Computergrafik

Vorlesung gehalten im SS '97

Oliver Vornberger

Frank M. Thiesing

Frank Lohmeyer

Fachbereich Mathematik/Informatik
Universität Osnabrück

Inhaltsverzeichnis

Literatur

1. W. D. Fellner:
   "Computergrafik", BI Band 58, 1992.

2. Th. Rauber:
   "Algorithmen in der Computergraphik", Teubner 1993.

3. J. Pöpsel, U. Claussen, R.-D. Klein, J. Plate:
   "Computergrafik — Algorithmen und Implementierung", Springer, 1994.

4. J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, R. L. Phillips:
   "Grundlagen der Computergraphik", Addison Wesley, 1994.

5. C. Hornung, J. Pöpsel:
   "3-D à la carte", c't Magazin für Computertechnik, Heise Verlag, 4, 5, 7, 8, 10/1989.

6. H.-J. Brede, N. Josuttis, S. Lemberg, A. Lörke:
   "Programmieren mit OSF/Motif", Addison-Wesley, 1991.

7. D. L. McMinds:
   "Mastering OSF/Motif Widgets", Addison-Wesley, 1992.

8. K. Gottheil, H.-J. Kaufmann, Th. Kern, R. Zhao:
   "X und Motif", Springer, 1992.

9. A. Nye:
   "The X System Vol. 1, Xlib Programming Manual", O'Reilly, 1993.

10. V. Quercia, T. O'Reilly:
    "The X System Vol. 3, User's Guide, O'Reilly, 1993.

11. D. Flanagan:
    "Java in a Nutshell", O'Reilly, 1996.

12. St. Middendorf, R. Singer, St. Strobel:
    "Java", dpunkt, 1996.

13. J. Zukowski:
    "Java AWT Reference", O'Reilly, 1997.

14. J. D. Murray, W. van Ryper:
    "Encyclopedia of Graphics File Formats", O'Reilly, 1994.

Danksagung

Wir danken ...

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

... Herrn Jan Hannemann, Herrn Olaf Müller und Herrn Axel Hädicke für Anregungen bei der Gestaltung und gründliches Korrekturlesen des Skripts.

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

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

Osnabrück, im Juli 1998

(Oliver Vornberger)

(Frank M. Thiesing)

(Frank Lohmeyer)

Kapitel 1

Einführung

1.1 Motivation

"Ein Bild sagt mehr als 1000 Worte".
Das Auge erfaßt 40 Millionen Bits pro Sekunde.
Lesen kann man 1200 Worte pro Minute = $(1200 \times 8 \times 5)/60 = 800$ Bits pro Sekunde
$\Rightarrow$ Faktor 50.000.

1.2 Definition

Definition „Grafische Datenverarbeitung"

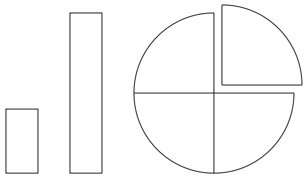
Computergrafik	Beschreibung von Bildern Eingabe der Beschreibung Manipulation der Beschreibung Ausgabe des zur Beschreibung gehörigen Bildes
Bildverarbeitung	Verbesserung von Bilddaten (z.B. Kontrasterhöhung) Vereinfachung von Bilddaten (z.B. Farbreduzierung)
Mustererkennung	Analyse von Bilddaten z.B. wo verläuft die Straße? z.B. um welchen Buchstaben handelt es sich?

1.3 Anwendungen

Typische Hauptanwendungsbereiche von Computergrafik:

- Grafische Benutzeroberflächen

- Business-Grafik
  Statt Zahlenfriedhof: Balken + Torten



- Kartografie
  Landkarten, Flächennutzungsplan
- CAD (Computer aided design)
  Entwurf von Autos, Häusern, Maschinen, VLSI-Chips
- Echtzeitsimulation und -animation
  Flug- und Fahrsimulator
- Überwachungs- und Steuerungssysteme
- Medizin
  3D-Darstellung einer Computer-Tomographie
- Visualisierung im Forschungsbereich
  Darstellung von Funktionen, Darstellung von Molekülen
- Multimedia
- Unterhaltung
  Computergrafikkunst, Computerspiele, Spielfilme, Virtual Reality (Cyberspace)

1.4 Kurze Geschichte der Computergrafik

vor 1950	grobe Grafikdarstellung mit Matrixdrucker
ab 1950	computergesteuerte Kathodenstrahlröhre
1963	"Sketchpad": Dissertation von Ivan Sutherland; Software-theoretische Grundlage der interaktiven Grafik: Datenstrukturen zur Speicherung symbolischer Hierarchien, aufgebaut aus Standardkomponenten
ab 1964	Automobilentwurf bei GM; Anfänge von CAD/CAM
Mitte der 60er	Doug Engelbart: Maus ersetzt Lichtgriffel
Anfang der 80er	Apple Macintosh und IBM PC mit Bitmap-Grafik und Maus

offizielle Standards

1977	Core Graphics System vom ACM SIGGRAPH
1985	ISO GKS: Grafisches Kernsystem
1988	GKS-3D
1988	ANSI PHIGS: Programmer's Hierarchical Interactive Graphics System
1992	ISO PHIGS PLUS: (+ Rendering)

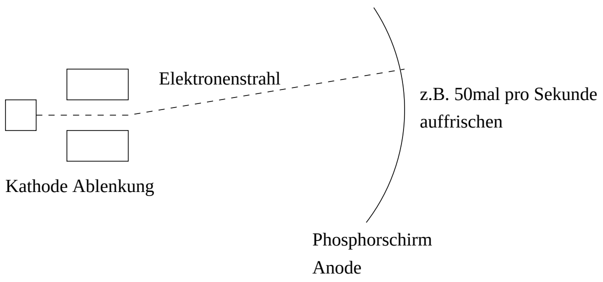
de-facto-Standards

- Postscript von Adobe
- X vom MIT (Massachusetts Institute of Technology)
- PEX: PHIGS für X

1.5 Ausgabegeräte

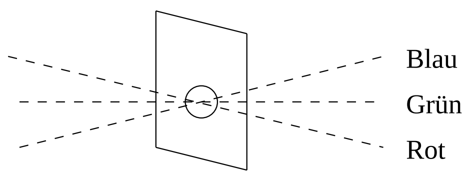
1.5.1 Sichtgeräte (soft copy)

- Refresh-Bildschirm
  Kathode emittiert Elektronen zur Anode (Phosphorschirm). Ablenkung erfolgt durch Magnetfeld. Da Phosphor nur kurz nachleuchtet, muß mit mind. 50 Hz aufgefrischt werden.



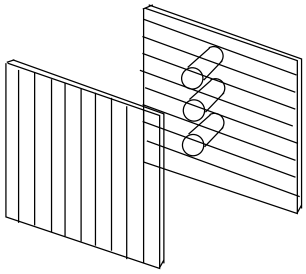
- Speicherbildschirm
  leuchtet extrem lang nach
  Vorteil: flimmerfrei
  Nachteil: kein selektives Löschen

- 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-Schirm (Liquid Crystal Display)
- TFT-Schirm (Thin Film Transistor)

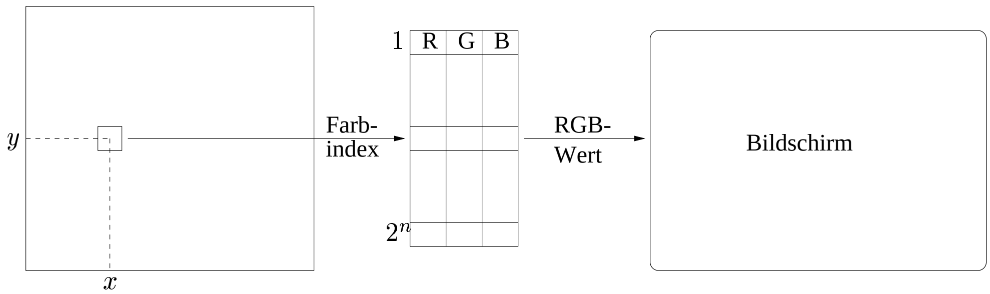
Vorteil: flach, leicht, flimmerfrei, hoher Kontrast (TFT)
Nachteil: noch relativ teuer

Random Scan versus Raster Scan

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

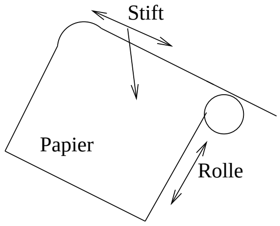
- Raster scan
  Elektronenstrahl tastet Schirm zeilenweise ab. Holt Information für jedes Pixel (**pic**ture **el**ement) aus frame buffer (Bildwiederholspeicher) = VIDEO-RAM, beschrieben vom Grafikprozessor.
  Vorteil: beliebig komplexer Bildaufbau
  Nachteil: diskrete Darstellung der Pixel: Linien müssen durch Rasterung (Scan Conversion) dargestellt werden. Aliasing.
  Bei Farbe muß pro Pixel ein RGB-Tripel bestimmt werden.

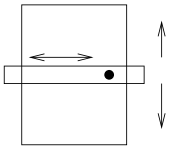
Video-RAM	Color-Table	
pro Pixel ein	pro Index	gleichzeitig darstellbar
8-Bit-Index	3 Farbwerte à 8 Bit	2^8 =256 von 2^{24} =16 Mill. Farben
1280×960 Schirm	256×3 Bytes	
$\Rightarrow 1,2$ MBytes	$\Rightarrow 768$ Bytes	
für Video RAM	für Color-Table	



1.5.2 Hardcopy

- Plotter
  bunte Stifte aus einem Magazin



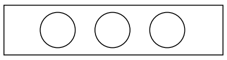


- Matrixdrucker
- Tintenstrahldrucker
- Laserdrucker
- Thermotransferdrucker, Thermosublimationsdrucker

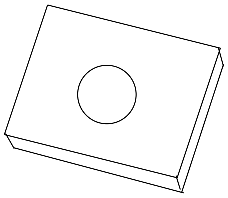
Farbdrucker benutzen Gelb, Magenta, Cyan

1.6 Eingabegeräte

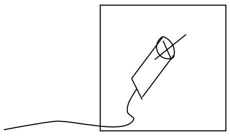
- Keyboard (Tastatur)
- Paddles (Drehknopf)
  z.B. bei CAD-Arbeitsplätzen zum Transformieren



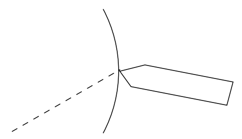
- Joy-Stick
- Track-Ball (versenkte Kugel)



- Maus
  optisch, mechanisch, mit Buttons
- Tablett mit Fadenkreuz



- Lichtgriffel, Touch Panel = Bildschirm zum Anfassen



- Frame Grabber = Übernahme des Signals einer Videokamera

- Scanner = optisches Abtasten und Digitalisieren einer Bildvorlage

1.6.1 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.

1.6.2 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.

1.6.3 Typische Eingabetechnik mit der Maus

- Objekte picken und plazieren
- Objekte picken und modifizieren (skalieren, dehnen, rotieren, ...)
- Objekte zeichnen (z.B. bei *xfig*)
  Typ auswählen,
  Maus-Button drücken,
  Maus bewegen,
  Maus-Button loslassen.

Hilfestellungen

- *snap* (Absetzen nach Bewegen nur an Gitterpunkten eines Gitters mit vorgegebener Maschenweite)
- eingeblendetes Gitter
- *rubber-band* (Andeutung der Umrisse während des Bewegens)
- horizontale oder vertikale Beschränkung beim Bewegen

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.

2.2 Eventhandling

Eine grafische Benutzeroberfläche muß in der Lage sein, auf zahlreiche verschiedene Ereignisse zu reagieren. Typische Events sind ein Mausklick, die Bewegung der Maus oder ein

Tastendruck. Die Ereignisse betreffen aber nicht nur die Benutzereingaben, sondern auch die Interaktion zwischen Applikationen.
Z.B. ein Expose-Event wird beim Start einer neuen Applikation ausgelöst. Wird die Maus in ein Fenster gefahren, gibt es einen Enter-Event.
Die Zahl verschiedener Events wird durch Kombinationen multipliziert. Applikationen sollen häufig auf einen Doppelklick reagieren oder erst auf das Loslassen der Maustaste; die Bewegung der Maus bei gedrückter Taste wird oft zur Auswahl in Menüs verwendet.
Die unterschiedlichsten Ereignisse können zu beliebiger Zeit in beliebiger Reihenfolge auftreten. Ein gutes Anwendungsprogramm ist so geschrieben, daß es vom Benutzer gesteuert wird und nicht umgekehrt.
Das typische Schema für die Verarbeitung von Interaktionen in einem Anwendungsprogramm ist die Event-Loop. Nach dem Schema eines endlichen Automaten, der einen zentralen Wartezustand kennt, werden die Übergänge zu anderen Zuständen durch Benutzereingaben ausgelöst.

```
Darstellung des Ausgangsbilds
do {
    Auswahl von Befehlen oder Objekten ermöglichen
    /* Das Programm wartet unbestimmte Zeit, bis Benutzer etwas eingibt */
    warte auf Benutzereingabe
    switch (Eingabe) {
        bearbeite Eingabe und führe ggf. Kommando aus
        bei Bedarf Bildschirm aktualisieren
    }
}
while(!Ende) /* Benutzer hat nicht "Ende" gewählt */
```

2.3 Eigene Applikationen

Während der Window-Manager mit einer Reihe von Events umgehen kann, die zu seiner Funktionalität gehören, muß sich der Entwickler einer grafischen Applikation mit den Ereignissen, die diese betreffen, selbst auseinandersetzen.
Soll eine Anwendung z.B. auf einen Knopfdruck reagieren, so muß der Event eines Mausklicks im Bereich der grafischen Darstellung des Buttons vom Entwickler abgehandelt werden. Dazu ist es notwendig, ein Applikationsprogramm zu schreiben, das auf dieses Ereignis hin die gewünschte Funktion ausführt, den sogenannten *Callback*.
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.
Unter UNIX ist die z.Zt. am meisten verbreitete Kombination X-Windows mit Xlib, Xt und dem Motif-Widget-Set. Die Programmierung erfolgt dabei meist mit Hilfe von C. Eine weitere Variante sind Bibliotheken, die auf diese Umgebung aufsetzen, wie z.B. Tk, das dann in Skript-Sprachen wie Tcl oder Perl5 verwendet werden kann.

Eine spezielle Umgebung kann unter Openstep verwendet werden, wo das dortige Applikation-Kit zusammen mit der Sprache Objective-C und diversen Openstep-Tools direkt mit dem Betriebssystem zusammenarbeitet. Noch enger ist die Verbindung zwischen Betriebssystem und Grafik-API unter MS-Windows, wo kaum noch zwischen beiden unterschieden werden kann. Die Programmierung erfolgt dort meist mit Hilfe von grafischen Interface-Buildern, wo nur noch wenige Teile einer Applikation in einer klassischen Programmiersprache (C, C++, Basic) implementiert werden müssen.
Im folgenden werden anhand des X-Window-Systems typische Vokabeln der API/GUI-Programmierung dargestellt.

2.4 Das X-Window-System

2.4.1 Begriffe

Das X-Window-System wurde 1984 am MIT entworfen, seit 1987 ist es in der Version X11 verfügbar, mittlerweile als Release 6. Es werden Routinen bereitgestellt zur maschinenunabhängigen und netzwerktransparenten Formulierung von interaktiven grafischen Bedienoberflächen. Der Bildschirm wird aufgeteilt in mehrere, ggf. sich überlappende, *Fenster* (windows) mit jeweils eigenen Ein-/Ausgabefunktionalitäten. Die Fenster bilden eine baumartige Hierarchie (Kind-Fenster, Eltern-Fenster, Geschwister). Das oberste Fenster (root window) ist der Bildschirm selbst. Hierunter liegen die *Top-Level-Fenster* der einzelnen Anwendungen. Ihre Plazierung übernimmt der Window-Manager. Die Plazierung der Kinder und Enkel der Top-Level-Fenster regelt der Anwendungsprogrammierer.
X-Programme laufen auf verschiedenen Plattformen unter diversen Betriebssystemen. Das Anwendungsprogramm, genannt *X-Client*, schickt Aufträge an den *X-Server*, der die Hardware bedient.
Aus Performancegründen läuft die Client-Software häufig auf leistungsstarken Rechnern im Netz, während für den Server ein kleiner PC oder ein X-Terminal verwendet werden kann.
X-Server und X-Client können auf denselben oder auf verschiedenen Rechnern laufen. Sie kommunizieren mit Hilfe des *X-Protokolls* auf asynchrone Weise (gepuffert). Es gibt vier Arten von Nachrichten: *requests, replies, events* und *errors*.
Der X-Server hat exklusive Kontrolle über die grafischen Ausgabegeräte einschließlich Tastatur. Der Server verwaltet die *Ressourcen*, das sind komplexe Datenstrukturen für Windows, Pixmaps, Fonts, Grafik-Kontexte. Der Client referiert die Ressource über ihre ID. Dadurch wird der Kommunikationsbedarf reduziert; außerdem können mehrere Clients dieselben Ressourcen verwenden.
Der X-Client schickt Kommandos oder fragt Zustände ab. Ein Client kann Kontakt zu mehreren Servern halten. Der *Window-Manager* ist ein weiterer X-Client — und damit austauschbar. Er plaziert, verziert und ikonifiziert die Top-Level-Fenster.
Ein grafisches Ausgabegerät wird *display* genannt. Dazu gehören ein Grafik-Speicher und ggf. mehrere Bildschirme, genannt *screens*. Ein display-Name setzt sich zusammen aus

```
(<Hostname> | <Internet-Nr> ) : <Server-Nr> [.<Screen-Nr>]
```

und kann über die Environmentvariable `DISPLAY` gesetzt werden:

```
sh% export DISPLAY=magni:0
sh% export DISPLAY=131.173.12.30:0
```

Zuvor muß der angesprochene Server seine Einwilligung gegeben haben. Um z.B. dem Rechner *zeus* die Ausgabe auf dem Rechner *magni* zu erlauben, setzt man auf *magni* das Kommando ab

```
sh% xhost +zeus
```

X-Clients arbeiten ereignisorientiert, d.h., sie reagieren auf *events*, die der Server an sie schickt. Durch Event-Masken legt der Client fest, welches Fenster über welche Ereignisse informiert werden soll.

2.4.2 Architektur

Das X-Window-System ist das Fenstersystem der UNIX-Welt. X-Anwendungen verwenden Routinen aus der Xlib-Bibliothek, aus den X Toolkit Intrinsics und aus einem Widget-Set. `Xlib` enthält verschiedene Low-Level-Funktionen auf der Ebene des X-Protokolls. Z.B.

```
XDrawLine(display, window, gc, x1, y1, x2, y2);
```

zeichnet auf dem Ausgabegerät `display` im Fenster `window` unter Verwendung des Grafik-Kontextes `gc` eine Linie von Punkt `x1,y1` zu Punkt `x2,y2`.
Die X Toolkit Intrinsics, kurz *Xt Intrinsics*, sind ein Aufsatz auf die Xlib. Hiermit lassen sich die grafischen Grundobjekte, die *Widgets*, kreieren und manipulieren. *Widget* ist ein aus den Begriffen *Window* und *Gadget* zusammengesetztes Kunstwort und bedeutet Dialogobjekt. Z.B.

```
XtVaGetValues(widget, XmNwidth, &breite, XmNheight, &hoehe, NULL);
```

besorgt von einem Widget die Breite und Höhe und überträgt diese Werte in die Variablen `breite` bzw. `hoehe`.
Das X-Window-System stellt nur die Technik zur Verwaltung von grafischen Oberflächen zur Verfügung, die Toolkits hingegen bringen Verhalten ins Spiel, ein spezielles *Look and Feel* der Benutzerschnittstelle.
Ein *Widget-Set* ist eine Ansammlung von aufeinander abgestimmten Widget-Klassen, die sich durch Design und Funktionalität charakterisieren. Bekannte Beispiele für Widget-Sets sind OSF/Motif, Open Look und Athena Widgets.
In Motif gibt es z.B. einen dreidimensional aussehenden Druckknopf in der Klasse `XmPushButton` oder ein Anzeigefeld mit verschiebbarem Inhalt in der Klasse `XmScrolledWindow`. Widgets werden in Motif wie folgt klassifiziert:

Display-Widgets:	zum Anzeigen von Information z.B. `XmPushButton, XmToggleButton, XmText`
Container-Widgets:	zum Zusammenfassen der Kind-Widgets z.B. `XmForm, XmRowColumn`
Shell-Widgets:	spezielle Container-Widgets, eingesetzt als Top-Level-Fenster, stehen unter Verwaltung des Window-Managers. z.B. `XmDialogShell`

Die Widget-Klassen bilden einen Baum, bei dem die Kinder die Eigenschaften ihrer Eltern erben und erweitern. Aussehen und Verhalten der Widgets werden durch deren Ressourcen gesteuert. Z.B. besitzt ein Label-Widget die Ressourcen Widget-Größe, auszugebender Text, Font und Farbe des auszugebenden Textes. Manchen Widgets kann man die Adresse von Funktionen mitgeben. Eine solche Funktion, genannt *callback*, wird dann bei Eintreten eines bestimmten Ereignisses, z.B. Drücken eines Mausknopfes innerhalb des Widget-Fensters, aufgerufen.

2.4.3 Ressourcen

Die wichtigsten Ressourcen:

Eine *Pixmap* ist eine rechteckige Fläche von Farbwerten im Off-Screen-Bereich, geeignet für Hintergrund und Rand eines Fensters und für gefüllte Flächen. Die Farbtiefe, *depth*, legt fest, wieviel Bits pro Farbwert zur Verfügung stehen.

Eine *Bitmap* ist eine Pixmap mit Farbtiefe 1, geeignet für Schwarzweißbilder, Cursor-Formen und Masken für Maloperationen.
Ein *Cursor* ist eine sichtbare Repräsentation der Mausposition. Seine Form wird durch eine Bitmap definiert.

Ein *Font* ist ein nicht skalierbarer Zeichensatz. Nicht jeder X-Server kennt alle Fonts. Mit dem Kommando `xfontsel` können die verfügbaren Fonts und ihr Aussehen ermittelt werden.

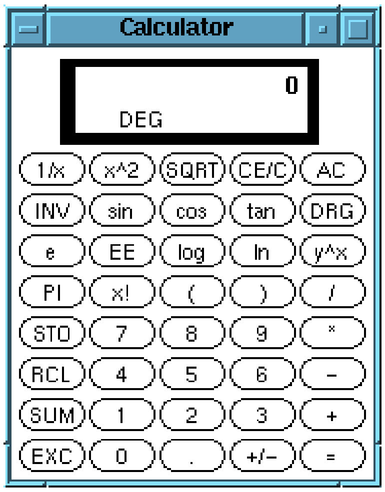
Ein *Grafik-Kontext* legt für ein Fenster gewisse Parameter für Zeichenoperationen fest, z.B. Vordergrundfarbe, Strichstärke, Strichart, Verknüpfungsart.

Ressourcen können gesetzt werden im Programmtext, durch Flags beim Programmstart oder durch Erwähnung in einer Ressource-Datei.

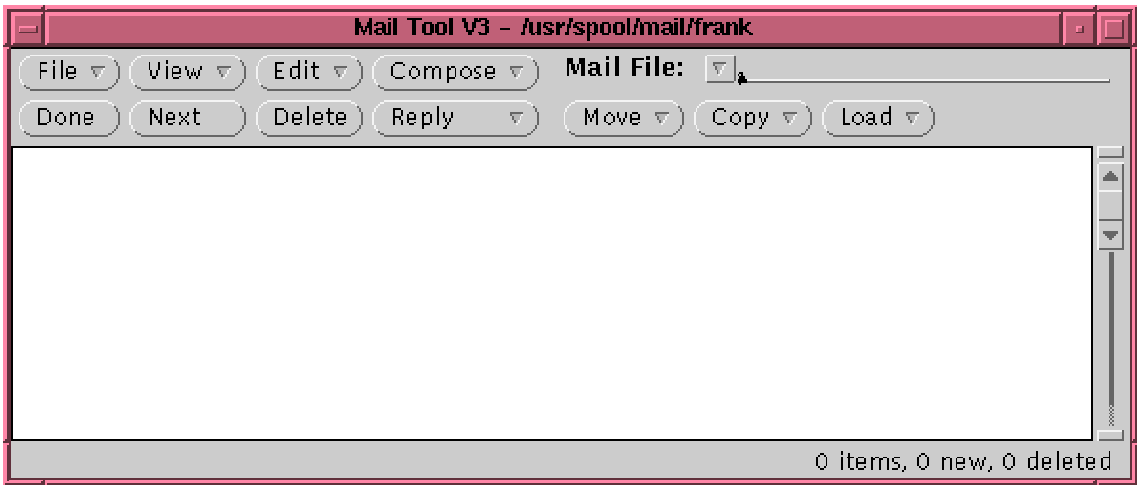
2.5 Beispiele für Widget-Sets

Im folgenden werden mit verschiedenen Widget-Sets erstellte Applikationen gezeigt.

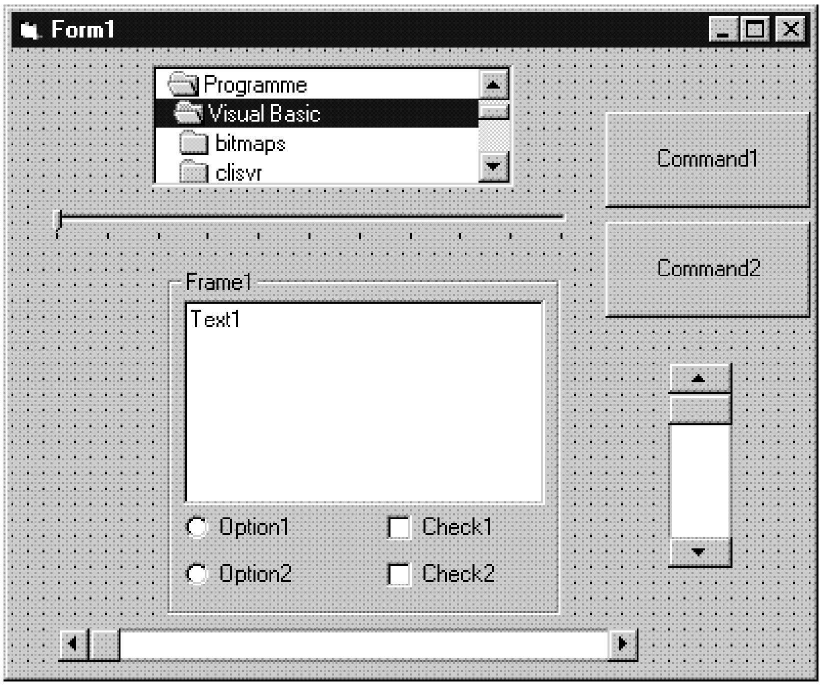
OSF/Motif: `listbox`



Athena: `xcalc`



Open Look: `mailtool`



Windows-API mit VisualBasic

2.6 AWT

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 Benutzerschnittstelle entwickeln, die auf allen verwendeten Systemen eine identische Funktionalität aufweist, so benötigt man eine portable Programmierumgebung. Eine solche Umgebung ist das *Abstract Window Toolkit* (AWT) der Programmiersprache Java.
Das AWT umfaßt einige der Java-Standard-Klassen, die für die portable Programmierung von GUI-Applikationen entwickelt wurden. Es heißt *abstract*, weil es kein eigenes API enthält, sondern sich auf existierende Window-Systeme abstützt. So wird auf UNIX-Systemen das Motif-Widget-Set verwendet, unter MS-Windows dagegen dessen Standard-API. Diese Unabhängigkeit vom Betriebs- bzw. Window-System ist allgemein Teil der Java-Philosophie.

2.6.1 Java

Die Programmiersprache Java wird seit 1990 von der Arbeitsgruppe Gosling bei der Firma Sun entwickelt. Als architekturunabhängige Sprache wird Java seit Aufkommen des World Wide Web als ideal geeignet für die Internetprogrammierung angesehen.
Java wird von Sun beschrieben als

- einfach,
- objektorientiert,
- verteilt,
- interpretiert,
- robust,
- sicher,
- architekturunabhängig,
- portabel,
- hochperformant,
- multithreaded,
- dynamisch.

Java hat für die Basis-Befehle die Syntax von C übernommen und diese dann um Konstrukte zur objektorientierten Programmierung erweitert. Im Unterschied zu C++, wo ANSI-C als echte Teilmenge in der Sprache enthalten ist, wurden in Java Sprachelemente entfernt, hinzugefügt und geändert.
Ein weiterer, wesentlicher Unterschied ist, daß Java eine vollständig objektorientierte Sprache ist und keine Mechanismen besitzt, um z.B. Funktionen unabhängig von Klassen und Objekten zu implementieren.
Außerdem benutzt Java keine Zeiger, also insbesondere auch keine Funktionszeiger. Da Java keine Strukturen besitzt und Felder und Zeichenketten Objekte sind, besteht kein Bedarf für

Zeiger. Java übernimmt automatisch das Referenzieren und Dereferenzieren von Objekten. Java implementiert zusätzlich eine Garbage Collection zur Speicherverwaltung.

Ein Beispiel
Das folgende Beispiel zeigt ein einfaches Java-Programm: Hello World.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Dieses Programm besteht, wie jedes Java-Programm, aus einer öffentlichen Definition. Die Klasse enthält eine Methode mit dem Namen `main()`, in der der Interpreter mit der Programmausführung beginnt.
Dieses Programm muß in einer Datei gespeichert werden, die den Namen der öffentlichen Klasse, gefolgt von der Erweiterung `.java`, besitzt. Die Übersetzung geschieht mit:

```
sh% javac HelloWorld.java
```

Dieses Kommando erzeugt die Datei `HelloWorld.class`. Um das Programm ablaufen zu lassen, wird der Java-Interpreter benutzt:

```
sh% java HelloWorld
```

Ein Programm in Java besteht im allgemeinen aus einer oder mehreren Klassendefinitionen, von denen jede in ihre eigene `.class`-Datei in *Java Virtual Machine*-Code übersetzt wird. Eine dieser Klassen muß eine `main()`-Methode definieren, in der das Programm seine Ausführung beginnt.
Der Java-Compiler erzeugt Bytecode statt direkten Maschinencode. Dieser Bytecode stellt ein architekturunabhängiges Objektdateiformat zur Verfügung, d.h., der Code ist dazu entworfen worden, Programme effizient auf verschiedene Plattformen zu transportieren. Ein Java-Programm kann auf jedem System laufen, das einen Java-Interpreter und ein Laufzeitsystem implementiert. Zusammengenommen implementieren beide die Java Virtual Machine.

2.6.2 AWT-Übersicht

In diesem Abschnitt wird eine kurze Übersicht der wesentlichen Konzepte der AWT-Klassen gegeben. Die für die Programmierung notwendigen Details sind bei Java generell der HTML-On-Line-Dokumentation zu entnehmen.
Das Basis-Paket für Grafik- und Oberflächenprogrammierung ist *java.awt*. 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 *java.applet*, dessen einzige Klasse *Applet* verwendet wird, wenn man Java-Programme in einem HTML-Dokument benutzen will.
Die Klassen des AWT-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.

Grafik

Die Grafik-Klassen implementieren jeweils einen bestimmten Aspekt und sind weitgehend unabhängig voneinander. Wichtige Klassen 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.

GUI-Komponenten

Die Komponenten, mit deren Hilfe der Benutzer mit einer Oberfläche interagieren kann, sind im wesentlichen von zwei Klassen abgeleitet: *MenuComponent*, von der Klassen abgeleitet werden, mit denen Pulldown- und Popup-Menüs erzeugt werden können; und Component, von der alle anderen Komponenten abgeleitet werden. Diese kann man wiederum in zwei Gruppen einteilen. Zum einen gibt es Komponenten, die eine spezielle Funktionalität bereitstellen:

- *Button*: Es wird ein mit einem Titel versehenes Feld erzeugt, das mit der Maus angeklickt werden kann, wodurch ein entsprechender Event erzeugt wird.
- *Canvas*: Ein einfaches Feld ohne weitere Funktion, das z.B. als Zeichenfläche verwendet werden kann.
- *Checkbox*: Ein Schalter mit zwei Zuständen, der bei einem Mausklick diesen wechselt und dabei einen Event auslöst. Mehrere Checkbox-Objekte können entweder unabhängig voneinander arbeiten oder zu einer *CheckboxGroup* zusammengefaßt werden. In diesem Fall kann immer nur einer in dieser Gruppe eingeschaltet sein, so daß sich eine sogenannte *Radiobox* ergibt.
- *Choice*: Eine Gruppe von Einträgen, von der immer nur einer sichtbar ist. Das Auswählen eines Eintrags erzeugt einen entsprechenden Event.
- *FileDialog*: Diese komplexe Komponente erzeugt ein File-Listing, aus dem eine Datei ausgewählt werden muß.
- *Label*: Diese einfachste Komponente erzeugt ein Feld, in dem ein String ausgegeben wird.
- *List*: Es wird eine Liste von Strings angezeigt, aus der Einträge selektiert bzw. deselektiert werden können, was zur Erzeugung von entsprechenden Events führt.

- *Scrollbar*: Ein Rollbalken, der mit der Maus bewegt werden kann, wodurch entsprechende Events erzeugt werden.
- *TextArea*: Ein Feld, in dem ein längerer Text angezeigt und evtl. editiert werden kann.
- *TextField*: Ein Text-Feld, in dem ein einzelner String angezeigt und evtl. editiert werden kann.

Außerdem gibt es Komponenten, die keine eigene Funktionalität bereitstellen, sondern die andere Komponenten aufnehmen und anordnen können. Die wichtigsten dieser Komponenten, die von der Klasse *Container* abgeleitet werden, sind:

- *Frame*: Dieser Container erzeugt ein eigenständiges Fenster, das vom Window-Manager verwaltet wird.
- *Panel*: Ein Container, der zur Strukturierung der Komponenten innerhalb eines Fensters dient.
- *ScrollPane*: Ein Panel, das mit Scrollbars versehen ist und damit in der Lage ist, einen Ausschnitt eines größeren Feldes darzustellen.
- *Applet*: Ein Panel, das spezielle Methoden hat, die von einem Web-Browser aufgerufen werden können.

Layout-Manager
Die wesentliche Aufgabe eines Containers ist die Darstellung der darin enthaltenen Komponenten. Zur Steuerung der Anordnung dieser Komponenten werden sogenannte *Layout-Manager* verwendet. Ein solcher Layout-Manager implementiert dazu eine spezielle Layout-Politik, die mehr oder weniger konfigurierbar ist. Es gibt die folgenden vordefinierten Layout-Manager:

- *FlowLayout*: Dies ist das einfachste Layout, welches bei Panel 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 *Frame* verwendet wird. Es besteht aus maximal fünf Komponenten, die an den Positionen Nord, Süd, Ost, West und Zentrum eingefügt 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.

Falls die Anforderungen einer speziellen Applikation mit diesen existierenden Layout-Managern nicht erfüllt werden können, besteht die Möglichkeit, weitere Manager zu implementieren. Dies ist im allgemeinen aber nicht der Fall, da mit dem *GridBagLayout* alle praktisch vorkommenden Anordnungen ermöglicht werden können.

