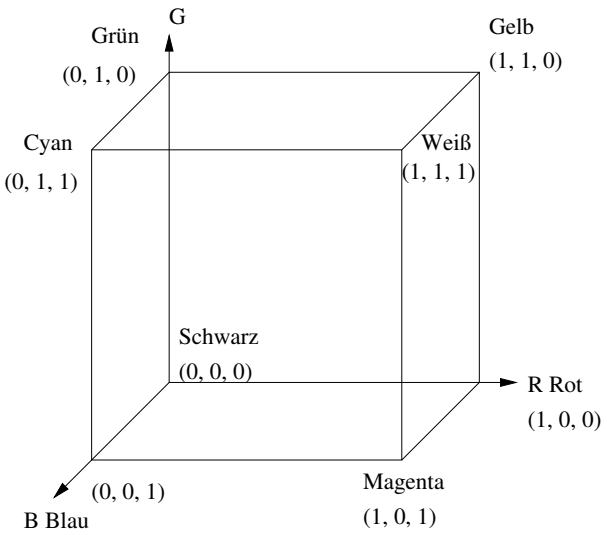


Abbildung 8.6: RGB-Würfel

Häufig werden die drei RGB-Werte statt im reellen Wertebereich $[0,1]$ in 256 Abstufungen als ganze Zahlen im Bereich $\{0,1,...,255\}$ angegeben, die in einem Byte kodiert werden. Dadurch ergibt sich die Darstellung einer Farbe in drei RGB-Bytes.

8.5 CMY-Modell (Cyan, Magenta, Yellow), (subtraktiv)

Bei Farbdrucken empfängt das Auge nur solche Anteile des weißen Lichts, die reflektiert werden. Es bietet sich daher ein subtraktives Modell an.
Ein *CMY*-Tripel beschreibt, wieviel von den Grundfarben Cyan, Magenta, Yellow reflektiert bzw. von den Grundfarben Rot, Grün, Blau absorbiert wird.

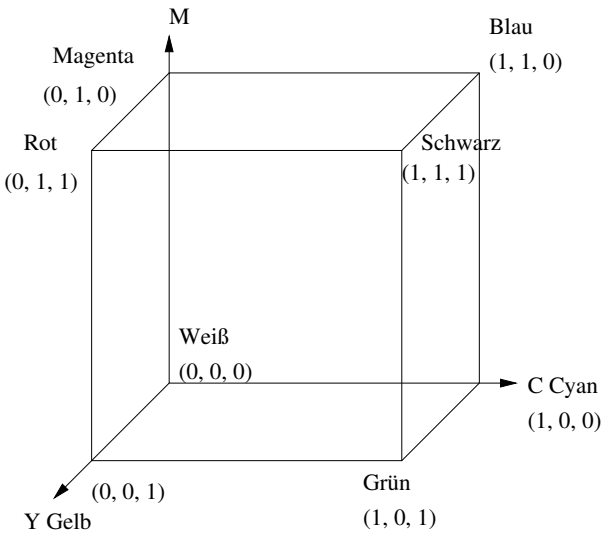


Abbildung 8.7: CMY-Würfel

Es gilt:	$(0,0,0)$	absorbiert nichts	bleibt Weiß
	$(0,0,1)$	absorbiert Blau	bleibt Gelb
	$(0,1,0)$	absorbiert Grün	bleibt Magenta
	$(1,0,0)$	absorbiert Rot	bleibt Cyan
	$(0,1,1)$	absorbiert Cyan	bleibt Rot
	$(1,0,1)$	absorbiert Magenta	bleibt Grün
	$(1,1,0)$	absorbiert Gelb	bleibt Blau
	$(1,1,1)$	absorbiert alles	bleibt Schwarz

Beispiel:	$(0,1,0)$ Magenta	gemischt mit	$(0,0,1)$ Gelb	ergibt	$(0,1,1)$ Rot
	$(1,0,0)$ Cyan	gemischt mit	$(0,0,1)$ Gelb	ergibt	$(1,0,1)$ Grün
	$(1,0,0)$ Cyan	gemischt mit	$(0,1,0)$ Magenta	ergibt	$(1,1,0)$ Blau

Die Umrechnung zwischen dem CMY- und dem RGB-Modell erfolgt in Vektorschreibweise über die Subtraktionen

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} S \\ S \\ S \end{pmatrix} - \begin{pmatrix} C \\ M \\ Y \end{pmatrix} \qquad \text{und} \qquad \begin{pmatrix} C \\ M \\ Y \end{pmatrix} = \begin{pmatrix} W \\ W \\ W \end{pmatrix} - \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

wobei die Vektoren $[S,S,S]$ im CMY-Modell und $[W,W,W]$ im RGB-Modell gleich $[1,1,1]$ sind.

8.6 YUV-Modell

Ein Farbwert wird beschrieben durch ein YUV-Tripel, wobei Y die Helligkeit (Luminanz) bezeichnet und U,V Farbdifferenzen (Chrominanz). Die Helligkeitsempfindung resultiert zu 59 % aus dem

Grünanteil, zu 30 % aus dem Rotanteil und zu 11 % aus dem Blauanteil:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

In den Farbdifferenzen (mit historisch begründeten Normierungsfaktoren) ist die restliche Information codiert:

$$\begin{aligned} U &= 0.493 \cdot (B - Y) \\ V &= 0.877 \cdot (R - Y) \end{aligned}$$

Der Vorteil dieses Farbmodells liegt darin begründet, daß in der Y-Komponente das Bild als Matrix von Grauwerten vorliegt und ggf. separat von der Farbinformation weiterverarbeitet werden kann.

8.7 YIQ-Modell

Beim 1953 in den USA eingeführten *NTSC-System* (National Television Standards Committee) werden die Farben durch die Farbparameter *YIQ* beschrieben, wobei Y die Helligkeit darstellt. Die Umrechnung zum RGB-Modell erfolgt durch die Formeln

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} .299 & .587 & .114 \\ .596 & -.274 & -.322 \\ .211 & -.522 & .311 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0.956 & 0.623 \\ 1 & -0.272 & -0.648 \\ 1 & -1.105 & 1.705 \end{pmatrix} \cdot \begin{pmatrix} Y \\ I \\ Q \end{pmatrix}$$

Beim europäischen *PAL-System* (Phase Alternating Line) werden statt der Parameter I und Q die Farbdifferenzen $R - Y$ und $B - Y$ übertragen.

Die Konvertierung der Farbinformation in Monochrom-Darstellung erfolgt in beiden Systemen durch Auswertung des Helligkeitsparameters Y.

8.8 HSV-Modell

Zur intuitiven Beschreibung einer Farbe eignet sich das HSV-Modell, welches jede Farbe durch ein Tripel

- Hue = Farbton
- Saturation = Sättigung
- Value = Helligkeit

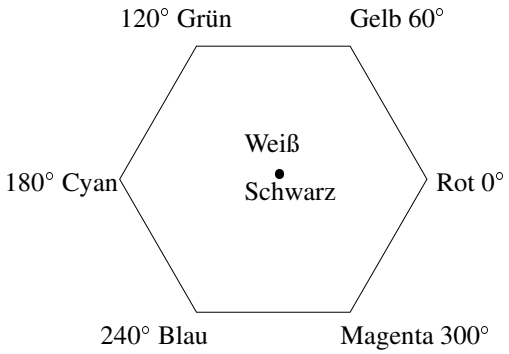


Abbildung 8.8: Gradeinteilung für Farbtöne im HSV-Modell

beschreibt.

Projiziere den *RGB*-Würfel längs der Weiß-Schwarz-Diagonale (Abbildung 8.8).

Dieses Sechseck bildet die Basis einer Pyramide. Die Wahl des Farbtons (hue) geschieht durch Angabe eines Winkels (0° = Rot).

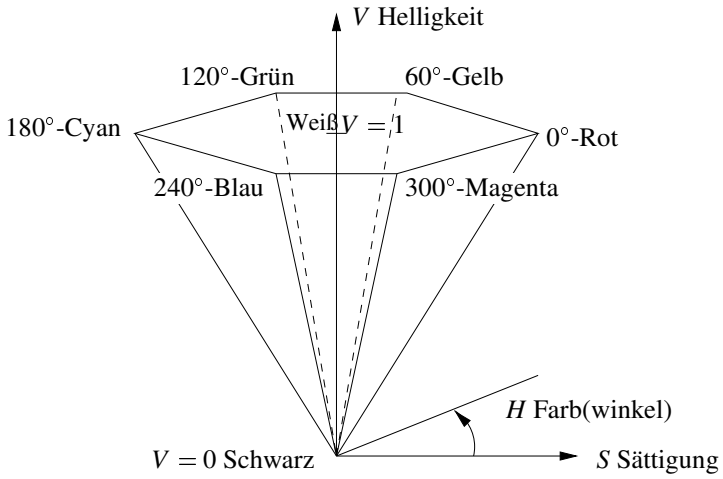


Abbildung 8.9: HSV-Modell

Der Parameter V liegt zwischen 0 und 1 und bestimmt die Intensität der Farbe (dargestellt durch die Senkrechte). Der Parameter S liegt zwischen 0 und 1 und bestimmt die Reinheit der Farbe (Entfernung von der Senkrechten). Die Farbselektion kann erfolgen, indem z.B. zunächst eine reine Farbe ausgewählt wird ($H = \alpha, V = 1, S = 1$). Das Hinzumischen von Weiß zur Erzeugung von Pastellfarben erfolgt durch Reduktion von S bei konstantem V. Das Hinzumischen von Schwarz (zur Erzeugung von dunklen Farben) erfolgt durch Reduktion von V bei konstantem S.

Umrechnung von *RGB* nach *HSV*

Die Achse V entspricht der Diagonalen im *RGB*-Würfel durch die Punkte Schwarz und Weiß, deshalb ist der Wert für V gleich dem Maximum der *RGB*-Intensitäten. Die Werte H und S können aus der Position des Punktes in jenem Sechseck berechnet werden, das durch Projektion des kleinsten, den *RGB*-Punkt beinhaltenden Würfels erzeugt wird.

Beispiel: Welche *HSV*-Darstellung haben die *RGB*-Bytes (64, 128, 32)?
Im *RGB*-Einheitswürfel entspricht dies

$$\left(\frac{1}{4},\frac{1}{2},\frac{1}{8}\right).$$

$$\begin{array}{lcllll} v & = & \max(r,g,b) & = & \frac{1}{2} & = & 50\,\% \\ mi & := & \min(r,g,b) & = & \frac{1}{8} & & \\ s & = & \dfrac{v-mi}{v} & = & \frac{\frac{3}{8}}{\frac{1}{2}} = \frac{3}{4} & = & 75\,\% \end{array}$$

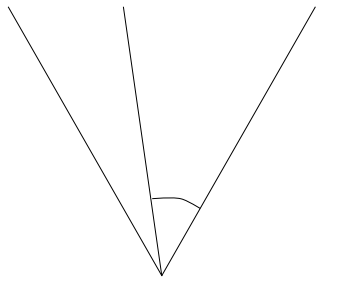
Die dominante Grundfarbe ist Grün, da $v = g$.
Am schwächsten ist Blau vertreten, da $mi = b$.
$\Rightarrow$ Farbe im Bereich Gelb … Grün
$\Rightarrow h = 60^\circ \ldots 120^\circ$.

$$h = \left(1+\frac{v-r}{v-mi}\right)\cdot 60^\circ = \left(1+\frac{\frac{1}{4}}{\frac{3}{8}}\right)\cdot 60^\circ = \left(1+\frac{2}{3}\right)\cdot 60^\circ = 100^\circ$$

Umrechnung von *HSV* nach *RGB*

Beispiel: Wie lauten die *RGB*-Bytes (Werte zwischen 0 und 255) für den Farbton 100° bei 75 % Sättigung und 50 % Helligkeit?

Farbton =	Grün	?	Gelb
h =	120°	100°	60°
RGB =	(0,1,0)	?	(1,1,0)



$$\begin{array}{lcccccc} f & = & \text{Winkel/60} & - & \text{Winkel div 60} & & \\ & = & \frac{5}{3} & - & 1 & = & \frac{2}{3} \end{array}$$

		R	G	B		R	G	B	
Farbton h	:	$1-f$	1	0	$\Rightarrow$	$\frac{1}{3}$	1	0	für $f=\frac{2}{3}$
Sättigung s	:	$1-s\cdot f$	1	$1-s$	$\Rightarrow$	$1-\frac{3}{4}\cdot\frac{2}{3}=\frac{1}{2}$	1	$\frac{1}{4}$	für $s=\frac{3}{4}$
Helligkeit v	:	$v\cdot(1-s\cdot f)$	v	$v\cdot(1-s)$	$\Rightarrow$	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{8}$	für $v=\frac{1}{2}$
Lösung:						64	128	32	

8.9 Color Table (Farbtabelle)

In einem Digitalrechner erwartet der Video-Controller die Farbinformation als Byte-Tripel. Aus Platzgründen können nicht immer für jedes Bildschirmpixel 3 Bytes gespeichert werden. Statt dessen speichert man pro Pixel einen Index, bestehend aus p Bits, mit dem man ein *RGB*-Byte-Tripel aus der Color Table referieren kann.

Beispiel: $p = 2$, d.h. $2^2 = 4$ Farben gleichzeitig darstellbar.

Index	R	G	B	
0	255	0	0	Rot
1	0	255	0	Grün
2	0	0	255	Blau
3	255	255	0	Gelb

Die Color Table kann verwendet werden, um mehrere Bilder dynamisch zu überlagern und wieder zu trennen, ohne die Bilder neu zu zeichnen.

Beispiel: In ein farbiges Bild mit den 8 Farben Schwarz, Grau, Rot, Gelb, Grün, Cyan, Blau, Magenta soll eine Grafik, bestehend aus weißen Linien, ein- und ausgeblendet werden. Pro Pixel seien 4 Bits als Index in der Color Table vorgesehen: Das vorderste Bit beschreibt die weiße Grafik, die hinteren 3 Bits beschreiben die Farbe.

Index	*R*	*G*	*B*	
0000	255	255	255	
0001	255	255	255	
0010	.	.	.	
0011	.	.	.	
0100	.	.	.	
0101	.	.	.	
0110	.	.	.	
0111	255	255	255	
1000	0	0	0	Schwarz
1001	150	150	150	Grau
1010	255	0	0	Rot
1011	255	255	0	Gelb
1100	0	255	0	Grün
1101	0	255	255	Cyan
1110	0	0	255	Blau
1111	255	0	255	Magenta

Offenbar werden alle Pixel mit vorderstem Bit= 0 weiß gezeichnet, alle Pixel mit vorderstem Bit= 1 erhalten die zugewiesene Farbe. Kopiert man nun die untere Hälfte der Color Table in die obere Hälfte, so verschwindet die weiße Grafik. Setzt man die obere Hälfte der Color Table auf Weiß, erscheint die Grafik wieder. Der Aufwand hängt nur von der Größe der Farbtabelle ab, nicht von der Größe der Bilder.
Die Color Table kann auch zur Animation verwendet werden.

Beispiel: Ein (beliebig komplexes) Objekt der Farbe Grün soll sich zyklisch über 15 disjunkte Positionen bewegen. Beschreibe die Szene wie folgt:
Alle Pixel des Objekts in Lage *i* erhalten den Index *i*.
Alle Pixel des Hintergrunds erhalten den Index 15.
Die Color Table ist für alle Einträge mit der Hintergrundfarbe geladen. Das Objekt taucht bei Position *i* auf durch `color_table[i] :=` Objektfarbe. Es verschwindet von Position *i* durch `color_table[i] :=` Hintergrundfarbe.

Bei sich überlagernden Objekten wird für jede Position eine Bit-Plane benötigt: Alle Pixel des Objekts in Lage *i* setzen das *i*-te Bit in ihrem Index auf 1. Das Objekt wird bei Position *i* sichtbar, indem für Indizes mit *i*-tem Bit = 1 die Objektfarbe gesetzt wird.

Die Anzahl der Bit-Planes entspricht der Anzahl der Bits im Index.

Darstellung eines True-Color-Bildes auf einem 8-Bit-Farbschirm

Es wird eine Farbtabelle initialisiert, die einem $6 \times 6 \times 6$ RGB-Würfel entspricht. D.h. auf insgesamt 216 Einträge verteilt befindet sich das Farbspektrum mit je 6 verschiedenen Rot-, Grün- und Blau-Abstufungen. Jedem RGB-Tripel des True-Color-Bildes wird der Index des nächstgelegenen Farbeintrags zugeordnet. Hierzu wird der Bereich $0 \ldots 255$ in 6 Intervalle partitioniert, denen ein quantisierter Wert zugeordnet ist:

x	0 ... 25	26 ... 76	77... 127	128 ... 178	179 ... 229	230 ... 255
$q(x)$	0	51	102	153	204	255

$$q(x) := \left\lfloor \frac{x+25}{51} \right\rfloor \cdot 51$$

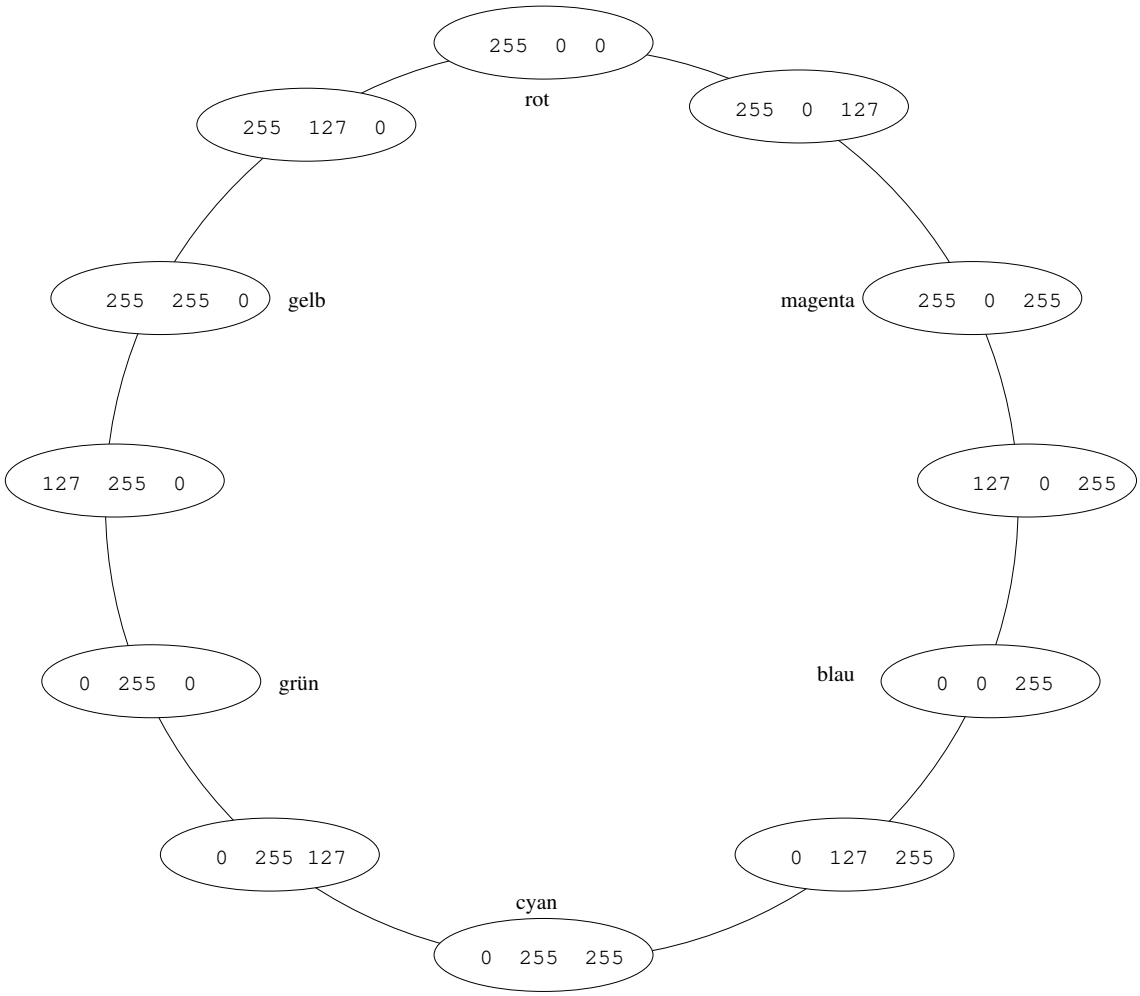


Abbildung 8.10: 12 typische Eintrage einer Farbtabelle fur einen Bildschirm mit 4 Bit Farbtiefe

Hi-Color-Modus

Beim Hi-Color-Modus werden die Rot-, Grün-, Blauanteile eines Pixels durch den Grafikkartentreiber auf $5+6+5=16$ Bit gerundet, d.h., die letzten 3 bzw. 2 Bit werden abgeschnitten. Der resultierende 2 Byte lange Index wird im Bildspeicher abgelegt. Beim Auslesen läßt sich daraus unmittelbar ein Spannungstripel erzeugen, eine Farbtabelle wird nicht benötigt. Es sind also $2^{16} = 65536$ Farben gleichzeitig darstellbar.

Beim Einlesen eines Palettenbildes wird jedem Bildpunkt der 2-Byte Index zugeordnet, der sich nach Rundung des referierten Farbtabelleneintrags ergibt. Danach wird die Farbtabelle nicht mehr benötigt.

8.10 Erzeugung einer bildbezogenen Farbtabelle

Gegeben sei ein Bild mit einer Menge F von beobachteten Farben. Es sei $n = |F|$die Anzahl der verschiedenen Farben in dem Bild. Wenn der Videocontroller zwar alle n Farben darstellen kann, aus Platzgründen aber dennoch eine Farbtabelle benötigt wird, so ist es sinnvoll die Farben so zu wählen, daß sie das gegebene Bild möglichst gut repräsentieren.

Zunächst wird für jede Farbe die Häufigkeit ihres Auftretens ermittelt. Ggf. muß "vorquantisiert" werden von 24 Bit auf 15 Bit (je 5 Bit für Rot, Grün, Blau); hierdurch erhält das Histogramm maximal 32768 Einträge. Sei $d(a,b)$ der Abstand zweier Farbtupel a,b, z.B.

$$\sqrt{(a_{Rot}-b_{Rot})^2+(a_{Gruen}-b_{Gruen})^2+(a_{Blau}-b_{Blau})^2}.$$

Optimaler Algorithmus

Gegeben sei eine Menge F von beobachteten Farben. Gesucht ist eine Menge M von Repräsentanten, so daß der Ausdruck

$$\Delta := \max_{x\in F} \min_{p\in M} d(p,x)$$

minimiert wird, d.h., Δ ist der maximale Abstand, den eine Farbe zu ihrem nächsten Repräsentanten hat.

Da die exakte Bestimmung von M eine exponentielle Laufzeit verursacht, begnügt man sich mit einer Näherungslösung.

Popularity-Algorithmus (1978)

```
Wähle die K häufigsten Farben.
```

Nachteil: Selten vorkommende Farben werden schlecht repräsentiert.

Diversity-Algorithmus (*xv*, John Bradley, 1989)

```
Initialisiere M mit der häufigsten Farbe
for i := 2 to K do
      erweitere M um die Farbe des Histogramms,
      die zu allen Farben aus M den größten Abstand hat
end
```

Median-Cut (Heckbert, MIT 1980)

```
Ziel: Jeder Repräsentant soll möglichst gleich viele Farben repräsentieren. Ini
Initialisiere Wurzel des Schnittbaums mit Gesamtzahl der Pixel
While noch_nicht_genügend_Blätter do
      Wähle Blatt mit der größten Pixelzahl
      Bestimme umschliessende Box
```

```
      Bestimme Achse mit größtem Wertebereich
      Durchlaufe Box längs dieser Achse
      Teile am Median in zwei Hälften
      Trage Hälften als Soehne ein
end
Für jedes Blatt wähle den Mittelwert aller in ihm liegenden Farben.
```

Wenn das Bild n verschiedene Farben enthält und diese durch K verschiedene Farben repräsentiert werden sollen, dann liegt die Laufzeit in $O(n \log K)$.

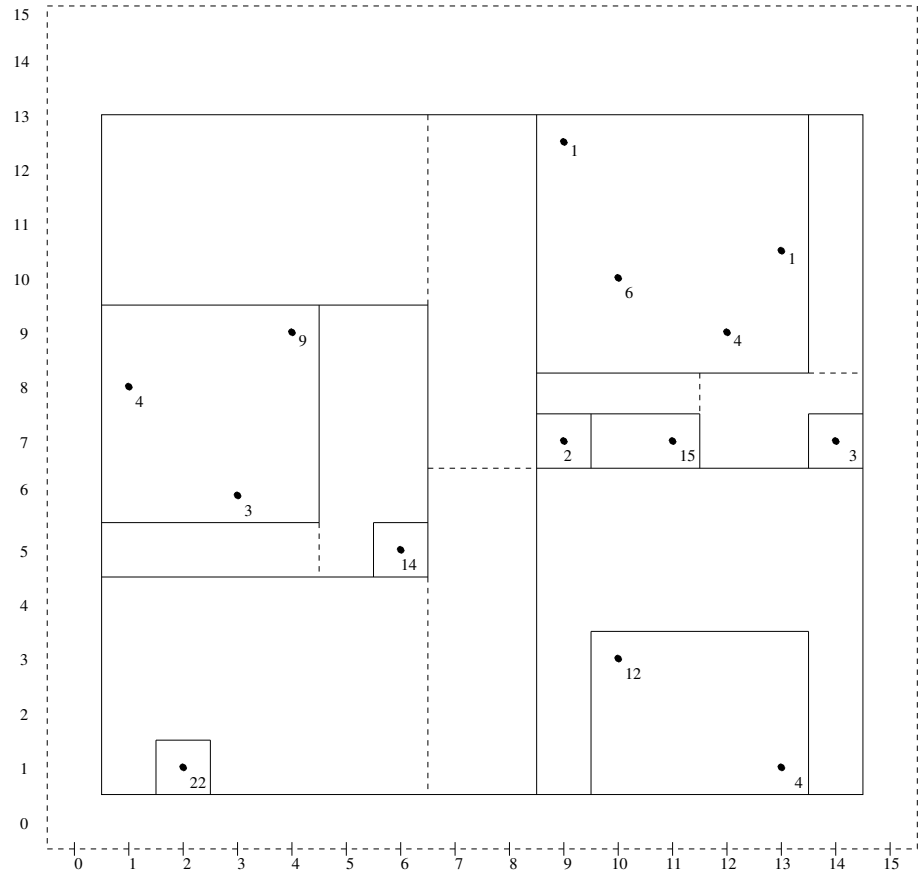


Abbildung 8.11: Partitionierung der Ebene beim Median-Cut-Algorithmus

Abbildung 8.11 zeigt die Anwendung des Median-Cut-Algorithmus anhand eines Beispiels. Zur einfachen Darstellung ist der Farbbaum zweidimensional gewählt. Gezeigt wird die Verteilung der Farbtupel aus dem Bereich $[0\ldots15]\times[0\ldots15]$ in einem Bild mit $10\times10=100$ Pixeln. Eingetragen sind an der Position x/y die Häufigkeit des Farbtupels x,y. Schnittlinien sind gestrichelt, umschließende Boxen durchgezogen gezeichnet.

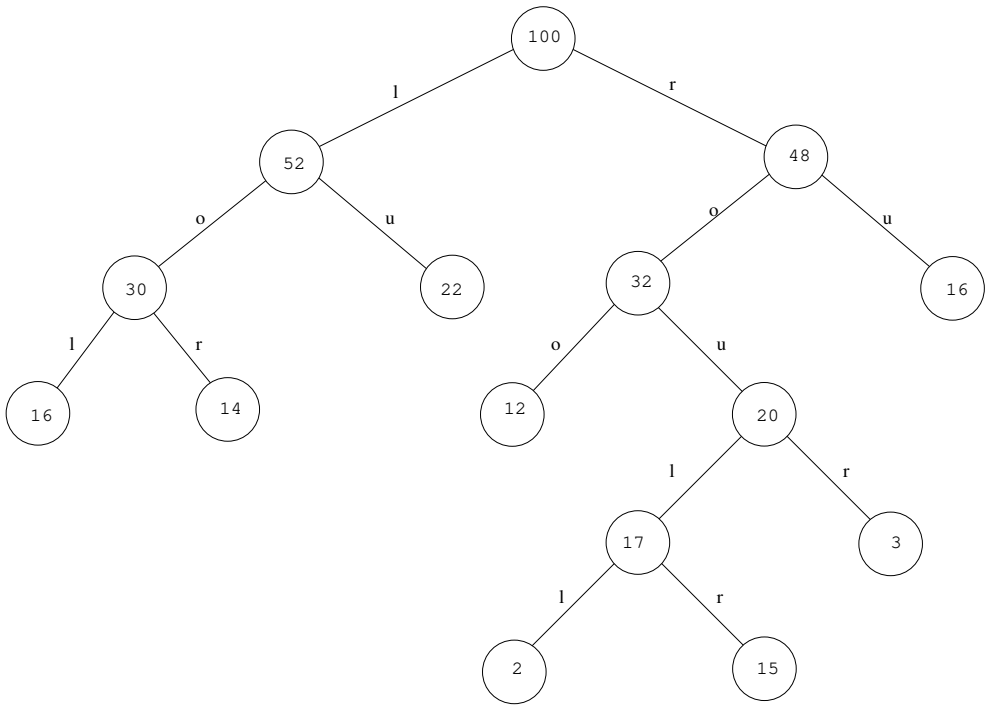


Abbildung 8.12: Schnittbaum beim Median-Cut-Algorithmus

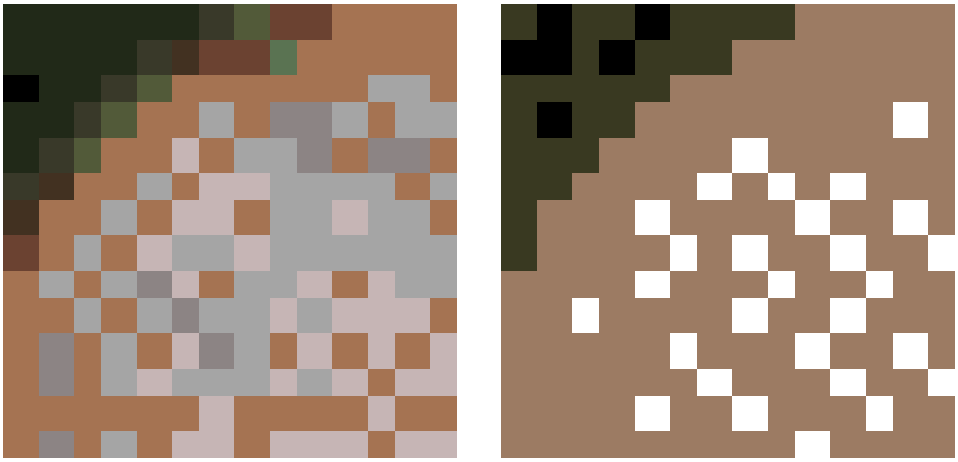
Abbildung 8.12 zeigt für das vorliegende Beispiel den Schnittbaum nach Anwendung des Median-Cut-Algorithmus. Die Knoten sind markiert mit der Anzahl der noch zu quantisierenden Pixel, an den Kanten ist vermerkt, ob es sich um einen Links/Rechts- oder um einen Oben/Unten-Schnitt handelt.

Floyd-Steinberg-Dithering (1975)
Der bei Verwendung einer Farbtabelle verursachte Fehler beim Quantisieren eines Pixels wird auf die Nachbarpixel verteilt (bevor diese quantisiert werden).

```
for (i = 0; i < M; i++)
for (j = 0; j < N; j++)
{
   x = f[i][j];   /* hole Original-Farbe */
   k = p(x);      /* finde naechsten Repraesentant */
   q[i][j] = k;   /* zeichne quantisiertes Pixel */
   e = d(x, k);   /* bestimme Fehlerabstand */
   f [i][j + 1]     = f[i][j + 1]     + e * 3.0/8.0;
   f [i + 1][j]     = f[i + 1][j]     + e * 3.0/8.0;
   f [i + 1][j + 1] = f[i + 1][j + 1] + e * 1.0/4.0;
}
```

24 Bit pro Pixel,
16 Mill. Farben

8 Bit pro Pixel,
256 Farben



4 Bit pro Pixel,
16 Farben

2 Bit pro Pixel,
4 Farben

Abbildung 8.13: Auswirkung des Median-Cut-Algorithmus

Abbildung 8.13 zeigt Original und quantisierte Versionen einer 14×14 Ausschnittvergrößerung, erstellt von einem 814×517 True-Color-Bild. Die Zahl der Farben bezieht sich jeweils auf das Gesamtbild. Verwendet wurde der Median-Cut-Algorithmus mit anschließendem Floyd-Steinberg-Dithering.

8.11 Alpha-Kanal

Neben dem Farbton, der Helligkeit und der Sättigung ist es häufig wünschenswert festlegen zu können, wie stark der Hintergund einer Grafik durch die einzelnen Pixel durchscheinen soll. Mit dem *Alpha-*

Kanal wird eingestellt, ob ein Pixel undurchsichtig (*opak*) ist oder nicht. Normalerweise werden für den Alpha-Kanal genausoviele Bits reserviert, wie für die Farbanteile. Bei 24 Bit-RGB-Farben ergeben sich also 32 Bit Gesamtplatzbedarf für ein Pixel. In Java entspricht dies genau einer Variable vom Typ `int`. Das entsprechende Farbmodell wird `RGBA`-Modell genannt.

Kapitel 9

Pixeldateiformate

Es gibt diverse Formate, in denen die generierten Grafiken abgespeichert werden können. Stellvertretend soll hier nur auf 2 Formate eingegangen werden; eines, bei dem die Pixel verlustfrei und unkomprimiert abgespeichert werden, und eines, bei dem durch die verwendete Kompression Verluste auftreten können.

9.1 PBM, PGM und PPM

Die *Portable Bitmap Utilities* (PBM, auch pbmplus oder netpbm) sind eine Sammlung freierhältlicher Programme, die von Jef Poskanzer gepflegt werden. Es handelt sich eigentlich um drei Programmpakete, die sich mit unterschiedlichen Bildarten und File-Formaten befassen:

- Die Portable Bitmap Utilities (PBM) manipulieren monochrome Bilder.
- Die Portable Greymap Utilities (PGM) manipulieren Grauwert-Bilder.
- Die Portable Pixmap Utilities (PPM) manipulieren Farbbilder.

Die Portable Anymap Utilities (PNM) arbeiten auf allen von den drei Programmpaketen erzeugten Bilddateien. Für PNM gibt es kein eigenes Dateiformat.

Die PBM-, PGM- und PPM-File-Formate sind so einfach wie möglich gehalten. Sie starten jeweils mit einem Header, dem die Bildinformation unmittelbar folgt. Der Header ist immer in ASCII geschrieben, wobei die einzelnen Einträge durch White Spaces getrennt werden. Die Bildinformation kann entweder im ASCII- oder Binärformat sein.

Es gibt jeweils zwei Header-Versionen, von denen eine für ASCII- und eine für binäre Bildinformationen benutzt wird.

PBM-Header

Ein PBM-Header besteht aus folgenden Einträgen, jeweils durch White Spaces getrennt:

Magic Value	`P1` für ASCII-, `P4` für binäre Bildinformation
Image Width	Breite des Bildes in Pixeln (ASCII-Dezimalwert)
Image Height	Höhe des Bildes in Pixeln (ASCII-Dezimalwert)

PGM-Header

Ein PGM-Header besteht aus folgenden Einträgen, jeweils durch White Spaces getrennt:

Magic Value	`P2` für ASCII-, `P5` für binäre Bildinformation
Image Width	Breite des Bildes in Pixeln (ASCII-Dezimalwert)
Image Height	Höhe des Bildes in Pixeln (ASCII-Dezimalwert)
Max Grey	Maximaler Grauwert (ASCII-Dezimalwert)

PPM-Header

Ein PPM-Header besteht aus folgenden Einträgen, jeweils durch White Spaces getrennt:

Magic Value	`P3` für ASCII-, `P6` für binäre Bildinformation
Image Width	Breite des Bildes in Pixeln (ASCII-Dezimalwert)
Image Height	Höhe des Bildes in Pixeln (ASCII-Dezimalwert)
Max Color	Maximaler Farbwert (ASCII-Dezimalwert)

Bildinformation in ASCII

Nach dem Header folgt die Beschreibung der Breite $\times$ Höhe vielen Pixel, beginnend in der linken oberen Bildecke zeilenweise von links nach rechts.

Bei PPM besteht jedes Pixel aus drei ASCII-Dezimalwerten zwischen 0 und dem angegebenen maximalen Farbwert, die die Rot-, Grün- und Blauwerte des Pixels darstellen.

Bei PBM und PGM gibt es nur einen ASCII-Dezimalwert pro Pixel. Bei PBM ist der maximale Wert implizit 1; die Werte müssen nicht durch White Spaces getrennt werden.

Beispiel:

Das folgende Beispiel stellt ein 4×4-Pixmap im ASCII-Format dar:

```
P3
# feep.ppm
4 4
15
 0  0  0    0  0  0    0  0  0   15  0 15
 0  0  0    0 15  7    0  0  0    0  0  0
 0  0  0    0  0  0    0 15  7    0  0  0
15  0 15    0  0  0    0  0  0    0  0  0
```

Wie zu sehen, können auch Kommentare in ein PNM-File eingefügt werden. Zeichen ab einem # bis zum Zeilenende werden ignoriert.

