Computergrafik

Vorlesung gehalten im SS 2010
[Version vom 18. März 2010]

Oliver Vornberger

Institut für Informatik
Universität Osnabrück

Literatur

- W. D. Fellner:
  ”Computergrafik”
  BI Wissenschaftsverlag, 2. Auflage, 1992
- J. D. Foley, A. van Dam, S. K. Feiner et al:
  ”Computer Graphics: Principle and Practice”
  Addison Wesley, 1995.
- Dave Shreiner:
  ”OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1”
  Addison-Wesley, 2009
- Randi J. Rost, Bill Licea-Kane:
  ”OpenGL Shading Language, Third Edition (Orange Book)”
  Addison-Wesley, 2009
- David Flanagan:
  ”Java in a nutshell”
  O’Reilly, 2005.
- Alan Watt:
  ”3D-Computergrafik”
  Addison Wesley, 3. Auflage, 2002
- Klaus Zeppenfeld:
  ”Lehrbuch der Grafikprogrammierung”
  Spektrum, 2004.

Danksagung

Ich danke ...

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

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

... Olaf Müller für eine gründliche Überarbeitung des Skripts und für umfangreiche Implementationen zur Viewing-Pipeline,

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

... Frank M. Thiesing, Ralf Kunze, Dorothee Langfeld und Patrick Fox für intensive Mitwirkung an der inhaltlichen Gestaltung der Vorlesung.

... Henning Wenke für die Erstellung des Kapitels zu OpenGL

... Nicolas Neubauer für die Übernahme des Übungsbetriebs

HTML-Version
Der Inhalt dieser Vorlesung kann online betrachtet werden unter
`http://www-lehre.inf.uos.de/~cg/2010`

Osnabrück, im März 2010

(Oliver Vornberger)

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

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

1.2 Definition

Der Begriff *Grafische Datenverarbeitung* umfaßt

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

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

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

In dieser Vorlesung geht es ausschließlich um generative Computergrafik!

1.3 Anwendungen

Typische Anwendungsbereiche von Computergrafik sind:

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

1.4 Kurze Geschichte der Computergrafik

1950	grobe Grafikdarstellung mit Matrixdrucker
1950	computergesteuerte Kathodenstrahlröhre
1963	Dissertation von Ivan Sutherland: *Sketchpad - A Man Machine Graphical Communication System*
1964	Automobilentwurf bei GM; Anfänge von CAD/CAM
1965	Doug Engelbart: Maus ersetzt Lichtgriffel
1980	Apple Macintosh und IBM PC mit Bitmap-Grafik und Maus
1985	GKS: Grafisches Kernsystem
1986	X-Windows vom MIT
1987	GKS-3D
1988	PHIGS: Programmer's Hierarchical Interactive Graphics System
1989	PEX: PHIGS Extension to X
1991	CGI: Computer Graphics Interface
1992	ISO PHIGS PLUS: (+ Rendering)
1993	OpenGL
1995	DirectX 1.0
1996	3D-Grafikkarte für den Consumerbereich (VooDoo 1)
1998	nVidia Riva TNT (2D/3D-Kombigrafikkarte)
2002	DirectX 9.0
2004	OpenGl 2.0
2007	DirectX 10.0
2008	OpenGl 3.0
2010	OpenGl 4.0

Kapitel 2

GUI-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 (*Graphical User Interfaces*, abgekürzt *GUI*) haben sich heute zum Standard für anwenderfreundliche Applikationsprogramme entwickelt und rein textuelle Mensch-Maschine-Kommunikation weitgehend abgelöst. Was dem Benutzer im allgemeinen das Leben erleichtern soll, stellt für den Programmierer eine echte Herausforderung dar.

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

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

2.1 Der Window-Manager

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

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

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

Außerdem kann der Window-Manager neue Applikationen starten.

Für (fast) jedes Betriebssystem gibt es ein zugehöriges Oberflächensystem, das mit Hilfe eines API (Application Programmer's Interface) manipuliert werden kann. Die Programmierung einer grafischen Applikation erfolgt mit Hilfe einer Graphical User Interface (GUI)-Sprache, die in einer Hochsprache alle notwendigen API-Vokabeln zur Beschreibung des Aufbaus und Ablaufs einer interaktiven grafischen Anwendung bereitstellt. Die Programmierung erfolgt zumeist in einer gängigen Program-

miersprache, die GUI-Bibliotheken verwendet.

Die Beispiele in dieser Vorlesung werden in Java 1.4 unter Zuhilfenahme der Swing-API implementiert.

2.2 Swing

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

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

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

2.2.1 Swing-Übersicht

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

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

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

Die Klassen des swing-Pakets lassen sich grob in drei Gruppen einteilen: Grafikklassen, in denen grafische Objekte wie Farben, Fonts, Bilder beschrieben werden; Komponenten, d.h. Klassen, die GUI-Komponenten wie z.B. Buttons, Menüs und Textfelder zur Verfügung stellen; sowie Layout-Manager, die die Anordnung von GUI-Komponenten in einer Applikation kontrollieren. Aus den verwandten Paketen sollen hier noch die Event-Listener besprochen werden, mit deren Hilfe die Interaktion zwi-

schen User und Applikationen implementiert werden muß. Die folgende Übersicht erhebt keinen Anspruch auf Vollständigkeit, sondern beschreibt nur kurz die für Computergrafik-Programme wichtigen Klassen. Weitere Klassen sowie alle Details sind der entsprechenden On-Line-Dokumentation zu entnehmen.

Die Grafik-Klassen

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

Die GUI-Komponenten

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

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

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

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

- `JButton`: Es wird ein mit einem Titel oder Bild versehenes Feld erzeugt, das mit der Maus angeklickt werden kann, wodurch ein entsprechender Event erzeugt wird.
- `JCheckbox`: Ein Schalter mit zwei Zuständen, der bei einem Mausklick diesen wechselt und dabei einen Event auslöst.
- `JComboBox`: Eine Gruppe von Einträgen, von der immer nur einer sichtbar ist. Das Auswählen eines Eintrags erzeugt einen entsprechenden Event.
- `JList`: Es wird eine Liste von Strings angezeigt, aus der Einträge selektiert bzw. deselektiert werden können, was zur Erzeugung von entsprechenden Events führt.

- `JScrollBar`: Ein Rollbalken, der mit der Maus bewegt werden kann, wodurch entsprechende Events erzeugt werden.

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

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

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

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

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

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

Die Layout-Manager

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

- `FlowLayout`: Dies ist das einfachste Layout, welches bei `JPanel` voreingestellt ist. Jede Komponente wird in seiner bevorzugten Größe rechts von der vorigen Komponente eingefügt. Ist der rechte Rand des Containers erreicht, wird eine weitere Zeile angehängt, in der weitere Komponenten eingefügt werden können.
- `BorderLayout`: Dies ist ebenfalls ein einfaches Layout, das als Default-Layout bei `JFrame` verwendet wird. Es besteht aus den fünf Gebieten, Nord, Süd, Ost, West und Zentrum, in denen jeweils beliebige Komponenten plaziert werden können. Im Gegensatz zum `FlowLayout` werden hier die Komponenten an die Größe des Containers angepaßt: Die Nord- und Süd-Komponenten werden in X-Richtung gestreckt bzw. gestaucht; die Ost- und West-Komponenten in Y-Richtung und die Zentrums-Komponente in beiden Richtungen.
- `CardLayout`: Dies ist ein spezielles Layout, bei dem beliebig viele Komponenten eingefügt werden können, von denen aber immer nur eine zu sehen ist, da sie wie bei einem Kartenspiel übereinander angeordnet werden.

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

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

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

Event-Handling

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

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

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

Tabelle 2.1: AWT-Event-Listener

2.3 Swing-Beispiel

In diesem Abschnitt wird eine einfache grafische Anwendung vorgestellt, in der eine Zahl per Knopfdruck wahlweise hoch- oder runtergezählt und sowohl numerisch als auch per Slider angezeigt wird. Diese Anwendung wird zum Einen als eigenständige Applikation und zum Anderen als Applet, welches in einem Webbrowser zu betrachten ist, realisiert.

Die Anwendung besteht aus fünf Klassen. Der Kern mit dem grafischen User-Interface und dem Eventhandling ist gemäß der *Model-View-Controller*-Architektur in den Klassen `Zustand` (*Model*), `RaufRunter` (*View*) und `KnopfKontrollierer` (*Controller*) implementiert. Die Klassen

`RaufRunterApp` und `RaufRunterApplet` stellen die Top-Level Container dar, in denen die Anwendung als Application bzw. als Applet ausgeführt wird.

```
import java.util.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RaufRunter extends JPanel implements Observer {

  private JButton rauf;                              // Button zum raufzaehlen
  private JButton runter;                            // Button zum runterzaehlen
  private Zustand z;                                 // Zustand
  private JLabel  ergebnis;                          // Label zur Anzeige
  private JSlider schieber;                          // Slider zur Visualisierung
  private Font    font;                              // Font fuer Label

  public RaufRunter() {                              // Konstruktor

    setLayout(new GridLayout(0,1));                  // einspaltiges Gridlayout

    rauf   = new JButton("Addiere");                 // Knopf rauf anlegen
    runter = new JButton("Subtrahiere");             // Knopf runter anlegen
    schieber = new JSlider(0,100,42);                // Slide von 0 bis 100
    ergebnis = new JLabel("42 ",JLabel.CENTER);      // JLabel ergebnis anlegen
    font = new Font("SansSerif",Font.BOLD,30);       // Schriftgroesse einstellen
    ergebnis.setFont(font);                          // an Label uebergeben

    add(rauf);                                       // JButton rauf einfuegen
    add(ergebnis);                                   // JLabel ergebnis einfuegen
    add(schieber);                                   // Slider einfuegen
    add(runter);                                     // JButton runter einfuegen

    z = new Zustand(42);                             // Zustand initialisieren
    z.addObserver(this);                             // dort als Observer eintragen

    KnopfKontrollierer raufK;                        // fuer rauf-Eventhandling
    raufK = new KnopfKontrollierer(z,+1);            // soll hochzaehlen
    rauf.addActionListener(raufK);                   // Listener an rauf haengen

    KnopfKontrollierer runterK;                      // fuer runter-Eventhandling
    runterK = new KnopfKontrollierer(z,-1);          // soll runterzaehlen
    runter.addActionListener(runterK);               // Listener an runter haengen
  }

  public void update(Observable z, Object dummy){ // wird aufgerufen bei notify
    ergebnis.setText(((Zustand)z).get() + " ");    // Zaehlerstand anzeigen
    schieber.setValue(((Zustand)z).get());         // Zaehlerstand visualisieren
  }
}
```

Die Klasse `RaufRunter` stellt die *View* dar und stammt von `JPanel` ab. Bei der Instanziierung wird zunächst das Layout festgelegt und im weiteren werden die benötigten Komponenten erstellt. Für das

Programm wird ein `JButton` zum Hochzählen, ein `JButton` zum Runterzählen, ein `JLabel` zur Anzeige und ein `JSlider` zur Visualisierung des Zahlenwerts benötigt. Alle Komponenten werden innerhalb des einspaltigen Gridlayouts auf dieselbe Größe gesetzt. Diese ist nicht fest und kann sich ggf. durch die Größe des Top-Level Containers (das Applet oder die Applikation) ändern. Diese vier Komponenten werden dem `JPanel` hinzugefügt, welches in den jeweiligen Top-Level Container eingefügt werden kann.

Weiterhin wird eine Instanz der Klasse `Zustand` initialisiert, welche die Rolle des *Model* übernimmt, sowie zwei Instanzen des `KnopfKontrollierer`, welche als `ActionListener` beim jeweiligen Knopf eingehängt werden.

Schließlich wird mit der Methode `update` das Interface `Observer` implementiert, wodurch auf eine Zustandsänderung reagiert werden kann, die vom `Observable` per `notify` gemeldet wird.

```
import java.util.Observer;
import java.util.Observable;

public class Zustand extends Observable{       // Zustand

  private int zaehler;                         // Zaehler

  public Zustand(int zaehler){                 // Konstruktor
    this.zaehler=zaehler;                      // initialisiere Zaehler
  }

  int get(){return zaehler;}                   // Zaehler abfragen

  void aendern(int delta){                     // Zaehlerstand aendern
    zaehler = zaehler + delta;                 // erhoehen oder erniedrigen
    setChanged();                              // notiere Aenderung
    notifyObservers();                         // Observer benachrichtigen
  }
}
```

Die Klasse `Zustand` stellt das *Model* dar und beinhaltet den aktuellen Zählerstand. Sie ist von der Klasse `Observable` abgeleitet und verfügt daher über Methoden `setChanged` und `notifyObservers`, durch die beim Observer die Methode `update` gestartet wird. Durch den Konstruktor erhält der Zähler seinen initialen Wert und mit der Methode get() kann der aktuelle Zählerstand in Erfahrung gebracht werden.

Die Klasse `KnopfKontrollierer` stellt den *Controller* dar und ist für das Eventhandlung zuständig. Dazu wird das Interface `ActionListener` implementiert, welches aus der einzigen Methode `actionPerformed` besteht. Dieser Listener wird in unterschiedlichen Instanziierungen den beiden `JButton`s übergeben. Bei einem klick auf einen Button wird dieser Event an den Listener übergeben und dort die Methode `actionPerformed` aufgerufen. Da der `JSlider` nur zu Visualisierung des aktuellen Zahlenwerts benötigt wird, muss auf benutzerdefiniertes Zerren am Slider nicht reagiert werden.

```
import java.awt.*;
import java.awt.event.*;

public class KnopfKontrollierer implements ActionListener {

  private Zustand z;                                  // Verweis auf Zustand
  private int delta;                                  // spezifischer Inkrement

  public KnopfKontrollierer(Zustand z, int delta) {
    this.z       = z;                                 // merke Zustand
    this.delta   = delta;                             // merke Inkrement
  }

  public void actionPerformed(ActionEvent e) {        // bei Knopfdruck
    z.aendern(delta);                                 // Zustand aendern
  }
}
```

Es fehlem noch die beiden Top-Level Container, welche den Rahmen für die Anwendung bilden, in dem die Instanz der Klasse `RaufRunter` ausgeführt wird.

```
import java.awt.BorderLayout;
import javax.swing.JFrame;

public class RaufRunterApp {

  public static void main(String args[]) {
    JFrame rahmen = new JFrame("RaufRunter-Applikation");
    rahmen.add(new RaufRunter(), BorderLayout.CENTER);
    rahmen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    rahmen.pack();
    rahmen.setVisible(true);
  }
}
```

In `RaufRunterApp` wird ein `JFrame` erzeugt und in dessen `ContentPane` wird die `RaufRunter`-Komponente eingefügt. Mit der Methode `setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)` wird festgelegt, dass die Applikation beim Schließen des Fensters endet, sonst würde sie weiterlaufen und wäre lediglich unsichtbar.

Mit der Methode `pack()` wird die Größe der einzelnen Componenten bestimmt und dem Top-Level Container mitgeteilt. Zuletzt muss das `JFrame` mit der Methode `setVisible` sichtbar gemacht werden.

Um die Anwendung als Applet im Browser zu betrachten wird die Klasse `RaufRunterApplet` benötigt:

```
import java.awt.BorderLayout;
import javax.swing.JApplet;

public class RaufRunterApplet extends JApplet {

  public void init() {
    add(new RaufRunter(), BorderLayout.CENTER);
  }
}
```

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

Das Applet wird dann in eine HTML-Seite eingebunden:

```
<HTML>
  <HEAD>
    <TITLE>RaufRunter-Applet</TITLE>
  </HEAD>
  <BODY>
    <CENTER>
    <P>
    <APPLET
       width=200
       height=150
       code=RaufRunterApplet.class
       archive=raufRunter.jar>
    </APPLET>
    </CENTER>
  </BODY>
</HTML>
```

Achtung: Da das Applet Swing-Klassen benutzt, kann es notwendig sein, ein aktuelles Java Plugin zu installieren.

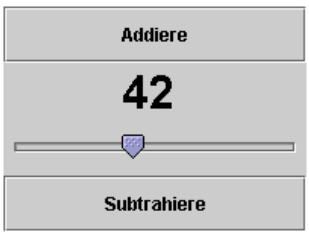


Abbildung 2.1: Screenshot vom RaufRunterApplet

Kapitel 3

2D-Grundlagen

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

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

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

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

3.1 Koordinatensysteme

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

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

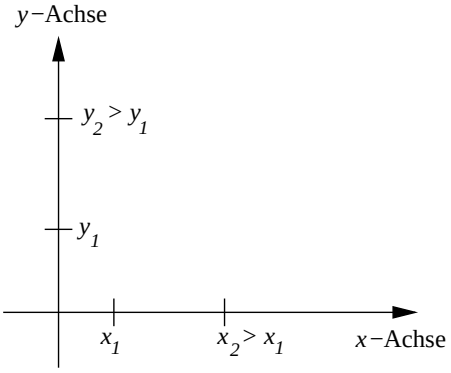
Aus Platzgründen wird stellenweise die Schreibweise als Zeilenvektor verwendet:

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

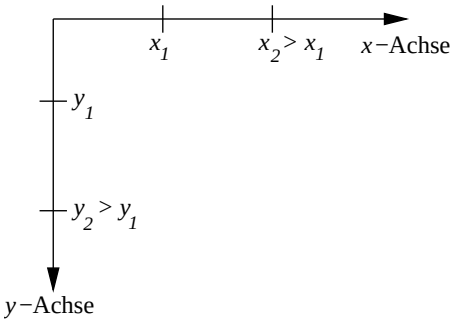
Die erste Koordinatenachse (x-Achse) wird immer von links nach rechts gezeichnet; d.h. die größeren Koordinatenwerte befinden sich weiter rechts.

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

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



Kartesisches Koordinatensystem an der Tafel



Kartesisches Koordinatensystem am Bildschirm

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

3.2 Punkt

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

$$P = (p_x, p_y)$$

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

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

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

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

mit den entsprechenden ganzzahligen Koordinaten aufrufen.

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

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

genau das, was wir erwarten.

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

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

3.3 Linie

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

3.3.1 Parametrisierte Geradengleichung

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

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

Es gilt

$$\begin{aligned} & \vec{p_1} + \vec{v} &&= \vec{p_2} \\ \Leftrightarrow \quad & \vec{v} &&= \vec{p_2} - \vec{p_1} \\ \Leftrightarrow \quad & \begin{pmatrix} v_x \\ v_y \end{pmatrix} &&= \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \end{pmatrix} \end{aligned}$$

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

$$g : \vec{u} = \vec{p_1} + r \cdot \vec{v}; r \in \mathbb{R}$$

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

$$l : \vec{u} = \vec{p_1} + r \cdot \vec{v}; r \in [0;1]$$

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

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

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

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

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

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

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

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

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

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

3.3.2 Geradengleichung als Funktion

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

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

Die Steigung s errechnet sich mit dem Steigungsdreieck wie folgt:

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

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

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

einsetzen in die Geradengleichung ergibt:

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

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

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

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

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

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

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

3.3.3 Bresenham-Algorithmus

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

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

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

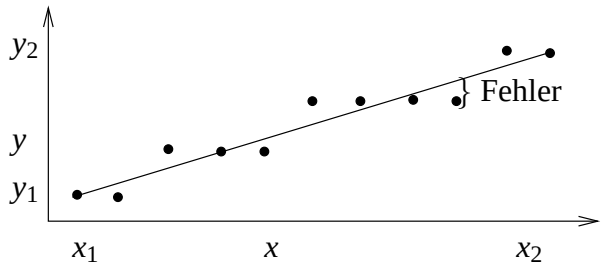


Abbildung 3.1: Fehler als Abweichung von der Ideal-Linie

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Bresenham-Algorithmus für Linien

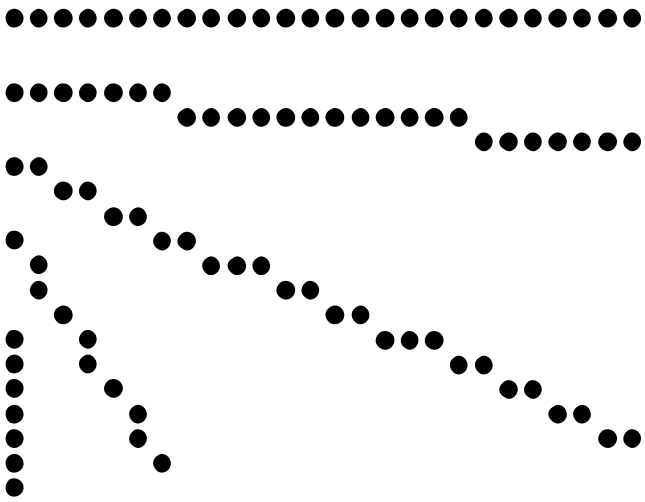


Abbildung 3.2: Vom Bresenham-Algorithmus erzeugte Linien

3.3.4 Antialiasing

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

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

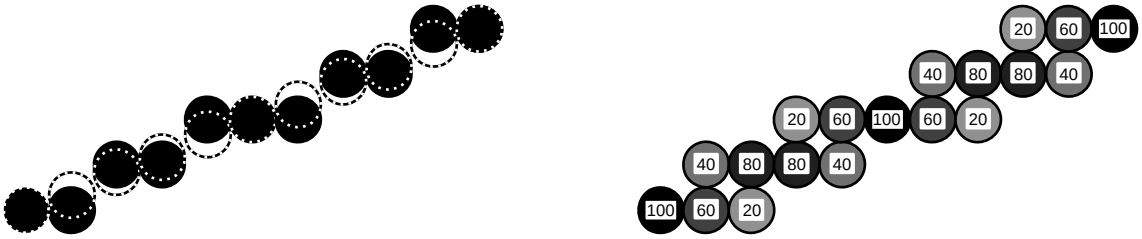


Abbildung 3.3: (a) Bresenham-Linie und Ideal-Linie (b) Resultierende Grauwerte

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

3.4 Polygon

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

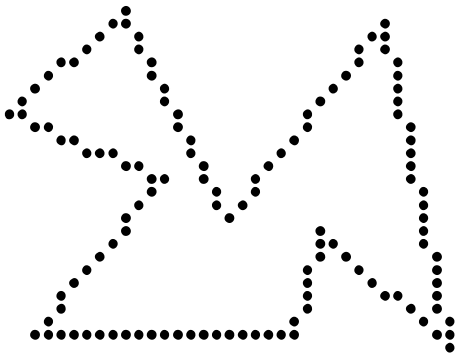
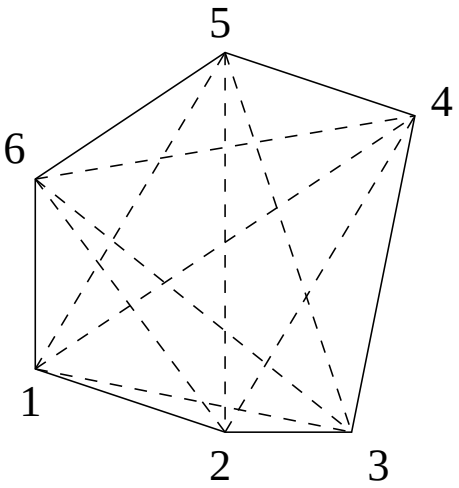


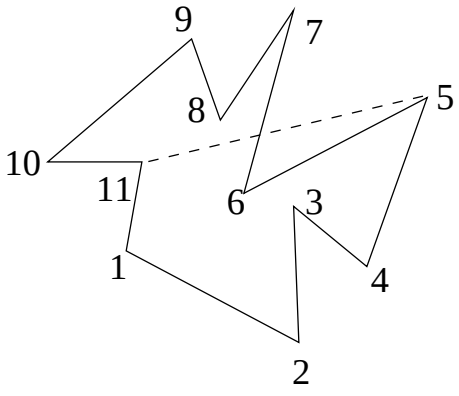
Abbildung 3.4: Polygon

3.4.1 Konvexität

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



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



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

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

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

$$\begin{aligned} g : \vec{u} &= \vec{p} + r \cdot \vec{v};\ r \in \mathbb{R} \\ \Rightarrow \begin{pmatrix} u_x \\ u_y \end{pmatrix} &= \begin{pmatrix} p_x \\ p_y \end{pmatrix} + r \cdot \begin{pmatrix} v_x \\ v_y \end{pmatrix} \\ \Rightarrow u_x &= p_x + r \cdot v_x \\ u_y &= p_y + r \cdot v_y \end{aligned}$$

Daraus ergibt sich die Funktion

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

mit der Eigenschaft:

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

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

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

3.4.2 Schwerpunkt

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

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

Für ein Dreieck gilt z.B.:

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

Und für ein Viereck gilt:

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

Bei baryzentrischen Kombinationen von Punkten gilt immer:

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

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

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

3.5 Kreis

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

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

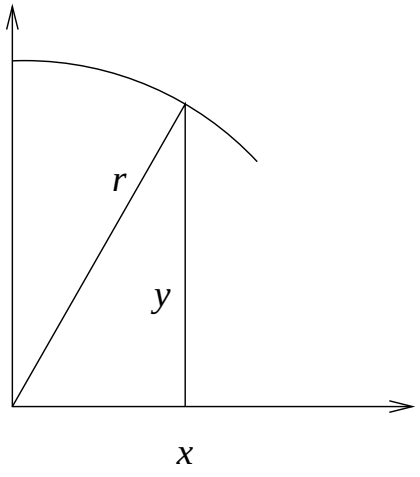


Abbildung 3.5: Verlauf des Kreises im 2. Oktanden

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

3.5.1 Trigonometrische Funktionen

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

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

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

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

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

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

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

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

```
package circle;

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

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

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

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

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

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

3.5.2 Bresenham-Algorithmus

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

Es gilt:

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

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

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

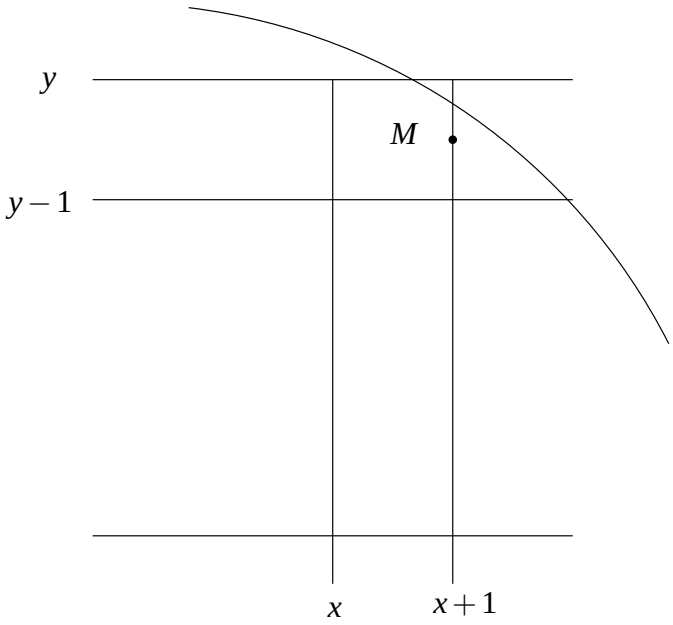


Abbildung 3.6: Testpunkt M für Entscheidungsvariable

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

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

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

Also ergibt sich:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

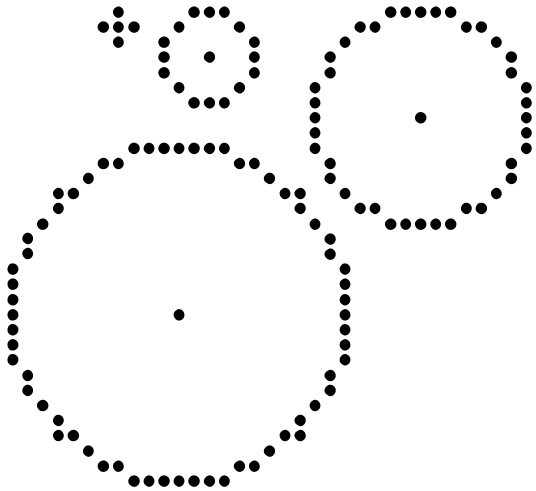


Abbildung 3.7: Vom Bresenham-Algorithmus erzeugte Kreise

3.6 Ellipse

Die beim Kreis angewendeten Techniken können mit einigen Änderungen auf die Ellipse übertragen werden. Eine Ellipse ist definiert als die Menge aller Punkte P, deren Abstandssumme zu zwei gegebenen Punkten P_1 und P_2 insgesamt $2 \cdot a$ beträgt.

Abbildung 3.8 zeigt die Beziehungen innerhalb einer Ellipse. Es gilt

$e = \sqrt{a^2 - b^2}, \quad \frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$

$x = a \cdot \cos(r), \;\; y = a \cdot \sin(r), \;\; 0 \leq r \leq 2 \cdot \pi$

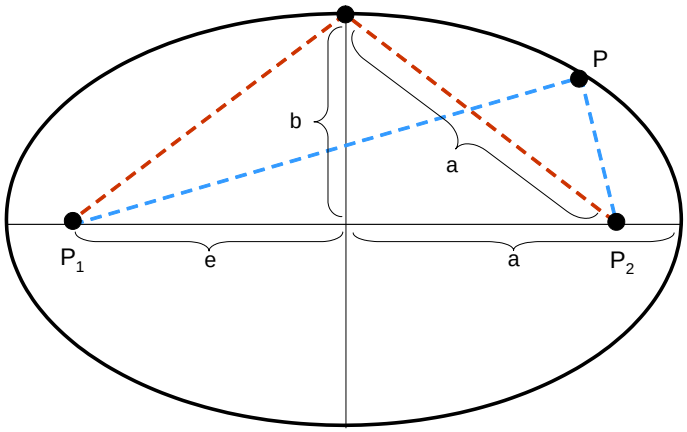


Abbildung 3.8: Beziehungen in einer Ellipse

Kapitel 4

2D-Füllen

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

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

4.1 Universelle Füll-Verfahren

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

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

Abbildung 4.1: Nachbarschaften bei universellen Füllverfahren

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

Vorteil: beliebige Umrandung möglich

Nachteil: hoher Rechen- und Speicherbedarf

Obacht:

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

Bild 4.2 zeigt beide Effekte.

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

Abbildung 4.2: Probleme bei universellen Füllverfahren

Benötigt werden

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

und

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

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

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

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

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

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

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

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

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

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

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

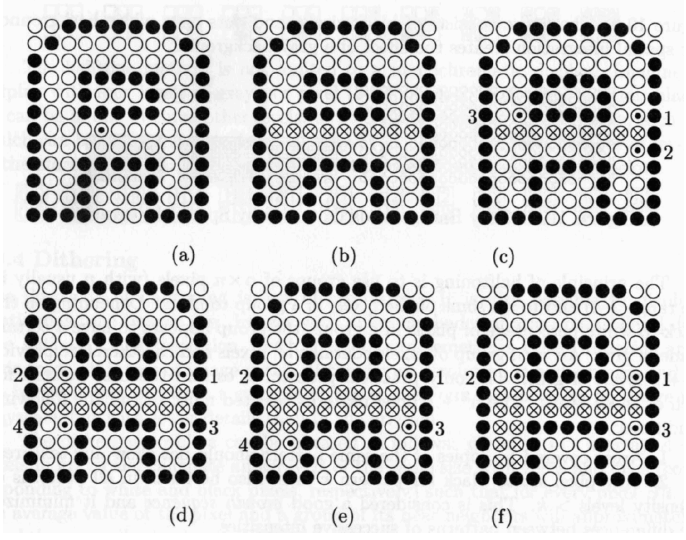


Abbildung 4.3: Beschleunigtes universelles Füllen (⊙: Saatpixel, ○: Pixel in Hintergundfarbe, ●: Randpixel, ⊗: vom Algorithmus gesetztes Pixel).