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. Eine Komponente kann für sie geeignete Events empfangen, 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. Die folgende Tabelle gibt eine Übersicht der verschiedenen Event-Listener:

Listener	Methoden	Komponenten
Action	actionPerformed	Button, List, MenuItem, TextField
Adjustment	adjustmentValueChanged	Scrollbar
Component	componentHidden	Component
	componentMoved	
	componentResized	
	componentShown	
Container	componentAdded	Container
	componentRemoved	
Focus	focusGained	Component
	focusLost	
Item	itemStateChanged	Checkbox, Choice, List
Key	keyPressed	Component
	keyReleased	
	keyTyped	
Mouse	mouseClicked	Component
	mouseEntered	
	mouseExited	
	mousePressed	
	mouseReleased	
MouseMotion	mouseDragged	Component
	mouseMoved	
Text	textValueChanged	TextComponent
Window	windowActivated	Window
	windowClosed	
	windowClosing	
	windowDeactivated	
	windowDeiconified	
	windowIconified	
	windowOpened	

Kapitel 3

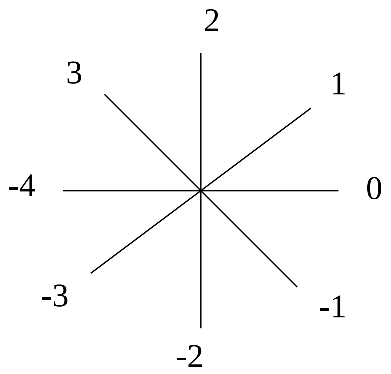
2D-Grundlagen

3.1 Koordinatensysteme

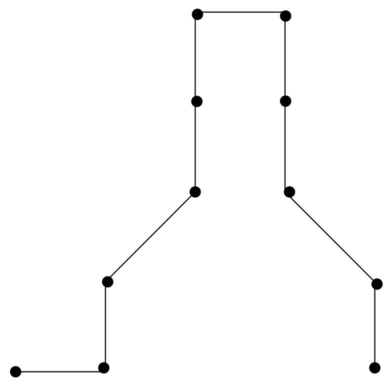
Meistens: Angabe eines Punktes durch kartesische Koordinaten x, y.
Eine Folge von Punkten kann

- absolut oder
- relativ spezifiziert werden.

Absolut bedeutet:	Jeder Punkt der Folge wird durch seine kartesischen Koordinaten beschrieben.
Relativ bedeutet:	Jeder Punkt der Folge wird unter Verwendung der Windrose relativ zum Vorgänger der Folge beschrieben, entweder mit absoluten Richtungscodes oder mit relativen Richtungscodes.



Beispiel: Gegeben sei folgende Figur:



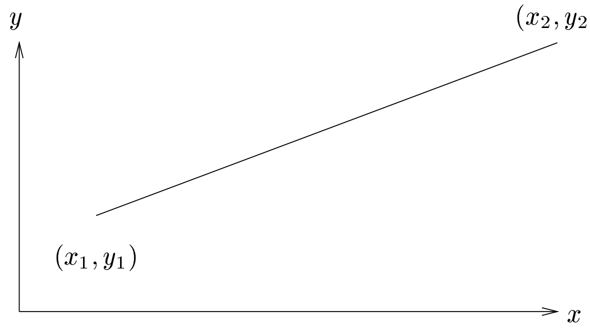
Absoluter Richtungscode ergibt für die Figur die Folge $0, 2, 1, 2, 2, 0, -2, -2, -1, -2$.
Relativer Richtungscode legt nach jedem Schritt die momentane Richtung als "0" fest. Es ergibt sich für Bild 2 die Folge $0, 2, -1, 1, 0, -2, -2, 0, 1, -1$.
Zur Platzeinsparung werden die "wahrscheinlicheren" Richtungsänderungen mit weniger Bits codiert:

Richtung	Code	Länge
0	00	2
1	10	2
-1	01	2
2	1100	4
-2	1101	4
3	111000	6
-3	111001	6
-4	111011	6

Für Bild 2 entstehen statt $(10 \times 3) = 30$ Bits nur noch 26 Bits.

3.2 Linie

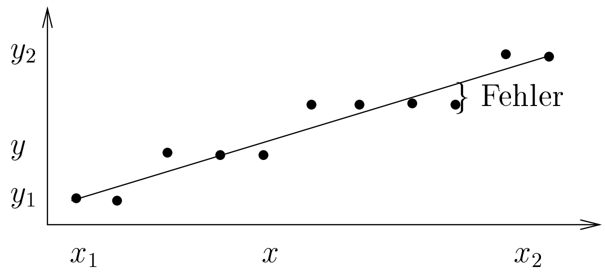
Problem: Gegeben sind Anfangs- und Endkoordinaten einer Linie. Berechne die "anzuschaltenden" Pixel.



1. Möglichkeit: Bestimme Geradengleichung $y = s \cdot x + c$
Es gilt:

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

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



Daraus ergibt sich folgender Algorithmus unter Verwendung der Klasse `java.awt.Point` mit den beiden Integer-Koordinaten `x` und `y`:

```
void plot_line(Point P, Point q)
{
        double s, c;
        Point p = new Point(P);

        s = (double) (q.y-p.y) / (double) (q.x-p.x);
        c = (double) (p.y*q.x - q.y*p.x) / (double) (q.x - p.x );

        while (p.x <= q.x) {
                p.y = (int)(s*p.x + c + 0.5);
                set_pixel(p);
                p.x++;
        }
}
```

Verfahren hat 2 Nachteile:

1. Pixel für steile Geraden sind nicht benachbart.
2. viel Gleitkomma-Arithmetik

Lösung:	Bresenham-Algorithmus
Idee:	Berechne den nächsten y-Wert aus dem vorherigen. Merke den momentanen Fehler.
Zunächst:	Gerade verläuft im 1. Oktanten (d.h. Steigung < 1)

```
void bresenham_linie_1(Point P, Point q)
{
        int dx, dy;
        double s, e;
        Point p = new Point(P);

        dx    = q.x - p.x;
        dy    = q.y - p.y;
        e     = 0.0;
        s     = (double) dy / (double) dx;

        while (p.x <= q.x) {
                set_pixel(p);
                p.x++;
                e += s;
                if (e > 0.5) { p.y++; e-- ; }
        }
}
```

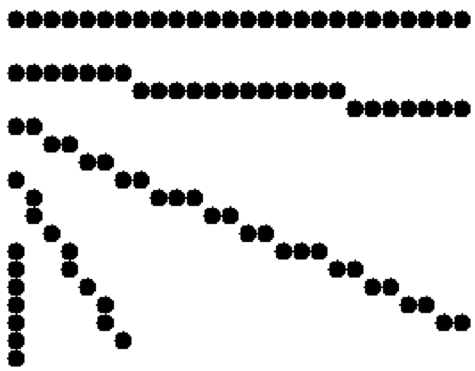
Eliminiere Gleitkommazahlen durch Multiplikation von `s` mit `2*dx`:

```
void bresenham_linie_2(Point P, Point q)
{
        int dx, dy, error, delta;
        Point p = new Point(P);

        dx    = q.x - p.x;
        dy    = q.y - p.y;
        error = 0;
        delta = 2*dy;

        while (p.x <= q.x) {
                set_pixel(p);
                p.x++;
                error += delta;
                if (error > dx) { p.y++; error -= 2*dx; }

        }
}
```

Vergleiche `error` mit 0 und verwende Abkürzung `schwelle` für `-2*dx`

```
void bresenham_linie_3(Point P, Point q)
{
        int dx, dy, error, delta, schwelle;
        Point p = new Point(P);

        dx       = q.x - p.x;
        dy       = q.y - p.y;
        error    = -dx;
        delta    = 2*dy;
        schwelle = -2*dx;

        while (p.x <= q.x) {
                set_pixel(p);
                p.x++;
                error += delta;
                if (error > 0) { p.y++; error += schwelle; }
        }
}
```

Geraden in den anderen 7 Oktanten müssen durch Spiegelung und/oder Vertauschen von x und y auf den 1. Oktanten zurückgeführt werden.

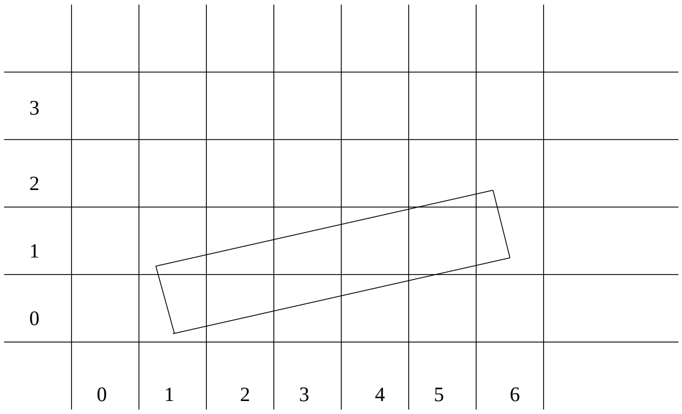


Vom Bresenham-Algorithmus erzeugte Linien

```
/***************************************************************************************/
/*                                                                                     */
/*                      Bresenham-Algorithmus zum Zeichnen einer Linie                 */
/*                                                                                     */
/***************************************************************************************/

private void bresenham_linie(                   // zeichnet mit Bresenham-Algorithmus
        Point P, Point q)                       // eine Linie von Punkt P nach Punkt q
{
        Point p = new Point(P);

        int error, delta, schwelle, dx, dy, inc_x, inc_y;
        dx = q.x - p.x;
        dy = q.y - p.y;

        if (dx>0) inc_x= 1; else inc_x=-1;
        if (dy>0) inc_y= 1; else inc_y=-1;

        if (Math.abs(dy) < Math.abs(dx)) {      // flach nach oben oder flach nach unten

                error = -Math.abs(dx);
                delta = 2*Math.abs(dy);
                schwelle = 2*error;
                while (p.x != q.x) {
                        set_pixel(p);
                        p.x+=inc_x;
                        error = error + delta;
                        if (error >0) { p.y+=inc_y; error = error + schwelle;}
                }
        } else                                  // steil nach oben oder steil nach unten
        {
                error = -Math.abs(dy);
                delta = 2*Math.abs(dx);
                schwelle = 2*error;
                while (p.y != q.y) {
                        set_pixel(p);
                        p.y+=inc_y;
                        error = error + delta;
                        if (error >0) { p.x+=inc_x; error = error + schwelle;}
                }

        }
        set_pixel(q);
}
```

3.2.1 Antialiasing

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.



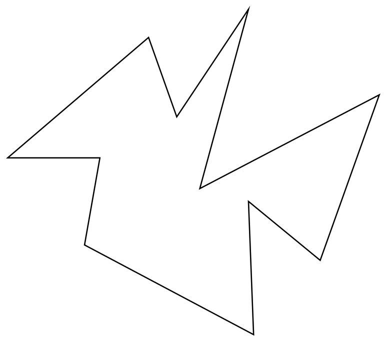
Beispiel: Linie von (1,0) nach (6,1) hat eine Überlappung

von	15 %	mit Pixel (5,2)
von	95 %	mit Pixel (5,1)
von	5 %	mit Pixel (5,0).

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.

3.3 Polygon

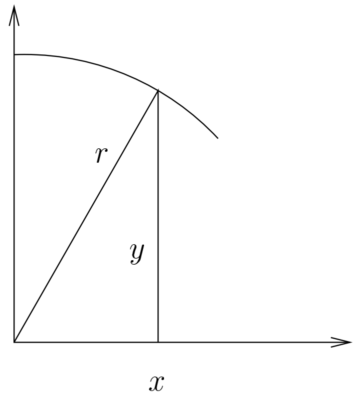
Ein Polygon wird spezifiziert durch eine Folge von Punkten, die jeweils durch Linien verbunden sind. Anfangs- und Endpunkt sind identisch.



3.4 Kreis

Problem: Gegeben Mittelpunkt (x, y) und Radius r. Bestimme die "anzuschaltenden" Pixel für einen Kreis mit Radius r um den Punkt (x, y).

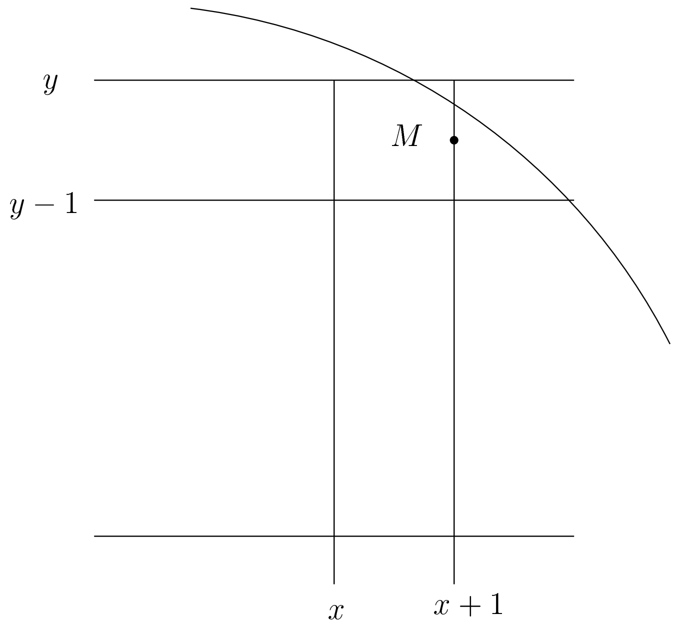
Betrachte zunächst den Verlauf eines Kreises um $(0, 0)$ im 2. Oktanten.
Für einen Punkt (x, y) sei $F(x, y) = x^2 + y^2 - r^2$



Es gilt:

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

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 $y - 1$.



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

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.

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

Also ergibt sich:

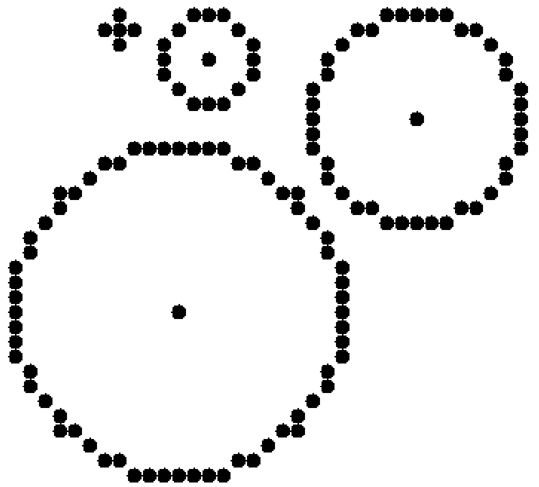
```
void bresenham_kreis_1 (             // zeichnet Kreis um Punkt 0,0
                        int r)       // mit Radius r
{
        double delta;
        int x, y;

        x=0; y=r;
        delta = 5.0/4.0 - r;

        while (y>=x) {

                set_pixel (new Point(x,y));

                if (delta < 0.0 ) { delta += 2*x + 3.0;       x++;      }
                            else { delta += 2*x - 2*y + 5.0; x++; y--; }
        }
}
```

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 `2x + 3` durch `dx` mit Initialwert `dx = 3;`
Ersetze `2x - 2y + 5` durch `dxy` mit Initialwert `dxy = -2*r + 5`
Es ergibt sich

```
void bresenham_kreis_2 (            // zeichnet Kreis um Punkt 0,0
                        int r)      // mit Radius r
{
        int x, y, d, dx, dxy;

        x=0; y=r; d=1-r;
        dx=3; dxy=-2*r+5;
        while (y>=x) {

                set_pixel (new Point(x,y));

                if (d < 0) { d += dx;  dx += 2; dxy += 2; x++;      }
                      else { d += dxy; dx += 2; dxy += 4; x++; y--; }
        }
}
```

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 $4r \cdot \sqrt{2}$ Punkte. Verglichen mit dem Kreisumfang vom $2r\pi$ liegt dieser Wert um ca. 10% zu tief.



Vom Bresenham-Algorithmus erzeugte Kreise

```
/****************************************************************************************/
/*                                                                                      */
/*                       Bresenham-Algorithmus zum Zeichnen eines Kreises               */
/*                                                                                      */
/****************************************************************************************/

private void bresenham_kreis (                      // zeichnet mit Bresenham-Algorithmus
        Point p,                                    // einen Kreis um den Punkt p
        int r)                                      // mit Radius r
{
        int x,y,d,dx,dxy;
        x=0; y=r; d=1-r;
        dx=3; dxy=-2*r+5;
        while (y>=x)
        {
                set_pixel( new Point( p.x+x, p.y+y) );  // alle 8 Oktanden werden
                set_pixel( new Point( p.x+y, p.y+x) );  // gleichzeitig gezeichnet
                set_pixel( new Point( p.x+y, p.y-x) );
                set_pixel( new Point( p.x+x, p.y-y) );
                set_pixel( new Point( p.x-x, p.y-y) );
                set_pixel( new Point( p.x-y, p.y-x) );
                set_pixel( new Point( p.x-y, p.y+x) );
                set_pixel( new Point( p.x-x, p.y+y) );

                if (d<0) { d=d+dx;  dx=dx+2; dxy=dxy+2; x++;      }
                    else { d=d+dxy; dx=dx+2; dxy=dxy+4; x++; y--; }
        }
}
```

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

Nachbarn zu $\bullet$ im 4-way-stepping:

$$\begin{array}{ccc} & \circ & \\ \circ & \bullet & \circ \\ & \circ & \end{array}$$

Nachbarn zu $\bullet$ im 8-way-stepping:

$$\begin{array}{ccc} \circ & \circ & \circ \\ \circ & \bullet & \circ \\ \circ & \circ & \circ \end{array}$$

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

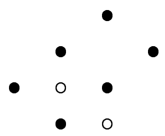
Vorteil: beliebige Umrandung möglich

Nachteil: hoher Rechen- und Speicherbedarf

Obacht: Gebiete, die nur durch 8-way-stepping erreicht werden können, werden beim Füllen mit 4-way-stepping "vergessen":

$$\begin{array}{ccccc} & & \bullet & \bullet & \bullet \\ & & \bullet & & \bullet \\ & & \bullet & & \bullet \\ & \bullet & \circ & \bullet & \\ \bullet & \circ & \circ & \bullet & \\ \bullet & \bullet & \bullet & \bullet & \end{array}$$

Wird hingegen die Nachbarschaft über 8-way-stepping definiert, so “läuft die Farbe aus”:



Benötigt werden

```
boolean get_pixel (            // liefert true, wenn Pixel p
                  Point p)     // die Vordergrundfarbe hat
```

und

```
boolean range_ok (             // liefert false, wenn Pixel p
                  Point p)     // ausserhalb des Bildbereichs
```

Der Nachteil der sehr ineffizienten Methode `boundary_fill` 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.

```
/*************************************************************************************/
/*                                                                                   */
/*                      Fuellen einer durch Vordergrundfarbe umrandeten Flaeche       */
/*                                                                                   */
/*************************************************************************************/

private void boundary_fill(            // setzt alle Nicht-Vordergrund-Pixel
                          Point p)     // beginnend bei Position p auf Vordergrund
{
        if ( range_ok(p) && !get_pixel(p) ) {     // falls p keine Vordergrundfarbe hat

                set_pixel(p);                     // setze Vordergrundfarbe

                boundary_fill( new Point(p.x+1, p.y   ));  // 4-way stepping
                boundary_fill( new Point(p.x,   p.y+1 ));
                boundary_fill( new Point(p.x-1, p.y   ));
                boundary_fill( new Point(p.x,   p.y-1 ));

        }
}




/*************************************************************************************/
/*                                                                                   */
/*                      Leeren einer durch Vordergrundfarbe definierten Flaeche       */
/*                                                                                   */
/*************************************************************************************/

private void boundary_empty(           // setzt alle Vordergrund-Pixel beginnend
                           Point p)    // bei Position p auf Hintergrundfarbe
{
        if ( range_ok(p) && get_pixel(p) ) {    // falls p Vordergrundfarbe hat

                del_pixel(p);                   // setze Hintergrundfarbe

                boundary_empty( new Point( p.x+1, p.y   ));  // 8-way-stepping
                boundary_empty( new Point( p.x+1, p.y+1 ));
                boundary_empty( new Point( p.x,   p.y+1 ));
                boundary_empty( new Point( p.x-1, p.y+1 ));
                boundary_empty( new Point( p.x-1, p.y   ));
                boundary_empty( new Point( p.x-1, p.y-1 ));
                boundary_empty( new Point( p.x,   p.y-1 ));
                boundary_empty( new Point( p.x+1, p.y-1 ));

        }
}
```

4.2 Scan-Line-Verfahren für Polygone

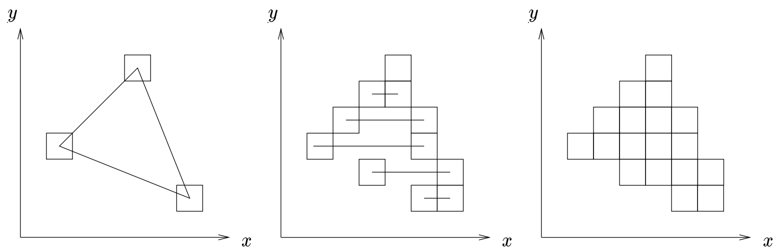
Idee: Bewege eine waagerechte Scan-Line schrittweise von oben nach unten über das Objekt, und berechne die Schnittpunkte der Scan-Line mit dem Objekt.

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.



Sortierung der Kanten nach ihrem größten y-Wert ergibt: $ABCFDEGHIJ$
Für die momentane Scan-Linie gilt: aktive Kanten sind $BDEF$
Die sortierten x-Werte der Schnittpunkte $x_1, x_2, \ldots, x_n$ ergeben die zu zeichnenden Segmente $(x_1, x_2), (x_3, x_4), \ldots$
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.

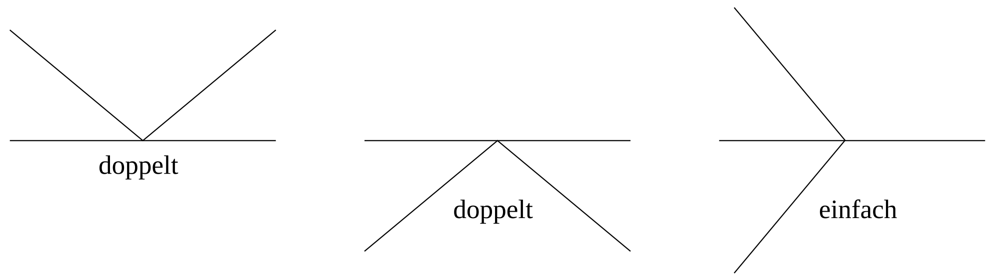
Allgemein gilt, daß ein Punkt genau dann im Inneren des Polygons liegt, wenn ein von ihm ausgehender Halbstrahl die Polygonkanten ungeradzahlig oft schneidet.

Sonderfälle

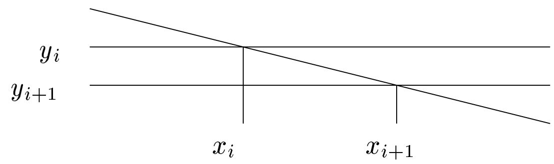
Horizontale Kanten werden nicht in die Kantenliste aufgenommen. Für sie wird eine Linie gezeichnet.

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.

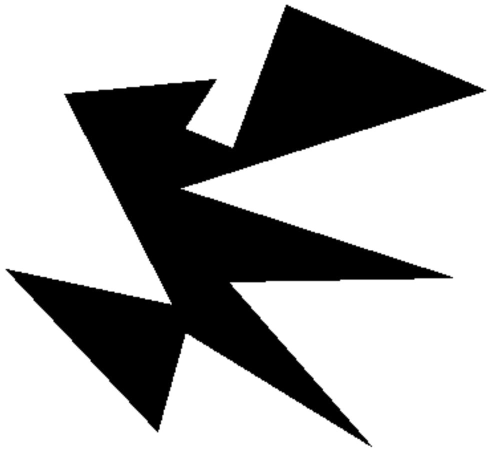
Dadurch wird sichergestellt, daß die Paare $(x_1, x_2), (x_3, x_4), \ldots$ der sortierten x-Werte der Schnittpunkte die zu zeichnenden Segmente im Inneren korrekt darstellen.



Kohärenz-Eigenschaft
Die Schnittpunkte für Scan-Line y_{i+1} lassen sich mit Hilfe der Schnittpunkte von Scan-Line y_i bestimmen:



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



Vom Scan-Line-Algorithmus gefülltes Polygon

In der folgenden Methode `scan_line_fill` wird als Datenstruktur für die Liste der Polygonkanten die Klasse `Edge` verwendet.

```
/** Klasse zur Implementation einer verzeigerten Kantenliste. */

public class Edge {

        /** groesste y-Koordinate der Kante. */
        public int      y_top;

        /** Schnittpunkt der Scan-Line mit der Kante. */
        public double   x_int;

        /** y-Ausdehnung der Kante. */
        public int      delta_y;

        /** inverse Steigung 1/s der Kante. */
        public double   delta_x;

        /** naechste Kante in der Kantenliste. */
        public Edge     next;

        /** Erzeugt ein Objekt vom Typ Kante mit den uebergebenen Parametern.
          * next ist die naechste Kante in der Liste.                       */

        public Edge(    int     y_top,
                        double  x_int,
                        int     delta_y,
                        double  delta_x,
                        Edge    next) {

                this.y_top    = y_top;
                this.x_int    = x_int;
                this.delta_y  = delta_y;
                this.delta_x  = delta_x;
                this.next     = next;
        }

        /** Erzeugt ein Objekt vom Typ Kante mit den uebergebenen Parametern. */

        public Edge(    int     y_top,
                        double  x_int,
                        int     delta_y,
                        double  delta_x) {

                this(y_top, x_int, delta_y, delta_x, null);
        }

        /** Erzeugt ein leeres Objekt vom Typ Kante. */

        public Edge() {

                this(0, 0.0, 0, 0.0, null);
        }
}
```

```
/*************************************************************************************/
/*                                                                                   */
/*                   fuellt mit dem Scan-Line-Algorithmus das Innere eines Polygons   */
/*                                                                                   */
/*************************************************************************************/

private       void Insert(                         /* fuegt in die Liste eine Kante ein */
        Edge edges,                                /* Beginn der Kantenliste            */
        Point P1, Point p2,                        /* einzufuegende Kante (P1,P2)       */
        int y_next)                                /* Behandlung von Sonderfaellen:     */
                                                   /* siehe Prozedur Next_y             */
{
        int max_y, dy;
        double x2, dx, max_x;

        Point P2 = new Point(p2);

        dx=(double)(P2.x-P1.x)/(double)(P2.y-P1.y);
        x2=(double) P2.x;

        if ((P2.y > P1.y) && (P2.y < y_next)) { P2.y--; x2=x2-dx; }     // Sonderfaelle
        if ((P2.y < P1.y) && (P2.y > y_next)) { P2.y++; x2=x2+dx; }

        dy=P2.y-P1.y;
        if (dy>0) { max_y = P2.y;
                    max_x = (double)x2;
                    dy++;
        } else    { max_y=P1.y;
                    max_x=(double)P1.x;
                    dy=1-dy;
        }

        Edge edge1 = edges;                                             // Hilfsobjekt

        while (edge1.next.y_top >= max_y) edge1 = edge1.next;

        Edge newedge = new Edge(max_y, max_x, dy, dx, edge1.next);      // einfuegen
                                                                        // sortiert nach
        edge1.next = newedge;                                           // max_y
}
```

```
private int Next_y(     /* liefert den y-Wert des naechsten Knoten laengs der Grenze   */
                        /* dessen y-Koordinate verschieden ist von P[k].y              */
        int k,                                          /* Index des Punktes           */
        Point[] Points,                                 /* Liste von Punkten           */
        int n)                                          /* Anzahl der Punkte           */
{
        int compare_y, new_y;

        compare_y = Points[k].y;

        do {
                k = (k+1)%n;
                new_y = Points[k].y;
        } while (new_y == compare_y);

        return(new_y);
}

private int Edge_Sort(                          /* erzeugt nach y sortierte Kantenliste */
                                                /* und liefert den kleinsten y-Wert     */
        int n,                                  /* Anzahl der Punkte                    */
        Point[] P,                              /* Punkteliste                          */
        Edge edges)                             /* Kantenliste                          */
{
        int bottom_y;
        Point P1;

        Edge edge1 = new Edge();

        edges.next=edge1;
        edge1.next=null;

        edges.y_top=Integer.MAX_VALUE;
        edge1.y_top=-Integer.MAX_VALUE;

        P1 = P[n-1];
        bottom_y = P1.y;

        for (int k=0; k<n; k++) {
                if (P1.y != P[k].y) Insert(edges,P1,P[k],Next_y(k,P,n));
                        else set_dither_line(P1,P[k].x);
                if (P[k].y < bottom_y) bottom_y = P[k].y;
                P1 = P[k];
        }
        return (bottom_y);
}
```

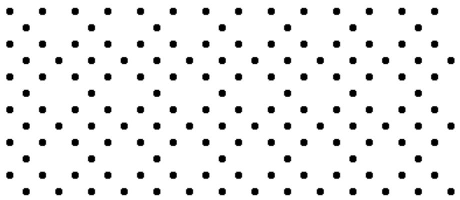
```
private Edge Update_List_Ptr(                /* aktualisiert den Zeiger auf die letzte */
                                             /* aktive Kante und gibt ihn zurueck      */
                                             /* wegen Dekrementieren der Scan-Line     */
                                             /* werden einige Kanten aktiv             */
        int scan,
        Edge l_act_edge)
{
        while (l_act_edge.next.y_top >= scan) l_act_edge=l_act_edge.next;

        return(l_act_edge);
}


private        Edge Sort_Intersections (      /* sortiert die aktive Kantenliste       */
        Edge edges,                           /* Beginn der Kantenliste                */
        Edge l_act_edge)                      /* Ende der Kantenliste                  */
                                              /* Liefert den ggf. modifizierten Zeiger */
                                              /* auf die letzte aktive Kante zurueck   */
{
        Edge edge1, edge2, edge3;

        edge2 = edges.next;

        do {
                edge1=edges;
                while (edge1.next.x_int < edge2.next.x_int)
                        edge1=edge1.next;
                if (edge1 != edge2) {              // tausche edge1.next und edge2.next
                        edge3          = edge2.next.next;
                        edge2.next.next = edge1.next;
                        edge1.next     = edge2.next;
                        edge2.next     = edge3;
                        if (edge1.next==l_act_edge) l_act_edge=edge2;
                } else edge2 = edge2.next;

        } while (edge2 != l_act_edge);

        return (l_act_edge);
}
```

```
private void Fill(                       /* generiert fuer je zwei Schnittpunkte   */
                                         /* aus der Kantenliste den Zeichne-Aufruf */
        Edge edges,                      /* Beginn der aktuellen Kantenliste       */
        Edge l_act_edge,                 /* Ende der aktuellen Kantenliste         */
        int scan)                        /* Scanline                               */
{
        Point Q = new Point();

        do {
                edges = edges.next;
                Q.x = (int) (edges.x_int+0.5);
                Q.y = scan;
                edges = edges.next;
                set_dither_line(Q,(int)(edges.x_int+0.5) );

        } while (edges != l_act_edge);
}

private        Edge Update_Edges(   /* aktual. die aktiven Kanten in der Kantenliste   */
        Edge edges,                 /* beginnend bei edges                             */
        Edge l_act_edge)            /* und endend bei l_act_edge                       */
                                    /* Kanten mit delta_y=0 werden entfernt. Der ggf.  */
                                    /* modifizierte Zeiger auf die letzte aktive Kante */
                                    /* wird zurueckgegeben                             */
{
        Edge prev_edge;

        prev_edge = edges;

        do {
                edges = prev_edge.next;
                if (edges.delta_y > 1) {
                        edges.delta_y--;
                        edges.x_int = edges.x_int - edges.delta_x;
                        prev_edge = edges;
                } else
                {
                        prev_edge.next = edges.next;
                        if (edges == l_act_edge) l_act_edge = prev_edge;
                        edges = null;        /* dispose edges */
                } /* if */
        } while (prev_edge != l_act_edge);
        return (l_act_edge);
}
```

```
private void scan_line_fill(                    /* Fuellt das Innere eines Polygons */
        int NumPoints,                          /* Anzahl der Punkte                */
        Point[] Points)                         /* Liste von Punkten                */
{
        Edge l_act_edge;
        int scan, bottom_y;

        Edge edges = new Edge();

        bottom_y = Edge_Sort(NumPoints, Points, edges);

        l_act_edge = edges.next;

        for (scan = edges.next.y_top; scan >= bottom_y; scan--) {
                l_act_edge = Update_List_Ptr(scan, l_act_edge);
                l_act_edge = Sort_Intersections(edges, l_act_edge);
                Fill (edges, l_act_edge, scan);
                l_act_edge = Update_Edges(edges, l_act_edge);
        }

        /* dispose dummies edges.next und edges */
        edges.next = null;
        edges = null;
}
```

4.3 Dithering

Def.: Eine $n \times n$-Dithermatrix M 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 $M[i \mod n, j \mod n] < k$.



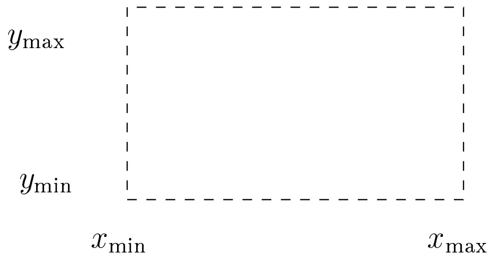
Zum Schwellwert 7 gehörender Grauwert

```
/***************************************************************************************/
/*                                                                                     */
/*                      Definition einer Dithermatrix mit Routinen dafuer             */
/*                                                                                     */
/***************************************************************************************/

private static final int DITHER_DIM  = 8;       // 8x8-Dithermatrix

private static final int[][] dit_mat =
{       { 0,32, 8,40, 2,34,10,42},              // 64-elementige Dithermatrix:
        {48,16,56,24,50,18,58,26},              // Bei Grauwert k werden alle
        {12,44, 4,36,14,46, 6,38},              // Pixel i,j auf schwarz
        {60,28,52,20,62,30,54,22},              // gesetzt, fuer die gilt:
        { 3,35,11,43, 1,33, 9,41},              // dit_mat[i%8,j%8] < k
        {51,19,59,27,49,17,57,25},
        {15,47, 7,39,13,45, 5,37},
        {63,31,55,23,61,29,53,21},
};

boolean kleiner_schwelle(                       // testet die Dither-Matrix
                          Point p )             // bezueglich des Punktes
{
        int x = p.x; int y = p.y;

        if (x<0) x=DITHER_DIM - (Math.abs(x)%DITHER_DIM);  // falls x negativ
        if (y<0) y=DITHER_DIM - (Math.abs(y)%DITHER_DIM);  // falls y negativ

        return (dit_mat[x%DITHER_DIM][y%DITHER_DIM] < schwelle);
}

int grauwert(                                   // liefert den Grauwert
              int i, int j )                    // an Position i, j

{       return(dit_mat[i][j]); }
```

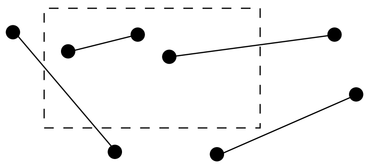
Kapitel 5

Clipping

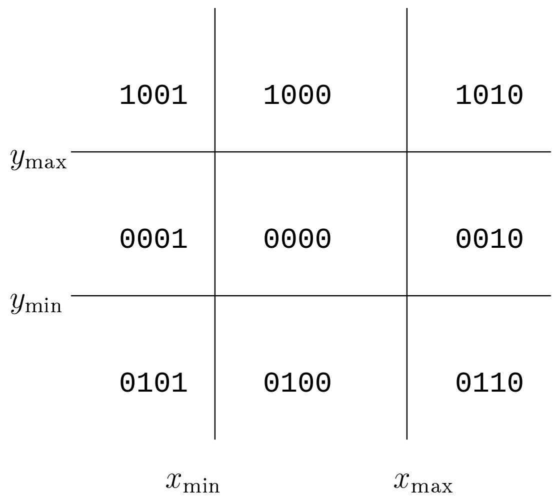
Ziel: Nur den Teil eines Objekts darstellen, der innerhalb eines Fensters sichtbar ist.