Binäre Bildinformation

Deutlich kleinere Dateien, die schneller gelesen und geschrieben werden können, werden mit Hilfe der binären Bilddarstellung erreicht.

Die Pixelwerte werden dabei wie folgt abgelegt:

Format	Magic Value	binäre Darstellung
PBM	`P4`	acht Pixel pro Byte
PGM	`P5`	ein Pixel pro Byte
PPM	`P6`	drei Bytes pro Pixel

Beim binären Format ist nach dem Header nur noch ein einziger White Space (typischerweise Newline) erlaubt. Innerhalb der binären Bilddaten sind keine White Spaces erlaubt. Die Bitorder innerhalb der Bytes ist "most significant bit first".

Beachtenswert ist, daß für das Binärformat nur maximale Werte bis 255 benutzt werden können.

Konvertierungsroutinen

Für die Umwandlung vom bzw. in das PNM-Format steht eine ganze Reihe von Routinen zur Verfügung, z.B.:

`anytopnm`, `asciitopgm`, `bmptoppm`, `giftopnm`, `pbmtopgm`, `pgmtopbm`, `pgmtoppm`, `ppmtopgm`, `pstopnm`, `rgb3toppm`, `tifftopnm`, `xbmtopbm`, `xwdtopnm`, `pbmtoascii`, `pbmtolps`, `pbmtopgm`, `pbmtoxbm`, `pgmtopbm`, `pgmtoppm`, `pnmtops`, `pnmtotiff`, `pnmtoxwd`, `ppmtobmp`, `ppmtogif`, `ppmtopgm`, `ppmtorgb3`.

9.2 Kompression nach JPEG

Die **J**oint **P**hotographic **E**xpert **G**roup bildete sich aus Mitgliedern der Standardisierungsgremien CCITT (Consultative Committee for International Telephone and Telegraph) und ISO (International Standardization Organization). JPEG wurde schließlich zum Namen für den Standard selbst.

Zunächst wird das RGB-Bild in den YUV-Raum transformiert. Da das Auge für Helligkeitssprünge sensitiver ist als für Farbdifferenzen, kann man nun die Y-Matrix in der vollen Auflösung belassen und in den U, V-Matrizen jeweils 4 Pixel mitteln (4:1:1 Subsampling).

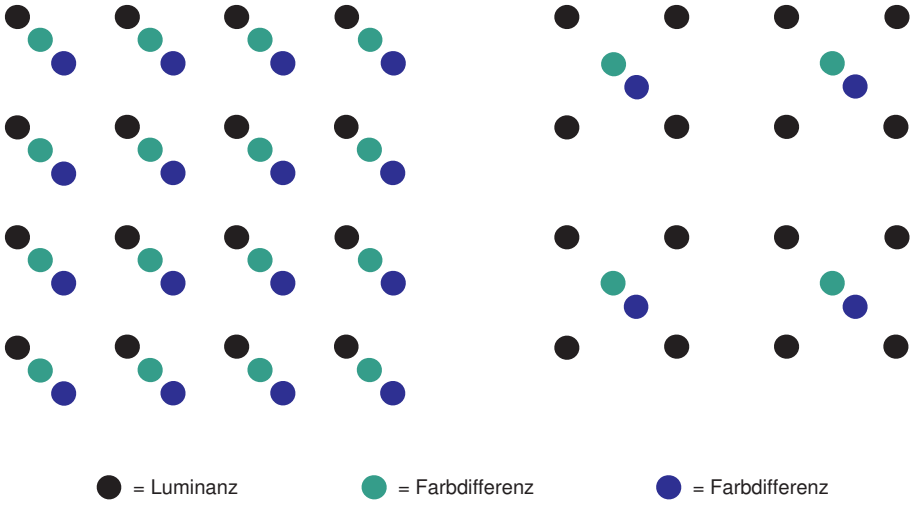


Abbildung 9.1: 4:1:1 Subsampling

Für je 4 Originalpixel mit insgesamt 12 Bytes werden nun $4+1+1=6$ Bytes benötigt (pro Bildpunkt also $6 \cdot 8/4 = 12$ Bit). Die Reduktion beträgt 50 %.

Beim JPEG-Verfahren werden nun die drei Matrizen in Blöcke mit 8×8 Abtastwerten aufgeteilt. Anschließend durchlaufen die Blöcke folgende Schritte:

1. Diskrete Cosinus Transformation
2. Rundung der Frequenzkoeffizienten
3. Lauflängenkodierung der quantisierten Werte
4. Huffmancodierung der Lauflängenbeschreibung.

Um aus dem komprimierten Bild das Original zu rekonstruieren, werden die Schritte in umgekehrter Reihenfolge und inverser Funktionalität durchlaufen.

Durch die Wahl der Rundungstabelle läßt sich der Tradeoff zwischen Qualität und Kompression beliebig steuern. Ein typisches Farbbild läßt sich ohne für das Auge sichtbare Artefakte auf 10 % seiner Originalgröße reduzieren. Eine Reduktion auf 5 % verursacht oft nur leichte, kaum wahrnehmbare Verzerrungen.

DCT (Diskrete Cosinus Transformation)
Die diskrete Cosinus-Transformation ist ein Spezialfall der Fouriertransformation. Es wird ausgenutzt, daß für "gerade" Funktionen (d.h. $f(-x) = f(x)$) der Sinus-Term mit seinem imaginären Anteil wegfällt. Ein zweidimensionaler Bildbereich läßt sich durch Spiegelung an der y-Achse künstlich gerade machen.

$$s[u,v] := \frac{1}{4} \cdot c_u \cdot c_v \cdot \sum_{x=0}^{7} \sum_{y=0}^{7} f[x,y] \cdot \cos \frac{(2x+1) \cdot u \cdot \pi}{16} \cdot \cos \frac{(2y+1) \cdot v \cdot \pi}{16}$$

$$c_u, c_v := \begin{cases} \frac{1}{\sqrt{2}} & \text{für } u,v = 0 \\ 1 & \text{sonst} \end{cases}$$

Hierdurch wird eine 8×8 Ortsmatrix in eine 8×8 Frequenzmatrix transformiert.

Mit Hilfe der inversen DCT lassen sich die Originalwerte rekonstruieren.

$$f[x,y] := \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} c_u \cdot c_v \cdot s[u,v] \cdot \cos \frac{(2x+1) \cdot u \cdot \pi}{16} \cdot \cos \frac{(2y+1) \cdot v \cdot \pi}{16}$$

Für die Bildverarbeitung wird der Pixelwertebereich 0..255 in das symmetrische Intervall $-128..127$ verschoben. Der Wertebereich von s liegt dann im Intervall $-1024.. + 1023$. Der errechnete Koeffizient s_{00} entspricht dem Anteil der Frequenz null in beiden Achsen und wird *DC*-Koeffizient (Gleichspannungsanteil) bezeichnet. Die übrigen Koeffizienten werden *AC*-Koeffizienten (Wechselspannungsanteil) genannt. Z.B. bezeichnet s_{77} die höchste, in beiden Richtungen auftretende Frequenz.

Die Eingangsmatrix M läßt sich auch durch $T \cdot M \cdot T'$ transformieren mit Hilfe der Matrix T:

```
0.353553   0.353553   0.353553   0.353553   0.353553   0.353553   0.353553   0.353553
0.490393   0.415735   0.277785   0.097545  -0.097545  -0.277785  -0.415735  -0.490393
0.461940   0.191342  -0.191342  -0.461940  -0.461940  -0.191342   0.191342   0.461940
0.415735  -0.097545  -0.490393  -0.277785   0.277785   0.490393   0.097545  -0.415735
0.353553  -0.353553  -0.353553   0.353553   0.353553  -0.353553  -0.353553   0.353553
0.277785  -0.490393   0.097545   0.415735  -0.415735  -0.097545   0.490393  -0.277785
0.191342  -0.461940   0.461940  -0.191342  -0.191342   0.461940  -0.461940   0.191342
0.097545  -0.277785   0.415735  -0.490393   0.490393  -0.415735   0.277785  -0.097545
```

Quantisierung
Die errechnete Matrix hat von links oben nach rechts unten Werte abnehmender Größe. Da die Werte rechts unten den hohen, eher unwichtigen Frequenzen entsprechen, werden alle Einträge mit Faktoren zunehmender Größe dividiert.

$$r[u,v] := \frac{s[u,v]}{q[u,v]}$$

95	88	87	95	88	95	95	95
143	144	151	151	153	170	183	181
153	151	162	166	162	151	126	117
143	144	133	130	143	153	159	175
123	112	116	130	143	147	162	189
133	151	162	166	170	188	166	128
160	168	166	159	135	101	93	98
154	155	153	144	126	106	118	133

Bildmatrix

↓

98	95	91	89	90	95	101	106
140	143	149	156	163	167	168	167
146	149	154	159	159	151	137	126
149	142	136	137	145	156	163	166
119	117	118	125	140	157	170	176
137	147	160	170	172	166	157	150
166	167	164	152	132	112	99	93
151	153	150	139	125	118	119	123

rekonstruierte Bildmatrix

↑

93	2	-8	-7	3	1	1	-2
-38	-58	11	17	-3	5	5	-3
-84	63	-1	-17	2	7	-4	-0
-51	-37	-10	13	-10	5	-1	-4
-85	-42	50	-8	18	-5	-1	1
-63	66	-13	-1	2	-6	-2	-2
-16	14	-37	18	-12	4	3	-3
-53	31	-7	-10	23	-0	2	2

DCT-Koeffizienten

→

31	0	-1	0	0	0	0	0
-7	-8	1	1	0	0	0	0
-12	7	0	-1	0	0	0	0
-5	-3	0	0	0	0	0	0
-7	-3	3	0	0	0	0	0
-4	4	0	0	0	0	0	0
-1	0	-1	0	0	0	0	0
-3	1	0	0	0	0	0	0

quantisierte DCT-Koeffizienten

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

Quantisierungsmatrix

Entropiekodierung
Die *DC*-Koeffizienten benachbarter Blöcke unterscheiden sich nur wenig und werden daher als Differenz zum Vorgängerblock übertragen.

Die *AC*-Koeffizienten werden zunächst in eine Zick-Zack-Sequenz umgeordnet:

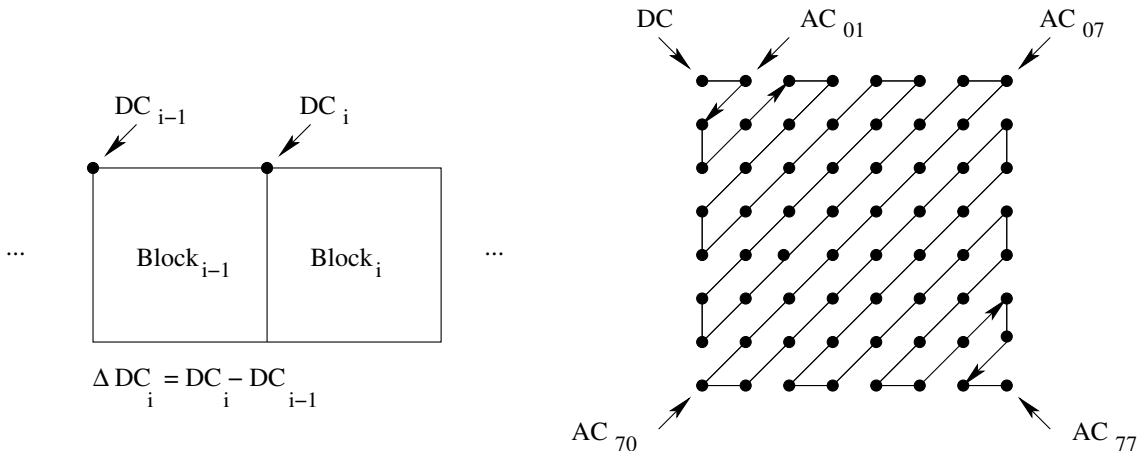


Abbildung 9.2: Durchlaufsequenz pro Macroblock

Die AC-Koeffizienten längs dieses Weges bis zum letzten Eintrag ungleich Null werden als Folge von Paaren beschrieben:

Symbol 1: Länge der ununterbrochenen Folge von Nullen vor diesem Wert (Runlength)
Anzahl der Bits, die zur Darstellung des Wertes erforderlich sind.
Symbol 2: der Wert selbst

Der quantisierte DCT-Koeffizient AC_{70} aus vorigem Beispiel wird beschrieben als Tupel $< (11,2), -3 >$, denn vor ihm in der Zickzacksequenz stehen 11 Nullen, sein Wert kann mit 2 Bit codiert werden, und sein Wert beträgt -3.

Für das Symbol 1 gibt es Häufigkeitsverteilungen, aus denen sich ein Huffman-Code konstruieren läßt. Für das Symbol 2 wählt man ein 2-er Komplement, welches berücksichtigt, daß bei angekündigten N Bits nur solche Zahlen kodiert werden müssen, für die $N-1$ Bits nicht ausreichen.

Länge/Bitzahl	Codierung
0/0 (EOB)	1010
0/1	00
0/2	01
0/3	100
0/4	1011
0/5	11010
0/6	1111000
0/7	11111000
0/8	1111110110
0/9	1111111110000010
0/10	1111111110000011
1/1	1100
1/2	11011
1/3	1111001
⋮	⋮
2/1	11100
2/2	11111001
2/3	1111110111
⋮	⋮
3/1	111010
3/2	111110111
3/3	111111110101
⋮	⋮
11/1	1111111001
11/2	1111111111010000
⋮	⋮
15/10	1111111111111110

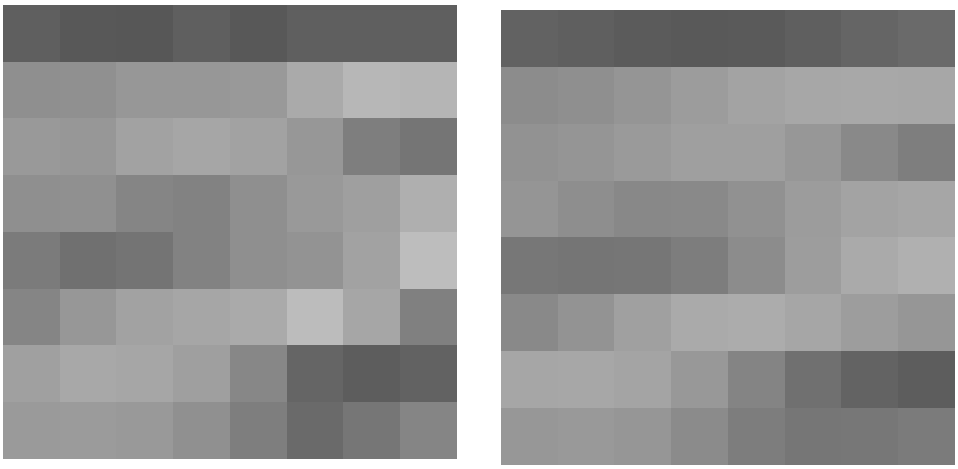
Huffman Codierung
für Symbol 1

Wert	Codierung
15	1111
14	1110
13	1101
12	1100
11	1011
10	1010
9	1001
8	1000
7	111
6	110
5	101
4	100
3	11
2	01
1	1
-1	0
-2	10
-3	00
-4	011
-5	010
-6	001
-7	000
-8	0111
-9	0110
-10	0101
-11	0100
-12	0011
-13	0010
-14	0001
-15	0000

Komplement-Codierung
für Symbol 2

Symbol 1	Symbol 2	Huffman für Symbol 1	2-er Komplement für Symbol 2
(1, 3)	-7	1111001	000
(0, 4)	-12	1011	0011
(0, 4)	-8	1011	0111
(0, 1)	-1	00	0
(1, 1)	1	1100	1
(0, 3)	7	100	111
(0, 3)	-5	100	010
(0, 3)	-7	100	000
(0, 2)	-3	01	00
(1, 1)	1	1100	1
(3, 1)	-1	111010	0
(1, 2)	-3	11011	00
(0, 3)	-4	100	011
(0, 1)	-1	00	0
(0, 3)	4	100	100
(0, 2)	3	01	11
(11, 2)	-3	1111111111010000	00
(0, 1)	1	00	1
(0, 1)	-1	00	0
EOB		1010	

Symbole und ihre Kodierung für die im Beispiel vorgestellten quantisierten *AC*-Koeffizienten



Ausgangsbildmatrix rekonstruierte Bildmatrix

Abbildung 9.3: 8 x 8 Matrix vor und nach der JPG-Kompression.

```
 95   88   87   95   88   95   95   95
143  144  151  151  153  170  183  181
153  151  162  166  162  151  126  117
143  144  133  130  143  153  159  175
123  112  116  130  143  147  162  189
133  151  162  166  170  188  166  128
160  168  166  159  135  101   93   98
154  155  153  144  126  106  118  133
```

Dezimaldarstellung der Bildbereich-Matrix

```
01011111  01011000  01010111  01011111  01011000  01011111  01011111  01011111
10001111  10010000  10010111  10010111  10011001  10101010  10110111  10110101
10011001  10010111  10100010  10100110  10100010  10010111  01111110  01110101
10001111  10010000  10000101  10000010  10001111  10011001  10011111  10101111
01111011  01110000  01110100  10000010  10001111  10010011  10100010  10111101
10000101  10010111  10100010  10100110  10101010  10111100  10100110  10000000
10100000  10101000  10100110  10011111  10000111  01100101  01011101  01100010
10011010  10011011  10011001  10010000  01111110  01101010  01110110  10000101
```

8-Bit-Codierung für Bildbereich-Matrix

```
00011111
```

quantisierter DC-Koeffizient

```
1111001000101100111011011100011001100111100010100000010011001
1110100110110010001100010010001111111111111010000000010001010
```

Huffman-Codierung für quantisierte AC-Koeffizienten

motel.tif
100%

motel80.jpg
9%

motel40.jpg
5%

motel20.jpg
4%

Abbildung 9.4: Kompression nach JPEG. Platzbedarf in Prozent bezogen auf tif-Datei

motel10.jpg
3%

motel05.jpg
2%

motel02.jpg
1.5%

motel01.jpg
1%

Abbildung 9.5: Kompression nach JPEG. Platzbedarf in Prozent bezogen auf tif-Datei

Kapitel 10

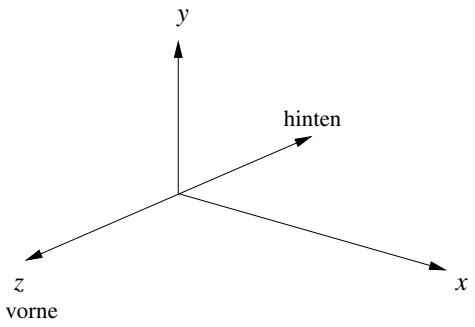
3D-Grundlagen

In diesem Kapitel werden die mathematischen Grundlagen dargelegt, die für die Darstellung von dreidimensionalen Objekten notwendig sind.

10.1 3D-Koordinatensystem

Weit verbreitet ist das kartesische Koordinaten-System, z.B. in der rechtshändigen Form.

Bei gespreizten Fingern der rechten Hand zeigt der Zeigefinger in x-Richtung, der Mittelfinger in y-Richtung, der Daumen in z-Richtung.



10.2 Länge und Kreuzprodukt

Gegeben sei ein 3D-Vektor

$$\vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

Seine **Länge** ist definiert als

$$|\vec{v}| := \sqrt{v_1^2 + v_2^2 + v_3^2}$$

Gegeben seien zwei 3D-Vektoren

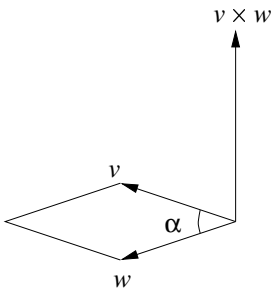
$$\vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \text{ und } \vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

Das **Kreuzprodukt** von $\vec{v}$ und $\vec{w}$ ist definiert als

$$\vec{v} \times \vec{w} = \begin{pmatrix} v_2 \cdot w_3 - v_3 \cdot w_2 \\ v_3 \cdot w_1 - v_1 \cdot w_3 \\ v_1 \cdot w_2 - v_2 \cdot w_1 \end{pmatrix}$$

Der Vektor $\vec{v} \times \vec{w}$ steht senkrecht auf $\vec{v}$ und steht senkrecht auf $\vec{w}$. Seine Länge entspricht der Fläche des durch $\vec{v}$ und $\vec{w}$ aufgespannten Parallelogramms, d.h.

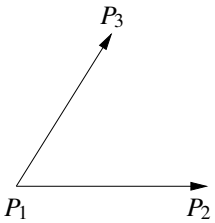
$$|\vec{v} \times \vec{w}| = |\vec{v}| \cdot |\vec{w}| \cdot \sin(\alpha)$$



In einem rechtshändigen Koordinatensystem entspricht bei gespreizten Fingern der Daumen $\vec{v}$, der Zeigefinger $\vec{w}$, der Mittelfinger $\vec{v} \times \vec{w}$.

Anwendung des Kreuzprodukts

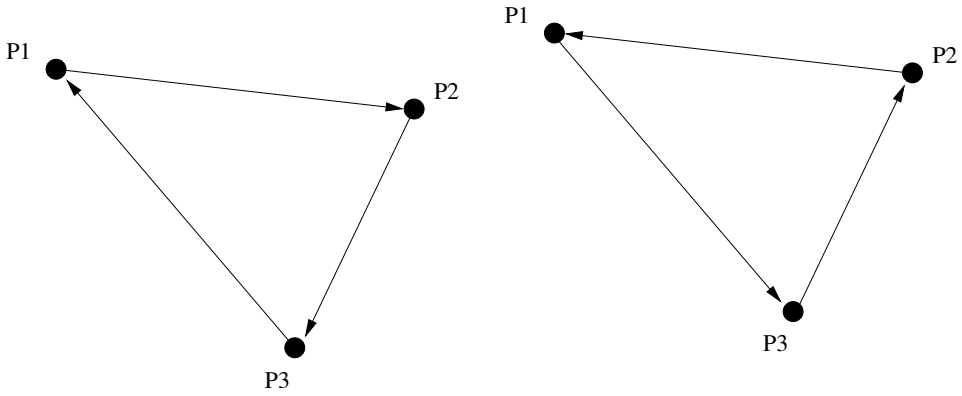
Gegeben sei eine Fläche durch 3 nicht kollineare Punkte P_1, P_2, P_3.



Der Vektor $P_2 \vec{-} P_1 \times P_3 \vec{-} P_1$ bildet den Normalenvektor zur Fläche.
Ist eine Ebene durch ihre Ebenengleichung $Ax + By + Cz + D = 0$ gegeben, so ergibt sich der Normalenvektor als $(A\ B\ C)^T$.
Ist ein Normalenvektor $(A\ B\ C)^T$ gegeben, so errechnet sich D durch Einsetzen eines beliebigen Punktes der Ebene.
Ein Punkt (x, y, z) liegt

oberhalb der Ebene,	falls	$Ax+By+Cz > -D$
unterhalb der Ebene,	falls	$Ax+By+Cz < -D$
in der Ebene,	falls	$Ax+By+Cz = -D$

Wir werden jede ebene Fläche als Ausschnitt einer Ebene ansehen und dem entsprechenden Polygon eine Flächennormale $\vec{n}$ zuweisen. Der Umlaufsinn des Polygons entscheidet, in welche Richtung die Flächennormale weist:



Dreieck mit Punkten im Uhrzeigersinn

Dreieck mit Punkten im Gegenuhrzeigersinn

Umlaufsinn bei Polygonen

ObdA werden der erste, der zweite und der letzte Punkt eines Polyogns zur Bestimmung der Normalen herangezogen. Im Fall eines im Uhrzeigersinn (*clockwise*) orientierten Polygons ergibt sich:

$$\vec{n_{cw}} = P_2 \vec{-} P_1 \times P_3 \vec{-} P_1 = \vec{v} \times \vec{w} = \begin{pmatrix} v_2 \cdot w_3 - v_3 \cdot w_2 \\ v_3 \cdot w_1 - v_1 \cdot w_3 \\ v_1 \cdot w_2 - v_2 \cdot w_1 \end{pmatrix}$$

Falls das Polygon gegen den Uhrzeigersinn (*counter clockwise*) orientiert ist, tauschen die Vektoren $\vec{v}$ und $\vec{w}$ die Plätze:

$$\vec{n_{ccw}} = P_3 \vec{-} P_1 \times P_2 \vec{-} P_1 = \vec{w} \times \vec{v} = \begin{pmatrix} w_2 \cdot v_3 - w_3 \cdot v_2 \\ w_3 \cdot v_1 - w_1 \cdot v_3 \\ w_1 \cdot v_2 - w_2 \cdot v_1 \end{pmatrix}$$

Es gilt also:

$$\vec{n_{cw}} = -\vec{n_{ccw}}$$

Die beiden Normalen weisen in entgegengesetzte Richtungen (Schiefsymmetrie des Kreuzproduktes).

Wir werden alle Polygone gegen den Uhrzeigersinn orientieren. Das hat den Vorteil, daß die Normale einer Fläche, die man so betrachtet, daß sie gegen den Uhrzeigersinn orientiert ist, in Richtung des Betrachters zeigt. Man "sieht" die Seite der Fläche, die die "Außenseite" sein soll.

Das Kreuzprodukt ist linear in jedem Argument. Es ist weder kommutativ (s.o.) noch assoziativ ($(\vec{a} \times \vec{b}) \times \vec{c} \neq \vec{a} \times (\vec{b} \times \vec{c})$).

10.3 Skalarprodukt

Gegeben seien zwei n-dimensionale Vektoren $\vec{v}, \vec{w}$.
Das **Skalarprodukt** lautet:

$$\vec{v} \cdot \vec{w} := \sum_{i=1}^{n} v_i \cdot w_i$$

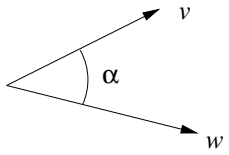
Für Vektoren $\vec{a}, \vec{b}, \vec{c}$ und $s \in \mathbb{R}$ gilt:

$$\begin{array}{rcll} \vec{a} \cdot \vec{b} & = & \vec{b} \cdot \vec{a} & \text{(Symmetrie)} \\ (\vec{a} + \vec{c}) \cdot \vec{b} & = & \vec{a} \cdot \vec{b} + \vec{c} \cdot \vec{b} & \text{(Linearität)} \\ (s\vec{a}) \cdot \vec{b} & = & s(\vec{a} \cdot \vec{b}) & \text{(Homogenität)} \\ |\vec{b}| & = & \sqrt{\vec{b} \cdot \vec{b}} & \text{(euklidische Norm)} \end{array}$$

Anwendungen des Skalarprodukts:

Gegeben zwei Vektoren $\vec{v}, \vec{w}$. Für den Winkel α zwischen $\vec{v}$ und $\vec{w}$ gilt:

$$\cos(\alpha) = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}| \cdot |\vec{w}|}.$$



Es gilt:
$\vec{v} \cdot \vec{w} < 0 \quad \Leftrightarrow \quad \vec{v}$ und $\vec{w}$ schließen einen Winkel von mehr als 90° ein
$\vec{v} \cdot \vec{w} = 0 \quad \Leftrightarrow \quad \vec{v}$ steht senkrecht auf $\vec{w}$
$\vec{v} \cdot \vec{w} > 0 \quad \Leftrightarrow \quad \vec{v}$ und $\vec{w}$ schließen einen Winkel von weniger als 90° ein

Es gilt: $\vec{n} \cdot \vec{r} = -D$ beschreibt eine Ebene für $D \in \mathbb{R}$ und Normalenvektor $\vec{n}$. Alle Lösungsvektoren $\vec{r}$ liegen (als Punkte aufgefaßt) auf der Ebene. Das Skalarprodukt aller Ebenenpunkte (als Vektoren geschrieben) mit dem Normalenvektor ist konstant.

10.4 Matrixinversion

Analog zum zweidimensionalen Fall werden die dreidimensionalen Transformationen durch Verknüpfung homogener Koordinaten mit 4×4-Transformationsmatrizen dargestellt.

Sei $A = (a_{ik})$, $1 \leq i,k \leq 4$, eine 4×4-Matrix:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

Ist $D = det\, A = |a_{ik}|$ die *Determinante* von A, so bezeichnet man als *Unterdeterminante* des Elementes a_{ik} diejenige 3-reihige Determinante, die aus D durch Streichen der i-ten Zeile und der k-ten Spalte hervorgeht. Unter der *Adjunkten* A_{ik} des Elementes a_{ik} versteht man die mit dem Faktor $(-1)^{i+k}$ versehene Unterdeterminante von a_{ik}.

Beispiel:

$$A_{23} = -\begin{vmatrix} a_{11} & a_{12} & a_{14} \\ a_{31} & a_{32} & a_{34} \\ a_{41} & a_{42} & a_{44} \end{vmatrix} = -[a_{11}(a_{32}a_{44} - a_{34}a_{42}) - a_{12}(a_{31}a_{44} - a_{34}a_{41}) + a_{14}(a_{31}a_{42} - a_{32}a_{41})]$$

Die Adjunkten sind nützlich zur Berechnung der Determinanten von A sowie der *inversen Matrix* A^{-1} (sofern diese existiert!):

$$\det A = \sum_{k=1}^{4} a_{1k}A_{1k} \qquad A^{-1} = \frac{1}{\det A}\begin{pmatrix} A_{11} & A_{21} & A_{31} & A_{41} \\ A_{12} & A_{22} & A_{32} & A_{42} \\ A_{13} & A_{23} & A_{33} & A_{43} \\ A_{14} & A_{24} & A_{34} & A_{44} \end{pmatrix}$$

Kapitel 11

3D-Transformationen

Wie im zweidimensionalen Fall, werden die Definitionspunkte der Objekte als Spaltenvektoren mit homogener Koordinate geschrieben. Die notwendigen Transformationen werden wieder durch Matrizen realisiert. Im dreidimensionalen Fall handelt es sich um 4×4-Matrizen.

11.1 Translation

Mit homogenen Koordinaten läßt sich der um den Translationsvektor $\vec{t} = (t_x\, t_y\, t_z)^T$ verschobene Punkt $P = (x, y, z)$

$$(x', y', z') := (x + t_x, y + t_y, z + t_z)$$

in der folgenden Form darstellen:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = T(t_x, t_y, t_z) \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

mit

$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

11.2 Skalierung

Gegeben: Drei Skalierungsfaktoren $s_x \neq 0$, $s_y \neq 0$ und $s_z \neq 0$.

Es liege der Fixpunkt im Ursprung:

$$(x', y', z') := (x \cdot s_x, y \cdot s_y, z \cdot s_z)$$

Die daraus resultierende Transformationsmatrix lautet:

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Es liege der Fixpunkt bei (Z_x, Z_y, Z_z):

1. Translation um $(-Z_x, -Z_y, -Z_z)$,
2. Skalierung um (s_x, s_y, s_z),
3. Translation um (Z_x, Z_y, Z_z).

Die Transformationsmatrix lautet:

$$T(Z_x, Z_y, Z_z) \cdot S(s_x, s_y, s_z) \cdot T(-Z_x, -Z_y, -Z_z)$$

11.3 Rotation

Rotation um die z-Achse

$$\begin{aligned} x' &:= x \cdot \cos(\delta) - y \cdot \sin(\delta) \\ y' &:= x \cdot \sin(\delta) + y \cdot \cos(\delta) \\ z' &:= z \end{aligned}$$

Die daraus resultierende Transformationsmatrix lautet:

$$R_z(\delta) = \begin{pmatrix} \cos(\delta) & -\sin(\delta) & 0 & 0 \\ \sin(\delta) & \cos(\delta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um die x-Achse

$$\begin{aligned} x' &:= x \\ y' &:= y \cdot \cos(\delta) - z \cdot \sin(\delta) \\ z' &:= y \cdot \sin(\delta) + z \cdot \cos(\delta) \end{aligned}$$

Die daraus resultierende Transformationsmatrix lautet:

$$R_x(\delta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\delta) & -\sin(\delta) & 0 \\ 0 & \sin(\delta) & \cos(\delta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um die y-Achse

$$\begin{aligned} x' &:= z \cdot \sin(\delta) + x \cdot \cos(\delta) \\ y' &:= y \\ z' &:= z \cdot \cos(\delta) - x \cdot \sin(\delta) \end{aligned}$$

Die daraus resultierende Transformationsmatrix lautet:

$$R_y(\delta) = \begin{pmatrix} \cos(\delta) & 0 & \sin(\delta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\delta) & 0 & \cos(\delta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation um eine beliebige Achse

Voraussetzung: Die Rotationsachse stimme nicht mit einer der Koordinatenachsen überein.

Idee: Transformiere Rotationsachse und Objekt so, daß die Rotationsachse mit der z-Achse übereinstimmt, rotiere um vorgegebenen Winkel δ, transformiere zurück.

1. Translation von Rotationsachse (und Objekt), so daß die Rotationsachse durch den Ursprung läuft.
2. Rotation der Rotationsachse um die x-Achse in die xz-Ebene.
3. Rotation der Rotationsachse um die y-Achse in die z-Achse.
4. Rotation des Objekts um die z-Achse mit Winkel δ.
5. Rücktransformation des gedrehten Objekts durch Anwendung der inversen Transformationen der Schritte (3), (2) und (1).

Ist die Rotationsachse durch die Punkte P_1, P_2 gegeben, so gilt

$$\vec{v} = P_2 - P_1 = \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \end{pmatrix}.$$

Die Länge dieses Vektors lautet

$$|\vec{v}| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}.$$

Die Komponenten des zugehörigen Einheitsvektors

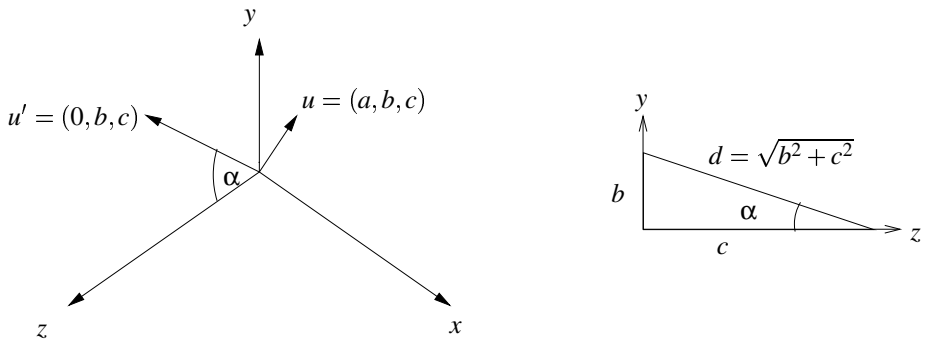
$$\vec{u} = \frac{\vec{v}}{|\vec{v}|} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}, |\vec{u}| = 1$$

lauten daher

$$a = \frac{x_2 - x_1}{|\vec{v}|},\ b = \frac{y_2 - y_1}{|\vec{v}|},\ c = \frac{z_2 - z_1}{|\vec{v}|}.$$

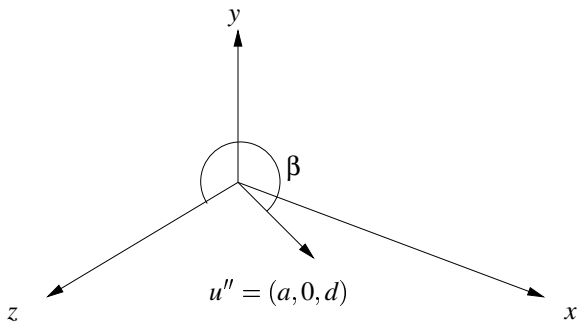
Schritt 1 läßt sich durch die Translation $T(-x_1, -y_1, -z_1)$ durchführen. Dadurch wird P_1, Ausgangspunkt des Einheitsvektors $\vec{u}$, in den Ursprung verschoben.

Für Schritt 2 sind Sinus und Cosinus des Rotationswinkels α erforderlich, der zwischen der Projektion $\vec{u}'$ von $\vec{u}$ auf die yz-Fläche und der z-Achse, repräsentiert durch den Vektor $\vec{u}_z = (0\ 0\ 1)^T$, liegt.

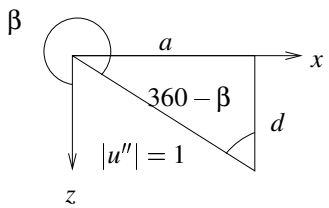


$\Rightarrow \cos(\alpha) = \frac{c}{d}$
$\Rightarrow \sin(\alpha) = \frac{b}{d}$

Nach Schritt 2 befindet sich der ursprüngliche Vektor $\vec{u}$ als $\vec{u}''$ in der xz-Ebene:



Für Schritt 3 (Rotation um y-Achse) benötigt man Sinus und Cosinus des Rotationswinkels β. Positive Winkel ergeben eine Rotation gegen den Uhrzeigersinn, wenn man aus Richtung der Positiven y-Achse auf die xz-Ebene schaut:



$\Rightarrow \cos(\beta) = \cos(360° - \beta) = d$
$\Rightarrow \sin(\beta) = -\sin(360° - \beta) = -a$

Nach den ersten drei Schritten ist die Drehachse mit der z-Achse identisch, so daß Schritt (4) mit der Rotationsmatrix $R_z(\delta)$ durchgeführt werden kann. Schritt (5) beinhaltet die Anwendung der inversen Transformationen.