```
private void fillRowByRow(int x, int y,         // beginnend bei (x,y)
                          CGCanvas cgc) {       // fuelle eine durch die Vorder-
                                                // grundfarbe definierte Flaeche
                                                // linienweise

  int lg;                                       // linker Rand
  int rg;                                       // rechter Rand
  int px = x;                                   // Startpunkt

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

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

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

  for(x = rg; x >= lg; x--) {                   // von rechts nach links

    if(!cgc.getPixel(x, y-1)                    // falls Pixel drueber
                                                // nicht gesetzt:
      fillRowByRow(x, y-1, cgc);                // Repraesentant! Neuaufruf

    if(!cgc.getPixel(x, y+1))                   // falls Pixel drunter
                                                // nicht gesetzt:
      fillRowByRow(x, y+1, cgc);               // Repraesentant! Neuaufruf
  }
}
```

4.2 Scan-Line-Verfahren für Polygone

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

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

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

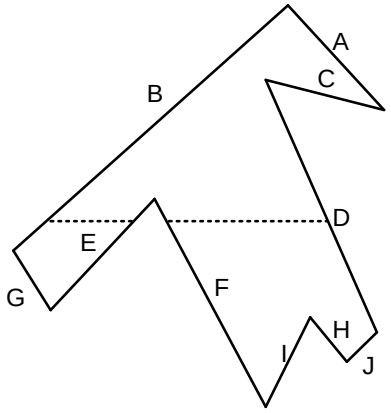


Abbildung 4.4: Polygon mit Scanline

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

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

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

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

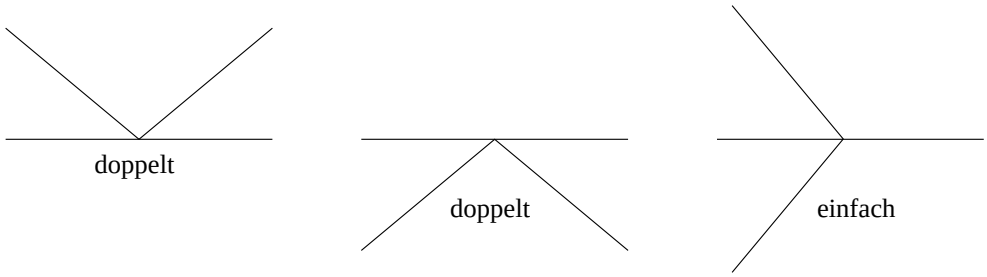


Abbildung 4.5: Fallunterscheidungen beim Berechnen der Schnittpunkte

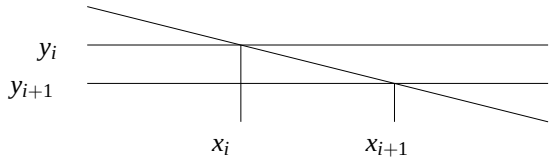


Abbildung 4.6: Fortschreiben der errechneten Schnittpunkte

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

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

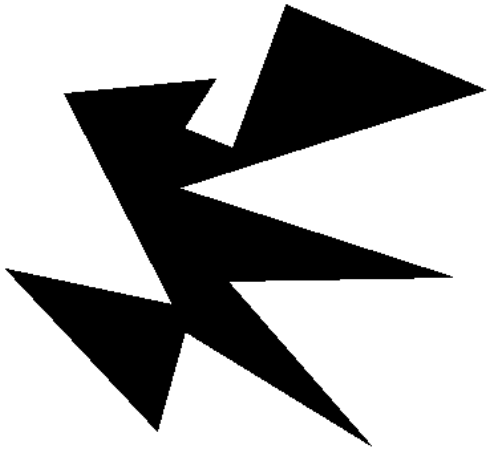


Abbildung 4.7: Vom Scanline-Algorithmus gefülltes Polygon

4.3 Dithering

Wenn einerseits die beiden ”Farben” Schwarz und Weiß zur Färbung der Objekte nicht ausreichen, andererseits der Bildschirm aber nur schwarze Pixel darstellen kann, wird wieder zu einem Trick gegriffen, bei dem das menschliche Auge betrogen wird.
Das Auge kann bei normalen Lichtverhältnissen Details unterscheiden, die mindestens eine Bogenminute ($1/60$ Grad) auseinander liegen. Wenn man in eine weiße Fläche eine Anzahl a von nicht benachbarten schwarzen Punkten hineinzeichnet und diese Fläche dann aus grösserer Entfernung betrachtet, dann ist das Auge nicht mehr in der Lage, die einzelnen Punkte voneinander zu unterscheiden. Die schwarzen und weißen Bereiche werden vom Auge integriert und dadurch entsteht der Eindruck einer grauen Fläche. Je größer a ist, desto dunkler wirkt die Fläche. Damit die Fläche gleichmäßig gefärbt erscheint, wird versucht die Punkte möglichst gleichmäßig zu verteilen. Eine Möglichkeit dies zu tun, ist das *Dithering*.

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

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

$$D_n = \begin{pmatrix} 4 \cdot D_{n-1} + 0 \cdot U_{n-1} & 4 \cdot D_{n-1} + 2 \cdot U_{n-1} \\ 4 \cdot D_{n-1} + 3 \cdot U_{n-1} & 4 \cdot D_{n-1} + 1 \cdot U_{n-1} \end{pmatrix}$$

Dabei bezeichnet U_n eine $n \times n$-Matrix, in der alle Elemente auf 1 gesetzt sind.

Es ergeben sich:

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

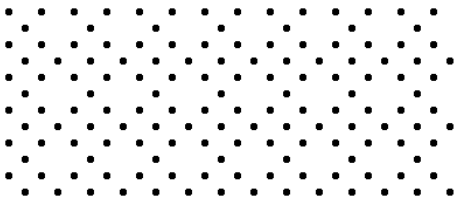


Abbildung 4.8: Zum Schwellwert 7 gehörender Grauwert

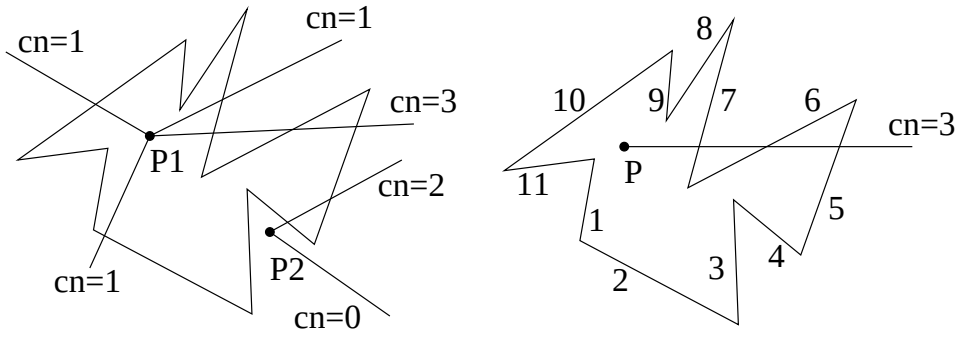
```
public class DitherMatrix extends NxNMatrix{

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

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

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

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

4.4 Punkt in Polygon

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

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

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



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

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

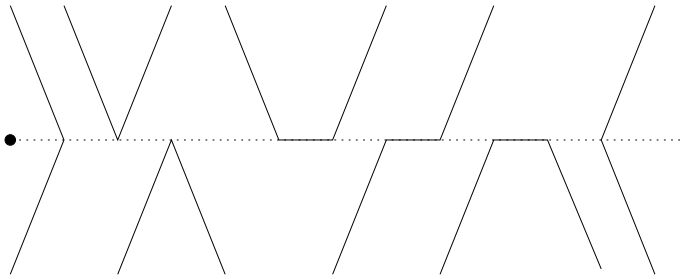
Abbildung 4.9: Ermittlung der Kreuzungszahl (cn) mit einem Teststrahl

Es muß für jede Polygonkante ermittelt werden, ob der Strahl sie schneidet. Wenn ja, wird die Kreuzungszahl um 1 erhöht, sonst nicht.

Dazu wird vom Testpunkt *P* der Strahl waagerecht nach rechts geschossen (Abbildung 4.9 b).

Um den Aufwand für die Schnittpunktberechnung auf diejenigen Kanten zu reduzieren, die den Strahl wirklich schneiden, werden zunächst einige Tests bezüglich der Koordinaten durchgeführt. Wenn Anfangs- und Endpunkt der aktuellen Polygonkante beide oberhalb oder beide unterhalb der Strahls liegen, kann die Kante den Strahl nicht schneiden (z.B. Kanten 1, 2, 3, 4, 8, 9, 11 in Abbildung 4.9 b)). Wenn ein Punkt oberhalb und der andere unterhalb liegt, kann ein Schnittpunkt vorliegen und die *x*-Koordinaten werden untersucht. Sind beide Punkte rechts von *P*, schneidet der Strahl die Kante irgendwo (diese Information ist ausreichend) und die Kreuzungszahl wird erhöht (Kanten 7, 6, 5). Sind beide links von *P* gibt es keinen Schnittpunkt. Ist einer links und einer rechts von *P* muß berechnet werden, wo die Kante den *y*-Wert von *P* erreicht; d.h. ob der Schnittpunkt rechts von *P* liegt (z.B. Kante 7). Wenn ja, wird die Kreuzungszahl erhöht.

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



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

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

Damit ergibt sich folgender Algorithmus:

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

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

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

  return inside;
}
```

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

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

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

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

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

Kapitel 5

2D-Clipping

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

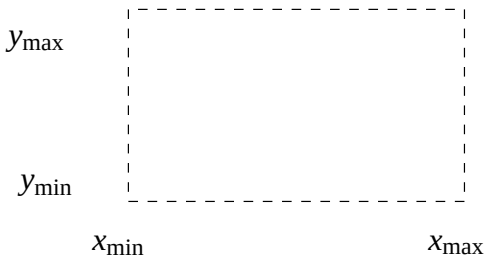


Abbildung 5.1: Clip-Fenster

5.1 Clipping von Linien

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

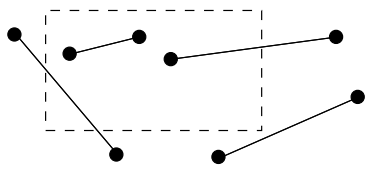


Abbildung 5.2: Ausgangslage beim Linien-Clipping

Idee von Cohen & Sutherland

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

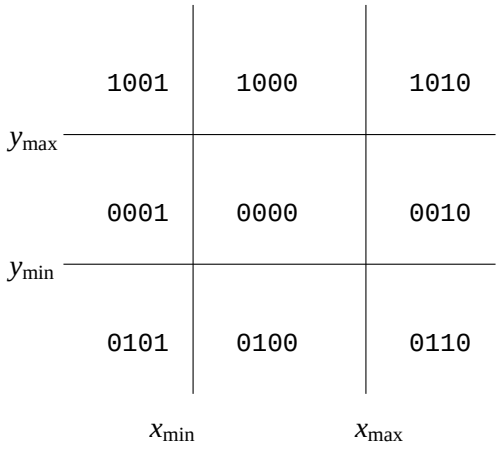


Abbildung 5.3: Bereichscodes nach Cohen & Sutherland

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

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

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

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

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

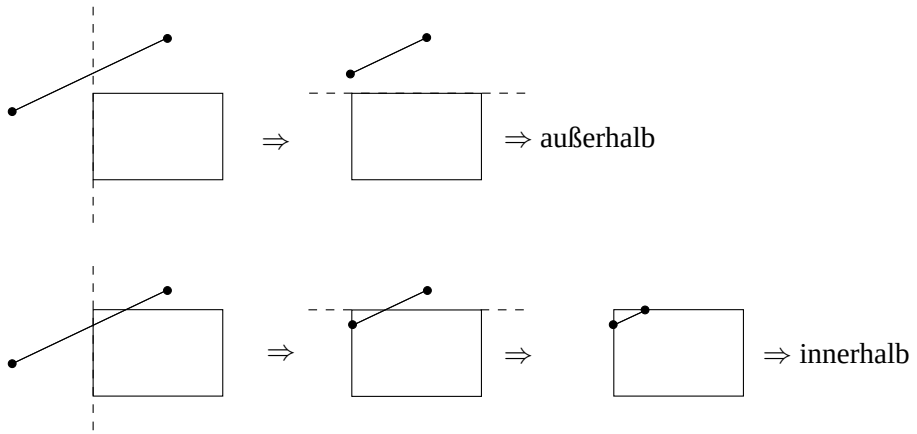


Abbildung 5.4: Möglichkeiten bei zwei außerhalb liegenden Punkten

```
/*****************************************************************************/
/*                                                                           */
/*                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:

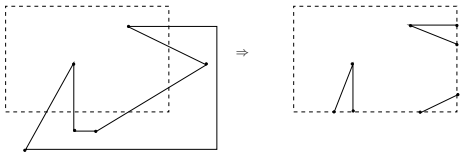


Abbildung 5.5: Zerfall eines Polygons bei reinem Linien-Clippen

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

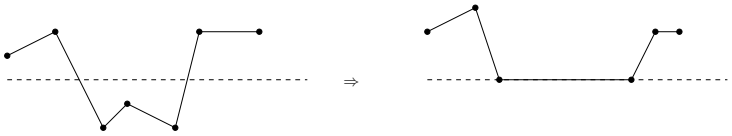


Abbildung 5.6: Verbinden der Ein- und Austrittspunkte

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

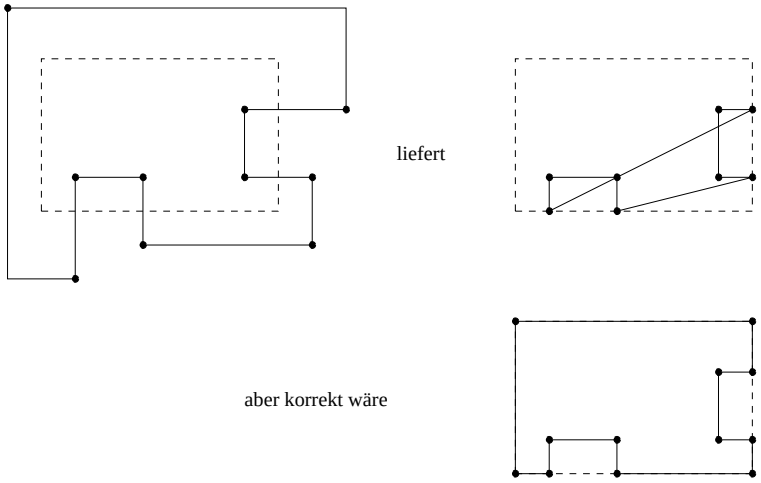


Abbildung 5.7: Spezialbehandlung an den Clipfensterecken

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

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

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

public boolean on_Visible_Side(              // Liefert true, wenn der Punkt p
                            Point p,      // bzgl. der waage- oder senkrechten
                            int value,    // Kante beim entspr.  y- oder x-Wert
                            int edge) {   // value auf der sichtb. Seite liegt.
                                          // edge symbolisiert die Lage der
                                          // Kante analog zum Regioncode.

    switch (edge) {
       case LEFT   : return p.x >= value;  // linke  Fenster-Kante
       case RIGHT  : return p.x <= value;  // rechte Fenster-Kante
       case TOP    : return p.y >= value;  // obere  Fenster-Kante
       case BOTTOM : return p.y <= value;  // untere Fenster-Kante
     }

     return false; // sonst: draussen
}


private Point intersection(               // liefert true, falls sich die Linie
                     Point p1, Point p2,  // p1-p2 mit waage- oder senkrechter
                     int value,           // Clipping-Kante bei y- oder x-Wert
                     int edge) {          // value schneidet. edge ist die
                                          // Clipping Kante relativ zum
                                          // Clipping-Rechteck

    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

    Point i = null;                       // liefert den Schnittpunkt i zurueck

    p1_vis=on_Visible_Side(p1,value,edge); // p1 sichtbar ?
    p2_vis=on_Visible_Side(p2,value,edge); // p2 sichtbar ?

    if ((p1_vis && p2_vis) || (!p1_vis && !p2_vis))
        return i;                         // kein Schnittpkt

    else {
        slope = (double)(p2.y-p1.y) / (double) (p2.x-p1.x);   // Steigung

        if (edge==TOP || edge == BOTTOM) {  // waagerechte Kante
            i.x = (int)((value-p1.y) / slope) + p1.x;
            i.y = value;
        }
        else {                            // senkrechte Kante
            i.x = value;
            i.y = (int)((value-p1.x) * slope) + p1.y;
        }
        return i;                        // Schnittpunkt liefern
    }
}
```

```
private boolean sutherland_hodgeman(Vector pv, int value, int edge) {
    int old; // durchlaeuft die alten Punkte
    Vector hilf = new Vector(pv.size(), 1); // Hilfsvektor
    Point s, i;

    for (old = 0; old < pv.size() - 1; old++) {
        s = (Point) pv.elementAt(old);
        if (on_Visible_Side(s, value, edge)) {
            hilf.addElement(s);
        }

        i = l.intersects(s, pv.elemenatAt(old+1), value, edge);
        if (i != null) {
            hilf.addElement(i);
         }
     }
     if (hilf.size() > 0) {                  // Falls Punkte gemerkt
         hilf.addElement(hilf.elementAt(0)); // 1. nochmal ans Ende
     }
     pv = hilf;                              // neuen Vektor merken
     return hilf.size() != 0;                // wenn draussen: Vektor leer
}
```

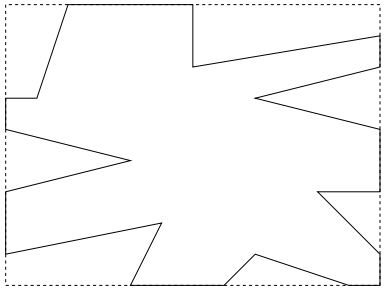


Abbildung 5.8: Vom Sutherland-Hodgman-Algorithmus geclipptes Polygon

5.3 Java-Applet zu 2D-Operationen

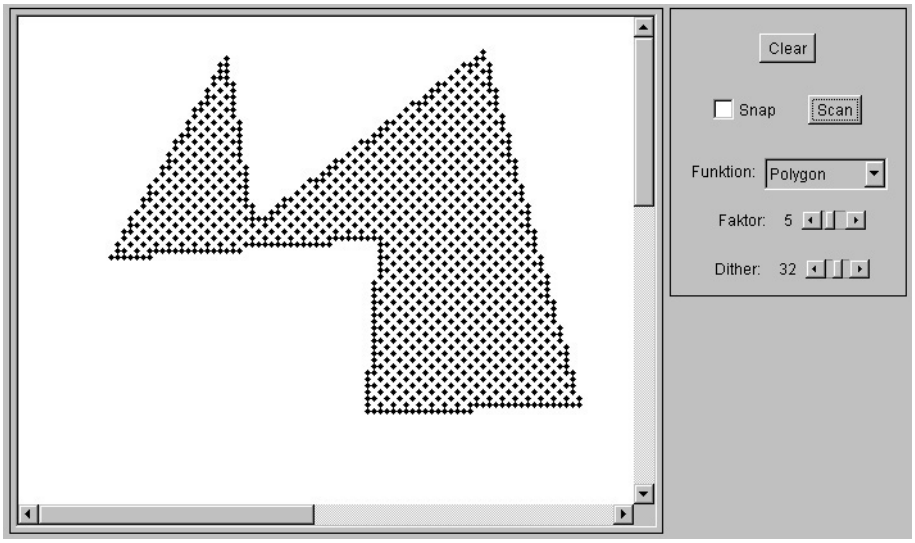


Abbildung 5.9: Screenshot vom 2D-Basic-Applet

Kapitel 6

2D-Transformationen

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

6.1 Translation

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

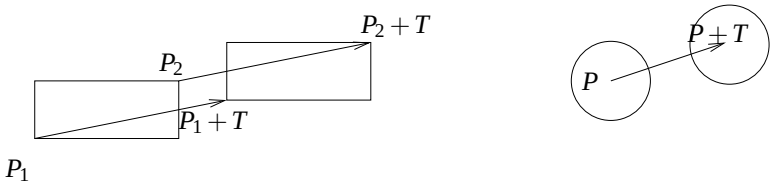


Abbildung 6.1: Translation von Rechtecken und Kreisen

6.2 Skalierung

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

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

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

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

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

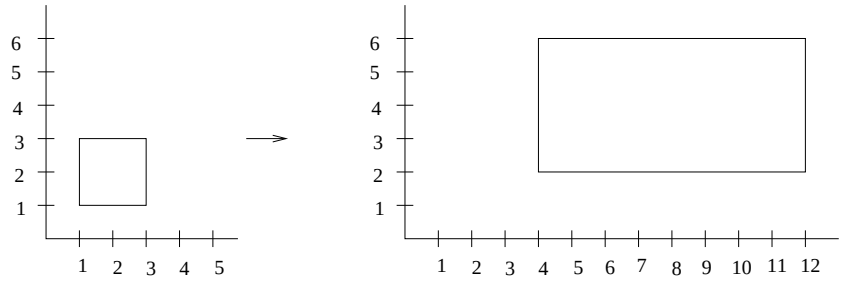


Abbildung 6.2: Skalierung mit $s_x = 4$ und $s_y = 2$

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

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

Die neuen Koordiaten berechnen sich dann wie folgt:

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

Vorteilhafter bei mehreren Objekten:

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

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

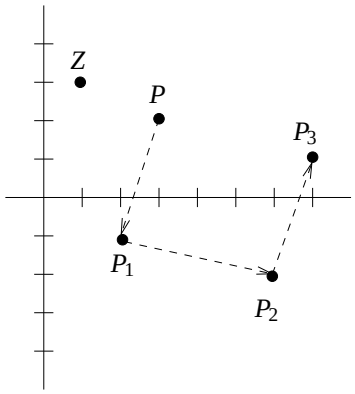
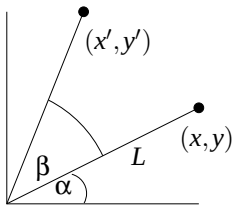


Abbildung 6.3: Skalierung bzgl. Punkt (1,3) mit Faktoren $s_x = 3$ und $s_y = 2$

6.3 Rotation

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



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

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

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

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

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

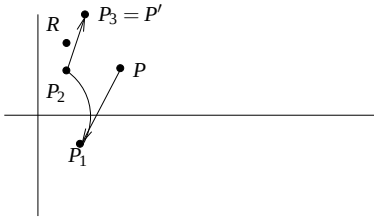


Abbildung 6.5: Rotation des Punktes P um 90 Grad bzgl Punkt R

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

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

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

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

Eine Skalierung sieht dann z.B. folgendermaßen aus:

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

Eine Rotation hat folgendes Aussehen:

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

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

$$A \cdot B = (B^T \cdot A^T)^T$$

6.5 Homogene Koordinaten

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

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

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

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

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

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

Translation

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

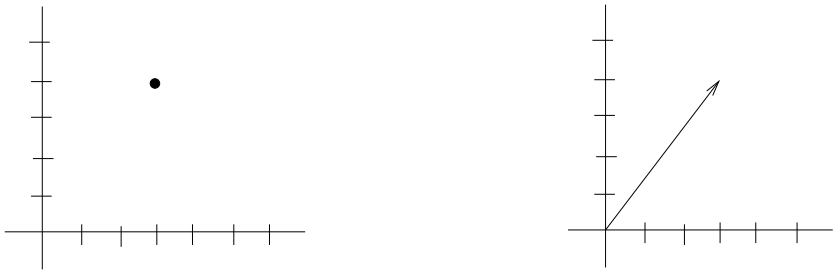


Abbildung 6.6: Punkt $(3,4)$ und Richtungsvektor $(3,4)^T$

Skalierung

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

Rotation

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

Für die Auswertung einer zusammengesetzten Transformation

- gilt das Assoziativgesetz, d.h. $A \cdot B \cdot C = (A \cdot B) \cdot C = A \cdot (B \cdot C)$,
- gilt im allgemeinen nicht das Kommutativgesetz, d.h. $A \cdot B \neq B \cdot A$.
  Z.B. ist eine Translation um $(5,8)$ mit anschließender Rotation um 90° verschieden von Rotation um 90° mit anschließender Translation um $(5,8)$.

Beispiel für zusammengesetzte Transformation: Rotation bzgl. $(3,5)$ um 60°

Matrix für Translation um $(-3,-5)$ lautet

$$A = \begin{pmatrix} 1.0000000 & 0.0000000 & -3.0000000 \\ 0.0000000 & 1.0000000 & -5.0000000 \\ 0.0000000 & 0.0000000 & 1.0000000 \end{pmatrix}$$

Matrix für Rotation um 60° lautet

$$B = \begin{pmatrix} 0.5000000 & -0.8660254 & 0.0000000 \\ 0.8660254 & 0.5000000 & 0.0000000 \\ 0.0000000 & 0.0000000 & 1.0000000 \end{pmatrix}$$

Matrix für Translation um $(3,5)$ lautet

$$C = \begin{pmatrix} 1.0000000 & 0.0000000 & 3.0000000 \\ 0.0000000 & 1.0000000 & 5.0000000 \\ 0.0000000 & 0.0000000 & 1.0000000 \end{pmatrix}$$

Matrix für gesamte Transformation lautet

$$D = C \cdot B \cdot A = \begin{pmatrix} 0.5000000 & -0.8660254 & 5.8301270 \\ 0.8660254 & 0.5000000 & -0.0980762 \\ 0.0000000 & 0.0000000 & 1.0000000 \end{pmatrix}$$

6.6 Allgemeine Transformationen

Weitere Transformationen, die sich durch Matrizen darstellen lassen, sind

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

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

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

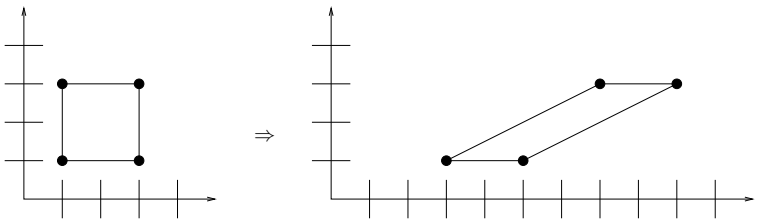


Abbildung 6.7: Scherung mit $Sch_x = 2$

6.7 Raster-Transformationen

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

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

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