5.1 Clipping von Linien

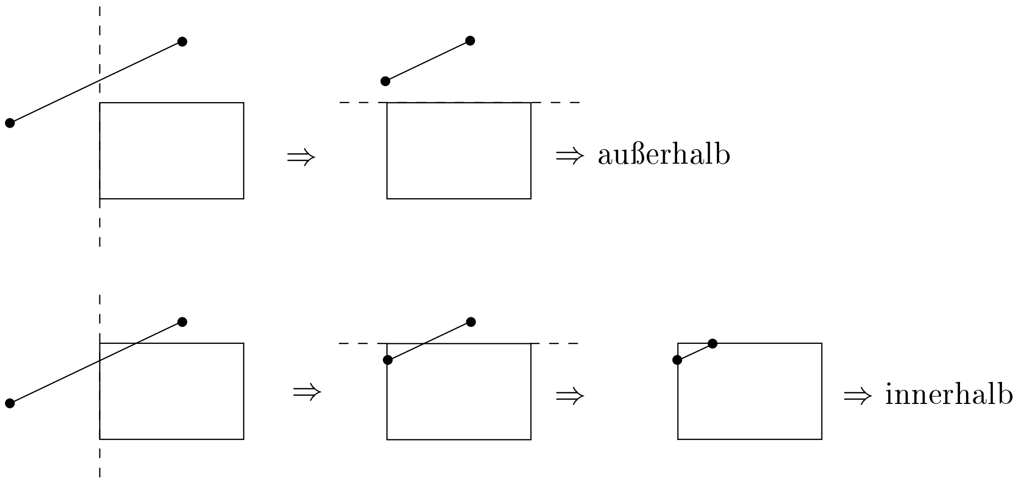


Idee von Cohen & Sutherland
Teile Ebene anhand des Clip-Fensters in 9 Bereiche ein, wähle 4-Bit-Bereichscode für jeden Punkt:



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.

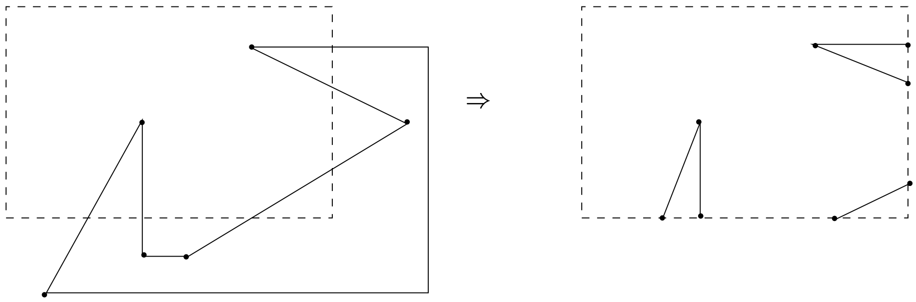


```
/***********************************************************************************/
/*                                                                                 */
/*              Clippen von Linien an einem Fenster nach Cohen-Sutherland          */
/*                                                                                 */
/***********************************************************************************/

private static final byte EMPTY  = 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;

private int xmin, xmax, ymin, ymax;             // Clip-Fensterraender

private byte region_code(                       // liefert den region-code
                          Point P )             // fuer den Punkt P
{
        byte c;

        c = EMPTY;
        if (P.x < xmin) c = LEFT; else
        if (P.x > xmax) c = RIGHT;
        if (P.y < ymin) c |= BOTTOM; else
        if (P.y > ymax) c |= TOP;
        return(c);
}


private void set_clip_window(         // setzt die Variablen xmin, ymin, xmax, ymax
                Point P,              // anhand des Ursprungs P des Clip-Fensters
                Point delta )         // und anhand seiner Breite/Hoehe delta
        {
                xmin = P.x; xmax = P.x + delta.x;
                ymin = P.y; ymax = P.y + delta.y;
        }


private boolean cohen_sutherland(     // liefert true,
                Point p1, Point p2,   // falls die Gerade p1-p2 sichtbar ist
                Point Q1, Point Q2)   // liefert ggf. sichtbaren Teil Q1-Q2 zurueck
{
        boolean finite_slope;         // true falls Gerade p1-p2 nicht senkrecht laeuft
        double slope = 0.0;           // Steigung der Geraden p1-p2
        byte C, C1, C2;               // 4-Bit-Bereichs-Code
        Point Q = new Point();        // zu berechnender Schnittpunkt mit Gerade

        Point P1 = new Point(p1);     // lokale
        Point P2 = new Point(p2);     // Variablen
        finite_slope = (P1.x != P2.x);
        if (finite_slope) slope = (double)(P2.y-P1.y)/(double)(P2.x-P1.x);
        C1 = region_code(P1);
        C2 = region_code(P2);
```

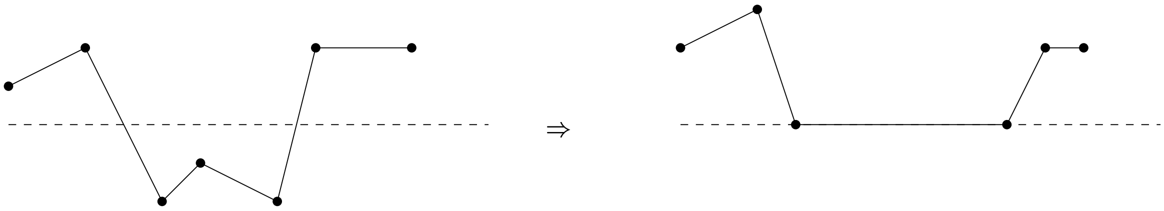
```
        while ((C1 != EMPTY) || (C2 != EMPTY))  { // mind. ein Endpunkt noch ausserhalb

            if ((C1&C2) != EMPTY) return(false);  // beide auf derselben Seite ausserhalb
            else
            {
                if (C1 == EMPTY) C = C2; else C = C1; // C ist ausserhalb. Berechne
                                                      // einen Schnittpunkt mit den
                                                      // verlaengerten Fensterkanten

                if ((C & LEFT) != EMPTY) {         // schneide mit linker Fenster-Kante
                        Q.x = xmin;
                        Q.y = (int)((Q.x-P1.x)*slope + P1.y);
                        } else

                if ((C & RIGHT) != EMPTY){        // schneide mit rechter Fenster-Kante
                        Q.x = xmax;
                        Q.y = (int)((Q.x-P1.x)*slope + P1.y);
                        } else

                if ((C & BOTTOM) != EMPTY) {      // schneide mit unterer Fenster-Kante
                        Q.y = ymin;
                        if (finite_slope)
                                Q.x = (int)((Q.y-P1.y)/slope + P1.x); else
                                Q.x = P1.x;
                        }else

                if ((C & TOP) != EMPTY) {         // schneide mit oberer Fenster-Kante
                        Q.y = ymax;
                        if (finite_slope)
                                Q.x = (int)((Q.y-P1.y)/slope + P1.x); else
                                Q.x = P1.x;
                        }

                if (C==C1) {
                        P1.x = Q.x; P1.y = Q.y; C1 = region_code (P1);
                } else {
                        P2.x = Q.x; P2.y = Q.y; C2 = region_code (P2);
                }
            }

        }
        // uebergib Anfang und Ende des sichtbaren Teils
        Q1.x=P1.x; Q1.y=P1.y;
        Q2.x=P2.x; Q2.y=P2.y;

        return(true);
}
```

5.2 Clipping von Polygonen

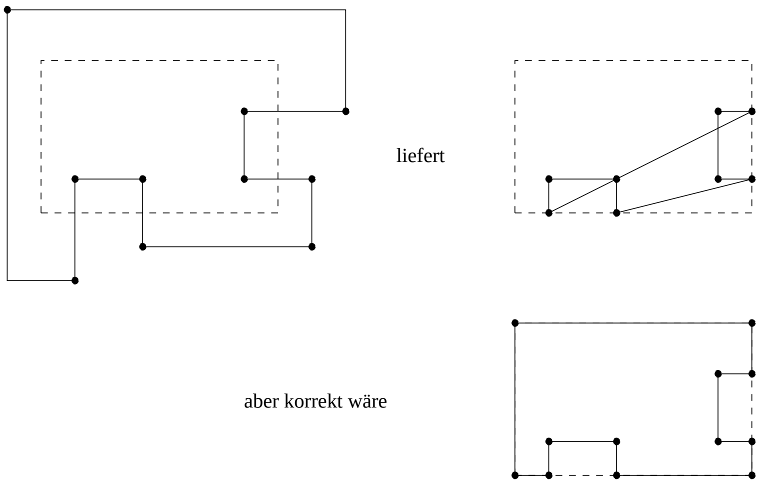
Für das Clippen von Polygonen reicht es nicht, jede beteiligte Kante zu clippen.



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:

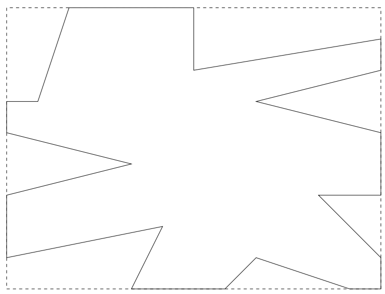


Der **Sutherland & Hodgman-Algorithmus** clippt daher an den vier Fensterkanten nacheinander:

```
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
```

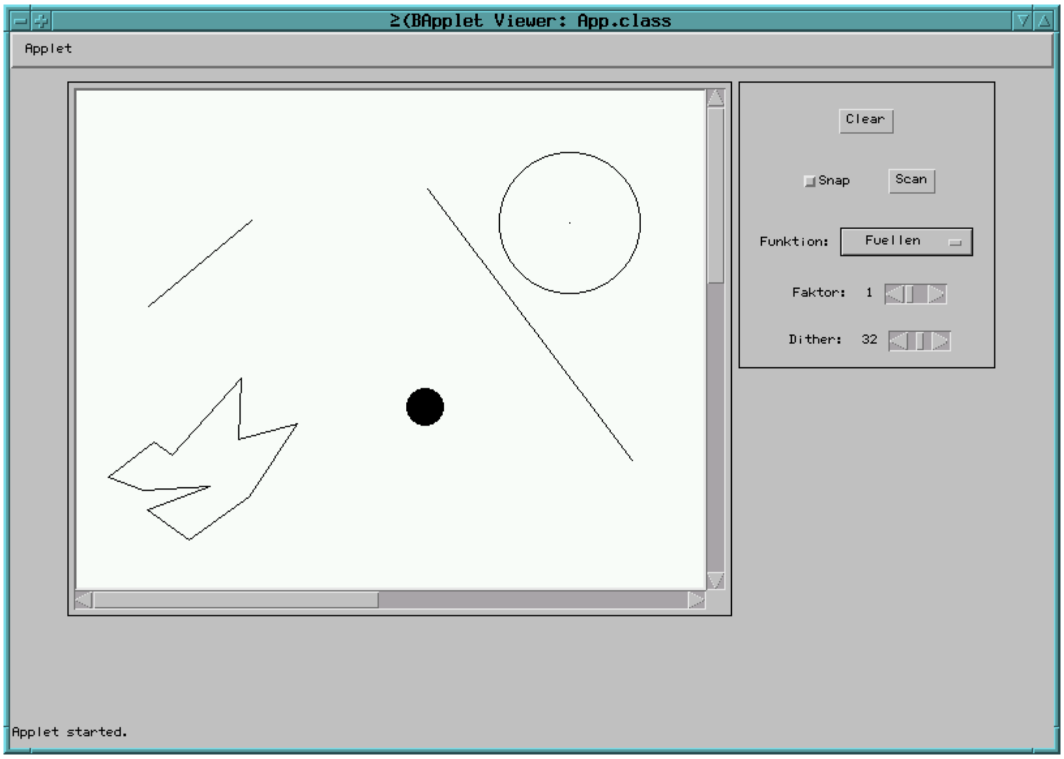
```
/*****************************************************************************************/
/*                                                                                       */
/*           Clippen von Polygonen an einem Fenster nach Sutherland-Hodgmann             */
/*                                                                                       */
/*****************************************************************************************/

private boolean On_Visible_Side(                        /* liefert true, falls          */
                        Point P,                        /* Punkt p auf der sichtbaren   */
                        int wert,                       /* Seite der Kante wert liegt   */
                        int fall)                       /* gemaess Fall fall            */
{
        switch (fall) {

                case 0: return (P.x >= wert);        /* linke  Fenster-Kante            */
                case 1: return (P.y >= wert);        /* untere Fenster-Kante            */
                case 2: return (P.x <= wert);        /* rechte Fenster-Kante            */
                case 3: return (P.y <= wert);        /* obere  Fenster-Kante            */
        }

        return false;
}


private boolean Intersection(                   /* liefert true,                        */
                        Point P1, Point P2,     /* falls sich die Linie P1-P2           */
                        int wert,               /* mit Clipping-Kante wert schneidet    */
                        int fall,               /* gemaess Fall fall                    */
                        Point I)                /* liefert den Schnittpunkt I zurueck   */
{
        boolean P1_vis, P2_vis;                 /* true, falls P1 bzw P2 sichtbar       */
        double slope;                           /* Steigung der Geraden P1-P2           */
                                                /* y = (x-P1.x)*slope + P1.y            */

        P1_vis = On_Visible_Side(P1,wert,fall);        /* P1 sichtbar ?                 */
        P2_vis = On_Visible_Side(P2,wert,fall);        /* P2 sichtbar ?                 */

        if ((P1_vis && P2_vis) || (!P1_vis && !P2_vis)) return (false);/*kein Schnittpkt*/
        else
        {
                slope = (double)(P2.y-P1.y) / (double) (P2.x-P1.x);   /* Steigung       */

                if (fall%2==1) {
                        I.x = (int)((wert-P1.y) / slope) + P1.x;
                        I.y = wert;
                } else
                {
                        I.x = wert;
                        I.y = (int)((wert-P1.x) * slope) + P1.y;
                }
                return(true);
        }
}
```

```
private int sutherland_hodgman(                /* fuehrt ein Clipping durch          */
                        int num_points,         /* fuer num_points Punkte             */
                        Point[] points,         /* der Polygonliste Points            */
                        int wert,               /* bezueglich Clipping-Kante wert     */
                        int fall)               /* gemaess Fall fall                  */
/* liefert neue Anzahl Punkte */
{
        int   i;
        int   old_ptr;
        int   new_ptr;
        Point S;
        Point I;
        Point[] hpoints = new Point[MAX_POINTS];

        if (num_points > 2) {
                old_ptr = 0;
                new_ptr = -1;
                points[num_points] = points[0];

                while (old_ptr < num_points) {

                        S = points[old_ptr];
                        if (On_Visible_Side(S,wert,fall)) {
                                new_ptr++;
                                hpoints[new_ptr] = S;
                        }
                        old_ptr++;
                        I = new Point();
                        if (Intersection(S,points[old_ptr],wert,fall,I)) {
                                new_ptr++;
                                hpoints[new_ptr] = I;
                        }
                }

                num_points = new_ptr+1;
                for (i=0; i<num_points; i++) points[i]=hpoints[i];
        }
        return num_points;
}
```



Vom Sutherland-Hodgman-Algorithmus geclipptes Polygon

5.3 Beispiel-Applet zu 2D-Operationen



Screenshot vom 2D-basic-Applet

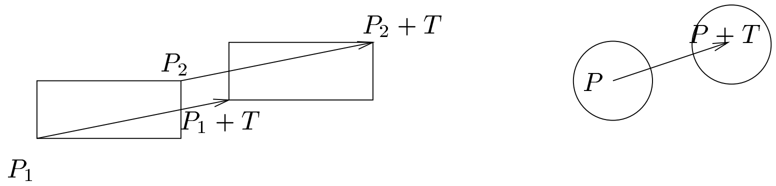
Kapitel 6

2D-Transformationen

Transformationen eines Objekts werden durch Operationen auf den Definitionspunkten realisiert.
Durch 2 Punkte definierte Rechtecke müssen zunächst zu Polygonen gemacht werden.

6.1 Translation

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



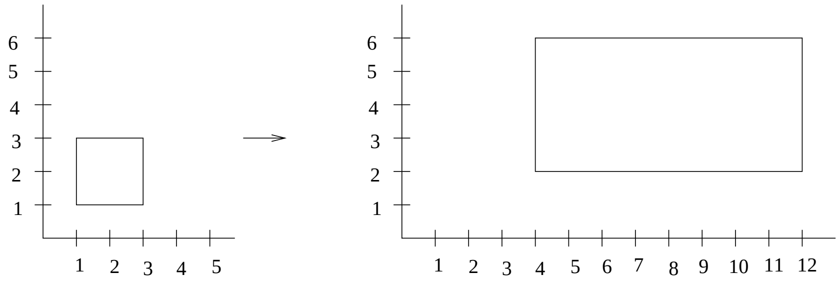
6.2 Skalierung (Scaling, Zooming)

Vergrößerung bzw. Verkleinerung bzgl. eines Fixpunktes. Es 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

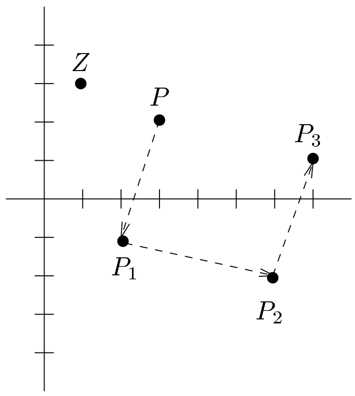
Beispiel: $s_x = 4, s_y = 2$



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 $P_3 = P'$.

Beispiel: $Z = (1, 3)$, $s_x = 3$, $s_y = 2$



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

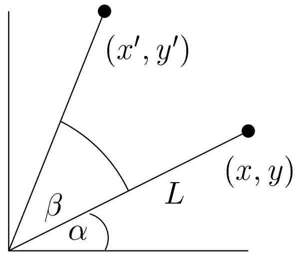
Vorteilhafter bei mehreren Objekten:

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

6.3 Rotation

Drehung des Objekts bzgl. eines Fixpunktes um einen Winkel β.
Es liege der Fixpunkt im Ursprung.
Es gilt der Winkelsummensatz

$$\begin{aligned}\sin(\alpha+\beta) &= \cos(\beta)\cdot\sin(\alpha)+\sin(\beta)\cdot\cos(\alpha)\\ \cos(\alpha+\beta) &= \cos(\beta)\cdot\cos(\alpha)-\sin(\beta)\cdot\sin(\alpha)\end{aligned}$$



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

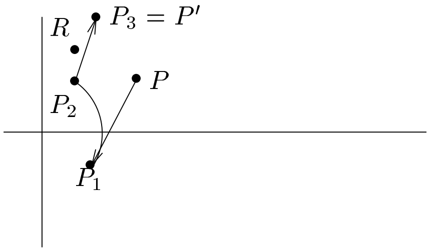
$$\begin{aligned}\sin(\alpha+\beta)=y'/L &= \cos(\beta)\cdot\sin(\alpha)+\sin(\beta)\cdot\cos(\alpha)\\ \Rightarrow y' &= L\cdot\cos(\beta)\cdot\frac{y}{L}+L\cdot\sin(\beta)\cdot\frac{x}{L}\\ \Rightarrow y' &= y\cdot\cos(\beta)+x\cdot\sin(\beta)\end{aligned}$$

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

Positive Werte für den Drehwinkel β bewirken eine Rotation gegen den Uhrzeigersinn.
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 $P_3 = P'$

Beispiel für $\beta = 90°$:



```
/*****************************************************************************/
/*                                                                           */
/*                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 Bogenmass


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 Homogene Koordinaten

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.

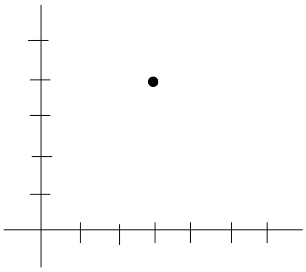
Def.: Ein Punkt $P = (x, y)$ hat die *homogenen* Koordinaten $[x_h, y_h, w]$ mit $w \neq 0$ und

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

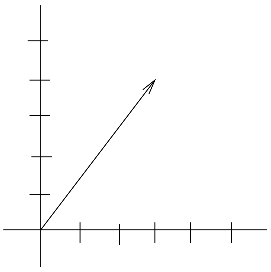
Beispiel: Das homogene Koordinatentripel $[6, 8, 2]$ gehört zum Punkt $P = (3, 4)$, $[3, 4, 1]$ auch.

Ein Richtungsvektor $R = (x, y)$ hat die homogenen Koordinaten $[x, y, 0]$.

Beispiel: Das homogene Koordinatentripel $[3, 4, 0]$ gehört zum Richtungsvektor $(3, 4)$.



Punkt (3, 4)



Richtungsvektor (3, 4)

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

Translation

$$[x', y', 1] := [x, y, 1] \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{bmatrix} = [x + t_x, y + t_y, 1]$$

Skalierung

$$[x', y', 1] := [x, y, 1] \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} = [x \cdot s_x, y \cdot s_y, 1]$$

Rotation

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

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

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

Matrix für Rotation um 60° lautet

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

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

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

Matrix für gesamte Transformation lautet

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

6.5 Allgemeine Transformationen

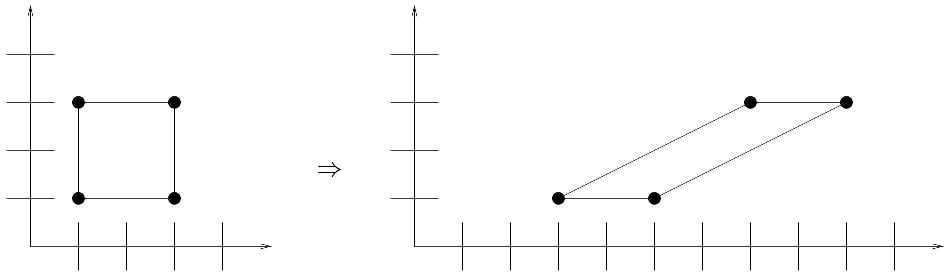
Weitere Transformationen, die sich durch Matrizen darstellen lassen, sind

- 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:

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

Beispiel: Scherung mit $Sch_x = 2$



6.6 Raster-Transformationen

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

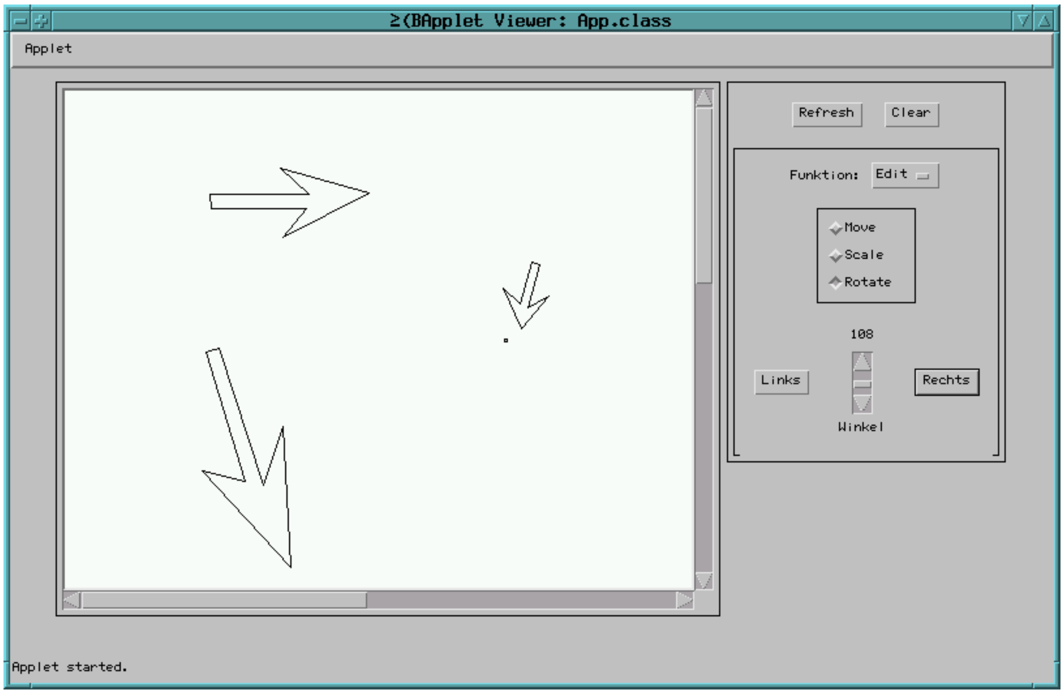
- Verschieben eines Fensterinhalts (Move),
- Kopieren eines Fensterinhalts (Copy),
- Vergrößern eines Fensterinhalts (Zoom).

Statt Pixelwerte zu setzen, können auch AND-, OR-, XOR-Verknüpfungen verwendet werden. Es gilt (x XOR y) 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.

6.7 Beispiel-Applet zu 2D-Transformationen



Screenshot vom 2D-trafo-Applet

Kapitel 7

Kurven

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.

7.2 Splines

Gegeben n Stützpunkte.
Wähle $n-1$ Kurven kleiner Ordnung für den Verlauf innerhalb der $n-1$ Intervalle.
Kurven erster Ordnung (Geraden) ergeben nur den Linienzug.
Kurven zweiter Ordnung (Parabeln) sind an den Intervallgrenzen nur einmal differenzierbar.
Kurven dritter Ordnung mit stetigen ersten und zweiten Ableitungen an den Intervallgrenzen (kubische Splines) können wie folgt bestimmt werden:

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

Bewertung: f ist in parametrischer 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.

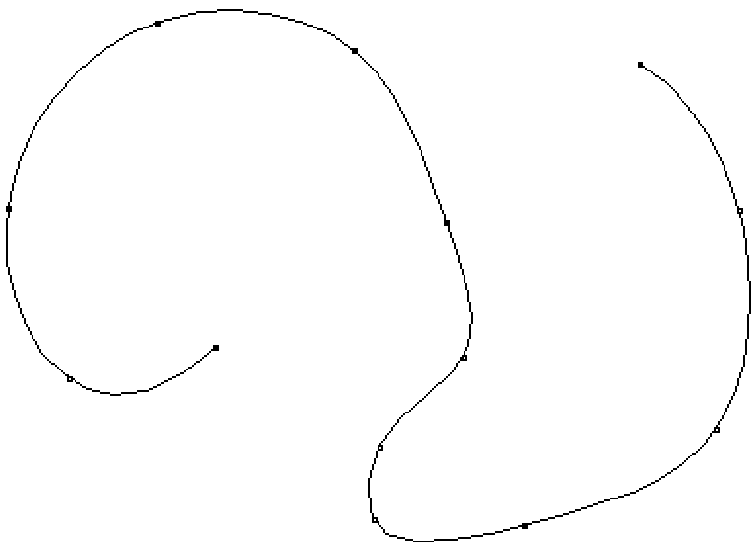
Die n Stützpunkte seien $f(t_i)$, die resultierenden Intervalle seien $[t_i, t_{i+1}]$, die für das Intervall i zuständige Funktion sei $f_i(t)$, $1 \leq i \leq n-1$. Jedes f_i hat die Form

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

Es muß gelten

$$
\begin{array}{lclll}
f_i(t_{i+1}) & = & f_{i+1}(t_{i+1}) & \text{für} \quad i = 1, \ldots, n-2 & \text{stetig} \\
f_i'(t_{i+1}) & = & f_{i+1}'(t_{i+1}) & \text{für} \quad i = 1, \ldots, n-2 & \text{differenzierbar} \\
f_i''(t_{i+1}) & = & f_{i+1}''(t_{i+1}) & \text{für} \quad i = 1, \ldots, n-2 & \text{keine Steigungsänderung}
\end{array}
$$

Nach Vorgabe von a_1 und a_{n-1} entsteht ein Gleichungssystem mit $n-3$ Gleichungen und $n-3$ 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 $\frac{1}{3} \cdot (dx + dy + 2 \cdot \max(dx, dy))$.



Vom Spline-Algorithmus erzeugte Kurveninterpolation mit 13 Stützpunkten und 65 Interpolationspunkten (pro Intervall)

```
/*************************************************************************************/
/*              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-Koordinaten der Stuetzpunkte
private static double[] y = new double[MAX_POINTS];     // y-Koordinaten der Stuetzpunkte
private static double[] t = new double[MAX_POINTS];     // t-Werte der Stuetzpunkte

void besetze_arrays(                            // berechnet Intervallgrenzen
              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-Koordinaten
              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 der Stuetzpunke
              ax=Math.abs(x[i]-x[i-1]);
              ay=Math.abs(y[i]-y[i-1]);
              dd = ax + ay;
              if (ax > ay) dd = dd+2*ax;        // Naeherungsformel fuer Euklid:
                      else dd = dd+2*ay;        // Abstand = 1/3*(dx+dy+2*max{dx,dy})
              t[i] = t[i-1]+dd/3;
      }
}

void curve_fitting(                             // berechnet die Approximation der Kurve
      int point_cnt,                            // fuer 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 und
                                                              // Gleitkommaversion von punkt
      cubic_spline(point_cnt, t, x, spline_size, calcfx);  // bestimme die Interpol.werte
      cubic_spline(point_cnt, t, y, spline_size, calcfy);  // bestimme die Interpol.werte
      for (i=0; i<spline_size; i++)                           // und uebertrage sie
        spline_punkt[i+1] = new Point((int)calcfx[i],(int)calcfy[i]);
}
void cubic_spline(                      // loest Gleichungssystem zur Bestimmung
                                        // der Koeffizienten a,b,c
              int n,                    // Anzahl der Stuetzpunkte
              double t[], double f[], // Stuetzpunkte in der Form (t[i],f[t[i]])
              int spline_size,          // Anzahl der Interpolationswerte
              double calcf[])           // Ausgabearray: 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

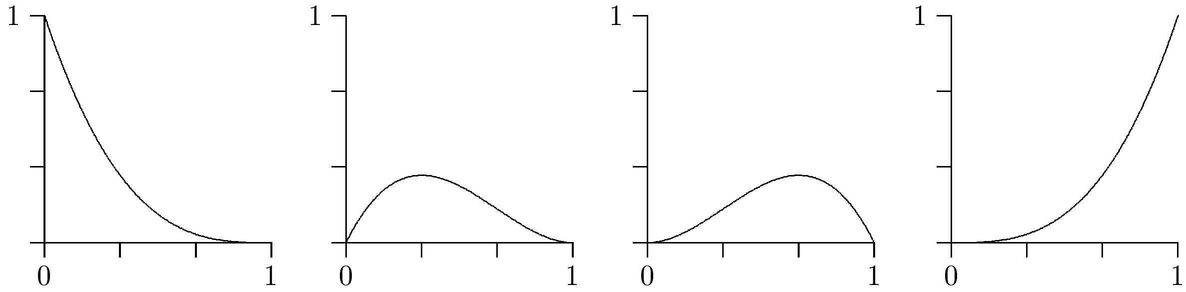
Spezifiziere die Kurve durch $n+1$ Stützpunkte $P_0, P_1, \ldots, P_n$. Die Kurve läuft nicht durch alle Stützpunkte, sondern wird von ihnen beeinflußt.

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

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

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

Beispiel: $n = 3$



$B_{0,3} = (1-t)^3$ $B_{1,3} = 3 \cdot t \cdot (1-t)^2$ $B_{2,3} = 3 \cdot t^2 \cdot (1-t)$ $B_{3,3} = t^3$

Die wichtigsten Eigenschaften der Bernsteinpolynome:

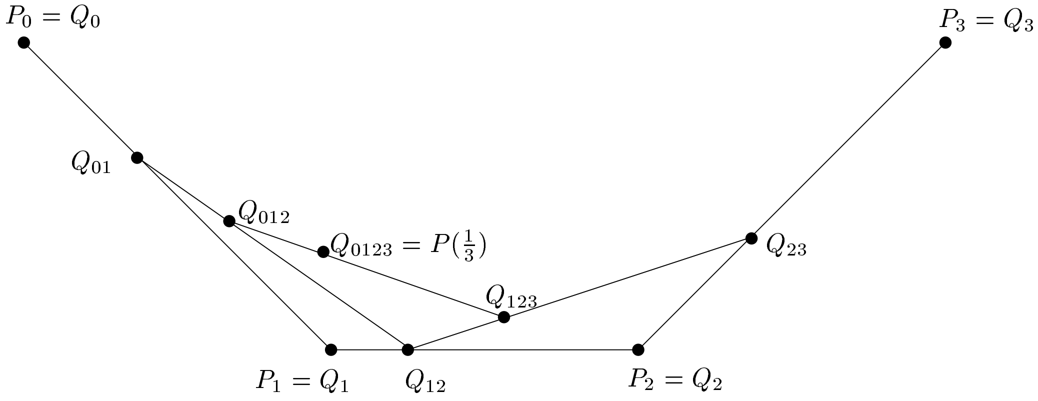
- alle Bernsteinpolynome sind positiv auf $[0, 1]$,
- für jedes feste t gilt $\sum_{i=0}^{n} B_{i,n}(t) = 1$,
- $B_{i,n}(t) = B_{n-i,n}(1-t)$,
- $\frac{dB_{i,n}(t)}{dt} = n \cdot (B_{i-1,n-1}(t) - B_{i,n-1}(t)), i > 0$,
- Maxima von $B_{i,n}(t)$ in $[0, 1]$ an den Stellen $t = \frac{i}{n}, i = 0, \ldots,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

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 $\overline{P_0 P_1}$, 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
Es gilt:

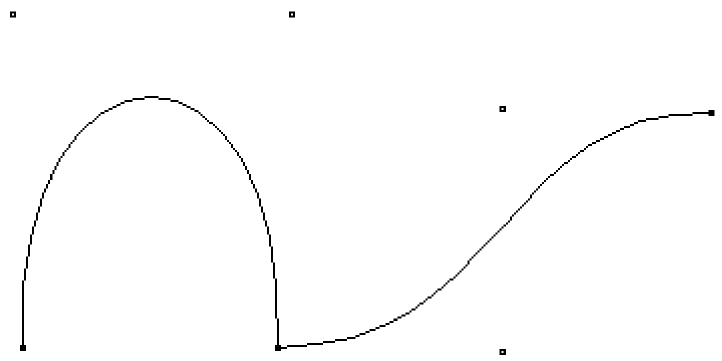
$$P(t)_{0,n} = (1-t) \cdot P(t)_{0,n-1} + t \cdot P(t)_{1,n}$$

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.