Die Rotation um die Achse $\vec{v} = \overline{P_1P_2}$ um den Winkel δ läßt sich daher wie folgt darstellen:

$$R(\vec{v}, \delta) = T(P_1)R_x^{-1}(\alpha) \cdot R_y^{-1}(\beta) \cdot R_z(\delta) \cdot R_y(\beta) \cdot R_x(\alpha) \cdot T(-P_1) \cdot$$

11.4 Transformation von Koordinatensystemen

Wenn wir ein Koordinatensystem wählen, um die Punkte des $\mathbb{R}^3$ zu beschreiben, dann wählen wir damit auch eine Basis, die den Vektorraum aufspannt. Die Basisvektoren sind die Einheitsvektoren in Richtung der Koordinatenachsen. Zwei verschiedene Koordinatensysteme haben zwei verschiedene Basen.

Die Transformation von einem Koordinatensystem in ein anderes bedeutet also einen Basiswechsel. Gegeben sei ein Koordinatensystem A (z.B. das Kartesische Koordinatensystem) mit zugehöriger Basis $\mathcal{A}$. Z.B. die kanonische Basis (in Matrixschreibweise mit Spaltenvektoren):

$$\mathcal{A}_{kart} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

In homogenen Koordinaten werden als vierter Spaltenvektor die Koordinaten des Ursprungs von Koordinatensystem A, beschrieben bzgl. Basis $\dashv$ (hier also: $(0\,0\,0\,1)^T$), hinzugenommen:

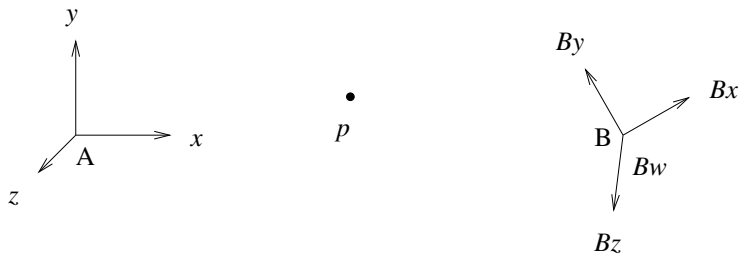
$$\mathcal{A}_{kart} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Weiterhin sei ein zweites - von A verschiedenes - Koordinatensystem B gegeben. Auch dieses Koordinatensystem spezifiziert 4 ausgezeichnete Elemente: Seinen Ursprungspunkt B_w und die drei Einheitsvektoren $\vec{B}_x$, $\vec{B}_y$, $\vec{B}_z$.

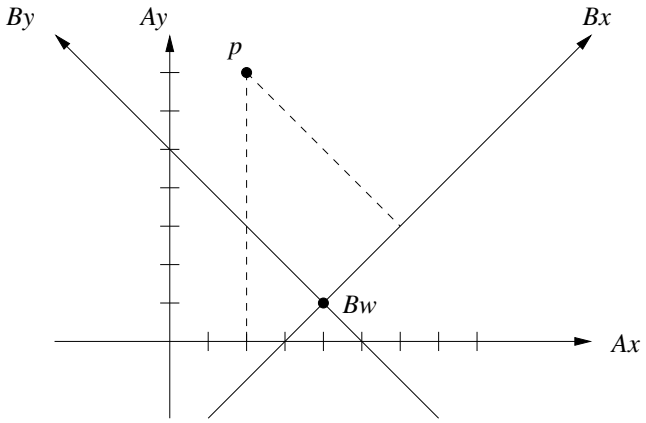
Diese Elemente lassen sich sowohl bzgl. der Basis $\mathcal{A}$ als auch bzgl. der Basis $\mathcal{B}$ darstellen. Um von einer Darstellung in die andere zu kommen, muß die gegebene Darstellung nur mit der im Folgenden beschriebenen Matrix mutlipliziert werden.

Wenn man die homogenen Koordinatenvektoren in Spaltenschreibweise nebeneinander anordnet ergeben sie die Matrix $M_{\mathcal{B}\to\mathcal{A}}$, die den Übergang von Basis $\mathcal{B}$ zur Basis $\mathcal{A}$ beschreibt:

$$M_{\mathcal{B}\to\mathcal{A}} = \left(\vec{B}_x\ \vec{B}_y\ \vec{B}_z\ B_w\right)$$



Beispiel (für den 2-dimensionalen Fall):



x-Achse:	$\vec{B}_x$	lautet	$(\sqrt{\frac{1}{2}}\ \sqrt{\frac{1}{2}}\ 0)^T$
y-Achse:	$\vec{B}_y$	lautet	$(-\sqrt{\frac{1}{2}}\ \sqrt{\frac{1}{2}}\ 0)^T$
Ursprung:	B_w	lautet	$(4,1,1)$
Punkt:	$P_\mathcal{B}$	lautet	$(2\cdot\sqrt{2}, 4\cdot\sqrt{2}, 1)$

Der Aufbau der Matrix $M_{\mathcal{B}\to\mathcal{A}}$ repräsentiert die erforderliche Drehung und Verschiebung, um einen aus der Sicht von Koordinatensystem B (also bzgl. Basis $\mathcal{B}$ beschriebenen Punkt P aus der Sicht von Koordinatensystem A (also bzgl. $\mathcal{A}$) zu beschreiben. Im zwei-dimensionalen Fall wird zunächst der Punkt P um den Winkel α gedreht, der sich zwischen den Achsen der Koordinatensysteme A und B befindet. Dann wird eine Translation durchgeführt mit dem Wert der Ursprungsposition von B. Cosinus und Sinus des Drehwinkels α ergeben sich gerade aus den Werten a bzw. b, wobei die Einheitsvektoren $(a\ b)^T$ und $(-b\ a)^T$ die Basis für Koordinatensystem B darstellen.

Also läßt sich in obigem Beispiel der Punkt p wie folgt transformieren:

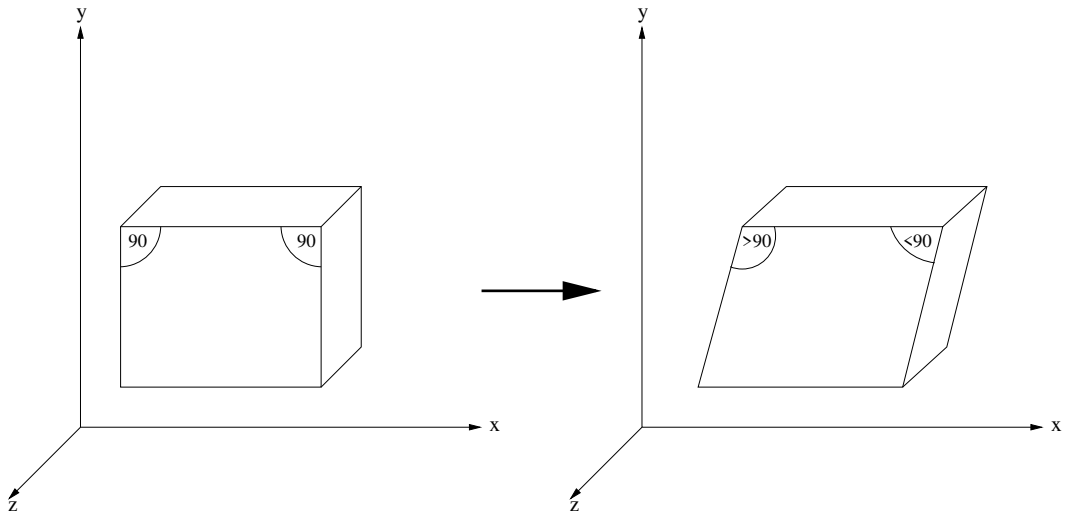
$$\begin{pmatrix} \sqrt{\frac{1}{2}} & -\sqrt{\frac{1}{2}} & 4 \\ \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & 1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2\cdot\sqrt{2} \\ 4\cdot\sqrt{2} \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 7 \\ 1 \end{pmatrix}$$

Um einen Punkt $P_{\mathcal{A}}$ im Koordinatensystem A bzgl. des Koordinatensystems B zu spezifizieren, verwendet man die inverse Matrix zu $M_{\mathcal{B}\to\mathcal{A}}$:

$$M_{\mathcal{B}\to\mathcal{A}} \cdot \vec{p}_{\mathcal{B}} = \vec{p}_{\mathcal{A}}$$
$$\Leftrightarrow \quad \vec{p}_{\mathcal{B}} = M^{-1}_{\mathcal{B}\to\mathcal{A}} \cdot \vec{p}_{\mathcal{A}}$$

11.5 Transformation der Normalenvektoren

Die Normalenvektoren müssen bei der Transformation von Objektpunkten ebenfalls abgebildet werden. Wenn diese Transformation z.B. eine nicht-uniforme Skalierung ist, dann bleiben die Winkel zwischen einzelnen Flächen nicht erhalten.



Winkeluntreue unter nicht uniformer Skalierung

Wenn die Normale $\vec{n}$ mit derselben Matrix M transformiert wird, wie die Objektpunkte einer Fläche F, ist $\vec{n}$ anschließend evtl. nicht mehr senkrecht zu F.

Wie muß $\vec{n}$ transformiert werden?

Gegeben seien zwei Punkte $P_1 = (p_{1x}, p_{1y}, p_{1z}, 1)$ und $P_2 = (p_{2x}, p_{2y}, p_{2z}, 1)$, die in der Fläche F liegen und die Normale $\vec{n} = (n_x\ n_y\ n_z\ 0)^T$. Weiterhin sei die Transformationsmatrix M gegeben. Es gilt:

$$P_1' = MP_1 \text{ und } P_2' = MP_2$$

wobei die Punktkoordinaten als Spaltenvektoren aufzufassen sind.

Der Vektor

$$\vec{r} = P_2 - P_1$$

liegt in der Ebene von F und ist deshalb senkrecht zu $\vec{n}$:

$$\vec{r} \cdot \vec{n} = 0$$

Diese Beziehung soll auch für die projizierten Vektoren $\vec{n}'$ und

$$\vec{r}' = P_2' - P_1' = MP_2 - MP_1 = M(P_2 - P_1) = M\vec{r}$$

gelten:

$$\vec{n}' \cdot \vec{r}' = 0$$

Vorletzte Gleichung ausnutzen ergibt

$$\vec{n}' \cdot M\vec{r} = 0$$

Matrizenmultiplikation ausführen und Skalarprodukt ausführlich schreiben (dabei $n'_w = 0$ ausnutzen):

$$\begin{aligned} n'_x(m_{11}r_x + m_{12}r_y + m_{13}r_z + m_{14}) &+ \\ n'_y(m_{21}r_x + m_{22}r_y + m_{23}r_z + m_{24}) &+ \\ n'_z(m_{31}r_x + m_{32}r_y + m_{33}r_z + m_{34}) &= 0 \end{aligned}$$

Nach den Komponenten von $\vec{r}$ sortieren:

$$\begin{aligned} r_x(m_{11}n'_x + m_{21}n'_y + m_{31}n'_z) &+ \\ r_y(m_{12}n'_x + m_{22}n'_y + m_{32}n'_z) &+ \\ r_z(m_{13}n'_x + m_{23}n'_y + m_{33}n'_z) &+ \\ 1(m_{14}n'_x + m_{24}n'_y + m_{34}n'_z) &= 0 \end{aligned}$$

Matrixmultiplikation und Skalarprodukt erkennen und kompakt schreiben:

$$\vec{r} \cdot \begin{pmatrix} m_{11}m_{21}m_{31}m_{41} \\ m_{12}m_{22}m_{32}m_{42} \\ m_{13}m_{23}m_{33}m_{43} \\ m_{14}m_{24}m_{34}m_{44} \end{pmatrix} \vec{n}' = \vec{r} \cdot \vec{n} = 0$$

Die Matrix ist die Transponierte M^T zu M.

Es gilt also

$$\vec{n} = M^T\vec{n}'$$

Nach $\vec{n}'$ auflösen, indem zunächst alles transponiert wird

$$\vec{n}^T = \vec{n}'^T M$$

Dann mit M^{-1} von rechts multiplizieren

$$\vec{n}^T M^{-1} = \vec{n}'^T$$

Alles nochmal transponieren und die Seiten der Gleichung tauschen

$$\vec{n}' = (M^{-1})^T\vec{n}$$

Es gilt also, daß bei der Transformation eines Objekts mit der Matrix M ein Normalenvektor $\vec{n}$ dieses Objekts mit der transponierten Inversen von M abgebildet werden muß.

Kapitel 12

Projektion

12.1 Bildebene

Für die Anzeige am zweidimensionalen Ausgabegerät muß eine Abbildung (*Projektion*) der räumlichen, dreidimensionalen Szene auf eine zweidimensionale *Projektionsebene* erfolgen.

Gegeben sind

- das zu projizierende Objekt,
- die Bildebene,
- das Projektionszentrum.

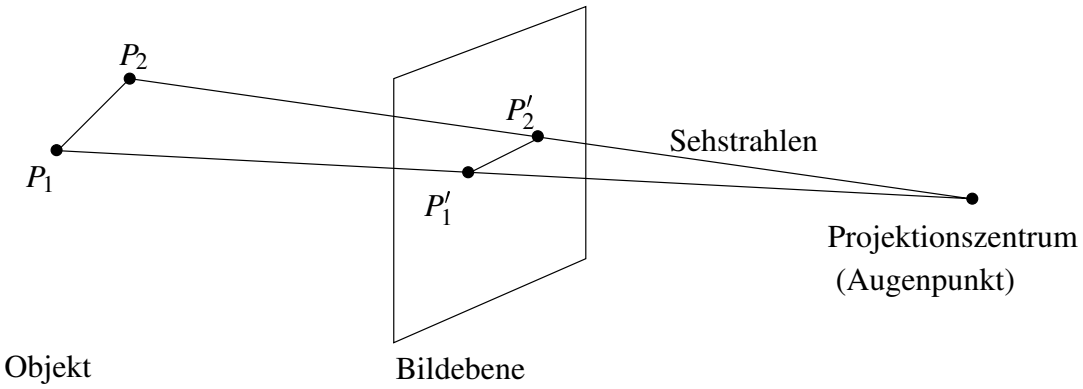
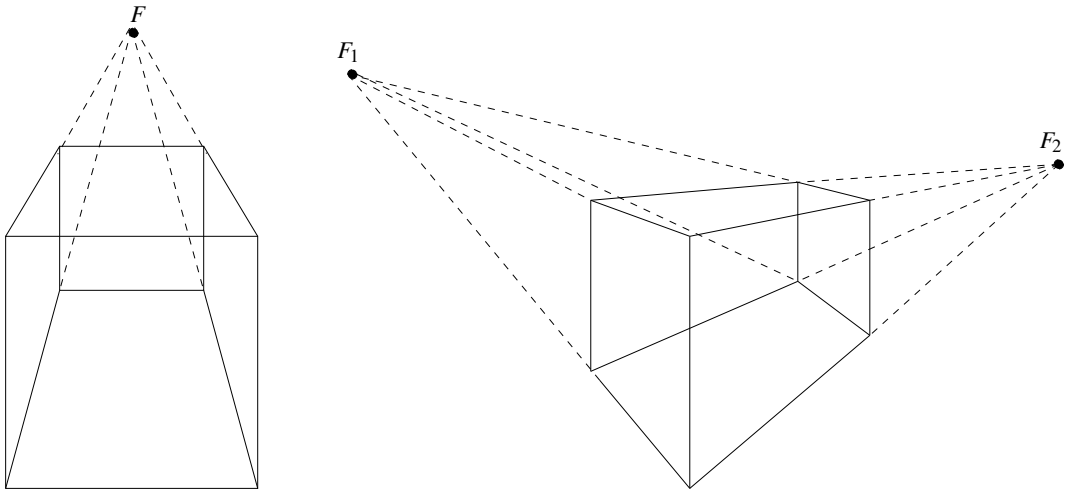


Abbildung 12.1: Zentralprojektion

Ist der Abstand des Projektionszentrums von der Bildebene endlich, so handelt es sich um eine Zentralprojektion (perspektivische Projektion), andernfalls handelt es sich um eine Parallelprojektion (die Projektionsstrahlen sind zueinander parallel).

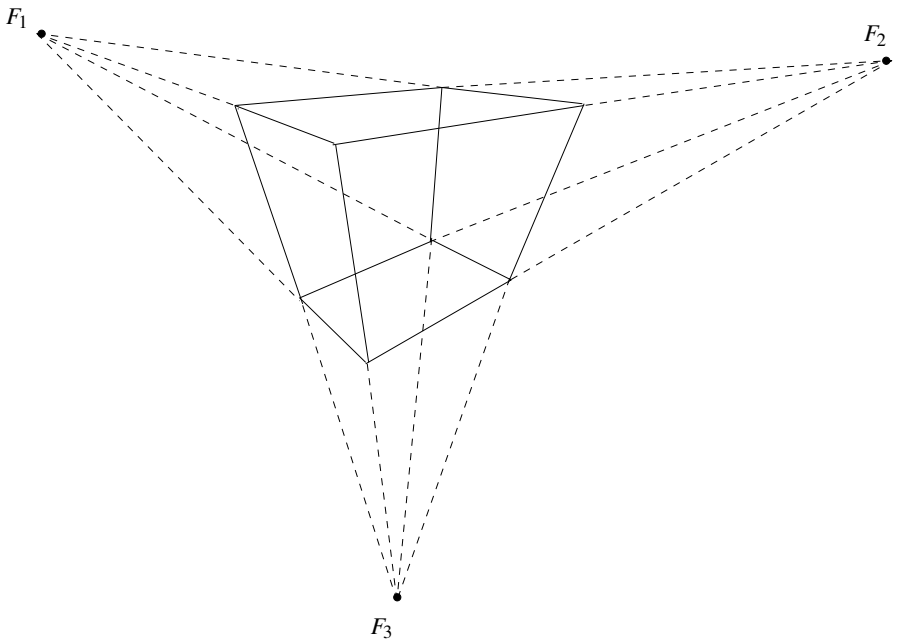
12.2 Perspektivische Projektion

Die abgebildeten Objekte werden proportional zu ihrem Abstand zur Bildebene verkleinert. Je nachdem, ob die Bildebene eine, zwei oder drei der Koordinatenachsen schneidet, entstehen ein, zwei oder drei Fluchtpunkte.



1 Fluchtpunkt F

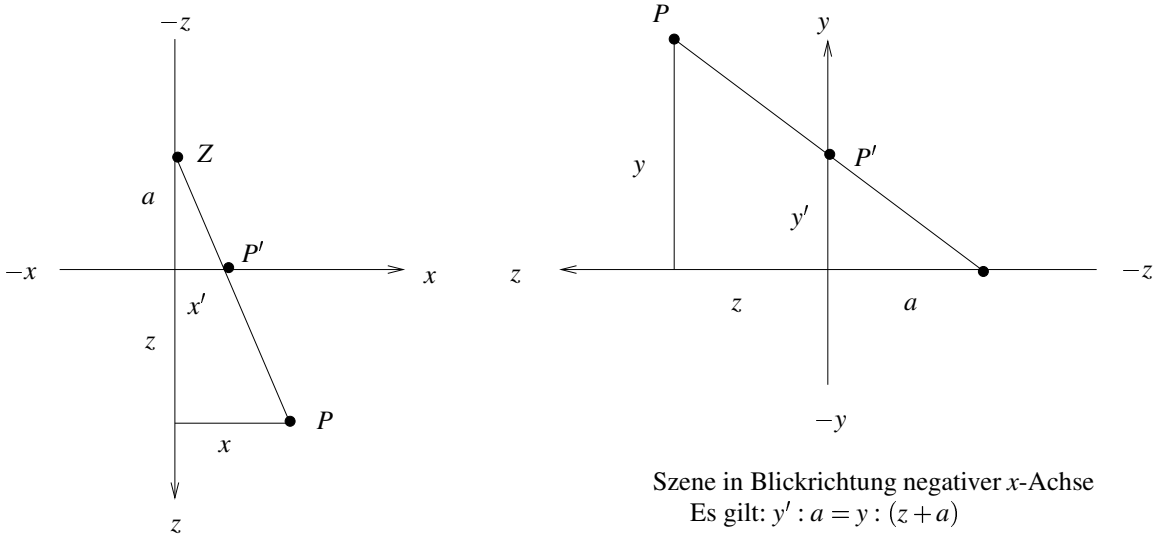
2 Fluchtpunkte F_1, F_2



3 Fluchtpunkte F_1, F_2, F_3

Abbildung 12.2: Zentralprojektionen mit unterschiedlich vielen Fluchtpunkten

OBdA sei die Bildebene gleich der xy-Ebene, und das Projektionszentrum liege auf der negativen z-Achse im Punkt $Z = (0,0,-a)$. Gegeben Punkt $P = (x,y,z)$. Gesucht sind auf der Bildebene die Koordinaten des projizierten Bildpunktes $P' = (x',y',0)$.



Szene in Blickrichtung negativer y-Achse
Es gilt: $x' : a = x : (a+z)$

Szene in Blickrichtung negativer x-Achse
Es gilt: $y' : a = y : (z+a)$

Abbildung 12.3: Anwendung der Strahlensätze

Betrachtet man die Szene “von oben” und “von der Seite”, so erhält man aufgrund der Strahlensätze die Beziehung

$$x' = \frac{x}{1+z/a}\,, \quad y' = \frac{y}{1+z/a}\,, \quad z' = 0.$$

Die homogenen Koordinaten des projizierten Punktes lauten

$$P' = (x/w, y/w, 0, 1) = (x, y, 0, w) \text{ mit } w = 1 + z/a.$$

Für die Transformationsmatrix der perspektivischen Projektion ergibt sich also:

$$P_{persp_{xy}}(-a) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/a & 1 \end{pmatrix}.$$

12.3 Parallelprojektion

12.3.1 Normalprojektionen

Stehen die Sehstrahlen normal zur Bildebene, liegt eine *orthogonale* Projektion vor.

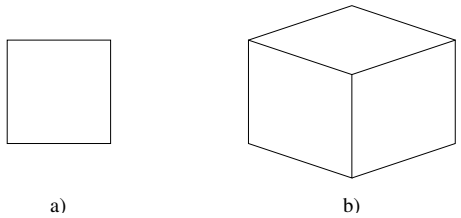


Abbildung 12.4: Grund-, Seiten-, Aufriss (a) bzw. axononometrische Projektion (b)

Die häufigste Anwendung orthogonaler Projektionen liegt in der Darstellung eines Objekts durch *Grund-*, *Auf-* und *Seitenriß* (Abbildung 12.4 a).

Eine weitere Form der Normalprojektionen sind die *axonometrischen Projektionen*, bei denen die Bildebene auf keiner der Körper-Hauptachsen senkrecht steht. Dadurch sind in der Abbildung mehrere zueinander normal stehende Flächen gleichzeitig sichtbar. Die resultierenden Darstellungen sind der perspektivischen Projektion ähnlich. Anstelle der zur Entfernung vom Auge proportionalen Verkürzung erfolgt aber eine gleichmäßige Verkürzung aller Kanten (Abbildung 12.4 b).

Die Transformationsmatrix für die orthogonale Parallelprojektion auf die xy-Ebene lautet

$$P_{ortho_{xy}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

da für jeden Punkt $P = (x, y, z, 1)$ die Koordinaten des projizierten Punktes gleich $(x, y, 0, 1)$ sind.

12.3.2 Schiefe Projektionen

Bei den *schiefen* Parallelprojektionen stehen die Sehstrahlen nicht normal auf der Bildebene, sondern schneiden sie unter dem Winkel β (Abbildung 12.5). Die schiefe Projektion auf die xy-Ebene entspricht einer Scherung der x- und y-Koordinaten proportional zu z.

Seien $P_0 = (x, y, 0)$ die orthogonale und $P' = (x', y', 0)$ die schiefe Projektion von Punkt $P = (x, y, z)$. Sei L die Entfernung von P_0 nach P'. Dann gilt

$$\begin{aligned} x' &= x - L \cdot \cos(\alpha) \\ y' &= y + L \cdot \sin(\alpha) \\ z' &= 0 \end{aligned}$$

Wegen $\tan(\beta) = \frac{z}{L}$ folgt

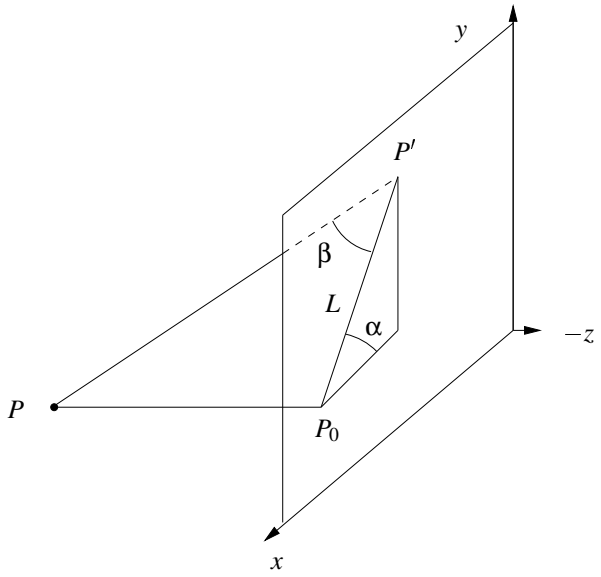


Abbildung 12.5: Bildebene bei der schiefen Projektion

$$x' = x - z \cdot \frac{\cos(\alpha)}{\tan(\beta)}$$
$$y' = y + z \cdot \frac{\sin(\alpha)}{\tan(\beta)}$$

Der Winkel β regelt im projizierten Bild das Verhältnis von x-Ausdehnung zu z-Ausdehnung.

Die Koordinaten zweier projizierter Punkte P_1', P_2' lauten:

$$x_1' = x_1 - z_1 \cdot \cos(\alpha)/\tan(\beta)$$
$$y_1' = y_1 + z_1 \cdot \sin(\alpha)/\tan(\beta)$$
$$x_2' = x_2 - z_2 \cdot \cos(\alpha)/\tan(\beta)$$
$$y_2' = y_2 + z_2 \cdot \sin(\alpha)/\tan(\beta)$$

Für zwei Punkte, die sich nur bzgl. ihrer z-Koordinaten unterscheiden, betragen die Abstände ihrer Bilder in x- bzw. y-Richtung

$$|x_1' - x_2'| = |(z_1 - z_2) \cdot \cos(\alpha)/\tan(\beta)|$$
$$|y_1' - y_2'| = |(z_1 - z_2) \cdot \sin(\alpha)/\tan(\beta)|$$

für den Abstand

$$|P_1' - P_2'| = \sqrt{|x_1' - x_2'|^2 + |y_1' - y_2'|^2}$$

ergibt sich

$$|P_1' - P_2'| = \sqrt{\frac{(z_1 - z_2)^2}{\tan^2(\beta)} \cdot (\cos^2(\alpha) + \sin^2(\alpha))} = \frac{z_1 - z_2}{\tan(\beta)}$$

wegen

$$\cos^2(\alpha) + \sin^2(\alpha) = 1.$$

Für die Berechnung der Transformationsmatrix benötigt der Algorithmus den Verkürzungsfaktor d und den Scherwinkel α. d gibt an, um welchen Faktor zur Bildebene normal stehende Strecken verkürzt werden. Es gilt

$$d = \frac{1}{\tan(\beta)}.$$

α definiert den Winkel zur Horizontalen, unter dem diese Kanten aufgetragen werden. Für die Koordinaten des so projizierten Punktes $P = (x, y, z, 1)$ gilt

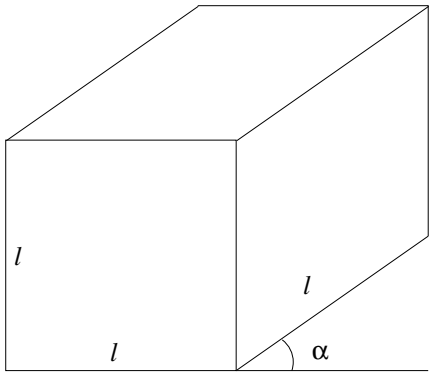
$$\begin{aligned} x' &= x - z \cdot (d \cdot \cos(\alpha)), \\ y' &= y + z \cdot (d \cdot \sin(\alpha)), \\ z' &= 0, \\ w' &= 1 \end{aligned}$$

Die entsprechende Transformationsmatrix lautet

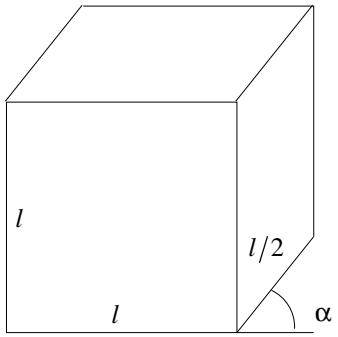
$$P_{schief_{xy}}(\alpha, d) = \begin{bmatrix} 1 & 0 & -d \cdot \cos(\alpha) & 0 \\ 0 & 1 & d \cdot \sin(\alpha) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Für $d = 0$ ($\beta = 90°$) ergibt sich daraus die orthogonale Projektion. Bei schiefen Projektionen ist der Wert für d immer ungleich Null.

Zwei häufig als Ersatz für Perspektive verwendete Projektionen haben die Werte $d = 1$ ($\beta = 45°$), $\alpha = 35°$ (*Kavalierprojektion*) und $d = 0.5$ ($\beta = 63.43°$), $\alpha = 50°$ (*Kabinettprojektion*). Bei der Kavalierprojektion werden alle auf der Bildebene normal stehenden Strecken unverkürzt abgebildet. Bei der Kabinettprojektion ergibt sich eine Verkürzung auf die Hälfte ihrer ursprünglichen Länge.



Kavalierprojektion: $\beta = 45°$
kombiniert mit $\alpha = 35°$



Kabinettprojektion: $\beta = 63.43°$
kombiniert mit $\alpha = 50°$

Abbildung 12.6: Zwei schiefe Projektionen

Kapitel 13

Betrachtungstransformationen

Die Abbildung dreidimensionaler Objekte auf den Bildschirm wird in eine Reihe von Elementartransformationen zerlegt:

- Konstruktion von komplexen Szenen aus elementaren Objekten (*Modeling*),
- Festlegen der Bildebene (*View Orientation*),
- Projektion auf ein normiertes Gerät (*View Mapping*),
- Abbildung auf ein Ausgabegerät (*Device Mapping*).

13.1 Die synthetische Kamera

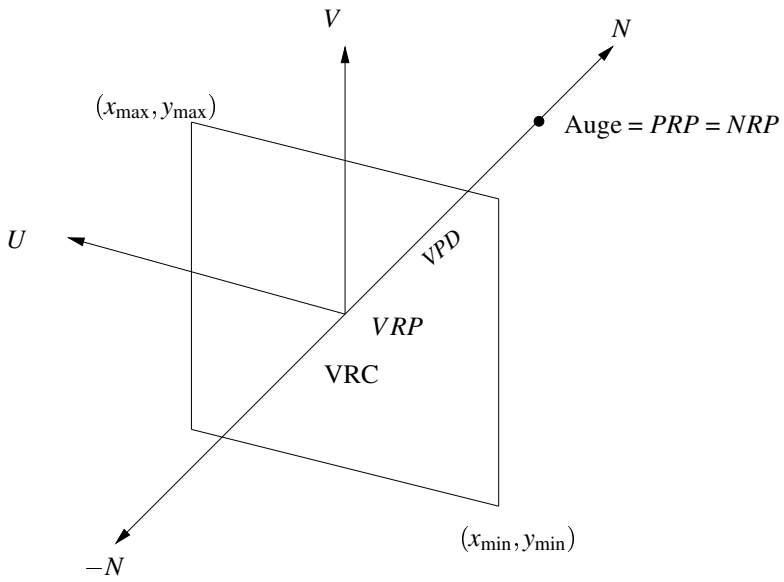


Abbildung 13.1: Parameter zur Beschreibung der syntetischen Kamera

Die Parameter zur Durchführung der 3D-auf-2D-Abbildung (*viewing transformation*) sind im wesentlichen die Position und Orientierung einer *synthetischen Kamera* und ein Punkt in der Szene, auf den die Kamera fokussiert ist: Der View Reference Point (VRP).

Die View Plane ist die Bildebene, auf die die Szene projiziert wird. Diese ist definiert durch die View Plane Normal N, die Richtung, aus der die Kamera die Szene aufnimmt, und den VRP. Die Kamera befindet sich im PRP (Projection Reference Point oder Normal Reference Point (NRP)), der dem Augenpunkt eines natürlichen Betrachters entspricht.

Die Orientierung der Kamera wird durch den View Up Vector (VUV oder VUP) festgelegt. Durch Projektion des VUV in die View Plane erhält man den Vektor V. Der Vektor U in der View Plane wird so gewählt, daß U, V, N ein rechtshändiges kartesisches Koordinatensystem bilden, das sogenannte View Reference Coordinate System VRC mit dem VRP als Ursprung und dem Augenpunkt PRP auf der positiven N-Achse bei $N = VPD$ (View Plane Distance).

Die beiden Punkte in den UV-Koordinaten der Viewplane $(x_{\min}, y_{\min})$, $(x_{\max}, y_{\max})$ legen das sogenannte View Window fest, d.h., was von der Szene zu sehen ist (entspricht der Brennweite eines Kameraobjektivs). Der sichtbare Teil kann noch weiter eingeschränkt werden durch Angabe einer *front plane* und einer *back plane*.

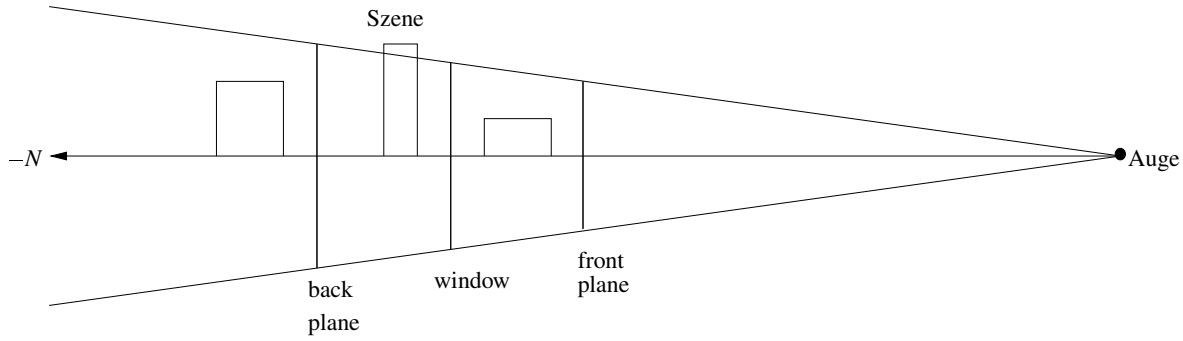


Abbildung 13.2: Einschränkung des sichtbaren Teils

Bemerkung: Zur Berechnung der Projektion auf die Bildebene ist es unerheblich, wo die Bildebene in der Szene arrangiert ist.

13.2 Viewing Pipeline

Folgende Informationen sind erforderlich:

- Jedes Objekt wird beschrieben durch Modellkoordinaten (z.B. Würfel ist beschrieben durch Mittelpunkt $(0,0,0)$ und Kantenlänge 1).
- Die Szene wird beschrieben durch eine Menge von Objekten, deren Lage und Größe in Weltkoordinaten beschrieben sind.
- Die synthetische Kamera wird beschrieben durch $U, V, N, VPD, (x_{\max}, y_{\max}), (x_{\min}, y_{\min})$.

Folgende Abkürzungen werden verwendet:

MC	model coordinate system
WC	world coordinate system
VRC	view reference coordinate system
NPC	normalized projection coordinate system
DC	device coordinate system

Jedes Polygon durchläuft die folgende Viewing-Pipeline:

1.) Modeling: MC $\rightarrow$ WC
Beschreibe Polygonpunkte in Weltkoordinaten (dies geschieht durch Translation, Rotation und Skalierung).

2.) View Orientation: WC $\rightarrow$ VRC
Beschreibe Polygonpunkte bzgl. des UVN-Systems (dies geschieht durch Wechsel des Koordinatensystems, anschließend ist die Szene beschrieben mit xy-Ebene = Bildebene, das Auge liegt bei $z = VPD$).

3.) View Mapping: VRC $\rightarrow$ NPC
Transformiere die Szene derart, daß der sichtbare (= im Pyramidenstumpf aus view window, front plane, back plane liegende) Teil abgebildet wird auf einen Einheitswürfel, dessen Vorder- und Rückseite der *front plane* bzw. *back plane* entsprechen.

4.) Device Mapping: NPC $\rightarrow$ DC
Bilde jeden Bildpunkt auf Gerätekoordinaten ab = übernimm x, y-Koordinaten; z liefert Tiefeninformation.

13.2.1 Modeling-Transformationen

Punkt 1.) wird *Modeling* genannt: Das Anordnen von Objekten aus dem *Modellkoordinatensystem* (MC) zu einer Szene in dem sogenannten *Weltkoordinatensystem* (WC). Im MC liegen die Objekte als Prototypen vor. Ihre Definitionspunkte sind unabhängig von der späteren Größe und Position im WC in Modellkoordinaten gegeben. Der Ursprung des MC befindet sich sinnvollerweise im Zentrum des Objekts. Erst durch das Modeling erhalten die Objekte im WC ihre individuelle Größe, Orientierung und Position. Dieses wird im wesentlichen durch drei Arten von Transformationen realisiert: Translation, Rotation und Skalierung.