```
zeichne Objekt mit XOR
repeat
   loesche Objekt mit XOR
```

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

Allerdings ändert das bewegte Objekt je nach Hintergrundfarbe sein Aussehen.

6.8 Java-Applet zu 2D-Transformationen

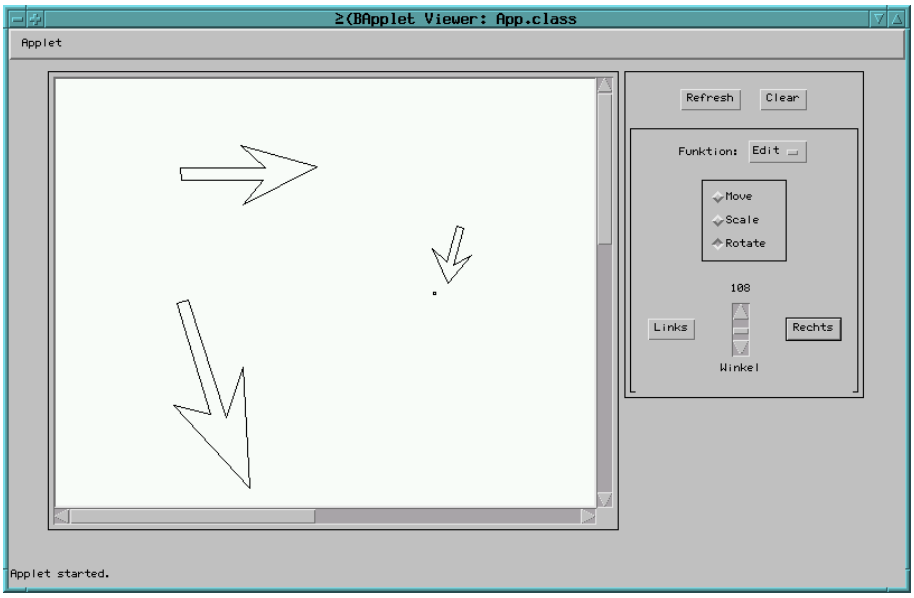


Abbildung 6.8: Screenshot vom 2D-Trafo-Applet

Kapitel 7

Kurven

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

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

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

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

7.1 Algebraischer Ansatz

Gegeben $n+1$ Stützpunkte.
Bestimme Polynom n-ten Grades

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

welches durch alle Stützpunkte läuft.
Problem: Bei der Auswertung des Polynoms treten wegen der hohen Potenzen hohe Rechenzeiten, große Rundungsfehler und Oszillationen auf.

7.2 Kubische Splines

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

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

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

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

Gesucht: Approximation einer Kurve $f(t)$, die durch $n+1$ vorgegebene Punkte laufen soll mithilfe von n Kurvenabschnitten.

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

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

Jedes f_i hat die Form

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

Es muß gelten

$f_i(t_i)$	$=$	$f_{i+1}(t_i)$	für	$i = 1, \ldots, n-1$	gleicher Wert
$f_i'(t_i)$	$=$	$f_{i+1}'(t_i)$	für	$i = 1, \ldots, n-1$	gleiche Steigung
$f_i''(t_i)$	$=$	$f_{i+1}''(t_i)$	für	$i = 1, \ldots, n-1$	gleiche Steigungsänderung

Durch Festlegung von $f''(t_0) = f''(t_n) = 0$ wird das Gleichungssystem abgeschlossen.

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 $\frac{1}{3} \cdot (|dx| + |dy| + 2 \cdot \max(|dx|, |dy|))$.

7.3 Bézier-Kurven

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

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

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

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

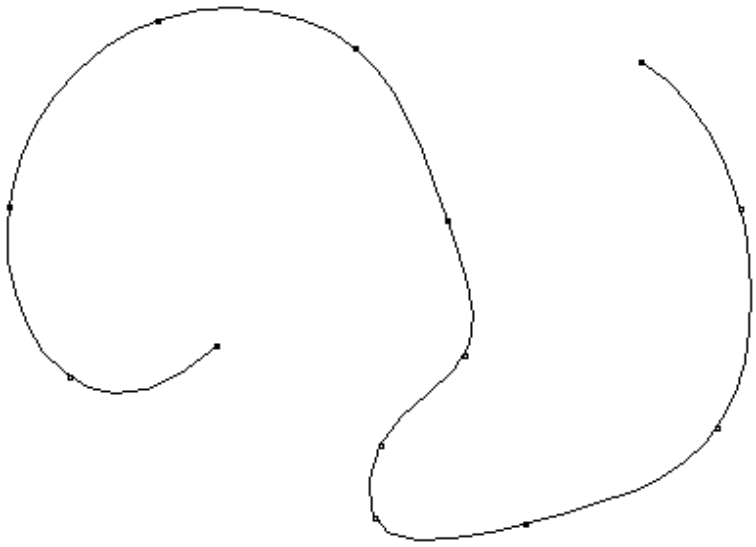


Abbildung 7.1: Vom Spline-Algorithmus erzeugte Kurveninterpolation

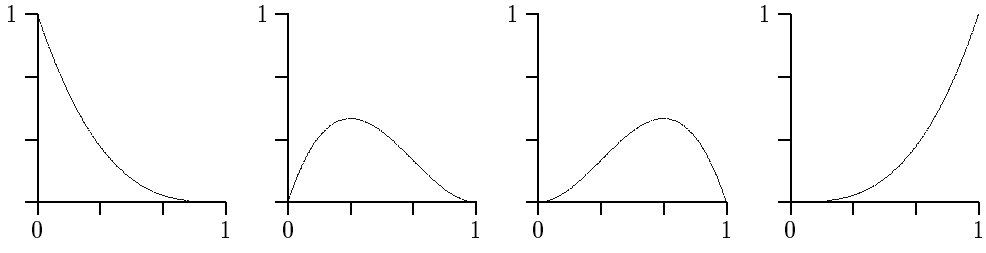


Abbildung 7.2: Verlauf der Bernsteinpolynome für n=3

Die wichtigsten Eigenschaften der Bernsteinpolynome:

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

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

Die Kurve beginnt im Stützpunkt P_0 tangential zur Geraden $\overline{P_0P_1}$, endet im Stützpunkt P_n tangential zur Geraden $\overline{P_{n-1}P_n}$ und verläuft innerhalb der konvexen Hülle der Stützpunkte. Somit können

mehrere Bézier-Kurven aneinandergesetzt werden durch Identifikation von Endpunkt und Anfangspunkt aufeinanderfolgender Bézierkurven. Einen stetig differenzierbaren Übergang erreicht man bei Kollinearität der Punkte $P_{n-1}, P_n = Q_0, Q_1$.

Berechnung der Bézier-Kurve nach de Casteljau

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

Es gilt:

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

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

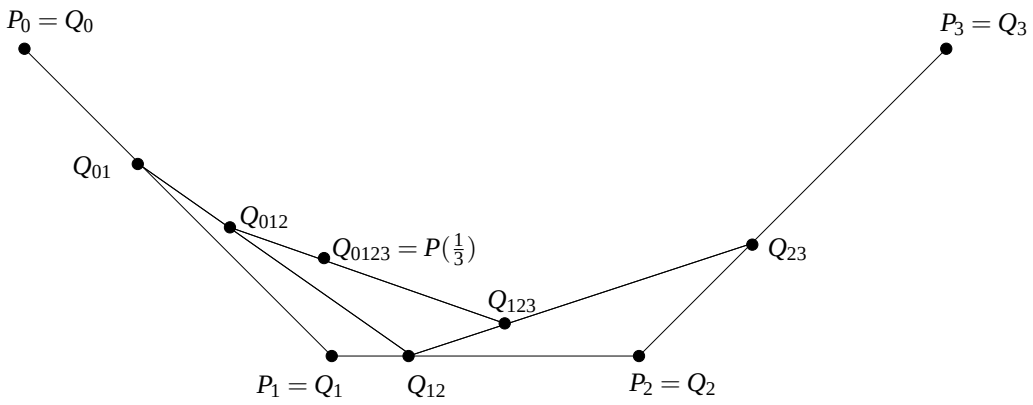


Abbildung 7.3: Approximation einer Bezier-Kurve nach de Casteljau

Abbildung 7.3 zeigt die Berechnung eines Kurvenpunktes für $t = \frac{1}{3}$. Die Indizes geben an, welche Stützpunkte beteiligt sind. Abbildung 7.4 zeigt das Ergebnis dieser Iteration für zwei aneinanderliegende kubische Bézierkurven (Rekursionstiefe 3).

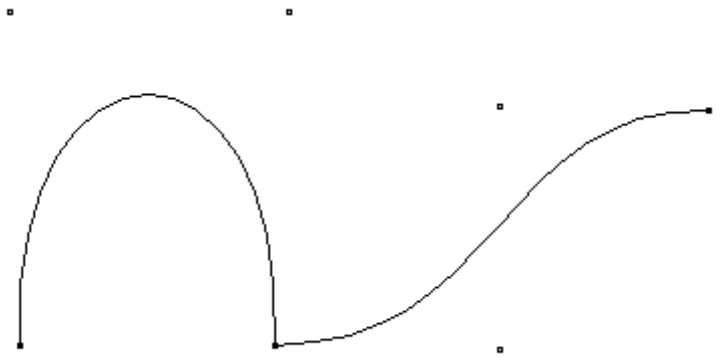


Abbildung 7.4: Vom De Casteljau-Algorithmus mit Rekursionstiefe 3 gezeichnete Bezierkurven

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

  if (depth == 0)                               // Iterationstiefe erreicht
    drawLine(new Point( (int)px0, (int)py0),    // Linie zeichnen
             new Point( (int)px3, (int)py3));
    else {
      depth--;

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

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

7.4 B-Splines

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

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

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

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

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

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

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

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

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

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

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

Sonderfall

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

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

Die zugehörigen B-Splinefunktionen haben die schon bekannte Form

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

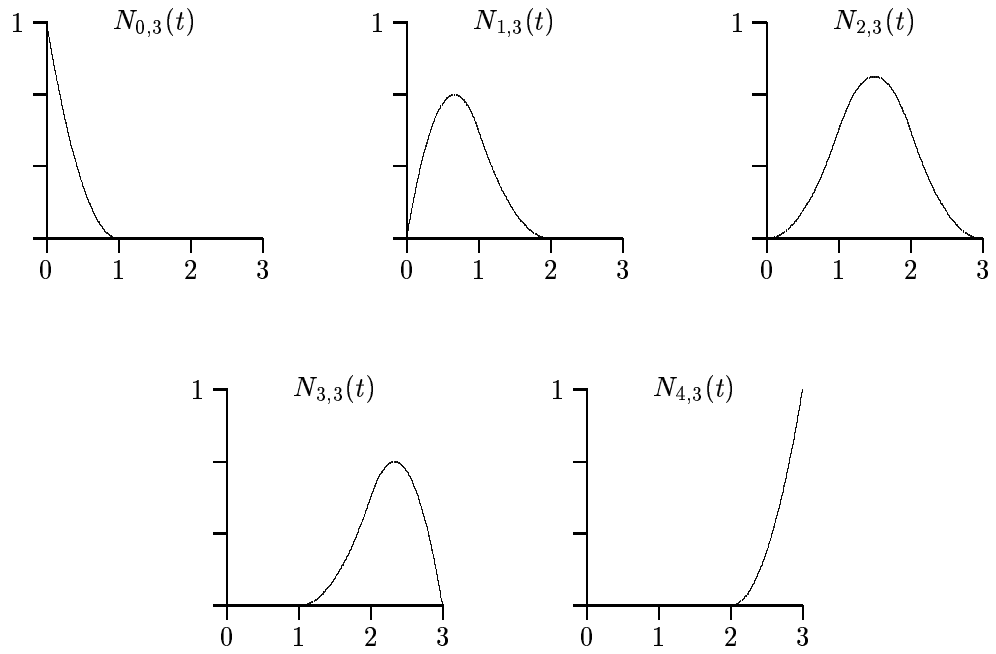


Abbildung 7.5: Verlauf der Gewichtungspolynome $N_{i,k}$ für $k = 3$

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

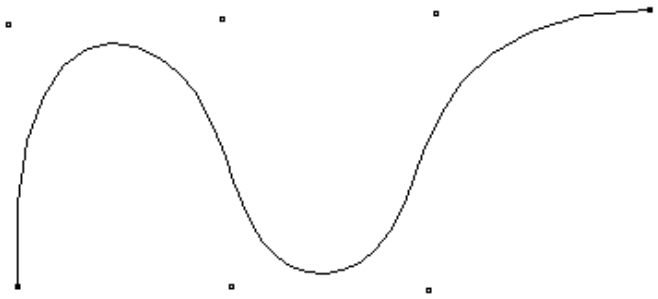


Abbildung 7.6: B-Spline mit $n = 6$, $k = 4$ und 35 Interpolationspunkten

Eigenschaften

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

- Sie verlaufen innerhalb des aus den Stützpunkten gebildeten konvexen Polygons (*konvexe Hülle*.

- Sie sind invariant unter *affinen Abbildungen*.

Konvexe Hülle

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

7.5 Affine Abbildungen und Invarianz

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

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

Es handelt sich um eineindeutig umkehrbare Abbildungen, bei denen

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

erhalten bleiben. Wogegen

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

sich ändern können.

Alle bisher in der Vorlesung behandelten Transformationen sind affine Abbildungen.

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

7.6 NURBS

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

Um möglichst große Freiheit bei den darzustellenden Formen zu haben, sollten die Kurven beliebige Kegelschnitte (also Kreise, Parabeln und Hyperbeln) annähern können, wie sie in Abbildung 7.7 gezeigt werden.

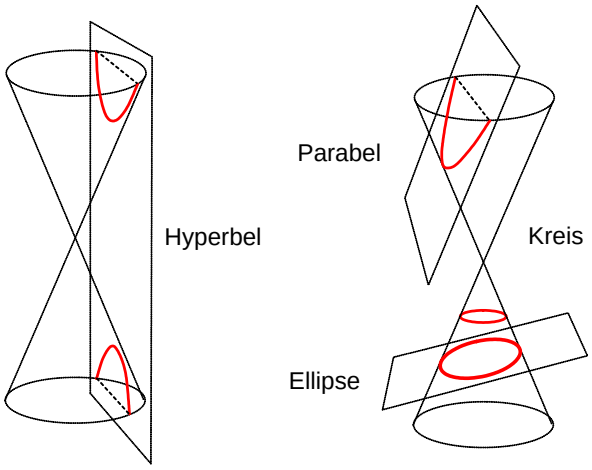


Abbildung 7.7: Kreis, Ellipse, Parabel und Hyperbel erzeugt durch Kegelschnitte

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

$$P(t) = \sum_{i=0}^{n} R_{i,k}(t) \cdot P_i \quad \text{mit}$$

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

Nonuniform bedeutet, dass die Knoten auf dem Knotenvektor nicht äquidistant sein müssen; *rational* bedeutet, dass die Gewichtung eines Kontrollpunktes durch den Quotienten zweier Polynome definiert wird.

Abbildung 7.8 zeigt den Einfluss der Gewichte auf den Verlauf der Kurve. Ein Gewicht $h_2 > 1$ verschiebt die Kurve in Richtung Kontrollpunkt P_2, ein Gewicht $h_2 < 1$ verringert den Einfluss von Kontrollpunkt P_2 (und erhöht dadurch den Einfluss von Kontrollpunkt P_1.

Zur Berechnung von NURBS-Kurven bedient man sich der homogenen Koordinaten: Zunächst wird ein Punkt $P_i(x_i, y_i)^T$ in seine homogenen Koordinaten $P(x_i, y_i, 1)^T$ überführt und dann mit seinem Gewicht h_i multipliziert. Die so entstandenen Punkte $P_i'(w_i \cdot x_i, w_i \cdot y_i, h_i)^T$ werden nun mit den Gewichtspolynomen $N_{i,k}$ verarbeitet: $P'(t) = \sum_{i=0}^{n} N_{i,k}(t) \cdot P_i'$

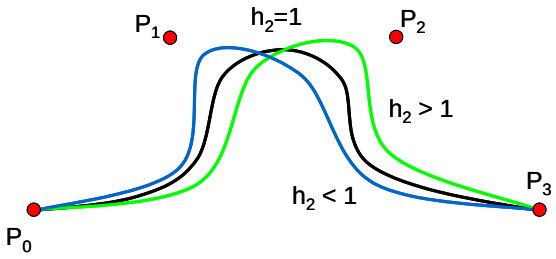


Abbildung 7.8: Einfluss des Gewichtes auf den Verlauf der Kurve

Zur Anzeige der errechneten Kurve werden die homogenen Koordinaten durch Division der dritten Komponente in Punktkoordinaten überführt:

$$(x,y,z)^T \rightarrow (\frac{x}{z},\frac{y}{z})^T$$

NURBS bieten gegenüber B-Splines einige Vorteile:

- NURBS sind eine Verallgemeinerung der B-Splines. Für $h_i = 1 \ \ \forall i$ reduziert sich die NURBS-Kurve zur entsprechenden B-Spline-Kurve.

- NURBS sind invariant bzgl. perspektivischer Projektion. Dies bedeutet, daß zur Projektion einer Kurve nicht alle Kurvenpunkte projeziert werden müssen, sondern es reicht, die Stützpunkte zu projezieren und dann zu einer Kurve zu verbinden.

- NURBS sind (im Gegensatz zu nicht-rationalen *B-Splines*) in der Lage, Kreise zu beschreiben. Abbildung 7.9 zeigt die 8 Kontrollpunkte zusammen mit ihren Gewichten für den Einheitskreis mit Radius 1. Der zugehörige Knotenvektor lautet (0,0,1,1,2,2,3,3,4,4), wobei der erste und der letzte Kontrollpunkt jeweils doppelt benutzt werden.

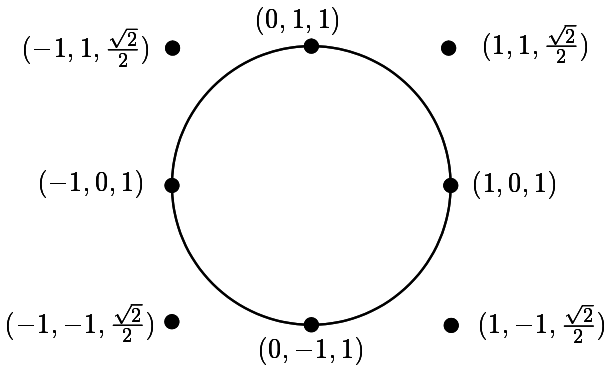


Abbildung 7.9: Kreis erzeugt durch NURBS

7.7 Java-Applet zu Splines

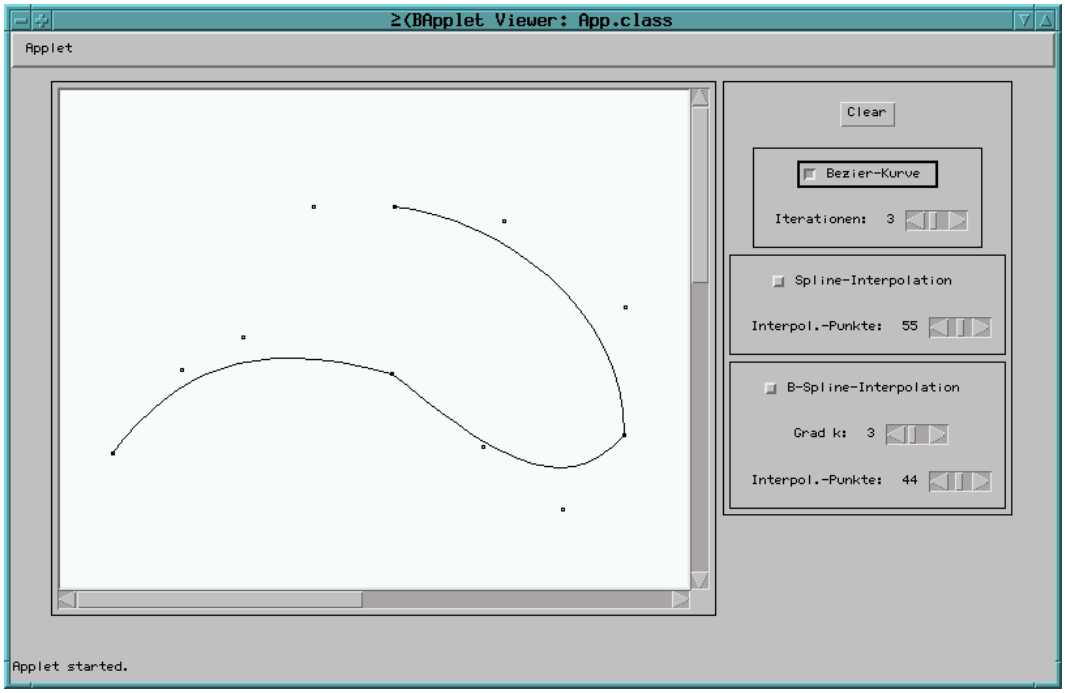


Abbildung 7.10: Screenshot vom Splines-Applet

Kapitel 8

Farbe

8.1 Physik

Ein Teil des elektromagnetischen Spektrums wird vom Auge wahrgenommen:

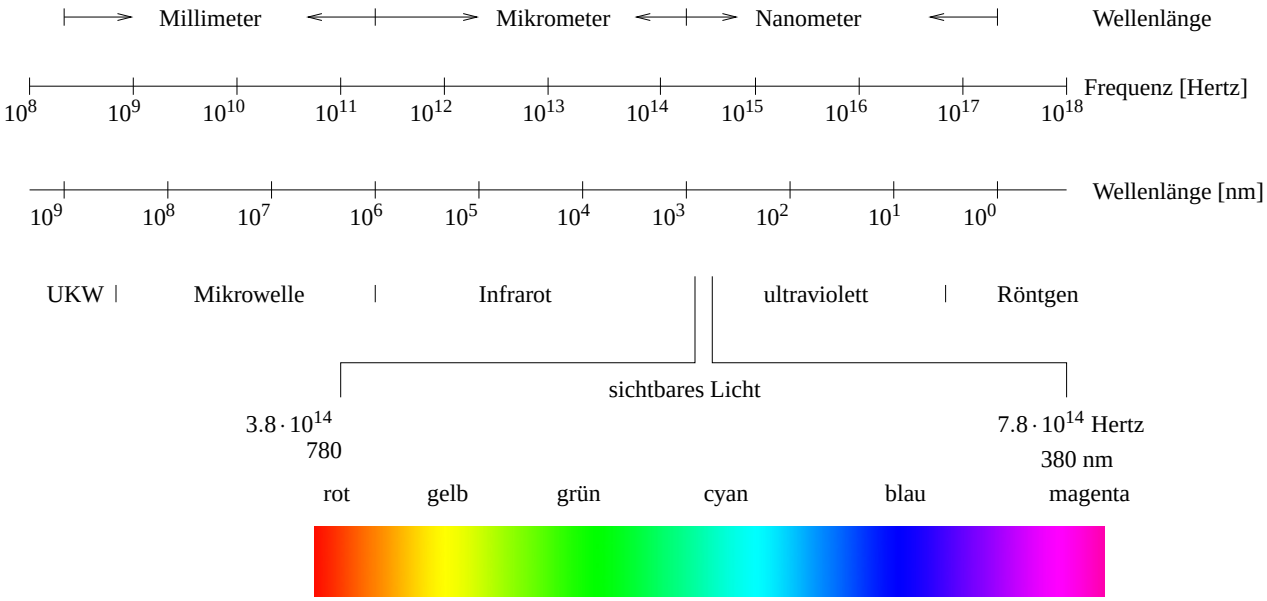


Abbildung 8.1: Elekromagnetisches Spektrum

Es gilt:

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

8.2 Dominante Wellenlänge

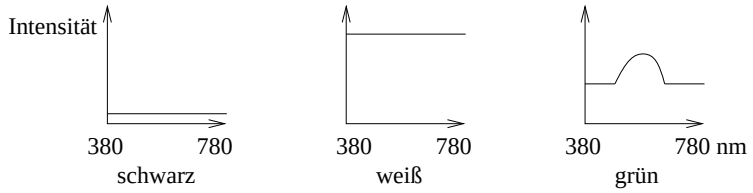


Abbildung 8.2: Spektren zu drei Farben

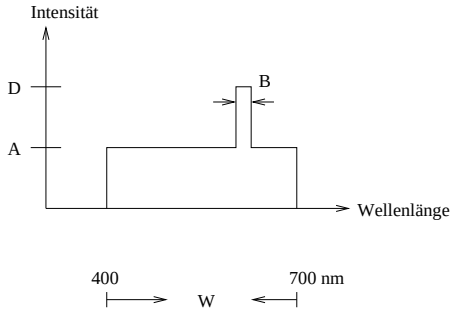


Abbildung 8.3: Dominante Wellenlänge

Eine mögliche intuitive Charakterisierung für den Farbeindruck lautet:

hue: Farbton, gegeben durch dominante Wellenlänge.

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

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

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

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

8.3 Grundfarben

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

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

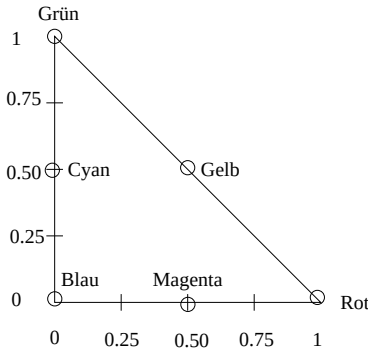


Abbildung 8.4: 2-dimensionale Beschreibung von Farbtönen in baryzentrischen Koordinaten.

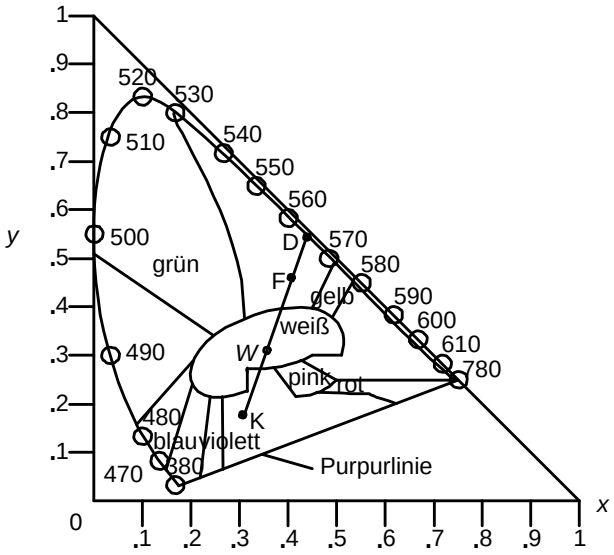


Abbildung 8.5: CIE-Farbdiagramm des sichtbaren Spektrum, Angaben zur Wellenlänge in nm

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

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

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

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

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

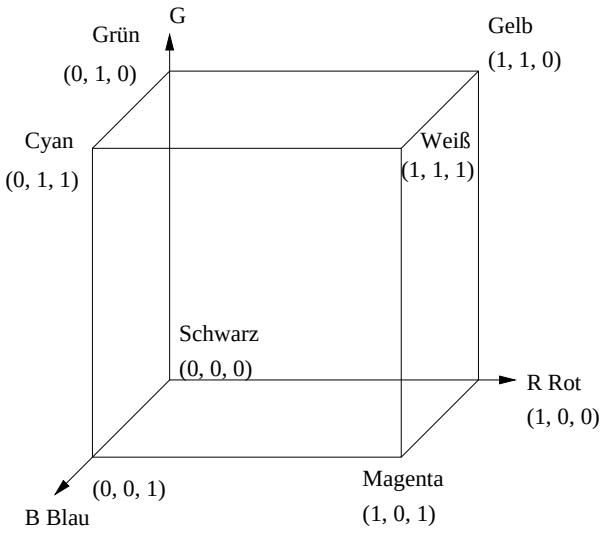


Abbildung 8.6: RGB-Würfel

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

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

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

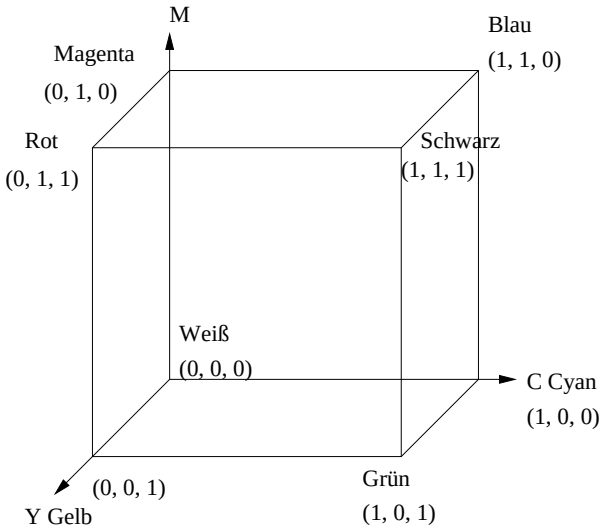


Abbildung 8.7: CMY-Würfel

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

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

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

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

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

8.6 YUV-Modell

Ein Farbwert wird beschrieben durch ein YUV-Tripel, wobei Y die Helligkeit (Luminanz) bezeichnet und U,V Farbdifferenzen (Chrominanz). Die Helligkeitsempfindung resultiert zu 59 % aus dem Grünanteil, zu 30 % aus dem Rotanteil und zu 11 % aus dem Blauanteil:

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

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

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

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

8.7 YIQ-Modell

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

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

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

Beim europäischen *PAL-System* (Phase Alternating Line) werden statt der Parameter I und Q die Farbdifferenzen $R-Y$ und $B-Y$ übertragen. Die Konvertierung der Farbinformation in Monochrom-Darstellung erfolgt in beiden Systemen durch Auswertung des Helligkeitsparameters Y.

8.8 HSV-Modell

DAs HSV-Modell, beschreibt jede Farbe durch folgendes intuitive Tripel:

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

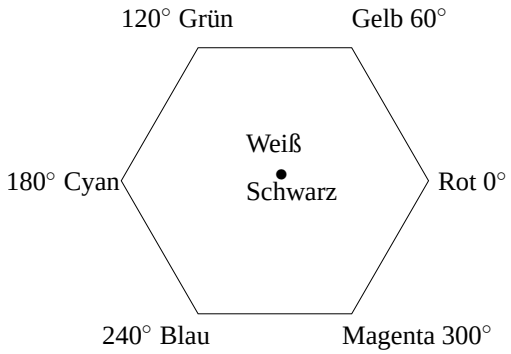


Abbildung 8.8: Gradeinteilung für Farbtöne im HSV-Modell

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

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

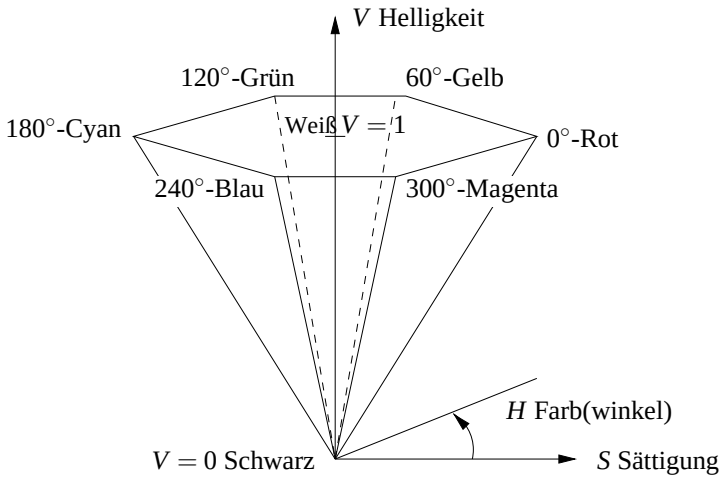


Abbildung 8.9: HSV-Modell

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

Umrechnung von *RGB* nach *HSV*

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

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

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

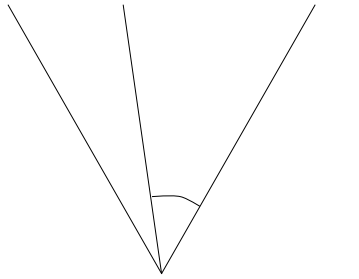
$$\begin{array}{lcllcl} v & = & \max(r,g,b) & = & \frac{1}{2} & = & 50\,\% \\ mi & := & \min(r,g,b) & = & \frac{1}{8} & & \\ s & = & \dfrac{v-mi}{v} & = & \frac{\frac{3}{8}}{\frac{1}{2}}=\frac{3}{4} & = & 75\,\% \end{array}$$

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

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

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

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



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

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

8.9 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*.

8.10 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
```

8.11 Java-Applet zu Farbe

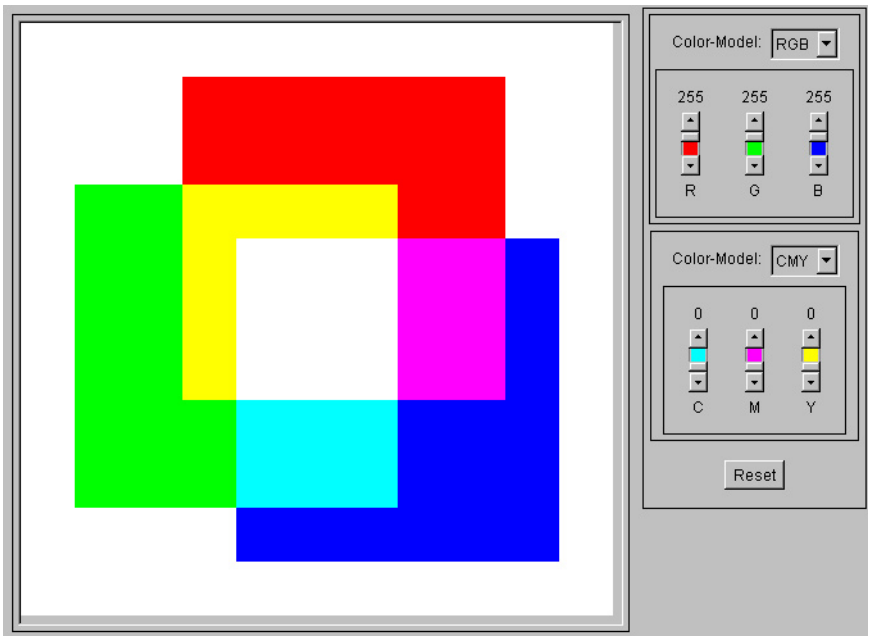


Abbildung 8.10: Screenshot vom Farben-Applet

Kapitel 9

Pixeldateien

In diesem Kapitel geht es um die Beschreibung von Bildern, die durch eine Menge von Farbpixeln definiert sind. Neben den wichtigsten Komprimierungsverfahren GIF und JPEG werden auch einige Dateiformate wie z.B. TIF und PPM vorgestellt.

9.1 Auflösung

Für die Qualität eines Bildes sind die am Erstellungsprozeß beteiligten Auflösungen verantwortlich. Die Auflösung wird meistens gemessen als *Dots per Inch* (dpi).

Scanner-Auflösung:	Gegeben durch Anzahl der CCD-Elemente in einer Sensorzeile (z.B. 300 dpi).
Scan-Auflösung:	Gewählte Auflösung beim Scanner.
Bildauflösung:	Entspricht zunächst der Scan-Auflösung, kann später ggf. runtergerechnet werden (durch Mittelung) oder hochgerechnet (durch Interpolation).
Bildschirmauflösung:	Ein 17-Zoll Monitor mit Seiten-Verhältnis 4:3 hat eine Breite von $(17 \cdot 4)/5 = 13.6$ inch. Bei 1024×768 Pixeln ergibt das $1024/13.6 \approx 75$ dpi.
Druckerauflösung:	Gegeben durch die Anzahl der Druckerpunkte, die der Druckkopf nebeneinander setzen kann (z.B. 300 dpi). Da bei Tintenstrahldruckern nur 3 Grundfarben (ggf. zusätzlich Schwarz) zur Verfügung stehen, wird jedes Farbpixel durch eine Rasterzelle mit 16×16 Druckerpunkten dargestellt.
Druckauflösung:	Druckerauflösung/Rasterzellengröße, d.h. ein 400 dpi-Belichter mit 16×16 Rasterzellen kann nur $400/16 = 25$ dpi erzeugen.