Berechnung eines Kurvenpunktes für $t = \frac{1}{3}$. Die Indizes geben an, welche Stützpunkte beteiligt sind.

```
/*****************************************************************************************/
/*                                                                                       */
/*                     Zeichnen einer Bezierkurve 3. Grades nach De Casteljau            */
/*                                                                                       */
/*****************************************************************************************/

private void bezier(                            // Zeichne eine Bezierkurve 3. Grades
                    double px0, double py0,     // anhand des Stuetzpunktes p0
                    double px1, double py1,     // anhand des Stuetzpunktes p1
                    double px2, double py2,     // anhand des Stuetzpunktes p2
                    double px3, double py3,     // anhand des Stuetzpunktes p3
                    int depth)                  // aktuelle Rekursionstiefe
{
        double qx01,qy01,qx12,qy12,qx23,qy23,qx012,qy012,qx123,qy123,qx0123,qy0123;
                                                // Hilfspunkte

        if (depth > iter)                       // Iterationstiefe erreicht
                set_line( new Point( (int)px0, (int)py0), 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);
        }
}
```



Vom De Casteljau-Algorithmus mit Rekursionstiefe 3 gezeichnete Bézierkurven

7.4 B-Splines

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

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

Die Punkte P_i werden gewichtet durch die Basis des B-Splines, Polynome $N_{i,k}(t)$ vom Grad $k-1$ $(0 \le i \le n)$:

$$\begin{aligned}
N_{0,k}(0) &= 1, \text{ für alle } k \\
N_{i,1}(t) &= \begin{cases} 1 & \text{falls } t_i < t \le t_{i+1} \\ 0 & \text{sonst} \end{cases} \\
N_{i,k}(t) &= \frac{t - t_i}{t_{i+k-1} - t_i} \cdot N_{i,k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} \cdot N_{i+1,k-1}(t), \ k > 1
\end{aligned}$$

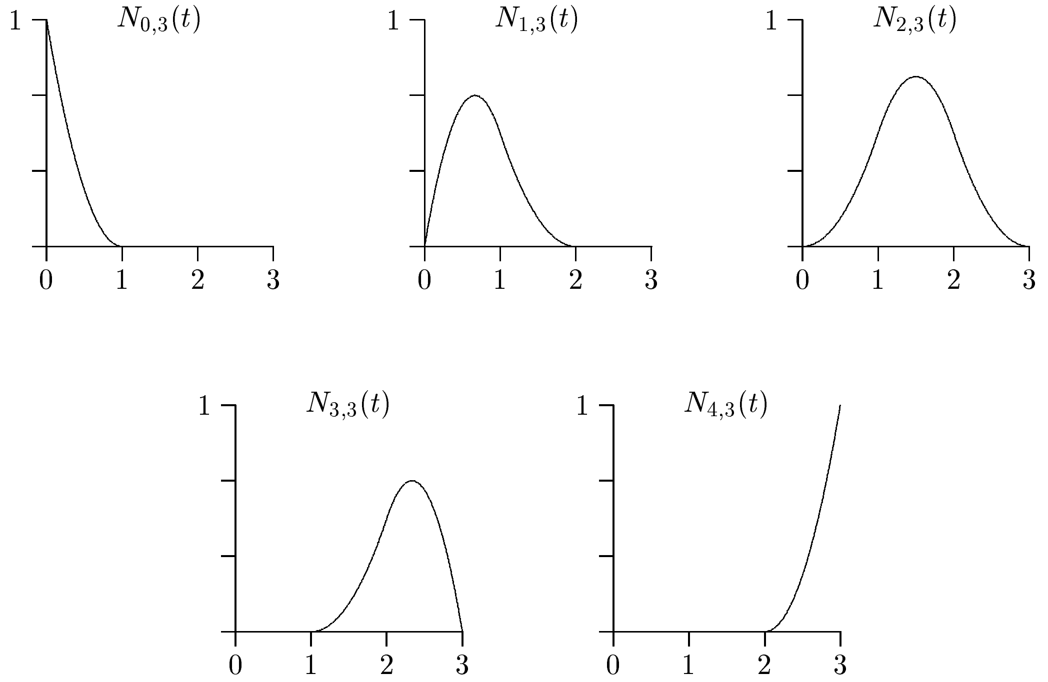
Bei Division durch Null wird der Quotient gleich 0 gesetzt.
Der Knotenvektor wird häufig wie folgt gewählt für $0 \le j \le n+k$:

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

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

Beispiel: $k = 3$, $n = 4$ ergibt einen 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

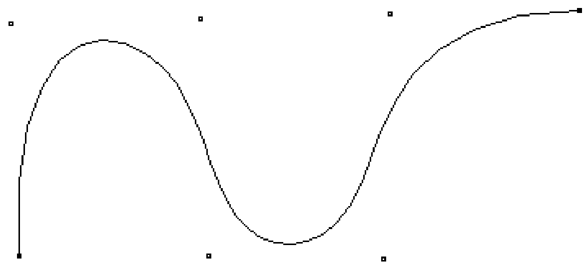
Kurvenverlauf, und zwar in den Intervallen $t \in [0,1], [0,2], [0,3], [1,3], [2,3]$.



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

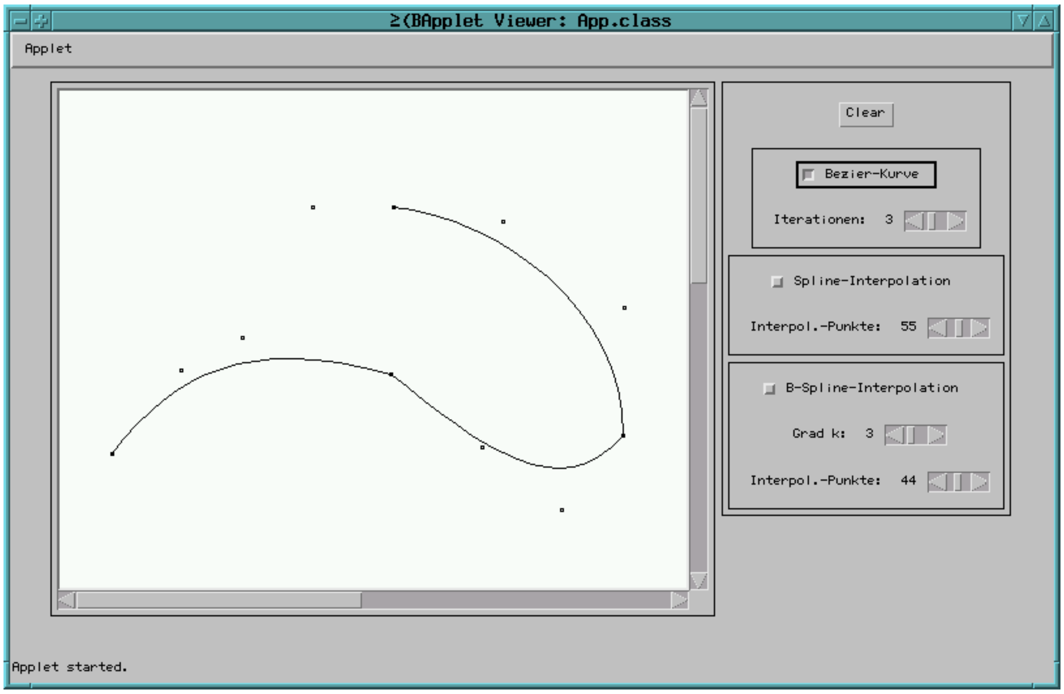
$$N_{i,k}(t) = \frac{(k-1)!}{i! \cdot (k-1-i)!} \cdot t^i \cdot (1-t)^{k-1-i}, \ i = 0, ..., k-1$$

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 $\sum_{i=0}^{n} N_{i,k} = 1$.



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

7.5 Beispiel-Applet zu Splines



Screenshot vom Splines-Applet

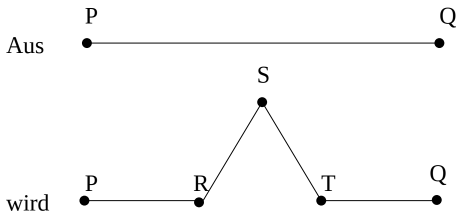
Kapitel 8

Fraktale

Die Erzeugung von Fraktal-Kurven erfolgt durch wiederholte Anwendung einer Transformation f auf eine Menge von Punkten. Charakteristisch für Fraktale ist ihre Selbstähnlichkeit (vgl. Stein im Gebirge, Blumenkohl, Broccoli).

8.1 Koch'sche Schneeflocke

Gegeben ein Polygon. Transformiere jede Polygonkante $\overline{PQ}$ zu einer Folge von Kanten $\overline{PR}$, $\overline{RS}$, $\overline{ST}$, $\overline{TQ}$:



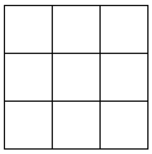
d.h., R und T dritteln die Kante $\overline{PQ}$, S ist eine 60°-Drehung des Knotens T um das Zentrum R gegen den Uhrzeigersinn.
Das so erhaltene Polygon kann erneut transformiert werden.

8.2 Fraktale Dimension

Ein selbstähnliches Objekt hat *Dimension* D, falls es in N identische Kopien unterteilt werden kann, die jeweils skaliert sind mit dem Faktor $r = \dfrac{1}{N^{\frac{1}{D}}}$.

Beispiel: Linie hat Dimension 1, denn sie besteht aus N Stücken der Größe $\frac{1}{N}$.

Quadratische Fläche hat Dimension 2, denn sie besteht aus N Flächen der Größe $\dfrac{1}{N^{\frac{1}{2}}}$, z.B. 9 Teilflächen mit Kantenlängen skaliert um $r = \dfrac{1}{9^{\frac{1}{2}}} = \dfrac{1}{3}$

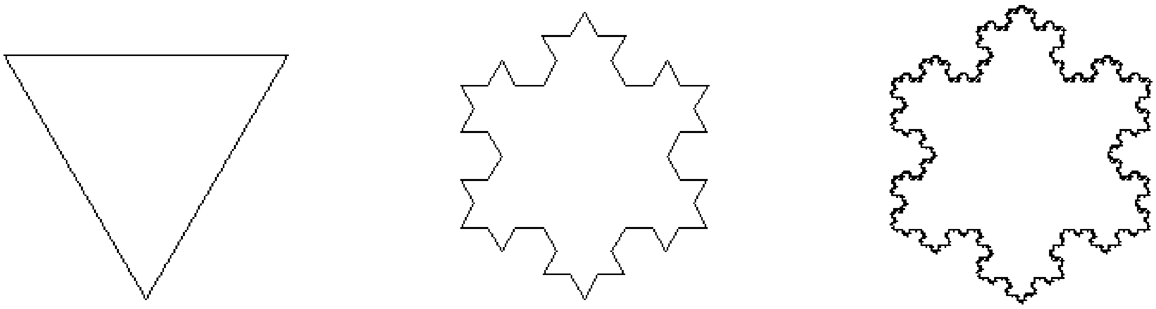


Sind N und r bekannt, läßt sich D bestimmen:

$r = \dfrac{1}{N^{\frac{1}{D}}} \Rightarrow r \cdot N^{\frac{1}{D}} = 1 \Rightarrow N^{\frac{1}{D}} = \dfrac{1}{r} \Rightarrow \dfrac{1}{D} \cdot \log(N) = \log(\dfrac{1}{r})$

$\Rightarrow D = \dfrac{\log(N)}{\log(\frac{1}{r})}$

Beispiel: Die Koch'sche Schneeflocke hat Dimension $D = \frac{\log(4)}{\log(3)} = 1.2618$, denn jeder Kantenzug besteht aus $N = 4$ Kopien, jeweils skaliert um den Faktor $r = \frac{1}{3}$.



Koch'sche Schneeflocken mit 3, 48 und 3072 Polygonpunkten

```
/***************************************************************************/
/*                                                                         */
/*                        Koch'sche Schneeflocke                           */
/*                                                                         */
/***************************************************************************/

private int koch_stern(          // wendet eine Koch-Iteration an auf
        Point poly_punkt[],      // das Polygon poly_punkt
        int poly_punkt_cnt)      // mit poly_point_cnt Punkten
                                 // neue Anzahl Punkte wird zurueckgeliefert
{
        int i,j;
        double x0,x1,x2,x3,x4,y0,y1,y2,y3,y4;
        Point[] HPunkt = new Point[MAX_POINTS];
        j=0;

        for (i=0; i<poly_point_cnt; i++)
        {
                x0 = poly_punkt[i].x;
                y0 = poly_punkt[i].y;

                x4 = poly_punkt[(i+1)%(poly_point_cnt)].x;
                y4 = poly_punkt[(i+1)%(poly_point_cnt)].y;

                x1 = (x4-x0)/3+x0;
                y1 = (y4-y0)/3+y0;

                x3 = x4 - (x4-x0)/3;
                y3 = y4 - (y4-y0)/3;

                x2 = (int)(x1+ (x3-x1)*cos_60 - (y3-y1)*sin_60);
                y2 = (int)(y1+ (y3-y1)*cos_60 + (x3-x1)*sin_60);

                HPunkt[j]   = new Point();
                HPunkt[j+1]   = new Point();
                HPunkt[j+2]   = new Point();
                HPunkt[j+3]   = new Point();

                HPunkt[j]        = poly_punkt[i];
                HPunkt[j+1].x        = (int)x1;
                HPunkt[j+1].y        = (int)y1;
                HPunkt[j+2].x        = (int)x2;
                HPunkt[j+2].y        = (int)y2;
                HPunkt[j+3].x        = (int)x3;
                HPunkt[j+3].y        = (int)y3;
                j=j+4;
        }

        for (i=0; i<j; i++)                      // neu berechnetes Polygon
                poly_punkt[i] = HPunkt[i];       // kopieren

        return j;                                // neue Anzahl Punkte zurueck
}
```

8.3 Lindenmayer-Systeme

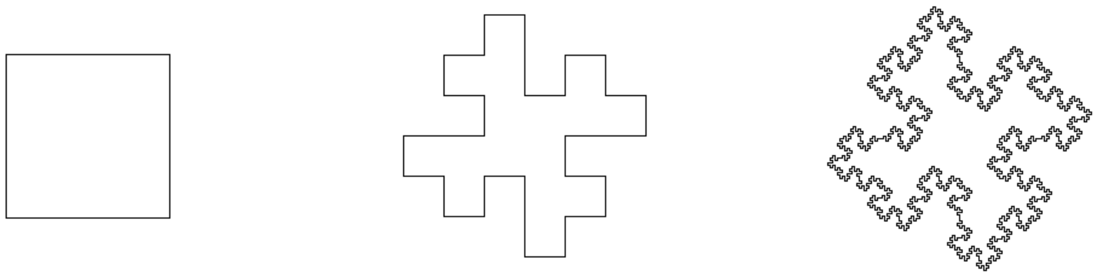
Eine nicht-grafische Beschreibung mancher Fraktale erfolgt mit Lindenmayer-Systemen, kurz L-Systems.
Ein Beispiel für ein L-System ist die quadratische Koch-Kurve:

$$\begin{array}{rlcl} \text{Alphabet } \sum = \{r, u, l, d\} & \multicolumn{3}{l}{\text{für right, up, left, down}} \\ \text{Regelmenge } f = \{ & r & \Rightarrow & r\ u\ r\ d\ d\ r\ u\ r, \\ & u & \Rightarrow & u\ l\ u\ r\ r\ u\ l\ u, \\ & l & \Rightarrow & l\ d\ l\ u\ u\ l\ d\ l, \\ & d & \Rightarrow & d\ r\ d\ l\ l\ d\ r\ d\ \} \end{array}$$

Ausgehend von einem Startwort w wird in jedem Iterationsschritt auf jedes Zeichen von w eine Regel aus f angewandt.
Sei $w = r\ u$. Dann ist $f(w) = r\ u\ r\ d\ d\ r\ u\ r\ u\ l\ u\ r\ r\ u\ l\ u$.

w $f(w)$

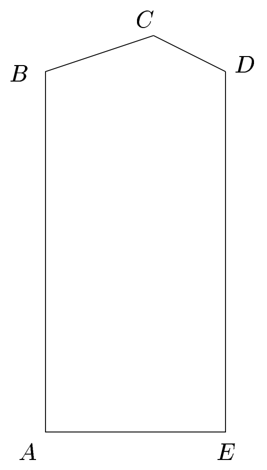
Ist nach n Iterationen das Wort $f^n(w)$ entstanden, so kann es, zusammen mit dem Parameter n (erforderlich für die Schrittweite), einer Ausgabeprozedur übergeben werden.
Die quadratische Koch-Kurve hat die Dimension $D = \frac{\log(8)}{\log(4)} = 1.5$, denn jeder Kantenzug besteht aus $N = 8$ Kopien, jeweils skaliert um den Faktor $r = \frac{1}{4}$.



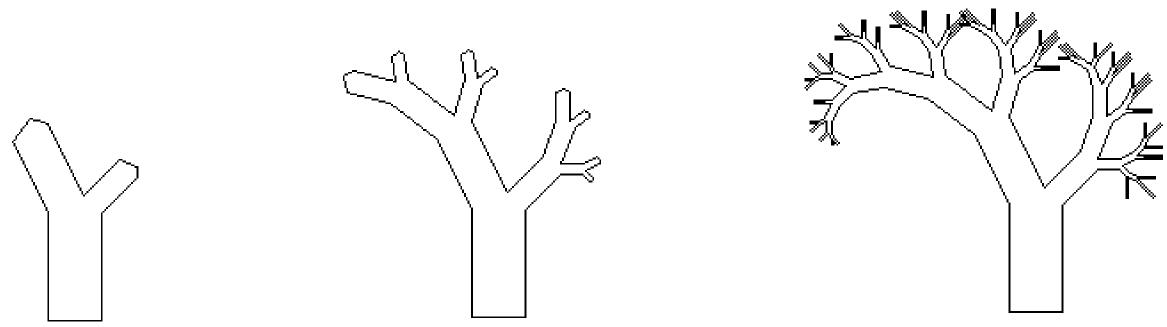
L-System mit 4, 32 und 16384 Polygonpunkten

8.4 Baumstrukturen

Gegeben ein "Ast", repräsentiert durch ein 5-seitiges Polygon:



An den Kanten $\overline{BC}$ bzw. $\overline{CD}$ können nun weitere "Äste" als Polygone angesetzt werden, die bzgl. des gegebenen Polygons mit 0.75 bzw. 0.47 skaliert sind. Durch wiederholtes Anstückeln entsteht ein Baum mit immer feinerer Verästelung. Statt dieses deterministischen Verfahrens kann man z.B. die Lage des Punktes C von einem Zufallsparameter beeinflussen lassen.



Vom Baum-Algorithmus erzeugter Baum mit 11, 47 bzw. 3071 Polygonpunkten

```
/****************************************************************************/
/*                                                                          */
/*                            Baum                                          */
/*                                                                          */
/****************************************************************************/

public void baum()             // wendet eine Iteration auf die Ansatzstuecke an
        //Point ansatz[],      // alte und neue Folge von Ansatzstuecken
        //int   anzahl,        // Anzahl der Anzsatzstueckendpunkte
        //Point poly_punkt[],  // alte und neue Folge von Polygonecken
        //int   poly_point_cnt // Anzahl der Polygonecken
{
        int i,j,k,p;
        double a,b,l,si_de,co_de;
        Point P;
        Point Q;
        Point A;
        Point B;
        Point C;
        Point[] ast = new Point[3];
        Point[] Hpunkt = new Point[MAX_POINTS];
        Point[] Hansatz = new Point[MAX_POINTS];

        DoublePoint[] schema = new DoublePoint[3];

        ast[0] = new Point();
        ast[1] = new Point();
        ast[2] = new Point();

        schema[0] = new DoublePoint();
        schema[1] = new DoublePoint();
        schema[2] = new DoublePoint();

        schema[0].x = 0.0;                    // generiere Schema
        schema[0].y = 2.0;
        schema[1].x = 2.0/3.0;
        schema[1].y = 7.0/3.0;
        schema[2].x = 1.0;
        schema[2].y = 2.0;

        j=0;                                  // Index fuer neue Ansatzstuecke
        k=0;                                  // Index fuer neue Polygonpunkte
        for (p=0; p<anzahl; p=p+2)
        {
                P = ansatz[p];
                Q = ansatz[p+1];

                a     = P.y - Q.y;            // bestimme l, sinus, cosinus
                b     = Q.x - P.x;
                l     = Math.sqrt(a*a + b*b);
                si_de = a/l;
                co_de = b/l;
```

```
                for (i=0; i<3; i++)           // fuehre Skalierung und Rotation aus
                {
                    ast[i].x = P.x + (int)(l*schema[i].x*co_de-l*schema[i].y*si_de);
                    ast[i].y = P.y - (int)(l*schema[i].y*co_de+l*schema[i].x*si_de);
                }

                A = ast[0];                   // kreiere neue Ast-Punkte
                B = ast[1];
                C = ast[2];

                if (anzahl>2) del_line(P, Q); // loesche Ansatzstueck

                set_line(P,A);                // zeichne Figur P,A,B,C,Q
                set_line(A,B);
                set_line(B,C);
                set_line(C,Q);
                                              // speichere neue Ansatzstuecke
                Hansatz[j]         = new Point(A);
                Hansatz[j+1]       = new Point(B);
                Hansatz[j+2]       = new Point(B);
                Hansatz[j+3]       = new Point(C);
                j=j+4;
        }

        i=0;                                  // Index fuer altes Polygon
        k=0;                                  // Index fuer neues Polygon
        for (p=0; p<anzahl; p=p+2)            // Index fuer alte Ansatzstuecke
        {
                while  ((poly_punkt[i].x !=ansatz[p].x) ||
                        (poly_punkt[i].y !=ansatz[p].y))
                                              // uebernimm unveraenderte Polygonpkte
                    Hpunkt[k++] = poly_punkt[i++];

                Hpunkt[k]   = ansatz[p];      // uebernimm neue Polygonpunke
                Hpunkt[k+1] = Hansatz[2*p];
                Hpunkt[k+2] = Hansatz[2*p+1];
                Hpunkt[k+3] = Hansatz[2*p+3];
                Hpunkt[k+4] = ansatz[p+1];
                k+=4;
                i++;
        }

        while (i<poly_point_cnt) Hpunkt[k++]=poly_punkt[i++];

        for (i=0; i<j; i++)
                ansatz[i] = Hansatz[i];       // neu berechnete Ansatzstuecke kopieren
        anzahl=j;

        for (i=0; i<k; i++)
                poly_punkt[i] = Hpunkt[i];    // neu berechnete Polygonpunkte kopieren
        poly_point_cnt=k;
}
```

8.5 Mandelbrot-Menge

Sei z eine komplexe Zahl mit Realteil $z.re$ und Imaginärteil $z.im$:

```
public class Komplex
{
        double re;
        double im;

...
```

Das Quadrat einer komplexen Zahl z hat den Realteil $z.re^2 - z.im^2$ und den Imaginärteil $2 \cdot z.re \cdot z.im$. Der Betrag einer komplexen Zahl c sei $|c| = \sqrt{c.re^2 + c.im^2}$.
Betrachte die Funktion $f : \mathbb{C} \to \mathbb{C}, f(z) = z^2 + c$ für festes c.

```
Komplex f(Komplex z)
{
        double tmp;

        tmp   = z.re;
        z.re  = z.re * z.re - z.im * z.im + c.re;
        z.im  = 2 * tmp * z.im + c.im;

        return z
}
```

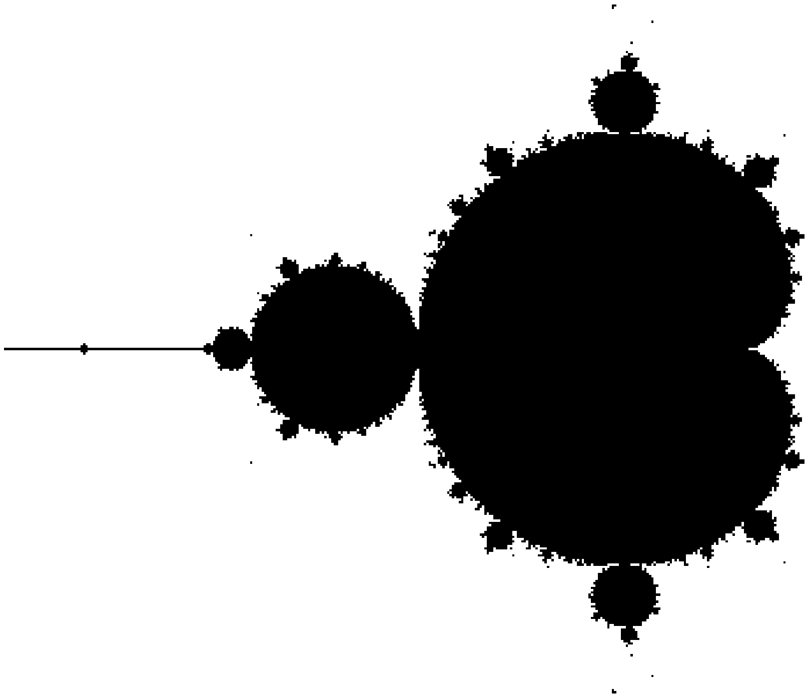
Betrachte die Folge $z, f(z), f^2(z), f^3(z), \ldots$ beginnend bei $z = 0$ für festes c.
Für manche c wächst der Betrag beliebig, für manche c konvergiert er zu einem Fixpunkt. Die *Mandelbrotmenge* besteht aus solchen komplexen Zahlen c, die beim Startwert $z = 0$ zur Konvergenz führen.
Um die Mandelbrotmenge grafisch darzustellen, definiert man

$$\text{farbe}(c) := \begin{cases} \text{schwarz}, & \text{falls die zu } c \text{ gehörende Folge konvergiert} \\ \text{weiß}, & \text{falls die zu } c \text{ gehörende Folge nicht konvergiert} \end{cases}$$

Ordne jedem Pixel $(p.x, p.y)$ des Bildschirms eine komplexe Zahl zu wie folgt. Die linke obere Ecke bezeichne die komplexe Zahl `start` (z.B. $-2.15, 2.15$). Die rechte obere Ecke bezeichne die komplexe Zahl `ende` (z.B. $0.85, 2.15$)
Dann ergibt sich bei 300 Pixeln pro Zeile eine Schrittweite von $(0.85 + 2.15)/300 = 0.01$.
Somit läßt sich ein Koordinatenpaar $(p.x, p.y)$ umrechnen durch den folgenden Konstruktor.

```
public Komplex(Point p, Komplex start, double schritt)
// bildet die komplexe Zahl zum Pixel p mit linker/oberer Ecke start
// und Schrittweite schritt
{
        this.re = start.re + schritt * (double)p.x;
        this.im = start.im - schritt * (double)p.y;
}
```

Sei $(p.x, p.y)$ das zur komplexen Zahl c gehörende Pixel. Dann färbe es mit farbe(c).



Mandelbrotmenge: $-2.2 \leq z.re \leq 0.6$, $-1.2 \leq z.im \leq 1.2$, 100 Iterationen

Implementation der Mandelbrot-Menge:

Sobald während des Iterierens der Betrag von $z > 2 \Rightarrow$ Folge konvergiert definitiv nicht.
Falls Betrag von z nach z.B. 100 Iterationen noch $< 2 \Rightarrow$ Folge konvergiert vermutlich.
D.h., bei Erreichen einer vorgegebenen Iterationenzahl wird das Pixel, welches der komplexen Zahl c entspricht, schwarz gefärbt, da c vermutlich zur Konvergenz führt.

```
void mandel ()
{
     Point p;
     int zaehler;
     Komplex c, z;

     for (p.x = 0; p.x < WIDTH;  p.x++)
     for (p.y = 0; p.y < HEIGHT; p.y++)
     {
          zaehler = 0;
          c       = new Komplex(p, start, schritt);
          z       = new Komplex(c);

          while ((betrag (z) < 2.0) && (zaehler++ < max_iter))

             z = f(z);

          if (betrag(z) < 2.0) set_pixel(p);
      }
}
```

Bemerkung: Bei Erhöhung der Iterationszahl können einige Pixel weiß werden, da sich herausgestellt hat, daß durch weitere Iterationen die Betragsgrenze 2 überschritten wird. Um die Zahlen c, die nicht zur Konvergenz geführt haben, weiter zu klassifizieren, setzt man

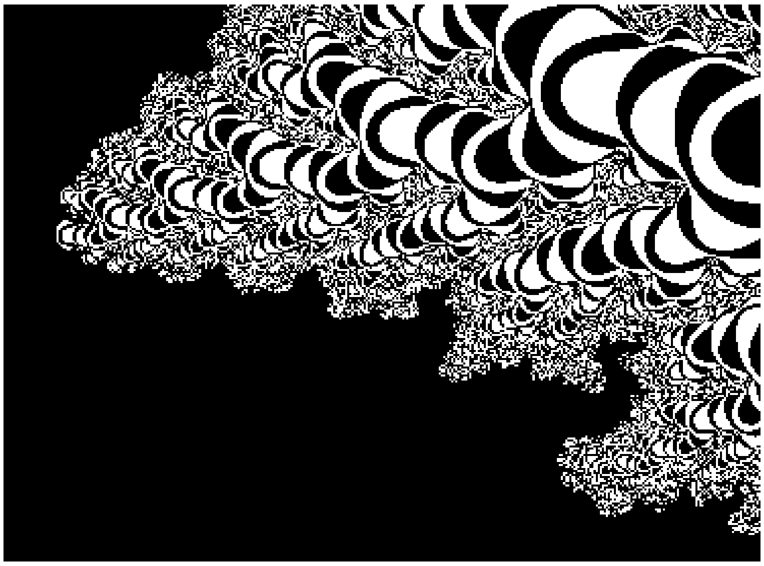
$$\text{farbe}(c) := \begin{cases} \text{schwarz}, & \text{falls konvergiert} \\ \text{weiß}, & \text{falls Iterationszahl nach Überschreiten von Betrag 2 gerade} \\ \text{schwarz}, & \text{falls Iterationszahl nach Überschreiten von Betrag 2 ungerade} \end{cases}$$

Also ergibt sich:

```
if (betrag < 2.0)        set_pixel (p); else
if ((zaehler %  2) != 0) set_pixel (p);
```

Bei Farbbildschirmen gibt es folgende Möglichkeit, die Divergenzgeschwindigkeit der Folge für ein c zu veranschaulichen:
Es seien die Farben `farbe[0], ..., farbe[NUM_COL-1]` vorhanden

```
set_col_pixel (p, farbe [zaehler % NUM_COL])
```



Vom Mandelbrot-Algorithmus erzeugtes Bild mit Modulo 2 und 100 Iterationen
$-1.2548 \leq Re \leq -1.2544$, $0.3816 \leq Im \leq 0.3822$

```
/***************************************************************************/
/*                      Mandelbrotmenge                                    */
/***************************************************************************/
void  mandel(                     // berechnet und zeichnet Mandelbrotmenge ab
        Komplex start,            // linker oberer Ecke start der komplexen Flaeche
        double  schritt,          // mit Schrittweite schritt und
        int     max_iter,         // Iterationsanzahl max_iter (muss gerade sein)
        int     fall)             // gemaess Fall fall
                                  // Abbruch erfolgt, wenn der Betrag groesser als 2 ist
                                  // oder wenn Iterationszahl erreicht ist
{
    Point p = new Point();
    int zaehler ;
    Komplex c;
    Komplex z;
    double tmp;

    for (p.x = 0; p.x < WIDTH;  p.x++)
    for (p.y = 0; p.y < HEIGHT; p.y++)
    {
        c = new Komplex(p, start, schritt);
        z = new Komplex(0.0, 0.0);
        zaehler = 0;

        while( (q_betrag(z) < 4.0)  && (zaehler++ < max_iter) ) {    // berechne f(z)
                tmp    = z.re;
                z.re   = z.re * z.re - z.im * z.im + c.re;
                z.im   = 2 * tmp * z.im + c.im;
        }

        switch (fall) {

        case 1: if (q_betrag(z) < 4.0) set_pixel(p); break;          // schwarz

        case 2: if (q_betrag(z) < 4.0) set_pixel(p); else            // schwarz
                if ((zaehler %2) != 0) set_pixel(p); break;     // mit modulo

        case 3: if(zaehler == max_iter+1)
                set_col_pixel(p,farbe[0]); else                 // Mandelbrotmenge selbst
                set_col_pixel(p,farbe[zaehler%NUM_COL]); break; // farbe modulo
        }
    }
}
```

8.6 Julia-Menge

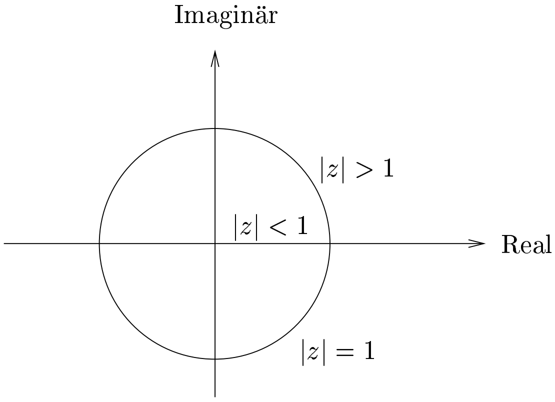
Gegeben eine komplexe Funktion $f : \mathbb{C} \to \mathbb{C}$, z.B. $f(z) = z^2$. Wähle Startwert z_0. Betrachte die Folge $z_0, {z_0}^2, z_0^4, z_0^8, \ldots$. Es gilt:

a) für $|z_0| < 1$ werden die erzeugten Zahlen immer kleiner und konvergieren zum Nullpunkt,

b) für $|z_0| > 1$ werden die erzeugten Zahlen immer größer und laufen gegen unendlich,

c) für $|z_0| = 1$ bleiben die Zahlen auf dem Einheitskreis um den Ursprung, der die Gebiete a) und b) trennt.

Die Menge c) wird *Julia-Menge* genannt. Sie ist invariant bzgl. der komplexen Funktion $f(z)$ und punktsymmetrisch zum Ursprung.



Julia-Kurve für $f(z) = z^2$

Es gilt: Julia-Menge für $f(z) = z^2 + c$ ist genau dann zusammenhängend, wenn c in der Mandelbrotmenge für $f(z)$ liegt.