13.2.2 View Orientation

Zur Abbildung einer dreidimensionalen Szene aus den Weltkoordinaten auf den Bildschirm muß eine natürliche Betrachtersicht (*View Orientation*) definiert werden. Eine solche Betrachtersicht besteht aus den Viewing-Parametern

- Betrachterstandpunkt *PRP* (*Projection Reference Point*; auch *Eyepoint*),
- Blickrichtung *VRP* (*View Reference Point*),

- vertikale Orientierung VUV (*View Up Vector*) oder VUP (*View Up Point*),

- Blickwinkel (Brennweite).

Der fiktive Betrachter blickt vom *PRP* in Richtung *VRP* und kippt dabei die Kamera so um die Blickrichtung, daß VUP von ihm aus gesehen vertikal nach oben zeigt. Der Blickwinkel entspricht der Brennweite und regelt den Bildausschnitt (*Zooming*).

Durch diese Parameter wird das VRC (*View Reference Coordinate System*) definiert mit *VRP* als Ursprung und *PRP* auf der positiven z-Achse. Die xy-Ebene wird als *View Plane* bezeichnet und steht senkrecht auf der Blickrichtung. Die y-Achse ist gegeben durch die Projektion von VUP in die View Plane. Deshalb darf VUP nicht parallel zur Blickrichtung sein.

Das VRC hat die gleiche Metrik wie das WC. Es handelt sich also um einen Wechsel des Koordinatensystems. Die entsprechende Matrix, die den Wechsel vom WC ins VRC durchführt ergibt sich, wenn man die Vektoren U, V und N und die homogenen Koordinaten des *VRP* bzgl. des WC als Spaltenvektoren nebeneinander schreibt und diese Matrix anschließend invertiert.

Der Abstand zwischen *VRP* und *PRP* wird als VPD (*View Plane Distance*) bezeichnet.

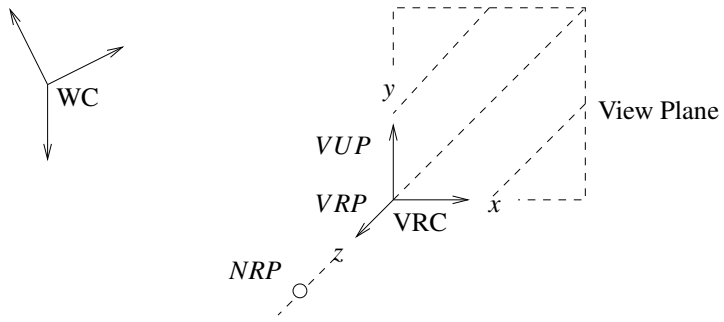


Abbildung 13.3: Definition des VRC

Für die Anwendung des Viewings ergeben sich im Programm verschiedene Möglichkeiten:

- Soll ein Objekt von verschiedenen Seiten betrachtet werden, so wird *VRP* in das Objekt gelegt. Durch *PRP* läßt sich nun die Blickrichtung frei wählen. Der Vektor VUP kann dabei konstant bleiben, sofern nicht senkrecht von oben oder unten geschaut wird.

- Soll sich die Szene bei konstantem Standpunkt um den Betrachter drehen, wird *PRP* festgesetzt. Durch Veränderung des *VRP* bewegt sich dann die Szene vor dem Auge des Betrachters. Der VUP kann dabei konstant bleiben, solange er nicht kollinear zum Vektor durch *VRP* und *PRP* wird.

- Eine weitere Möglichkeit sind Animationen, bei denen sich z.B. der Beobachter vorwärts in die Szene hineinbewegt. Dieser Effekt läßt sich durch Verschieben von *VRP* und *PRP* um denselben Vektor erzielen.

13.2.3 View Volume

Bei einer Kamera ist der maximal abbildbare Ausschnitt einer Szene durch das Objektiv festgelegt. Im Programm wird zur Erzielung desselben Effekts ein rechteckiger Ausschnitt aus der Bildebene — das *Window* — gewählt. Das Seitenverhältnis entspricht dem des Ausgabegeräts. Durch das View Window und die gewählte Projektion wird ein Bildraum definiert, der *View Volume* genannt wird. Nur jene Objekte und Objektteile, die sich innerhalb dieses Bildraumes befinden, werden auf die Bildebene (und dadurch in das View Window) abgebildet.

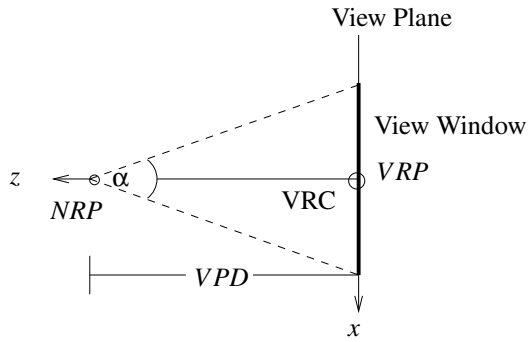


Abbildung 13.4: Definition des View Windows

Die Form des Bildraumes ist bei der perspektivischen Projektion eine unendliche Pyramide, deren Spitze im Projektionszentrum *PRP* (*Projection Reference Point*) liegt und deren Kanten durch die Eckpunkte des View Windows verlaufen. Hierbei wird im Programm der Betrachterstandpunkt *PRP* als Projektionszentrum gewählt.

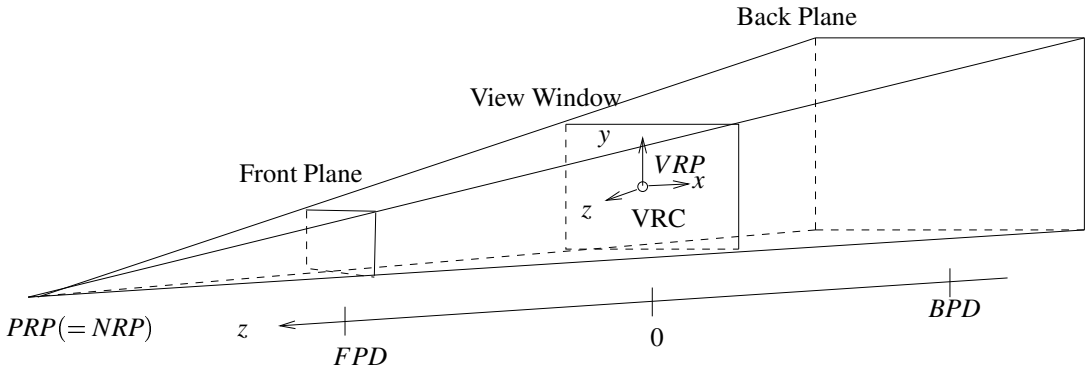


Abbildung 13.5: Bildraum bei perspektivischer Projektion

Die Begrenzung des Bildraumes in Richtung der z-Achse des VRC erfolgt durch zwei zur View Plane parallele Ebenen, die *Front* und *Back Plane* genannt werden. Die Front Plane liegt vom Projektionszentrum PRP aus gesehen vor der Back Plane. Ihre z-Komponenten werden als *Front Plane Distance* (FPD) und *Back Plane Distance* (BPD) bezeichnet. Es gilt $BPD < FPD$. Mit diesen Ebenen können Teile der Szene von der Projektion auf die Bildebene ausgeschlossen werden. Speziell bei der perspektivischen Projektion erweist sich diese Möglichkeit als notwendig, da sonst sehr nahe Objekte alle anderen verdecken bzw. sehr weit entfernte als nicht mehr erkennbare (d.h. zu kleine) Figuren

abgebildet würden. Mit Front und Back Plane ergibt sich als View Volume ein Pyramidenstumpf.

13.2.4 View Mapping

Statt nun das View Volume aus dem VRC direkt auf den Bildschirm zu projizieren, wird im Programm eine weitere Transformation durchgeführt, die nicht nur zur Effizienzsteigerung des Algorithmus führt, sondern auch die Projektion der Szene auf die Bildebene erleichert: Der Bildraum wird in einen zur Bildebene normal ausgerichteten Einheitswürfel $(0 \leq x \leq 1; 0 \leq y \leq 1; 0 \leq z \leq 1)$ umgewandelt. Anstelle mehrerer unterschiedlicher Projektionen muß so anschließend nur noch die orthogonale Parallelprojektion auf die Ebene $z = 0$ durchgeführt werden. Das sogenannte *View Mapping* wird deshalb auch als Ausgabe auf einen normierten Bildschirm bezeichnet. Das Koordinatensystem, in dem sich der Einheitswürfel befindet, heißt *Normalized Projection Coordinate System* (NPC). Der Betrachterstandpunkt befindet sich im Unendlichen auf der positiven z-Achse und hat die homogenen Koordinaten $(0,0,1,0)$. Vom Betrachter aus gesehen liegt im NPC der Punkt $(0,0,0)$ "links unten hinten" und $(1,1,1)$ "rechts oben vorne".

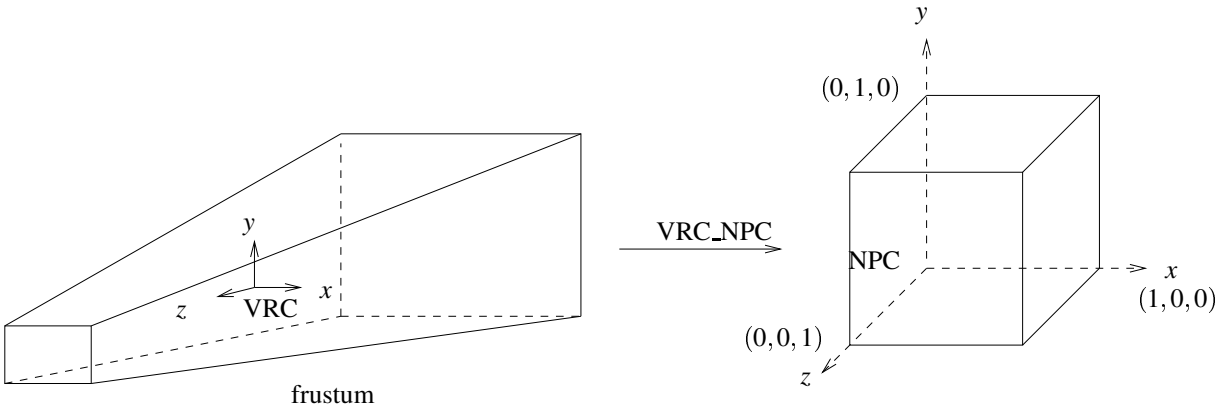


Abbildung 13.6: View Mapping

Bei der perspektivischen Projektion wird in einem ersten Schritt der Pyramidenstumpf, der den Bildraum darstellt und *frustum* genannt wird, auf einen regelmäßigen Pyramidenstumpf abgebildet. Dessen Grundfläche ist ein Quadrat, das mit jeder Seitenfläche einen Winkel von 45° einschließt. Dabei werden die z-Koordinaten nicht und die x- und y-Koordinaten proportional zur z-Koordinate verändert, was einer Scherung an der z-Achse entspricht. Im zweiten Schritt wird der regelmäßige Pyramidenstumpf in den normierten Einheitswürfel transformiert. Dazu müssen die x- und y-Koordinaten proportional zu den reziproken z-Werten skaliert werden. Die Front Plane entspricht im NPC der Ebene $z = 1$ und die Back Plane der Ebene $z = 0$.

Beide Schritte werden im Programm zu einer Transformationsmatrix zusammengefaßt. Bei deren Anwendung auf die homogenen Koordinaten ist zu beachten, daß die resultierenden Punkte im allgemeinen w-Werte ungleich 1 haben, die affinen Koordinaten also erst nach der Division durch w vorliegen. Außerdem ist diese Abbildung nur bezüglich der x- und y-Koordinaten linear. In z-Richtung werden die Werte durch die Scherung im zweiten Schritt reziprok verzerrt, d.h., äquidistante Punkte längs der z-Achse im VRC häufen sich im NPC bei $z = 0$ nahe der Back Plane.

Zur Erleichterung der Herleitung wird zunächst das Koordinatensystem so transformiert, daß der *PRP*

im Ursprung sitzt. Danach wird das Koordinatensystem an der xy-Ebene gespiegelt, indem die z-Koordinaten mit -1 multipliziert werden. Danach ist das Koordinatensystem linkshändig. Es seien $d_{min}, dund d_{max}$ die Abstände der Frontplane, Bildebene und Backplane vom Augenpunkt.

Zur Durchführung von Punkt 3.) der Viewing Pipeline wird zunächst die abgeschnittene Pyramide (= frustum) transformiert in einen symmetrischen Pyramidenstumpf mit quadratischer Grundfläche und Kanten unter 45°.

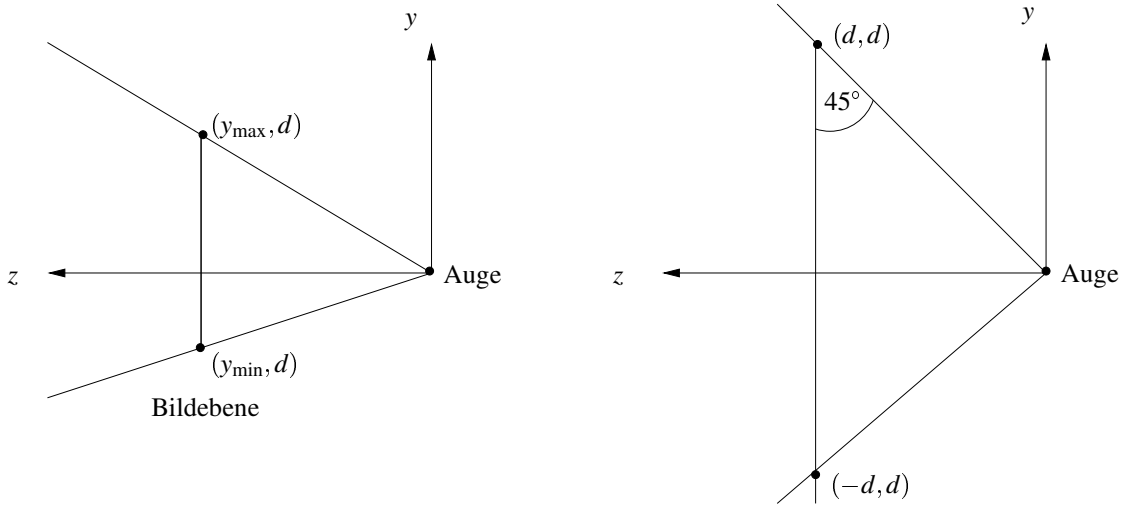


Abbildung 13.7: Überführung in Pyramidenstumpf

Es müssen also die y-Koordinaten proportional zur z-Koordinate verändert werden. Dies entspricht einer Scherung an der z-Achse, so daß die Achse vom *PRP* zum Zentrum des ViewWindows mti der z-Achse zusammenfällt. Zusätzlich wird in y-Richtung so skaliert, daß die Grundseite des entstandenen Pyramidenstumpfs eine Kantenlänge von $2d$ bekommt:

$$\begin{aligned} y' &= k_1 \cdot y + k_2 \cdot z \\ z' &= z \end{aligned}$$

Punkt $(y_{\max}, d)$ soll abgebildet werden auf (d, d);
Punkt $(y_{\min}, d)$ soll abgebildet werden auf $(-d, d)$.

Durch Lösen des Gleichungssystems

$$\begin{aligned} d &= k_1 \cdot y_{\max} + k_2 \cdot d \\ -d &= k_1 \cdot y_{\min} + k_2 \cdot d \end{aligned}$$

erhält man

$$k_1 = \frac{2d}{y_{\max} - y_{\min}}, \quad k_2 = -\frac{y_{\max} + y_{\min}}{y_{\max} - y_{\min}}$$

Analoge Überlegungen für die x-Werte ergibt:

$$k_1 = \frac{2d}{x_{\max} - x_{min}}, \quad k_2 = -\frac{x_{\max} + x_{\min}}{x_{max} - x_{min}}$$

Als nächstes wird die regelmäßige Pyramide in den Einheitswürfel $(0 \leq x \leq 1, 0 \leq y \leq 1, 0 \leq z \leq 1)$ transformiert. Die *front plane* entspricht der Ebene $z = 0$ und die *back plane* der Ebene $z = 1$.

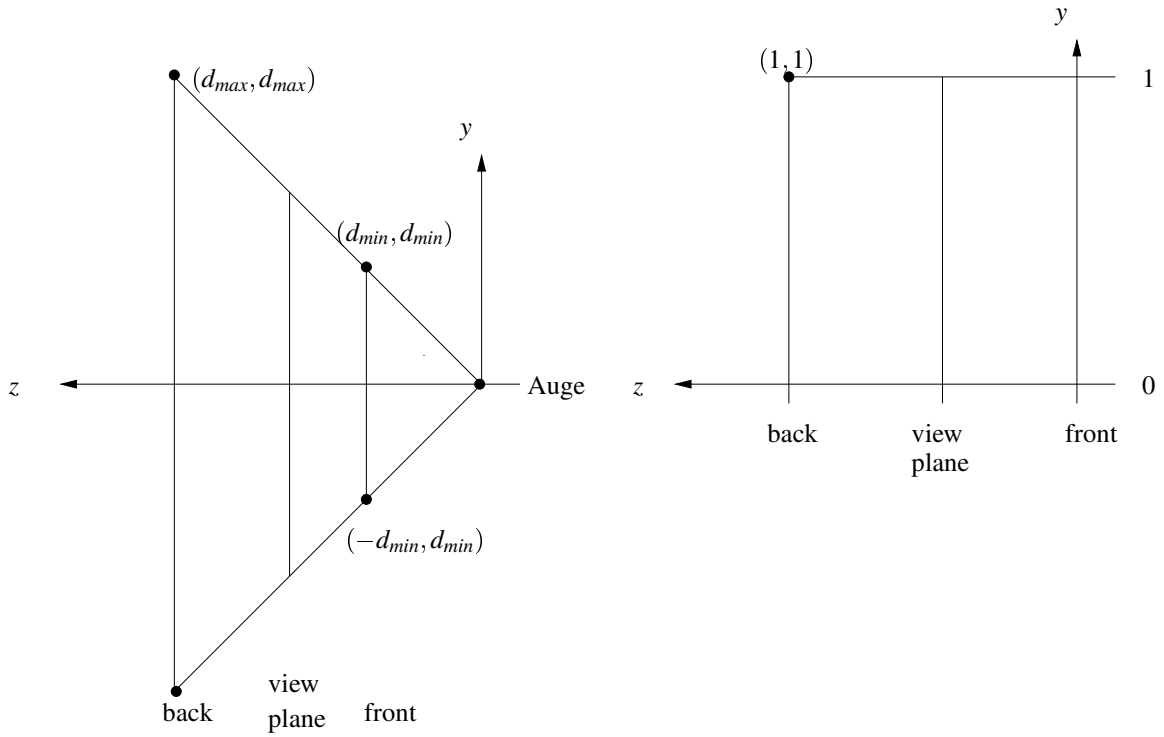


Abbildung 13.8: Überführung in Einheitswürfel

Da die y-Werte proportional zu den reziproken z-Werten skaliert werden müssen, ergibt sich als Transformation

$$\begin{aligned} y' &= k_1 + k_2 \cdot \frac{y}{z} \\ z' &= k_3 + k_4 \cdot \frac{1}{z} \end{aligned}$$

Der Kehrwert von z im Term zu y' kann nur erreicht werden, indem die homogene Koordinate z enthält. Dadurch erhalten aber neben dem Term für y' und dem für x' den Kehrwert von z bei der Transformation, also auch der Term zu z'.

Punkt $(d_{\max}, d_{\max})$ soll abgebildet werden auf $(1,1)$,
Punkt $(-d_{\min}, d_{\min})$ soll abgebildet werden auf $(0,0)$.
Durch Lösen des Gleichungssystems

$$\begin{aligned} 1 &= k_1 + k_2 \cdot \frac{d_{\max}}{d_{\max}} \\ 0 &= k_1 + k_2 \cdot \frac{-d_{\min}}{d_{\min}} \end{aligned}$$

erhält man

$$k_1 = \frac{1}{2}, \quad k_2 = \frac{1}{2}.$$

Durch Lösen des Gleichungssystems

$$\begin{aligned} 1 &= k_3 + k_4 \cdot \frac{1}{d_{\max}} \\ 0 &= k_3 + k_4 \cdot \frac{1}{d_{\min}} \end{aligned}$$

erhält man

$$k_3 = \frac{d_{\max}}{d_{\max} - d_{\min}}, \quad k_4 = -\frac{d_{\min} \cdot d_{\max}}{d_{\max} - d_{\min}}.$$

Die x-Werte werden analog zu den y-Werten skaliert. Insgesamt ergibt sich somit

$$\begin{aligned} x' &= \frac{1}{2} + \frac{x}{2} \cdot \frac{1}{z} \\ y' &= \frac{1}{2} + \frac{y}{2} \cdot \frac{1}{z} \\ z' &= \frac{d_{\max}}{d_{\max} - d_{\min}} - \frac{d_{\min} \cdot d_{\max}}{d_{\max} - d_{\min}} \cdot \frac{1}{z}. \end{aligned}$$

Zwar wird hierdurch die Szene im vorderen z-Bereich nicht-linear gestaucht, zur Bestimmung der Sichtbarkeit reichen die ermittelten z-Werte jedoch aus, da ihre Ordnung erhalten bleibt.

Durch Verknüpfen der beiden letzten Transformationen erhält man in Schritt 3.) als Transformationsmatrix

$$\begin{bmatrix} \frac{d}{x_{\max} - x_{\min}} & 0 & \frac{1}{2}(1 - \frac{x_{\min} + x_{\max}}{x_{\max} - x_{\min}}) & 0 \\ 0 & \frac{d}{y_{\max} - y_{\min}} & \frac{1}{2}(1 - \frac{y_{\min} + y_{\max}}{y_{\max} - y_{\min}}) & 0 \\ 0 & 0 & \frac{d_{\max}}{d_{\max} - d_{\min}} & \frac{-d_{\max} \cdot d_{\min}}{d_{\max} - d_{\min}} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

die den sichtbaren Teil der Szene in den Einheitswürfel transformiert.

Um wieder ein rechtshändiges Koordinatensystem zu erhalten, wird zunächst das Koordinatensystem so transformiert, daß die Back Plane in die xy-Ebene verschoben wird. Abschließend wird wieder an der xy-Ebene gespiegelt.

13.2.5 Device Mapping

Die abschließende Projektion der Szene in Schritt 4.) aus dem NPC auf den Bildschirm wird als *Device Mapping* bezeichnet. Der Einheitswürfel enthält dank der vorangegangenen Transformationen die gesamte darzustellende Szeneninformation. Die Abbildung muß lediglich die

x- und y-Koordinaten aus dem NPC so in die Bildschirmkoordinaten DC (*Device Coordinate System*) transformieren, daß eine anschließende Rundung die ganzzahligen Koordinaten der Pixel ergibt.

DC ist auf den meisten Bildschirmen ein linkshändiges Koordinatensystem. (Die y-Achse zeigt nach unten, oben links ist der Ursprung $(0,0)$.) Die Anzahl der Pixel im Ausgabefenster ist flexibel und betrage horizontal *xsize* und vertikal *ysize*. In x-Richtung muß dann das Intervall $[0,1]$ auf die diskreten Werte $\{0,...,xsize-1\}$ und in y-Richtung $[0,1]$ umgekehrt auf $\{ysize-1,...,0\}$ abgebildet werden. Die z-Koordinaten dienen später zur Bestimmung und Unterdrückung verdeckter Flächen, die sich durch die Staffelung der Objekte in der Tiefe des Bildraumes ergeben.

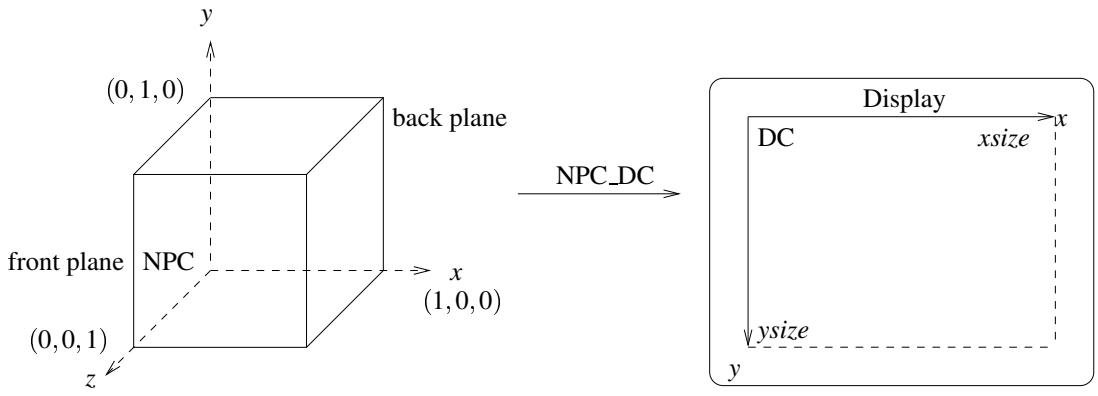


Abbildung 13.9: Device Mapping

Die Transformationsmatrix entspricht einer Skalierung um den Vektor $(xsize, -ysize, 1)$ konkateniert mit einer Translation des Ursprungs in die linke untere Ecke des Bildschirms $(0, ysize, 0)$.

$$T_{\text{NPC_DC}} = \begin{bmatrix} xsize & 0 & 0 & 0 \\ 0 & -ysize & 0 & ysize \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Für die Projektion werden einfach die x- und y-Werte übernommen; die z-Werte sind erforderlich zur Regelung der Sichtbarkeit.

13.2.6 Zusammenfassung

Abbildung 13.10 zeigt die schematische Transformationspipeline im Überblick. Sie wird *ViewingPipeline* genannt. Diese wird bei der Ausführung des Programms von Polygonen durchlaufen, aus denen sich die Elementarobjekte zusammensetzen.

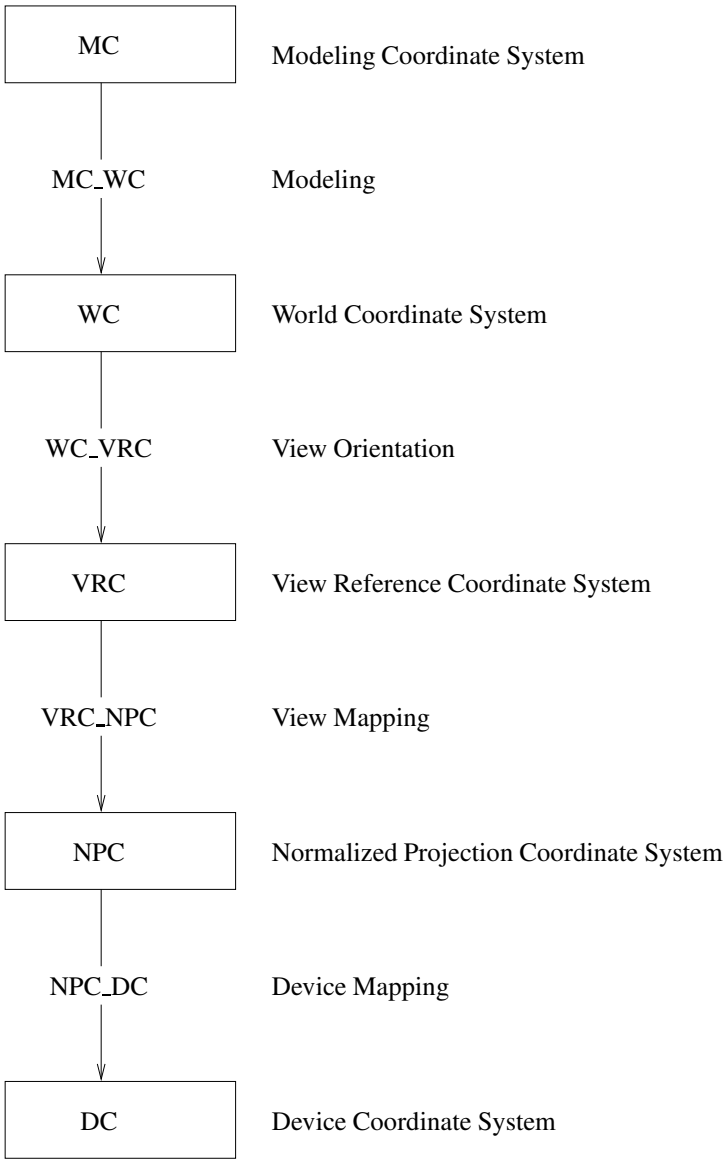


Abbildung 13.10: Transformationspipeline

13.3 Clipping

Jedes Polygon, das die Viewing Pipeline durchläuft, kostet Rechenaufwand zur Transformation seiner Eckpunte und damit Zeit. Noch mehr Zeit kostet das Schattieren des Polygons, das wir später noch

kennenlernen werden.

Da nur der Teil der Szene, der sich im *frustum* befindet, später auf dem Bilbschirm zu sehen sein wird, ist es sinnvoll, den unsichtbaren Teil der Szene so früh wie möglich im Ablauf der Viewing Pipeline "loszuwerden". Im average case dürfen die Punkte als gleichverteilt im Raum angenommen werden. Der unsichtbare Teil der Szene wird also im Allgemeinen einen nicht zu vernachlässigenden Anteil an der Gesamtszene haben.

Um den unsichtbaren Teil auszublenden, müssen wir die gesamte Szene auf irgendeiner Stufe der Viewing Pipeline am *frustum* clippen. Wir wollen zwei Stufen, die sich anbieten näher betrachten:

- Clipping im WC
- Clipping im NPC

Zunächst gehen wir davon aus, däs neben den MC-Koordinaten auch die WC-Koordinaten für jeden Polygonpunkt vorliegen.

13.3.1 Clipping im WC

Diese Strategie beginnt damit, die Ebenengleichungen für die sechs Seiten des *frustums* im WC zu bestimmen. Dies muß im Allgemeinen für jedes Frame neu geschehen. Der Aufwand ist aber unabhängig von der Szene und ihrer Komplexität. Jeder Polygonpunkt wird durch Auswertung der Ebenengleichung an jeder der sechs Seiten geclippt. Wenn ein Polygon eine oder mehrere der Seiten schneidet, müssen noch die neuen Polygonpunkte berechnet werden, indem die Schnittpunkte der Polygonkanten mit den Seiten ermittelt werden.
Nur die verbleibenden Punkte müssen ins DC transformiert und auf dem Bildschirm angezeigt werden. Dazu ist eine Multiplikation mit der Matrix WC_DC notwendig.

13.3.2 Clipping im NPC

Zunächst müssen alle Polygonpunkte mit der Matrix WC_NPC ins NPC transformiert werden. Dann wird die dreidimensionale Erweiterung des 2D-Polygon-Clippings von Cohen und Sutherland durchgeführt. Es wird für jeden Punkt ein 6-Bit-Bereichscode bestimmt, der angibt, ob ein Punkt im View Volume (der Einheitswürfel) liegt oder nicht. Dieser Code läßt sich besonders einfach bestimmen, da die Clippingebenen unabhängig von der Szene und der Lage der Kamera immer die Seiten des Einheitswürfels sind. Es reicht also ein einfacher Vergleich mit den Werten 0 bzw. 1, um ein Bit des Bereichscodes festzulegen. Falls eine Polygonkante eine oder mehrere der Seiten schneidet müssen wider die neuen Polygonpunkte errechnet werden.
Danach müssen die verbleibenden Punkte mit einer weiteren Multiplikation (mit NPC_DC) in Bildschirmkoordinaten gebracht und angezeigt werden.

13.3.3 Vergleich der beiden Vorgehensweisen

Beim Clipping im WC spart man eine Transformation **aller** Punkte im Gegensatz zum Clipping im NPC. Dafür ist das eigentliche Clipping eines Polygon(punkte)s etwas aufwändiger als im zweiten Fall. Auch die Schnittpunktberechnung ist im NPC etwas günstiger als im WC.

Beim Clipping im WC muß das *frustum* noch ins WC gebracht werden. Dieser Vorgang hat konstanten Aufwand und fällt ab einer gewissen Szenengröße nicht mehr ins Gewicht.
Die Zahl der Punkte, die nach dem Clipping übrig bleibt ist in beiden Fällen gleich, damit auch der Aufwand für die abschließsende Transformation.

Man kann nicht entscheiden, welche Strategie die bessere ist, denn die Effizienz der mathematischen Operationen auf dem verwendeten Prozessor bzw. in der verwendeten Programmiersprache spielen eine Rolle.

Der unnötige Aufwand der durch die Transformation der unsichtbaren Punkte ins NPC entsteht läßt sich folgendermaßen abschätzen:

Angenommen die ganze Szene passe in einen Würfel der Kantenlänge g. Dann hat sie ein Volumen von g^3. Der Betrachter stehe im Schwerpunkt des Würfels und blicke so in die Szene, daß das View Window genau einer Würfelseite entspricht. Dann sieht er fünf Sechstel der Szene nicht. Bei gleichverteilten Punkten entspricht das auch fünf Sechstel unnötig vom WC ins NPC transformierten Punkte.

Jede Transformation verursacht 16 Multiplikationen und 12 Additionen. Die Entscheidung, ob ein Punkt im Frustum liegt, kostet im NPC sechs Vergleiche (die Hälfte der Vergleiche findet gegen 0 statt, was vermutlich nochmal schneller als ein beliebiger Vergleich ist). Dieselbe Entscheidung kostet im WC 18 Multiplikationen und 6 Vergleiche. D.h. das Clipping im WC ist pro Knoten zwei Mutliplikationen teurer. Allerding braucht man im NPC zusätzlichen Speicherplatz für die NPC-Koordinaten und den Bereichscode.

Wenn die WC-Koordinaten allerdings nicht vorliegen, sondern für jedes Frame neu aus den MC-Koordinaten errechnet werden müssen, sieht es etwas anders aus. Dann bleibt der Aufwand des Clipping im NPC gleich, denn der erste Schritt besteht jetzt in einer Transformation mit MC_NPC. Der Aufwand des Clipping im WC steigt aber um eine Transformation MC_WC für jeden Polygonpunkt.

Trotzdem hat die parallele Speicherung der WC-Koordinaten ihre Berechtigung. Sie werden für die Beleuchtung benötigt (hier allerdings nur die der sichtbaren Punkte) und viele Datenstrukturen, die die Bestimmung des sichtbaren Teils der Szene wesentlich effizienter machen, arbeiten auf den WC-Koordinaten.

Kapitel 14

3D-Repräsentation

Für 3-dimensionale Objekte gibt es mehrere Möglichkeiten der Repräsentation (d.h. Definition des Objekts) und der Darstellung (d.h. Projektion des Objekts auf den Bildschirm auch *rendering* genannt).

14.1 Übersicht

Repräsentation

- Elementarobjekt mit Definitionspunkten,
- Flächenmodell mit Punkt- und Flächenliste,
- CSG (constructive solid geometry) mit mengentheoretischer Verknüpfung von Elementarobjekten.

Darstellung

- Punktmodell
- Drahtmodell mit sämtlichen Kanten,
- Drahtmodell mit Entfernung verdeckter Kanten,
- Flächenmodell mit Schattierung, ohne abgewandte Flächen,
- Flächenmodell mit Berechnung von Lichtreflektion, ohne verdeckte Teile von Flächen,
- Körpermodell mit Berechnung von Schattenbildung, Spiegelungen und Brechungen.

14.2 Repräsentation

Die Repräsentation dreidimensionaler Objekte in der Computergrafik teilt sich in verschiedene Repräsentationsklassen, die hierarchisch aufgebaut sind.

14.2.1 Elementarobjekte

Für den Benutzer sollte die Beschreibung einer Szene durch *Elementarobjekte* erfolgen. Diese können unterschiedlich kompliziert sein, sollten aber durch wenige Parameter beschrieben werden können. Eine Kugel z.B. ist bereits durch Mittelpunkt und Radius eindeutig im Raum plaziert. Es ist sinnvoll, jedes Objekt in seinem eigenen, lokalen Modellkoordinatensystem zu definieren. Dessen Ursprung wird im Inneren des Objekts gewählt und das gesamte Objekt in ein Einheitsvolumen (z.B. $-1 \leq x, y, z \leq +1$) eingeschlossen. Für Orts- und Größenveränderungen sind Transformationen zuständig.

14.2.2 Drahtmodell

14.2.3 Flächenmodell

Beim *Flächenmodell* werden Objekte durch approximierte oder analytische Flächen, z.B. durch eine Liste von konvexen Polygonen, repräsentiert. Ein solches Polygon wird durch seine Eckpunkte beschrieben, die durch Kanten verbunden sind.

Beim Würfel verbinden die Kanten die Eckpunkte, bei einer Kugel werden die Längen- und Breitenkreise durch n-Ecke angenähert, wobei mit n die Güte der Approximation steigt.