Thermosublimationsdrucker lösen durch Heizelemente Farbpigmente aus einer Folie, die anschließend in ein Spezialpapier eindringen. Jedes True-Color-Pixel wird daher als ein Farbpunkt gedruckt

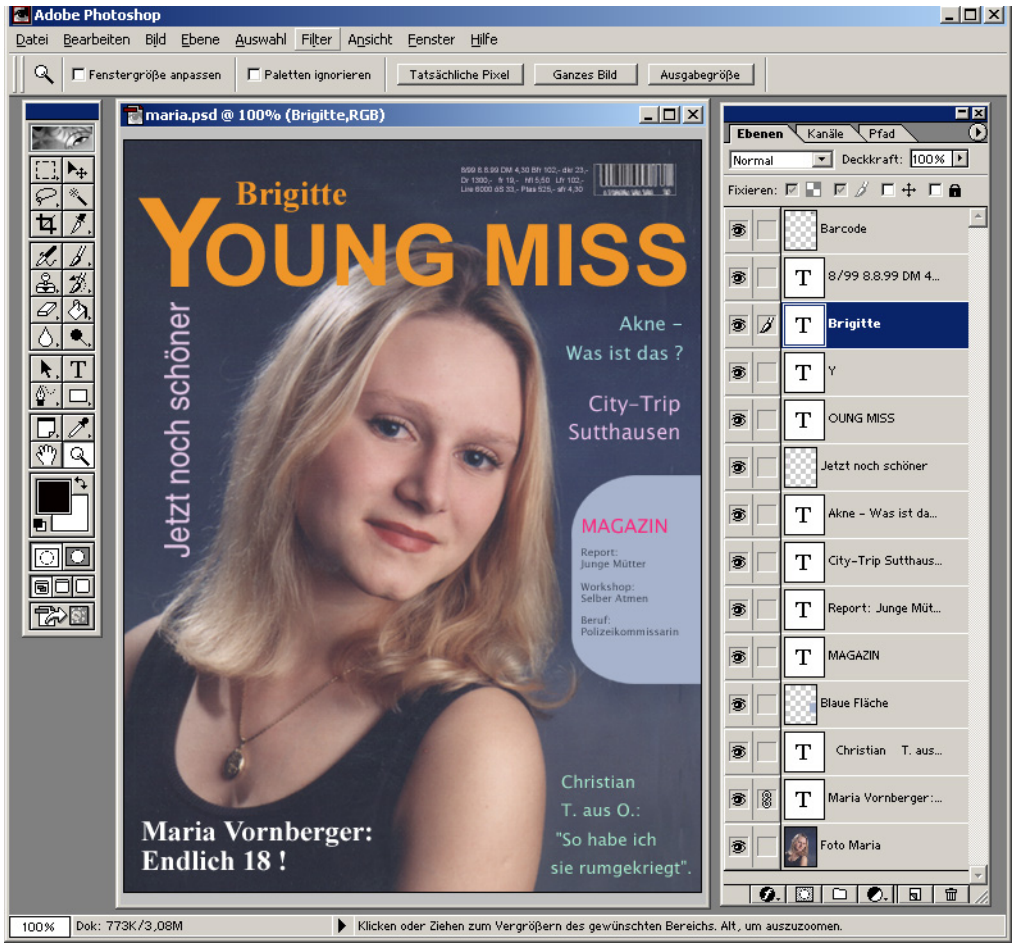
(*Continuous Tone-Druck*). Ein 300 dpi Thermosublimationsdrucker ermöglicht daher eine 300 dpi Druckauflösung. Ein 100 ASA-Kleinbilddia, welches eine reale Auflösung von 2500 dpi in sich birgt, kann daher mit einem 300 dpi-Thermosublimationsdrucker in idealer Weise auf das $2000/300 = 8.33$-fache vergrößert werden, welches eine Kantenlänge von $8.66 * 36$ mm $= 30$ cm bedeutet (etwa DIN-A4).

Auflösung KB-Dia

Die Projektion eines 24×36 mm Kleinbilddias auf eine 1.80 m breite Leinwand stellt eine 50-fache Vergrößerung dar. Sind dann auf 1 cm Leinwand 10 Linien zu unterscheiden, entspricht das 20 dots per cm Leinwand= 1000 dots per cm Dia $\approx$ 2500 dpi.

Ein Kleinbildnegativ oder -Dia, eingescannt mit 2500 dpi, ergibt eine Datei mit 3.6 cm / 2.54 * 2500 = 3543 x 2.4 cm / 2.54 * 2500 = 2362 Pixel. Auf einem 300 dpi-Drucker entsteht eine Druckfläche von 3543 / 300 * 2.56 = 30 cm Breite und 2362 / 300 * 2.56 = 20 cm Höhe (DIN-A-Format).

Der im analogen Dia enthaltene Helligkeitsumfang ist etwa um den Faktor 10 größer als ein Farbdruck oder Monitorbild darstellen kann.



Screenshot vom Bildbearbeitungsprogramm Photoshop

9.2 GIF

Das *Graphics Interchange Format* (GIF) wurde von CompuServe eingeführt und wird benutzt, um ggf. mehrere Bilder (Animation) 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.

Bei den auf WWW-Seiten verwendeten Grafikformaten nimmt GIF den Spitzenplatz in punkto Häufigkeit ein. Insbesondere bei künstlich erzeugten Bildern mit einheitlich gefärbten Farbflächen ist es an Kompaktheit nicht zu schlagen.

Abbildung 9.1: Statisches GIF, Dateigröße: 2K

Zwei Kompressionsideen tragen zur Datenreduktion bei:

Farbpalette: Statt in jedem Pixel das komplette RGB-Tripel mit 3 Byte = 24 Bit Farbinformation zu speichern, werden für geeignetes p die 2^p wichtigsten Farben in der Farbpalette, gehalten und über einen p-Bit langen Index referiert. Für $p = 8$ schrumpft der Platzbedarf daher auf ein Drittel.

LZW: Das von *Lempel*, *Ziv* und *Welch* entwickelte und als Patent geschützte Verfahren zur Kompression beliebiger Zeichenfolgen basiert auf der Idee, in einer sogenannten Präfix-Tabelle die Anfangsstücke bereits gelesener Strings zu speichern und wiederholtes Auftauchen derselben Strings durch Verweise in die Tabelle zu kodieren.

Beide Ansätze zahlen sich insbesondere dann aus, wenn die Vorlage weite Bereiche mit identischer Information enthält, wie es bei computergenerierten Grafiken, wie z.B. Logos, der Fall ist. Zum einen enthält das Bild dann gar nicht die theoretisch verfügbare Zahl von $256^3 \approx 16$ Millionen Farben, sondern nur wenige Dutzend, und kann daher völlig verlustfrei durch eine Palette mit 256 Einträgen dargestellt werden. Zum anderen führen Folgen von identischen Pixelindizes zu kompakten Einträgen in der Präfixtabelle.

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

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

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

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

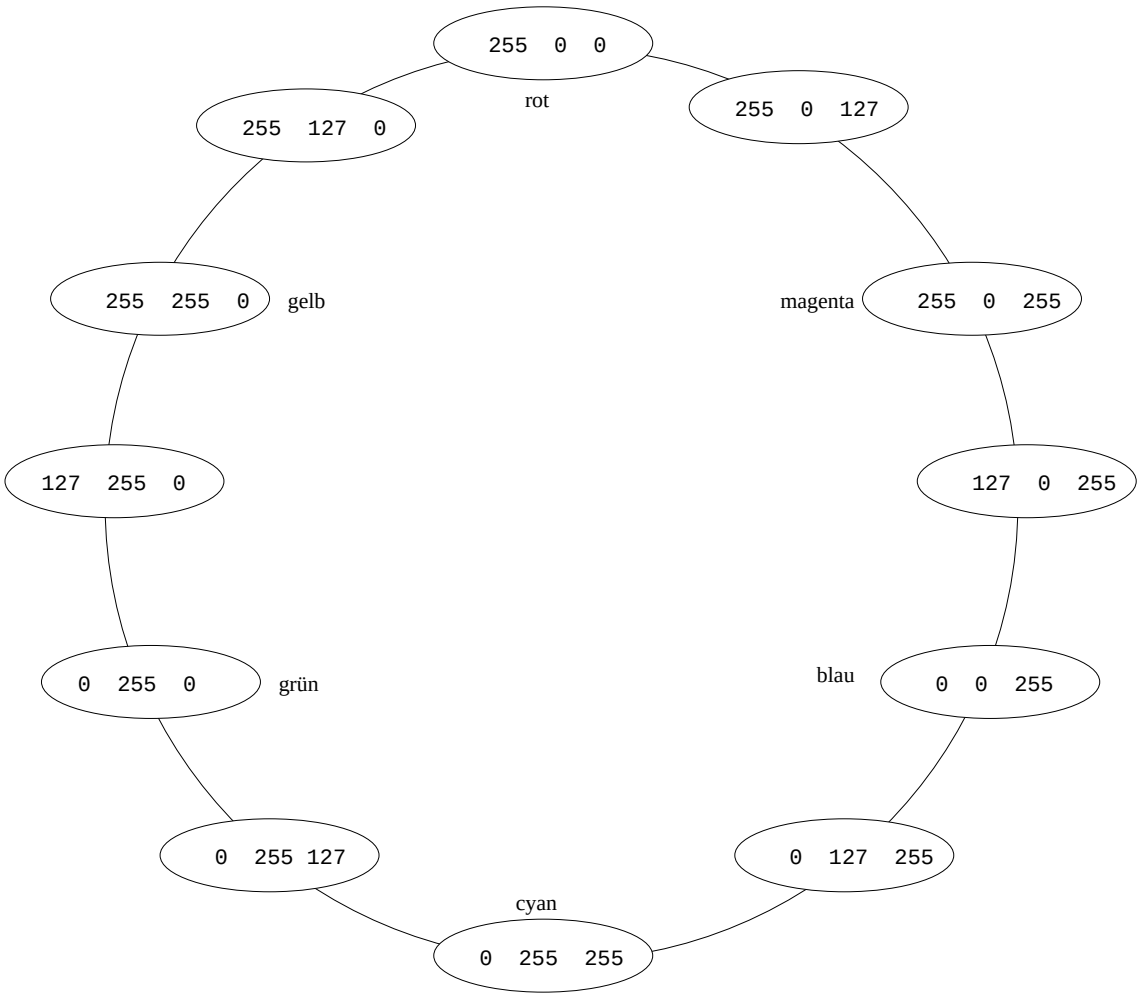


Abbildung 9.2: 12 typische Eintrage einer Farbtabelle fur einen Bildschirm mit 4 Bit Farbtiefe

9.3 Erzeugung einer bildbezogenen Farbtabelle

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

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

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

Optimaler Algorithmus

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

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

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

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

Popularity-Algorithmus (1978)

`Wähle die K häufigsten Farben.`

Nachteil: Selten vorkommende Farben werden schlecht repräsentiert.

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

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

Median-Cut (Heckbert, MIT 1980)

Ziel:
Finde K Repräsentanten, die jeweils möglichst gut gleich viele Farben repräsentieren.

Idee:
Zerlege den RGB-Würfel mit den beobachteten Farbhäufigkeiten solange sukzessive in Unterwürfel durch Aufsplitten an einer Trennfläche, bis K Unterwürfel entstanden sind.

```
Initialisiere RGB-Wuerfel mit Häufigkeiten der beobachteten Farbtupel
Initialisiere Wurzel des Schnittbaums mit Gesamtzahl der Pixel
While noch_nicht_genügend_Blätter do
      Wähle Blatt mit der größten Pixelzahl
      Bestimme umschliessende Box
      Bestimme Achse mit größtem Wertebereich
      Durchlaufe Box längs dieser Achse
      Teile am Median in zwei Hälften
      Trage Hälften als Soehne ein
end
Für jedes Blatt wähle den Mittelwert aller in ihm liegenden Farben.
```

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

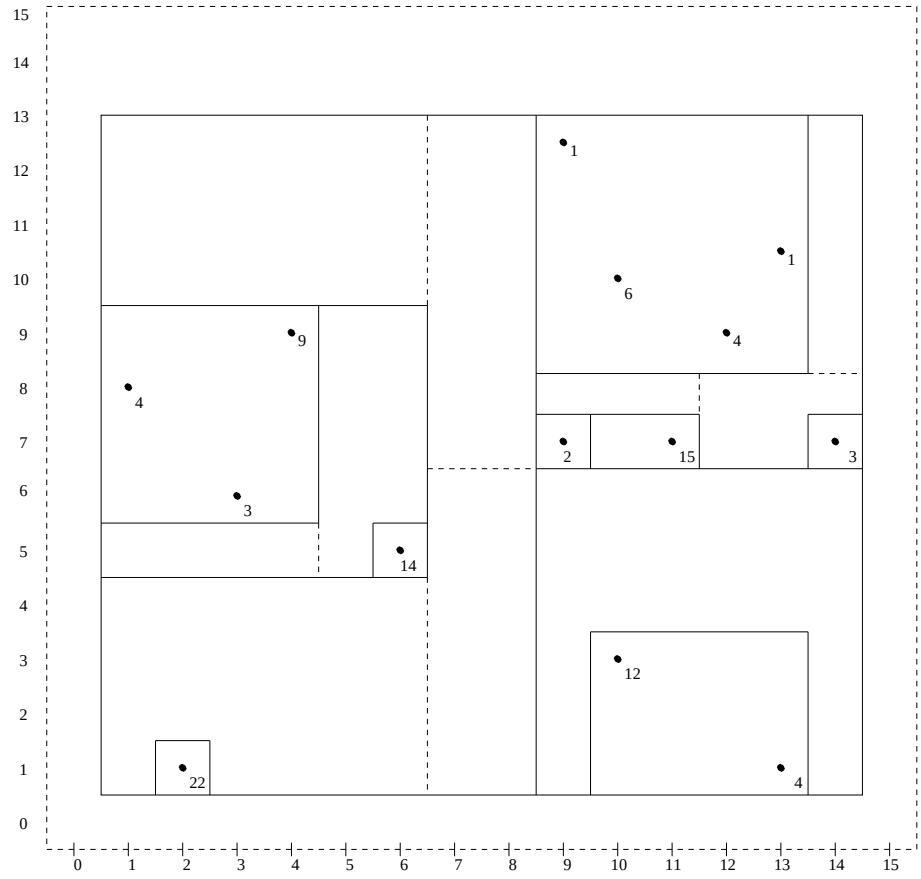


Abbildung 9.3: Partitionierung der Ebene beim Median-Cut-Algorithmus

Abbildung 9.3 zeigt die Anwendung des Median-Cut-Algorithmus anhand eines Beispiels. Zur einfachen Darstellung ist der Farbbaum zweidimensional gewählt. Gezeigt wird die Verteilung der Farbtupel aus dem Bereich $[0\ldots15] \times [0\ldots15]$ in einem Bild mit $10 \times 10 = 100$ Pixeln. Eingetragen sind

an der Position x/y die Häufigkeit des Farbtupels x,y. Schnittlinien sind gestrichelt, umschließende Boxen durchgezogen gezeichnet.

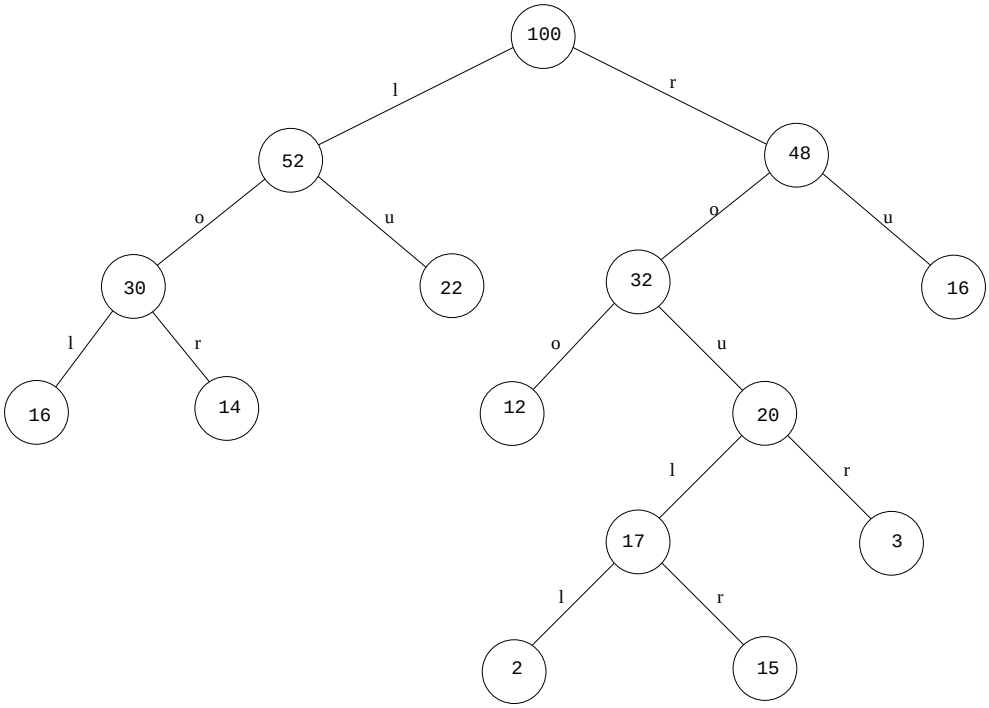
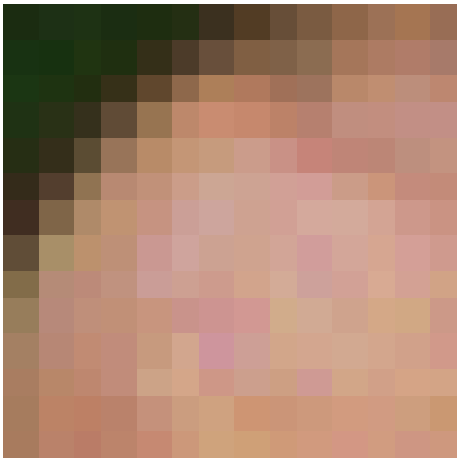


Abbildung 9.4: Schnittbaum beim Median-Cut-Algorithmus

Abbildung 9.4 zeigt für das vorliegende Beispiel den Schnittbaum nach Anwendung des Median-Cut-Algorithmus. Die Knoten sind markiert mit der Anzahl der noch zu quantisierenden Pixel, an den Kanten ist vermerkt, ob es sich um einen Links/Rechts- oder um einen Oben/Unten-Schnitt handelt.

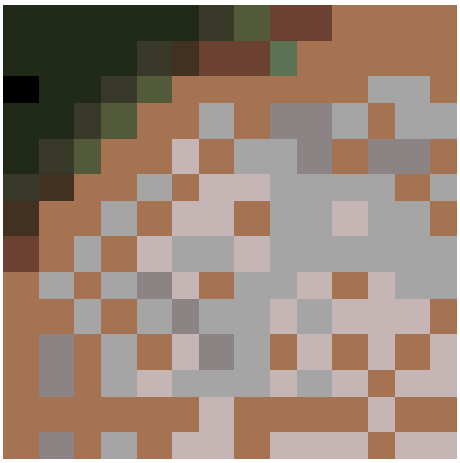
Floyd-Steinberg-Dithering (1975)
Der bei Verwendung einer Farbtabelle verursachte Fehler beim Quantisieren eines Pixels wird auf die Nachbarpixel verteilt (bevor diese quantisiert werden).

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

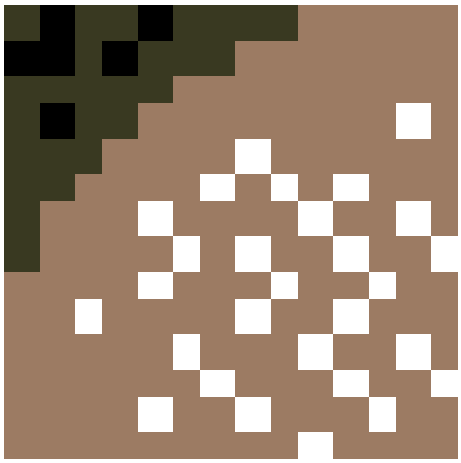


24 Bit pro Pixel,
16 Mill. Farben

8 Bit pro Pixel,
256 Farben



4 Bit pro Pixel,
16 Farben



2 Bit pro Pixel,
4 Farben

Abbildung 9.5: Auswirkung des Median-Cut-Algorithmus

Abbildung 9.5 zeigt Original und quantisierte Versionen einer 14×14 Ausschnittvergrößerung, erstellt von einem 814×517 True-Color-Bild. Die Zahl der Farben bezieht sich jeweils auf das Gesamtbild. Verwendet wurde der Median-Cut-Algorithmus mit anschließendem Floyd-Steinberg-Dithering.

9.4 LZW-Komprimierung (Lempel/Ziv/Welch, 1984)

Ein wichtiger Bestandteil des GIF-Formates ist die LZW-Komprimierung für Zeichenketten.

Starte mit Stringtabelle, gefüllt mit `characters`, und fülle sie mit Verweisen auf bereits gelesene Substrings.

```
x := get_char();
w := x;
repeat
   x := get_char();
   if wx in Tabelle
      then w := wx
      else put_string(code(w));
           trage wx in Tabelle ein
           w := x
   endif
until x = EOF
```

w	Input	Output	String	Code
			a	1
			b	2
			c	3
	a			
a	b	1	ab	4
b	a	2	ba	5
a	b			
ab	c	4	abc	6
c	b	3	cb	7
b	a			
ba	b	5	bab	8
b	a			
ba	b			
bab	a	8	baba	9
a	a	1	aa	10
a	a			
aa	a	10	aaa	11
a	a			
aa	a			
aaa	a	11	aaaa	12

Entwicklung von Output und Tabelle für den Input `ababcbababaaaaaaa`

String Präfix-String-Index	Erweiterungs-character	Code
	a	1
	b	2
	c	3
1	b	4
2	a	5
4	c	6
3	b	7
5	b	8
8	a	9
1	a	10
10	a	11
11	a	12

Implementation der Stringtabelle

9.5 Kompression nach JPEG

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

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

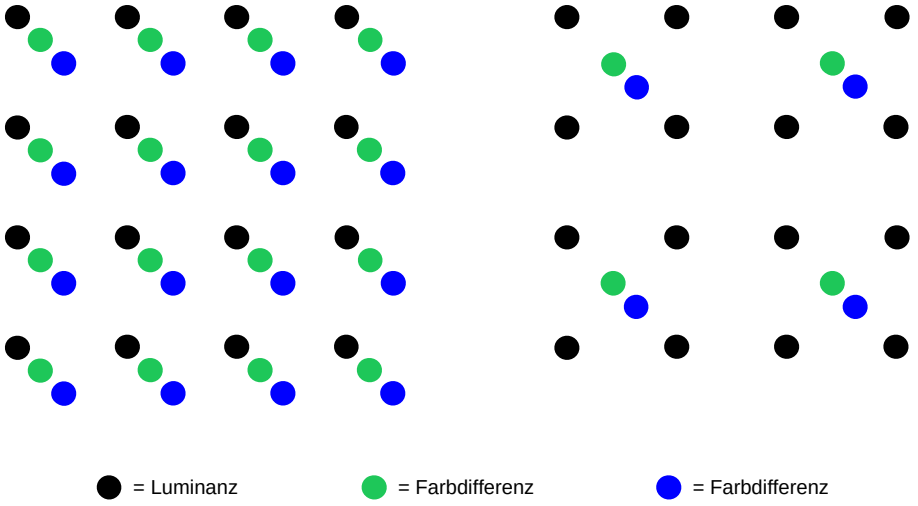


Abbildung 9.6: 4:1:1 Subsampling

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

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

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

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

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

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

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

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

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

Mit Hilfe der inversen DCT lassen sich die Originalwerte rekonstruieren.

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

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

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

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

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

$$r[u,v] := \left\lfloor \frac{s[u,v]}{q[u,v]} \right\rfloor$$

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

Bildmatrix

↓

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

rekonstruierte Bildmatrix

↑

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

DCT-Koeffizienten

⟶

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

quantisierte DCT-Koeffizienten

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

Quantisierungsmatrix

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

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

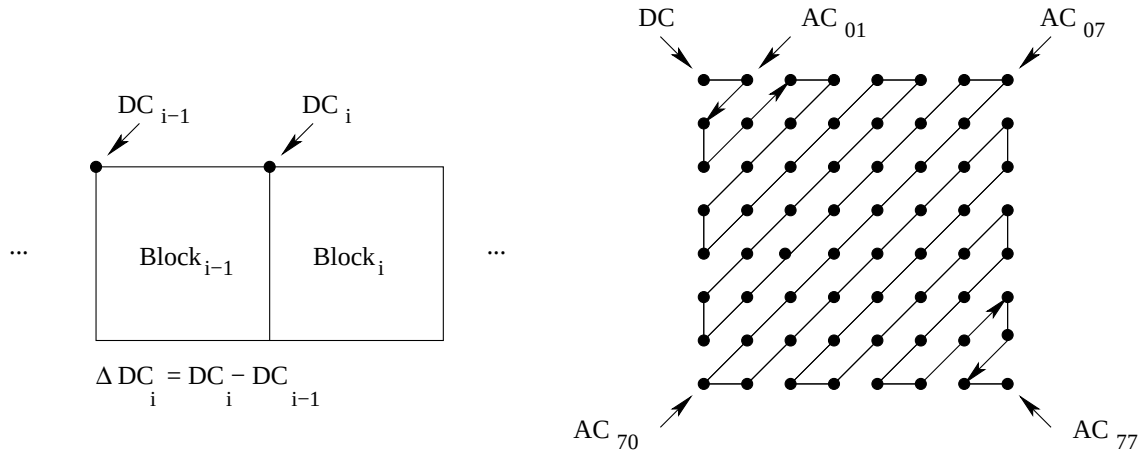


Abbildung 9.7: Durchlaufsequenz pro Macroblock

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

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

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

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

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

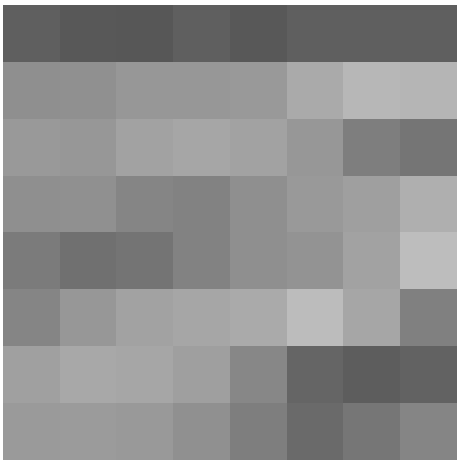
Huffman Codierung
für Symbol `1`

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

Komplement-Codierung
für Symbol `2`

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

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



Ausgangsbildmatrix

rekonstruierte Bildmatrix

Abbildung 9.8: 8 x 8 Matrix vor und nach der JPG-Kompression.

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

Dezimaldarstellung der Bildbereich-Matrix

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

8-Bit-Codierung für Bildbereich-Matrix

```
00011111
```

quantisierter DC-Koeffizient

```
1111001000101100111011011100011001100111100010100000010011001
1110100110110010001100010010001111111111111010000000010001010
```

Huffman-Codierung für quantisierte AC-Koeffizienten

motel.tif
100%

motel80.jpg
9%

motel40.jpg
5%

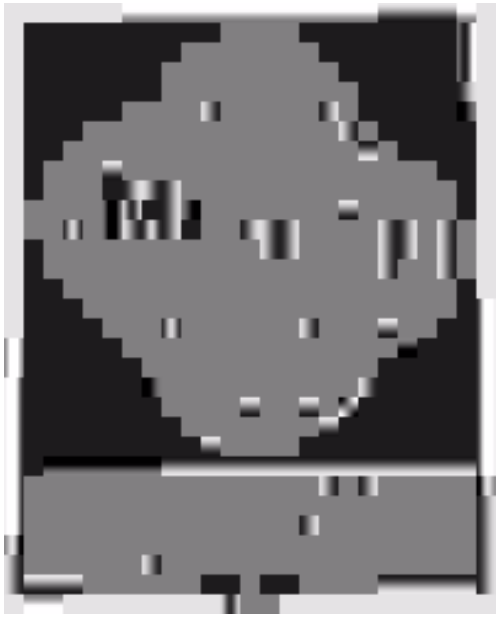
motel20.jpg
4%

Abbildung 9.9: Kompression nach JPEG. Platzbedarf in Prozent bezogen auf tif-Datei

motel10.jpg
3%

motel05.jpg
2%

motel02.jpg
1.5%



motel01.jpg
1%

Abbildung 9.10: Kompression nach JPEG. Platzbedarf in Prozent bezogen auf tif-Datei

9.6 TIF

Eines der häufig verwendeten Dateiformate zur Speicherung von Pixelbildern wurde von Aldus Corporation, Seattle, USA, entwickelt: *Tag Image File Format*, abgekürzt TIF.

Dateikopf:	1. Wort	`4d4d` Motorola (high order first) `4949` Intel (low order first)
	2. Wort	`002a` Versionsnummer (konstant)
	3. + 4. Wort	Offset des ersten IFD (Image File Directory)
IFD:	1. Byte	Anzahl der Tags im IFD, jedes Tag besteht aus 12 Bytes
	Aufbau eines Tag:	
	1. Wort	Identifikation des Tags (es gibt 45 Tags)

	2. Wort	Typ der Daten
		`1:` Byte
		`2:` 0-terminierter String von ASCII-Zeichen
		`3:` short = 16-Bit unsigned integer `(0000-ffff)`
		`4:` long = 32-Bit unsigned integer `(00000000-ffffffff)`
		`5:` Rational = Bruch aus 2 long-Werten
		`11:` float = im IEEE-Format, einfache Genauigkeit
		`12:` double = im IEEE-Format, doppelte Genauigkeit

	3. + 4. Wort	Anzahl der Daten für dieses Tag
	5. + 6. Wort	Tag-Daten falls mit ≤ 4 Bytes darstellbar : von links nach rechts füllen falls mehr als 4 Bytes benötigt: 32-Bit Startadresse für Daten
Datenblock:		RGB-Werte oder Indizes (ggf. komprimiert) für die Farbtabelle

Offset	Bedeutung	Wert
0100	ImageWidth	z.B. 814
0101	ImageLength	z.B. 517
0102	Bits per Sample	z.B. 8, ggf. Zeiger auf 3 shorts
0103	Compression	1 keine Komprimierung 2 CCITT Gruppe 3 (nur bei Schwarz/Weiß) 3 CCITT Gruppe T4 (nur bei Schwarz-Weiß) 4 CCITT Gruppe T6 (nur bei Schwarz/Weiß) 5 LZW 6 JPEG 32773 Packbit mit Lauflängen-Kodierung
0106	Photometric Interpretation	0 S/W mit 0=Weiß, wachsende Nummern gegen Schwarz 1 S/W mit 0=Schwarz, wachsende Nummern gegen Weiß 2 RGB-Farbbild, pro Pixel drei Farbwerte, 0,0,0=Schwarz 3 Palettenbild, pro Pixel ein Index in Palette 4 Transparenzmaske für ein weiteres Bild 5 CMYK-Farbsystem, pro Pixel 4 Farbwerte 6 YUV-Farbsystem, pro Pixel Luminanz + 2 Chrominanz
0111	Strip Offset	Startadresse des Datenblocks (meistens $0008)
0112	Orientation	1 horizontal, beginnend oben links
0115	Samples per Pixel	3 bei RGB-Bildern 1 sonst
0116	Rows per Strip	wenn 1 Block: Bildhöhe wenn >1 Block: Anzahl Zeilen pro Block
0117	Strip Byte Counts	wenn 1 Block: Anzahl der Bytes im Block wenn >1 Block: Zeiger auf Feld mit Streifenlängen
011a	X Resolution	Zeiger auf 2 long Werte, zB. 300 1
011b	Y Resolution	Zeiger auf 2 long Werte, zB. 300 1
0128	Resolution unit	1 keine Einheit 2 Inch 3 Zentimeter
011c	Planar Configuration	1 RGBRGBRGB ... 2 RRRR... GGGG.... BBBB....
0140	Color map	Anzahl der Einträge + Startadresse Jeder Farbwert $0 \leq w = 255$ wird als Wort ww abgelegt

Tag-Art : True Color Bild

						Position 0:
4d4d						Motorola
002a						Versionsnr. (konstant)
0013	43ba					Offset des IFD $001343ba = 1262522
						Position 8:
ebe9	e7e2	e4c0	b3bd	9a9c	a387	Beginn der RGB-Werte, insgesamt 814*517*3 Bytes
9178	5294	724b	8d6f	4b8e	7453	
a79e	6da8	a06f	a19b	759d	a486	
a6ac	8aa2	aa92	a7ac	9ba5	a995	
...						
...						
...						
						Position 1262522: Beginn des Image File Directory
000b						$000b = 11 Tags zu je 12 Bytes
0100	0003	0000	0001	032e	0000	Breite, 1 short, Wert 3*256 + 2*16 + 14 = 814
0101	0003	0000	0001	0205	0000	Höhe, 1 short, Wert 2*256 + 0*16 + 5 = 517
0102	0003	0000	0003	0013	4444	Bits per sample, 3 short, Adresse $134444=1262660
0103	0003	0000	0001	0001	0000	Compression, 1 short, Wert=1 (keine Kompression)
0106	0003	0000	0001	0002	0000	Photometric, 1 short, Wert=2 (RGB-Farbbild)
0111	0004	0000	0001	0000	0008	StripOffsets,1 long, Wert=8 (Startadresse)
0112	0003	0000	0001	0001	0000	Orientierung,1 short, Wert = 1 (zeilenweise links/oben)
0115	0003	0000	0001	0003	0000	Samples per Pixel, 1 short, Wert=3 (RGB)
0116	0004	0000	0001	0000	0205	Rows per Strip, 1 long, Wert $205 = 517 (Zeilen)
0117	0004	0000	0001	0013	43b2	StripByteCounts, 1 long, Wert = 1262514 (Bytes)
011c	0003	0000	0001	0001	0000	Planar Konfiguration, 1 short, Wert=1 (RGBRGB...)
0000	0000					Ende IFD, kein weiteres IFD
						Position 1262660:
0008	0008	0008				jeweils 8 Bit pro Farbwert

kommentierter Hex-Dump zu einem True-Color-Bild im tif-Format

Vorspann	8 Bytes
814 * 517 RGB-Tripel	1262514 Bytes
IFD 11 * 12 + 12	144 Bytes
→ Dateilänge	1262666 Bytes

Tag-Art : Palettenbild

```
                                                Position 0:
4d4d                                            Motorola
002a                                            Versionsnr. (konstant)
0006  6be                                       Offset des IFD $00066bee = 420846

                                                Position 8:
fe68  bd52  6f6f  6f6f  0101  01f4              Beginn der Indizes, insgesamt 814*517 Bytes
5252  e952  2abd  522a  5252  b252
0000  016f  076f  01bd  e952  3877
ea3a  3838  38a1  0101  3a38  01be
.
.
                                                Position 420846: Beginn des Image File Directory
000c                                            $000c = 12 Tags zu je 12 Bytes
0100  0003  0000  0001  032e  0000              Breite, 1 short, Wert $32e = 814
0101  0003  0000  0001  0205  0000              Höhe, 1 short, Wert $205 = 517
0102  0003  0000  0001  0008  0000              Bits per sample, 1 short, Wert = 8 (Adresse)
0103  0003  0000  0001  0001  0000              Compression, 1 short, Wert=1 (keine Kompression)
0106  0003  0000  0001  0003  0000              Photometric, 1 short, Wert=3 (Palette)
0111  0004  0000  0001  0000  0008              StripOffsets,1 long, Wert=8 (Startadresse)
0112  0003  0000  0001  0001  0000              Orientierung,1 short, Wert=1 (zeilenweise l.o.)
0115  0003  0000  0001  0001  0000              Samples per Pixel, 1 short, Wert=1 (Palette)
0116  0004  0000  0001  0000  0205              Rows per Strip, 1 long, Wert $205 = 517 (Zeilen)
0117  0004  0000  0001  0006  6be6              StripByteCounts,1 long, Wert = 420838 (Bytes)
011c  0003  0000  0001  0001  0000              Planar Konfiguration, 1 short, Wert=1 (RGBRGB..)
0140  0003  0000  0300  0006  6c84              Colormap, $300 = 768 short, beginnend bei $66c84

0000  0000                                      Ende IFD, kein weiteres IFD

                                                Position: $66c84 = 420996
                                                Palette mit 768 Doppelbytes

b8b8  a8a8  1010  3c3c  0808  2424
b4b4  a8a8  1c1c  9898  6868  3838
3c3c  5858  2c2c  7070  6464  8080
.
.
3030  1414  1010  3030  2020  3c3c
7070  2828  d8d8  1010  2828  2020
7474  1414  acac  a8a8  e4e4  e4e4
```

kommentierter Hex-Dump zu einem Palettenbild im tif-Format

Vorspann	8 Bytes
814 * 517 Indizes	420838 Bytes
IFD 12 * 12 + 6	150 Bytes
Palette 3 * 256 * 2	1536 Bytes
→ Dateilänge	422532 Bytes

9.7 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 $\times$ Höhe vielen Pixel, beginnend in der linken oberen Bildecke zeilenweise von links nach rechts.