Implementation der Julia-Menge:

1. Möglichkeit: Sei c gegeben. Jeder Wert z der komplexen Fläche wird überprüft, ob er als Startwert zur Konvergenz führt:

```
void  julia_voll(         // berechnet Julia-Menge als gefuelltes Gebiet
        Komplex c,        // fuer komplexes c,
        Komplex start,    // die linke obere Ecke start der komplexen Flaeche,
        double schritt,   // Schrittweite schritt
        int max_iter)     // mit vorgegebener Iterationszahl max_iter
{
        Point p = new Point();
        Point q = new Point();          // Symmetrischer Punkt
        int zaehler ;
        Komplex z;

        for (p.x = 0; p.x <= WIDTH; p.x++)
        for (p.y = 0; p.y <= HEIGHT/2;  p.y++)
        // Wegen Symmetrie nur bis zur Haelfte laufen
```

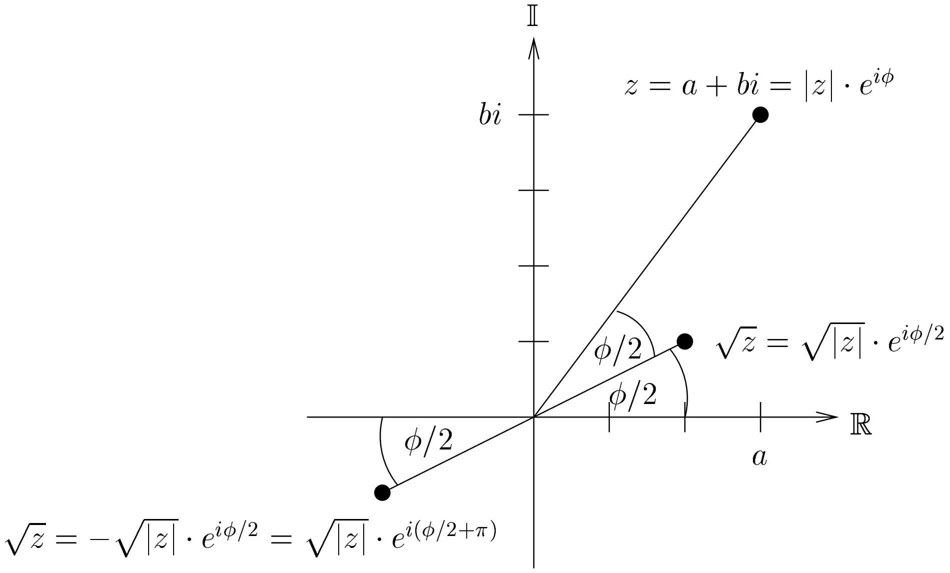
```
        {
                z = new Komplex(p, start, schritt);

                q.x = WIDTH  - p.x;     // Symmetrie nutzen
                q.y = HEIGHT - p.y;

                zaehler = 0;
                while ((q_betrag(z) < 4.0) && (zaehler++ < max_iter))
                {
                        z = f(z,c);
                }

                if (q_betrag(z) < 4.0)
                {
                        set_pixel(p);
                        set_pixel(q);
                }
        }
}
```

Es gilt: Der Rand der berechneten Fläche stellt die Julia-Menge für c dar.

2. Möglichkeit: *Inverse Iteration Method*:
Aus $f(z) = z^2 + c \Rightarrow f^{-1}(z) = \pm\sqrt{z-c}$

Sei $z = a + b \cdot i$ eine komplexe Zahl. Es gilt:

$$\sqrt{z} = \begin{cases} \pm\left(\sqrt{\frac{|z|+a}{2}} + i \cdot \sqrt{\frac{|z|-a}{2}}\right) & \text{falls } b \geq 0, \\ \pm\left(\sqrt{\frac{|z|+a}{2}} - i \cdot \sqrt{\frac{|z|-a}{2}}\right) & \text{falls } b < 0 \end{cases}$$



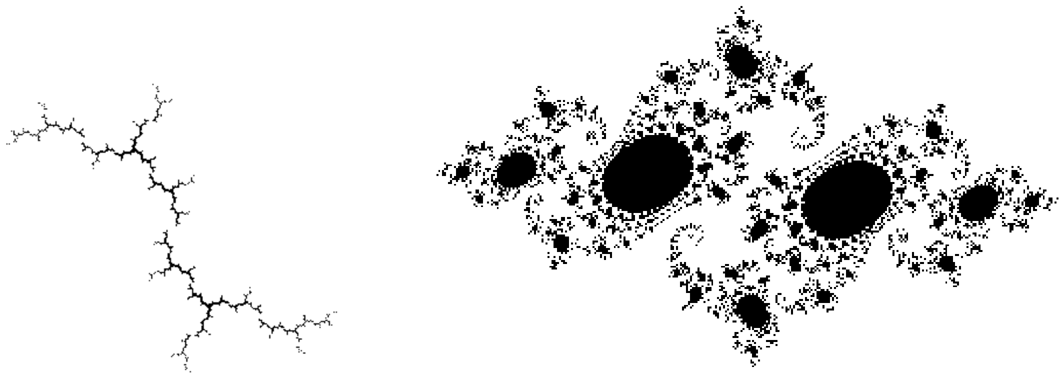
Realisiere f^{-1} durch

```
Komplex backward_random (Komplex z, Komplex c)
{
     /* z = z - c */
     /* bestimme r mit r * r = z */
     /* wuerfel das Vorzeichen von r */
     return (r);
}
```

Die Wahl des Vorzeichens bei dieser inversen Iteration erfolgt zufällig.

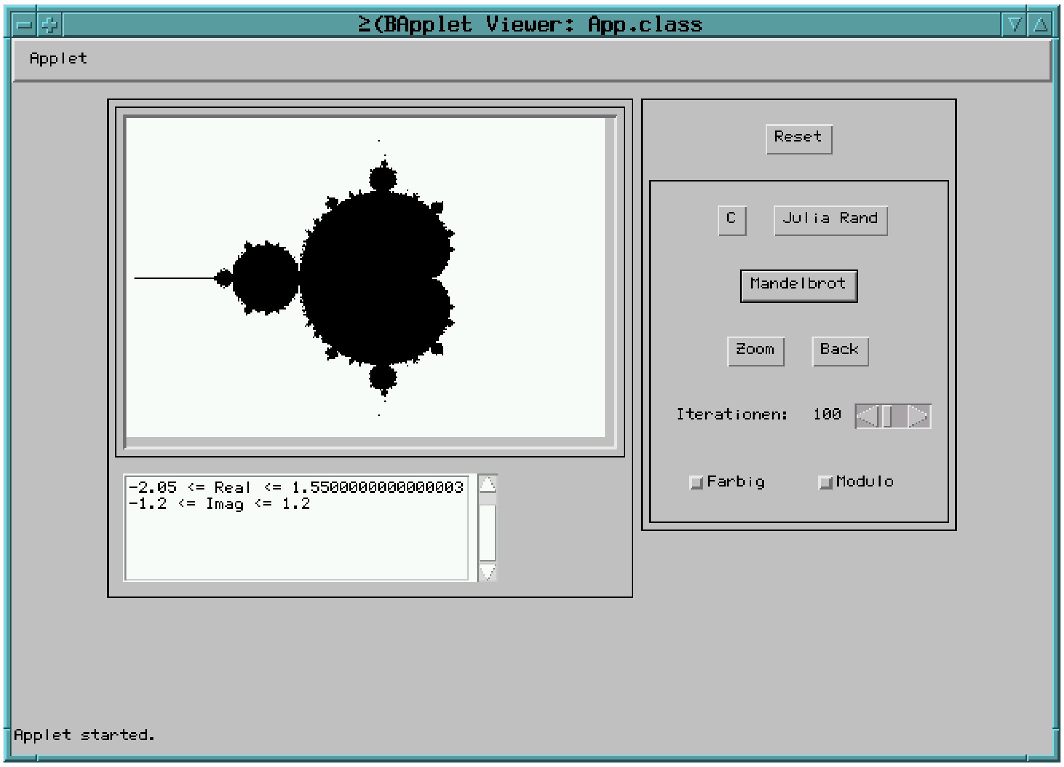
Ausgehend von $z = 1+0i$ werden zunächst 50 Iterationen $z := f^{-1}(z)$ durchgeführt, um die berechneten Punkte nahe genug an die Julia-Kurve heranzuführen. Iteriere danach weiter $z := f^{-1}(z)$ und zeige jeweils die berechneten z:

```
void julia_rueck(          // berechnet Julia-Menge
        Komplex c,         // fuer komplexes c
        Komplex start,     // linke obere Ecke der komplexen Flaeche
        double schritt,    // Schrittweite
        int anzahl)        // durch anzahl Rueckwaertsspruenge
{
        int k;
        Komplex z = new Komplex(1.0, 0.0);   // Startposition

        for (k=0; k<50; k++)        // tastet sich an die Julia-Kurve heran
            z = backward_random(z,c);


        for (k=0; k < anzahl; k++) {        // bestimme anzahl Vorgaenger
            z = backward_random(z,c);
            set_pixel(z.getPoint(start, schritt)); // und zeichne sie
        }

}
```



Vom Julia-Algorithmus erzeugte Bilder mit $c = i$ und $c = 0.746 + 0.112i$

8.7 Beispiel-Applet zu Fraktalen

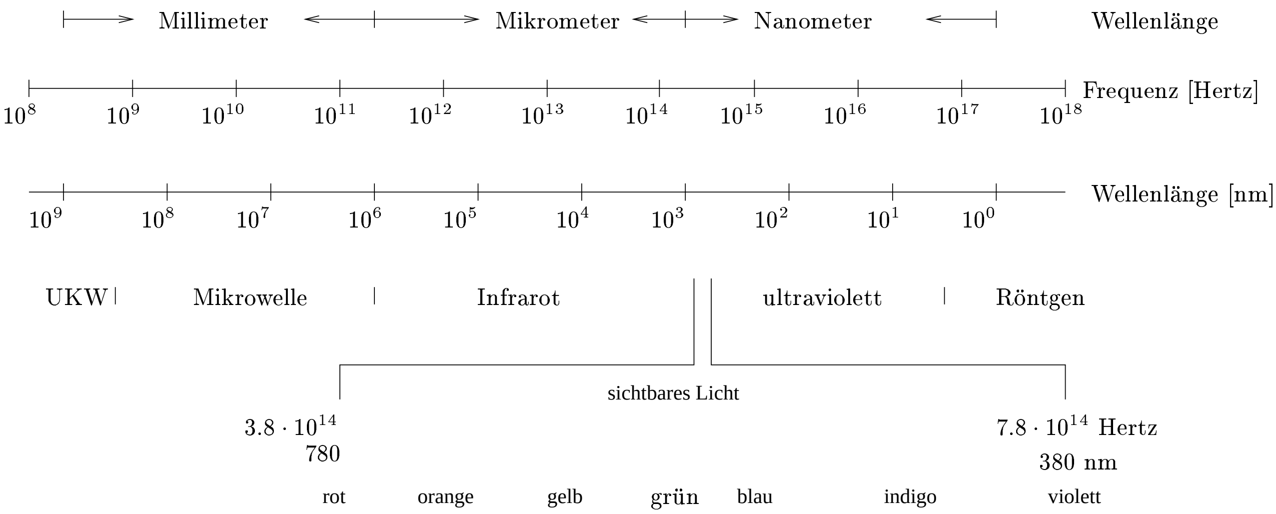


Screenshot vom Fraktale-Applet

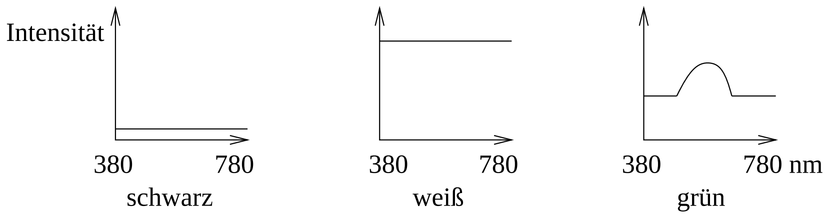
Kapitel 9

Farbe

Ein Teil des elektromagnetischen Spektrums wird vom Auge wahrgenommen:

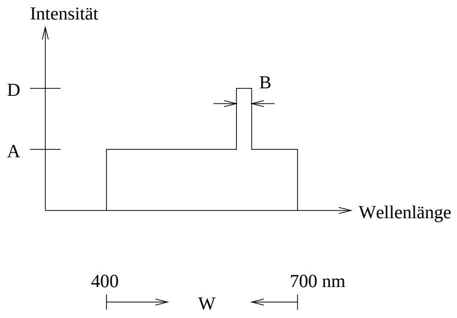


Es gilt: Wellenlänge $\cdot$ Frequenz = Lichtgeschwindigkeit ($= 2.998 \cdot 10^8$ m/s).
Spektralfarben bestehen aus Licht einer einzigen Wellenlänge.
In der Natur vorkommende Farben bestehen aus Licht, das aus verschiedenen Wellenlängen zusammengesetzt ist. Die Verteilung der Wellenlängen bezeichnet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



Viele verschiedene Spektren verursachen beim Betrachter denselben Farbeindruck.
Frage: Wie läßt sich eine Farbe numerisch beschreiben?

9.1 Dominante Wellenlänge



Das Licht wird charakterisiert als “normiertes” Spektrum.

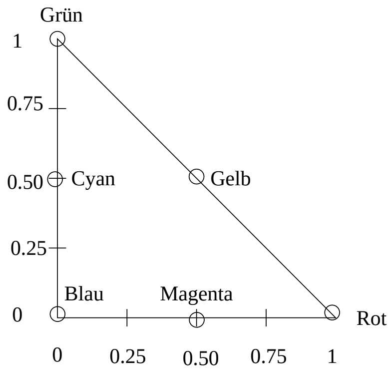
hue	=	Farbton, gegeben durch dominante Wellenlänge.
luminance	=	Helligkeit, gegeben durch die Gesamtfläche unter der Kurve: $L = (D - A) \cdot B + A \cdot W$
saturation	=	Sättigung, Reinheit, gegeben durch das Verhältnis der Fläche im “Turm” zur Gesamtfläche = $\frac{(D-A)\cdot B}{L}$.

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.

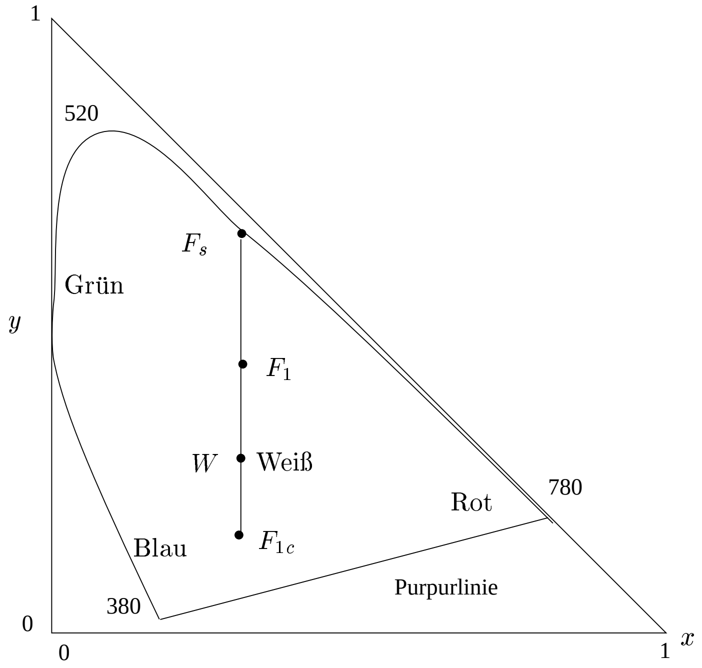
Bemerkung: Der Mensch kann etwa 128 reine Farbtöne unterscheiden. Pro Farbton können etwa 20 Sättigungsgrade unterschieden werden.

9.2 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 beschreiben (da z.B. B aus R und G folgt).



Im CIE-Standard von 1931 wurden drei (künstliche) Grundfarben festgelegt, die alle sichtbaren Farben erzeugen können.



CIE-Farbdiagramm des sichtbaren Spektrums
(Auf der Kurve liegen die Spektralfarben mit der Wellenlänge in nm.)

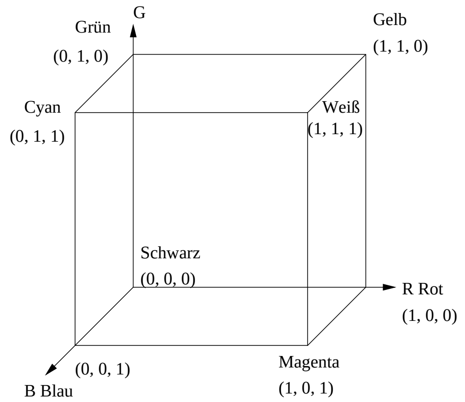
Die Grundfarben eines typischen Farbbildschirms haben die (x, y)-Koordinaten

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

Die Reinheit der Farbe F_1 ist der relative Abstand von F_1 zu W, bezogen auf die Strecke $\overline{WF_s}$, wobei F_s den Schnittpunkt der Geraden durch W und F_1 mit der Kurve bildet. F_s ist die dominante Wellenlänge in F_1.
Das Komplement F_{1c} der Farbe F_1 ergibt sich durch Spiegelung von F_1 an W mit entsprechender Skalierung.

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

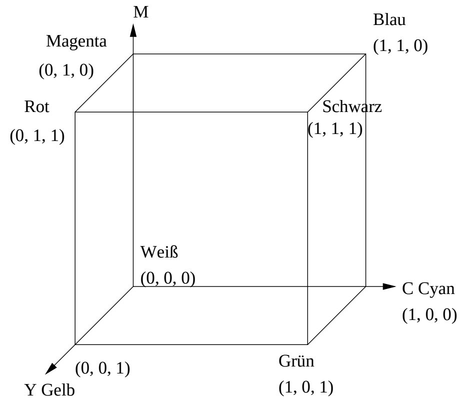
Zur Ansteuerung der dreifarbigen Phosporschicht mit roten, grünen, blauen Phosphorpunkten bietet sich das RGB-Modell an. Es handelt sich um ein additives Farbmodell, da das von den Phosphorpunkten ausgehende Licht addiert wird.
Typische Darstellung durch Einheitswürfel:



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.

9.4 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.



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.

9.5 Das 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.

9.6 YUV-Modell

Ein Farbwert wird beschrieben durch ein YUV-Tripel, wobei Y die Helligkeit (Luminanz) bezeichnet und U, V Farbdifferenzen (Chrominanz).

Die Helligkeitsempfindung resultiert wie beim YIQ-Modell zu 60 % aus dem Grünanteil, zu 30 % aus dem Rotanteil und zu 10 % aus dem Blauanteil.

Genauer:

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

In den Farbdifferenzen 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.

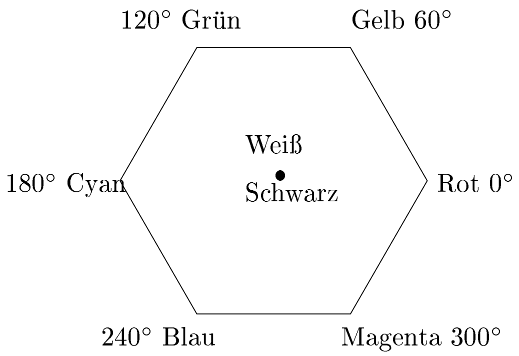
9.7 HSV-Modell

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

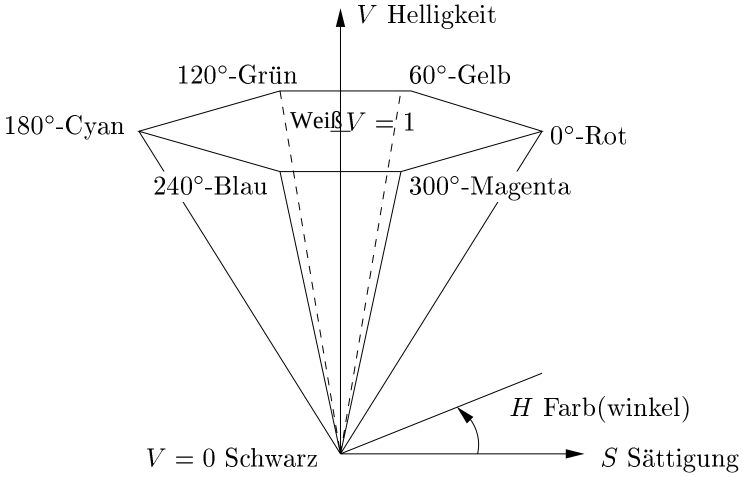
Hue	=	Farbton
Saturation	=	Sättigung
Value	=	Helligkeit

beschreibt.

Projiziere den RGB-Würfel längs der Weiß-Schwarz-Diagonale:



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



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

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

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

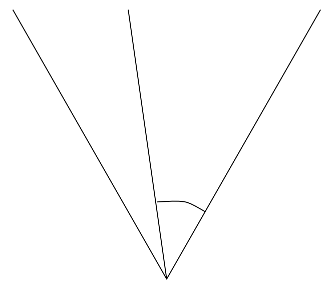
Die dominante Grundfarbe ist Grün, da $v = g$.
Am schwächsten ist Blau vertreten, da $mi = b$.
$\Rightarrow$ Farbe im Bereich Gelb ... Grün
$\Rightarrow h = 60° \ldots 120°$.

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

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°
$RGB =$	$(0,1,0)$	?	$(1,1,0)$



$$\begin{aligned} f &= \text{Winkel}/60 - \text{Winkel div } 60 \\ &= \tfrac{5}{3} - 1 = \tfrac{2}{3} \end{aligned}$$

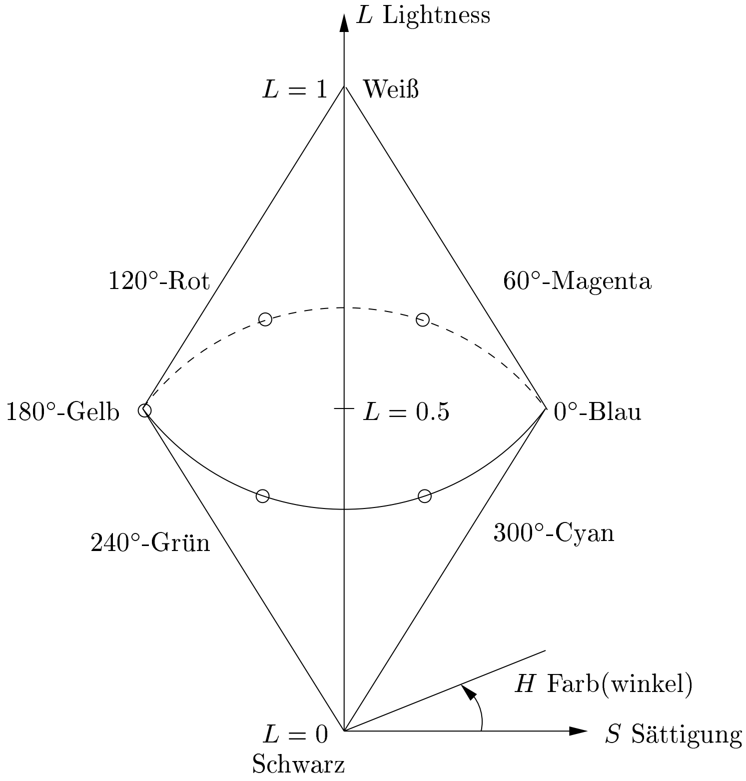
		R	G	B		R	G	B	
Farbton h	:	$1-f$	1	0	$\Rightarrow$	$\frac{1}{3}$	1	0	für $f = \frac{2}{3}$
Sättigung s	:	$1-s\cdot f$	1	$1-s$	$\Rightarrow$	$1-\frac{3}{4}\cdot\frac{2}{3}=\frac{1}{2}$	1	$\frac{1}{4}$	für $s = \frac{3}{4}$
Helligkeit v	:	$v\cdot(1-s\cdot f)$	v	$v\cdot(1-s)$	$\Rightarrow$	$\frac{1}{4}$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

9.8 HLS-Modell

Das HLS-Modell (hue, lightness, saturation) ist dem HSV-Modell ähnlich. Die Farbe (hue) wird ebenfalls als Winkel in einem Farbenkreis angegeben. Die Reihenfolge der Farben auf diesem Kreis ist zwar identisch der beim HSV-Modell, jedoch liegt beim HLS-Modell die Farbe Blau bei 0°, d.h. um 120° verdreht.

Auch der Parameter S(aturation) entspricht dem HSV-Modell und liegt ebenfalls im Bereich $[0,1]$. Der Wert $S=1$ entspricht maximaler Reinheit, und für $S=0$ erhält man die Grauskala.

Der Parameter L(ightness) entspricht in etwa dem Parameter V(alue), wobei allerdings aus der Pyramide durch "Hochziehen" des Punktes $S=0$, $V=1$ ein Doppelkegel entsteht. Die Werte $L=0$ und $L=1$ beschreiben die Farben Schwarz und Weiß, und die Farben maximaler Sättigung liegen auf der Ebene durch $L=0.5$.



HLS-Modell

Bemerkung

Die lineare Interpolation zwischen zwei Farben geschieht beim RGB-, CMY-, YIQ- und YUV-Modell längs der Verbindungsstrecke zwischen diesen Farben im dreidimensionalen Farbraum. Im allgemeinen ist dies aber beim HSV- und HLS-Modell nicht der Fall.

9.9 Color Data Base

In einer Datenbank sind Byte-Tripel abgelegt zu einer Auswahl von Farbbeschreibungen, z.B.

```
205   92   92  indian red
124  252    0  lawn green
 25   25  112  midnight blue
210  105   30  chocolate
```

9.10 CNS

Zur verbalen Beschreibung einer Farbe eignet sich das CNS-Modell (*Color Name System*).
Zur Beschreibung des Farbtons verwende *red*, *orange*, *yellow*, *green*, *blue*, *purple*. Zur Beschreibung der Sättigung verwende *grayish*, *moderate*, *strong*, *vivid*. Zur Beschreibung der Helligkeit verwende *very dark*, *dark*, *medium*, *light*, *very light*.
Die achromatische Skala besteht aus den sieben Grautönen *black*, *very dark gray*, *dark gray*, *gray*, *light gray*, *very light gray*, *white*.

9.11 Color Table (Farbtabelle)

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

Beispiel: $n = 2$, 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.

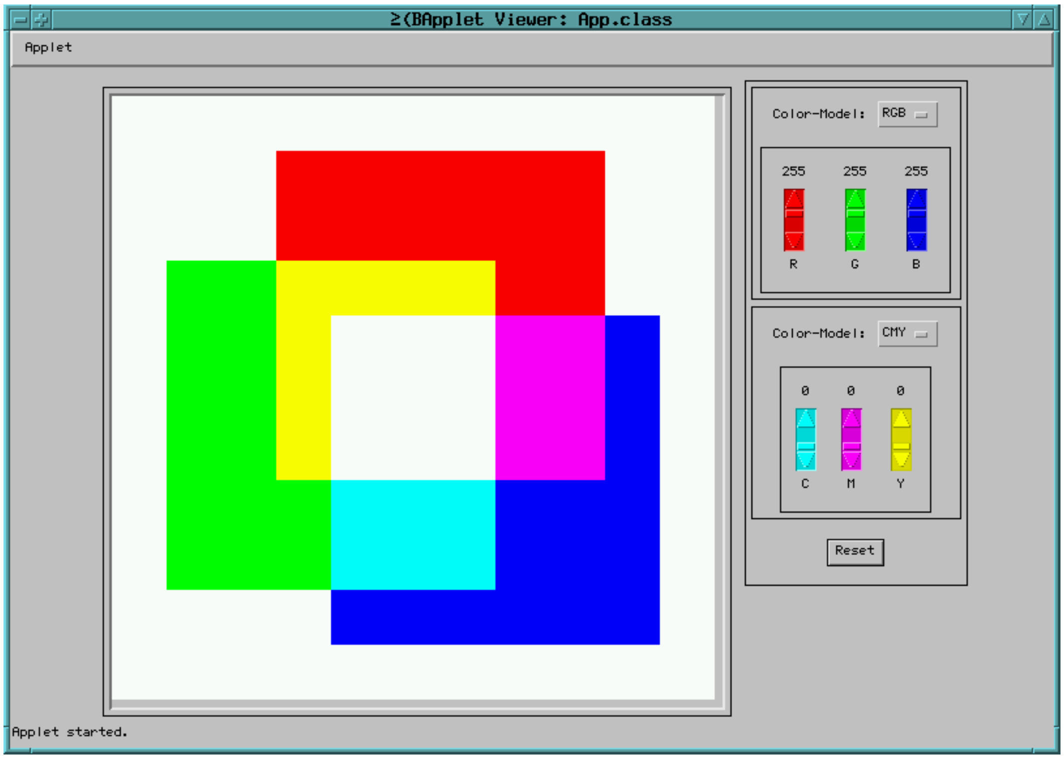
Index	R	G	B	
0000	255	255	255	
0001	255	255	255	
0010	.	.	.	
0011	.	.	.	
0100	.	.	.	
0101	.	.	.	
0110	.	.	.	
0111	255	255	255	
1000	0	0	0	Schwarz
1001	150	150	150	Grau
1010	255	0	0	Rot
1011	255	255	0	Gelb
1100	0	255	0	Grün
1101	0	255	255	Cyan
1110	0	0	255	Blau
1111	255	0	255	Magenta

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.

9.12 Beispiel-Applet zu Farbe



Screenshot vom Farben-Applet

Kapitel 10

Grafik-File-Formate

Für den Austausch von Bildinformation sind standardisierte Grafik-File-Formate sinnvoll und erforderlich. Aus der großen Menge von verschiedenen Grafik-File-Formaten werden hier einige wenige vorgestellt.

10.1 PBM, PGM, PNM 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 × 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`.

10.2 GIF

Das *Graphics Interchange Format* (GIF) wurde von CompuServe eingeführt und wird benutzt, um ggf. mehrere Bilder in einer einzigen Datei zu speichern und zwischen verschiedenen Rechnersystemen auszutauschen. Bezogen auf die Anzahl der existierenden Dateien ist GIF eines der weitverbreitetsten Formate zum Speichern von Bilddaten.
Im Gegensatz zu vielen anderen Dateiformaten basiert GIF auf einem Strom der Daten. Das Format besteht aus einer Reihe von Datenpaketen, Blöcke genannt, kombiniert mit zusätzlicher Protokollinformation.
Das GIF-Format ist in der Lage, Bilddaten mit einer Farbtiefe von 1 bis 8 Bits zu speichern. Die Bilder werden immer im RGB-Modell unter Benutzung einer Farbtabelle gespeichert.
Die Bildinformation in GIF-Dateien ist immer LZW-komprimiert. Diese von Lempel und Ziv 1977 eingeführte sowie 1984 von Welch verbesserte Datenkompression wird häufig und nicht nur für Bilddaten verwendet. Der Algorithmus reduziert Folgen von gleichen Bytewerten in ein einzelnes Codewort und erreicht auf diese Weise für typische 8-Bit-Pixel-Daten eine Reduktion der Datenmenge um 40%.
GIF gibt es in zwei Versionen: GIF87a und GIF89a.

GIF87a Format

Header
Logical Screen Descriptor
Global Color Table
Local Image Descriptor for Image 1
Local Color Table for Image 1
Image Data for Image 1
Local Image Descriptor for Image 2
Local Color Table for Image 2
Image Data for Image 2
...
Trailer

GIF89a Format

Header
Logical Screen Descriptor
Global Color Table
Comment Extension
Application Extension
Graphics Control Extension
Local Image Descriptor for Image 1
Local Color Table for Image 1
Image Data for Image 1
...
Comment Extension
Plain Text Extension
Trailer

10.3 JPEG

Die **J**oint **P**hotographic **E**xperts **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-Pixel-Matrix in der vollen Auflösung belassen und in den U- und V-Matrizen jeweils 4 Pixel mitteln (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 %.
Werden je 16 Originalpixel bzgl. der Farbwerte gemittelt, so werden für 16 Originalpixel statt 48 Bytes nur noch $16 + 1 + 1 = 18$ Bytes benötigt (pro Bildpunkt also $18 \cdot 8/16 = 9$ Bit). Die Bilddatei schrumpft auf 37.5 %.
Beim JPEG-Verfahren werden 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. Lauflängencodierung 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.

10.4 Erzeugung einer Farbtabelle

Zunächst wird für jede Farbe die Häufigkeit ihres Auftretens ermittelt. Hierdurch erhält man ein Histogramm.
Sei $d(a,b)$ der Abstand zweier Farbtupel a, b,
z.B. $d(a,b) = \sqrt{(a_{Rot} - b_{Rot})^2 + (a_{Grün} - b_{Grün})^2 + (a_{Blau} - b_{Blau})^2}$.