Punkteliste	Flächenliste
$P_1 : (x_1, y_1, z_1)$	$F_1 : p_1, p_2, p_4$
$P_2 : (x_2, y_2, z_2)$	$F_2 : p_1, p_4, p_3$
$P_3 : (x_3, y_3, z_3)$	$F_3 : p_1, p_3, p_3$
$P_4 : (x_4, y_4, z_4)$	$F_4 : p_4, p_2, p_3$

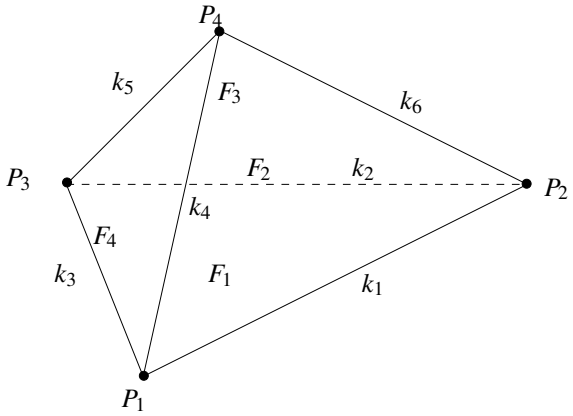


Abbildung 14.1: Tetraeder als Flächenmodell

Zur vollständigen Beschreibung einer Fläche gehört noch die Angabe, welche Seite "innen" und welche Seite "außen" liegt. Dies geschieht durch Angabe des Normalenvektors: Er steht senkrecht auf der Fläche und zeigt von innen nach außen. Für die Approximation gekrümmter Flächen wird häufig pro Eckpunkt eine Normale verwendet.

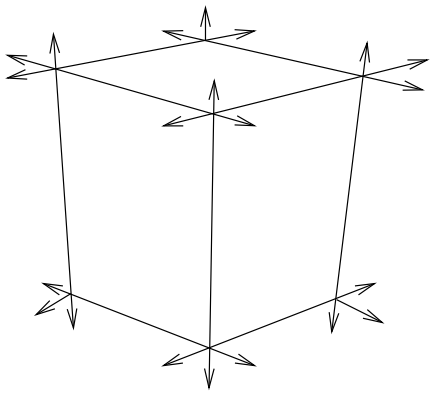


Abbildung 14.2: Würfel mit Normalenvektoren

14.2.4 CSG (constructive solid geometry)

Jedes Objekt wird beschrieben durch einen binären Baum, dessen Blätter beschriftet sind mit Elementarobjekten und dessen innere Knoten beschriftet sind mit den Mengenoperationen $\cup$ (Vereinigung), $\cap$ (Durchschnitt), $\setminus$ (Differenz).

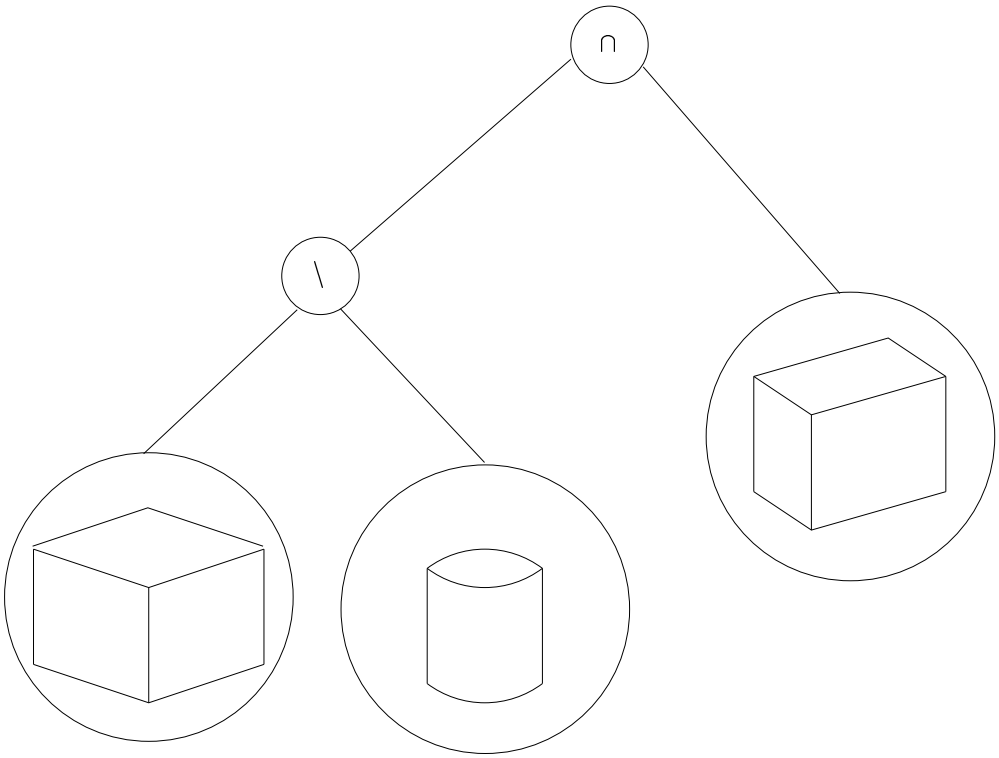


Abbildung 14.3: CSG-Repräsentation eines Körpers

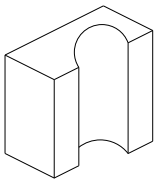


Abbildung 14.4: Darstellung des resultierenden Körpers

Kapitel 15

3D-Objekte

Im Flächenmodell werden Objekte durch ihre Begrenzungsflächen beschrieben und diese wiederum durch ihre Eckpunkte im Modellkoordinatensystem. Außerdem wird noch ein Normalenvektor im MC benötigt. Auf diese Weise kann ein Programm Polyeder exakt darstellen und gekrümmte Oberflächen durch konvexe Polygone beliebig gut approximieren.

15.1 Polyeder

Abbildung 15.1 zeigt eine vom Projektionsalgorithmus erzeugte Szene mit Würfel und Tetraeder in der Drahtmodell-Darstellung mit gestrichelten Rückkanten.

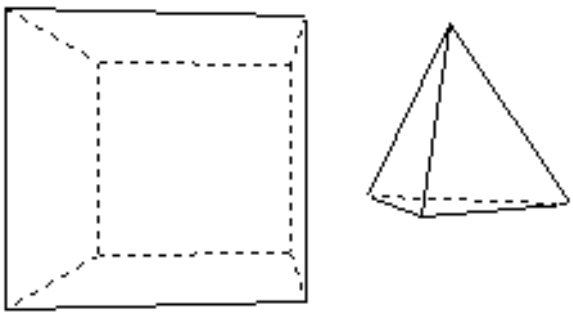


Abbildung 15.1: Vom Projektionsalgorithmus berechnete Drahtgitter-Darstellung

Am Beispiel des Würfels soll die Repräsentation von Polyedern im Flächenmodell erläutert werden.

15.1.1 Würfel

Der *Würfel* mit der Kantenlänge 1 hat im MC die homogenen Eckpunktkoordinaten

$$\begin{array}{llll} (+0.5, & +0.5, & +0.5, & 1), \\ (+0.5, & -0.5, & +0.5, & 1), \\ (-0.5, & +0.5, & +0.5, & 1), \\ (-0.5, & -0.5, & +0.5, & 1), \end{array} \qquad \begin{array}{llll} (+0.5, & +0.5, & -0.5, & 1), \\ (+0.5, & -0.5, & -0.5, & 1), \\ (-0.5, & +0.5, & -0.5, & 1), \\ (-0.5, & -0.5, & -0.5, & 1). \end{array}$$

Da jede der sechs Flächen des Würfels eben ist, gibt es pro Fläche nur einen Normalenvektor, der durch das Kreuzprodukt zweier benachbarter Kanten berechnet werden kann. Die Reihenfolge der Punkte ist dabei signifikant, da jeweils aufeinanderfolgende Punkte eine Kante bilden. Ein *Quader* entsteht aus dem Würfel durch ungleichmäßige Skalierung beim Modeling.

Auch alle anderen Polyeder werden im MC in der Flächendarstellung definiert, ein Tetraeder beispielsweise mit Kantenlänge 1 und Schwerpunkt im Ursprung.

15.2 Gekrümmte Flächen

Eine gekrümmte Oberfläche, die analytisch beschrieben werden kann, läßt sich beliebig gut durch Polygone approximieren. Die Gruppe der Objekte mit analytisch gekrümmter Oberfläche besteht aus zwei Typen: Während z.B. die Kugel nur aus einer solchen Fläche besteht, sind Zylinder und Kegel aus mehreren Flächen zusammengesetzt.

Die Repräsentation der gekrümmten Körper erfolgt im Flächenmodell mit einer Flächen- und Eckpunkteliste sowie einer Normalenliste, die pro Eckpunkt die zugehörige Normale enthält.

15.2.1 Zylinder

Der zur z-Achse symmetrische *Zylinder* mit Höhe 2 und Radius 1 besteht aus einer Mantelfläche und zwei Deckflächen. Die beiden Kreisscheiben bei $z = 1$ und $z = -1$ lassen sich durch regelmäßige n-Ecke darstellen. Die Mantelfläche wird durch n Rechtecke approximiert. Dabei wachsen mit n sowohl die Genauigkeit der Approximation als auch der Rechenaufwand. Sei $\alpha = (2\pi)/n$, so lauten für ein solches Rechteck die vier Eckpunkte und die zugehörigen Normalenvektoren:

Eckpunkt	Normalenvektor
$(\cos(\phi), \sin(\phi), +1, 1)$	$(\cos(\phi), \sin(\phi), 0, 0)$
$(\cos(\phi+\alpha), \sin(\phi+\alpha), +1, 1)$	$(\cos(\phi+\alpha), \sin(\phi+\alpha), 0, 0)$
$(\cos(\phi+\alpha), \sin(\phi+\alpha), -1, 1)$	$(\cos(\phi+\alpha), \sin(\phi+\alpha), 0, 0)$
$(\cos(\phi), \sin(\phi), -1, 1)$	$(\cos(\phi), \sin(\phi), 0, 0)$

mit $\phi = k \cdot \alpha$, $k \in \{0, ..., n-1\}$.

Als Normalenvektor in einem Eckpunkt dieser Fläche wird also der tatsächliche Normalenvektor der Mantelfläche genommen. Dadurch erhalten aneinandergrenzende Flächen in ihren gemeinsamen Eckpunkten auch denselben Normalenvektor. Auf diese Weise erzeugt das Programm bei der späteren Beleuchtung den Eindruck eines stetigen Übergangs zwischen den Flächen. Der Betrachter nimmt statt einer n-eckigen Säule den Zylinder wahr.

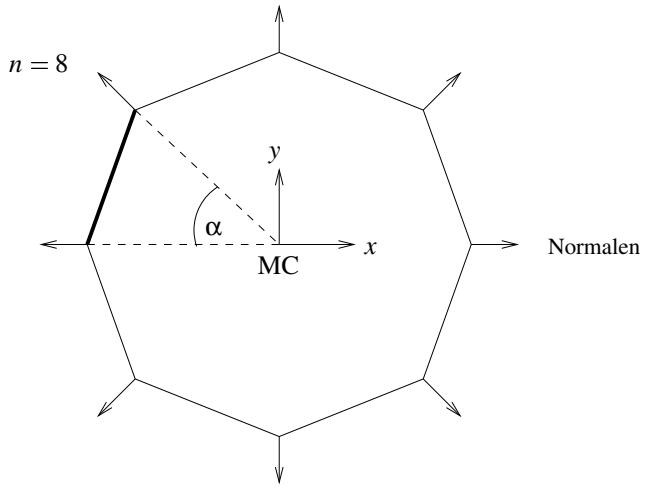


Abbildung 15.2: Zylinder im Grundriß mit Normalenvektoren der Mantelfläche

15.2.2 Kugel

Die Oberfläche einer *Kugel* mit Radius 1 kann beschrieben werden durch

$$(\sin(\theta)\cos(\phi), \sin(\theta)\sin(\phi), \cos(\theta)), 0 \leq \phi < 2\pi, 0 \leq \theta < \pi\,.$$

Zur Approximation durch Flächen wird der Vollwinkel in n Teile zerlegt:

$$\alpha = (2\pi)/n, \;\; n \in \mathbb{N} \text{ gerade}\,.$$

Dadurch entstehen auf der Kugel $n/2$ gleichgroße Längenkreise und $(n/2-1)$ verschieden große Breitenkreise. Diese schneiden n Dreiecke an jedem Pol und $n(n/2-2)$ Vierecke aus der Kugeloberfläche. Die Ortsvektoren der Eckpunkte eines Dreiecks am Nordpol $(\theta = 0)$ lauten

$$(0,0,+1,1), (\sin(\alpha)\cos(\phi), \sin(\alpha)\sin(\phi), \cos(\alpha), 1), (\sin(\alpha)\cos(\phi+\alpha), \sin(\alpha)\sin(\phi+\alpha), \cos(\alpha), 1),$$

mit $\phi = k \cdot \alpha$, $k \in \{0, ..., n-1\}$.

Die Eckpunkte eines der Vierecke haben die Ortsvektoren

$$(\sin(\theta)\cos(\phi), \sin(\theta)\sin(\phi), \cos(\theta), 1), (\sin(\theta)\cos(\phi+\alpha), \sin(\theta)\sin(\phi+\alpha), \cos(\theta), 1), (\sin(\theta+\alpha)\cos(\phi+\alpha), \sin(\theta+\alpha$$

mit $\phi = k \cdot \alpha, k \in \mathbb{N}, k < n$ und $\theta = l \cdot \alpha, l \in \mathbb{N}, 0 < l < (n/2-1)$.

Als Normalenvektor wird in jedem Eckpunkt der Ortsvektor als Richtungsvektor ($w = 0$) eingetragen, denn der Radiusvektor steht senkrecht auf der Kugeloberfläche. Einen *Ellipsoid* erzeugt der Rendering-Algorithmus aus der Kugel durch ungleichmäßige Skalierung beim Modeling.

Abbildung 15.3 zeigt eine vom Projektionsalgorithmus erzeugte Szene mit Kugel und Zylinder in der Drahtmodell-Darstellung ohne Rückkanten ($n = 32$).

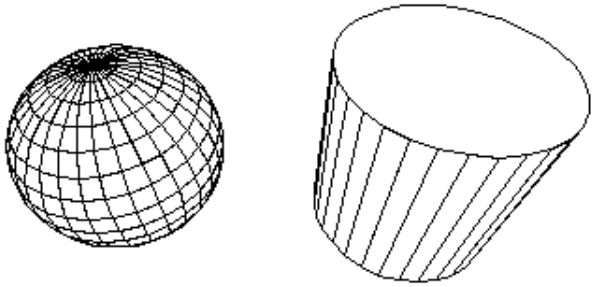


Abbildung 15.3: Kugel und Zylinder als Drahtgitter

15.2.3 NURBS-Flächen

Gekrümmte Flächen können auch mit Hilfe von Kurven beschrieben werden. Da eine Fläche zweidimensional ist, muß eine weitere Parameterdimension hinzugefügt werden. Eine NURBS-Fläche sieht in homogenen Koordinaten folgendermaßen aus:

$$\vec{q}^h(u,w) = \sum_{i=0}^{n}\sum_{j=0}^{m} P^h_{i,j} N_{i,k}(u) M_{j,l}(w)$$

Die $P^h_{i,j}$ sind die vierdimensionalen homogenen Stützpunkte. Die $N_{i,k}$ und die $M_{j,l}$ sind die nichtrationalen B-Spline-Basisfunktionen, laut Gleichung 7.4.1 und 7.4.2. Um affine Koordinaten zu erreichen, wird wieder durch die homogenen Korrdinaten dividiert:

$$\vec{q}(u,w) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} h_{i,j} P_{i,j} N_{i,k}(u) M_{j,l}(w)}{\sum_{i=0}^{n}\sum_{j=0}^{m} h_{i,j} N_{i,k}(u) M_{j,l}(w)} = \sum_{i=0}^{n}\sum_{j=0}^{m} P_{i,j} S_{i,k}(u,w)$$

wobei $P_{i,j}$ die dreidimensionalen Stützpunkte sind. Sie ergeben das *Kontrollnetz* für die Oberfläche. Bei den $S_{i,j}(u,w)$ handelt es sich um die biparametrisierten Flächenbasisfunktionen

$$S_{i,j}(u,w) = \frac{h_{i,j} N_{i,k}(u) M_{j,l}(w)}{\sum_{i1=0}^{n}\sum_{j1=0}^{m} h_{i1,j1} N_{i1,k}(u) M_{j1,l}(w)} = \frac{h_{i,j} N_{i,k}(u) M_{j,l}(w)}{\mathrm{SUM}(u,w)}$$

Die $S_{i,j}(u,w)$ sind nicht das Produkt der $R_{i,k}(u)$ und der $R_{j,l}(w)$ gemäß Abschnitt 7.5. Sie haben aber ähnliche analytische Eigenschaften, so daß die NURBS-Fläche ähnliche analytische und geometrische Eigenschaften, wie die früher erwähnten NURBS-Kurven:

- Die $S_{i,j}(u,w)$ addieren für ein festes Paar von u und w zu 1.
- $S_{i,j}(u,w) \geq 0 \forall u,w$
- Der maximale Grad der Basispolygone ist gleich der Zahl der Kontrollpunkte -1 in der etnsprechenden Dimension.
- Eine NURBS-Kurve der Ordnung k,l (Grad $k-1, l-1$) ist überall C^{k-2}, C^{l-2} stetig.

- NURBS-Flächen sind invariant bgzl. perspektivischer Projektion. Es genügt, die Stützpunkte mit der Viewing Pipeline zu transformieren und die eigentliche Interpolation im DC vorzunehmen.
- Wenn gilt $h_{i,j} \geq 0 \forall i, j$, liegt die Fläche innerhalb der kovexen Hülle des Kontrollnetzes.
- Der Einfluß eines Stützpunktes ist begrenzt auf $\pm k/2, \pm l/2$ Interpolationsabschnitte in jeder Parameterdimension.
- Um die Normale der Fläche in einem beliebigen Punkt zu berechnen, müssen die Richtungsableitungen bzgl. u und w in diesem Punkt gebildet werden.

Algorithmus

```
Spezifiziere die Zahl der Stuetzpunkte in u und w Dimension
Spezifiziere die Ordnung k und l in u und w Dimension
Spezifiziere die Zahl der Interpolationspunkte in jeder Dimension
Spezifiziere das Kontrollnetz (in DC-Koordinaten)
Berechne die Knotenvektoren in u und v Dimension
Fuer jeden Parameterwert u
  Berechne die Basisfunktionen N_ik(u)
  Fuer jeden Parameterwert w
    Berechne die Basisfunktionen M_jl(w)
    Berechne die Summenfunktion
    Fuer jeden Kontrollpunkt in der u Dimension
      Fuer jeden Kontrollpunkt in der w Dimension
        Berechne den Flaechenpunkt q(u,w)
      }
    }
  }
}
Zeichne die Dreiecke (q(u_i,w_j), q(u_i+1, w_j), q(u_i+1,w_j+1))
und (q(u_i, w_j), q(u_i, w_j+1), q(u_i+1, w_j+1))
```

Kapitel 16

Culling Techniken

Unter dem Stichwort Culling werden alle Techniken zusammengefaßt, die zur Entfernung von unsichtbaren Kanten, Flächen und Objekten dienen.

Beim Culling werden unterschieden:

- Objektraum-Algorithmen (arbeiten auf Weltkoordinaten, vergleichen Objekte)
- Bildraum-Algorithmen (arbeiten auf Device-Koordinaten, vergleichen Pixel)

Das zunächst vorgestellte Back-Face Removal entfernt nur die Rückflächen einzelner Objekte, indem es im Objektraum (WC oder NPC) die Beziehung zwischen Betrachter und Objektfläche untersucht. Ob ein Objekt ein anderes verdeckt, kann dadurch nicht festgestellt werden.

16.1 Back-Face Removal/Culling

Ein Körper besteht aus Flächen, die dem Betrachter zugewandt sind (sogenannte Vorderflächen oder Front Faces) und solchen, die vom Betrachter abgewandt sind (sogenannte Rückflächen oder Back Faces). Die Rückflächen sind nicht sichtbar, da sie stets von Vorderflächen verdeckt sind. Die Unterdrückung der Rückflächen (*Back-Face Removal* oder *Back-Face Culling*) ist daher ein erster Schritt in Richtung einer natürlichen Darstellung des Kantenmodells. Für einen konvexen Körper bestimmt das Back-Face Culling exakt den sichtbaren Teil. Ist er dagegen nicht konvex, so werden zwar mehr Kanten angezeigt, als wirklich sichtbar sind, die Darstellung ist jedoch schon sehr realistisch. Ein weiterer Aspekt ist, daß bei einer komplexen Szene circa die Hälfte aller Flächen Rückflächen sind. Durch ihren Ausschluß halbieren sich in etwa die weiteren Berechnungen für Lighting und Shading.

Das Programm benutzt eine sehr effiziente Methode unter Verwendung der Normalenvektoren, um Vorder- und Rückflächen zu unterscheiden. Für die Flächen eines Objekts sind die Normalen so definiert, daß sie nach außen zeigen. Liegt der Betrachterstandpunkt auf der Außenseite einer Fläche, so handelt es sich um eine Vorder-, sonst um eine Rückfläche.

Wenn $\vec{n}$ der Normalenvektor der Fläche und $\vec{a}$ ein Eckpunkt ist, dann kann hieraus die Gleichung der Ebene, in der die Fläche liegt, in der *Hesseschen Normalform* bestimmt werden:

$$\vec{p} \cdot \vec{n} - \vec{a} \cdot \vec{n} = e.$$

Beim Einsetzen verschiedener Punkte $\vec{p}$ ergeben sich unterschiedliche Werte für e. Gilt $e = 0$, so liegt $\vec{p}$ in der Ebene, bei $e > 0$ befindet sich $\vec{p}$ außen, d.h., die Fläche ist von $\vec{p}$ aus sichtbar, und bei $e < 0$ liegt $\vec{p}$ innen, d.h., die Fläche ist von $\vec{p}$ aus unsichtbar.

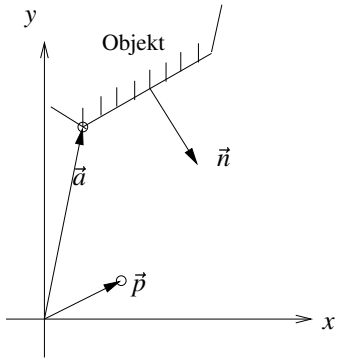


Abbildung 16.1: Back-Face Removal mit Hessescher Normalform

Um die Rückflächen zu erkennen, ist für alle Flächen die obige Ebenengleichung zu berechnen, in die der Betrachterstandpunkt eingesetzt wird. Ergibt sich $e \geq 0$, so handelt es sich um eine Vorderfläche, sonst um eine Rückfläche. Dabei ist zu beachten, daß ggf. für jeden Eckpunkt eine eigene Normale definiert ist. Dies ist zum Approximieren von gekrümmten Flächen notwendig. Bei einer Rückfläche sind definitionsgemäß alle Normalenvektoren abgewandt. Flächen, bei denen einige Normalen zum Betrachter und andere von ihm weg zeigen, werden teilweise dargestellt. Es ist Aufgabe des später vorzustellenden Shaders, die sichtbaren Pixel zu bestimmen. Bei der Liniendarstellung wird eine Kante gezeichnet, wenn mindestens ein Eckpunkt sichtbar ist.

Eine algorithmische Vereinfachung des Tests auf Sichtbarkeit ergibt sich durch die Transformation der Ebenengleichung ins NPC. Hierbei wird ausgenutzt, daß der Betrachterstandpunkt im NPC per Definition die homogenen Koordinaten $(0,0,1,0)$ besitzt. (Er liegt im Unendlichen auf der positiven z-Achse.) Daraus folgt, daß das Skalarprodukt zwischen Ebene und Betrachterstandpunkt im NPC identisch mit der z-Komponente des homogenen Normalenvektors der Fläche ist.

Zur Klassifizierung des Flächentyps wird daher im Programm zunächst nur die z-Komponente der homogenen Normalenvektoren ins NPC transformiert. Durch diese sehr effiziente Methode, Rückflächen zu eliminieren, halbiert sich in etwa die Berechnung zur Darstellung eines Objekts.

16.2 Hidden-Surface Removal

Bei der Darstellung von 3-D-Szenen ist das Problem zu lösen, daß der Betrachter nur die Objekte sehen sollte, die im Vordergrund liegen; alle anderen sind zumindest teilweise verdeckt. Dabei können Objekte sich z.T. selbst verdecken oder zwischen Betrachter und anderen Objekten liegen.

Das Entfernen der nicht sichtbaren Flächen wird als *Hidden-Surface Removal (HSR)* bezeichnet.

16.2.1 z-Buffer-Algorithmus

Der hier vorgestellte *z-Buffer-Algorithmus* löst das Problem nach dem folgenden Prinzip: Für alle Pixel des Bildschirms wird die z-Komponente als Wert für die Tiefe im Raum gespeichert. Die benötigte Datenstruktur, der sogenannte z-Buffer, ist im Programm ein 2-dimensionales Array, das für jedes Pixel den z-Wert des in diesem Punkt dem Betrachter am nächsten liegenden Objekts enthält. In einem gleichgroßen Feld, dem Frame Buffer, werden die Farbwerte der Pixel gespeichert. Die Menge der Farbwerte stellt den Bildschirmspeicher dar.

Der z-Wert wird mit Null und der Farbwert mit der Hintergrundfarbe der Szene initialisiert.

Laut Definition der Transformationspipeline treten nach dem Clipping im NPC nur noch z-Werte zwischen Null und Eins auf. Die Initialisierung mit Null entspricht dem größtmöglichen Abstand vom Betrachter (back plane).

Durch die Rasterung der Flächen erhalten die Pixel auch einen interpolierten z-Wert, der mit dem z-Buffer-Wert an dieser Stelle verglichen wird. Ist der z-Wert des Pixels größer, so ist das Pixel (vorläufig) sichtbar. Sein z-Wert wird in den z-Buffer und sein Farbwert in den Bildschirmspeicher eingetragen. Ist der z-Wert des Pixels kleiner, so ist der zugehörige Teil der Fläche verdeckt; die Inhalte von z-Buffer und Bildschirmspeicher bleiben erhalten. Nach der Abarbeitung aller Flächen enthält der Bildschirmspeicher die Abbildung der sichtbaren Flächen bzw. Flächenteile und der z-Buffer die zugehörige Tiefeninformation.

Der z-Buffer-Algorithmus entscheidet pixelweise über die Verdeckungseigenschaften und ist daher sehr allgemeingültig. Die darzustellenden ebenen Flächen brauchen nicht vorsortiert zu werden und dürfen sich gegenseitig durchdringen. Auch transparente Polygone sind realisierbar, allerdings nur mit großem Zusatzaufwand. Für den Fall, daß das transparente Polygon dem Betrachter am nächsten liegt, müssen Polygone, die *nach* dem transparenten Polygon gerendert werden und *hinter* diesem liegen, durch aufwändige Verknüpfungen eingeblendet werden.

Da der z-Buffer-Algorithmus am Ende der Transformationspipeline im DC arbeitet, wird er als Bildraumalgorithmus klassifiziert.

Der z-Buffer-Algorithmus wird vom Programm in der inneren Schleife des Scanline-Algorithmus aufgerufen und läßt sich wie folgt skizzieren:

```
Für jede Fläche F tue:
       Für jedes Pixel (x, y) auf dieser Fläche tue:
       berechne Farbe c und Tiefe z
       falls z > tiefe[x, y]:
       dann wird c an der Stelle x,y im Frame Buffer eingetragen
       und tiefe[x, y] auf z gesetzt.
```

Ist die Fläche durch $Ax+By+Cz+D=0$ gegeben, so ist die Tiefe im Punkt (x,y):

$$z = -\frac{Ax+By+D}{C}$$

Für auf einer Scanline benachbarte Punkte (x_i, y_j) und (x_{i+1}, y_j) ergibt sich

$$z_{i+1} = z_i - \frac{A}{C}$$

Für die Punkte (x_i, y_j) und (x_i, y_{j+1}) zweier benachbarter Scanlines ergibt sich

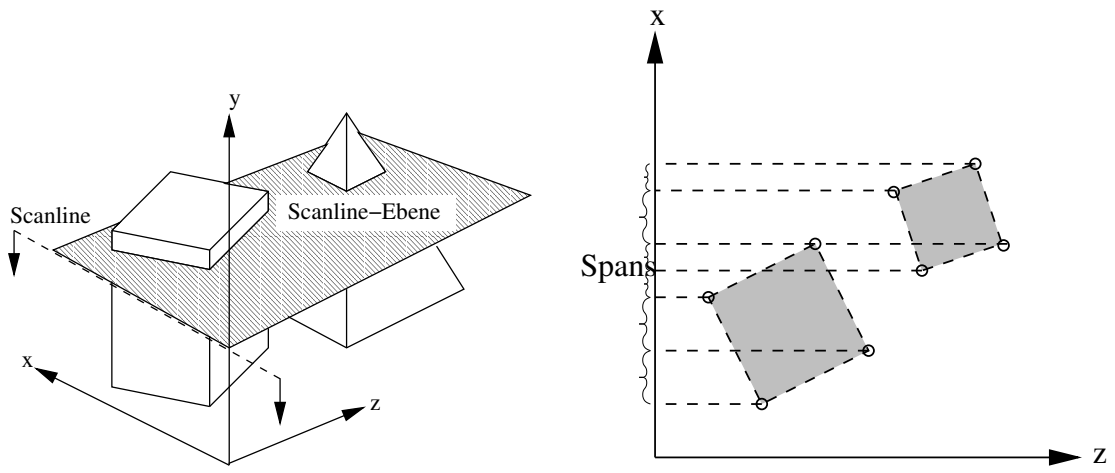
$$z_{j+1} = z_j - \frac{B}{C}$$

Eines der größten Probleme bei der Implementierung des z-Buffers ist sein Speicherplatzbedarf. Eine ausreichende Auflösung der Tiefeninformation ergibt sich erst mit einem 32-Bit z-Wert, da die z-Werte durch die Projektion in den Einheitswürfel nahe der Backplane (also bei $z = 0$) sehr dicht beieinander liegen. Für die RGB-Tripel und den Alpha-Kanal werden vier Byte benötigt, die in einem 32-Bit Integer kodiert werden. Pro Pixel belegen der z-Buffer und der Bildschirmspeicher also acht Byte. Bei einer Bildschirmauflösung von 1024×768 Pixeln ergeben sich folglich 6 MB.

16.2.2 Span-Buffer

Der z-Buffer-Algorithmus muß für jedes Pixel einer Scanline die z-Koordinate berechnen, um zu entscheiden, ob dieses Pixel gesetzt wird oder nicht. Selbst wenn es gesetzt wurde, kann es sein, daß das Pixel später von einem Polygon, das noch näher am Betrachter liegt, überschrieben wird.

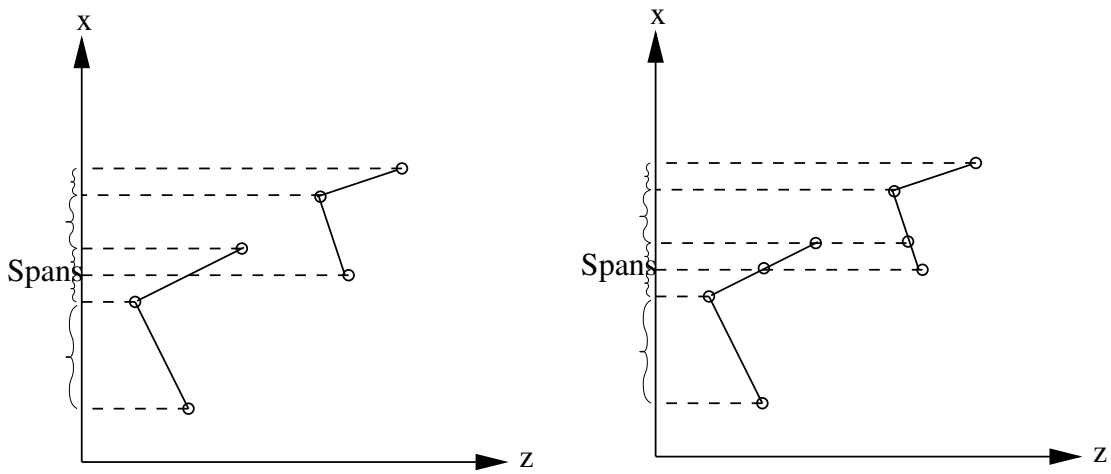
Der Span-Buffer-Algorithmus löst diese beiden Probleme. Normalerweise wird ein Polygon für mehrere aufeinanderfolgende Pixel einer Scanline vorherrschend sein. Ein solcher Scanline-Abschnitt, wird *Span* genannt. Der Algorithmus ermittelt für jede Scanline, welches Polygon in welchem Span vorherrschend ist. Dadurch können die z-Werte der Punkte innerhalb des Spans interpoliert werden und müssen nicht einzeln berechnet werden. Außerdem wird jeder Span genau einmal gezeichnet, es muß also kein Pixel mehrfach eingefärbt werden. Dieser Effizienzsteigerung steht zusätzlicher Aufwand zur Ermittlung der Spans entgegen. Der Abtastzeilenalgorithmus schiebt eine Ebene, die parallel



a) Scanline-Ebene schiebt sich durch die Szene b) Schnittfiguren innerhalb einer Scanline-Ebene

Abbildung 16.2: Pro Scanline wird der Span-Buffer einmal aufgebaut.

zur xz-Ebene ist, die y-Achse herunter (s. Abb. 16.2a). Diese Ebene schneidet ggf. einige Objekte. Die Schnitte mit den Flächen der Objekte sind Punkte oder Strecken (Span). Damit ist das Problem auf zwei Dimensionen (x und z) reduziert (s. Abb. 16.2b).



a) Nur die Spans der Vorderseiten werden betrachtet.

b) Teilung der Spans bzgl. der Endpunkte von anderen Spans.

Abbildung 16.3: Die Schnittfiguren bestimmen die Spans.

Dann werden die Spans der Vorderflächen nach ihren x-Werten sortiert (s. Abb. 16.3a) und anschließend miteinander verglichen. Jeder Span wird an allen - innerhalb seines Intervalls liegenden - x-Werten der anderen Spans in zwei neue Spans zerschnitten (s. Abb. 16.3b).

Hierbei ist zu beachten, daß einander durchdringende Polygone (und damit sich schneidende Spans) *nicht* an ihrem Schnittpunkt in je zwei neue Spans zerlegt werden. Der Algorithmus versagt in diesem Fall und erzeugt keine korrekte Darstellung (s. Abb. 16.4b). Jetzt ist eine Liste von Gruppen mit deckungsgleichen Spans enstanden. Für jede Gruppe muß nur entschieden werden, welcher Span dem Betrachter am nächsten liegt. Dazu werden für die Anfangspunkte der Spans die z-Werte berechnet. Die z-Werte der Endpunkte müssen nicht berechnet werden, da ein Span, der am Anfangspunkt vorherrschend ist, auch am Endpunkt vorherrscht (weil keine Spans existieren, die sich schneiden) (s. Abb. 16.4a).
Die entstandene Liste von Spans wird anschließend noch danach untersucht, ob benachbarte Spans ursprünglich vom selben Polygon stammen. Wenn ja, werden sie wieder zu einem Span zusammengefaßt und erst dann auf dem Bildschirm dargestellt.

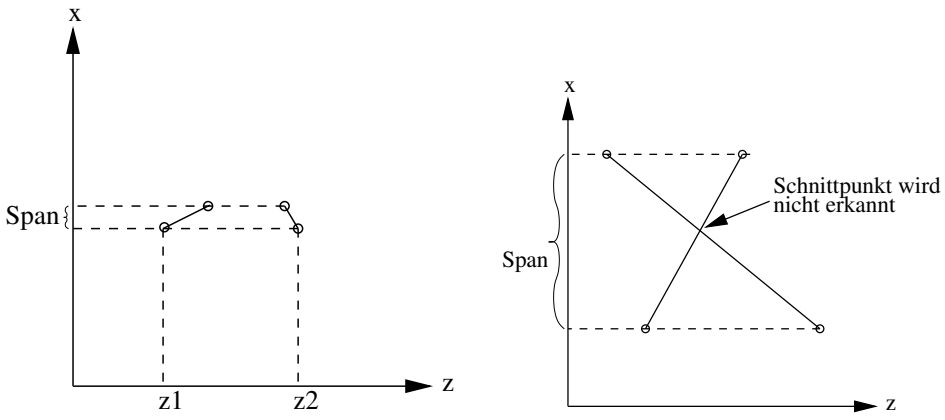
Der Aufwand zur Ermittlung der Spans ist allerdings so hoch, daß der Vorteil gegenüber dem z-Buffer mit steigender Polygonanzahl schwindet.

Falls die Polygone in korrekter z-Sortierung vorliegen, kann man mit Hilfe von einem SpanBuffer-Baum oder einem C-Buffer eine weitere Effizienzsteigerung erreichen.

16.2.3 Binary Space Partitioning

Unter Binary Space Partitioning (BSP) versteht man einen Objektraum-Algorithmus, der die Lage der Objekte untereinander berücksichtigt.

Zunächst wird im Vorhinein ein *BSP-Tree* konstruiert, in dem die räumliche Anordnung der Objekte



a) Zwei Polygone im selben Span. Das linke Polygon ist vorherrschend.

b) Probleme bei sich durchdringenden Seiten.

Abbildung 16.4: Ermittlung des vorherrschenden Polygons.

repräsentiert ist. Dann wird für jedes Frame der Betrachterstandpunkt mit dieser Datenstruktur verglichen, um die Sichtbarkeit der einzelnen Flächen zu ermitteln.