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

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

Beispiel:

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

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

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

Binäre Bildinformation

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

Die Pixelwerte werden dabei wie folgt abgelegt:

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

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

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

Konvertierungsroutinen

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

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

9.8 Photo-CD

Die von Kodak entwickelte Photo-CD weist eine Verzeichnisstruktur auf, die zum Ansteuern eines Photo-CD-Players ausgelegt ist und außerdem von diversen Bildverarbeitungsprogrammen gelesen werden kann. Hierfür stellt Kodak eine Programmierbibliothek zur Verfügung; das Erstellen einer Photo-CD ist nur speziellen Labors vorbehalten. Es können 100 Photos, ggf. in mehreren Sessions, aufgebracht werden. Da sich dadurch das Verzeichnis an verschiedenen Stellen der CD befinden kann, muß das CD-Laufwerk “multisessionfähig” sein.

Jedes Photo wird in 5 verschiedenen Auflösungen in einem (patentrechtlich geschützten) Image-Pac abgelegt.

Bezeichnung	Auflösung	unkomprimierter Platzbedarf
Base/16	192×128	72 KBytes
Base/4	384×256	288 KBytes
Base	768×512	1152 KBytes
Base$*$4	1536×1024	4608 KBytes
Base$*$16	3072×2048	18432 KBytes

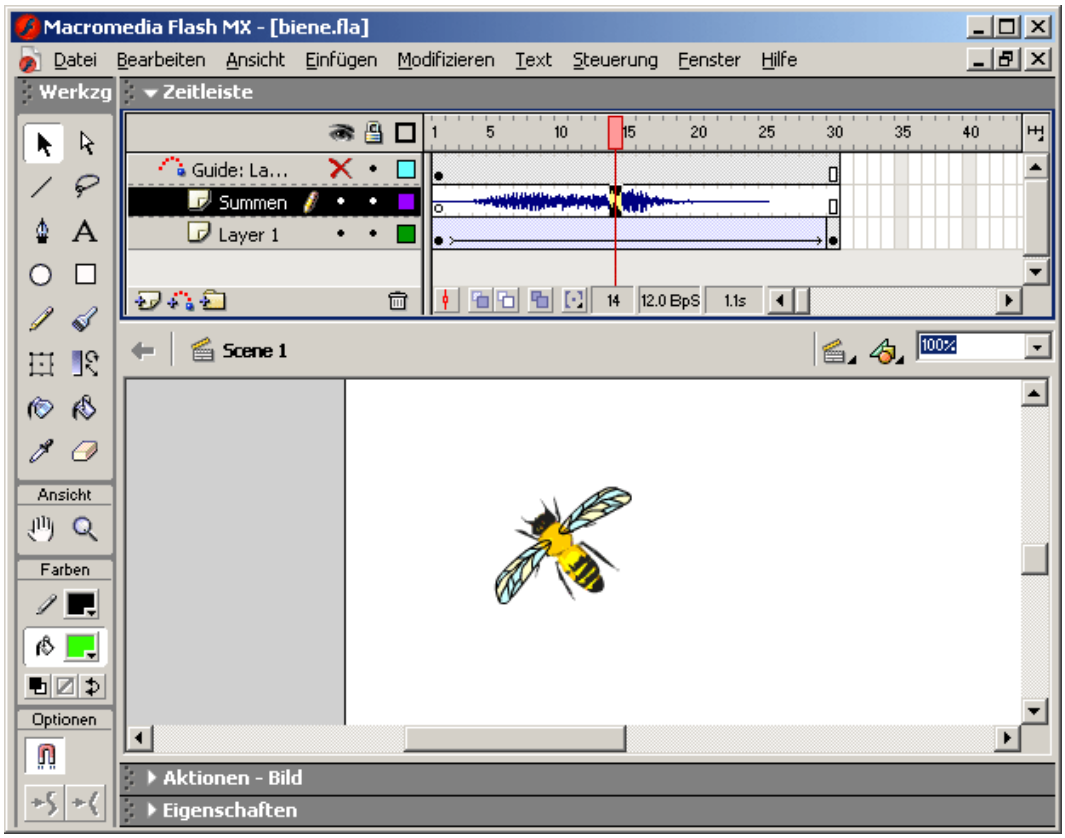
Beim Einscannen werden die RGB-Werte für jeden Bildpunkt mit 12 Bit Genauigkeit pro Farbanteil abgetastet und anschließend in das Kodak-eigene YCC-Format konvertiert (8 Bit für Helligkeit, je 8 Bit für zwei Chrominanzwerte). Durch 4:1:1 Subsampling wird die Chrominanz über je 4 Pixel gemittelt. Pro Image-Pac werden die ersten 3 Auflösungen explizit gespeichert, für Base$*$4 und Base$*$16 nur die Differenzen. Dadurch reichen etwa 4.5 MByte aus, also etwa 20 % des aufsummierten Platzbedarfs.

Kapitel 10

2D-Grafik im Web

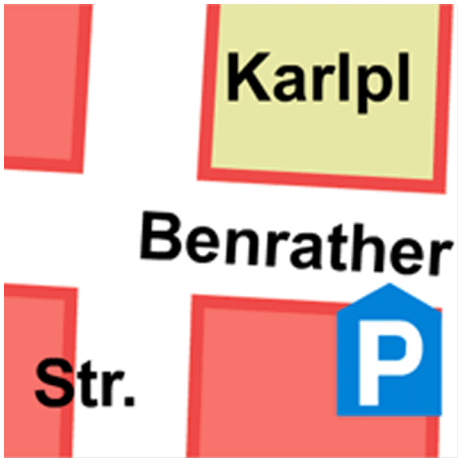
10.1 Macromedia Flash

Macromedia Flash ist ein Werkzeug zum Editieren und Animieren von zweidimensionalen Vektorgrafiken. Durch sein kompaktes Speicherformat ist es besonders geeignet für plakative, animierte Grafiken im Internet. Zum Abspielen in einem Web-Browser ist das Macromedia-Flash-Plugin erforderlich.



Vektorgrafikwerkzeug Macromedia Flash MX

Gesamtansicht des Stadtplans



Teilansicht des Stadtplans

Rasterfreies Zoomen bei Dateien im Flash-Format durch Vektorgrafik

```
<HTML>
  <HEAD>
    <TITLE>Macromedia Flash</TITLE>
  </HEAD>
  <BODY BGCOLOR="f9d1a1">
    <CENTER>
      <H1>Eingebetteter Flash-Film</H1>
        <EMBED
           src="vogel.swf"
           width=700
           height=200
           Loop="True"
           Play="True"
           BGColor="f9d1a1"
           Quality="Autohigh"
           Scale="showall"
           SAlign=""
        >
      <P>vogel.swf, 6 K
    </CENTER>
  </BODY>
</HTML>
```

Flash mit Einzelbild-Animation

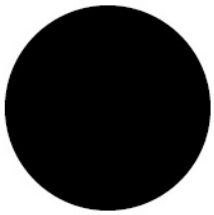
10.2 SVG

Im Gegensatz zum proprietären Binärformat Flash handelt es sich bei SVG (Scalable Vector Graphics) um einen offenen Standard eines Vektorgrafikformates auf XML-Basis. Entwickelt wurde dieses Format von einem Verbund von Firmen, darunter Adobe, Apple, Autodesk, BitFlash, Corel, HP, IBM, ILOG, Macromedia, Microsoft, Netscape, OASIS, Quark, RAL, Sun, Visio, W3C, Xerox.

SVG ...

- kann nach Text durchsucht werden
- erlaubt Textgestaltung
- kann Objekte zeitlich koordiniert bewegen
- enthält Eventhandling
- verfügt über photoshopartige Filter-Effekte
- ist sowohl unkomprimiert als auch komprimiert einsetzbar
- erfordert zum Abspielen ein Plugin
- verlangt Rechenleistung auf der Clientseite

Die SVG-Datei `kreis.svg` platziert einen Kreis mit Radius 80 in die Mitte der Zeichenfläche:

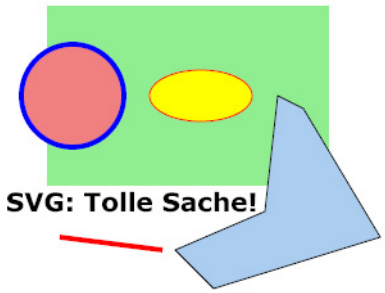


SVG-Screenshot: Kreis

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink" >
  <circle  cx="100" cy="100" r="80" />
</svg>
```

SVG-Quelltext: Kreis

Die SVG-Datei `basics.svg` enthält die grafischen Objekte Gerade, Rechteck, Kreis, Ellipse, Polygon sowie Text. Die Objekte erhalten als Attribute ihre Koordinaten sowie Angaben zur Farbe der Füllung, Farbe der Umranduung und Stärke der Umrandung.



SVG-Screenshot: Gerade, Rechteck, Kreis, Ellipse, Polygon, Text

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="310" height="270" viewBox="0 0 310 270"
xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink" >

  <title>Gerade, Rechteck, Kreis, Ellipse, Polygon, Text</title>

  <line    x1="50" y1="210" x2="130" y2="220"
           stroke="red" stroke-width="4" />

  <rect    x="40" y="30" width="220" height="140"
           fill="lightgreen" stroke-width="12" />

  <circle  cx="60" cy="100" r="40"
           fill="lightcoral" stroke="blue" stroke-width="4px" />

  <ellipse cx="160" cy="100" rx="40" ry="20"
           fill="orange" stroke="red" stroke-width="1" />

  <polygon points="220,100 240,110 300,210 170,250 140,220 210,190"
           fill="#abcdef" stroke="#000000" stroke-width="1"/>

  <text    x="8" y="190"
           style="font-family:verdana; font-size:20px; font-weight:bold">
           SVG: Tolle Sache! </text>

</svg>
```

SVG-Quelltext: Gerade, Rechteck, Kreis, Ellipse, Polygon, Text

Die SVG-Datei `gruppierung.svg` fasst mit dem `<g>`-Element mehrere Objekte zusammen und referenziert sie später über das `xlink:href`-Element (erst dann werden sie gezeichnet). Weiterhin finden vier Transformationen statt. Die erste besteht aus einer einzelnen Translation. Die zweite und dritte bestehen aus einer Translation, gefolgt von einer Rotation. Die vierte wird beschrieben durch Angabe von sechs Zahlen $(a\ b\ c\ d\ e\ f)$, welche eine 3 x 3-Transformationsmatrix beschreiben mit erster Spalte $(a\ b\ 0)^T$, zweiter Spalte $(c\ d\ 0)^T$ und dritter Spalte $(e\ f\ 1)^T$.

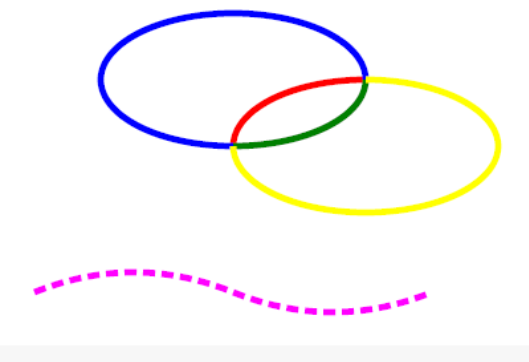
Obacht: Transformationen beziehen sich auf das Koordinatensystem des Objekts. `transform= ̈rotate(45) translate (200, 0) ¨` bedeutet daher zunächst eine Drehung im Uhrzeigersinn um 45 Grad und dann eine Verschiebung um 200 Pixel längs der Diagonalen nach rechts unten.

SVG-Screenshot: Gruppierung, Wiederverwendung, Transformation

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="400" height="100" viewBox="0 0 400 100"
xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink" >

  <title>Gruppierung und Wiederverwendung</title>

  <defs>
  <g id="vorlage">
    <rect   x="-40" y="-20" width="80" height="40" fill="blue" />
    <circle cx="0"  cy ="0" r="36" fill="red" />
  </g>
  </defs>

  <use xlink:href="#vorlage"
       transform="translate( 80 50)" />

  <use xlink:href="#vorlage"
       transform="translate(160 50) rotate(30)" fill-opacity="0.8"/>

  <use xlink:href="#vorlage"
       transform="translate(240 50) rotate(60)" fill-opacity="0.5"/>

  <use xlink:href="#vorlage"
       transform="matrix(0.0 -1.0 1.0 0.0 320 50)" fill-opacity="0.2"/>

</svg>
```

SVG-Quelltext: Gruppierung, Wiederverwendung, Transformation

In der Datei `kurven.svg` werden unter Verwendung des `path`-Element Kurven gezeichnet. Die vier Kreisbögen (Kommando A) beginnen jeweils bei (300,100) haben einen x-Radius von 100 und einen y-Radius von 50 und enden bei (200,100). Sie unterscheiden sich durch die vier Kombinationen von *large-arc-flag* und *sweep-flag*. Die erste kubische Bezierkurve (Kommando C) beginnt bei (50,260) und hat als weitere Kontrollpunkte (100,240), (150,240) und (200,260). Mit dem Kommando S wird der erste Kontrollpunkt der nächsten kubischen Bezierkurve automatisch errechnet durch die beiden letzten Kontrollpunkte der Vorgängerkurve. Daher reicht nun die Angabe zweier weiterer Punkte (300,280) und (350,260). Über das stroke-dasharray-Attribut wird die Kurve gestrichtelt gezeichnet: auf 9 gefüllte Pixel folgen 5 leere Pixel.



SVG-Screenshot: Kreisbögen und Bezierkurve

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="500" height="300" xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink" >

  <title>Kurven</title>

  <path fill="none" stroke="red" stroke-width="5"
   d="M 300 100 A 100 50  0 0 0 200 150" />

  <path fill="none" stroke="green" stroke-width="5"
   d="M 300 100 A 100 50  0 0 1 200 150" />

  <path fill="none" stroke="blue" stroke-width="5"
   d="M 300 100 A 100 50  0 1 0 200 150" />

  <path fill="none" stroke="yellow" stroke-width="5"
   d="M 300 100 A 100 50  0 1 1 200 150" />

  <path fill="none" stroke="magenta" stroke-width="5" style="stroke-dasharray: 9,5;"
    d="M 50 260  C 100 240 150 240 200 260 S 300 280 350 260"/>

</svg>
```

SVG-Quelltext: Kreisbögen und Bezierkurve

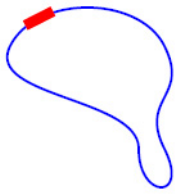
In der Datei `laufrad.svg` wird durch vierfache Wiederverwendung eines Viertelrades ein Speichenrad erzeugt. Das Speichenrad erhält durch `animateTransform` eine horizontale Verschiebung und eine Rotation.

SVG-Screenshot: Animation von Transformationen

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg width="400" height="150" xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink" >

  <title>Laufrad</title>

  <defs>
     <g id="rad">
     <path id="viertel" fill="white" stroke="black"
           d="M 10,10 L 40,10 A  30,30 0 0,1 10,40  L 10,10 " />
     <circle cx="0" cy="0" r="50" fill="red"   stroke="black" />
     <circle cx="0" cy="0" r="6"  fill="white" stroke="black" />
     <use xlink:href="#viertel" x="0" y="0" transform="rotate(  0)" />
     <use xlink:href="#viertel" x="0" y="0" transform="rotate( 90)" />
     <use xlink:href="#viertel" x="0" y="0" transform="rotate(180)" />
     <use xlink:href="#viertel" x="0" y="0" transform="rotate(270)" />
    </g>
  </defs>

  <use xlink:href="#rad" x="-50" y="75"/>

  <animateTransform xlink:href="#rad"
     attributeName="transform"
     type="translate"
     from="-50,0" to="500,0"
     begin="0s" dur="6s"
     repeatCount="indefinite"/>

  <animateTransform xlink:href="#rad"
     attributeName="transform"
     type="rotate"
     from="0" to="360"
     begin="0s" dur="3s"
     additive="sum"
     repeatCount="indefinite"/>

</svg>
```

SVG-Quelltext: Animation von Transformationen

Die Datei `bezier.svg` definiert eine Bezierkurve durch das `path`-Element. Dabei wird zunächst mit `M` der Startpunkt festgelegt, dann folgen durch `C` drei weitere Stützpunkte, die zusammen mit dem Vorgänger eine kubische Bezierkurve beschreiben. Durch `S` werden jeweils der dritte und vierte Stützpunkt für eine weitere kubische Bezierkurve definiert; automatisch übernommen werden als erster Stützpunkt der letzte Kurvenpunkt und als zweiter Stützpunkt. die Verlängerung der Geraden durch die beiden letzten Kurvenpunkte. Weiterhin wird durch `animateMotion` das rote Rechteck längs der Bezierkurve bewegt. Hierbei entsteht die resultierende Kurve durch Verknüpfung der Pfad-Koordinaten mit den Objektkoordinaten.



SVG-Screenshot: Bezier-Kurve als Animationspfad

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="260" height="200"  viewBox="0 0 260 200"
xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink" >

  <title>Bezier-Kurve mit Pfadanimation</title>

  <rect id="kasten" x="-12" y="-4" fill="red" width="24" height="8" />

  <path id="kurve" stroke="blue" stroke-width="2" fill="none"
   d="M  40,60
      C  60,20  140,20  160,60
      S 140,100 160,140
      S 140,180 140,140
      S 20,100 40,60" />

  <use xlink:href="#kasten" x="0" y="0" />
  <animateMotion xlink:href="#kasten"
    begin    ="0s" dur="5"
    rotate   ="auto"
    fill     ="freeze"
    calcMode ="linear"
    repeatCount="indefinite" >
    <mpath xlink:href="#kurve" />
  </animateMotion>

</svg>
```

SVG-Quelltext: Bezier-Kurve als Animationspfad

In der Datei `hyperlink.svg` wird durch Anklicken des grünen, abgerundeten Rechtecks die Webseite `http://www.inf.uos.de` aufgerufen.

SVG-Screenshot: Hyperlink

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="200" height="200" xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink" >

  <title>Hyperlink</title>

  <a xlink:href="http://www.inf.uos.de/">
  <rect x="50" y="50" rx="8" ry="8" fill="green" width="100" height="100"/></a>
  <text x="80" y="105" font-size="20" fill="white" >GO !</text>

</svg>
```

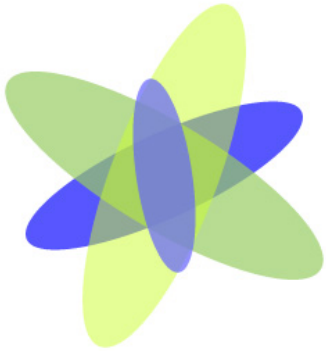
SVG-Quelltext: Hyperlink

In der Datei `synchronisation.svg` wird durch Klicken der roten Kugel eine Bewegung gestartet und nach 4 Sekunden wieder gestoppt. Weiterhin wird 4 Sekunden nach Beginn der Bewegung die Wave-Datei `signal.wav` abgespielt.

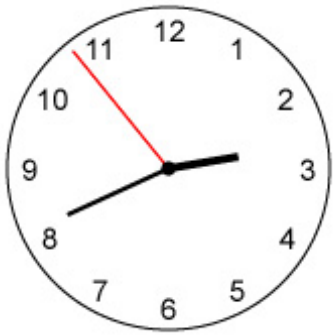
SVG-Screenshot: Synchronisation mit Audio

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20010904//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg xmlns:a="http://www.adobe.com/svg10-extensions" width="400"
height="100" xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">

  <title>Synchronisation</title>

  <circle id="kugel" cx="30" cy="50" r="25" fill="red"/>
  <rect   id="wand" x="325" y="0" width="25" height="100" fill="blue" />

  <animateMotion xlink:href="#kugel"
    begin="kugel.click" dur="4"
    path ="M  0,0 270,0"
    fill="freeze" />

 <a:audio xlink:href="signal.wav" begin="kugel.click+4s"></a:audio>

</svg>
```

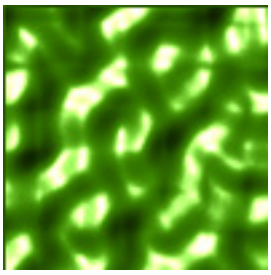
SVG-Quelltext: Synchronisation mit Audio



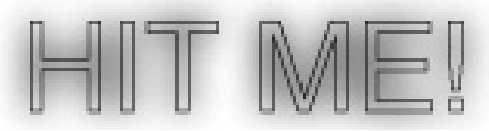
animierte Ellipsen



Systemzeit mit Javascript übertragen



Photoshopartige Filter



Button mit Interaktion

Schatteneffekt auf Pixelgrafik

In Kombination mit PHP können die Ergebnisse einer Datenbankquery dynamisch zu einer SVG-Grafik aufbereitet werden. Der folgende Screenshot zeigt eine Tortengrafik als Visualisierung der Lehrbelastung der Professoren aus der Datenbank UNI aus der Vorlesung Datenbanksysteme im SS 2003.

dynamisch erzeugte SVG-Grafik nach Datenbankquery

Kapitel 11

Fraktale

11.1 Selbstähnlichkeit

Viele in der Natur vorkommende Strukturen weisen eine starke Selbstähnlichkeit auf. Beispiele sind Gebirgsformationen, Meeresküsten oder Pflanzenblätter. Solche in sich wiederholende Muster, die beim Hereinzoomen immer wieder zutage treten, lassen sich ausnutzen, wenn man ein umfangreiches Gebilde durch ein kleines Erzeugendensystem darstellen möchte.

Die Produktion von Fraktal-Bildern erfolgt daher durch wiederholte Anwendung einer Transformation f auf einen bestimmten Teil des Bildes.

Abbildung 11.1: Farnblatt

11.2 Koch'sche Schneeflocke

Gegeben ein Polygon. Transformiere jede Polygonkante $\overline{PQ}$ zu einer Folge von Kanten $\overline{PR}$, $\overline{RS}$, $\overline{ST}$, $\overline{TQ}$, wie in Abbildung 11.2 gezeigt.

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.

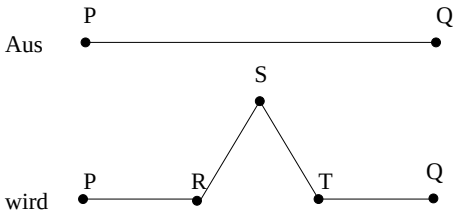
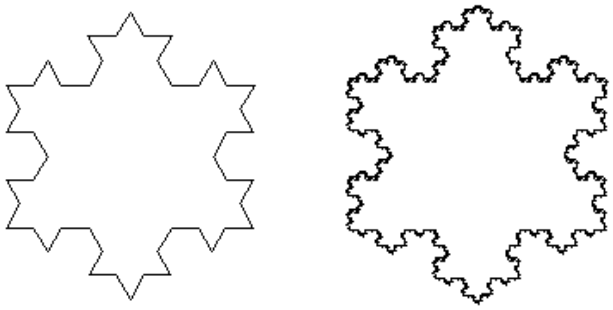


Abbildung 11.2: Iterationsvorschrift zur Koch'schen Schneeflocke



48 Polygonpunkte 3072 Polygonpunkte

Abbildung 11.3: Koch'sche Schneeflocke

11.3 Fraktale Dimension

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

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 $\frac{1}{N^{\frac{1}{2}}}$,

z.B. 9 Teilflächen mit Kantenlängen skaliert um $r = \frac{1}{9^{\frac{1}{2}}} = \frac{1}{3}$

Sind N und r bekannt, läßt sich D bestimmen:
$r = \frac{1}{N^{\frac{1}{D}}} \Rightarrow r \cdot N^{\frac{1}{D}} = 1 \Rightarrow N^{\frac{1}{D}} = \frac{1}{r} \Rightarrow \frac{1}{D} \cdot \log(N) = \log(\frac{1}{r})$
$\Rightarrow D = \frac{\log(N)}{\log(\frac{1}{r})}$

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 $r = \frac{1}{3}$.

11.4 Lindenmayer-Systeme

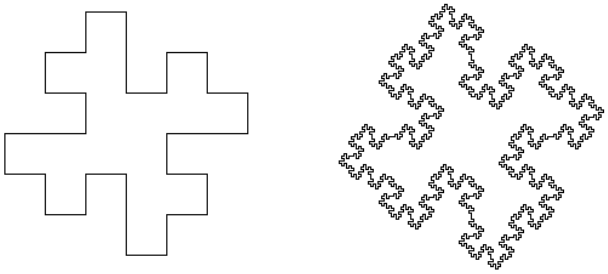
Eine nicht-grafische Beschreibung mancher Fraktale erfolgt mit Lindenmayer-Systemen, kurz *L*-Systems.
Ein Beispiel für ein L-System ist die quadratische Koch-Kurve:

$$\begin{aligned} &\text{Alphabet } \sum = \{r,u,l,d\} \text{ für right, up, left, down} \\ &\text{Regelmenge } f = \{ \quad r \Rightarrow r\,u\,r\,d\,d\,r\,u\,r, \\ &\qquad\qquad\quad u \Rightarrow u\,l\,u\,r\,r\,u\,l\,u, \\ &\qquad\qquad\quad l \Rightarrow l\,d\,l\,u\,u\,l\,d\,l, \\ &\qquad\qquad\quad d \Rightarrow d\,r\,d\,l\,l\,d\,r\,d\,\} \end{aligned}$$

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 $f(w) = r\,u\,r\,d\,d\,r\,u\,r\,u\,l\,u\,r\,r\,u\,l\,u$.

Abbildung 11.4: Anwendung von 2 Regeln eines Lindenmayer-Systems

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 $r = \frac{1}{4}$.



32 Polygonpunkte 16384 Polygonpunkte

Abbildung 11.5: L-System

11.5 Baumstrukturen

Gegeben ein “Ast”, repräsentiert durch ein 5-seitiges Polygon:

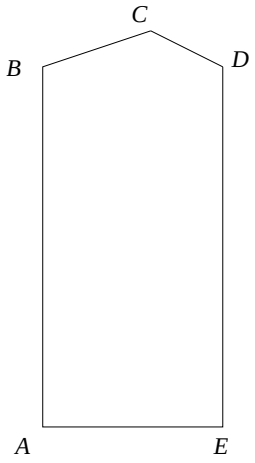
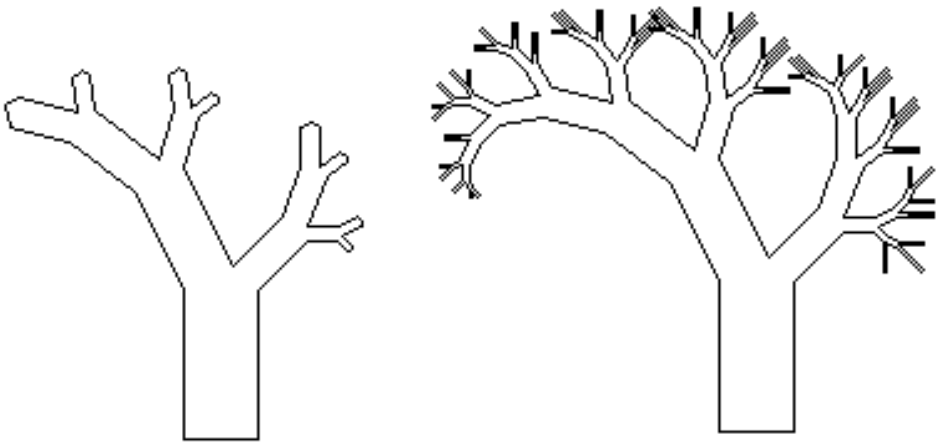


Abbildung 11.6: Ausgangspolygon für Baum

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.