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

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

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 groessten Abstand hat
end
```

Median-Cut (Heckbert, MIT 1980)

```
Initialisiere RGB-Wuerfel mit Haeufigkeiten der beobachteten Farbtupel
Initialisiere Wurzel des Schnittbaums mit Gesamtzahl der Pixel

While noch_nicht_genuegend_Blaetter do
   Waehle Blatt mit der groessten Pixelzahl
   Bestimme umschliessende Box
   Bestimme Achse mit groesstem Wertebereich
   Durchlaufe Box laengs dieser Achse
   Teile am Median in zwei Haelften
   Trage Haelften als Soehne ein
end
Fuer jedes Blatt waehle den Mittelwert aller in ihm liegenden Farben.
```

Der Median-Cut-Algorithmus von Heckbert wird verwandt im Programm `ppmquant`, das Teil des PBM-Pakets ist.

Kapitel 11

3D-Grundlagen

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).

11.1 Repräsentation und Darstellung

Repräsentation

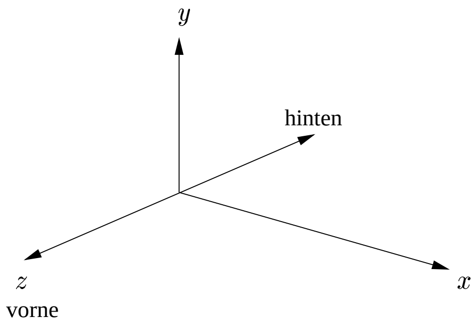
- Elementarobjekt mit Definitionspunkten,
- Drahtmodell mit Kantenliste,
- Flächenmodell mit Flächenliste,
- Flächenmodell mit Halbkantendarstellung,
- CSG (constructive solid geometry) mit mengentheoretischer Verknüpfung von Elementarobjekten.

Darstellung

- 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.

11.2 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.

11.3 Kreuzprodukt und Skalarprodukt

Def.: Gegeben sei ein 3D-Vektor

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

Seine Länge ist definiert als

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

Def.: Gegeben seien zwei 3D-Vektoren

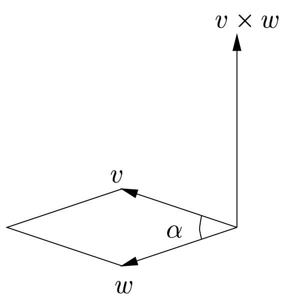
$$v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \text{ und } w = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

Das **Kreuzprodukt** von v und w ist definiert als

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

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

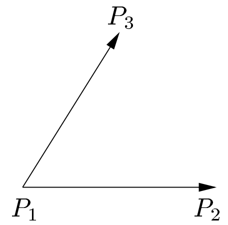
$$|v \times w| = |v| \cdot |w| \cdot \sin(\alpha)$$



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

Anwendung des Kreuzprodukts

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



Der Vektor $(P_2 - P_1) \times (P_3 - P_1)$ 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).
Ist ein Normalenvektor (A, B, C) gegeben, so errechnet sich D durch Einsetzen eines beliebigen Punktes der Ebene.
Ein Punkt (x, y, z) liegt

"oberhalb" der Ebene,	falls	$Ax + By + Cz + D > 0$
"unterhalb" der Ebene,	falls	$Ax + By + Cz + D < 0$
in der Ebene,	falls	$Ax + By + Cz + D = 0$

Def.: Gegeben seien zwei n-dimensionale Vektoren v, w.
Das **Skalarprodukt** lautet:

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

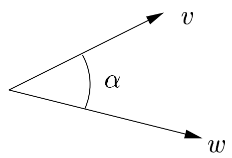
Für Vektoren a, b, c und $s \in \mathbb{R}$ gilt:

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

Anwendungen des Skalarprodukts:

Gegeben zwei Vektoren v, w.
Für den Winkel α zwischen v und w gilt:

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



$$\text{Es gilt:} \quad \begin{array}{lcl} v \cdot w < 0 & \Leftrightarrow & v \text{ und } w \text{ sind mehr als } 90^\circ \text{ auseinander} \\ v \cdot w = 0 & \Leftrightarrow & v \text{ steht senkrecht auf } w \\ v \cdot w > 0 & \Leftrightarrow & v \text{ und } w \text{ sind weniger als } 90^\circ \text{ auseinander} \end{array}$$

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

11.4 Matrixinversion

Analog zum zweidimensionalen Fall werden die dreidimensionalen Transformationen durch Verknüpfung homogener Koordinaten mit 4×4-Transformationsmatrizen dargestellt.
Sei $A = (a_{ik})$, $1 \le i, k \le 4$, 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:

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

Für die *inverse Matrix* A^{-1} gilt:

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

Für die Entwicklung der Determinante von A nach der ersten Zeile können die Adjunkten benutzt werden. Es gilt:

$$\text{det } A = \sum_{k=1}^{4} a_{1k} A_{1k}$$

Kapitel 12

3D-Repräsentation

Die Repräsentation dreidimensionaler Objekte in der Computergrafik teilt sich in verschiedene Repräsentationsklassen, die hierarchisch aufgebaut sind.

12.1 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.

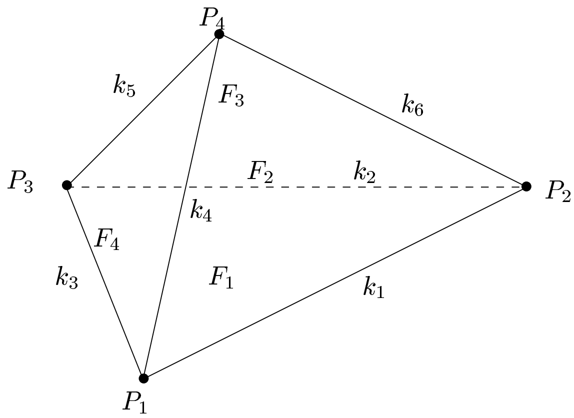
12.2 Drahtmodell

In der nächsten Repräsentationsklasse wird das Elementarobjekt als *Drahtmodell* durch eine Liste von Kanten repräsentiert. Jede Kante besteht aus zwei Punkten im kartesischen Koordinatensystem.
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. Das Drahtmodell skizziert nur die Umrisse eines Objekts und enthält keine zusätzlichen Flächen- oder Volumeninformationen.

12.3 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.
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.

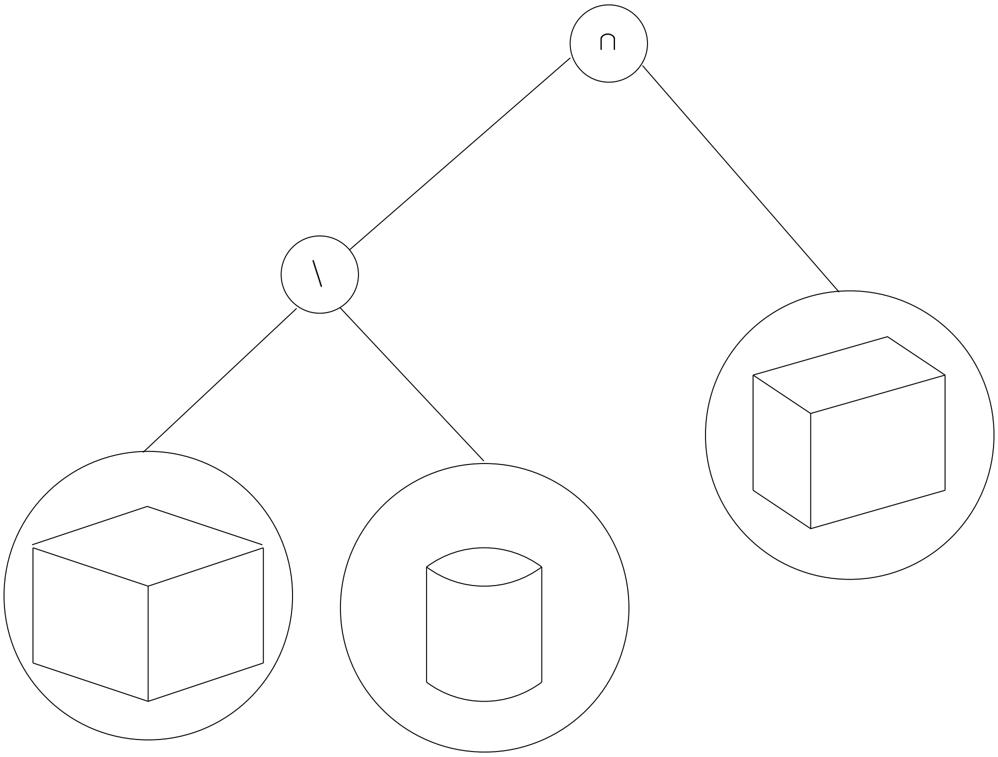
Beispiel: Tetraeder als Flächenmodell



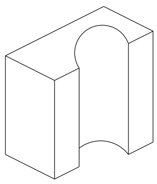
Punkteliste	Kantenliste	Flächenliste
$P_1 : (x_1, y_1, z_1)$	$k_1 : P_1, P_2$	$F_1 : k_1, k_2, k_3$
$P_2 : (x_2, y_2, z_2)$	$k_2 : P_2, P_3$	$F_2 : k_1, k_6, k_4$
$P_3 : (x_3, y_3, z_3)$	$k_3 : P_3, P_1$	$F_3 : k_2, k_6, k_5$
$P_4 : (x_4, y_4, z_4)$	$k_4 : P_1, P_4$	$F_4 : k_3, k_4, k_5$
	$k_5 : P_4, P_3$	
	$k_6 : P_4, P_2$	

12.4 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).



bezeichnet das folgende Objekt:

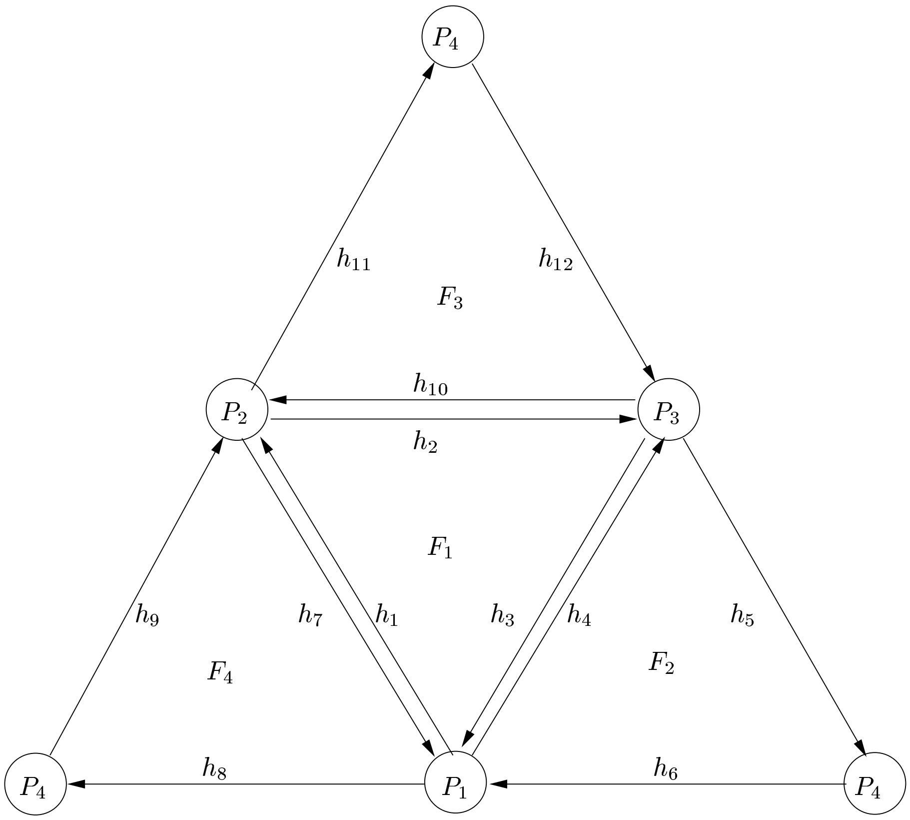


12.5 Flächenmodell mit Halbkantendarstellung

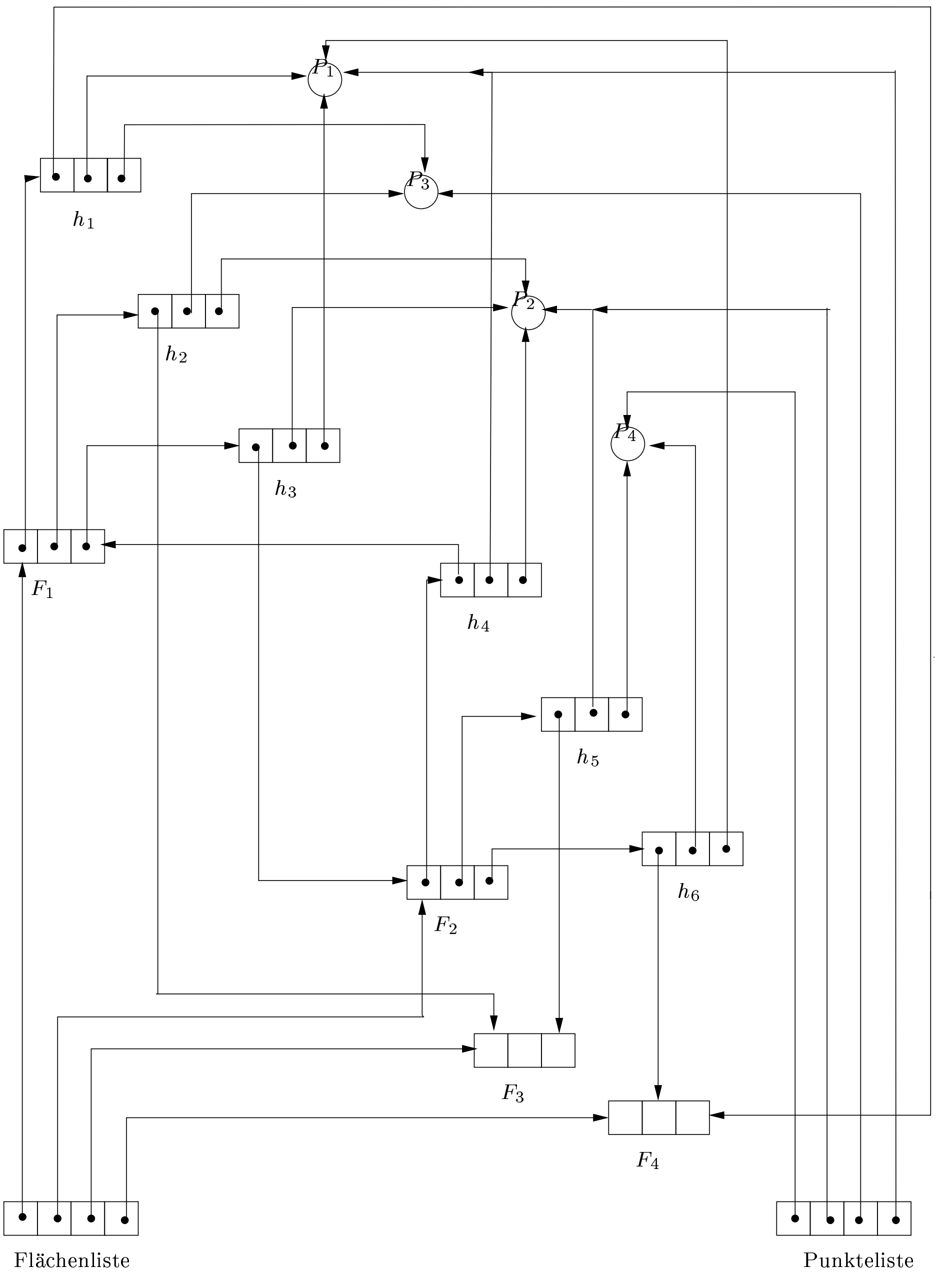
Jedes Objekt enthält eine Liste von Flächen, die von Halbkanten begrenzt werden. Die Halbkanten sind von außen betrachtet im Uhrzeigersinn orientiert und zeigen auf ihre jeweils linke Nachbarfläche sowie ihre Anfangs- und Endpunkte.

Die Halbkantendarstellung eignet sich zur effizienten Entfernung von verdeckten Kanten und Flächen. Eine Kante zwischen Punkt P_1 und Punkt P_2, welche die Flächen F_1 und F_2 trennt, taucht einmal als Halbkante (P_1, P_2) in der Kantenliste zu F_2 auf mit einem Verweis auf die Nachbarfläche F_1 und ein weiteres Mal als (P_2, P_1) in der Kantenliste zu F_1 mit einem Verweis auf die Nachbarfläche F_2. Werden nun alle Halbkanten einer Fläche F_1 bearbeitet, so regelt die Sichtbarkeit von F_1 und die Sichtbarkeit der jeweils anstoßenden Fläche die Sichtbarkeit der jeweiligen Halbkante. Das doppelte Zeichnen einer Kante läßt sich vermeiden, indem bei jeder Fläche vermerkt wird, ob ihre Halbkanten bereits bearbeitet wurden.

Auch die Flächennormalen können in der Datenstruktur gespeichert werden.



Tetraeder mit 4 Knoten, 12 Halbkanten, 4 Flächen



Objektstruktur für Tetraeder. Nicht gezeichnet sind die Halbkanten der Flächen F_3 und F_4.

Kapitel 13

3D-Transformationen

13.1 Translation

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

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

in der folgenden Form darstellen:

$$[x', y', z', 1] = [x, y, z, 1] \cdot T(t_x, t_y, t_z)$$

mit

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

13.2 Skalierung

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:

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

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:

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

13.3 Rotationen

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:

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

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:

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

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:

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

13.3.1 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

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

Die Länge dieses Vektors lautet

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

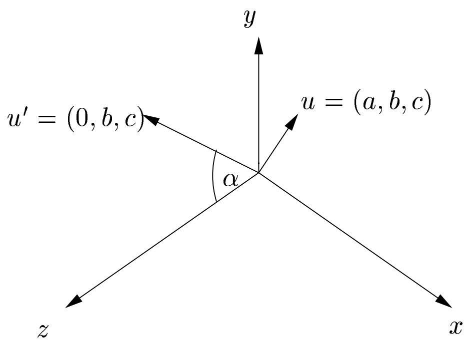
Die Komponenten des zugehörigen Einheitsvektors

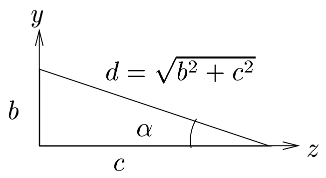
$$u = \frac{v}{|v|} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}, |u| = 1$$

lauten daher

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

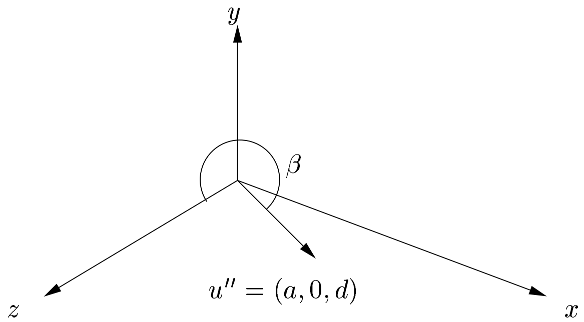
Schritt 1 läßt sich durch die Translation $T(-x_1, -y_1, -z_1)$ durchführen. Dadurch wird P_1, Ausgangspunkt des Einheitsvektors u, in den Ursprung verschoben.
Für Schritt 2 sind Sinus und Cosinus des Rotationswinkels α erforderlich, der zwischen der Projektion u' von u auf die yz-Fläche und der z-Achse, repräsentiert durch den Vektor $u_z = (0, 0, 1)$, liegt.



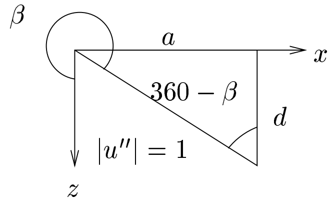


$\Rightarrow \cos(\alpha) = \frac{c}{d}$
$\Rightarrow \sin(\alpha) = \frac{b}{d}$

Nach Schritt 2 befindet sich der ursprüngliche Vektor u als u'' in der xz-Ebene:



Für Schritt 3 (Rotation um y-Achse) benötigt man Sinus und Cosinus des Rotationswinkels β.



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

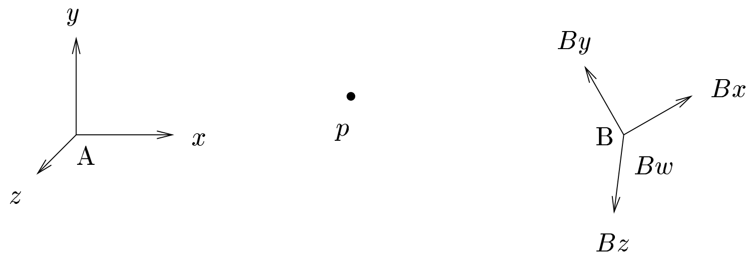
Nach den ersten drei Schritten ist die Drehachse mit der z-Achse identisch, so daß Schritt (4) mit der Rotationsmatrix $R_z(\delta)$ durchgeführt werden kann. Schritt (5) beinhaltet die Anwendung der inversen Transformationen.
Die Rotation um die Achse $v = \overline{P_1P_2}$ um den Winkel δ läßt sich daher wie folgt darstellen:

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

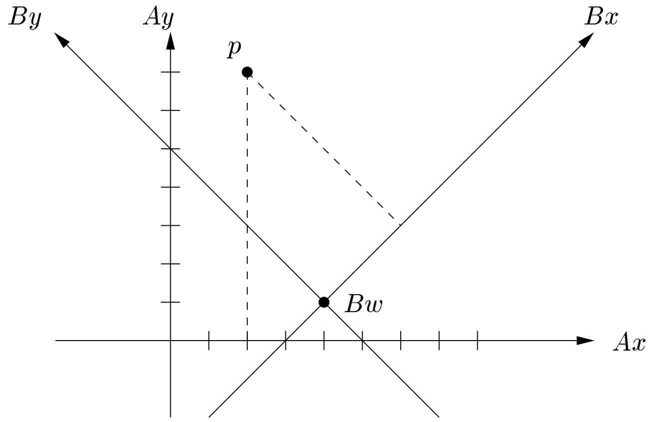
13.4 Transformation von Koordinatensystemen

Die Transformation von einem Koordinatensystem in ein anderes läßt sich folgendermaßen formalisieren: Gegeben sei ein Koordinatensystem A, in dem ein anderes Koordinatensystem B durch die homogenen Koordinaten des Ursprungs (Bw) und die drei Richtungsvektoren der Achsen (Bx, By, Bz) definiert ist. Zeilenweise angeordnet ergeben diese Vektoren die Matrix B_A, die den Übergang vom Koordinatensystem B zum Koordinatensystem A beschreibt:

$$B\_A = \begin{bmatrix} Bx \\ By \\ Bz \\ Bw \end{bmatrix}.$$



Beispiel (für den 2-dimensionalen Fall):



x-Achse:	Bx	lautet	$[\sqrt{\frac{1}{2}}, \sqrt{\frac{1}{2}}, 0]$
y-Achse:	By	lautet	$[-\sqrt{\frac{1}{2}}, \sqrt{\frac{1}{2}}, 0]$
Ursprung:	Bw	lautet	$[4, 1, 1]$
Punkt:	p_B	lautet	$[2 \cdot \sqrt{2}, 4 \cdot \sqrt{2}, 1]$

Also läßt sich der Punkt p wie folgt transformieren:

$$[2 \cdot \sqrt{2}, 4 \cdot \sqrt{2}, 1] \cdot \begin{bmatrix} \sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & 0 \\ -\sqrt{\frac{1}{2}} & \sqrt{\frac{1}{2}} & 0 \\ 4 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 2 \\ 7 \\ 1 \end{bmatrix}$$

Um einen Punkt p_A des Koordinatensystems A im Koordinatensystem B zu spezifizieren, verwendet man die inverse Matrix zu B_A :

$$p\_B = p\_A \cdot (B\_A)^{-1}.$$

13.5 Transformation der Normalenvektoren

Die Normalenvektoren müssen bei der Transformation von Objektpunkten ebenfalls abgebildet werden. Allgemein gilt, daß bei der Transformation eines Objekts mit der Matrix M ein Normalenvektor n dieses Objekts mit der transponierten Inversen von M abgebildet werden muß:

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

Beweis: Sei n der Normalenvektor einer Ebene, seien r_1 und r_2 beliebige Punkte in der Ebene. Betrachte n, r_1 und r_2 als Zeilenvektoren und arbeite mit dem Matrixprodukt (Zeile mal Spalte).

Dann gilt:

$$(r_1 - r_2) \cdot n^T = 0$$

Bei einem Wechsel des Koordinatensystems werden die Ortsvektoren r_1 und r_2 mit der Matrix M transformiert:

$$r_1' = r_1 \cdot M$$

$$r_2' = r_2 \cdot M$$

Es gilt:

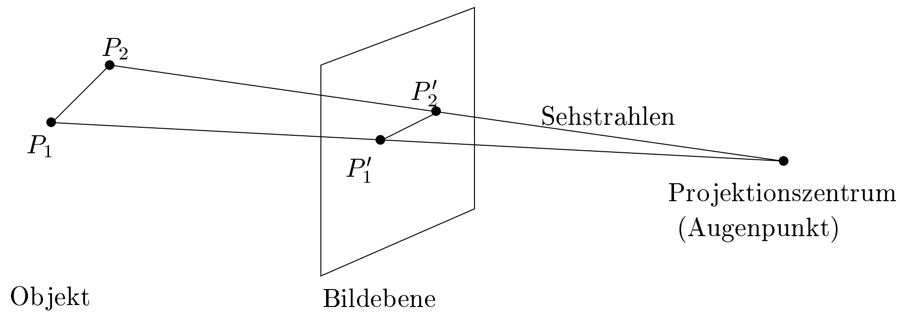
$$\begin{array}{rcl} & (r_1 - r_2) \cdot n^T & = 0 \\ \Leftrightarrow & (r_1 - r_2) \cdot (M \cdot M^{-1}) \cdot n^T & = 0 \\ \Leftrightarrow & ((r_1 - r_2) \cdot M) \cdot (M^{-1} \cdot n^T) & = 0 \\ \Leftrightarrow & (r_1' - r_2') \cdot (n \cdot (M^{-1})^T)^T & = 0 \\ \Leftrightarrow & (r_1' - r_2') \cdot (n')^T & = 0 \\ & & \\ \Rightarrow & n' = n \cdot (M^{-1})^T & \end{array}$$

Kapitel 14

Projektion

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

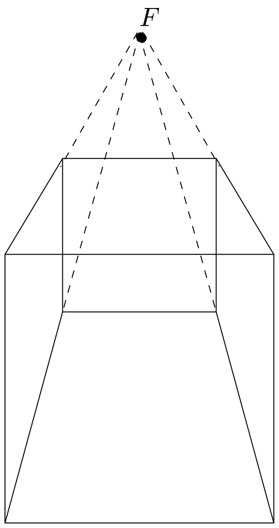
- das zu projizierende Objekt,
- die Bildebene,
- das Projektionszentrum.



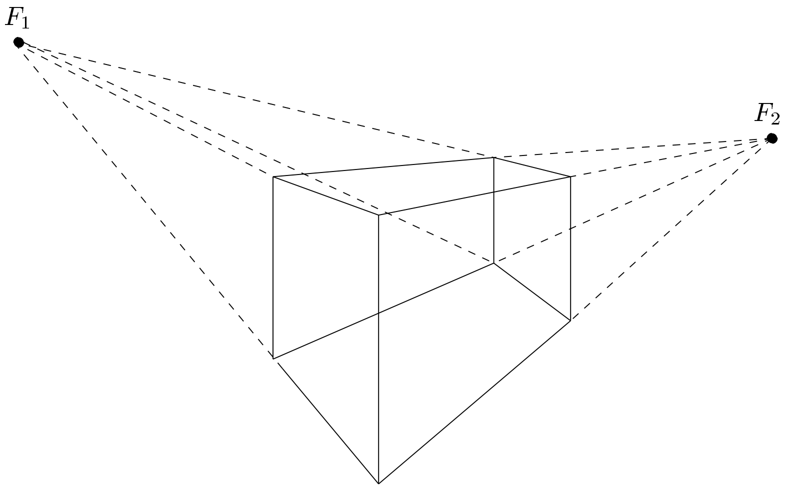
Ist der Abstand des Projektionszentrums von der Bildebene endlich, so handelt es sich um eine Zentralprojektion (perspektivische Projektion), andernfalls handelt es sich um eine Parallelprojektion (die Projektionsstrahlen sind zueinander parallel).

14.1 Perspektivische Projektion

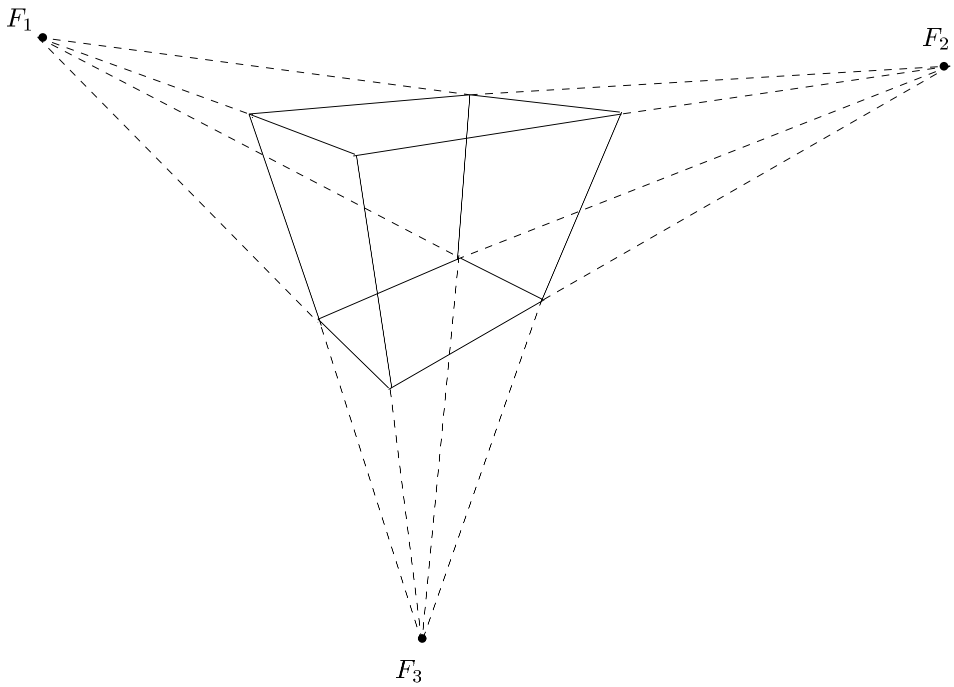
Die abgebildeten Objekte werden proportional zu ihrem Abstand zur Bildebene verkleinert. Je nachdem, ob die Bildebene eine, zwei oder drei der Koordinatenachsen schneidet, entstehen ein, zwei oder drei Fluchtpunkte.



1 Fluchtpunkt F

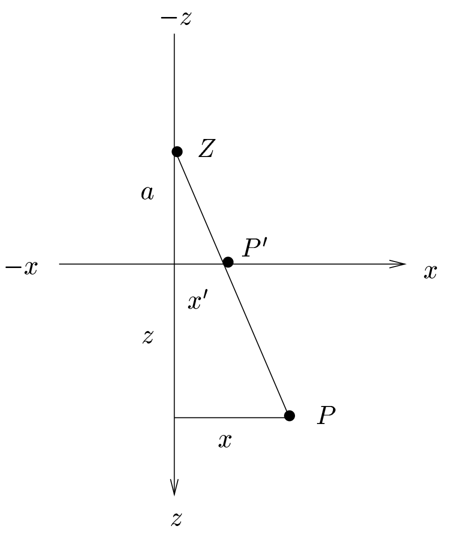


2 Fluchtpunkte F_1, F_2

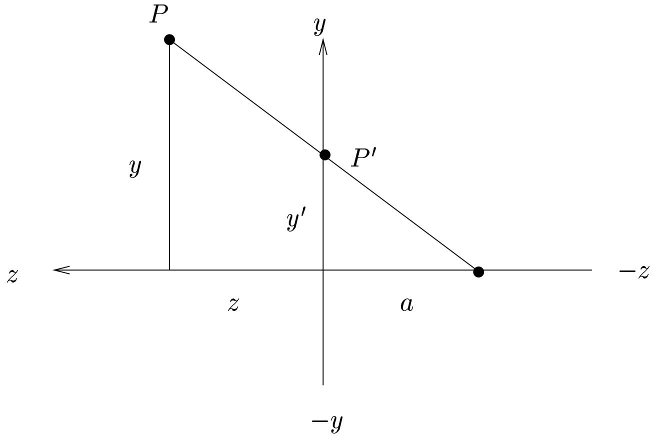


3 Fluchtpunkte F_1, F_2, F_3

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: $y' : a = y : (z + a)$

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

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

Die homogenen Koordinaten des projizierten Punktes lauten

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

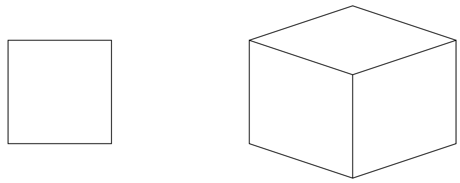
Für die Transformationsmatrix der perspektivischen Projektion ergibt sich also:

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

14.2 Parallelprojektion

14.2.1 Normalprojektionen

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



a) b)

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

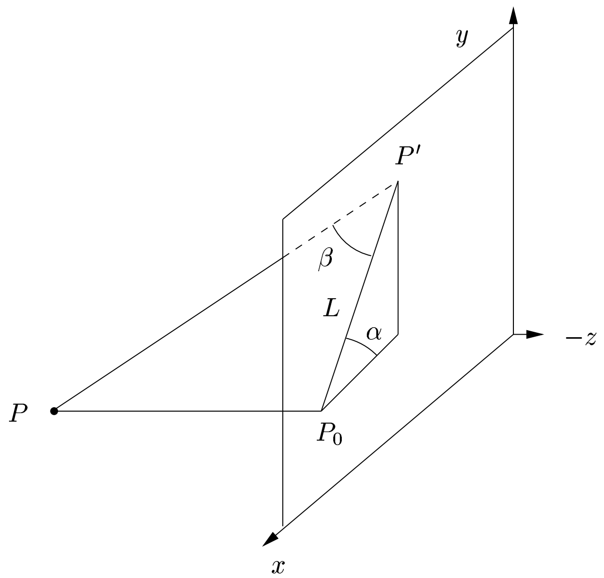
Eine weitere Form der Normalprojektionen sind die *axonometrischen Projektionen*, bei denen die Bildebene auf keiner der Hauptachsen senkrecht steht. Dadurch sind in der Abbildung mehrere zueinander normal stehende Flächen gleichzeitig sichtbar. Die resultierenden Darstellungen sind der perspektivischen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

14.2.2 Schiefe Projektionen

Bei den *schiefen* Parallelprojektionen stehen die Sehstrahlen nicht normal auf der Bildebene, sondern schneiden sie unter dem Winkel β. 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

$$x' = x - L \cdot \cos(\alpha)$$
$$y' = y + L \cdot \sin(\alpha)$$
$$z' = 0$$

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

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

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:

$$\begin{aligned}
x_1' &= x_1 - z_1 \cdot \cos(\alpha)/\tan(\beta) \\
y_1' &= y_1 + z_1 \cdot \sin(\alpha)/\tan(\beta) \\
x_2' &= x_2 - z_2 \cdot \cos(\alpha)/\tan(\beta) \\
y_2' &= y_2 + z_2 \cdot \sin(\alpha)/\tan(\beta)
\end{aligned}$$

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

$$\begin{aligned}
|x_1' - x_2'| &= |(z_1 - z_2) \cdot \cos(\alpha)/\tan(\beta)| \\
|y_1' - y_2'| &= |(z_1 - z_2) \cdot \sin(\alpha)/\tan(\beta)|
\end{aligned}$$

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

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

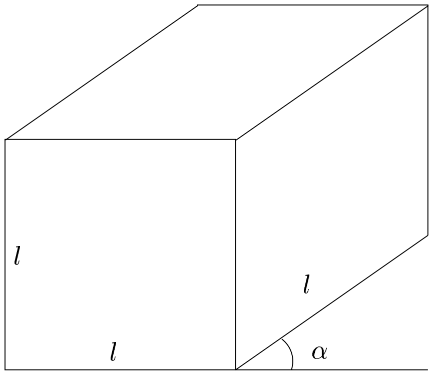
α definiert den Winkel zur Horizontalen, unter dem diese Kanten aufgetragen werden. Für die Koordinaten des so projizierten Punktes $P = (x, y, z, 1)$ gilt

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

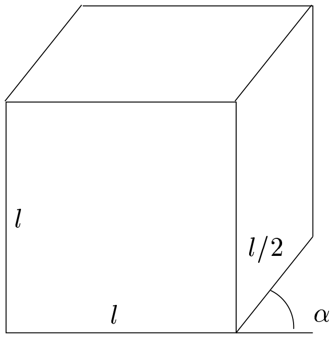
Die entsprechende Transformationsmatrix lautet

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

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 $\alpha = 50°$

Kapitel 15

Betrachtungstransformationen

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

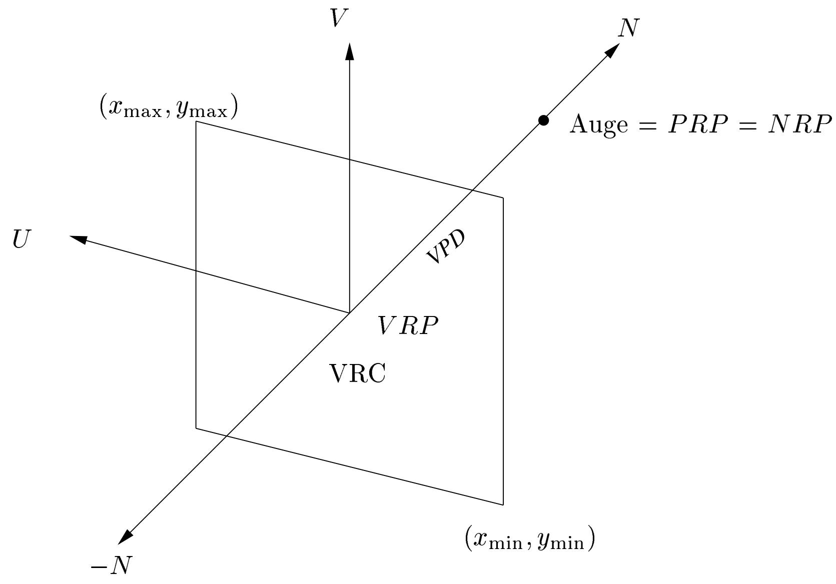
- Bilden von komplexen Szenen aus elementaren Objekten (*Modeling*),
- Festlegen der Bildschirmebene (*View Orientation*),
- Projektion auf ein normiertes Gerät (*View Mapping*),
- Abbildung auf ein Ausgabegerät (*Device Mapping*).

15.1 Die synthetische Kamera

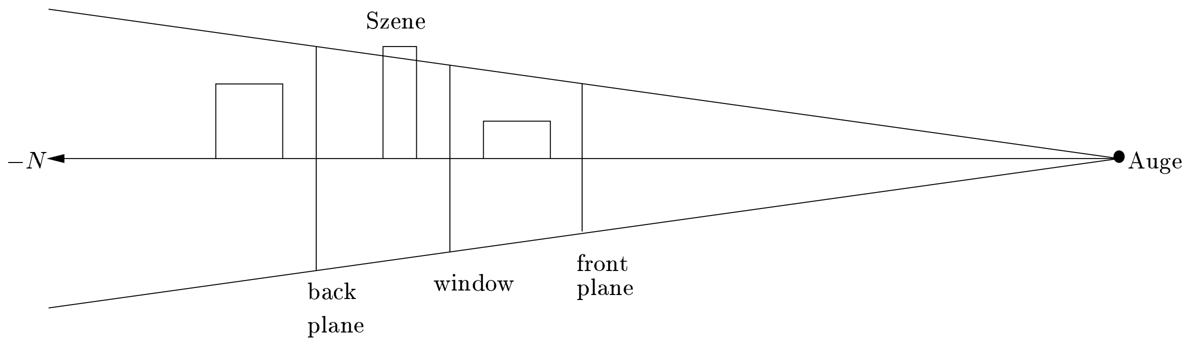
Die Parameter zur Durchführung der 3D-auf-2D-Abbildung (*viewing transformation*) sind im wesentlichen die Position und Orientierung einer *synthetischen Kamera* und ein Punkt in der Szene, auf den die Kamera fokussiert ist: Der View Reference Point (VRP).

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 NRP (Projection oder Normal Reference Point), der dem Augenpunkt eines natürlichen Betrachters entspricht.

Die Orientierung der Kamera wird durch den View Up Vector (VUV) 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 NRP 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*.



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

15.2 Viewing Pipeline

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. Clippe jede Polygonkante am Einheitswürfel.

4.) Device Mapping: NPC $\rightarrow$ DC
Bilde jeden Bildpunkt auf Gerätekoordinaten ab = übernimm x, y-Koordinaten; z liefert Tiefeninformation.

Beim Flächenmodell werden nach dem Clipping in Schritt 3.) die Polygone in Dreiecke zerlegt. Die Dreiecke sind versehen mit den Normalen bzgl. des Weltkoordinatensystems. Diese werden benötigt für das Schattieren der Dreiecke.

15.2.1 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.

15.2.2 View Orientation

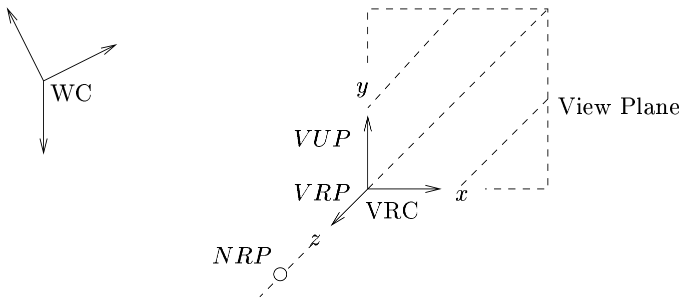
Zur Abbildung einer dreidimensionalen Szene aus den Weltkoordinaten auf den Bildschirm muß eine natürliche Betrachtersicht (*View Orientation*) definiert werden. Eine solche Betrachtersicht besteht aus den Viewing-Parametern

- Betrachterstandpunkt NRP (*Normal 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 NRP 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 NRP 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.
Der Abstand zwischen VRP und NRP wird als VPD (*View Plane Distance*) bezeichnet.



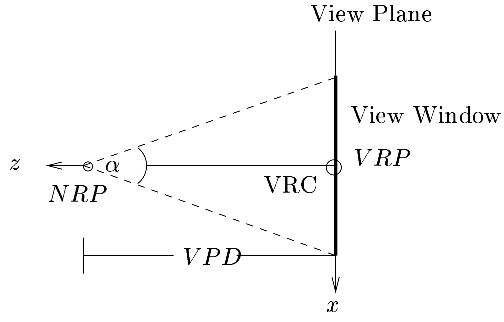
Definition des VRC

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 NRP 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 NRP 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 NRP wird.
- Eine weitere Möglichkeit sind Animationen, bei denen sich der Beobachter durch die Szene bewegt. Dieser Effekt läßt sich durch Verschieben von VRP und NRP um denselben Vektor erzielen.

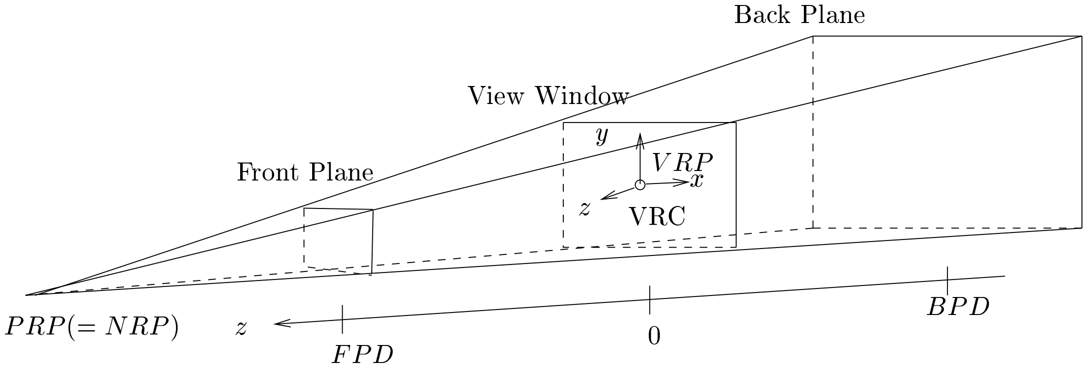
15.2.3 View Volume

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.



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 NRP als Projektionszentrum gewählt.

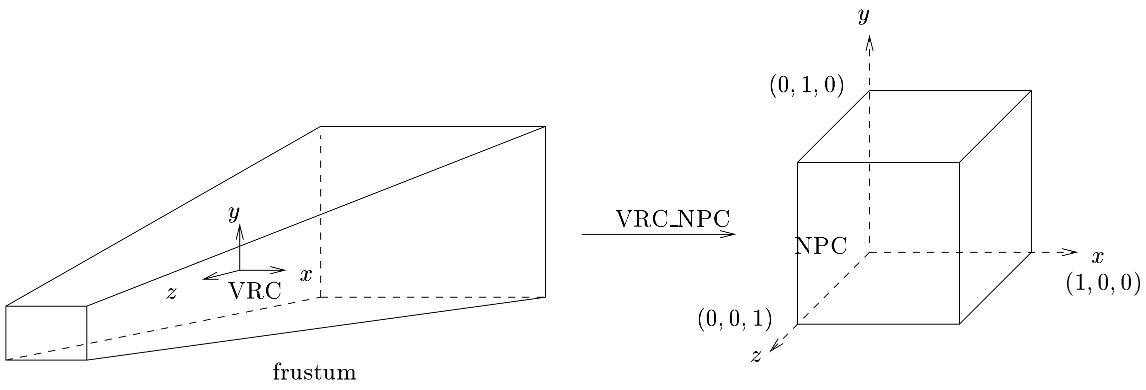


Bildraum bei perspektivischer Projektion

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.

15.2.4 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".



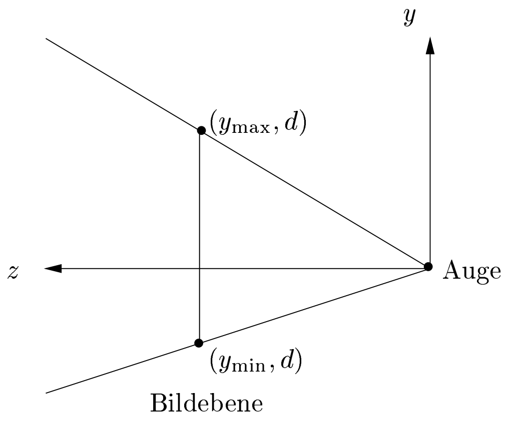
View Mapping

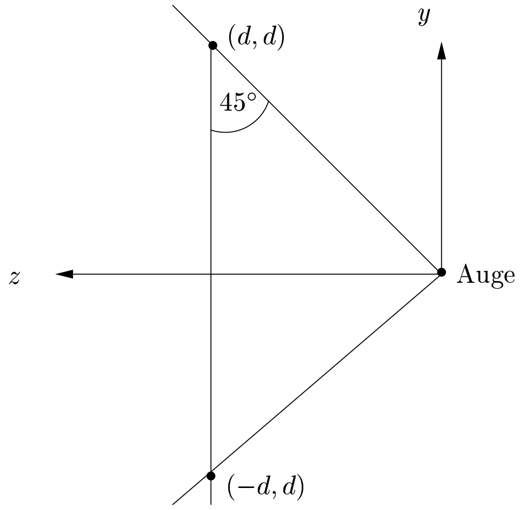
Bei der perspektivischen Projektion wird in einem ersten Schritt der Pyramidenstumpf, der den Bildraum darstellt und *frustum* genannt wird, auf einen regelmäßigen Pyramidenstumpf abgebildet. Dessen Grundfläche ist ein Quadrat, das mit jeder Seitenfläche einen Winkel von 45° einschließt. Dabei werden 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.

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°.





Es müssen also die y-Koordinaten proportional zur z-Koordinate verändert werden. Dies entspricht einer Scherung an der z-Achse und Skalierung, d.h.

$$\begin{aligned} y' &= k_1 \cdot y + k_2 \cdot z \\ z' &= z \end{aligned}$$

Punkt $(y_{\max}, d)$ soll abgebildet werden auf (d, d);
Punkt $(y_{\min}, d)$ soll abgebildet werden auf $(-d, d)$.

Durch Lösen des Gleichungssystems

$$\begin{aligned} d &= k_1 \cdot y_{\max} + k_2 \cdot d \\ -d &= k_1 \cdot y_{\min} + k_2 \cdot d \end{aligned}$$

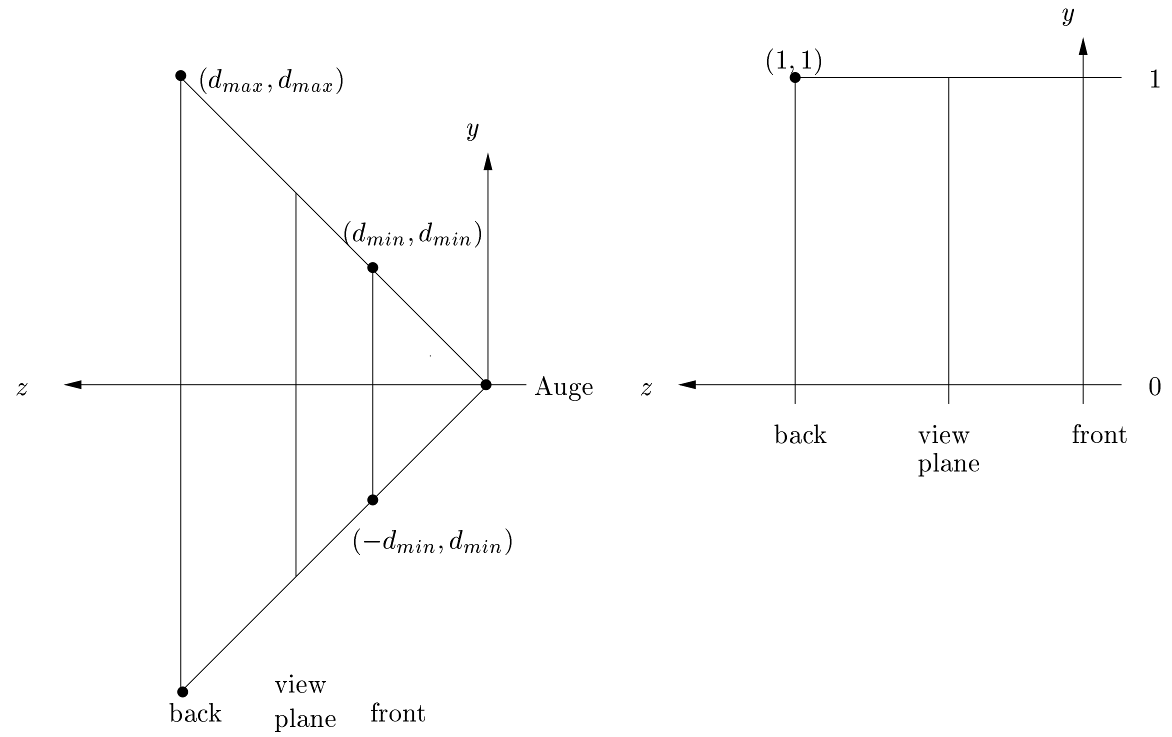
erhält man

$$k_1 = \frac{2d}{V_y}, \quad k_2 = -\frac{y_{\max} + y_{\min}}{V_y}$$

wobei $V_y = (y_{\max} - y_{\min})$ die Höhe des Bildfensters ist.
Analoge Scherung der x-Werte ergibt:

$$k_1 = \frac{2d}{V_x}, \quad k_2 = -\frac{x_{\max} + x_{\min}}{V_x}$$

wobei $V_x = (x_{\max} - x_{\min})$ die Breite des Bildfensters ist.
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 $z = 1$.



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 z' ist erforderlich wegen der Verwendung von homogenen Koordinaten mit 4×4-Transformationsmatrizen.
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

$$k_1 = \frac{1}{2}, \qquad k_2 = \frac{1}{2}.$$

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

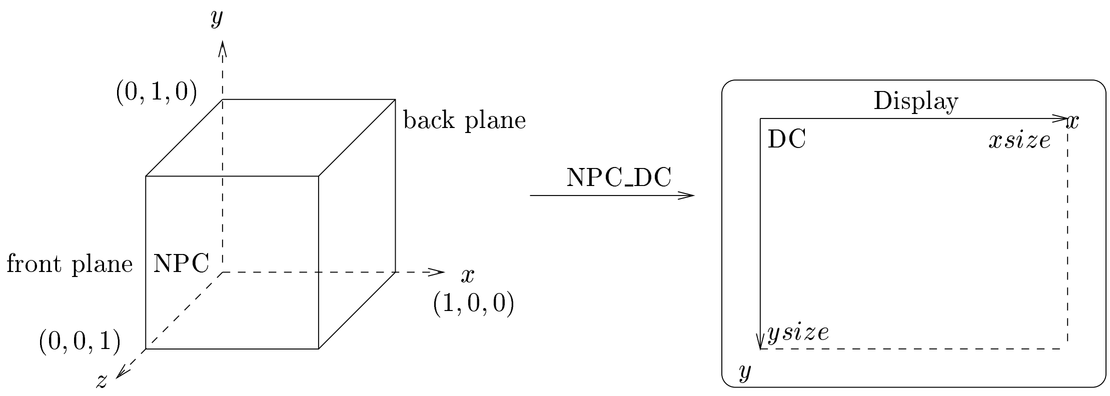
$$\begin{bmatrix} \frac{d}{x_{\max} - x_{\min}} & 0 & 0 & 0 \\ 0 & \frac{d}{y_{\max} - y_{\min}} & 0 & 0 \\ \frac{1}{2}(1 - \frac{x_{\min} + x_{\max}}{x_{\max} - x_{\min}}) & \frac{1}{2}(1 - \frac{y_{\min} + y_{\max}}{y_{\max} - y_{\min}}) & \frac{d_{\max}}{d_{\max} - d_{\min}} & 1 \\ 0 & 0 & \frac{-d_{\max} \cdot d_{\min}}{d_{\max} - d_{\min}} & 0 \end{bmatrix}$$

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.
Für das Clippen von Kanten am Einheitswürfel bietet sich die dreidimensionale Erweiterung des Algorithmus von Cohen & Sutherland an: Der Bereichs-Code benötigt 6 Bits.

15.2.5 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 (siehe z-Buffer-Algorithmus).



Device Mapping

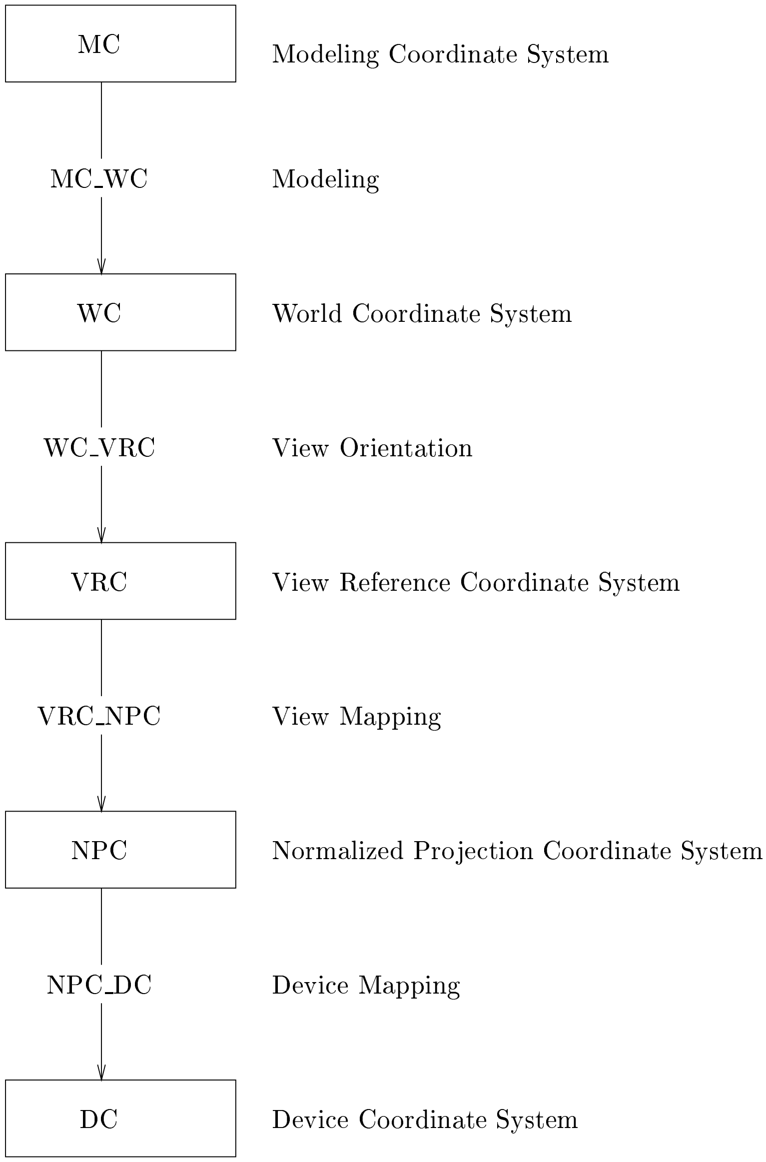
Die Transformationsmatrix entspricht einer Skalierung um den Vektor $(xsize, -ysize, 1)$ konkateniert mit einer Translation des Ursprungs in die linke untere Ecke des Bildschirms $(0, ysize, 0)$.

$$T_{\mathrm{NPC\_DC}} = \begin{bmatrix} xsize & 0 & 0 & 0 \\ 0 & -ysize & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & ysize & 0 & 1 \end{bmatrix}$$

Für die Projektion werden einfach die x- und y-Werte übernommen; die z-Werte sind erforderlich zur Regelung der Sichtbarkeit beim Tiefenpufferalgorithmus.

15.2.6 Zusammenfassung

Die Abbildung zeigt die schematische Transformationspipeline im Überblick. Diese wird bei der Ausführung des Programms von Polygonen durchlaufen, aus denen sich die Elementarobjekte zusammensetzen.



Transformationspipeline

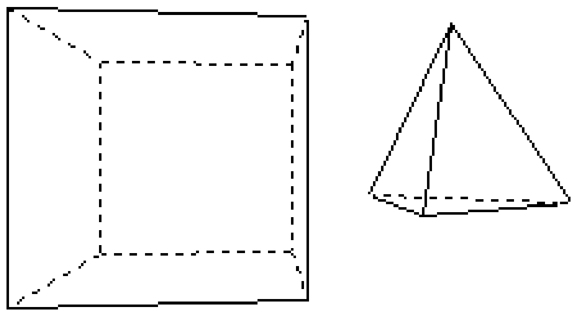
Kapitel 16

3D-Objekte

Im Flächenmodell werden Objekte durch ihre Begrenzungsflächen beschrieben und diese wiederum durch ihre Kanten mit den Eckpunkten 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.

16.1 Polyeder

Am Beispiel des Würfels wird die Repräsentation von Polyedern im Halbkantenmodell erläutert.



Vom Projektionsalgorithmus erzeugte Szene mit Würfel und Tetraeder in der Drahtmodell-Darstellung mit gestrichelten Rückkanten

16.1.1 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 Halbkantendarstellung definiert, ein Tetraeder beispielsweise mit Kantenlänge 1 und Schwerpunkt im Ursprung.

16.2 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.

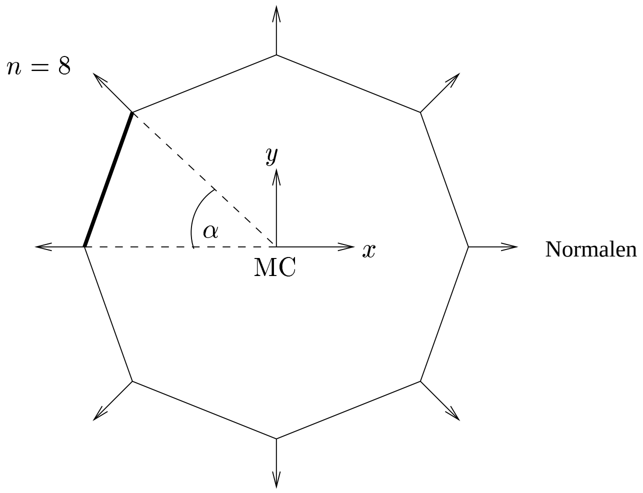
16.2.1 Zylinder

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.



Zylinder im Grundriß mit Normalenvektoren der Mantelfläche

16.2.2 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:

$$\alpha = (2\pi)/n, \quad n \in \mathbb{N} \text{ gerade} \ .$$

Dadurch entstehen auf der Kugel n Längenkreise und $(n/2)$ Breitenkreise. Diese schneiden n Dreiecke an jedem Pol und $n(n/2-2)$ Vierecke aus der Kugeloberfläche. Die Ortsvektoren 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 $\phi = k \cdot \alpha$, $k \in \{0, ..., n-1\}$.

Eines der Vierecke hat die Ortsvektoren

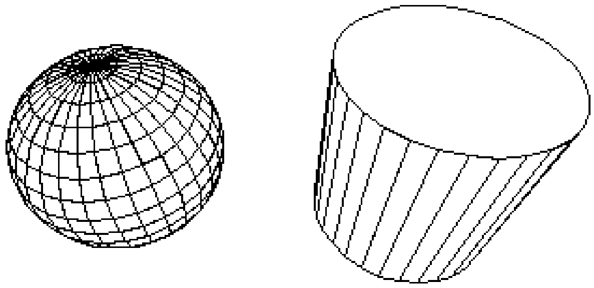
$$(\sin(\theta)\cos(\phi), \sin(\theta)\sin(\phi), \cos(\theta), 1),$$

$$(\sin(\theta)\cos(\phi+\alpha), \sin(\theta)\sin(\phi+\alpha), \cos(\theta), 1),$$

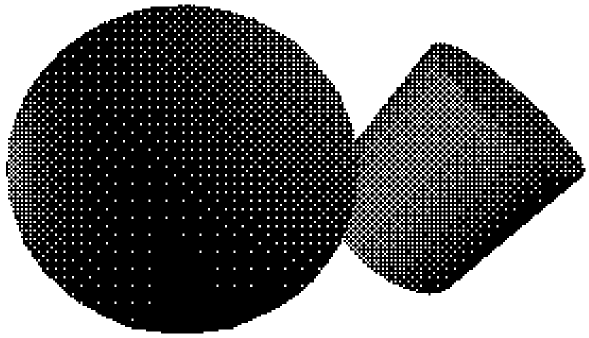
$$(\sin(\theta+\alpha)\cos(\phi+\alpha), \sin(\theta+\alpha)\sin(\phi+\alpha), \cos(\theta+\alpha), 1),$$

$$(\sin(\theta+\alpha)\cos(\phi), \sin(\theta+\alpha)\sin(\phi), \cos(\theta+\alpha), 1),$$

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.



Vom Projektionsalgorithmus erzeugte Szene mit Kugel und Zylinder in der Drahtmodell-Darstellung ohne Rückkanten ($n = 32$).



Vom Renderingalgorithmus erzeugte Szene mit Kugel und Zylinder von zwei Strahlern beleuchtet, Gouraud-schattiert.

Kapitel 17

Entfernen verdeckter Kanten und Flächen

Beim Entfernen verdeckter Kanten und Flächen werden unterschieden:

- Objektraum-Algorithmen (arbeiten auf Weltkoordinaten, vergleichen Objekte)
- Bildraum-Algorithmen (arbeiten auf Device-Koordinaten, vergleichen Pixel)

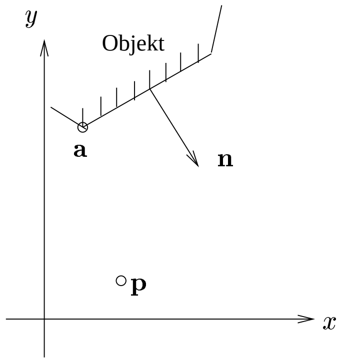
Das zunächst vorgestellte Back-Face Removal entfernt nur die Rückflächen einzelner Objekte, indem es im Objektraum (WC oder NPC) die Beziehung zwischen Betrachter und Objektfläche untersucht. Ob ein Objekt ein anderes verdeckt, kann dadurch nicht festgestellt werden.

17.1 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 $\mathbf{n}$ der Normalenvektor der Fläche und $\mathbf{a}$ ein Eckpunkt ist, dann kann hieraus die Gleichung der Ebene, in der die Fläche liegt, in der *Hesseschen Normalform* bestimmt werden:

$$\mathbf{p} \cdot \mathbf{n}^{\mathrm{T}} - \mathbf{a} \cdot \mathbf{n}^{\mathrm{T}} = e.$$

Beim Einsetzen verschiedener Punkte $\mathbf{p}$ ergeben sich unterschiedliche Werte für e. Gilt $e = 0$, so liegt $\mathbf{p}$ in der Ebene, bei $e > 0$ befindet sich $\mathbf{p}$ außen, d.h., die Fläche ist von $\mathbf{p}$ aus sichtbar, und bei $e < 0$ liegt $\mathbf{p}$ innen, d.h., die Fläche ist von $\mathbf{p}$ aus unsichtbar.



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.

17.2 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. Das Entfernen der nicht sichtbaren Flächen wird als *Hidden-Surface Removal* bezeichnet.

17.2.1 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. Da er 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:

```
Fuer jede Flaeche F tue:
  Fuer jedes Pixel (x, y) auf dieser Flaeche 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

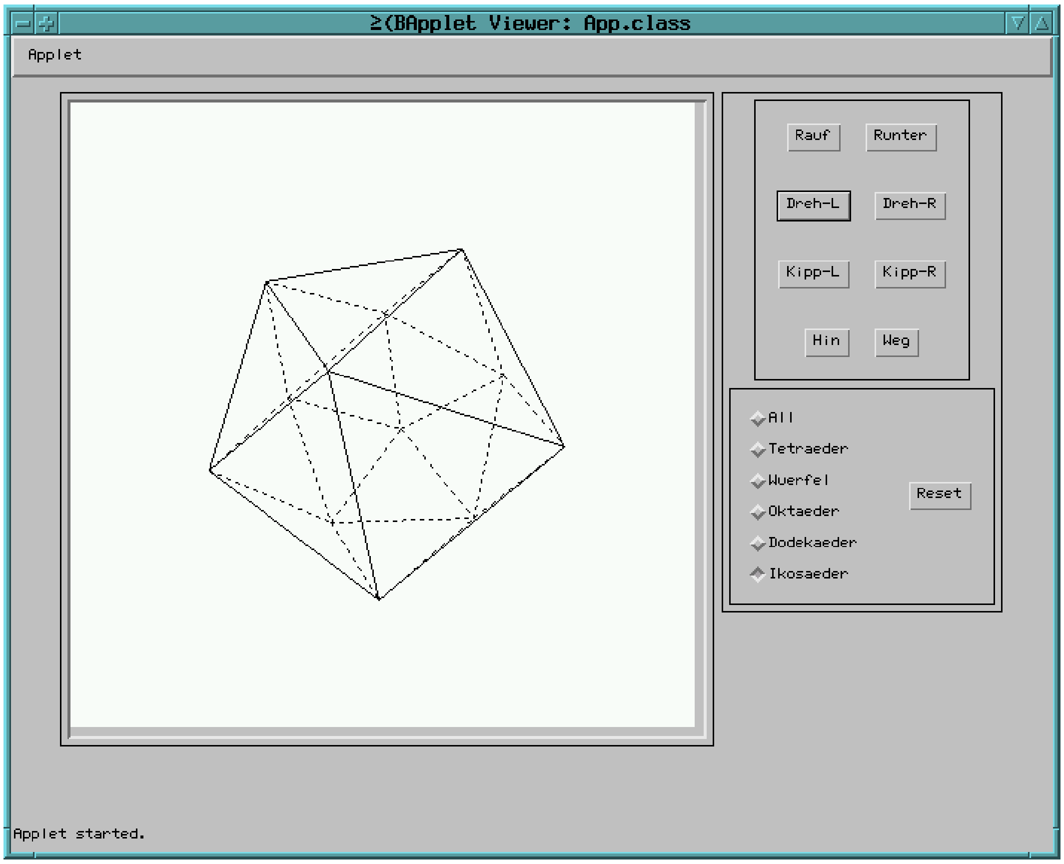
$$z_{i+1} = z_i - \frac{A}{C}$$

Für die Punkte (x_i, y_j) und (x_i, y_{j+1}) zweier benachbarter Scanlines ergibt sich

$$z_{j+1} = z_j - \frac{B}{C}$$

Das größte Problem bei der Implementierung des z-Buffers ist sein Speicherplatzbedarf. Eine ausreichende Auflösung der Tiefeninformation ergibt sich erst mit einem 32-Bit z-Wert, d.h. $maxint = (2^{31} - 1)$. Für die RGB-Tripel werden drei 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 800×600 Pixeln ergeben sich folglich 3,84 MB.

17.3 Beispiel-Applet zu 3D-Darstellung

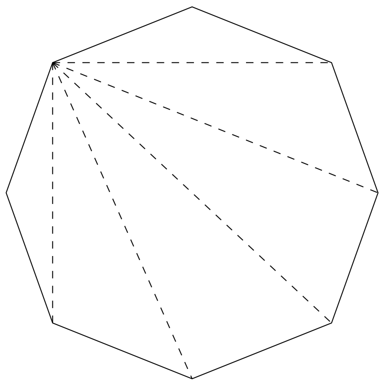


Screenshot vom 3D-wire-Applet

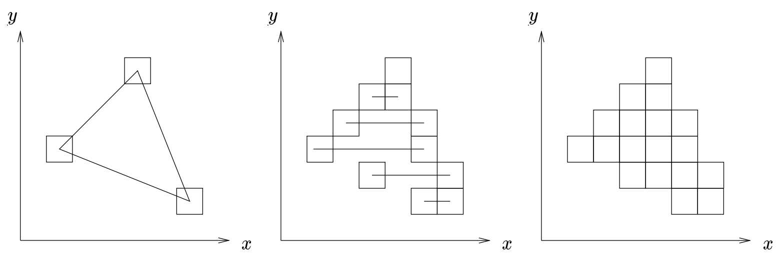
Kapitel 18

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.



Triangulisierung eines konvexen Polygons



Scanline-Verfahren zur Rasterung von Dreiecken

Unter Rasterung wird hier nicht nur das Füllen der Fläche in der Objektfarbe verstanden. Neben den Pixelkoordinaten müssen auch die z-Werte der Polygonpunkte interpoliert werden, um mit Hilfe dieser Tiefeninformation verdeckte Flächen zu unterdrücken (siehe z-Buffer-Algorithmus).
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.

18.1 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.

18.1.1 Lichtquellen

Der Grafikstandard PHIGS PLUS definiert vier Modelle von Lichtquellen:

- Umgebungslicht (*ambient light*),
- gerichtetes Licht (*directed light* oder *infinite light*),
- Punktlicht (*point light* oder *positional light*),
- Strahler (*spot light*).

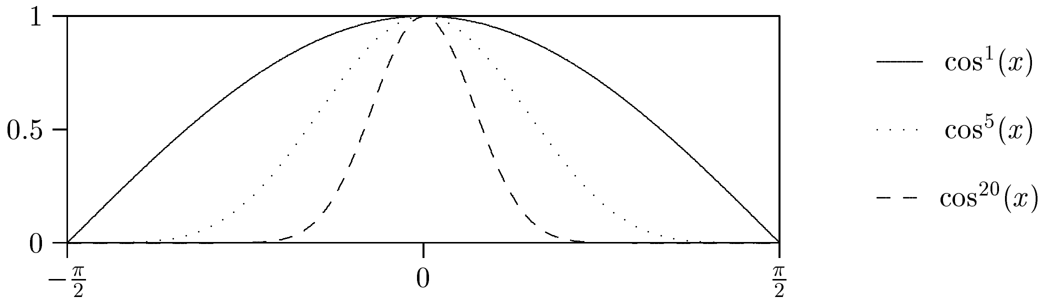
Das einfachste Lichtmodell ist das *Umgebungslicht*, das durch eine konstante Intensität I_a definiert ist. Diese ungerichtete Hintergrundbeleuchtung simuliert das Licht, das durch Reflexion an den einzelnen Gegenständen überall in einer Szene vorhanden ist.
Gerichtetes Licht ist definiert durch eine 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

$$I(r) = \frac{I_0}{C_1 + C_2 \cdot r}.$$

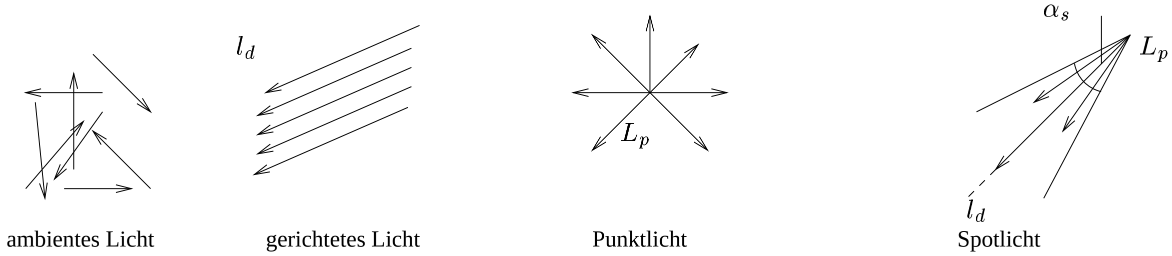
Dabei bezeichnet r den Abstand von der Punktlichtquelle. C_1 und C_2 heißen Abschwächungskoeffizienten (*attenuation coefficients*, $C_1 \geq 1$, $C_2 \geq 0$).
Das aufwendigste Lichtmodell ist der *Strahler* (*Spot*). Dieser hat wie das Punktlicht eine Intensität I_0, eine Position $\mathbf{L}_p$ sowie 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.



Lichtintensitätsverteilung beim Strahler (schematisch)



Vier Modelle von Lichtquellen (schematisch)

18.1.2 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.