Die binären Knoten des BSP-Trees enthalten jeweils eine $n-1$-dimensionale Hyper-Ebene, die den n dimensionalen konvexen Gesamtraum (C_i in Abb. 16.5b) in zwei konvexe Halbräume (C_{i+1} und C_{i+2}) unterteilt. Zusätzlich enthalten sie zwei Verweise auf Unterbäume, die jeweils einen der beiden Halbräume repräsentieren. Diese rekursive Unterteilung führt im dreidimensionalen Fall dazu, daß der gesamte $\mathbb{R}^3$ durch die Knotenebenen in viele (kleine) konvexe Teilräume zerlegt wird (s. Abb 16.6).

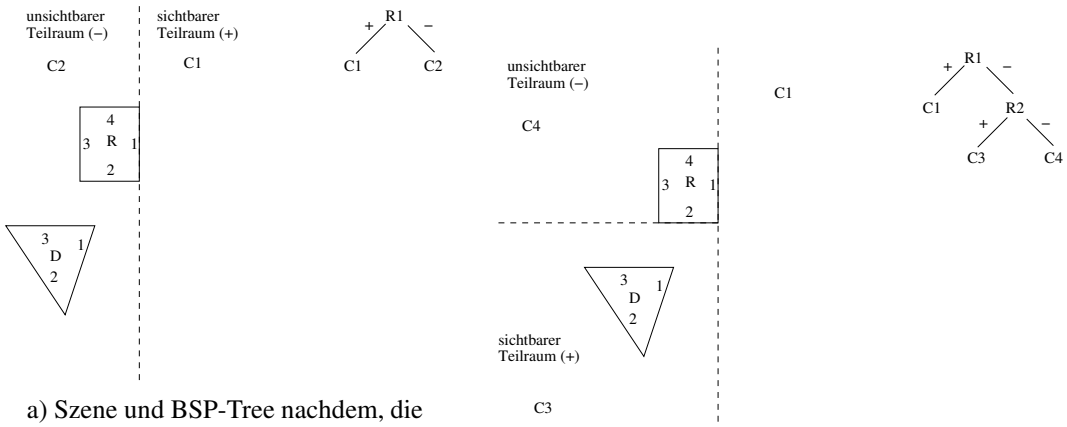
Die Ebenen im Baum sind identisch mit den Flächen der Objekte, d.h. der aktuelle Teilraum des Vaterknotens wird bzgl. dieser Polygonfläche in einen Bereich zerlegt, der ”vor” der Fläche liegt und in einen Bereich, der ”hinter” der Fläche liegt. Alle Punkte im Raum vor der Fläche sind von ihr aus sichtbar (bzw. die Fläche ist von jedem Punkt dieses Raumes aus sichtbar) und alle Punkte im Raum hinter der Fläche sind von der Fläche aus unsichtbar (bzw. die Fläche ist von jedem Punkt des Raumes aus unsichtbar).

Alle Flächen aller Körper der Szene werden in den BSP-Tree eingefügt (s. Abb 16.6). Dabei kommt es vor, daß eine neu einzufügende Fläche eine (oder mehrere) der Knotenebenen schneidet. Die Fläche wird dann an der Knotenebene in zwei Teilflächen zerschnitten, die jeweils komplett in einem der beiden Teilräume liegen (s. Abb. 16.7). Da im average case $O(n \cdot \log n)$ und im worst case $O(n^2)$ Polygone durch Splits enstehen, ist die Konstruktion des BSP-Trees zu aufwändig, um in Echtzeit zu geschehen. Sie wird offline durchgeführt und daher eignet sich der BSP-Tree nur für statische Szenen.

Jedes Blatt des Baums repräsentiert eine (Teil-)Fläche der Szene.

Pro Frame wird der BSP-Tree bzgl. des aktuellen Betrachterstandpunkts traversiert, um die Tiefensortierung zu erreichen. In jedem inneren Knoten gilt:

Die Flächen im Teilraum, in dem sich auch der Betrachter befindet, sind näher am Betrachter als die Flächen, die sich jenseits der Ebene (also im anderen Teilraum) befinden.



a) Szene und BSP-Tree nachdem, die Seite 1 des Rechtecks R als Wurzelebene aufgenommen wurde.

b) BSP-Tree nach Einordnen der Seite 2 des Rechtecks R.

Abbildung 16.5: Einfache Szene aus einem Rechteck (R) und einem Dreieck (D). Erste Schritte beim Aufbau eines 2D-BSP-Trees. Die Knotenhyperebenen sind im 2D-Fall Geraden.

Ziel ist es, eine lineare Liste der Flächen aufzubauen, in der sich die am weitesten vom Betrachter entfernte Fläche am Listenanfang befindet und die am wenigsten entfernte am Listenende (back to front order). Deswegen wird in jedem Knoten zunächst in den Teilraum abgestiegen, in dem sich der Betrachter *nicht* befindet. Erst, wenn alle Flächen aus diesem Raum in die Liste eingefügt wurden, wird der Teilraum, in dem sich der Betrachter befindet, rekursiv untersucht.

Im einfachsten Fall werden anschließend die Polygone gemäß der Liste auf den Bildschirm gezeichnet. Dabei muß beim Setzen eines Pixels kein Test bzgl. der z-Werte mehr durchgeführt werden. Allerdings werden weiterhin viele Pixel mehrfach gesetzt, da die Polygone sich zum Teil überdecken.

Dieser Nachteil kann beseitigt werden, wenn die Liste von hinten abgearbeitet wird. Dann wird das vorderste Polygon zuerst gezeichnet. Beim Zeichnen der weiteren Polygone muß darauf geachtet werden, daß deren verdeckte Teile nicht gezeichnet werden. Dies kann z.B. erreicht werden, indem man beim Zeichnen einen 2D-BSP-Tree aufbaut, indem man alle Polygone anhand der bereits gezeichneten in komplett sichtbare und komplett unsichtbare Teile zerlegt.

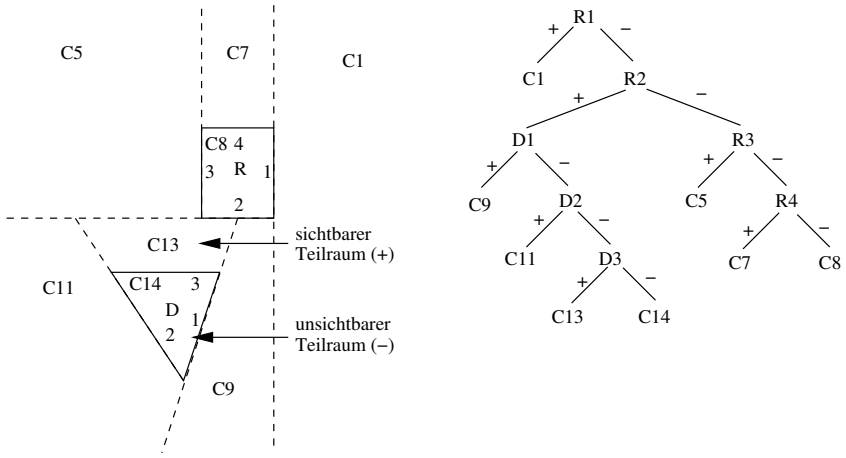


Abbildung 16.6: Gesamte Szene als BSP-Tree; zuletzt wurde Seite 3 des Dreiecks in den Baum aufgenommen. Dabei sind C13 und C14 aus C12 entstanden.

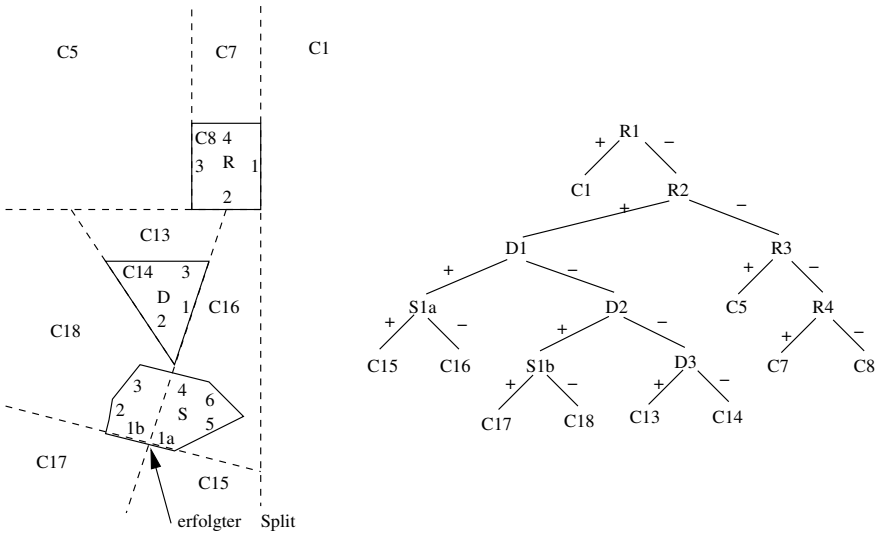


Abbildung 16.7: Neue Szene: Das Sechseck S kommt hinzu. Kante 1 des Sechsecks wurde beim Aufnehmen geteilt. Kante 3 wird auch geteilt werden, wenn sie aufgenommen wird.

Kapitel 17

Rasterung von Flächen

Die reellwertigen Koordinaten der Objekte im DC am Ende der Transformationspipeline müssen in ganzzahlige Pixelkoordinaten gerundet werden. Die Rasterung von Flächen kann mit Hilfe des Scanline-Verfahrens auf die von Linien zurückgeführt werden. In einem zweistufigen Algorithmus wird die Fläche zunächst in eine Menge horizontaler Linien, die sogenannten *Spans*, zerlegt. Die Eckpunkte dieser Spans ergeben sich durch Rasterung der Kanten nach dem vorgestellten Verfahren. In der zweiten Stufe wird dann jeder der Spans gerastert.

Das *Scanline*-Verfahren wird zur Vereinfachung und Beschleunigung des Rendering-Programms nur auf Dreiecke angewandt, denn Dreiecke sind die einfachsten Polygone. Gegenüber allgemeinen Polygonen bieten sie den Vorteil, daß sie planar und konvex sind. Für das Scanline-Verfahren eignen sie sich ausgezeichnet, da für jede Bildschirmzeile (Scanline) maximal zwei Schnittpunkte mit den Dreieckskanten auftreten.

Ein konvexes Polygon läßt sich gemäß der Abbildung triangulisieren, indem von einem beliebigen Eckpunkt aus zu allen nicht benachbarten Eckpunkten Diagonalen gezogen werden.

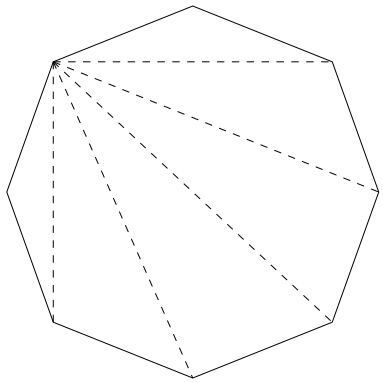


Abbildung 17.1: Triangulisierung eines konvexen Polygons

Unter Rasterung wird hier nicht nur das Füllen der Fläche in der Objektfarbe verstanden. Beim Einsatz des z-Buffers müssen neben den Pixelkoordinaten auch die z-Werte der Polygonpunkte interpoliert werden, um mit Hilfe dieser Tiefeninformation verdeckte Flächen zu unterdrücken.

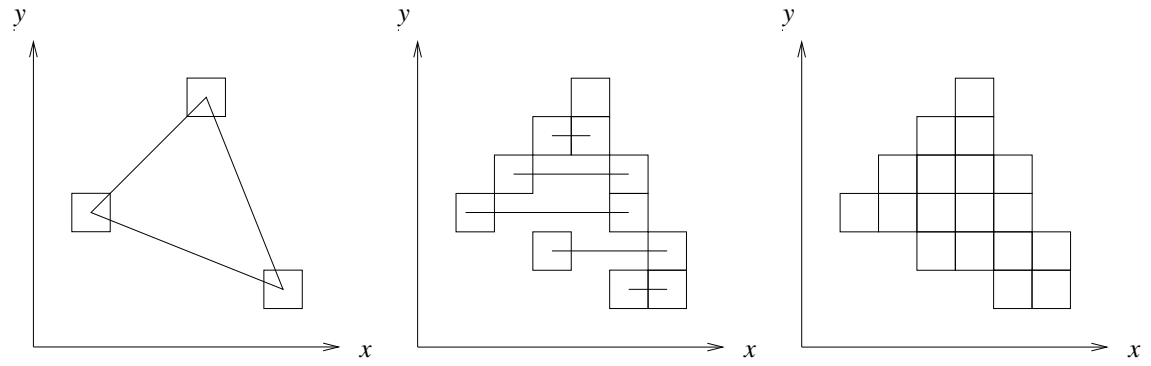


Abbildung 17.2: Scanline-Verfahren zur Rasterung von Dreiecken

Dabei ist zu beachten, daß die Anfangs- und Endpunkte der Spans sich mit einem von Swanson und Thayer 1986 entwickelten Algorithmus berechnen lassen, der dem Bresenham-Algorithmus für Linien ähnlich ist. Er kommt auch ausschließlich mit Ganzzahlarithmetik aus, setzt aber pro Scanline nur genau zwei Punkte und damit weniger als der Bresenham-Algorithmus.

```
public void drawSpanStartPoints(int xa, int xe, int ya, int ye) {
  int dx, dy, x, y, error, mi, mf, schritt;

  dx = xe-xa;
  dy = ye-ya;

  error = -dy;
  mi = dx / dy;
  mf = 2*(dx%dy);
  schritt = -2*dy;

  for(y=ya; y<ye; y++) {
    setPixel(x, y);
    x += mi;
    error += mf;
    if(error > 0) {
      x++;
      error += schritt;
    }
  }
}
```

Außerdem sind je nach Schattierungs-Verfahren auch die Farbwerte bzw. die Normalenvektoren, die in den Eckpunkten gegeben sind, über die Fläche zu interpolieren. Verschiedene Farbwerte auf den Objektflächen ergeben sich durch die individuelle Beleuchtung der Punkte. Die Interpolation der Normalen simuliert dabei einen gekrümmten Flächenverlauf. Grundlegend für die Schattierungs-Verfahren ist die Beleuchtung einer Szene mit Hilfe von unterschiedlichen Lichtmodellen.

17.1 Beleuchtung

Um künstlich hergestellte Bilder wirklich realistisch aussehen zu lassen, muß eine Beleuchtung der darzustellenden Objekte durchgeführt werden. Dieser Prozeß wird in der Computergrafik *Lighting* genannt. Die dabei verwendeten Modelle bilden die Lichtverhältnisse sowie die Oberflächenbeschaffenheiten der Objekte nicht völlig naturgetreu nach, sondern sind nur Näherungen, deren Qualität vom Rechenaufwand abhängt. Die erzeugten Bilder bzw. die eingesetzten Modelle stellen somit einen Kompromiß zwischen Darstellungsgenauigkeit und verfügbarer Rechenzeit dar.

Die Intensität des Lichts, das von einer Oberfläche empfangen wird, hängt von den Lichtquellen in der Umgebung sowie von der Struktur und der Materialart der beleuchteten Fläche ab. Ein Teil des empfangenen Lichts wird vom Körper absorbiert und der Rest reflektiert. Wird das gesamte Licht absorbiert, ist der Körper nicht direkt sichtbar, sondern er hebt sich schwarz vom Hintergrund ab. Erst durch die Reflexion des Lichts wird das Objekt selbst sichtbar. Seine Farbe hängt vom Anteil der absorbierten und reflektierten Farben ab.

In die Berechnung des Farbwertes eines Objektpunktes fließen ein:

- der Augenpunkt des Betrachters,
- die Zahl und Art der Lichtquellen,
- die Materialeigenschaften des Objekts,
- der Normalenvektor des Objekts in diesem Punkt.

17.1.1 Lichtquellen

In den meisten Grafiksystemen werden vier Modelle von Lichtquellen definiert:

- Umgebungslicht (*ambient light*),
- gerichtetes Licht (*directed light* oder *infinite light*),
- Punktlicht (*point light* oder *positional light*),
- Strahler (*spot light*).

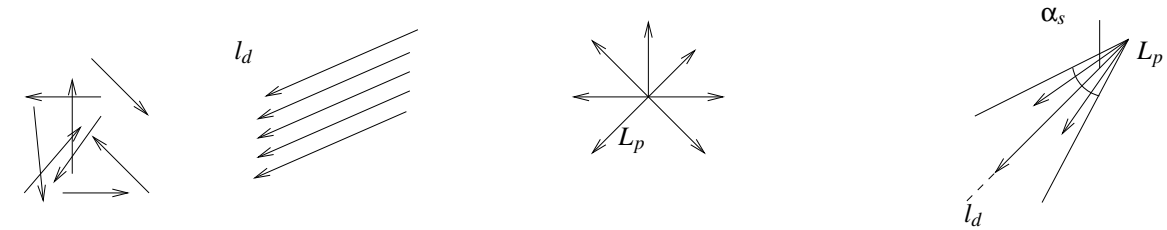


Abbildung 17.3: Vier Modelle von Lichtquellen (schematisch)

Das einfachste Lichtmodell ist das *Umgebungslicht*, das durch eine konstante Intensität I_a definiert ist. Diese ungerichtete Hintergrundbeleuchtung simuliert das Licht, das durch Reflexion an den einzelnen Gegenständen überall in einer Szene vorhanden ist und dessen Quelle nicht feststellbar ist.

Gerichtetes Licht ist definiert durch eine Intensität I und eine Lichtrichtung $\mathbf{l}_d$. Es ist gekennzeichnet durch Lichtstrahlen, die parallel aus dem Unendlichen in eine bestimmte Richtung strahlen, vergleichbar mit dem Sonnenlicht.

Ein *Punktlicht* mit der Intensität I_0 hat eine Position $\mathbf{L}_p$ im Raum (WC), von der aus es in alle Richtungen gleichförmig abstrahlt wie eine Glühbirne. Die Intensität des Lichts nimmt mit zunehmender Entfernung r ab nach der Formel

$$I(r) = \frac{I_0}{C_1 + C_2 \cdot r}.$$

Dabei bezeichnet r den Abstand von der Punktlichtquelle. C_1 und C_2 heißen Abschwächungskoeffizienten (*attenuation coefficients*, $C_1 \geq 1, C_2 \geq 0$).

Das aufwendigste Lichtmodell ist der *Strahler* (*Spot*). Dieser hat wie das Punktlicht eine Intensität I_0, eine Position $\mathbf{L}_p$ sowie die Abschwächungskoeffizienten C_1 und C_2. Dazu kommt eine bevorzugte Lichtrichtung $\mathbf{l}_d$, um die herum das Licht nur im Winkel α_s (*spread angle*) abgestrahlt wird. Außerhalb dieses Lichtkegels ist die Intensität Null. Der *concentration exponent* L_e definiert, wie stark die ausgesandte Lichtenergie mit zunehmendem Winkel zwischen $\mathbf{l}_d$ und der Strahlrichtung $\mathbf{r}$ abnimmt. Die Abnahme ist proportional zu

$$\cos(\mathbf{r}, \mathbf{l}_d)^{L_e}$$

innerhalb des durch α_s definierten Lichtkegels. Je größer L_e, um so stärker ist die Intensität im Zentrum des Lichtkegels gebündelt.

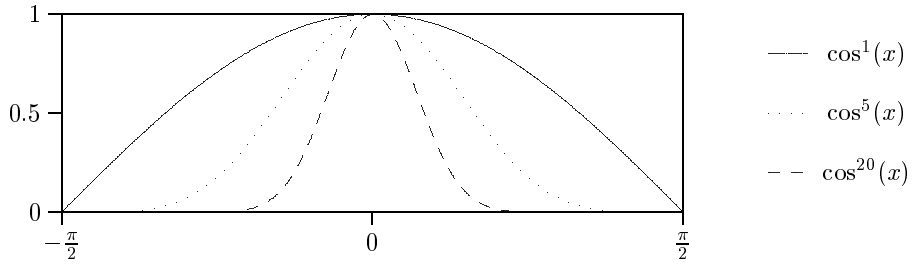


Abbildung 17.4: Lichtintensitätsverteilung beim Strahler (schematisch)

17.1.2 Reflexionseigenschaften

Die Lichtreflexion an Objekten im fast realistischen *Phong-Modell* besteht aus drei Termen:

- ambientes Licht,
- diffus reflektiertes gerichtetes Licht,
- spekular reflektiertes gerichtetes Licht.

Diese Terme werden berechnet und zu einer Farbe zusammengefaßt, um den resultierenden Farbeindruck zu erhalten:

$$\overline{C} = \sum_{i=1}^{n} (\overline{C_{a_i}} + \overline{C_{d_i}} + \overline{C_{s_i}}) \ .$$

Dabei ist $\overline{C}$ das gesamte von einem Punkt der Oberfläche reflektierte farbige Licht, n ist die Anzahl der Lichtquellen, und $\overline{C_{a_i}}, \overline{C_{d_i}}, \overline{C_{s_i}}$ sind die ambienten, diffusen und spekularen Anteile des reflektieren Lichts für die Lichtquelle i. Die Herleitungen müssen im RGB-Modell für jede der drei Grundfarben getrennt betrachtet werden.

17.1.3 Oberflächeneigenschaften

Das Reflexionsverhalten (und damit das Erscheinungsbild) eines Körpers wird durch folgende Oberflächeneigenschaften bestimmt:

- k_a ambienter Reflexionskoeffizient,
- k_d diffuser Reflexionskoeffizient,
- k_s spekularer Reflexionskoeffizient,
- $\overline{O_d}$ diffuse Objektfarbe,
- $\overline{O_s}$ spekulare Objektfarbe,
- O_e spekularer Exponent.

Die *ambiente Reflexion* gibt das Verhalten eines Körpers bei ambienter Beleuchtung wieder. Der ambiente Reflexionskoeffizient k_a gibt an, wieviel des einfallenden ambienten Lichts der Intensität I_a vom Objekt in seiner diffusen Objektfarbe $\overline{O_d}$ reflektiert wird:

$$\overline{C_a} = k_a \cdot I_a \cdot \overline{O_d} \ .$$

Die *diffuse Reflexion* basiert auf dem Phänomen der Streuung. Eintreffendes Licht dringt in den Körper ein und wird von seinen tieferen Schichten gleichmäßig in alle Richtungen gestreut. Das reflektierte Licht hat die diffuse Objektfarbe $\overline{O_d}$. Die Intensität I_e des einfallenden Lichts in einem Punkt ist proportional zum Cosinus des Winkels zwischen der Flächennormale $\mathbf{N}$ in dem Punkt und der Richtung zur Lichtquelle $\mathbf{L}$ (*Cosinusgesetz von Lambert*):

$$I_e \propto \cos(\mathbf{L}, \mathbf{N}) \ .$$

Das ambiente Licht wird hierbei nicht berücksichtigt, weshalb auch von gerichteter Reflexion gesprochen wird. Für einen diffusen Reflektor gilt

$$\overline{C_d} = k_d \cdot I_e \cdot \overline{O_d},$$

wobei k_d angibt, wie stark die diffuse Reflexion ist.

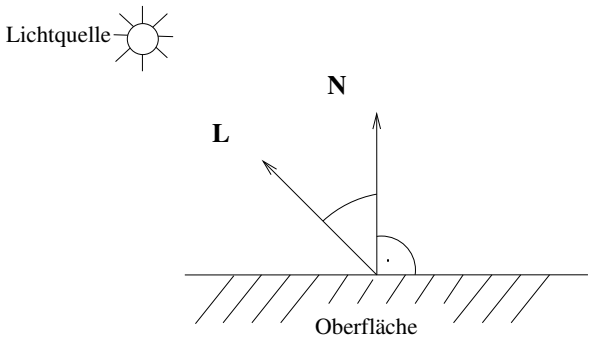


Abbildung 17.5: Cosinusgesetz von Lambert (für diffuse Reflexion)

Die *spekulare Reflexion* beruht auf dem Phänomen der Spiegelung. Eintreffendes Licht wird an der Oberfläche des Objekts gespiegelt (Einfallswinkel = Ausfallswinkel). Das reflektierte Licht hat die spekulare Objektfarbe $\overline{O_s}$, da keine tiefere Wechselwirkung mit dem Objekt stattfindet. Wird das Objekt aus der Reflexionsrichtung betrachtet, so hat es an dieser Stelle einen Glanzpunkt (*Highlight*) in der Farbe $\overline{O_s}$.

Natürliche Materialien sind zumeist keine perfekten Spiegel und strahlen deshalb nicht nur genau in die Reflexionsrichtung, sondern auch mit abnehmender Intensität in andere Richtungen. Von Phong stammt das Modell, daß die abgestrahlte Lichtmenge exponentiell mit dem Cosinus zwischen Reflexionsrichtung **R** und Betrachterrichtung **A** abnimmt. Der spekulare Exponent O_e bestimmt dabei den Öffnungswinkel des Streukegels um **R**, unter dem viel Licht reflektiert wird. Für $O_e \leq 5$ ergeben sich große, für $O_e \gg 10$ kleine Streukegel.

Für einen spekularen Reflektor gilt:

$$\overline{C_s} = k_s \cdot I_e \cdot \overline{O_s} \cdot \cos(\mathbf{R}, \mathbf{A})^{O_e} .$$

Dabei ist I_e die Intensität des einfallenden nicht-ambienten Lichts. k_s gibt an, wieviel von I_e gespiegelt wird.

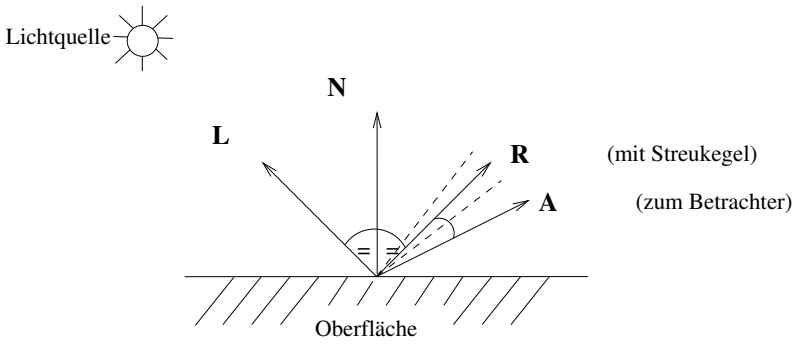


Abbildung 17.6: Spekulare Reflexion nach Phong

17.1.4 Materialeigenschaften

Um im Rendering-Programm Materialeigenschaften nachzuahmen, müssen vom Benutzer insbesondere k_a, k_d und k_s sinnvoll eingestellt werden. Damit ein Objekt nicht mehr Licht aussendet als es empfängt, sollte grundsätzlich $0 \leq k_a, k_d, k_s \leq 1$ gelten, für die Reflexion von gerichtetem Licht zusätzlich $k_d + k_s \leq 1$. Wird k_a im Verhältnis zu k_d und k_s sehr groß gewählt, so ist der beleuchtete Gegenstand sehr kontrastarm. Für matte Gegenstände sollte $k_d \gg k_s$, für spiegelnde $k_s > k_d$ gelten. Je größer O_e ist, um so ähnlicher wird das Reflexionsverhalten dem eines idealen Spiegels.

17.2 Schattierungsalgorithmen

Nach den Grundlagen über die Beleuchtung und die Rasterung von Flächen werden in diesem Abschnitt drei Schattierungsalgorithmen vorgestellt, die mit unterschiedlicher Genauigkeit reale Beleuchtungsverhältnisse nachbilden. Hierbei handelt es sich um inkrementelle Verfahren, die die vorgestellten empirischen Beleuchtungsmodelle verwenden. Die Rasterung der Flächen erfolgt nach der Triangulisierung; deshalb heißen diese Verfahren auch *Dreieck-Shading*-Algorithmen.

17.2.1 Flat-Shading

Das einfachste Schattierungsmodell für ebenflächig begrenzte Objekte ist das *Flat-Shading*. Es verwendet konstante Farbwerte für die einzelnen Dreiecke der Oberfläche. Dazu werden die drei Eckpunkte im WC (wegen der erforderlichen Entfernungen) mit dem vorgestellten Phong-Modell beleuchtet. Nach der Abbildung in die Pixelkoordinaten (DC) werden mit dem Rasteralgorithmus alle Pixel der Dreiecksfläche in der Farbe des Mittelwertes der drei Eckfarbwerte gesetzt:

$$\overline{C_i} = \frac{\overline{C_A} + \overline{C_B} + \overline{C_C}}{3} \; \forall P_i \in \Delta(P_A, P_B, P_C) \; .$$

Dabei bezeichnet $\overline{C_i}$ den RGB-Farbwert für Pixel P_i (siehe Abbildung 17.7).

Beim Flat-Shading wird die Oberfläche des Objekts grob schattiert, die Helligkeitsübergänge sind nicht fließend. Die Qualität der erzeugten Bilder hängt vor allem von der Größe der Dreiecke ab. Das Hauptproblem dieses Verfahrens hängt mit der Eigenschaft des menschlichen Auges zusammen, sprunghafte Intensitätsunterschiede verstärkt wahrzunehmen. Der Randbereich einer helleren Fläche erscheint heller und der Randbereich einer angrenzenden dunkleren Fläche dunkler als der Rest des jeweiligen Polygons. Dieser Effekt wird *Mach-Band-Effekt* genannt.

17.2.2 Gouraud-Shading

Eine Verbesserung der konstanten Schattierung stellt der Algorithmus von Gouraud dar, der die Farbe eines Pixels im Inneren des Dreiecks durch Interpolation der Eckfarbwerte bestimmt. Dazu werden in diesem scanline-orientierten Algorithmus in den Eckpunkten die Farbwerte ermittelt, die dann zur Interpolation entlang der Polygonkanten verwendet werden.

In der Abbildung berechnen sich die Farbwerte der Pixel $P_1(x_1, y)$ und $P_2(x_2, y)$ wie folgt:

$$\overline{C_1} = \overline{C_A}\frac{y - y_C}{y_A - y_C} + \overline{C_C}\frac{y_A - y}{y_A - y_C}, \quad \overline{C_2} = \overline{C_A}\frac{y - y_B}{y_A - y_B} + \overline{C_B}\frac{y_A - y}{y_A - y_B} \; .$$

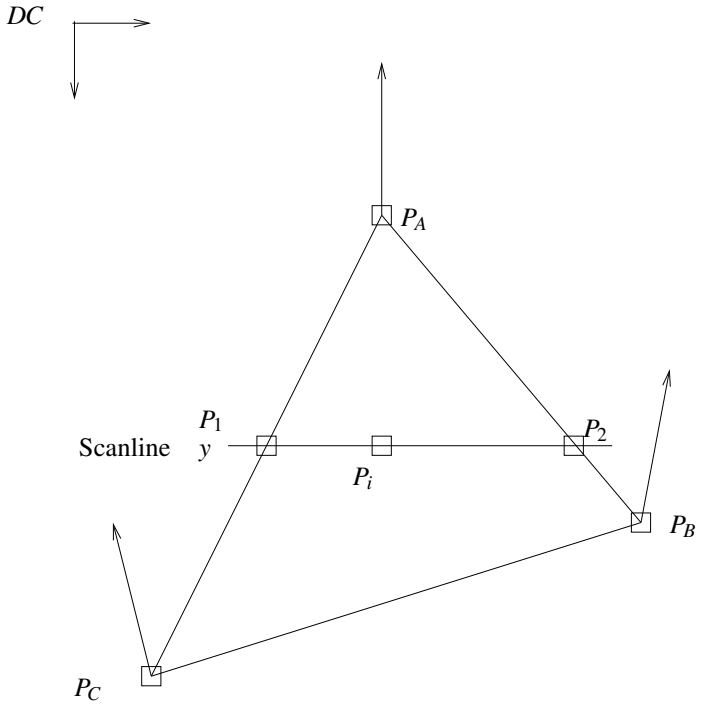


Abbildung 17.7: Dreieck-Shading

Für den Punkt $P_i(x_i, y)$ innerhalb der Fläche ergibt sich die Farbe aus den Randpunkten P_1 und P_2 der Scanline:

$$\overline{C_i} = \overline{C_1}\frac{x_2 - x_i}{x_2 - x_1} + \overline{C_2}\frac{x_i - x_1}{x_2 - x_1} \; .$$

Die Interpolation der Farbwerte ist Teil des Rasteralgorithmus.

Das Gouraud-Shading beseitigt den Mach-Band-Effekt nur zum Teil, da das Auge sogar auf Unstetigkeiten in der zweiten Ableitung der Helligkeitskurve in der beschriebenen Art reagiert. Ein weiterer Nachteil liegt in den verformt dargestellten Spiegelungsflächen, die auf die Polygonaufteilung zurückzuführen sind. Dadurch erscheinen die Highlights der spekularen Reflexion auf großen ebenen Flächen evtl. gar nicht oder auf gekrümmten Flächen verzerrt. Eine Möglichkeit, diese unrealistischen Effekte zu begrenzen, besteht in der Verkleinerung der approximierenden Polygone, was einen höheren Rechenaufwand zur Folge hat.

17.2.3 Phong-Shading

Eine zufriedenstellende Lösung der angesprochenen Probleme läßt sich durch den Algorithmus von Phong erzielen. Basierend auf dem besprochenen Beleuchtungsmodell führt dieser Algorithmus die Farbwertberechnung für jedes Pixel der Dreiecksfläche explizit durch (*per pixel shading*). Die dazu benötigten Normalenvektoren müssen zunächst aus den Normalenvektoren in den Eckpunkten berechnet werden. Für die Fläche in der Abbildung ergeben sich die Normalenvektoren in P_1 und P_2 durch lineare Interpolation der in P_A und P_C bzw. P_A und P_B und daraus wiederum die Normalenvektoren entlang der Scanline. Dabei ist zu beachten, daß für die Beleuchtung die Koordinaten und Normalen

im WC herangezogen werden.

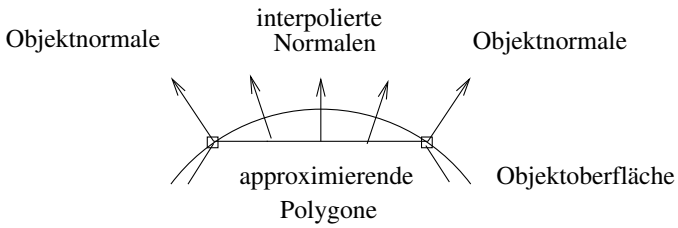


Abbildung 17.8: Interpolation der Normalen beim Phong-Shading

Durch die Interpolation der Normalen ist das Phong-Shading in der Lage, den ursprünglich gekrümmten Verlauf der Oberfläche wiederherzustellen, obwohl das Objekt durch planare Polygone approximiert wird. Dadurch ergibt sich eine fast natürliche spekulare Reflexion mit scharfen Highlights.

Auch mit weniger approximierenden Polygonen ergeben sich bessere Bilder als beim Gouraud-Shading; der Mach-Band-Effekt wird weitgehend unterdrückt. Diese hohe Qualität hat ihren Preis: Statt der Beleuchtung von drei Eckpunkten beim Flat- und Gouraud-Shading müssen beim Phong-Shading alle Pixel des Dreiecks beleuchtet werden,

17.3 Schatten

Zur Berechnung von Schatten eignen sich alle *Hidden-Surface*-Algorithmen, da die verdeckten Flächen einer Szene genau den beschatteten Flächen entsprechen, wenn die Position der Lichtquelle und des Betrachters zusammenfallen.

Zunächst wird daher als Phase 1 aus der Sicht der Lichtquelle die Szene in einen Schattentiefenpuffer abgebildet. Phase 2 berechnet dann für den jeweiligen Betrachtungsstandpunkt die Szene mit einem modifizierten Tiefenpuffer-Algorithmus: Ergibt die Überprüfung des z-Wertes mit dem Eintrag `tiefe[x,y]` im Tiefenpuffer, daß dieses Pixel sichtbar ist, so wird der Punkt $P(x,y,z)$ in den Koordinatenraum von Phase 1 transformiert. Ist die z-Koordinate z' des transformierten Punktes P größer oder gleich dem Eintrag $[x',y']$ im Schattentiefenpuffer, dann liegt der Punkt P nicht im Schatten, andernfalls liegt er im Schatten, und seine Intensität muß entsprechend reduziert werden.
Schatten können auch mit Hilfe von Maps berechnet werden (s. folgendes Kapitel).

Kapitel 18

Texturing

Unter dem Begriff *Texturing* werden neben dem *Texture Mapping* auch alle anderen Verfahren zusammengefasst, bei denen das Aussehen einer Fläche an jedem Punkt mit Hilfe einer BitMap, einer Funktion oder sonstigen Daten verändert wird.

Der dem Texturing zugrunde liegende Prozeß kann in mehrere Schritte aufgeteilt werden:

- Raumkoordinaten des Flächenpunktes berechnen,
- Projektion in den Parameterraum durchführen,
- Texturkoordinaten berechnen
- Texturwerte ggf. anpassen
- Erscheinungsbild mit dem Texturwert modifizieren.

Nachdem zunächst das Texturing anhand der häufigsten Anwendung - dem *Texture Mapping* - beschrieben wird, werden eine Reihe weiterer Texturing-Techniken vorgestellt.