47 Polygonpunkte 3071 Polygonpunkte

Abbildung 11.7: Vom Baum-Algorithmus erzeugter Baum

11.6 Mandelbrot-Menge

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

```
public class Complex
{
        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.

```
Complex f(Complex 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 konvergiert sie zu einem Fixpunkt, für manche c gerät sie in einen (beschränkten) Zyklus, für manche c verhält sie sich (beschränkt) chaotisch, für manche c strebt die Folge gegen Unendlich.

Die *Mandelbrotmenge* besteht aus solchen komplexen Zahlen c, die beim Startwert $z = 0$ zu einer beschränkten Folge führen. Um die Mandelbrotmenge grafisch darzustellen, definiert man

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

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 Complex(Point p, Complex 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).

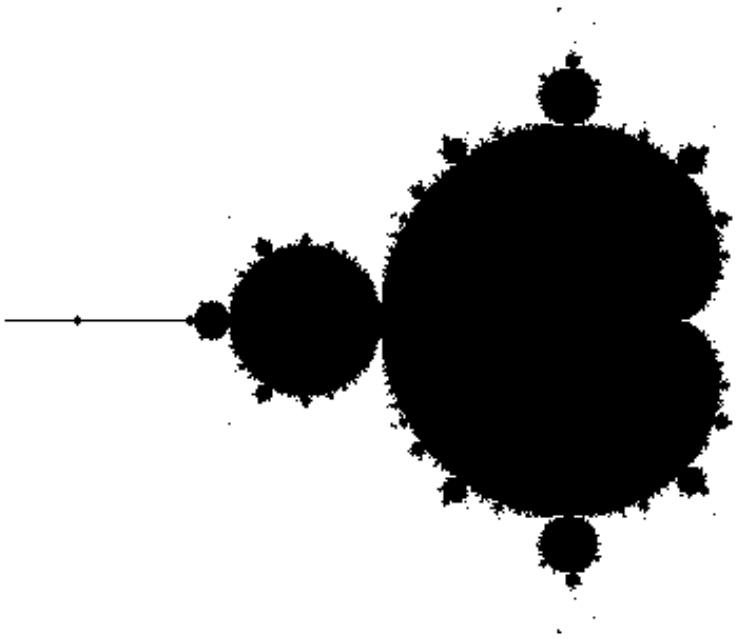


Abbildung 11.8: Mandelbrotmenge: $-2.2 \leq z.re \leq 0.6$, $-1.2 \leq z.im \leq 1.2$, 100 Iterationen

Implementation der Mandelbrot-Menge

Sobald während des Iterierens der Betrag von $z > 2 \Rightarrow$ Folge wächst mit Sicherheit über alle Grenzen (Satz von Fatou). Falls Betrag von z nach z.B. 100 Iterationen noch $< 2 \Rightarrow$ Folge bleibt vermutlich beschränkt.

D.h., bei Erreichen einer vorgegebenen Iterationenzahl wird das Pixel, welches der komplexen Zahl c entspricht, schwarz gefärbt, da c vermutlich zu einer beschränkten Folge führt.

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

     for (p.x = 0; p.x < WIDTH;  p.x++)
     for (p.y = 0; p.y < HEIGHT; p.y++)
     {
          zaehler = 0;
          c       = new Complex(p, start, schritt);
          z       = new Complex(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 zu einer unbeschränkten Folge geführt haben, weiter zu klassifizieren, setzt man

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

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])
```

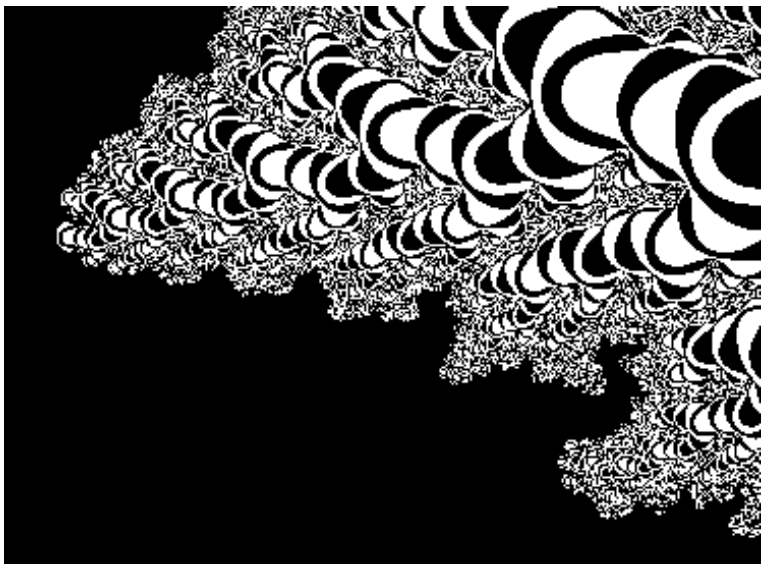


Abbildung 11.9: Mandelbrotmenge: $-1.2548 \leq Re \leq -1.2544$, $0.3816 \leq Im \leq 0.3822$, 100 Iterationen

11.7 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.

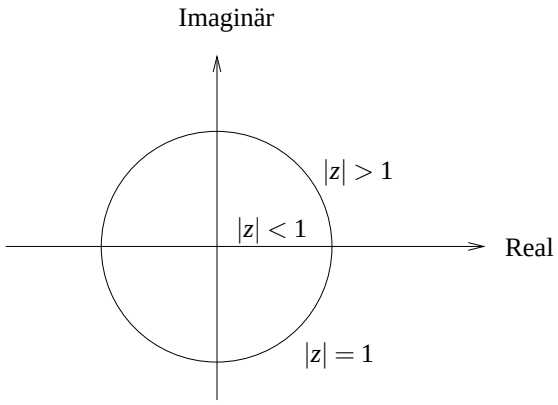


Abbildung 11.10: Julia-Kurve für $f(z) = z^2$

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

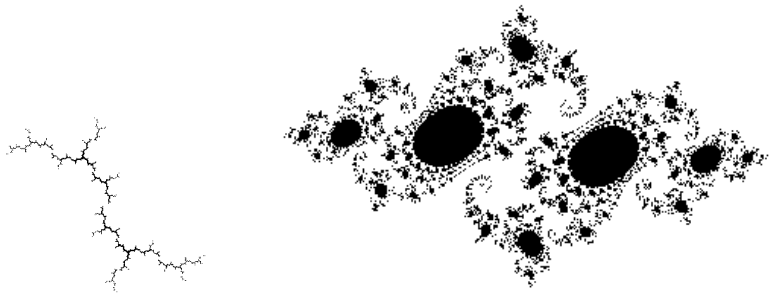


Abbildung 11.11: Vom Julia-Algorithmus erzeugte Bilder mit $c = i$ und $c = 0.746 + 0.112i$

Implementation der Julia-Menge:

1. Möglichkeit: Sei *c* gegeben. Jeder Wert *z* der komplexen Fläche wird überprüft, ob er als Startwert zu einer beschränkten Folge führt:

```
void  julia_voll(         // berechnet Julia-Menge als gefuelltes Gebiet
        Complex c,        // fuer komplexes c,
        Complex 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 ;
        Complex 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 Complex(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:

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

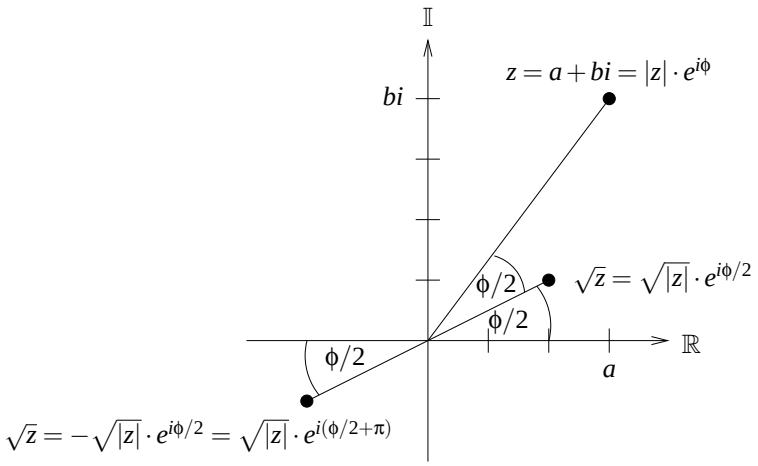


Abbildung 11.12: Zur Berechnung der Wurzel bei der Inverse Iteration Method

Realisiere f^{-1} durch

```
Complex backward_random (Complex z, Complex 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
        Complex c,         // fuer komplexes c
        Complex start,     // linke obere Ecke der komplexen Flaeche
        double schritt,    // Schrittweite
        int anzahl)        // durch anzahl Rueckwaertsspruenge
{
        int k;
        Complex z = new Complex(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
        }

}
```

11.8 Java-Applet zu Fraktalen

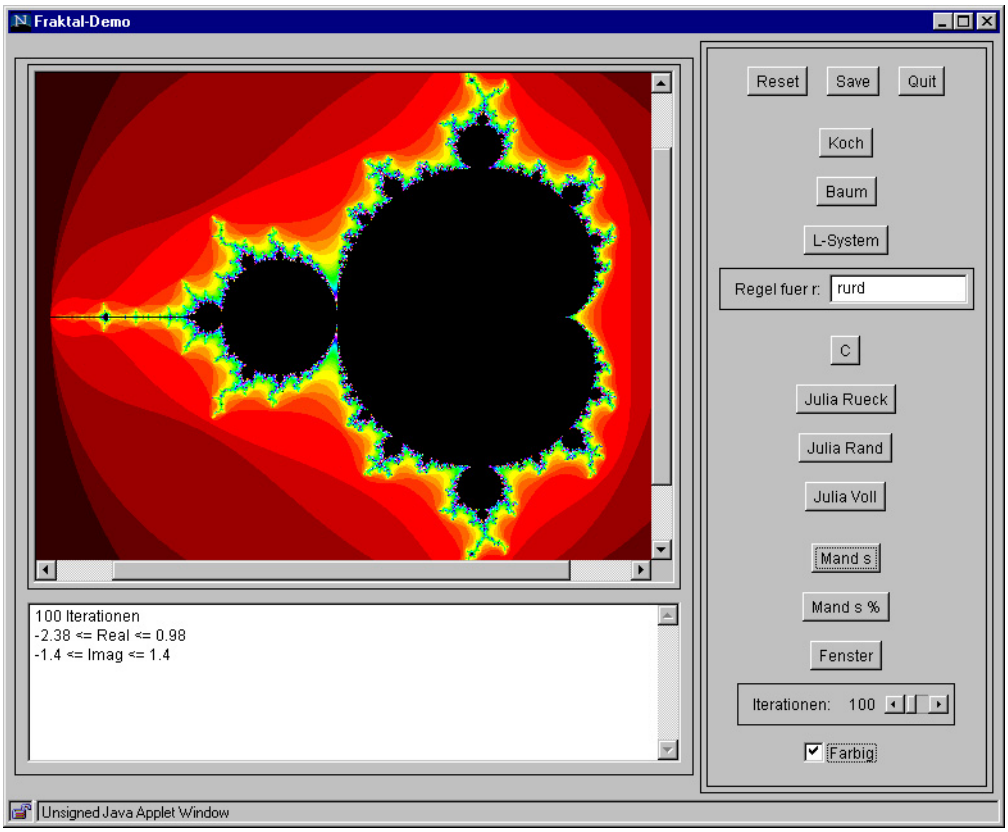


Abbildung 11.13: Screenshot vom Fraktale-Applet

11.9 Iterierte Funktionensysteme

Iterierte Funktionssysteme sind in der Lage, mit wenigen Regeln komplexe, natürlich aussehende Bilder zu erzeugen. Hierbei wird eine Folge von Punkten im $\mathbf{R}^2$ durch fortgesetzte Anwendung von affinen Transformationen durchlaufen. Jede Transformation ist definiert durch eine 2×2- Deformationsmatrix A, einen Translationsvektor b und eine Anwendungswahrscheinlichkeit w, wodurch ein Punkt $\bar{x}$ abgebildet wird auf $A\bar{x} + b$. Zum Beispiel erzeugen folgende 4 Regeln das Farnblatt in Abbildung 11.14 :

	A	b	w
r_1	$\begin{pmatrix} 0.85 & 0.04 \\ -0.04 & 0.85 \end{pmatrix}$	$\begin{pmatrix} 0.0 \\ 1.6 \end{pmatrix}$	85 %
r_2	$\begin{pmatrix} 0.20 & -0.26 \\ 0.23 & 0.22 \end{pmatrix}$	$\begin{pmatrix} 0.0 \\ 1.6 \end{pmatrix}$	7 %
r_3	$\begin{pmatrix} -0.15 & 0.28 \\ 0.26 & 0.24 \end{pmatrix}$	$\begin{pmatrix} 0.0 \\ 0.44 \end{pmatrix}$	7 %
r_4	$\begin{pmatrix} 0.00 & 0.00 \\ 0.00 & 0.16 \end{pmatrix}$	$\begin{pmatrix} 0.0 \\ 0.0 \end{pmatrix}$	1 %

Abbildung 11.14: Vom Iterierten-Funktionen-System erzeugter Farn

Eine alternative Sichtweise zur Definition eines fraktalen Bildes verlangt die Selbstähnlichkeit von Teilen des Bildes mit dem Ganzen.

D_1	
D_2	D_3

Beispielsweise sei ein Rechteck R gesucht, so daß sich sein Inhalt, jeweils auf ein Viertel verkleinert, im linken oberen, linken unteren und rechten unteren Quadranten wiederfindet. Beginnend mit einem beliebigen Rechteckinhalt werden die 3 Quadranten so lange als skalierte Versionen des Gesamtrechtecks ersetzt, bis wir nahe am Fixpunkt dieser Iterationen angelangt sind. Das Ergebnis ist das sogenannte *Sierpinsky-Dreieck*.

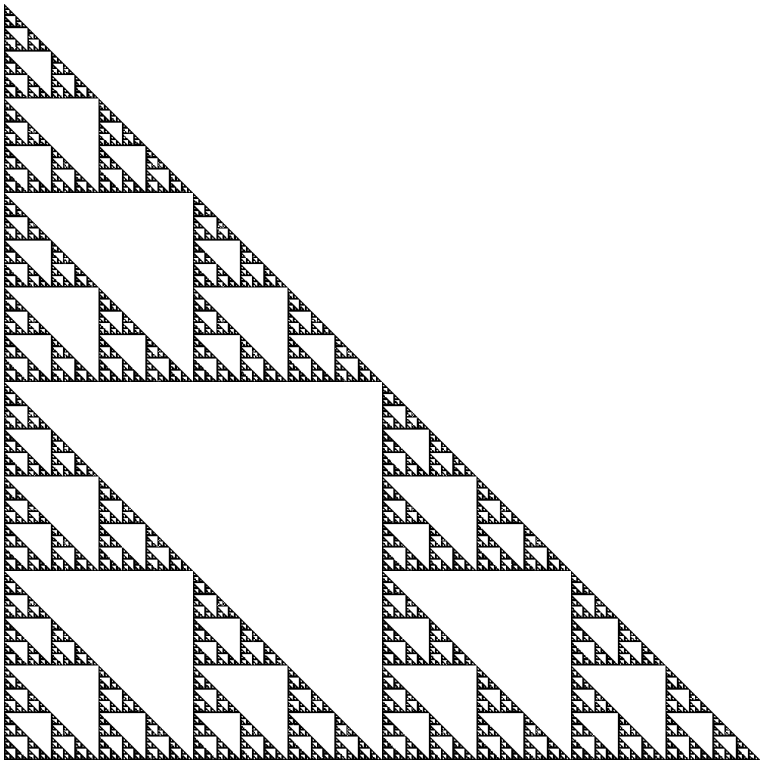


Abbildung 11.15: Sierpinsky-Dreieck

11.10 Java-Applet zu Iterierten Funktionensystemen

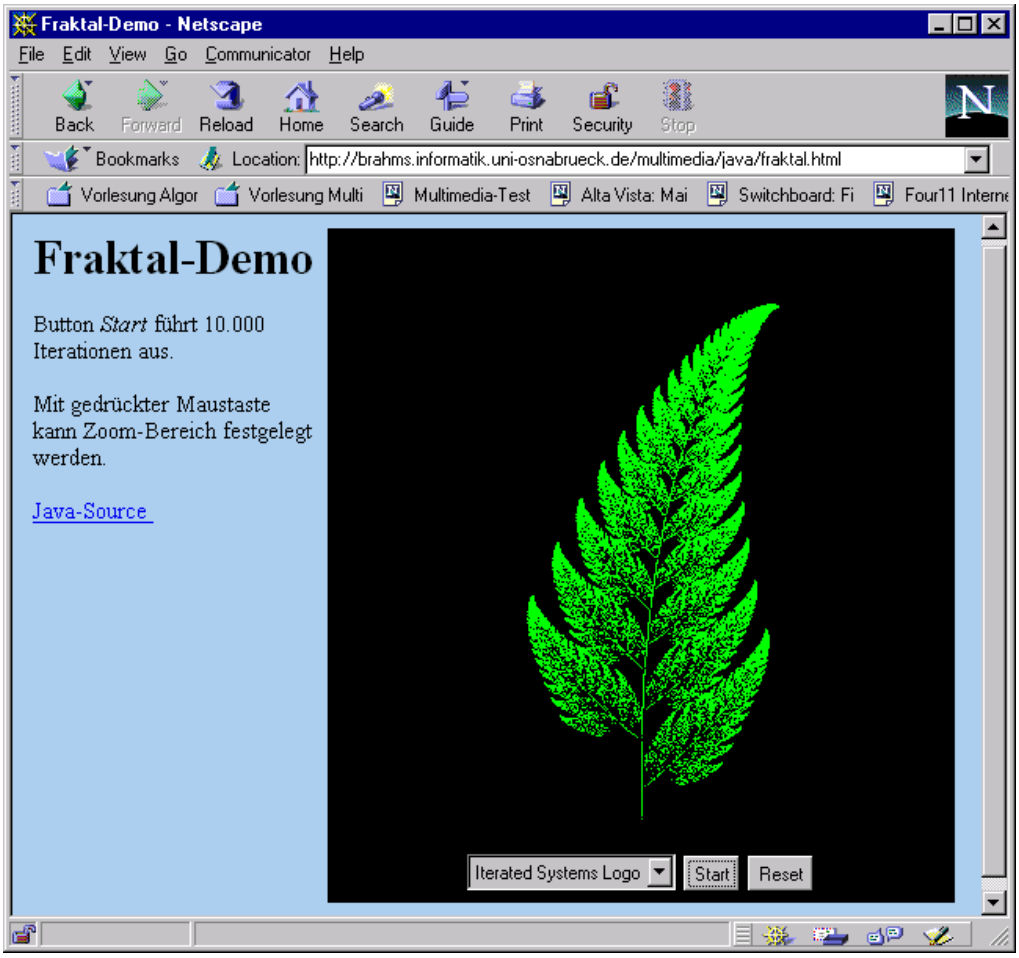


Abbildung 11.16: Screenshot vom Iterierte-Funktionen-Systeme-Applet

Kapitel 12

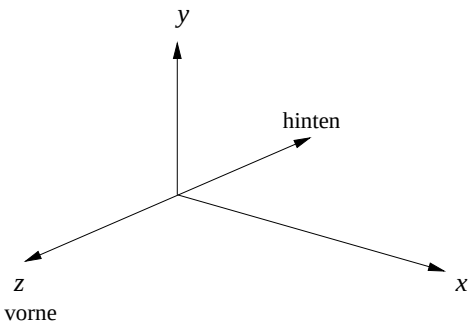
Mathematische Grundlagen

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

12.1 3D-Koordinatensystem

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

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



12.2 Länge und Kreuzprodukt

Gegeben sei ein 3D-Vektor

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

Seine **Länge** ist definiert als

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

Gegeben seien zwei 3D-Vektoren

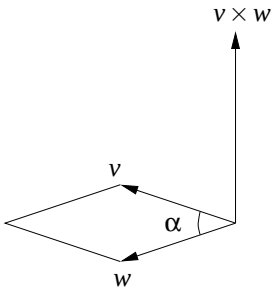
$$\vec{v} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \text{ und } \vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$$

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

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

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

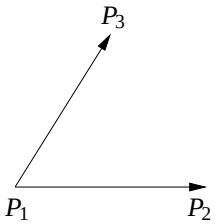
$$|\vec{v} \times \vec{w}| = |\vec{v}| \cdot |\vec{w}| \cdot \sin(\alpha)$$



In einem rechtshändigen Koordinatensystem entspricht bei gespreizten Fingern der Daumen $\vec{v}$, der Zeigefinger $\vec{w}$, der Mittelfinger $\vec{v} \times \vec{w}$. Ein Applet zur Visualisierung liegt unter `http://www-lehre.inf.uos.de/~cg/2008/skript/Applets/CrossProduct`.

Anwendung des Kreuzprodukts

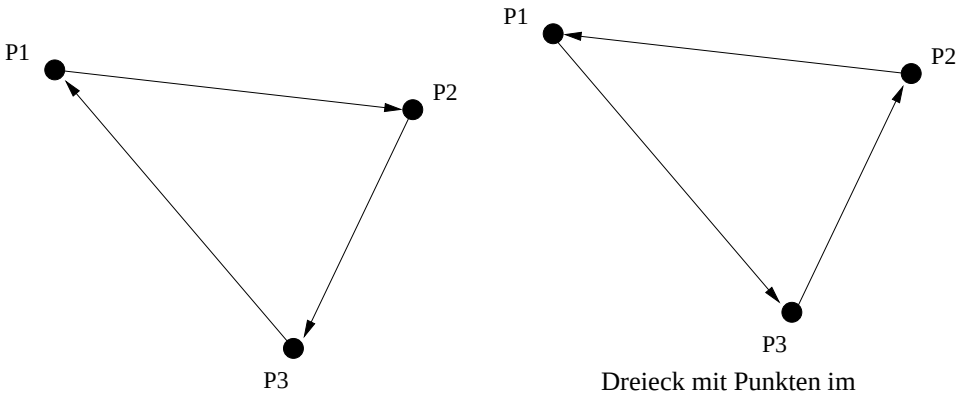
Gegeben sei eine Fläche durch 3 nicht kollineare Punkte P_1, P_2, P_3.



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

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

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



Dreieck mit Punkten im Uhrzeigersinn

Dreieck mit Punkten im Gegenuhrzeigersinn

Umlaufsinn bei Polygonen

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

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

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

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

Es gilt also:

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

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

Wir werden alle Polygone gegen den Uhrzeigersinn orientieren. Für konvexe Polygone gilt dann: Die Normale einer Fläche, die man so betrachtet, daß sie gegen den Uhrzeigersinn orientiert ist, zeigt in Richtung des Betrachters. Man "sieht" die Seite der Fläche, die die "Außenseite" sein soll.

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

12.3 Skalarprodukt

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

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

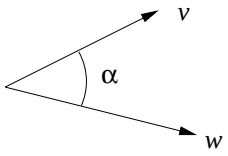
Für Vektoren $\vec{a}, \vec{b}, \vec{c}$ und $s \in \mathbb{R}$ gilt:

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

Anwendungen des Skalarprodukts:

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

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



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

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

12.4 Matrixinversion

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

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

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

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

Beispiel:

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

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

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

12.5 Wechsel eines Koordinatensystems

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

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

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

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

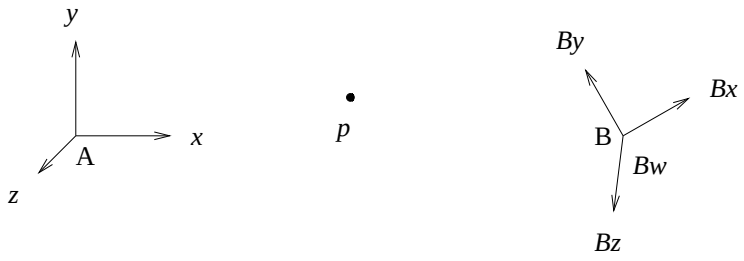
$$\mathcal{A}_{kart} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

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

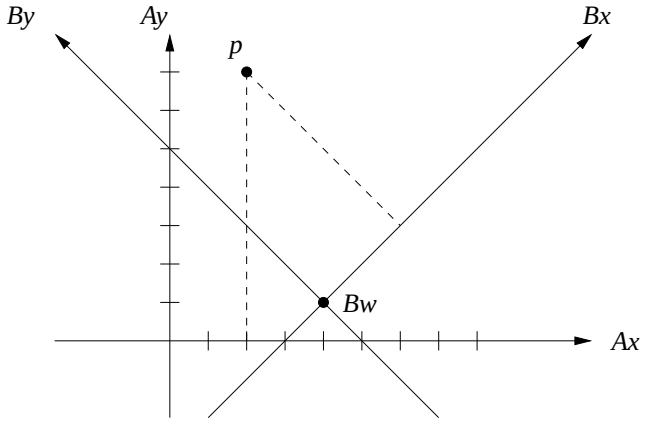
Diese Elemente lassen sich sowohl bzgl. der Basis $\mathcal{A}$ als auch bzgl. der Basis $\mathcal{B}$ darstellen. Um von einer Darstellung in die andere zu kommen, muß die gegebene Darstellung nur mit der im Folgenden beschriebenen Matrix multipliziert werden.

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

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



Beispiel (für den 2-dimensionalen Fall):



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

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

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

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

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

$$M_{\mathcal{B}\to\mathcal{A}} \cdot \vec{p}_{\mathcal{B}} = \vec{p}_{\mathcal{A}}$$

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

Kapitel 13

3D-Transformationen

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

13.1 Translation

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

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

in der folgenden Form darstellen:

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

mit

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

13.2 Skalierung

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

Es liege der Fixpunkt im Ursprung:

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

Die daraus resultierende Transformationsmatrix lautet:

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

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

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

Die Transformationsmatrix lautet:

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

13.3 Rotation

Rotation um die z-Achse

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

Die daraus resultierende Transformationsmatrix lautet:

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

Rotation um die x-Achse

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

Die daraus resultierende Transformationsmatrix lautet:

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

Rotation um die y-Achse

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

Die daraus resultierende Transformationsmatrix lautet:

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

Rotation um eine beliebige Achse

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

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

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

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

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

Die Länge dieses Vektors lautet

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

Die Komponenten des zugehörigen Einheitsvektors

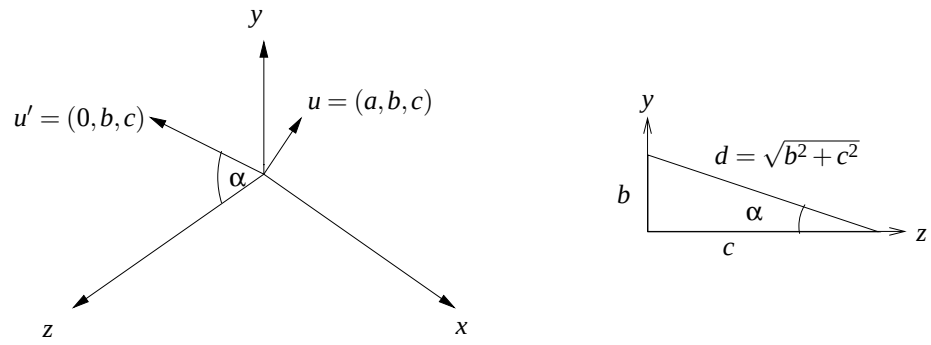
$$\vec{u} = \frac{\vec{v}}{|\vec{v}|} = \begin{pmatrix} a \\ b \\ c \end{pmatrix}, |\vec{u}| = 1$$

lauten daher

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

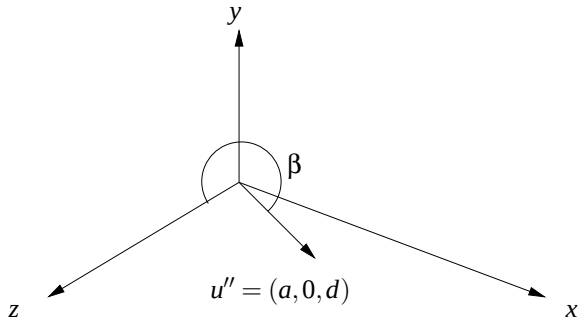
Schritt 1 läßt sich durch die Translation $T(-x_1, -y_1, -z_1)$ durchführen. Dadurch wird P_1, Ausgangspunkt des Einheitsvektors $\vec{u}$, in den Ursprung verschoben.

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

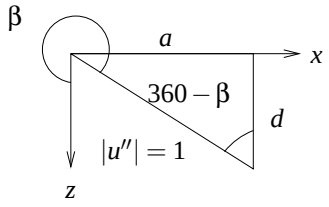


$\Rightarrow \cos(\alpha) = \frac{c}{d}$
$\Rightarrow \sin(\alpha) = \frac{b}{d}$

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



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



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

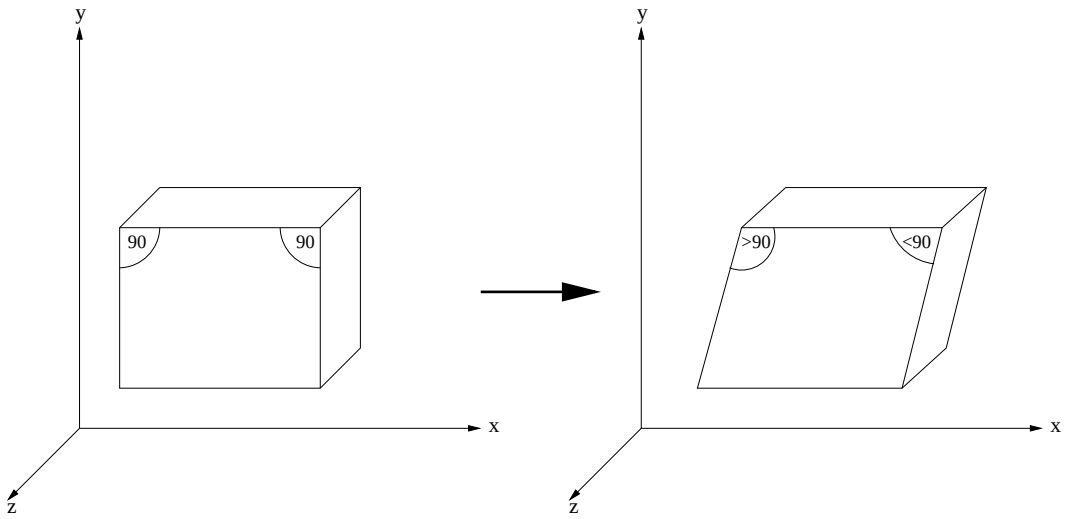
Nach den ersten drei Schritten ist die Drehachse mit der z-Achse identisch, so daß Schritt (4) mit der Rotationsmatrix $R_z(\delta)$ durchgeführt werden kann. Schritt (5) beinhaltet die Anwendung der inversen Transformationen.

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

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

13.4 Transformation der Normalenvektoren

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



Winkeluntreue unter nicht uniformer Skalierung

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

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

Seien P_1, P_2 zwei Punkte der Ebene mit Normalenvektor $\vec{n}$. Sei $\vec{r} = P_2 - P_1$.

Offenbar gilt

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

Daraus folgt

$$\Rightarrow \vec{n}^T \cdot M^{-1}M \cdot \vec{r} = 0$$

Durch zweimaliges Transponieren erhält man

$$((M^{-1})^T \cdot \vec{n})^T \vec{r'} = 0$$

Für den transformierten Vektor $\vec{n'}$ muss offenbar gelten

$$\vec{n'}^T \cdot \vec{r'} = 0$$

Aus den beiden Gleichungen folgt daher

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

Also muss bei einer Fläche der Normalenvektor $\vec{n}$ mit der transponierten Inversen der Transformationsmatrix M transformiert werden.