18.1.3 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:

$$\overline{C_a} = k_a \cdot I_a \cdot \overline{O_d} \ .$$

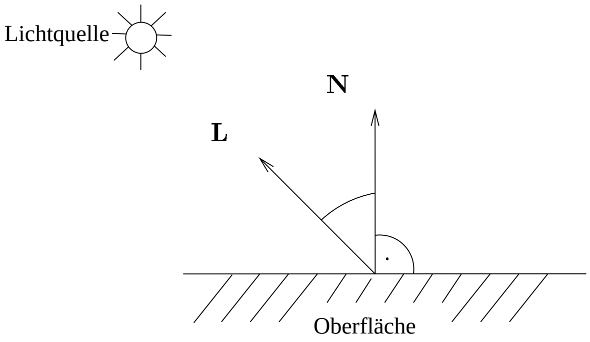
Die *diffuse Reflexion* basiert auf dem Phänomen der Streuung. Eintreffendes Licht dringt in den Körper ein und wird von seinen tieferen Schichten gleichmäßig in alle Richtungen gestreut. Das reflektierte Licht hat die diffuse Objektfarbe $\overline{O_d}$. Die Intensität I_e des einfallenden Lichts in 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.



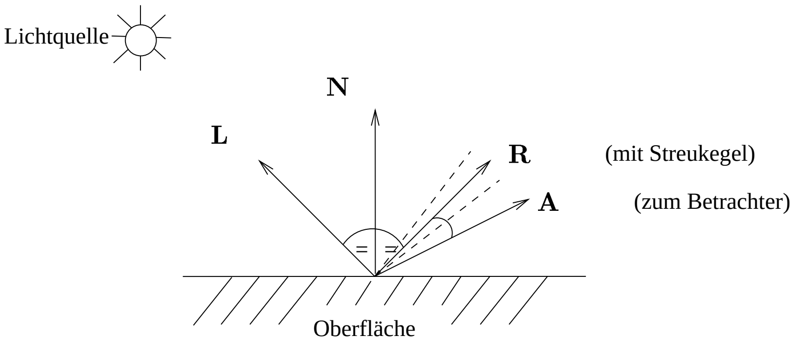
Cosinusgesetz von Lambert (für diffuse Reflexion)

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.



Spekulare Reflexion nach Phong

18.1.4 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.

18.2 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.

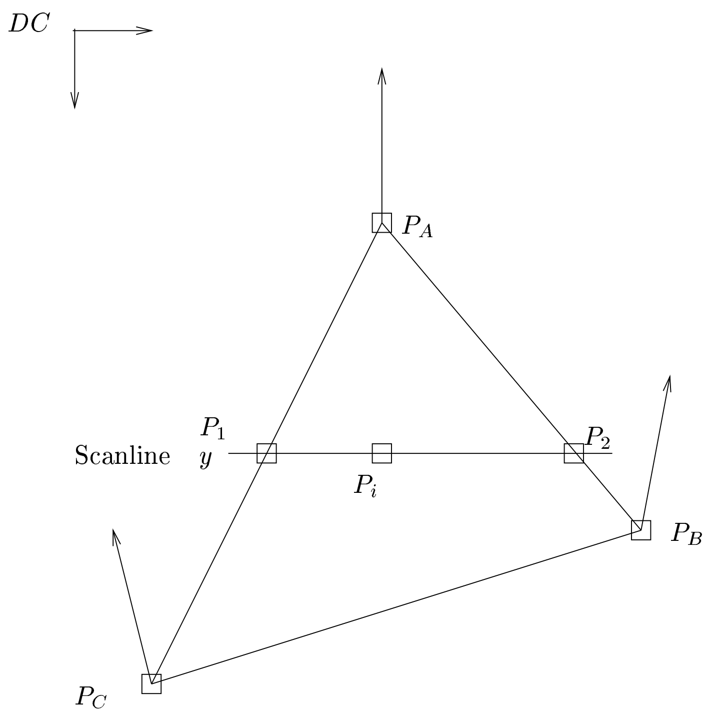
18.2.1 Flat-Shading

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 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).

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.



Dreieck-Shading

18.2.2 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:

$$\overline{C_1} = \overline{C_A}\frac{y - y_C}{y_A - y_C} + \overline{C_C}\frac{y_A - y}{y_A - y_C}, \quad \overline{C_2} = \overline{C_A}\frac{y - y_B}{y_A - y_B} + \overline{C_B}\frac{y_A - y}{y_A - y_B} \ .$$

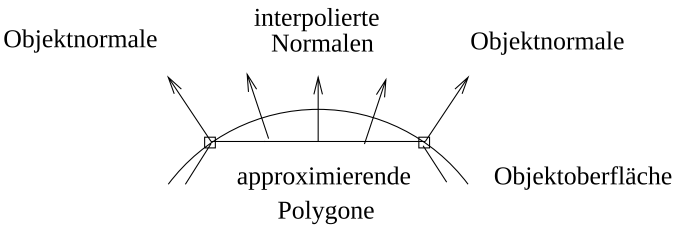
Für den Punkt $P_i(x_i, y)$ innerhalb der Fläche ergibt sich die Farbe aus den Randpunkten P_1 und P_2 der Scanline:

$$\overline{C_i} = \overline{C_1}\frac{x_2 - x_i}{x_2 - x_1} + \overline{C_2}\frac{x_i - x_1}{x_2 - x_1} \ .$$

Die Interpolation der Farbwerte ist im entwickelten Programm 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 Farbwertkurve 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.

18.2.3 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. 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.



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, was den Rechenaufwand vervielfacht.

18.3 Schatten

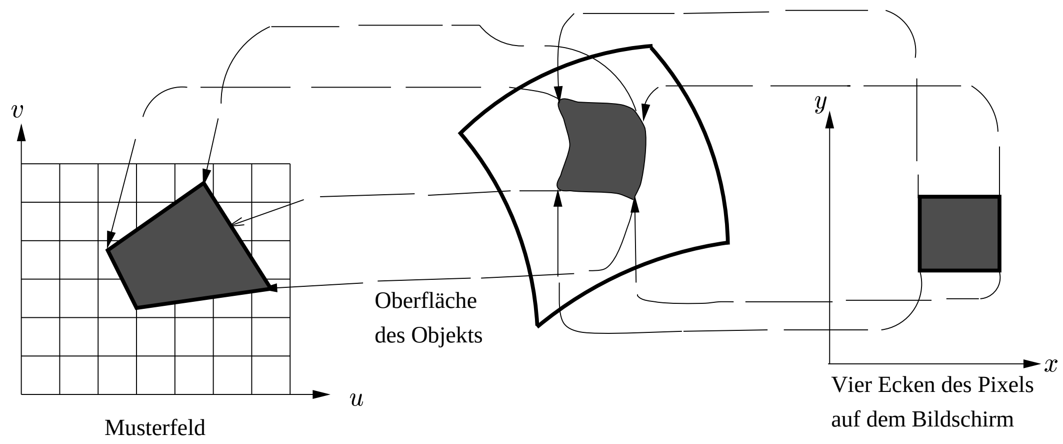
Zur Berechnung von Schatten eignen sich alle *Hidden-Surface*-Algorithmen, da die verdeckten Flächen einer Szene genau den beschatteten Flächen entsprechen, wenn die Position der Lichtquelle und des Betrachters zusammenfallen.
Zunächst wird daher als Phase 1 aus der Sicht der Lichtquelle die Szene in 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.

18.4 Texture Mapping

Zur realistischen Gestaltung von Oberflächen verwendet man ein zweidimensionales Musterfeld (*texture map*), aus dem für jedes Pixel die zugehörige Farbe ermittelt werden kann. Zugrunde gelegt sind zwei Abbildungen

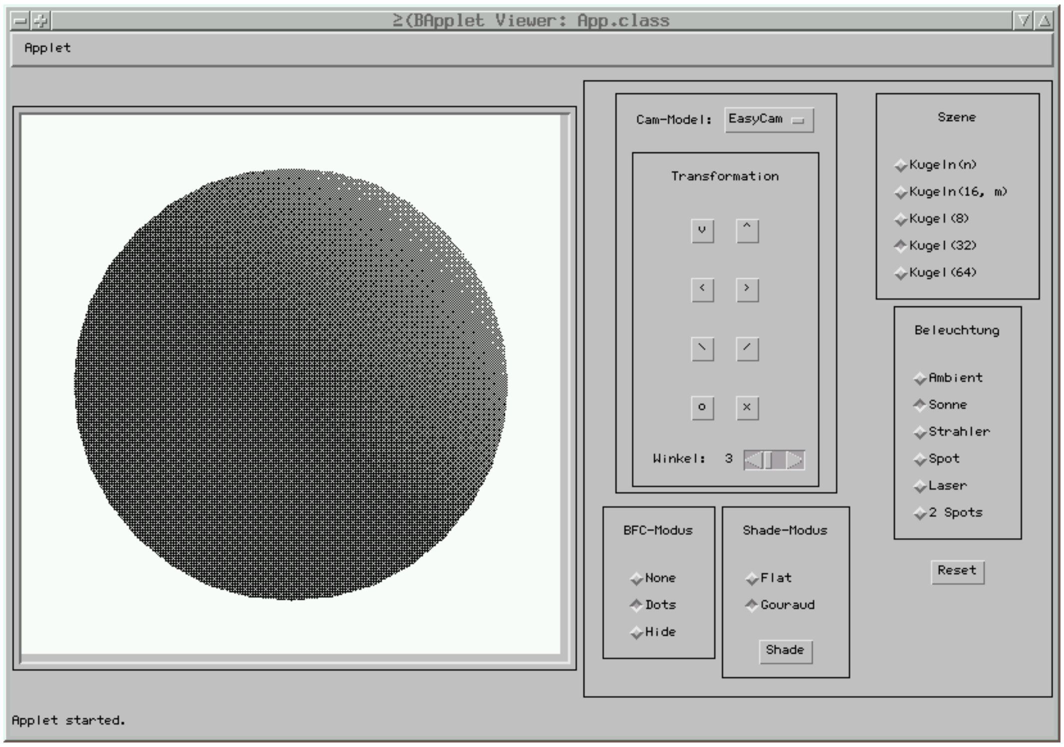
$$\begin{array}{ccc} \text{Musterraum} & (u,v) & \\ \downarrow & & \text{Abbildung des Musters} \\ \text{Objektraum} & (x_0,y_0,z_0) & \\ \downarrow & & \text{Projektion} \\ \text{Bildraum} & (x,y) & \end{array}$$

Die Verknüpfung der zugehörigen inversen Transformationen liefert die zum Einfärben eines Pixels benötigte Information.



Zum Einfärben eines Pixels im Bildraum wird die gewichtete Durchschnittsintensität aller überdeckten Pixel im Musterraum benutzt.
Für spezielle Objekte wie stark gemasertes Holz ist die Beschreibung über Musterfelder zu speicheraufwendig, so daß prozedural definierte Muster verwendet werden.
Um den Eindruck von rauhen Oberflächen zu erwecken, speichert man im Musterfeld Zusatzinformationen über die Veränderung des Normalenvektors (*bump mapping*).

18.5 Beispiel-Applet zu Shading-Modellen



Screenshot vom 3D-solid-Applet

Kapitel 19

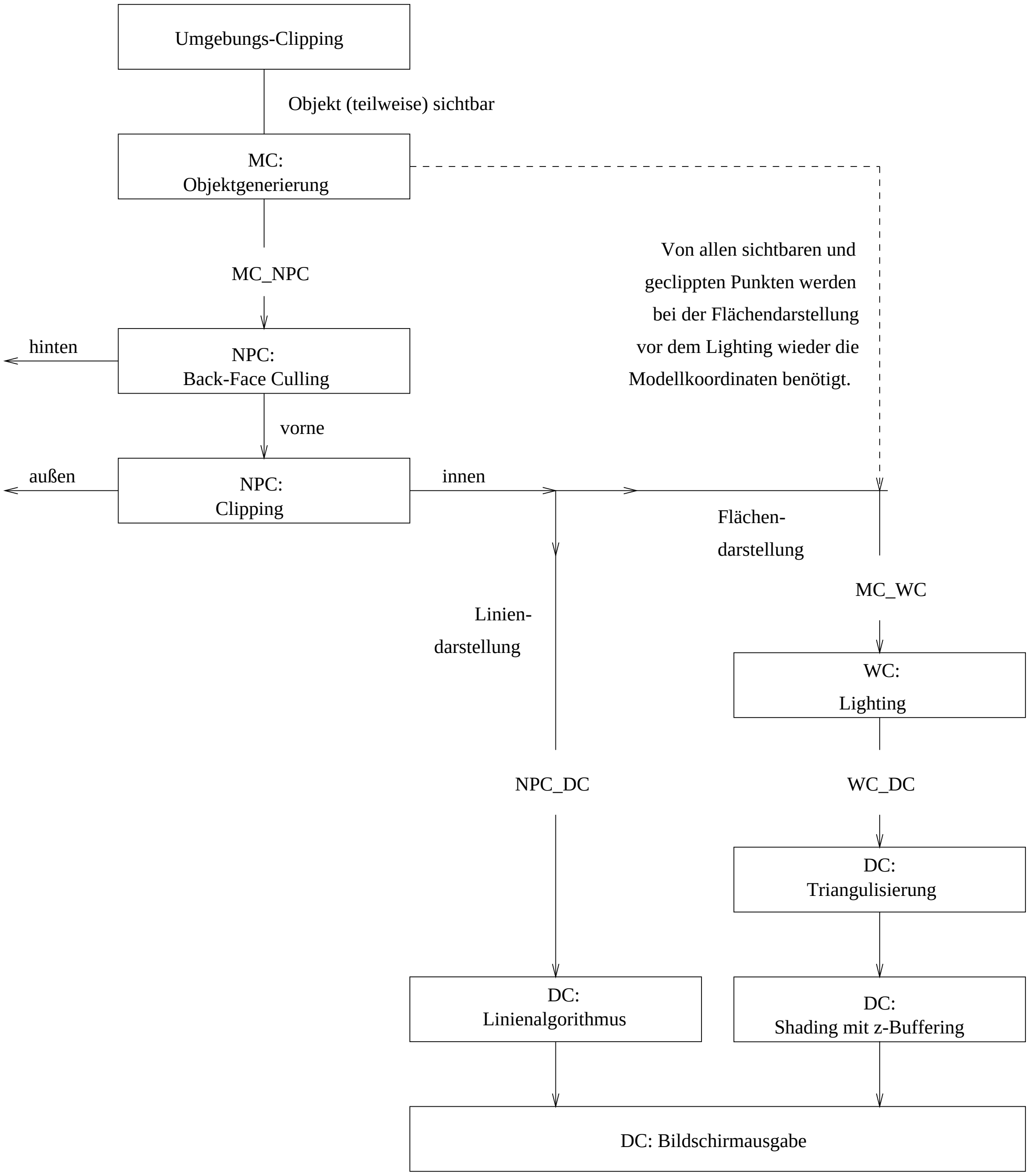
Die Viewing-Pipeline im Überblick

Hier wird im Überblick der gesamte Rendering-Prozeß schematisch dargestellt. Diese sogenannte *Viewing-Pipeline* beinhaltet zusammengefaßte Transformationen und zeigt die einzelnen Algorithmen in den jeweiligen Koordinatensystemen. Die Abweichung vom Vorgehen entlang der Transformationspipeline erklärt sich durch die Einbindung der aufwandsreduzierenden Verfahren Umgebungs-Clipping, Clipping und Back-Face Culling. Diese verringern die Anzahl und die Größe der Polygone, für die die aufwendigen Schattierungsalgorithmen durchgeführt werden müssen.

Durch die Einführung von Umgebungsboxen (Bounding Volumes) wird beim *Umgebungsclipping* zunächst statt eines komplexen Objekts ein umgebender Quader getestet. Ist dieser ganz unsichtbar, wird das Objekt nicht weiter bearbeitet. Ist er vollständig sichtbar, kann auf das (recht zeitaufwendige) Clipping verzichtet werden.

Die Viewing-Pipeline wird auf jedes Objekt der Szene angewandt. Wenn durch das Umgebungs-Clipping festgestellt wird, daß ein Objekt zumindest teilweise im View Window sichtbar ist, beginnt der Algorithmus mit dem Rendern des Objekts. Bei der Objektgenerierung werden die einzelnen Polygone des Objektprototyps im MC sukzessive berechnet und ggf. zusätzlich deformiert. Ein solches Polygon wird danach sofort gemäß der weiteren Viewing-Pipeline behandelt und unter Verwendung des z-Buffer-Algorithmus in den Bildschirmspeicher hineinschattiert. Bei der Liniendarstellung werden die Kanten des Polygons sukzessive gerastert.

Die Linien bzw. Polygone sind nach der Objektgenerierung unabhängig voneinander, so daß sie einzeln berechnet und dargestellt werden können. Die gesamte Szene besteht wegen der Flächenrepräsentation der Objekte praktisch nur aus ihnen. Der z-Buffer-Algorithmus sorgt bei der Flächendarstellung dafür, die Verdeckungseigenschaften pixelweise zu berechnen, so daß nach dem Rendering aller Polygone im Bildschirmspeicher das fertige Bild der Szene steht und ausgegeben werden kann. Die einzelnen Polygone sind sowohl bei der Liniendarstellung als auch triangulisiert beim Flat-Shading gut zu erkennen. Beim Gouraud- und besonders beim Phong-Shading tritt der gewünschte Effekt auf, daß sie fließend ineinander übergehen und nicht mehr explizit zu erkennen sind; das Bild wirkt jetzt realistisch.



Komplette Viewing-Pipeline

Kapitel 20

Algorithmen für 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. Der zweite Teil dieses Kapitels 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 Radiosity-Methoden 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.

20.1 Radiosity-Verfahren

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.

20.1.1 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:

$$\sum_{j=0}^{n-1} A[i,j]x[j] = b[i] \ .$$

Wenn alle Diagonalelemente von A ungleich Null sind, gilt:

$$x[i] = \frac{1}{A[i,i]} \left(b[i] - \sum_{j \neq i} A[i,j]x[j] \right) \ .$$

Das Iterations-Verfahren startet mit einer Schätzung x_0, die auf der rechten Seite eingesetzt wird zur Berechnung von $x_1[0]$. Im nächsten Schritt wird zur Berechnung von $x_1[1]$ bereits $x_1[0]$ benutzt. Ein Iterationsschritt lautet also:

$$x_k[i] = \frac{1}{A[i,i]} \left(b[i] - \sum_{j=0}^{i-1} x_k[j]A[i,j] - \sum_{j=i+1}^{n-1} x_{k-1}[j]A[i,j] \right) \ .$$

Bei Konvergenz wird das Verfahren nach k Iterationsschritten abgebrochen, wenn $b - Ax_k$ genügend klein geworden ist.

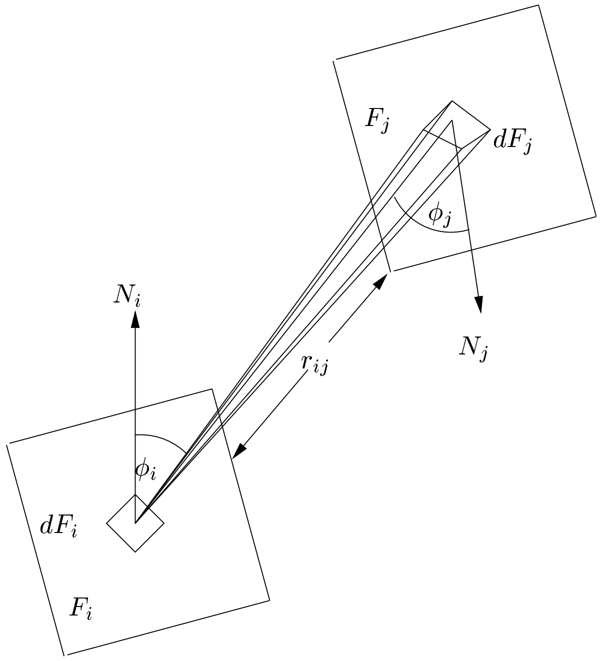
20.1.2 Berechnung der Formfaktoren

Die Formfaktoren werden bestimmt mit Hilfe der Gleichung

$$F_{ij} = \frac{1}{F_i} \int_{F_i} \int_{F_j} \frac{\cos(\phi_i)\cos(\phi_j)}{\pi r_{ij}^2} b_{ij} dF_j dF_i$$

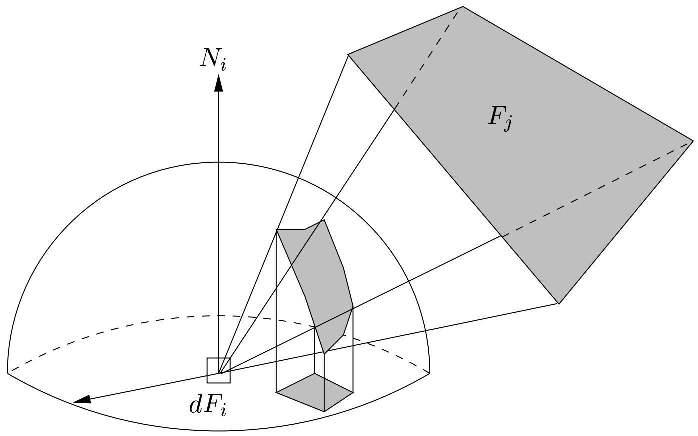
mit

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

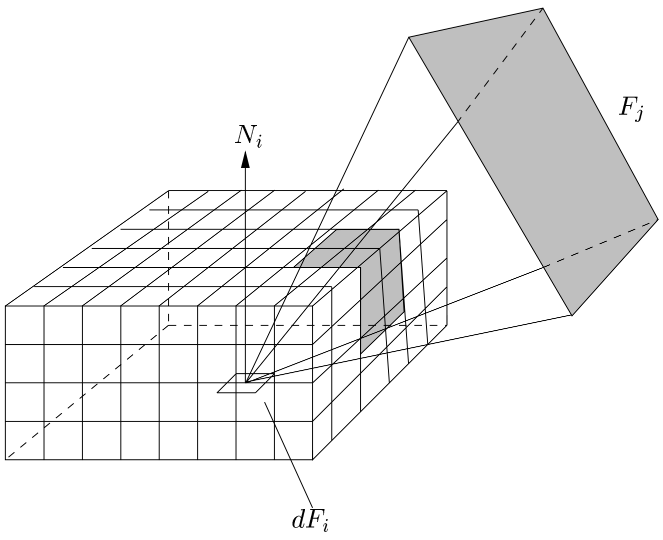


Der Berechnung von Formfaktoren zugrundeliegende Geometrie

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 $\cos\phi_j/r^2$, 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.



Geometrische Interpretation des Formfaktors



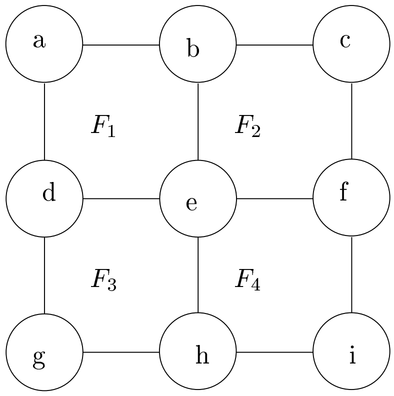
Der Halbwürfel ist die obere Hälfte eines Würfels, der um das Flächenelement zentriert ist.

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

$$\sum_{j=1}^{n} F_{ij} = 1 \ .$$

20.1.3 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.

20.1.4 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.

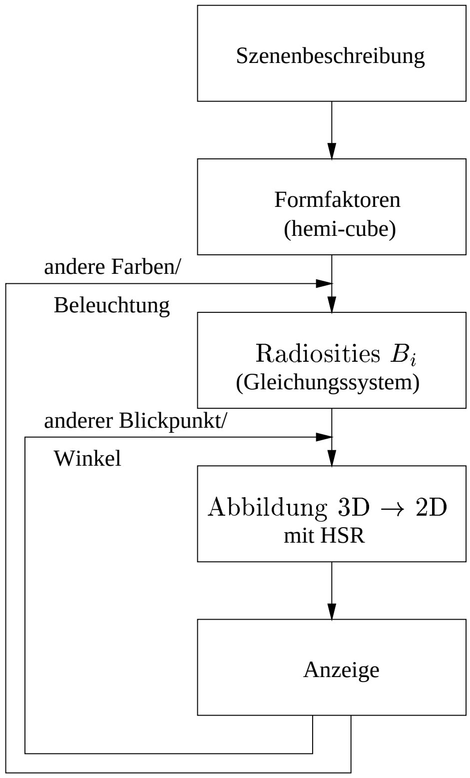
Im folgenden bezeichnet ΔB_i also die noch nicht verteilte Strahlung, d.h. die Differenz zwischen der *Radiosity* im letzten und im gegenwärtigen Iterationsschritt. Es wird ausgenutzt, daß gilt $F_{ji} = (F_{ij} \cdot F_i)/F_j$.

```
for i:= 1 to n do                          {initialisiere}
   if patch i ist Lichtquelle              {für jede Fläche}
     then B_i := ΔB_i := Emissionswert     {Strahlung und Differenz}
     else B_i := ΔB_i := 0
end;

repeat
   for {jede Fläche i, beginnend bei der mit der größten Ausstrahlung} do begin
     Plaziere Hemicube auf Fläche i
     Berechne F_ij für alle 1 ≤ j ≤ n
     for j := 1 to n do begin              {für jede Fläche j tue}
        ΔR := ρ_j * ΔB_i * F_ij * F_i/F_j  {Strahlung von Fläche i}
        ΔB_j := ΔB_j + ΔR                  {Differenz erhöhen}
        B_j := B_j + ΔR                    {Strahlung erhöhen}
     end;
     ΔB_i := 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.

20.1.5 Zusammenfassung

Als Gesamtablauf des Radiosity-Verfahrens ergibt sich

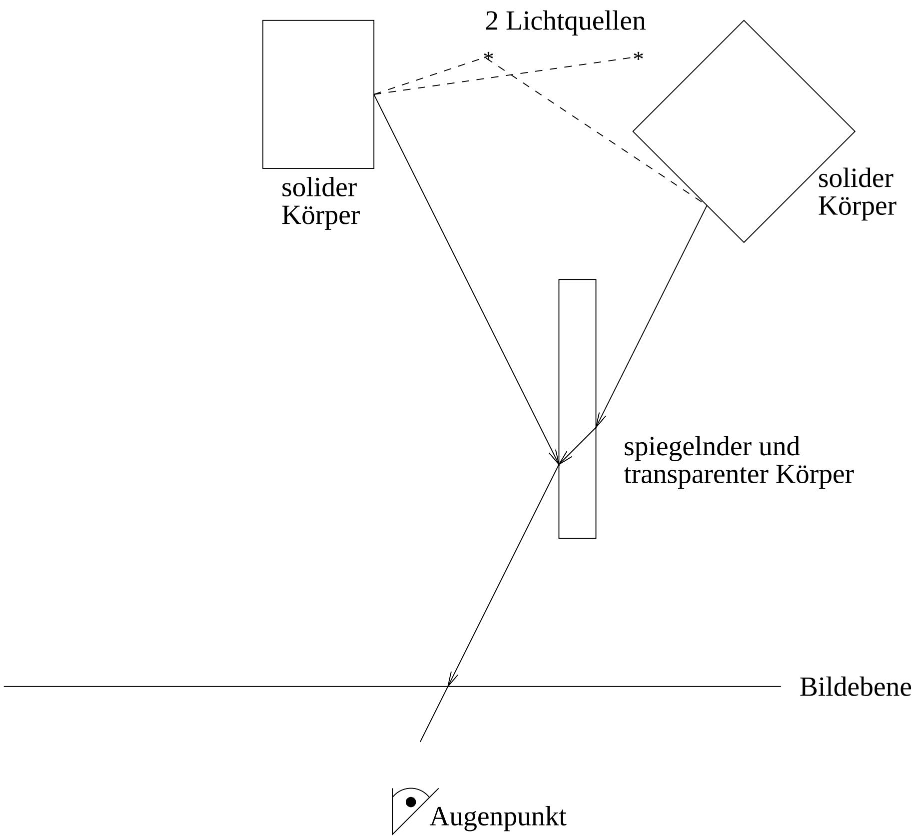


20.2 Ray Tracing

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.

20.2.1 Grundlagen

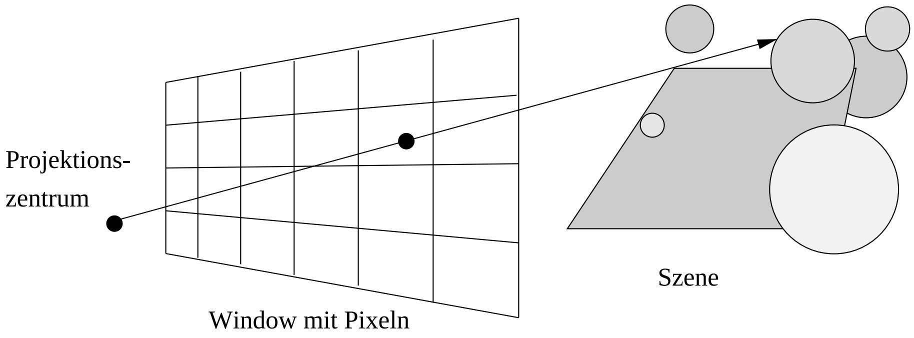
Ausgehend vom Augenpunkt wird durch jedes Pixel ein Strahl gelegt und der Schnittpunkt dieses Strahls mit dem ersten getroffenen Objekt bestimmt. Trifft der Strahl auf kein Objekt, so erhält das Pixel die Hintergrundfarbe. Ist das Objekt spiegelnd, so wird der Reflexionsstrahl berechnet und rekursiv 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.



20.2.2 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.




```
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;
   }
}
```

20.2.3 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.

20.2.4 Effizienz beim Ray Tracing 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

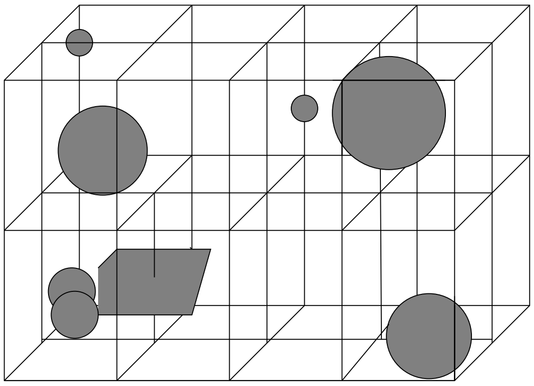
Die Formel für den Schnittpunkt mit einer Kugel läßt sich verbessern. Wenn man die Strahlen so transformiert, daß sie entlang der z-Achse verlaufen, kann man die gleiche Transformation auf die getesteten Objekte anwenden. Dann liegen alle Schnittpunkte bei $x = y = 0$. Dieser Schritt vereinfacht die 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

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.

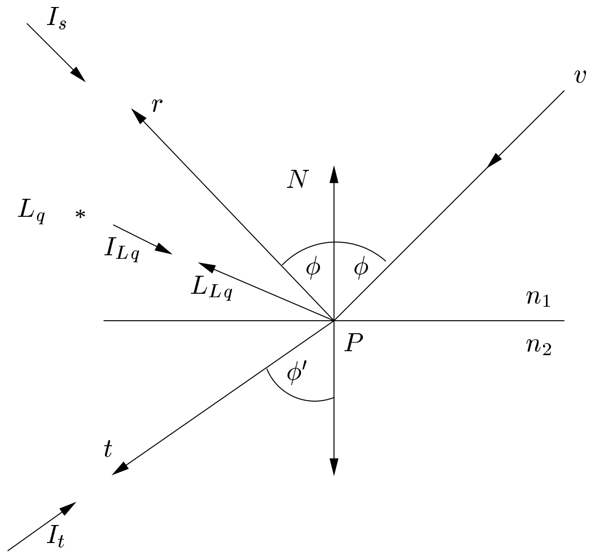


Die Szene wird durch ein regelmäßiges Gitter gleich großer Volumen unterteilt.

20.2.5 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). 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

$$\begin{aligned} r &= v + 2\cos(\phi)N \\ t' &= p(N + v') - N \\ \text{mit } v' &= \frac{v}{|v \cdot N|} \text{ und } p = \frac{1}{\sqrt{n^2|v'|^2 - |v' + N|^2}} \ . \end{aligned}$$

Dabei ist r ein Einheitsvektor und t' ein Vektor in Transmissionsrichtung t. Ist p ein imaginärer Ausdruck, liegt Totalreflexion vor, und der Transmissionsanteil wird gleich Null gesetzt.

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 Transmissionsintensität
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;
}
```

```
/* Berechnen 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 wird;
        Skaliere damit die Terme für diffuse und spiegelnde Reflexion, bevor sie
        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 */
}
```

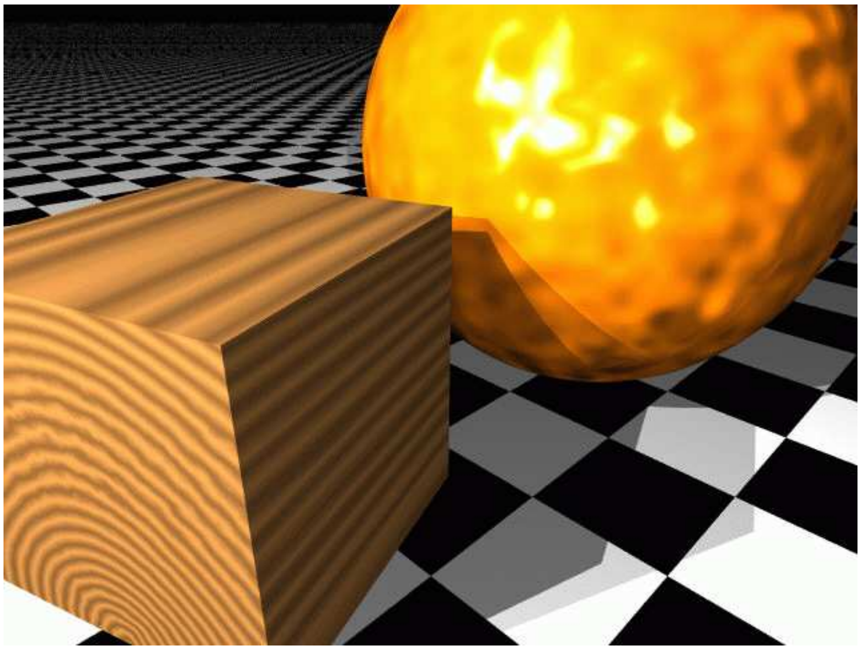
Ray Tracing ist besonders anfällig für Probleme, die durch die begrenzte Rechengenauigkeit entstehen. Dies zeigt sich vor allem bei der Berechnung der Schnittpunkte sekundärer Strahlen mit den Objekten. Wenn man die x-, y- und z-Koordinaten eines Objekts mit einem Augstrahl berechnet hat, dienen sie 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.

20.2.6 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. 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.

Beispiel einer Szenenbeschreibung

```
#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-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)
```

Mit Hilfe des folgenden Shell-Skripts wird eine GIF-Datei für eine Colormap mit 256 Farben erstellt:

```
#!/bin/sh

tgatoppm $1.tga | ppmquant 256 | ppmtogif > $1.gif
```

Die Ausgabe dabei lautet:

```
ppmquant: making histogram...
ppmquant: 2053 colors found
ppmquant: choosing 256 colors...
ppmquant: mapping image to new colors...
ppmtogif: computing colormap...
ppmtogif: 239 colors found
```