18.1 Texture Mapping

Beispiel: Eine Fläche der Szene soll den Eindruck erwecken als handle es sich um eine Mauer aus Steinen und Fugen. Dazu wird ein Foto von einer solchen Mauer auf die Fläche ”geklebt”. Immer, wenn die Fläche gerastert wird, werden die oben genannten Schritte für jedes Bildschirmpixel der Fläche durchgeführt.

- Die Koordinaten im DC werden für die Eckpunkte durch die Projektion WC_DC und für alle inneren Punkte durch Interpolation zwischen den Eckpunten bestimmt. Es ergibt sich ein Tripel (x, y, z).
- Eine zweidimensionale Textur, wie sie fast immer vorliegt, hat zwei Parameter u und $v \in [0; 1]$, die die Textur parametrisieren. Im zweiten Schritt wird die Projektion der DC-Koordinaten

in den (u,v)-Raum berechnet. Im Mauer-Beispiel würde man eine perspektivische Projektion durchführen, um den korrekten perspektivischen Eindruck zu erhalten. Wenn z.B. eine Weltkarte um eine Kugel "gewickelt" werden soll, um den Eindruck eines Globusses zu erwecken, müsste eine sphärische Projektion gewählt werden. Andere mögliche Projektionsarten sind zylindrisch, elliptisch oder kubisch. Alle Projektionen sind Abbildungen; daher das Wort "Mapping" im Titel der diversen Techniken.

- Im darauffolgenden Schritt wird aus den Werten für u und v das Texture Element oder *Texel* für das aktuelle Pixel bestimmt. Diese Abbildung führt also aus dem Parameterraum in den Teturraum. Wenn die Mauer-Textur beispielsweise 256×256 Pixel groß ist, würden die Werte für u und v mit 256 multipliziert und der Nachkommaanteil abgeschnitten, um die ganzzahligen Koordinaten des Texels zu erhalten. Bei der Wahl dieser *Korrespondenzfunktion (corresponder function)* stehen folgende Typen zur Verfügung:

  - *wrap, repeat, tile*: Das Bild wiederholt sich auf der Fläche. Dies wird erreicht, indem der Vorkommaanteil der Parameterwerte vernachlässigt wird. Diese Art des Texturing wird gewählt, wenn ein Material die ganze Fläche durch Wiederholung überziehen soll. Im Mauer-Beispiel wäre ein Bild ausreichend, das wenige Steine und Fugen enthält. Allerdings muß darauf geachtet werden, daß die Ränder des Bildes oben und unten bzw. rechts und links identisch sind; sonst sind die Nahstellen zwischen den Texturen erkennbar.
  - *mirror*: Wie wrap etc. allerdings wird die Textur abwechselnd im Original und in u- bzw. v-Richtung gespiegelt dargestellt. Dadurch läßt sich die Wiederholung schwerer feststellen.
  - *clamp*: Die Ränder der Textur werden für Werte > 1 wiederholt. Manche APIs erlauben die Einstellung clamp für einen Parameter und wrap für den anderen.
  - *border*: Werte > 1 werden mit einer extra festgelegten Farbe gerendert. Diese Einstellung läßt sich gut verwenden, wenn ein Aufkleber irgendwo auf einer Fläche plaziert werden soll.

- Die Werte, die bei diesen Koordinaten in der Textur hinterlegt sind können auf verschiedene Weise genutzt werden, um das Erscheinungsbild der Fläche zu ändern. Im Mauer-Beispiel wären RGB-Werte in der Textur enhalten. Falls die Werte in der Textur für diese Fläche angepasst werden müssen, kann dies im nächsten Schritt geschehen. Wenn die RGB-Werte beispielsweise zu dunkel sind, könnten sie mit einem Faktor > 1 multipliziert werden, um so eine Helligkeitssteigerung zu erreichen.

- Abschließend wird der Texturwert interpretiert. RGB-Werte aus Bildtexturen ersetzen z.B. die diffuse Objektfarbe in der Beleuchtungsgleichung. Neben der harten Ersetzung der Ursprungswerte, können diese auch moduliert werden. Zu welchen Ergebnissen das führt, wird in den folgenden Abschnitten beschrieben.

Wenn in jedem neuen Frame, das gerendert werden soll, ein neues Bild aus einem Film als Textur benutzt wird, erscheint es so als ob der laufende Film auf die Fläche projiziert wird, wie auf eine Kinoleinwand.
Wenn die Texturkoordinaten in jedem Frame neu festgelegt werden, "wandert" die Textur von Frame zu Frame über die Fläche.

18.2 Mip Mapping

Wenn ein Polygon sich nahe beim Betrachter befindet bzw. weit vom Betrachter entfernt ist, stellt ein Pixel nur einen Teil eines Texels dar bzw. übedeckt ein Pixel mehr als ein Texel. Um den korrekten perspektivischen Eindruck zu erzeugen, muß diese Tatsache durch Antialiasing berüecksichtigt werden. Eine Möglichkeit, dies zu tun ist das Mip Mapping. Damit nicht zur Laufzeit für jedes Pixel die Summe über alle überdeckten Texel gebildet werden muß, werden im Speicher schlechter aufgelöste Versionen der Textur vorgehalten. Jede Version hat ein Viertel der Texelzahl der nächstbesser aufgelösten Version (deshalb sind Texturen mit Zweierpotenzen als Kantenlänge ideal). Das Downsampling bei der Berechnung der Texturen kann mit einem Box-Filter - besser mit einem Gauss-Filter - geschehen.
Diese Texturen sind pyramidenförmig angeordnet; die beste Auflösung als Pyramidenboden; die schlechteste Auflösung (1 mal 1 Texel) als Pyramidenspitze. Die Koordinate vom Boden zur Spitze wird als Level of Detail (LOD) bezeichnet. Je nachdem, wie weit das Pixel vom Betrachter entfernt ist, wird es in die entsprechende Version der Textur projiziert, in der es ungefähr ein (oder zwei) Texel überdeckt.

18.3 Light, Gloss und Shadow Mapping

In einer statischen Szene kann die Auswirkung der Beleuchtung durch statische Lichtquellen (z.B.eine Wandlampe,die die Wand beleuchtet) vorab berechnet werden. Für jede Fläche wird eine Lightmap berechnet, die Werte zwischen 0 und 1 enthält. Mit ihr wird später die Beleuchtungsgleichung multipliziert:

$$C_{gesamt,diffus}[x,y] = C_{lighting,diffus}[x,y] * LightMap[u(x,y),v(x,y)]$$

Damit die Szene dabei nicht zuviel an Helligkeit verliert, kann anschließend mit einem Faktor zwischen 1 und 4 multipliziert werden, wobei maximal reines Weiß als Objektfarbe enstehen darf. Der Vorteil durch die Ersparnis an Rechenzeit ist erheblich.

Es besteht auch die Möglichkeit, eine LightMap in Scheinwerferform (weißer Kreis auf schwarzem Grund) zur Laufzeit über eine Fläche wandern zu lassen. Damit wird der Eindruck erzeugt, daß ein Scheinwerfer die Fläche überstreicht.

Gloss Mapping ist Light Mapping für die spekulare Objektfarbe. Z.B. könnten die Ziegel der Mauer glasiert sein. Dann würde die Fläche nur im Bereich der Ziegel spekulare Highlights zeigen.

Auch bei der Berechnung von Schatten kann eine Effizienzsteigerung erreicht werden, wenn die Szene statisch ist. Wie im vorangegangenen Kapitel beschrieben, wird die Szene aus der Sicht jeder Lichtquelle gerendert und die Schatten, die durch andere Flächen auf eine Fläche fallen, werden als Shadow Map bei dieser Fläche gespeichert.

18.4 Alpha Mapping

Wenn eine Fläche Löcher haben soll (z.B. ein Baum, bei dem zwischen den Blättern der Hintergund zu sehen ist), kann eine Aplha Map darauf plaziert werden. Auch hier sind Werte zwischen 0 (die Fläche ist bei diesem Texel völlig durchsichtig) und 1 (die Fläche ist völlig opak) abgelegt. Wieder wird die Beleuchtungsgleichung modifiziert:

$$C_{gesamt,alpha}[x,y] = C_{Baum,lighting}[x,y] * AlphaMap[u(x,y),v(x,y)] + C_{Hintergrund,lighting} * (1 - AlphaMap[u(x,y),v(x,y)])$$

18.5 Environment oder Reflection Mapping

Im übernächsten Kapitel wird beschrieben, wie beim Raytraycing berechnet wird, welche Objekte sich in anderen spiegeln. Derselbe Effekt kann - etwas weniger akkurat - mit Hilfe von Environment Mapping erzeugt werden. Es wir angenommen, daß die Objekte, die sich in der Fläche spiegeln sollen, weit entfernt sind und daß sich das Objekt, zu dem die Fläche gehört, nicht in sich selber spiegelt. Es wird ein Strahl vom Betrachter zum Pixel der Fläche geschossen und dort reflektiert. Der reflektierte Strahl wird als Verweis in die Environment Map interpretiert und das dort gefundene Texel wird als Modulation der spektularen Objektfarbe benutzt. Die Interpretation des reflektierten Strahls kann z.B. sphärisch, kubisch oder parabolisch geschehen.

18.6 Bump Mapping

Beim Bump Mapping wird durch Veränderung der Flächennormalen anhand einer Bump Map der Eindruck einer rauhen Oberfläche erzeugt. D.h. es wird nicht die Beleuchtungsgleichung direkt modifiziert, sondern indirekt durch Manipulation der Normalen im Punkt (x,y) der Fläche. Diese Vorgehensweise hat keine physikalische Entsprechung.
Es gibt zwei Möglichkeiten, die Normale zu variieren:

- Die Bump Map besteht aus zwei Werten b_u und b_v in jedem Punkt (u,v), die zwei Vektoren skalieren, die senkrecht zueinander und zur Normalen sind und die zur Normalen addiert werden.
- Die Bump Map besteht aus einem Wert pro Texel und die Werte b_u bzw. b_v werden durch Differenzenbildung mit Werten in der Nachbarspalte bzw. Nachbarzeile errechnet.

Per-Pixel Bump Mapping wirkt sehr echt und ist nicht aufwändig zu berechnen. Allerdings werfen die Bumps keine Schatten und wenn der Betrachter flach über eine Fläche hinwegschaut, sind die durch das Bump Mapping suggerierten Höhendifferenzen nicht zu sehen.

18.7 Multitexturing

Jede der Texturing-Techniken für sich betrachtet, erhöht den Realismus der Darstellung. Noch mehr Realismus läßt sich durch die Kombination mehrerer Techniken erzielen. Beim *Multitexturing* werden dazu mehrere Texturen pro Fläche verknüpft. Die Reihenfolge der Verknüpfung und die Art der mathematischen Operationen spielt dabei eine wesentliche Rolle.

18.8 Displacement Mapping

Das Displacement Mapping fällt etwas aus dem Rahmen der anderen Techniken heraus, denn es modifiziert nicht das Erscheinungsbild der Fläche, sondern ihre Geometrie. Dies geschieht, um die

Größe der Datei mit der Szenenbeschreibung klein zu halten. Als Basis dient ein Körper mit geringer Flächenanzahl. Jede Fläche des Körpers wird anhand der Displacement Map in mehrere kleine Flächen unterteilt. Dabei können die neuen Punkte nur entlang der Flächennormalen verschoben werden. Sie werden also entweder aus der Fläche herausgehoben oder in die Fläche hineingeschoben. Die Stärke der Verschiebung ist in der Displacement Map als Grauwert hinterlegt. Diese Technik eignet sich besonders gut, um Landschaften kompakt zu beschreiben (analog zu "Elevation Grid" von X3D) oder um verschiedene Geometrien auf Basis eines Grundkörpers zu definieren.

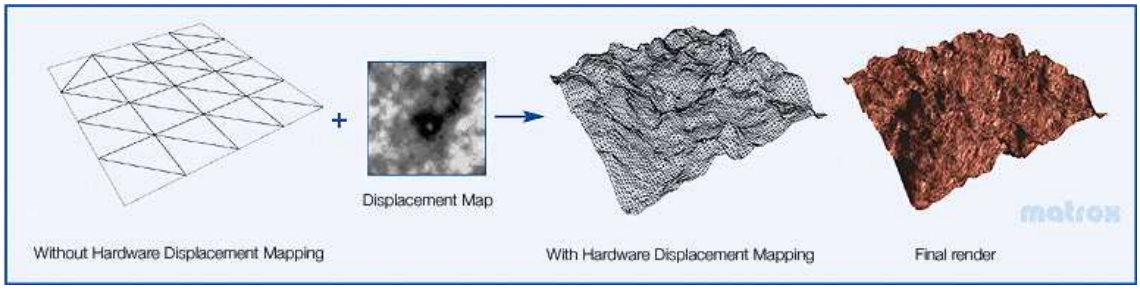


Abbildung 18.1: Displacement Mapping bei der Terraingenerierung

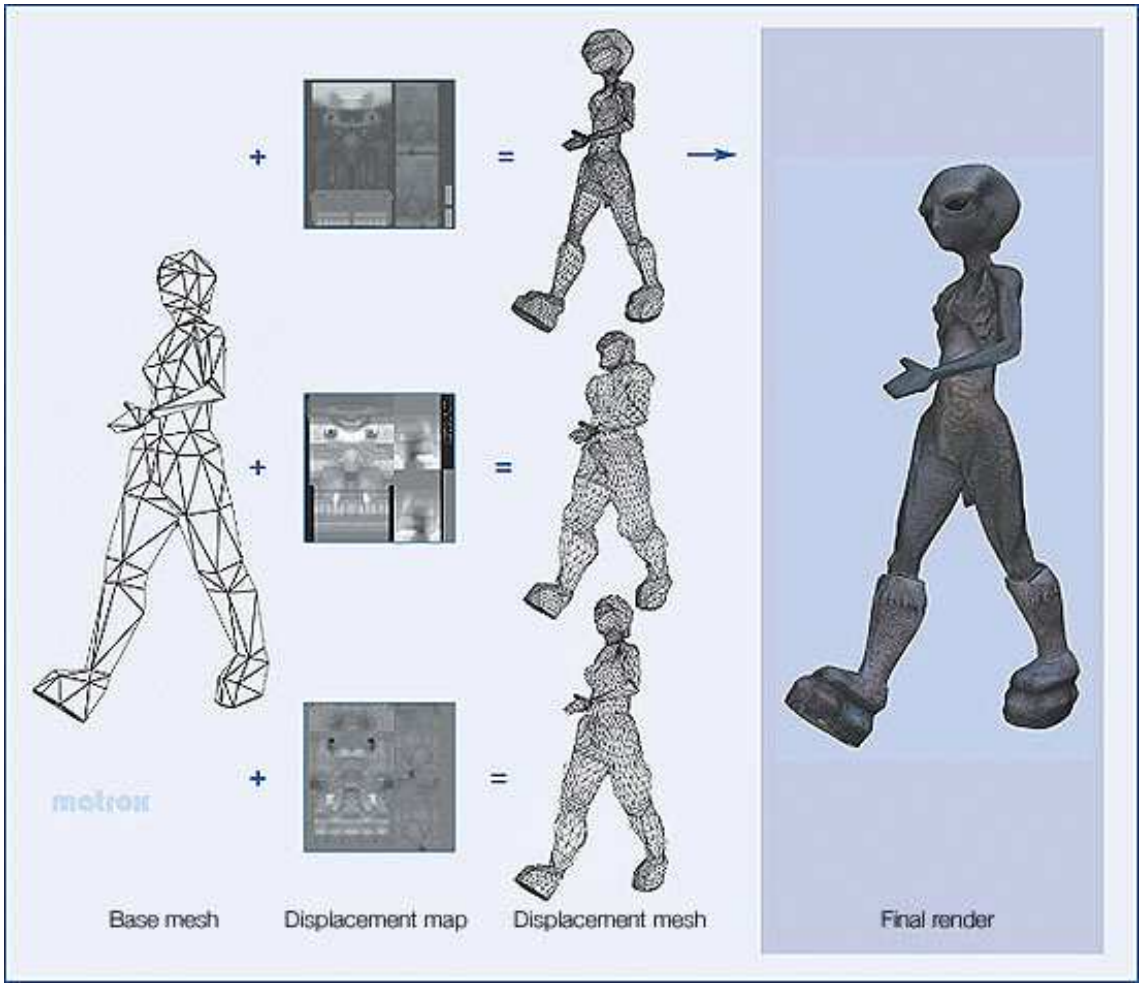


Abbildung 18.2: Generierung von drei Figuren aus einer Gemoetrie mit Hilfe von Displacement Mapping

Kapitel 19

Hardwarebeschleunigung

Trotz aller Optimierungen reicht die Rechenkapazität heutiger CPUs nicht aus, um komplexe Szenen in hoher Auflösung bei hohen Frameraten darzustellen. Deswegen wurden Grafikprozessoren (GPU) entwickelt, die für diese Aufgabe spezialisiert sind. Die Geschwindigkeit dieser GPUs verdoppelt sich zur Zeit etwas alle 6 Monate, was einerseits Moore's Law unterbietet (Verdoppelung der Rechnergeschwindigkeit alle 18 Monate) und andererseits dazu führt, daß es keine Software gibt, die die Fähigkeiten der Grafikchips ausnutzt.

Das aktuelle Top-Modell (GeForce4 Ti 4600) des Marktführers Nvidia für den heimischen PC kann pro Sekunde 136 Millionen Vertices verarbeiten. Dabei kann der Chip 4,8 Milliarden Pixel mit Antialiasing färben. Dazu sind 1,23 Billionen Operationen pro Sekunde notwendig. Diese Geschwindigkeiten werden unter anderem durch Parallelisierung der Viewing Pipeline erreicht.

Ein 3D-Grafiksystem besteht aus drei konzeptuellen Stufen, die nacheinander bei der Berechnung eines Frames durchlaufen werden:

- Applikation: Hier wird die Szene definiert; d.h. die Geometrie wird festgelegt. Dies geschieht durch Auswertung einer hierarchischen Struktur unter Berücksichtigung von Dynamik in der Szene:
  - Objekte kommen hinzu oder werden entfernt,
  - Objekte bewegen sich (insbesondere die Kamera),
  - Objekte kollidieren,
  - Objekte erhalten ein anderes Aussehen,
  - die Beleuchtung ändert sich,
  - die Sichtverhältnisse (Nebel etc.) ändern sich
  - usw.

  Ggf. wird hier die Sichtbarkeit geregelt (z.B. mit BSP-Tree).
- Geometriestufe: Hier werden die meisten pro-Polygon- und pro-Vertex-Operationen durchgeführt. Die Polygone durchlaufen die beschriebene Viewing Pipeline:
  - Model Transform

  - BFC
  - Clipping
  - Lighting
  - View Orientation
  - View Mapping
  - Device Mapping

- Rasterung: Hier werden die sichtbaren Polygone (bzw. Polygonteile) auf Basis ihrer korrekt transformierten, beleuchteten und mit Texturkoordinaten versehenen Vertices auf den Bildschirm gebracht. Dazu muß jedes Polygon in Dreiecke zerlegt (Tesselation) und für jedes Pixel im Bereich eines solchen Dreiecks die Farbe bestimmt werden. Ggf. sind Spezialeffekte wie Nebel, Partikelsysteme und Lens Flares einzublenden. Hier wird ggf. auch die Sichtbarkeit geregelt (Z-Buffer oder Span-Buffer). Damit der Wechsel zwischen zwei Frames nicht störend wirkt, sollte DoubleBuffering benutzt werden.

Grafik Hardware wurde ursprünglich entwickelt, um die letzte Stufe, die Rasterung, zu beschleunigen. Die Interpolation der x-, y-, z-Werte und der Texturkoordinaten und Beleuchtungsparameter entlang der Dreieckskanten bzw. der Scanline lassen sich in Form von hardware wesentlich schneller realisieren. Auch der z-Buffer wird durch Hardwareeinsatz beschleunigt.

Im Laufe der Zeit wurden die GPUs so leistungsfähig, daß sie immer mehr Aufgaben eines 3D-Systems übernehmen konnten. Die mittlere Stufe, in der sich alles um pro-Polygon-Operationen dreht, wird durch heutige Hardware bereits praktisch komplett ersetzt.

Die Applikationsstufe wird nicht durch Hardware ersetzt werden, denn die Abläufe in dieser Stufe sind zu vielfältig, um sie in Hardware umwandeln zu können.

Heutige Hardware beinhaltet (in alphabetischer Reihenfolge)

- 2D- und 3D-Texturing,
- Antialiasing,
- Alpha Blending/Mapping
- API (OpenGL, DirectX)
- Bump Mapping
- Clipping
- Displacement Mapping
- Double und Triple Buffering
- Environment Mapping
- Filterung (bi-, trilinear und (an)isotropisch)
- Lighting

- Light und Shadow Mapping
- Matrixtransformationen
- Mip Mapping
- Multitexturing
- Nebel
- Pixel Shader (programmierbar)
- Schattenberechnung
- Tesselation
- Vertex Shader (programmierbar)
- z-Buffer

Kapitel 20

Radiosity

20.1 Globale Beleuchtung

Ein Beleuchtungsmodell berücksichtigt bei der Berechnung der Farbe für einen Punkt das von den Lichtquellen direkt abgegebene Licht und das Licht, das den Punkt nach Reflexion und Transmission durch seine eigene und andere Flächen erreicht. Dieses indirekt reflektierte und hindurchgelassene Licht heißt *globale Beleuchtung*. Als *lokale Beleuchtung* bezeichnet man das Licht, das direkt von den Lichtquellen auf den schattierten Punkt fällt. Bisher wurde die globale Beleuchtung durch einen Term für ambiente Beleuchtung modelliert, der für alle Punkte auf allen Objekten konstant war. Dieser Term berücksichtigte weder die Positionen von Objekt und Betrachter noch Objekte, die das Umgebungslicht blockieren könnten.

In realen Szenen kommt nur ein geringer Teil des Lichts aus direkten Lichtquellen. Es gibt zwei Klassen von Algorithmen zur Erzeugung von Bildern, die die Bedeutung globaler Beleuchtung hervorheben. Das nächste Kapitel behandelt den Ray Tracing-Algorithmus, der neben der Ermittlung sichtbarer Flächen und deren Schattierung gleichzeitig Schatten, Reflexion und Brechung berechnet. Globale spiegelnde Reflexion und Transmission ergänzen dabei die für eine Fläche berechnete lokale spiegelnde, diffuse und ambiente Beleuchtung. Im Gegensatz dazu trennen die in diesem Kapitel behandelten Radiosity-Verfahren die Schattierung völlig von der Ermittlung sichtbarer Flächen. Sie berechnen erst in einem vom Blickpunkt unabhängigen Schritt alle Interaktionen einer Szene mit den Lichtquellen. Dann berechnen sie mit konventionellen Algorithmen zur Ermittlung sichtbarer Flächen und Schattierung durch Interpolation ein oder mehrere Bilder für die gewünschten Standpunkte des Betrachters.

Vom Blickpunkt abhängige Algorithmen (z.B. Ray Tracing) diskretisieren die Bildebene, um die Punkte zu ermitteln, an denen die Beleuchtungsgleichung für eine bestimmte Blickrichtung ausgewertet wird. Beleuchtungsalgorithmen, die vom Blickpunkt unabhängig sind (z.B. Radiosity), diskretisieren dagegen die Szene und verarbeiten sie weiter, um genügend Informationen zur Auswertung der Beleuchtungsgleichung an jedem beliebigen Punkt und für jede Blickrichtung zu erhalten. Die Algorithmen, die vom Blickpunkt abhängig sind, eignen sich gut zur Behandlung von Spiegelungen, die stark vom Standpunkt des Betrachters abhängen. Sie erfordern jedoch bei der Behandlung diffuser Phänomene zusätzlichen Aufwand, da sich diese über große Bildbereiche oder zwischen Bildern aus verschiedenen Blickpunkten wenig ändern. Die Algorithmen, die nicht vom Blickpunkt abhängen, modellieren diffuse Phänomene dagegen effizient, haben aber bei der Behandlung von Spiegelungen

einen enorm hohen Speicherbedarf.

20.2 Physikalische Ausgangslage

Dieser Algorithmus geht von der Erhaltung der Lichtenergie in einer geschlossenen Szene aus. Jeder von einer Fläche abgestrahlten oder reflektierten Energie entspricht die Reflexion oder Absorption durch andere Flächen. Die gesamte von einer Fläche abgegebene Energie heißt *Strahlung* oder *Radiosity*. Sie besteht aus der Summe der abgestrahlten Energie, der reflektierten Energie und der Energie, die durch die Fläche hindurchtritt. Ansätze, die die Strahlung der Flächen einer Szene berechnen, heißen daher *Radiosity-Verfahren*. Im Gegensatz zu konventionellen Rendering-Algorithmen berechnen Radiosity-Verfahren erst unabhängig vom Blickpunkt alle Lichtinteraktionen einer Szene. Der Term für das ambiente Licht ist nicht mehr nötig, da sie die Reflexionen zwischen den Objekten genauer behandeln. Dann werden eine oder mehrere Darstellungen gerastert. Dabei fällt nur noch der Aufwand für die Ermittlung der sichtbaren Flächen und für die Schattierung durch Interpolation an.

20.3 Die Radiosity-Gleichung (Beleuchtungsgleichung)

Die bisher betrachteten Schattierungsalgorithmen behandeln die Lichtquellen immer unabhängig von den beleuchteten Flächen. Bei den Radiosity-Verfahren kann dagegen jede Fläche Licht abstrahlen. Daher werden alle Lichtquellen mit einem inhärenten Flächeninhalt modelliert. Die Szene wird in eine endliche Anzahl n diskreter Flächenelemente (*Patches*) zerlegt. Jedes dieser Elemente hat endliche Größe, strahlt über die gesamte Fläche gleichmäßig Licht ab und reflektiert Licht. Wenn man jedes der n Flächenelemente als opaken, diffusen Lambertschen Strahler und Reflektierer betrachtet, gilt für Fläche i

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij}, 1 \leq i \leq n \; .$$

mit

E_i	=	von der Fläche i abgegebene Eigenstrahlung
ρ_i	=	Reflexionsvermögen der Fläche i
n	=	Anzahl der Flächen
F_{ij}	=	Anteil an der von Fläche j abgegebenen Energie, die auf Fläche i auftrifft, genannt *Formfaktor*
B_i	=	gesamte von Fläche i abgestrahlte Energie (Summe aus Eigenstrahlung und Reflexion als Energie pro Zeit und pro Flächeneinheit), genannt *Radiosity* der Fläche i

Die Gleichung besagt, daß die Energie, die eine Flächeneinheit verläßt, aus der Summe des abgestrahlten und des reflektierten Lichts besteht. Das reflektierte Licht berechnet sich aus der Summe des einfallenden Lichts, multipliziert mit dem Reflexionsvermögen. Das einfallende Licht besteht wiederum aus der Summe des Lichts, das alle Flächen der Szene verläßt, multipliziert mit dem Lichtanteil, der eine Flächeneinheit des empfangenden Flächenelements erreicht. $B_j F_{ij}$ ist der Betrag des Lichts, das die ganze Fläche j verläßt und eine Flächeneinheit von Fläche i erreicht.

Zwischen den Formfaktoren in diffusen Szenen besteht eine nützliche Beziehung:

$$F_i \cdot F_{ij} = F_j \cdot F_{ji} \,,$$

wobei F_i und F_j die Flächeninhalte sind.

Umordnen der Ausdrücke liefert

$$B_i - \rho_i \sum_{1 \le j \le n} B_j F_{ij} = E_i \,.$$

Also kann der Austausch von Licht innerhalb der Flächenelemente der Szene durch ein Gleichungssystem ausgedrückt werden:

$$\begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \dots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \dots & -\rho_2 F_{2n} \\ \cdot & \cdot & \cdot & \\ \cdot & \cdot & \cdot & \\ \cdot & \cdot & \cdot & \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \dots & 1-\rho_n F_{nn} \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ B_2 \\ \cdot \\ \cdot \\ \cdot \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ \cdot \\ \cdot \\ \cdot \\ E_n \end{pmatrix}$$

Man beachte, daß der Beitrag eines Flächenelements zu seiner eigenen reflektierten Energie berücksichtigt werden muß (es könnte zum Beispiel konkav sein). Daher haben im allgemeinen nicht alle Terme auf der Diagonalen den Wert Eins.

Wenn man die Gleichung löst, erhält man für jedes Flächenelement einen Strahlungswert. Die Elemente können dann für jeden gewünschten Blickpunkt mit einem gewöhnlichen Algorithmus zur Ermittlung sichtbarer Flächen gerastert werden. Die Strahlungswerte sind die Intensitäten dieses Elements.

Gauß-Seidel-Iterations-Verfahren zur Lösung von linearen Gleichungssystemen

Die i-te Gleichung eines linearen Gleichungssystems $Ax = b$ lautet:

$$\sum_{j=1}^{n} A[i,j]x[j] = b[i] \,.$$

Wenn alle Diagonalelemente von A ungleich Null sind, gilt:

$$x[i] = \frac{1}{A[i,i]} \left(b[i] - \sum_{j \ne i} A[i,j]x[j] \right) \,.$$

Das Iterations-Verfahren startet mit einer Schätzung x_1, die auf der rechten Seite eingesetzt wird zur Berechnung von $x_1[1]$. Im nächsten Schritt wird zur Berechnung von $x_1[2]$ bereits $x_1[1]$ benutzt. Ein Iterationsschritt lautet also:

$$x_k[i] = \frac{1}{A[i,i]} \left(b[i] - \sum_{j=1}^{i-1} x_k[j]A[i,j] - \sum_{j=i+1}^{n} x_{k-1}[j]A[i,j] \right) \,.$$

Bei Konvergenz wird das Verfahren nach k Iterationsschritten abgebrochen, wenn $b - Ax_k$ genügend klein geworden ist.

20.4 Berechnung der Formfaktoren

Die Formfaktoren werden bestimmt mit Hilfe der Gleichung

$$F_{ij} = \frac{1}{F_i} \int_{F_i} \int_{F_j} \frac{\cos(\phi_i)\cos(\phi_j)}{\pi r_{ij}^2} b_{ij} dF_j dF_i$$

mit

ϕ_i	=	Winkel zwischen Normale auf Fläche i und Verbindungslinie zwischen Fläche i und Fläche j
ϕ_j	=	Winkel zwischen Normale auf Fläche j und Verbindungslinie zwischen Fläche i und Fläche j
r_{ij}	=	Entfernung zwischen Fläche dF_i und Fläche dF_j
b_{ij}	=	Blockierungsfunktion, falls Teile von Fläche j aus der Sicht von Fläche i verdeckt sind.
F_i	=	Flächeninhalt von Fläche i

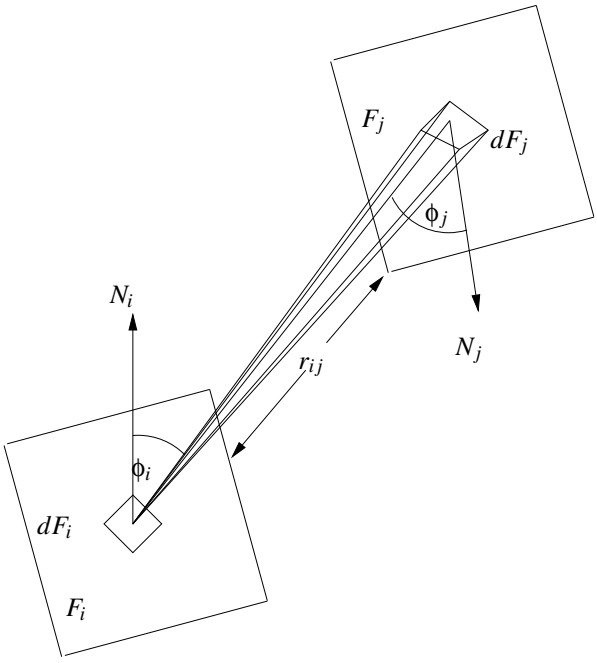


Abbildung 20.1: Der Berechnung von Formfaktoren zugrundeliegende Geometrie

Die geometrische Interpretation beschreibt den Formfaktor als das Verhältnis der Basisfläche einer Halbkugel zur Orthogonalprojektion der auf die Halbkugel projizierten Fläche: Erst projiziert man die von F_i aus sichtbaren Teile von F_j auf eine Halbkugel mit Radius Eins um dF_i, projiziert diese Projektion orthogonal auf die kreisförmige Grundfläche der Halbkugel und dividiert schließlich durch die Kreisfläche. Die Projektion auf die halbe Einheitskugel entspricht in der Gleichung dem Term $\cos\phi_j / r_{ij}^2$, die Projektion auf die Grundfläche entspricht der Multiplikation mit $\cos\phi_i$, und die Division

durch den Flächeninhalt des Einheitskreises liefert den Wert π im Nenner.

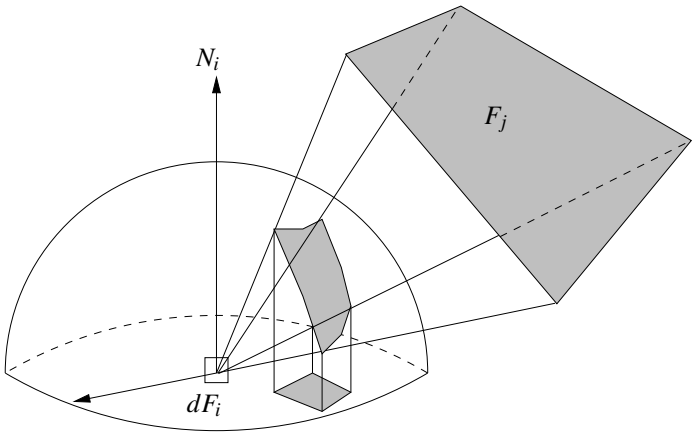


Abbildung 20.2: Geometrische Interpretation des Formfaktors

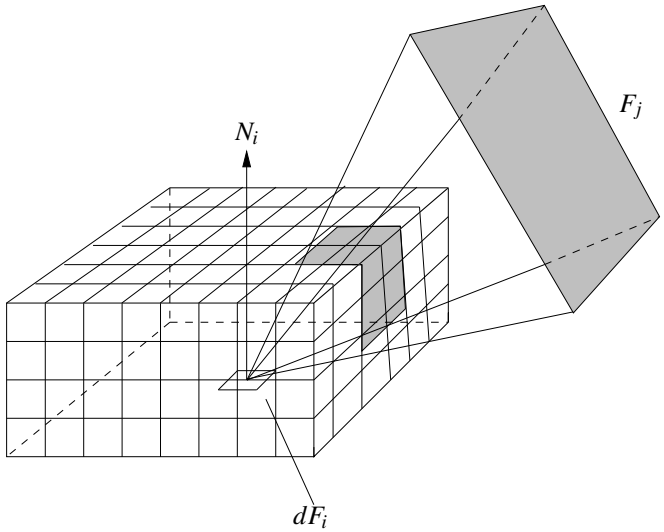


Abbildung 20.3: Simulation der Halbkugel durch Halbwürfel

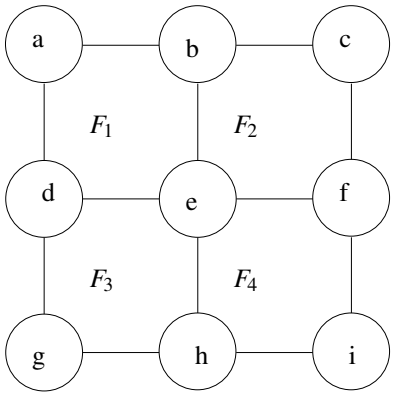
Zur numerischen Berechnung wird die Halbkugel durch einen *Halbwürfel* (hemi-cube) mit dem Zentrum im Ursprung und dem Normalvektor in der z-Achse ersetzt. Die Oberseite des Würfels ist dabei parallel zur Fläche. Jede Seite des Halbwürfels wird in ein Raster gleich großer quadratischer Zellen aufgeteilt. (Die Auflösungen reichen von 50×50 bis zu mehreren Hundert pro Seite.) Dann wird jedes Flächenelement auf die Seiten des Halbwürfels projiziert. Dazu benutzt man einen Algorithmus, der für jede Zeile die Kennung des nächsten schneidenden Elements speichert. Diese Elementpuffer kann man mit einem z-Puffer-Algorithmus für jede Seite des Halbwürfels berechnen, bei dem man für jede Zelle statt der Schattierung die Kennung der nächstliegenden Fläche speichert. Jeder Zelle des Halbwürfels ist ein *Delta-Formfaktor* zugeordnet, der von der Position der Zelle abhängt und vorher berechnet wird. Für eine beliebig feine Rasterung der Quaderoberfläche ergibt sich der Formfaktor F_{ij} als Summe aller von der Fläche F_j überdeckten Rasterzellen. Wird eine Rasterzelle von mehreren

Flächen überdeckt, wird sie der Fläche mit der geringsten Entfernung zugerechnet. Für die Summe aller Formfaktoren einer Fläche F_i gilt

$$\sum_{j=1}^{n} F_{ij} = 1 \ .$$

20.5 Interpolation der Pixelfarben

Für jede Fläche i liege nun ihre *Radiosity* B_i vor. Hieraus interpoliert man die Strahlung für die Eckpunkte:



$$\begin{array}{lcllcl}
B(e) & = & (B_1+B_2+B_3+B_4)/4 & & & \\
B(a) & = & B_1+(B_1-B(e)) & B(c) & = & B_2+(B_2-B(e)) \\
B(g) & = & B_3+(B_3-B(e)) & B(i) & = & B_4+(B_4-B(e)) \\
 & & & & & \\
B(b) & = & (B(a)+(B(c))/2 & B(d) & = & (B(a)+B(g))/2 \\
B(f) & = & (B(c)+(B(i))/2 & B(h) & = & (B(g)+B(i))/2
\end{array}$$

Nun wird jedes Patch in zwei Dreiecke zerteilt, so daß nach der Projektion die Strahlungswerte an den Dreiecksecken zur zweifachen Interpolation verwendet werden können.