Kapitel 14

Projektion

14.1 Bildebene

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

Gegeben sind

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

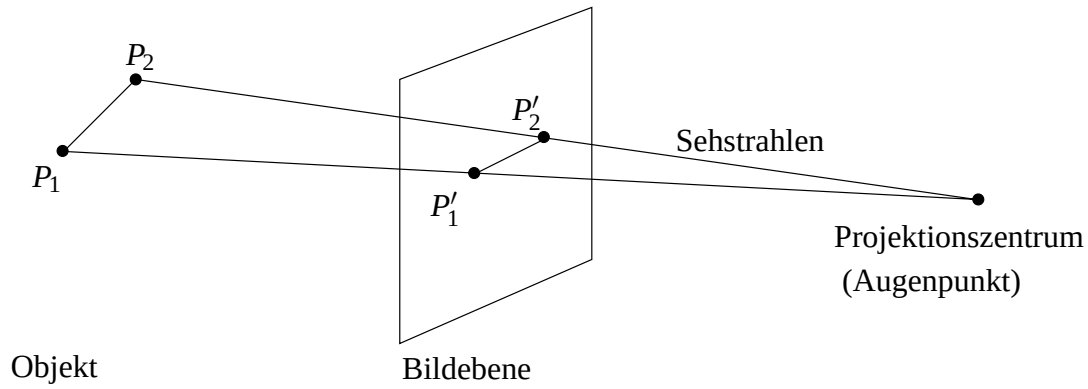
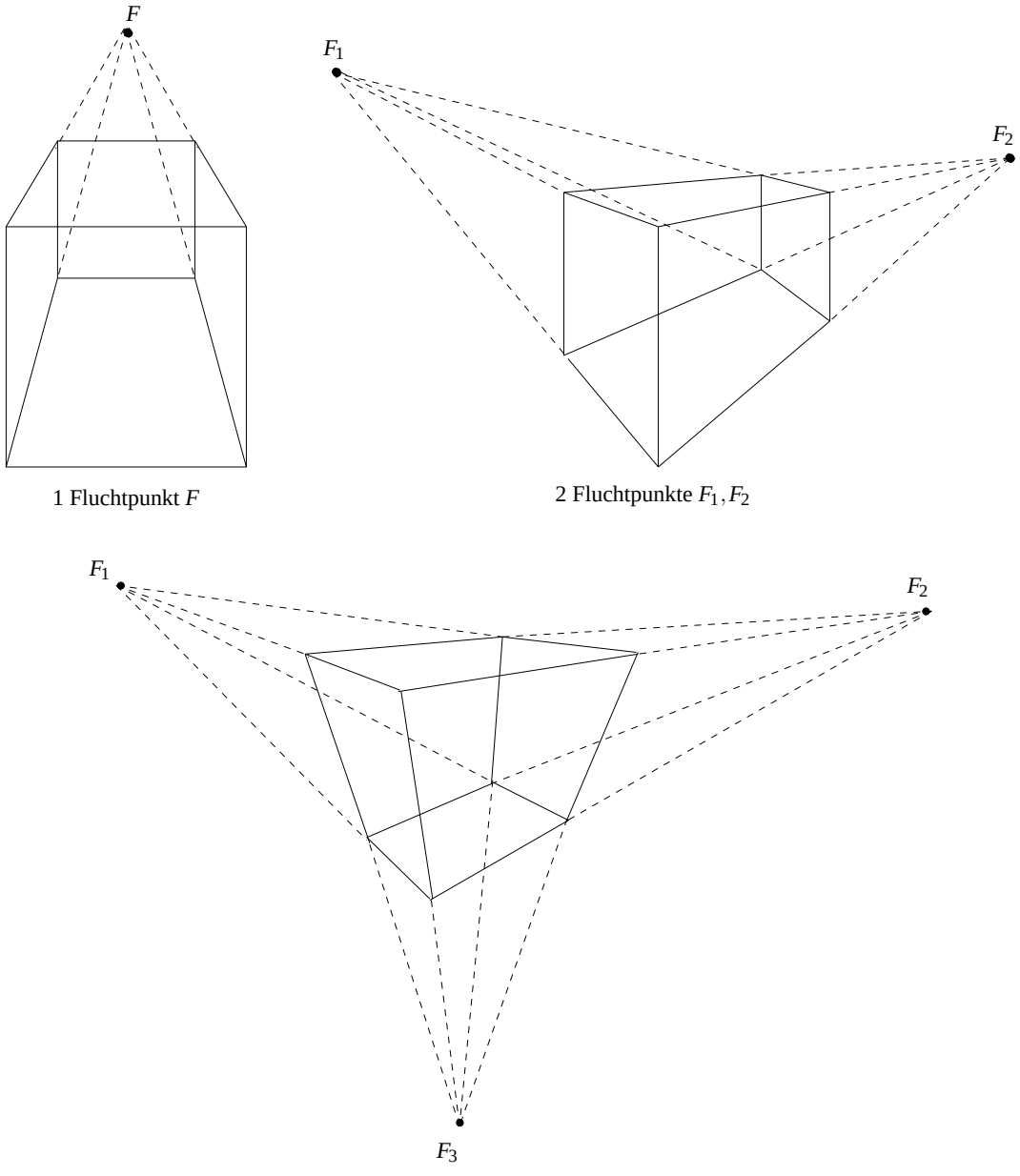


Abbildung 14.1: Zentralprojektion

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

14.2 Perspektivische Projektion

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



1 Fluchtpunkt F

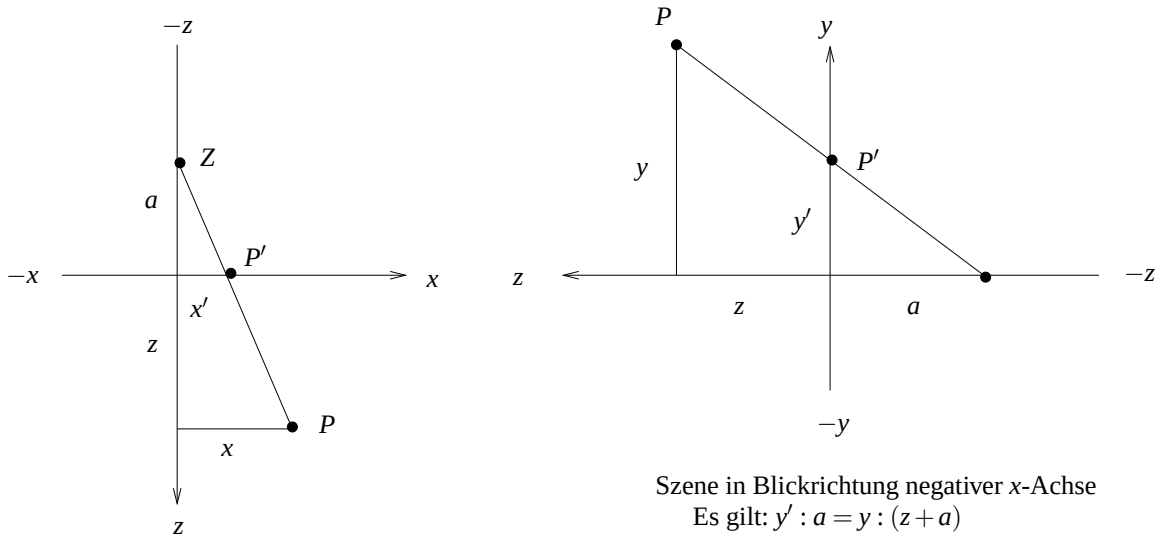
2 Fluchtpunkte F_1, F_2

3 Fluchtpunkte F_1, F_2, F_3

Abbildung 14.2: Zentralprojektionen mit unterschiedlich vielen Fluchtpunkten

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

Koordinaten des projizierten Bildpunktes $P' = (x', y', 0)$.



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

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

Abbildung 14.3: Anwendung der Strahlensätze

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

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

Die homogenen Koordinaten des projizierten Punktes lauten

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

Für die Transformationsmatrix der perspektivischen Projektion ergibt sich also:

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

14.3 Parallelprojektion

14.3.1 Normalprojektionen

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

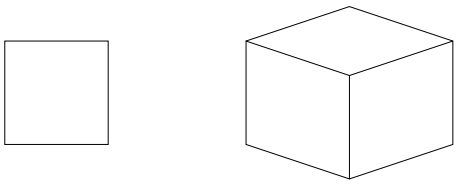


Abbildung 14.4: Grund-, Seiten-, Aufriss (a) bzw. axononometrische Projektion (b)

Die häufigste Anwendung orthogonaler Projektionen liegt in der Darstellung eines Objekts durch *Grund-*, *Auf-* und *Seitenriß* (Abbildung 14.4 a).

Eine weitere Form der Normalprojektionen sind die *axonometrischen Projektionen,* bei denen die Bildebene auf keiner der Körper-Hauptachsen senkrecht steht. Dadurch sind in der Abbildung mehrere zueinander normal stehende Flächen gleichzeitig sichtbar. Die resultierenden Darstellungen sind der perspektivischen Projektion ähnlich. Anstelle der zur Entfernung vom Auge proportionalen Verkürzung erfolgt aber eine gleichmäßige Verkürzung aller Kanten (Abbildung 14.4 b).

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

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

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

14.3.2 Schiefe Projektionen

Bei den *schiefen* Parallelprojektionen stehen die Sehstrahlen nicht normal auf der Bildebene, sondern schneiden sie unter dem Winkel β (Abbildung 14.5). Die schiefe Projektion auf die *xy*-Ebene entspricht einer Scherung der *x*- und *y*-Koordinaten proportional zu *z*.

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

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

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

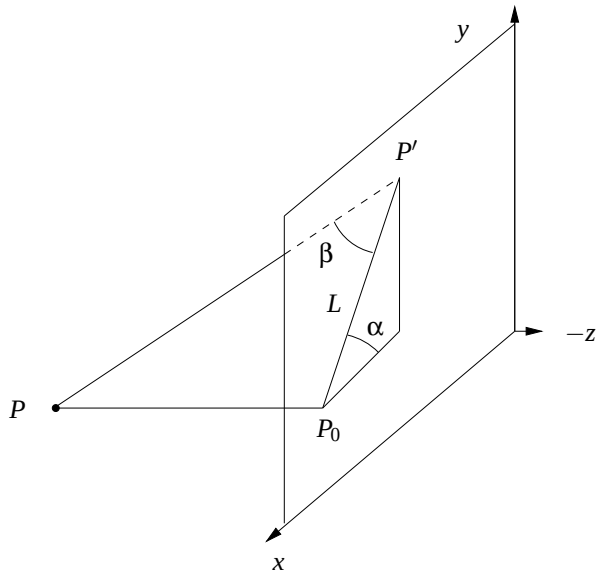


Abbildung 14.5: Bildebene bei der schiefen Projektion

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

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

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

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

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

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

für den Abstand

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

ergibt sich

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

wegen

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

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

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

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

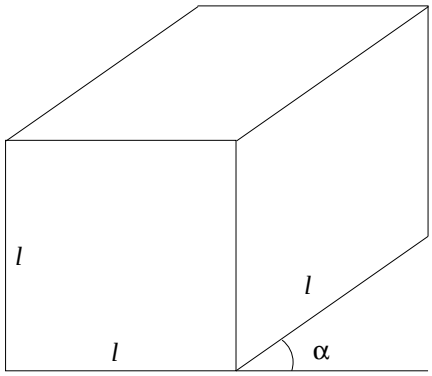
$$\begin{aligned} x' &= x - z \cdot (d \cdot \cos(\alpha)), \\ y' &= y + z \cdot (d \cdot \sin(\alpha)), \\ z' &= 0, \\ w' &= 1 \end{aligned}$$

Die entsprechende Transformationsmatrix lautet

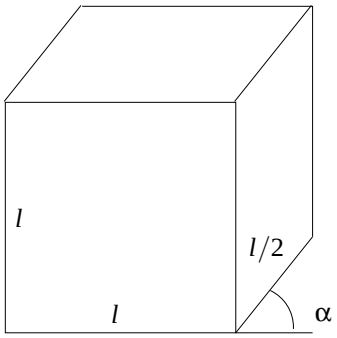
$$P_{schief_{xy}}(\alpha, d) = \begin{bmatrix} 1 & 0 & -d \cdot \cos(\alpha) & 0 \\ 0 & 1 & d \cdot \sin(\alpha) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

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

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



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



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

Abbildung 14.6: Zwei schiefe Projektionen

Kapitel 15

Viewing Pipeline

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

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

15.1 Die synthetische Kamera

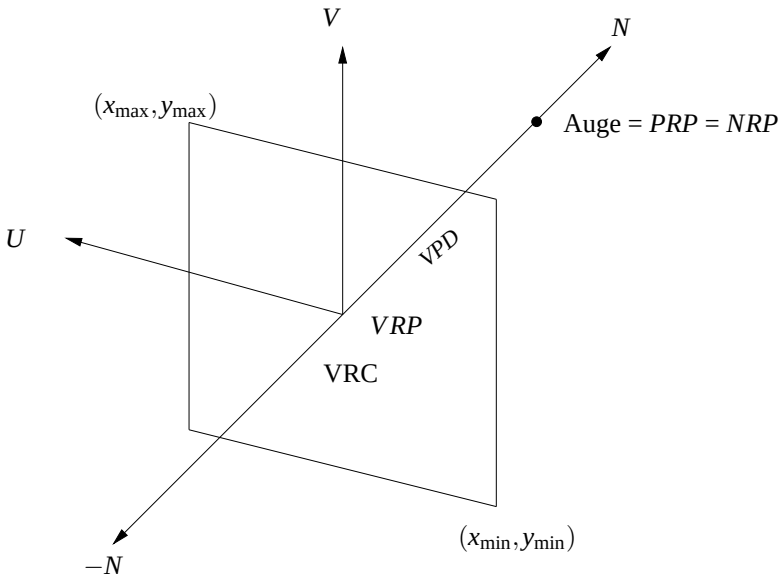


Abbildung 15.1: Parameter zur Beschreibung der synthetischen Kamera

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

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

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

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

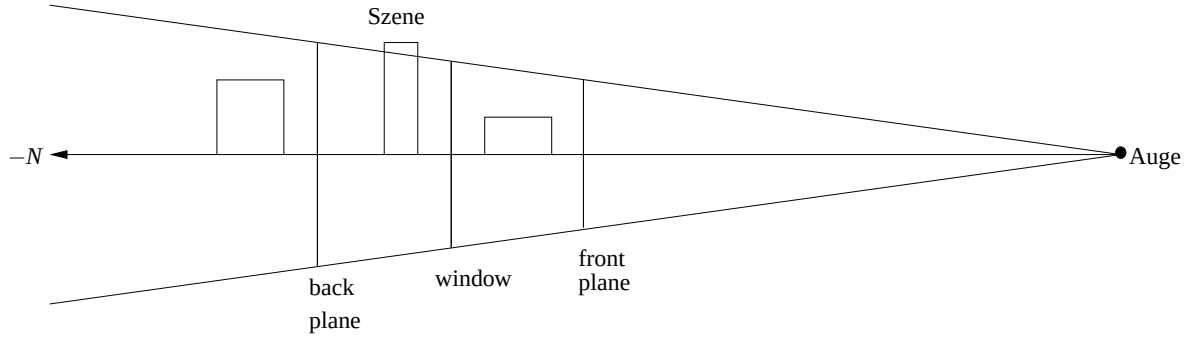


Abbildung 15.2: Einschränkung des sichtbaren Teils

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

15.2 Viewing Pipeline

Die fotorealistische Darstellung von 3D-Objekten einer Szene auf den 2D-Bildschirm läßt sich beschreiben durch eine Sequenz von Transformationen, genannt *Viewing Pipeline*.

Folgende Informationen sind erforderlich:

- Jedes Objekt wird beschrieben durch Modellkoordinaten (z.B. Würfel ist beschrieben durch Mittelpunkt $(0,0,0)$ und Kantenlänge 1).
- Die Szene wird beschrieben durch eine Menge von Objekten, deren Lage und Größe in Weltkoordinaten beschrieben sind.

- Die Beleuchtung wird beschrieben durch eine Menge von Lichtquellen, der Lage und Ausrichtung 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 → WC
Beschreibe Polygonpunkte in Weltkoordinaten (dies geschieht durch Translation, Rotation und Skalierung).

2.) View Orientation: WC → 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 → NPC
Transformiere die Szene derart, daß der sichtbare (= im Pyramidenstumpf aus view window, front plane, back plane liegende) Teil abgebildet wird auf einen Einheitswürfel, dessen Vorder- und Rückseite der *front plane* bzw. *back plane* entsprechen.

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

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 *PRP* (*Projection Reference Point*; auch *Eyepoint*),
- Blickrichtung *VRP* (*View Reference Point*),
- vertikale Orientierung *VUV* (*View Up Vector*) oder *VUP* (*View Up Point*),
- Blickwinkel (Brennweite).

Der fiktive Betrachter blickt vom *PRP* in Richtung *VRP* und kippt dabei die Kamera so um die Blickrichtung, daß *VUP* von ihm aus gesehen vertikal nach oben zeigt. Der Blickwinkel entspricht der Brennweite und regelt den Bildausschnitt (*Zooming*).

Durch diese Parameter wird das VRC (*View Reference Coordinate System*) definiert mit *VRP* als Ursprung und *PRP* auf der positiven *z*-Achse. Die *xy*-Ebene wird als *View Plane* bezeichnet und steht senkrecht auf der Blickrichtung. Die *y*-Achse ist gegeben durch die Projektion von *VUP* in die View Plane. Deshalb darf *VUP* nicht parallel zur Blickrichtung sein.

Das VRC hat die gleiche Metrik wie das WC. Es handelt sich also um einen Wechsel des Koordinatensystems. Die entsprechende Matrix, die den Wechsel vom WC ins VRC durchführt ergibt sich, wenn man die Vektoren *U*, *V* und *N* und die homogenen Koordinaten des *VRP* bzgl. des WC als Spaltenvektoren nebeneinander schreibt und diese Matrix anschließend invertiert.

Der Abstand zwischen *VRP* und *PRP* wird als *VPD* (*View Plane Distance*) bezeichnet.

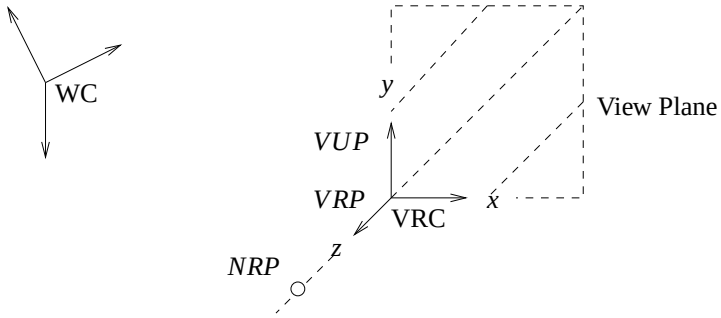


Abbildung 15.3: Definition des VRC

Für die Anwendung des Viewings ergeben sich im Programm verschiedene Möglichkeiten:

- Soll ein Objekt von verschiedenen Seiten betrachtet werden, so wird *VRP* in das Objekt gelegt. Durch *PRP* läßt sich nun die Blickrichtung frei wählen. Der Vektor *VUP* kann dabei konstant bleiben, sofern nicht senkrecht von oben oder unten geschaut wird.
- Soll sich die Szene bei konstantem Standpunkt um den Betrachter drehen, wird *PRP* festgesetzt. Durch Veränderung des *VRP* bewegt sich dann die Szene vor dem Auge des Betrachters. Der *VUP* kann dabei konstant bleiben, solange er nicht kollinear zum Vektor durch *VRP* und *PRP* wird.
- Eine weitere Möglichkeit sind Animationen, bei denen sich z.B. der Beobachter vorwärts in die Szene hineinbewegt. Dieser Effekt läßt sich durch Verschieben von *VRP* und *PRP* um denselben Vektor erzielen.

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.

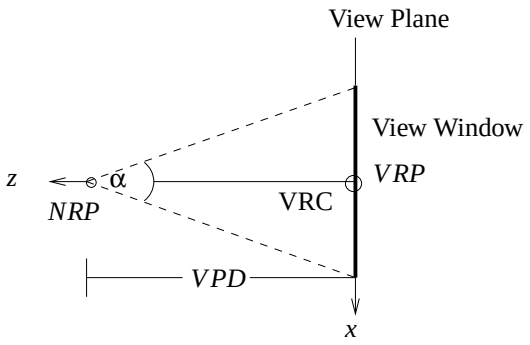


Abbildung 15.4: Definition des View Windows

Die Form des Bildraumes ist bei der perspektivischen Projektion eine unendliche Pyramide, deren Spitze im Projektionszentrum *PRP* (*Projection Reference Point*) liegt und deren Kanten durch die Eckpunkte des View Windows verlaufen. Hierbei wird im Programm der Betrachterstandpunkt *PRP* als Projektionszentrum gewählt.

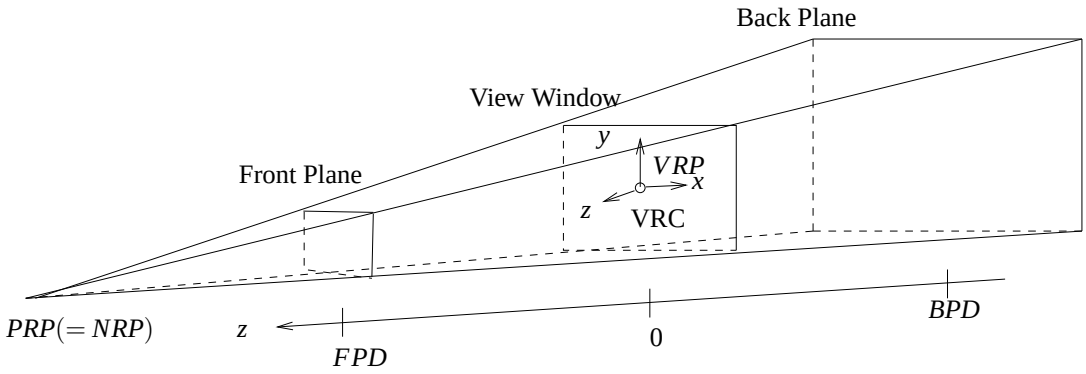


Abbildung 15.5: Bildraum bei perspektivischer Projektion

Die Begrenzung des Bildraumes in Richtung der z-Achse des VRC erfolgt durch zwei zur View Plane parallele Ebenen, die *Front* und *Back Plane* genannt werden. Die Front Plane liegt vom Projektionszentrum PRP aus gesehen vor der Back Plane. Ihre z-Komponenten werden als *Front Plane Distance* (FPD) und *Back Plane Distance* (BPD) bezeichnet. Es gilt $BPD < FPD$. Mit diesen Ebenen können Teile der Szene von der Projektion auf die Bildebene ausgeschlossen werden. Speziell bei der perspektivischen Projektion erweist sich diese Möglichkeit als notwendig, da sonst sehr nahe Objekte

alle anderen verdecken bzw. sehr weit entfernte als nicht mehr erkennbare (d.h. zu kleine) Figuren abgebildet würden. Mit Front und Back Plane ergibt sich als View Volume ein Pyramidenstumpf.

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".

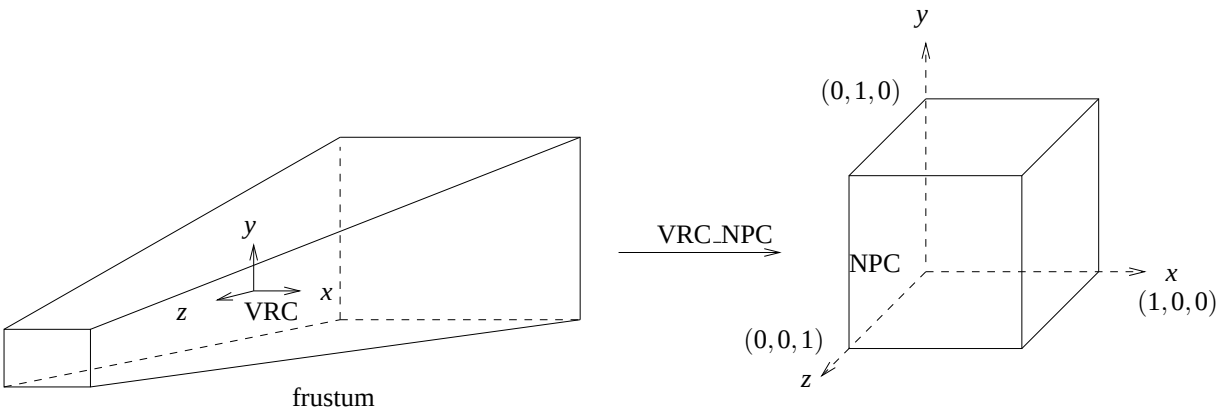


Abbildung 15.6: View Mapping

Bei der perspektivischen Projektion wird in einem ersten Schritt der Pyramidenstumpf, der den Bildraum darstellt und *frustum* genannt wird, auf einen regelmäßigen Pyramidenstumpf abgebildet. Dessen Grundfläche ist ein Quadrat, das mit jeder Seitenfläche einen Winkel von 45° einschließt. Dabei werden die z-Koordinaten nicht und die x- und y-Koordinaten proportional zur z-Koordinate verändert, was einer Scherung an der z-Achse entspricht. Im zweiten Schritt wird der regelmäßige Pyramidenstumpf in den normierten Einheitswürfel transformiert. Dazu müssen die x- und y-Koordinaten proportional zu den reziproken z-Werten skaliert werden. Die Front Plane entspricht im NPC der Ebene $z = 1$ und die Back Plane der Ebene $z = 0$.

Beide Schritte werden im Programm zu einer Transformationsmatrix zusammengefaßt. Bei deren Anwendung auf die homogenen Koordinaten ist zu beachten, daß die resultierenden Punkte im allgemeinen w-Werte ungleich 1 haben, die affinen Koordinaten also erst nach der Division durch w vorliegen. Außerdem ist diese Abbildung nur bezüglich der x- und y-Koordinaten linear. In z-Richtung werden die Werte durch die Scherung im zweiten Schritt reziprok verzerrt, d.h., äquidistante Punkte längs der z-Achse im VRC häufen sich im NPC bei $z = 0$ nahe der Back Plane.

Zur Erleichterung der Herleitung wird zunächst das Koordinatensystem so transformiert, daß der *PRP* im Ursprung sitzt. Danach wird das Koordinatensystem an der *xy*-Ebene gespiegelt, indem die *z*-Koordinaten mit -1 multipliziert werden. Danach ist das Koordinatensystem linkshändig. Es seien d_{min}, d und d_{max} die Abstände der Frontplane, Bildebene und Backplane vom Augenpunkt.

Zur Durchführung von Punkt 3.) der Viewing Pipeline wird zunächst die abgeschnittene Pyramide (= frustum) transformiert in einen symmetrischen Pyramidenstumpf mit quadratischer Grundfläche und Kanten unter 45°.

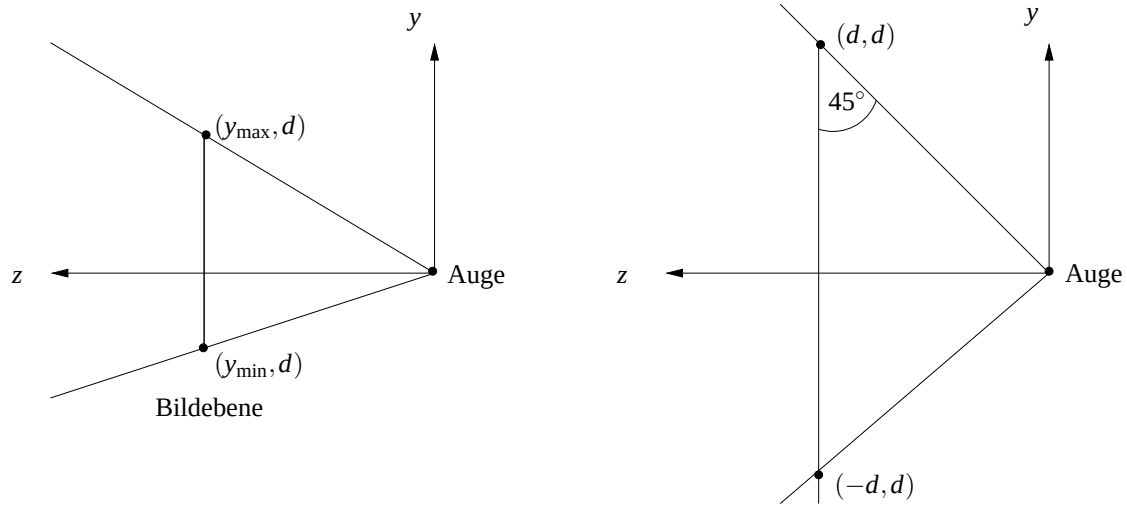


Abbildung 15.7: Überführung in Pyramidenstumpf

Es müssen also die *y*-Koordinaten proportional zur *z*-Koordinate verändert werden. Dies entspricht einer Scherung an der *z*-Achse, so daß die Achse vom *PRP* zum Zentrum des ViewWindows mti der *z*-Achse zusammenfällt. Zusätzlich wird in *y*-Richtung so skaliert, daß die Grundseite des entstandenen Pyramidenstumpfs eine Kantenlänge von $2d$ bekommt:

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

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

Durch Lösen des Gleichungssystems

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

erhält man

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

Analoge Überlegungen für die *x*-Werte ergibt:

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

Als nächstes wird die regelmäßige Pyramide in den Einheitswürfel $(0 \leq x \leq 1, 0 \leq y \leq 1, 0 \leq z \leq 1)$ transformiert. Die *front plane* entspricht der Ebene $z = 0$ und die *back plane* der Ebene $z = 1$.

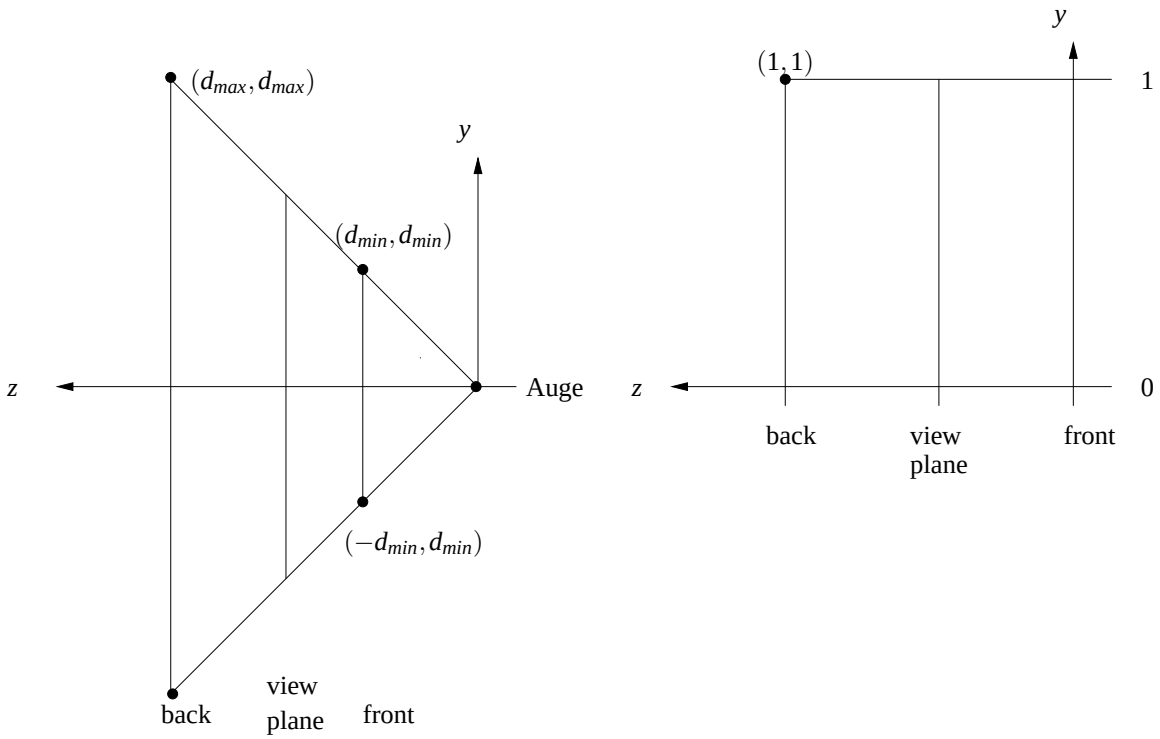


Abbildung 15.8: Überführung in Einheitswürfel

Da die y-Werte proportional zu den reziproken z-Werten skaliert werden müssen, ergibt sich als Transformation

$$\begin{aligned} y' &= k_1 + k_2 \cdot \frac{y}{z} \\ z' &= k_3 + k_4 \cdot \frac{1}{z} \end{aligned}$$

Der Kehrwert von z im Term zu y' ist nur möglich, indem durch einen geeigneten Eintrag in der vierten Zeile der noch zu konstruierenden Transformationsmatrix erreicht wird, dass die vierte Komponente des transformierten Punktes den Wert z enthält. Bei der üblichen Auswertung einer homogenen Koordinate wird dann durch z geteilt. Durch diesen Trick müssen aber neben dem Term für y' und dem für x' auch der Term zu z' den Kehrwert von z eingebaut bekommen.