20.6 Schrittweise Verfeinerung

In Anbetracht des hohen Aufwands beim bisher beschriebenen Radiosity-Algorithmus stellt sich die Frage, ob man die Ergebnisse der Beleuchtungsberechnung inkrementell approximieren kann. Die Auswertung der i-ten Zeile des Gleichungssystems liefert eine Schätzung für die Strahlung B_i des Flächenelements, die auf den Schätzungen für die Strahlungswerte der anderen Flächenelemente basiert. Jeder Term in der Summe der Gleichung beschreibt die Auswirkung des Elements j auf die Strahlung des Elements i:

$$\text{Beitrag von } B_j \text{ zu } B_i = \rho_i B_j F_{ij} \text{ für alle } j \ .$$

Diese Methode "sammelt" also das Licht der restlichen Szene ein. Der Ansatz zur schrittweisen Verfeinerung verteilt dagegen die Strahlung eines Flächenelements auf die Szene. Dazu bietet sich die

Anpassung obiger Gleichung zu

$$\text{Beitrag von } B_i \text{ zu } B_j = \rho_j B_i F_{ji} \text{ für alle } j\ .$$

an. Wenn man eine Schätzung für B_i hat, kann man den Beitrag des Flächenelements i zum Rest der Szene ermitteln, indem man vorstehende Gleichung für jedes Element j auswertet. Dazu braucht man leider F_{ji} für alle j. Jeder dieser Werte wird mit einem separaten Halbwürfel bestimmt. Dies erfordert ebensoviel Aufwand an Speicherplatz und Rechenzeit wie der ursprüngliche Ansatz. Man kann die Gleichung jedoch umschreiben, wenn man die Reziprozitätsbeziehung berücksichtigt.

$$\text{Beitrag von } B_i \text{ zu } B_j = \rho_j B_i F_{ij} \frac{F_i}{F_j} \text{ für alle } j\ .$$

Zur Auswertung dieser Gleichung für alle j sind nur die Formfaktoren nötig, die mit einem einzigen Halbwürfel um das Flächenelement i berechnet wurden. Kann man die Formfaktoren des Elements i schnell berechnen (z.B. mit z-Puffer-Hardware), kann man sie wieder löschen, sobald die Strahlungen vom Flächenelement i aus berechnet sind. Man muß also immer nur einen einzigen Halbwürfel und dessen Formfaktoren gleichzeitig berechnen und speichern.
Sobald die Strahlung eines Elements verteilt wurde, wählt man ein anderes Element aus. Ein Element kann wieder Strahlung verteilen, sobald es neues Licht von anderen Elementen erhält. Dabei wird nicht die gesamte geschätzte Radiosity des Elements i verteilt, sondern nur der Betrag ΔB_i, den das Element i seit dem letzten Verteilen empfing. Der Algorithmus läuft so lange weiter, bis die gewünschte Genauigkeit erreicht ist. Es ist sinnvoll, das Element mit der größten Differenz zu nehmen, statt die Elemente in zufälliger Reihenfolge auszuwählen. Man wählt also das Element, das noch am meisten Energie abzustrahlen hat. Da die Strahlung pro Flächeneinheit gemessen wird, wählt man ein Flächenelement, bei dem $\Delta B_i F_i$ maximal ist. Am Anfang gilt für alle Flächenelemente $B_i = \Delta B_i = E_i$. Dieser Wert ist nur bei Lichtquellen ungleich Null.

Im folgenden bezeichnet ΔB_i also die noch nicht verteilte Strahlung, d.h. die Differenz zwischen der *Radiosity* im letzten und im gegenwärtigen Iterationsschritt. Es wird ausgenutzt, daß gilt $F_{ji} = (F_{ij} \cdot F_i)/F_j$.

```
for i:= 1 to n do                          {initialisiere}
   if patch i ist Lichtquelle              {für jede Fläche}
     then Bi := ΔBi := Emissionswert       {Strahlung und Differenz}
     else Bi := ΔBi := 0
end;

repeat
   for {jede Fläche i, beginnend bei der mit der größten Ausstrahlung} do begin
     Plaziere Hemicube auf Fläche i
     Berechne Fij für alle 1 ≤ j ≤ n
     for j := 1 to n do begin              {für jede Fläche j tue}
        ΔR := ρj * ΔBi * Fij * Fi/Fj       {Strahlung von Fläche i}
        ΔBj := ΔBj + ΔR                    {Differenz erhöhen}
        Bj := Bj + ΔR                      {Strahlung erhöhen}
     end;
     ΔBi := 0;                             {Überschuß ist verteilt}
```

```
   end
until fertig                                    {bis zur Konvergenz}
```

Bei jeder Ausführung der äußeren FOR-Schleife verteilt ein weiteres Flächenelement seine unverbrauchte Strahlung auf die Szene. Daher werden nach der ersten Ausführung nur die Flächen beleuchtet, die selbst Lichtquellen sind sowie solche, die beim Verteilen der Strahlung des ersten Elements direkt beleuchtet werden. Rastert man am Ende jeder Ausführung des Codes ein neues Bild, so wird das erste Bild relativ dunkel und die nachfolgenden Bilder immer heller.

Abbildung 20.4 faßt den gesamten Ablauf zusammen.

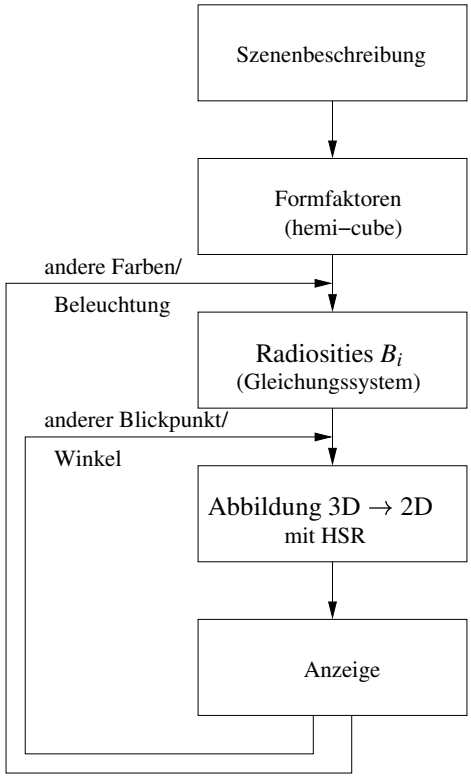
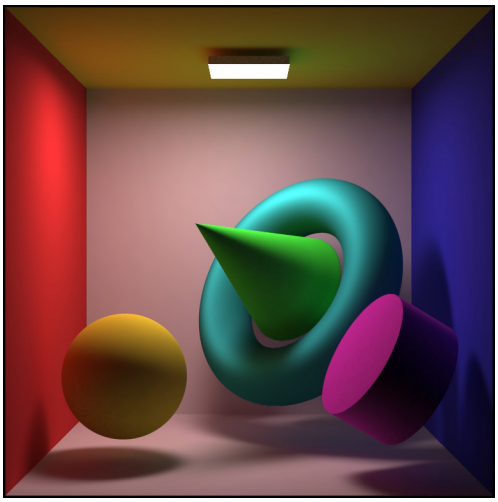


Abbildung 20.4: Ablaufschema beim Radiosityverfahren

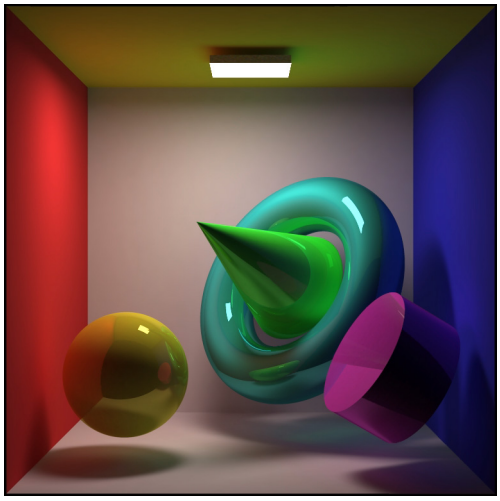
20.7 Screenshots

Auf der Seite `http://www.graphics.cornell.edu/online/research` gibt es einige Beispiele für Radiosity-Bilder.

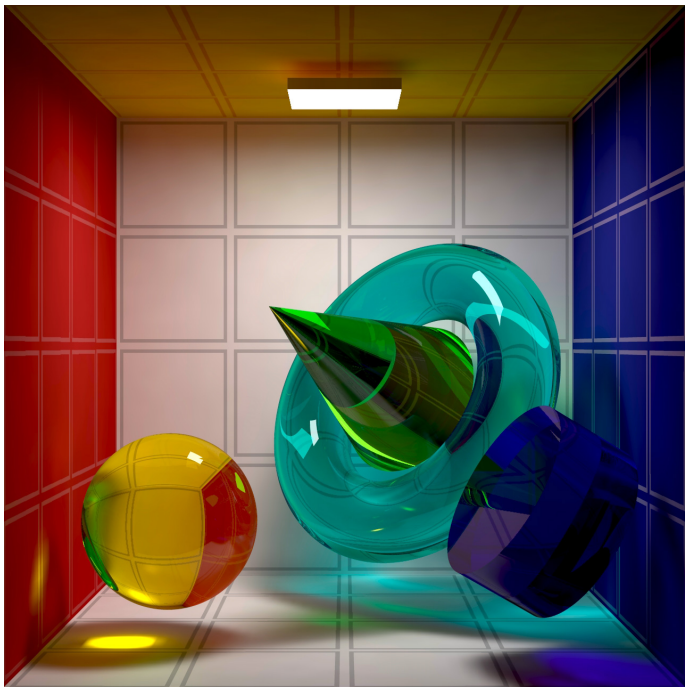
Die nächsten drei Bilder sind Screenshots vom Advanced Rendering Toolkit der Technischen Universität Wien (`http://www.cg.tuwien.ac.at/research/rendering/ART`). Hier wurde das klassische Radiosity-Verfahren um Ray-Tracing-Komponenten erweitert.



Diffuse Reflexion



Diffuse und spekulare Reflektion



Diffuse, spekulare und transparente Reflexion

Kapitel 21

Ray Tracing

21.1 Grundlagen

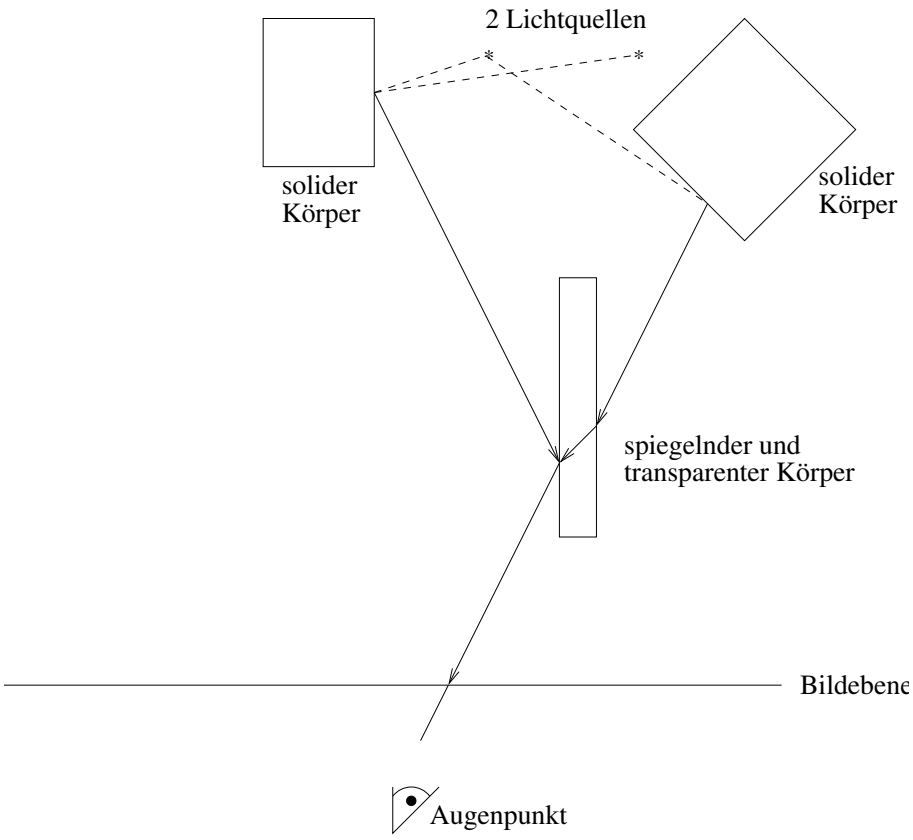


Abbildung 21.1: Prinzip der Strahlverfolgung

Verfahren mit *Ray Tracing* (Strahlverfolgung) eignen sich ausgezeichnet zur Modellierung von spiegelnden Reflexionen und von Transparenz mit Brechung (ohne Streuung). Globale Lichter werden mit einem ambienten Beleuchtungsterm ohne Richtung behandelt. Ausgehend vom Augenpunkt wird

durch jedes Pixel ein Strahl gelegt und der Schnittpunkt dieses Strahls mit dem ersten getroffenen Objekt bestimmt. Trifft der Strahl auf kein Objekt, so erhält das Pixel die Hintergrundfarbe. Ist das Objekt spiegelnd, so wird der Reflexionsstrahl berechnet und rekursiv weiterbehandelt. Ist das Objekt transparent, wird zusätzlich der gebrochene Strahl weiterbehandelt. Zur Berechnung von Schatten wird von jedem Schnittpunkt zwischen Strahl und Objekt zu jeder Lichtquelle ein zusätzlicher Strahl ausgesandt. Trifft dieser Strahl auf ein blockierendes Objekt, dann liegt der Schnittpunkt im Schatten dieser Lichtquelle, und das von ihr ausgestrahlte Licht geht in die Intensitätsberechnung des Punktes nicht ein.

21.2 Ermittlung sichtbarer Flächen durch Ray Tracing

Verfahren mit *Ray Tracing* ermitteln die Sichtbarkeit von Flächen, indem sie imaginäre Lichtstrahlen des Betrachters zu den Objekten der Szene verfolgen. Man wählt ein Projektionszentrum (das Auge des Betrachters) und ein Window in einer beliebigen Bildebene. Das Window wird durch ein regelmäßiges Gitter aufgeteilt, dessen Elemente den Pixeln in der gewünschten Auflösung entsprechen. Dann schickt man für jedes Pixel im Window einen *Augstrahl* vom Projektionszentrum durch den Mittelpunkt des Pixels auf die Szene. Das Pixel erhält die Farbe des ersten getroffenen Objekts. Das folgende Programm enthält den Pseudocode für diesen einfachen Ray Tracer.

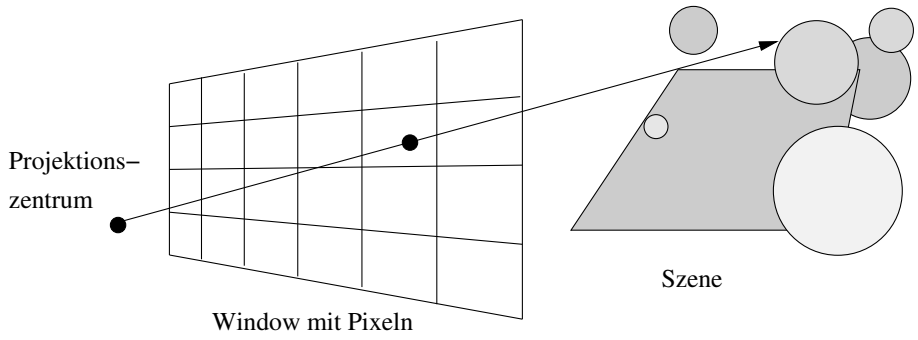


Abbildung 21.2: Strahl durch Bildebene

```
Wähle Projektionszentrum und Window in der Bildebene;
for(jede Rasterzeile des Bildes){
   for(jedes Pixel der Rasterzeile){
     Berechne den Strahl vom Projektionszentrum durch das Pixel;
     for(jedes Objekt der Szene){
        if(Objekt wird geschnitten und liegt bisher am nächsten)
           Speichere Schnittpunkt und Name des Objekts;
     }
     Setze das Pixel auf die Farbe des nächstliegenden Objektschnittpunkts;
   }
}
```

21.3 Berechnung von Schnittpunkten

Hauptaufgabe eines jeden Ray Tracers ist es, den Schnittpunkt eines Strahls mit einem Objekt zu bestimmen. Man benutzt dazu die parametrisierte Darstellung eines Vektors. Jeder Punkt (x,y,z) auf dem Strahl von (x_0,y_0,z_0) nach (x_1,y_1,z_1) wird durch einen bestimmten Wert definiert mit

$$x = x_0 + t(x_1 - x_0), \quad y = y_0 + t(y_1 - y_0), \quad z = z_0 + t(z_1 - z_0) \; .$$

Zur Abkürzung definiert man $\Delta x, \Delta y$ und Δz als

$$\Delta x = x_1 - x_0, \quad \Delta y = y_1 - y_0, \quad \Delta z = z_1 - z_0 \; .$$

Damit kann man schreiben

$$x = x_0 + t\Delta x, \quad y = y_0 + t\Delta y, \quad z = z_0 + t\Delta z \; .$$

Ist (x_0,y_0,z_0) das Projektionszentrum und (x_1,y_1,z_1) der Mittelpunkt eines Pixels im Window, so durchläuft t zwischen diesen Punkten die Werte von Null bis Eins. Negative Werte für t liefern Punkte hinter dem Projektionszentrum, Werte größer als Eins stellen Punkte auf der Seite des Windows dar, die vom Projektionszentrum abgewandt ist. Man braucht für jeden Objekttyp eine Darstellung, mit der man den Wert von t am Schnittpunkt des Strahls mit dem Objekt bestimmen kann. Die Kugel bietet sich dafür als eines der einfachsten Objekte an. Aus diesem Grund tauchen Kugeln auch so oft in Ray-Tracing-Bildern auf. Eine Kugel um den Mittelpunkt (a,b,c) und Radius r kann durch die Gleichung

$$(x-a)^2 + (y-b)^2 + (z-c)^2 = r^2$$

dargestellt werden. Zur Berechnung des Schnittpunkts multipliziert man die Gleichung aus und setzt x,y und z aus der Gleichung von oben ein.

Man erhält eine quadratische Gleichung in t, deren Koeffizienten nur die Konstanten aus den Gleichungen der Kugel und des Strahls enthalten. Man kann sie also mit der Lösungsformel für quadratische Gleichungen lösen. Wenn es keine reellen Lösungen gibt, schneidet der Strahl die Kugel nicht. Gibt es genau eine Lösung, dann berührt der Strahl die Kugel. Andernfalls geben die beiden Lösungen die Schnittpunkte mit der Kugel an. Der Schnittpunkt mit dem kleinsten positiven t-Wert liegt am nächsten.

Man muß zur Schattierung der Fläche die Flächennormale im Schnittpunkt berechnen. Im Fall der Kugel ist das besonders einfach, weil die (nicht normalisierte) Normale einfach der Vektor vom Kugelmittelpunkt zum Schnittpunkt ist: Die Kugel mit Mittelpunkt (a,b,c) hat im Schnittpunkt (x,y,z) die Flächennormale $((x-a)/r, (y-b)/r, (z-c)/r)$.

Es ist etwas schwieriger, den Schnittpunkt des Strahls mit einem Polygon zu berechnen. Um festzustellen, ob der Strahl ein Polygon schneidet, testet man zuerst, ob der Strahl die Ebene des Polygons schneidet und anschließend, ob der Schnittpunkt innerhalb des Polygons liegt.

21.4 Effizienzsteigerung zur Ermittlung sichtbarer Flächen

Die vorgestellte einfache, aber rechenintensive Version des Ray-Tracing-Algorithmus schneidet jeden Augstrahl mit jedem Objekt der Szene. Für ein Bild der Größe 1024×1024 mit 100 Objekten wären

daher 100 Mio. Schnittpunktberechnungen erforderlich. Ein System verbraucht bei typischen Szenen 75-95 Prozent der Rechenzeit für die Schnittpunktroutine. Daher konzentrieren sich die Ansätze zur Effizienzsteigerung auf die Beschleunigung der Schnittpunktberechnungen oder deren völlige Vermeidung.

Optimierung der Schnittpunktberechnungen

Die Formel für den Schnittpunkt mit einer Kugel läßt sich verbessern. Wenn man die Strahlen so transformiert, daß sie entlang der z-Achse verlaufen, kann man die gleiche Transformation auf die getesteten Objekte anwenden. Dann liegen alle Schnittpunkte bei $x = y = 0$. Dieser Schritt vereinfacht die Berechnung der Schnittpunkte. Das nächstliegende Objekt erhält man durch Sortieren der z-Werte. Mit der inversen Transformation kann man den Schnittpunkt dann für die Schattierungsberechnungen zurücktransformieren.

Begrenzungsvolumina eignen sich besonders gut dazu, die Zeit für die Berechnung der Schnittpunkte zu reduzieren. Man umgibt ein Objekt, für das die Schnittpunktberechnungen sehr aufwendig sind, mit einem Begrenzungsvolumen, dessen Schnittpunkte einfacher zu berechnen sind, z.B. Kugel, Ellipsoid oder Quader. Das Objekt wird nur dann getestet, wenn der Strahl das Begrenzungsvolumen schneidet.

Hierarchien

Begrenzungsvolumina legen zwar selbst noch keine Reihenfolge oder Häufigkeit der Schnittpunkttests fest. Sie können aber in verschachtelten Hierarchien organisiert werden. Dabei bilden die Objekte die Blätter, die inneren Knoten begrenzen ihre Söhne. Hat ein Strahl keinen Schnittpunkt mit einem Vaterknoten, dann gibt es garantiert auch keinen Schnittpunkt mit einem seiner Söhne. Beginnt der Schnittpunkttest also an der Wurzel, dann entfallen ggf. trivialerweise viele Zweige der Hierarchie (und damit viele Objekte).

Bereichsunterteilung

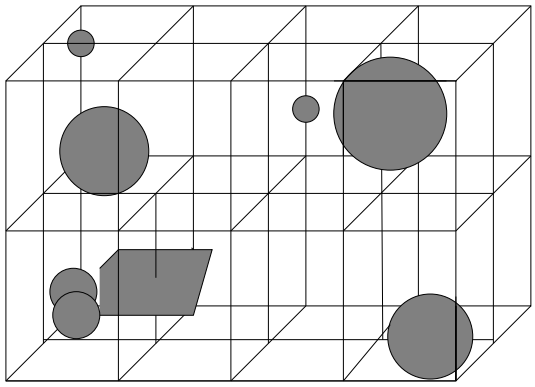


Abbildung 21.3: Unterteilung der Szene durch ein regelmäßiges Gitter

Die Hierarchie von Begrenzungsvolumina organisiert die Objekte von unten nach oben. Die Bereichsunterteilung teilt dagegen von oben nach unten auf. Zuerst berechnet man die *bounding box* der ganzen Szene. Bei einer Variante unterteilt man die *bounding box* dann in ein regelmäßiges Gitter gleich großer Bereiche. Jedem Bereich wird eine Liste mit den Objekten zugewiesen, die entweder ganz oder teilweise darin enthalten sind. Zur Erzeugung der Listen weist man jedes Objekt einem oder mehreren Bereichen zu, die dieses Objekt enthalten. Ein Strahl muß jetzt nur noch mit den Objekten in den

durchlaufenen Bereichen geschnitten werden. Außerdem kann man die Bereiche in der Reihenfolge untersuchen, in der sie vom Strahl durchlaufen werden. Sobald es in einem Bereich einen Schnittpunkt gibt, muß man keine weiteren Bereiche mehr testen. Man muß jedoch alle anderen Objekte des Bereichs untersuchen, um das Objekt mit dem nächstliegenden Schnittpunkt zu ermitteln.

21.5 Rekursives Ray Tracing

In diesem Abschnitt wird der Basisalgorithmus zum Ray Tracing auf die Behandlung von Schatten, Reflexion und Brechung erweitert. Der einfache Algorithmus ermittelte die Farbe eines Pixels am nächsten Schnittpunkt eines Augstrahls und eines Objekts mit einem beliebigen der früher beschriebenen Beleuchtungsmodelle. Zur Schattenberechnung wird ein zusätzlichen Strahl vom Schnittpunkt zu allen Lichtquellen geschickt. Schneidet einer dieser *Schattenstrahlen* auf seinem Weg ein Objekt, dann liegt das Objekt am betrachteten Punkt im Schatten, und der Schattierungsalgorithmus ignoriert den Beitrag der Lichtquelle dieses Schattenstrahls.

Der rekursive Ray-Tracing-Algorithmus startet zusätzlich zu den Schattenstrahlen weitere *Spiegelungs-* und *Brechungsstrahlen* im Schnittpunkt (siehe Abbildung 21.4). Diese Spiegelungs-, Reflexions- und Brechungsstrahlen heißen *Sekundärstrahlen*, um sie von den *Primärstrahlen* zu unterscheiden, die vom Auge ausgehen. Gibt es bei dem Objekt spiegelnde Reflexion, dann wird ein Spiegelungsstrahl um die Flächennormale in Richtung r gespiegelt. Ist das Objekt transparent, und es gibt keine totale innere Reflexion, startet man einen Brechungsstrahl entlang t in das Objekt. Der Winkel wird nach dem Gesetz von Snellius bestimmt.

Jeder dieser Reflexions- und Brechungsstrahlen kann wiederum neue Spiegelungs-, Reflexions- und Brechungsstrahlen starten.

Typische Bestandteile eines *Ray-Tracing*-Programms sind die Prozeduren `shade` und `intersect`, die sich gegenseitig aufrufen. `shade` berechnet die Farbe eines Punktes auf der Oberfläche. Hierzu werden die Beiträge der reflektierten und gebrochenen Strahlen benötigt. Diese Beiträge erfordern die Bestimmung der nächstgelegenen Schnittpunkte, welche die Prozedur `intersect` liefert. Der Abbruch der Strahlverfolgung erfolgt bei Erreichen einer vorgegebenen Rekursionstiefe oder wenn ein Strahl kein Objekt trifft.

Seien v, N die normierten Vektoren für Sehstrahl und Normale. Sei $n = n_2/n_1$ das Verhältnis der beiden Brechungsindizes. Dann sind die Vektoren r und t, die im Schnittpunkt P zur Weiterverfolgung des reflektierten und gebrochenen Strahls benötigt werden, definiert durch

$$\begin{aligned} r &= v + 2\cos(\phi)N \\ t &= (-\cos(\phi) - q)N + v \\ \text{mit } q &= \sqrt{n^2 - 1 + \cos^2(\phi)} \end{aligned}$$

Ist p ein imaginärer Ausdruck, liegt Totalreflexion vor, und der Transmissionsanteil wird gleich Null gesetzt.

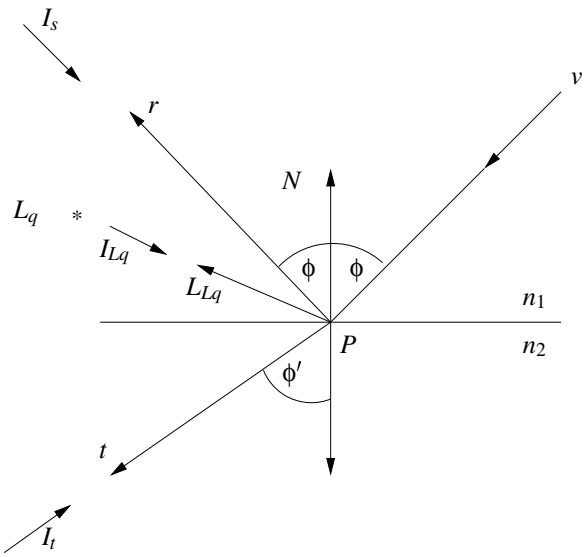


Abbildung 21.4: Einflußgrößen für Punkt P

In die Berechnung der Farbe im Punkt P geht die gewichtete Summe der in den einzelnen Rekursionsschritten berechneten Farbintensitäten ein. Unter Berücksichtigung des *Phong*-Modells ergibt sich

$$I = k_a I_H + k_d \sum_{Lq} I_{Lq}(N \cdot L_{Lq}) + k_s(I_s + \sum_{Lq} I_{Lq}(A \cdot R_{Lq})^c) + k_t I_t$$

mit

k_a	ambienter Reflexionskoeffizient
I_H	Hintergrundbeleuchtung
k_d	diffuser Reflexionskoeffizient
I_{Lq}	Intensität der Lichtquelle
N	Normale auf Ebene
L_{Lq}	Vektor zur Lichtquelle
k_s	spiegelnder Reflexionskoeffizient
I_s	globale Reflexionssintensität
A	Vektor zum Augenpunkt
R_{Lq}	Reflexionsvektor für Lichtquelle L_q
c	spekularer Exponent
k_t	Transmissionskoeffizient (materialabhängig)
I_t	Transmissionsintensität

Pseudocode für einfaches rekursives Ray Tracing

```
Wähle Projektionszentrum und Window in der view plane;
for(jede Rasterzeile des Bildes){
   for(jedes Pixel der Rasterzeile){
     Ermittle Strahl vom Projektionszentrum durch das Pixel;
     pixel = RT_intersect(ray, 1);
   }
}

/* Schneide den Strahl mit Objekten, und berechne die Schattierung am nächsten
/* Schnittpunkt. Der Parameter depth ist die aktuelle Tiefe im Strahlenbaum. */

RT_color RT_intersect(RT_ray ray, int depth)
{
   Ermittle den nächstliegenden Schnittpunkt mit einem Objekt;
   if(Objekt getroffen){
     Berechne die Normale im Schnittpunkt;
     return RT_shade(nächstliegendes getroffenes Objekt, Strahl, Schnittpunkt,
                     Normale, depth);
   }
   else
     return HINTERGRUND_FARBE;
}


/* Berechne Schattierung für einen Punkt durch Verfolgen der Schatten-, */
/* Reflexions- und Brechungsstrahlen */

RT_color RT_shade(
   RT_object object,          /* Geschnittenes Objekt */
   RT_ray ray,                /* Einfallender Strahl */
   RT_point point,            /* Schnittpunkt, der schattiert werden soll */
   RT_normal normal,          /* Normale in dem Punkt */
   int depth)                 /* Tiefe im Strahlenbaum */
{
RT_color color;               /* Farbe des Strahls */
RT_ray rRay, tRay, sRay;      /* Reflexions-, Brechungs- und Schattenstrahlen */
RT_color rColor, tColor;      /* Farbe des reflektierten und gebrochenen Strahls

   color = Term für das ambiente Licht;

   for(jede Lichtquelle){
     sRay = Strahl von der Lichtquelle zum Punkt;
     if(Skalarprodukt der Normalen mit der Richtung zum Licht ist positiv){
        Berechne, wieviel Licht von opaken und transparenten Flächen blockiert
```

```
        Skaliere damit die Terme für diffuse und spiegelnde Reflexion, bevor si
        zur Farbe addiert werden;
     }
}
if(depth < maxDepth){        /* Bei zu großer Tiefe beenden */
     if(Objekt reflektiert){
        rRay = Strahl vom Punkt in Reflexionsrichtung;
        rColor = RT_intersect(rRay, depth + 1);
        Skaliere rColor mit dem Spiegelungskoeffizienten,
        und addiere den Wert zur Farbe;
     }
     if(Objekt ist transparent){
        tRay = Strahl vom Punkt in Brechungsrichtung;
        if(es gibt keine totale interne Reflexion){
           tColor = RT_intersect(tRay, depth + 1);
           Skaliere tColor mit dem Transmissionskoeffizienten,
           und addiere den Wert zur Farbe;
        }
     }
   }
   return color;             /* Liefere die Farbe des Strahls zurück */
}
```

Ray Tracing ist besonders anfällig für Probleme, die durch die begrenzte Rechengenauigkeit entstehen. Dies zeigt sich vor allem bei der Berechnung der Schnittpunkte sekundärer Strahlen mit den Objekten. Wenn man die x-, y- und z-Koordinaten eines Objekts mit einem Augstrahl berechnet hat, dienen sie zur Definition des Startpunkts eines Sekundärstrahls. Für diesen muß dann der Parameter t bestimmt werden. Wird das eben geschnittene Objekt mit dem neuen Strahl geschnitten, hat t wegen der begrenzten Rechengenauigkeit oft einen kleinen Wert ungleich Null. Dieser falsche Schnittpunkt kann sichtbare Probleme bewirken.

21.6 Public Domain Ray Tracer Povray

Der “Persistence of Vision Ray Tracer” (POV-Ray) ist ein urheberrechtlich geschütztes Freeware-Programm zum Berechnen einer fotorealitischen Projektion aus einer 3-dimensionalen Szenenbeschreibung auf der Grundlage der Strahlverfolgung (`http://www.povray.org`). Seine Implementierung basiert auf DKBTrace 2.12 von David Buck & Aaron Collins. Zur Unterstützung des Anwenders gibt es einige include-Files mit vordefinierten Farben, Formen und Texturen.

```
#include "colors.inc"
#include "textures.inc"

camera {
       location <3, 3, -1>
       look_at  <0, 1,  2>
}
light_source { <0, 4, -3> color White}
light_source { <0, 6, -4> color White}
light_source { <6, 4, -3> color White}

plane {
       <0, 1, 0>, 0
       pigment {
               checker
                      color White
                      color Black
       }
}

sphere {
       <0, 2, 4>, 2
       texture {
               pigment {color Orange}
               normal  {bumps 0.4  scale 0.2}
               finish  {phong 1}
       }
}

box {
       <-1, 0, -0.5>,
       < 1, 2,  1.5>
       pigment {
               DMFWood4
               scale 2
       }
       rotate y*(-20)
}
```

Povray-Quelltext scene.pov

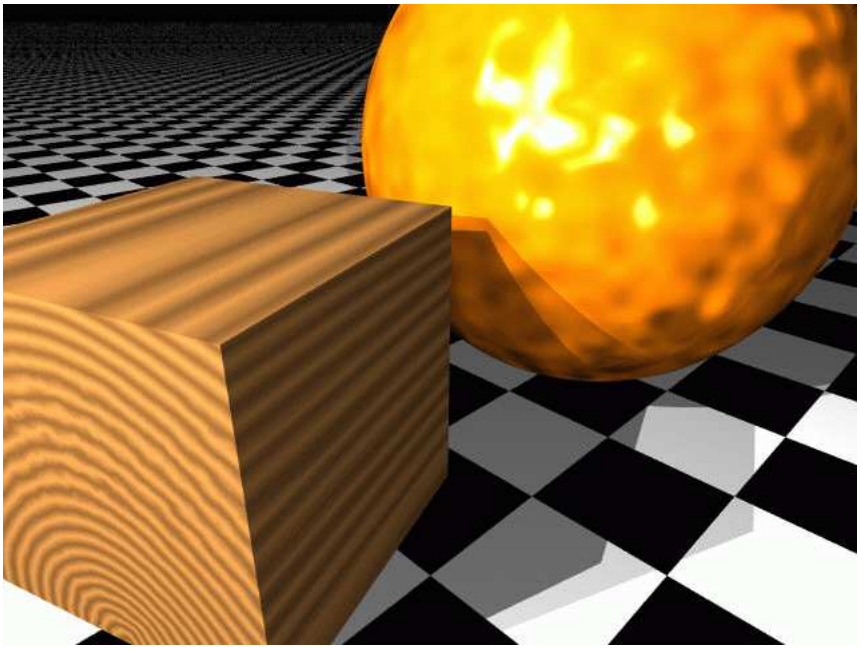


Abbildung 21.5: Povray-Bild zur Szenenbeschreibung

Der Aufruf von *povray* erfolgt zweckmäßigerweise mit einem Shell-Skript, dem der Name der zu bearbeitenden Datei als einziger Parameter übergeben wird:

```
#!/bin/sh

povray +I$1.pov +Oscene.tga +L/usr/lib/povray3/include +W320 +H200
```

Die vom Ray-Tracer erstellte Statistik lautet:

```
scene.pov Statistics, Resolution 320 x 200
-------------------------------------------------------------------------------
Pixels:           64000   Samples:           64000   Smpls/Pxl: 1.00
Rays:             64000   Saved:                 0   Max Level: 1/5
-------------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded  Percentage
-------------------------------------------------------------------------------
Box                             238037           40342     16.95
Plane                           238037           62400     26.21
Sphere                          238037           50948     21.40
-------------------------------------------------------------------------------
Calls to Noise:                      0   Calls to DNoise:          83403
-------------------------------------------------------------------------------
Shadow Ray Tests:           522111   Succeeded:                24497
-------------------------------------------------------------------------------
Smallest Alloc:                 12 bytes   Largest:           12308
Peak memory used:           103504 bytes
-------------------------------------------------------------------------------
Time For Trace:    0 hours  0 minutes  18.0 seconds (18 seconds)
    Total Time:    0 hours  0 minutes  18.0 seconds (18 seconds)
```