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

$$1 = k_1 + k_2 \cdot \frac{d_{\max}}{d_{\max}}$$

$$0 = k_1 + k_2 \cdot \frac{-d_{\min}}{d_{\min}}$$

erhält man

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

Durch Lösen des Gleichungssystems

$$\begin{aligned} 1 &= k_3 + k_4 \cdot \frac{1}{d_{\max}} \\ 0 &= k_3 + k_4 \cdot \frac{1}{d_{\min}} \end{aligned}$$

erhält man

$$k_3 = \frac{d_{\max}}{d_{\max} - d_{\min}}, \quad k_4 = -\frac{d_{\min} \cdot d_{\max}}{d_{\max} - d_{\min}}.$$

Die x-Werte werden analog zu den y-Werten skaliert. Insgesamt ergibt sich somit

$$\begin{aligned} x' &= \frac{1}{2} + \frac{x}{2} \cdot \frac{1}{z} \\ y' &= \frac{1}{2} + \frac{y}{2} \cdot \frac{1}{z} \\ z' &= \frac{d_{\max}}{d_{\max} - d_{\min}} - \frac{d_{\min} \cdot d_{\max}}{d_{\max} - d_{\min}} \cdot \frac{1}{z}. \end{aligned}$$

Zwar wird hierdurch die Szene im vorderen z-Bereich nicht-linear gestaucht, zur Bestimmung der Sichtbarkeit reichen die ermittelten z-Werte jedoch aus, da ihre Ordnung erhalten bleibt.

Durch Verknüpfen der beiden letzten Transformationen erhält man in Schritt 3.) als Transformationsmatrix

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

die den sichtbaren Teil der Szene in den Einheitswürfel transformiert.

Um wieder ein rechtshändiges Koordinatensystem zu erhalten, wird zunächst das Koordinatensystem so transformiert, daß die Back Plane in die xy-Ebene verschoben wird. Abschließend wird wieder an der xy-Ebene gespiegelt.

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.

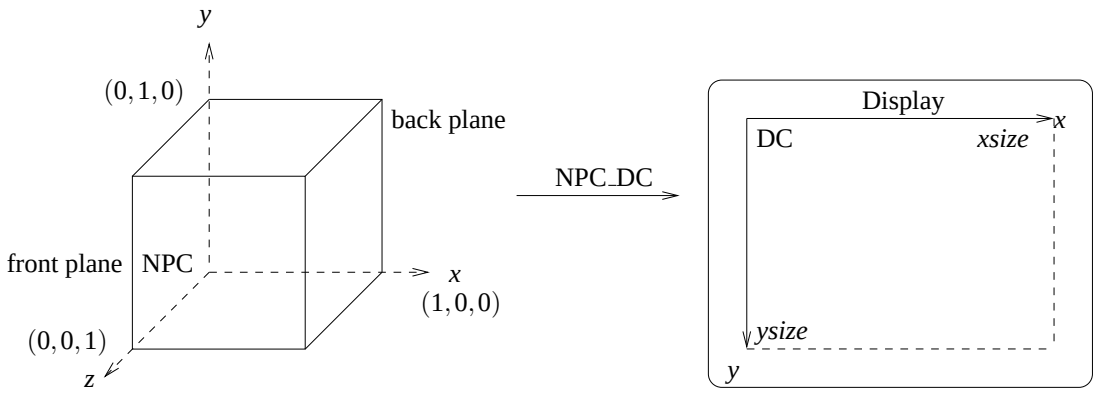


Abbildung 15.9: Device Mapping

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

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

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

15.2.6 Zusammenfassung

Abbildung 15.10 zeigt den Ablauf der Transformationen im Überblick. Diese *Viewing Pipeline* wird von den Eckpunkten aller Polygone durchlaufen, aus denen sich die Szene zusammensetzt.

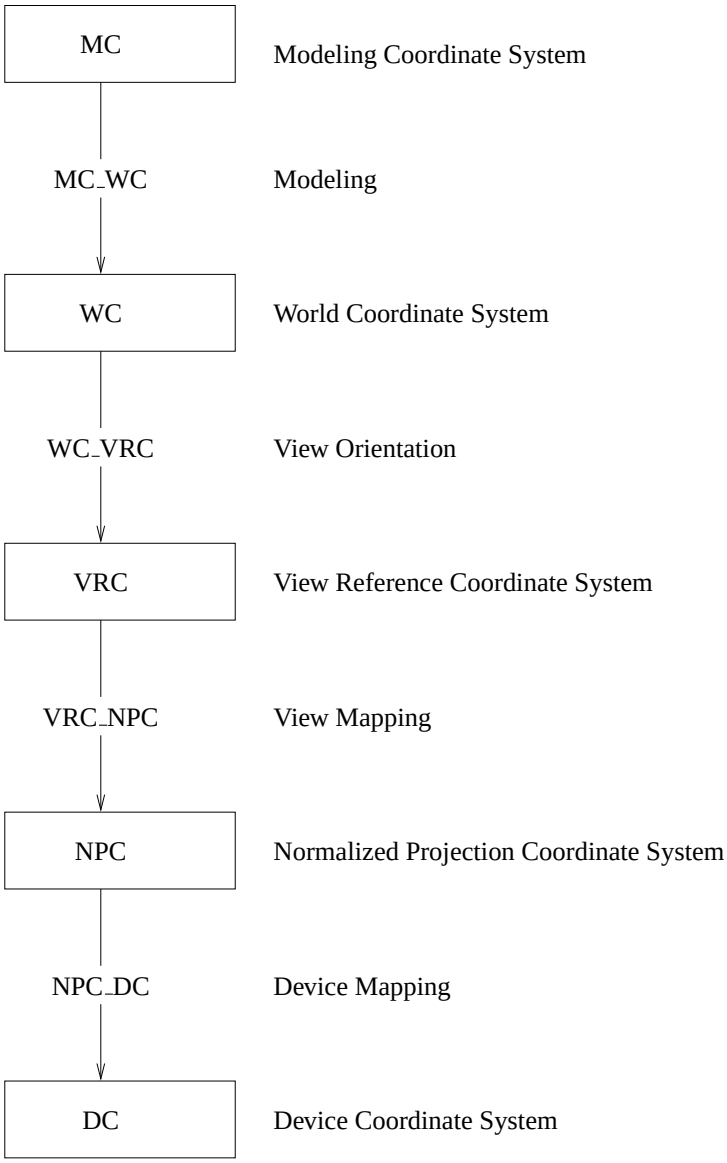


Abbildung 15.10: Transformationspipeline

15.3 Clipping

Jedes Polygon, das die Viewing Pipeline durchläuft, kostet Rechenaufwand zur Transformation seiner Eckpunte und damit Zeit. Noch mehr Zeit kostet das Schattieren des Polygons, das wir später noch kennenlernen werden.

Da nur der Teil der Szene, der sich im *frustum* befindet, später auf dem Bilbschirm zu sehen sein wird, ist es sinnvoll, den unsichtbaren Teil der Szene so früh wie möglich im Ablauf der Viewing Pipeline "loszuwerden". Im average case dürfen die Punkte als gleichverteilt im Raum angenommen werden. Der unsichtbare Teil der Szene wird also im Allgemeinen einen nicht zu vernachlässigenden Anteil an der Gesamtszene haben.

Um den unsichtbaren Teil auszublenden, müssen wir die gesamte Szene auf irgendeiner Stufe der Viewing Pipeline am *frustum* clippen. Wir wollen zwei Stufen, die sich anbieten, näher betrachten:

- Clipping im WC
- Clipping im NPC

Zunächst gehen wir davon aus, daß neben den MC-Koordinaten auch die WC-Koordinaten für jeden Polygonpunkt vorliegen.

15.3.1 Clipping im WC

Diese Strategie beginnt damit, die Ebenengleichungen für die sechs Seiten des *frustums* im WC zu bestimmen. Bei einer Kamerafahrt muß dies im Allgemeinen für jedes Frame neu geschehen. Der Aufwand ist aber unabhängig von der Szene und ihrer Komplexität. Jeder Polygonpunkt wird durch Auswertung der Ebenengleichung an jeder der sechs Seiten geclippt. Wenn ein Polygon eine oder mehrere der Seiten schneidet, müssen noch die neuen Polygonpunkte berechnet werden, indem die Schnittpunkte der Polygonkanten mit den Seiten ermittelt werden.
Nur die verbleibenden Punkte müssen ins DC transformiert und auf dem Bildschirm angezeigt werden. Dazu ist eine Multiplikation mit der Matrix WC_DC notwendig.

15.3.2 Clipping im NPC

Zunächst müssen alle Polygonpunkte mit der Matrix WC_NPC ins NPC transformiert werden. Dann wird die dreidimensionale Erweiterung des 2D-Polygon-Clippings von Cohen und Sutherland durchgeführt. Es wird für jeden Punkt ein 6-Bit-Bereichscode bestimmt, der angibt, ob ein Punkt im View Volume (der Einheitswürfel) liegt oder nicht. Dieser Code läßt sich besonders einfach bestimmen, da die Clippingebenen unabhängig von der Szene und der Lage der Kamera immer die Seiten des Einheitswürfels sind. Es reicht also ein einfacher Vergleich mit den Werten 0 bzw. 1, um ein Bit des Bereichscodes festzulegen. Falls eine Polygonkante eine oder mehrere der Seiten schneidet müssen wieder die neuen Polygonpunkte errechnet werden.
Danach müssen die verbleibenden Punkte mit einer weiteren Multiplikation (mit NPC_DC) in Bildschirmkoordinaten gebracht und angezeigt werden.

15.3.3 Vergleich der beiden Vorgehensweisen

Beim Clipping im WC spart man eine Transformation **aller** Punkte im Gegensatz zum Clipping im NPC. Dafür ist das eigentliche Clipping eines Polygon(punkte)s etwas aufwändiger als im zweiten Fall. Auch die Schnittpunktberechnung ist im NPC etwas günstiger als im WC.
Beim Clipping im WC muß das *frustum* noch ins WC gebracht werden. Dieser Vorgang hat konstanten Aufwand und fällt ab einer gewissen Szenengröße nicht mehr ins Gewicht.
Die Zahl der Punkte, die nach dem Clipping übrig bleibt ist in beiden Fällen gleich, damit auch der Aufwand für die abschließende Transformation.

Man kann nicht entscheiden, welche Strategie die bessere ist, denn die Effizienz der mathematischen Operationen auf dem verwendeten Prozessor bzw. in der verwendeten Programmiersprache spielen eine Rolle.

Der unnötige Aufwand der durch die Transformation der unsichtbaren Punkte ins NPC entsteht, läßt sich folgendermaßen abschätzen:

Angenommen die ganze Szene passe in einen Würfel der Kantenlänge g. Dann hat sie ein Volumen von g^3. Der Betrachter stehe im Schwerpunkt des Würfels und blicke so in die Szene, daß das View Window genau einer Würfelseite entspricht. Dann sieht er fünf Sechstel der Szene nicht. Bei gleichverteilten Punkten entspricht das auch fünf Sechstel unnötig vom WC ins NPC transformierten Punkte.

Jede Transformation verursacht 16 Multiplikationen und 12 Additionen. Die Entscheidung, ob ein Punkt im Frustum liegt, kostet im NPC sechs Vergleiche (die Hälfte der Vergleiche findet gegen 0 statt, was vermutlich nochmal schneller als ein beliebiger Vergleich ist). Dieselbe Entscheidung kostet im WC 18 Multiplikationen und 6 Vergleiche. D.h. das Clipping im WC ist pro Knoten zwei Mutliplikationen teurer. Allerding braucht man im NPC zusätzlichen Speicherplatz für die NPC-Koordinaten und den Bereichscode.

Wenn die WC-Koordinaten allerdings nicht vorliegen, sondern für jedes Frame neu aus den MC-Koordinaten errechnet werden müssen, sieht es etwas anders aus. Dann bleibt der Aufwand des Clipping im NPC gleich, denn der erste Schritt besteht jetzt in einer Transformation mit MC_NPC. Der Aufwand des Clipping im WC steigt aber um eine Transformation MC_WC für jeden Polygonpunkt.

Trotzdem hat die parallele Speicherung der WC-Koordinaten ihre Berechtigung. Sie werden für die Beleuchtung benötigt (hier allerdings nur die der sichtbaren Punkte) und viele Datenstrukturen, die die Bestimmung des sichtbaren Teils der Szene wesentlich effizienter machen, arbeiten auf den WC-Koordinaten.

15.3.4 Umgebungsclipping

Eine weitere Effizienzsteigerung wird möglich durch die Verwendung eines Umgebungsclippings. Hier wird ein Cluster von mehreren, komplexen Objekten mit einem großen Quader umgeben. Ergibt ein erster Clipping-Test, dass dieser Quader außerhalb des Frustums liegt, so erübrigen sich alle Clipping-Abfragen bzgl. seiner inneren Objekte.

Kapitel 16

3D-Repräsentation

Für 3-dimensionale Objekte gibt es mehrere Möglichkeiten der Repräsentation (d.h. Definition des Objekts) und der Darstellung (d.h. Projektion des Objekts auf den Bildschirm auch *rendering* genannt):

Repräsentation

- Elementarobjekt mit Definitionspunkten,
- Drahtmodell,
- Flächenmodell mit Punkt- und Flächenliste und Normalen,
- CSG (constructive solid geometry) mit mengentheoretischer Verknüpfung von Elementarobjekten.

Darstellung

- Punktmodell
- Drahtmodell mit sämtlichen Kanten,
- Drahtmodell mit Entfernung verdeckter Kanten,
- Flächenmodell mit Einfärbung, ohne abgewandte Flächen,
- Flächenmodell mit Einfärbung, ohne verdeckte Teile von Flächen,
- Flächenmodell mit Einfärbung, ohne verdeckte Teile von Flächen, mit Beleuchtungsmodell
- Körpermodell mit Berechnung von Schattenbildung, Spiegelungen und Brechungen.

Zur Berechnung der Darstellung wird die Viewing-Pipeline benötigt, welche eine Transformation der Definitionspunkte vornimmt (Kapitel 15), die nicht sichtbaren Flächen entfernt (Kapitel 17) sowie das Rastern der projezierten Flächen durchführt (Kapitel 18).

In diesem Kapitel werden die verschiedenen Repräsentationsarten behandelt und einige dazu gehörende Beispiele.

16.1 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 (Kugel, Quader, Pyramide, Kegel, Zylinder,...). 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.

16.2 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.

16.3 Flächenmodell

Es 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 und eine Fläche umschließen.

Beim Würfel verbinden die Kanten die Eckpunkte, bei einer Kugel werden die Längen- und Breitenkreise durch *n*-Ecke angenähert, wobei mit *n* die Güte der Approximation steigt.

Punkteliste	Flächenliste
$P_1 : (x_1, y_1, z_1)$	$F_1 : p_1, p_2, p_4$
$P_2 : (x_2, y_2, z_2)$	$F_2 : p_1, p_4, p_3$
$P_3 : (x_3, y_3, z_3)$	$F_3 : p_1, p_3, p_3$
$P_4 : (x_4, y_4, z_4)$	$F_4 : p_4, p_2, p_3$

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.

16.4 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.

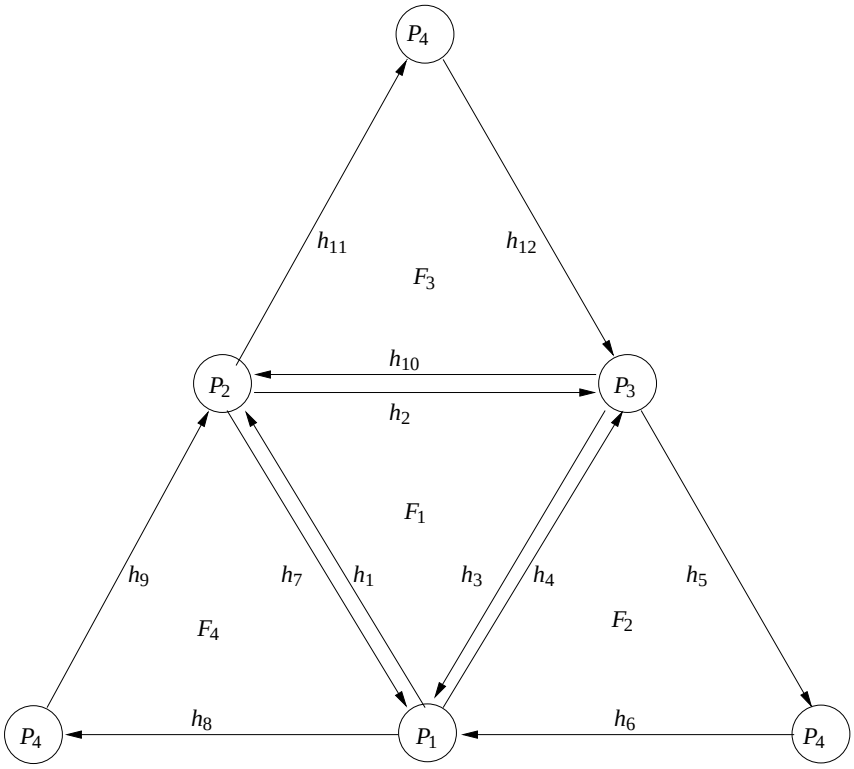


Abbildung 16.1: Tetraeder mit 4 Knoten, 12 Halbkanten, 4 Flächen

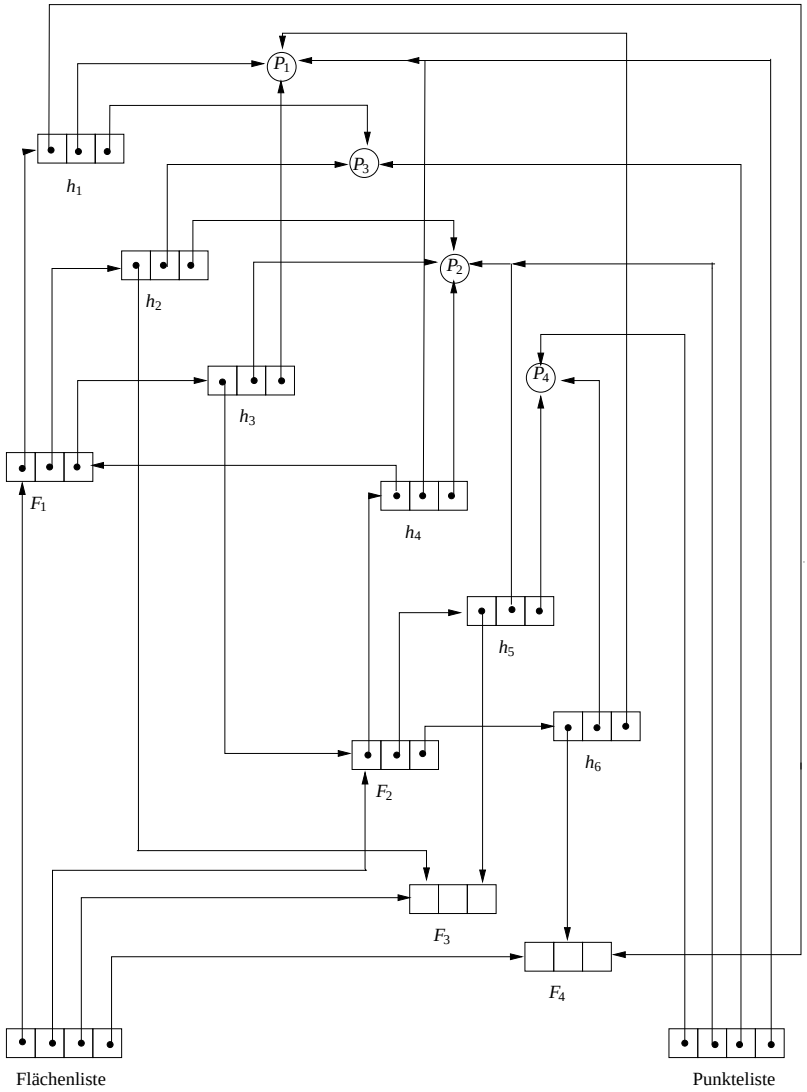


Abbildung 16.2: Objektstruktur für Tetraeder. Nicht gezeichnet sind Halbkanten für F_3 und F_4

16.5 Polyeder

Abbildung 16.3 zeigt eine vom Projektionsalgorithmus erzeugte Szene mit Würfel und Tetraeder in der Drahtmodell-Darstellung mit gestrichelten Rückkanten.

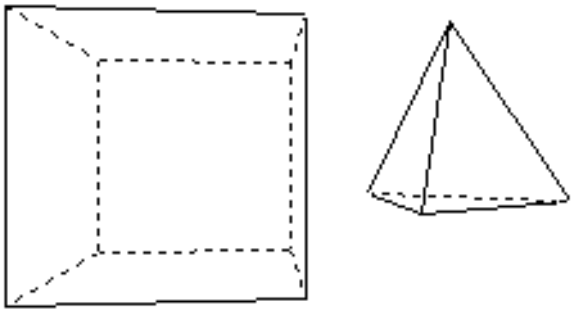


Abbildung 16.3: Vom Projektionsalgorithmus berechnete Drahtgitter-Darstellung

Ein *Würfel* mit der Kantenlänge 1 hat im MC die homogenen Eckpunktkoordinaten

$$
\begin{array}{llll}
(+0.5, & +0.5, & +0.5, & 1), \\
(+0.5, & -0.5, & +0.5, & 1), \\
(-0.5, & +0.5, & +0.5, & 1), \\
(-0.5, & -0.5, & +0.5, & 1),
\end{array}
\qquad
\begin{array}{llll}
(+0.5, & +0.5, & -0.5, & 1), \\
(+0.5, & -0.5, & -0.5, & 1), \\
(-0.5, & +0.5, & -0.5, & 1), \\
(-0.5, & -0.5, & -0.5, & 1).
\end{array}
$$

Da jede der sechs Flächen des Würfels eben ist, gibt es pro Fläche nur einen Normalenvektor, der durch das Kreuzprodukt zweier benachbarter Kanten berechnet werden kann. Die Reihenfolge der Punkte ist dabei signifikant, da jeweils aufeinanderfolgende Punkte eine Kante bilden. Ein *Quader* entsteht aus dem Würfel durch ungleichmäßige Skalierung beim Modeling.

Auch alle anderen Polyeder werden im MC in der Flächendarstellung definiert, ein Tetraeder beispielsweise mit Kantenlänge 1 und Schwerpunkt im Ursprung.

16.6 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.

16.7 Zylinder

Der zur *z*-Achse symmetrische *Zylinder* mit Höhe 2 und Radius 1 besteht aus einer Mantelfläche und zwei Deckflächen. Die beiden Kreisscheiben bei $z = 1$ und $z = -1$ lassen sich durch regelmäßige *n*-Ecke darstellen. Die Mantelfläche wird durch *n* Rechtecke approximiert. Dabei wachsen mit *n* sowohl die Genauigkeit der Approximation als auch der Rechenaufwand. Sei $\alpha = (2\pi)/n$, so lauten für ein solches Rechteck die vier Eckpunkte und die zugehörigen Normalenvektoren:

Eckpunkt	Normalenvektor
$(\cos(\phi), \sin(\phi), +1, 1)$	$(\cos(\phi), \sin(\phi), 0, 0)$
$(\cos(\phi+\alpha), \sin(\phi+\alpha), +1, 1)$	$(\cos(\phi+\alpha), \sin(\phi+\alpha), 0, 0)$
$(\cos(\phi+\alpha), \sin(\phi+\alpha), -1, 1)$	$(\cos(\phi+\alpha), \sin(\phi+\alpha), 0, 0)$
$(\cos(\phi), \sin(\phi), -1, 1)$	$(\cos(\phi), \sin(\phi), 0, 0)$

mit $\phi = k \cdot \alpha$, $k \in \{0, ..., n-1\}$.

Als Normalenvektor in einem Eckpunkt dieser Fläche wird also der tatsächliche Normalenvektor der Mantelfläche genommen. Dadurch erhalten aneinandergrenzende Flächen in ihren gemeinsamen Eckpunkten auch denselben Normalenvektor. Auf diese Weise erzeugt das Programm bei der späteren Beleuchtung den Eindruck eines stetigen Übergangs zwischen den Flächen. Der Betrachter nimmt statt einer *n*-eckigen Säule den Zylinder wahr.

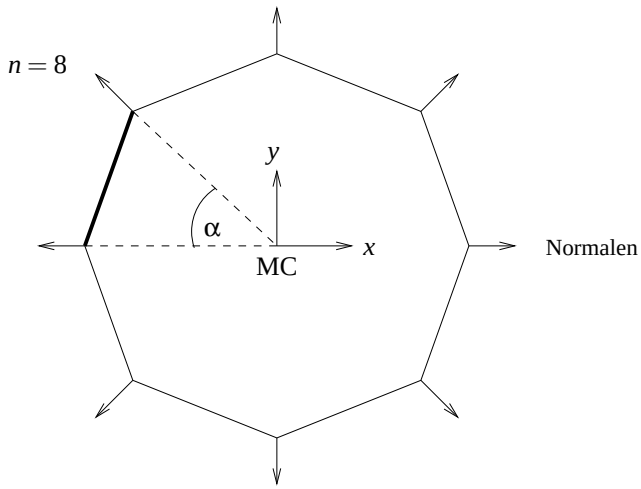


Abbildung 16.4: Zylinder im Grundriß mit Normalenvektoren der Mantelfläche

16.8 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:

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

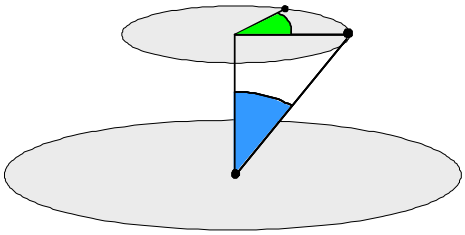


Abbildung 16.5: Konstruktion einer Kugel mit Radius 1 im Ursprung

Dadurch entstehen auf der Kugel $n/2$ gleichgroße Längenkreise und $(n/2-1)$ verschieden große Breitenkreise. Diese schneiden n Dreiecke an jedem Pol und $n(n/2-2)$ Vierecke aus der Kugeloberfläche. Die Ortsvektoren der Eckpunkte eines Dreiecks am Nordpol $(\theta = 0)$ lauten

$$(0,0,+1,1), (\sin(\alpha)\cos(\phi), \sin(\alpha)\sin(\phi), \cos(\alpha), 1), (\sin(\alpha)\cos(\phi+\alpha), \sin(\alpha)\sin(\phi+\alpha), \cos(\alpha), 1),$$

mit $\phi = k \cdot \alpha$, $k \in \{0, \ldots, n-1\}$.

Die Eckpunkte eines der Vierecke haben die Ortsvektoren

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

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.

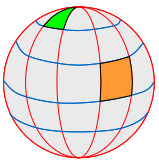


Abbildung 16.6: Breiten- und Längengrade einer Kugel-Approximation

16.9 Bezier-Flächen

Die in Kapitel 7.3 eingeführten Bezier-Kurven können auch zur Definition einer gekrümmten Fläche im Raum verwendet werden. Da eine Fläche zweidimensional ist, muß eine weitere Parameterdimension hinzugefügt werden, d.h. statt $0 \leq t \leq 1$ gilt nun $0 \leq u,v \leq 1$. Im folgenden Beispiel werden 16 Kontrollpunkte mit kubischen Bernsteinpolynomen gewichtet:

$$P(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} \cdot B_{i,3}(u) \cdot B_{j,3}(v) \cdot P_{i,j}$$

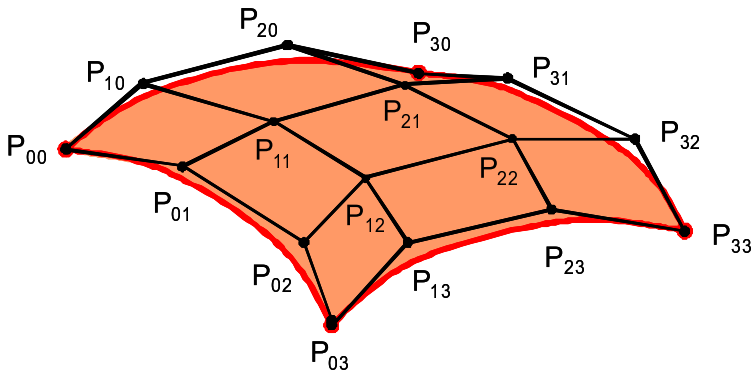


Abbildung 16.7: Gitternetz mit 16 Kontrollpunkten für Bezier-Fläche im Raum

Beim Verlängern einer Bezier-Fläche ist darauf zu achten, dass die neue Reihe von Kontrollpunkten tangential die bisherige Fläche verlängert, d.h. $P_{i,2}, P_{i,3} = Q_{i,0}, Q_{i,1}$ sind collinear und das Verhältnis der Abstände $|P_{i,3} - P_{i,2}| \ / \ |Q_{i,1} - Q_{i,0}|$ ist konstant.

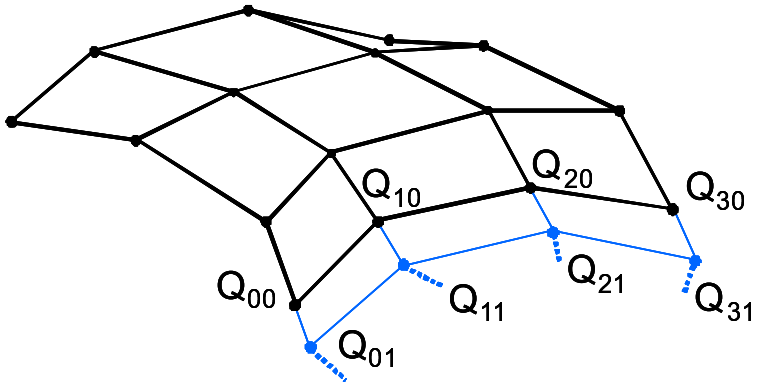


Abbildung 16.8: Verlängerung einer Bezierfläche

16.10 NURBS-Flächen

Eine NURBS-Fläche sieht in homogenen Koordinaten folgendermaßen aus:

$$\vec{q}^h(u,w) = \sum_{i=0}^{n}\sum_{j=0}^{m} P_{i,j}^h N_{i,k}(u) M_{j,l}(w)$$

Die $P_{i,j}^h$ sind die vierdimensionalen homogenen Stützpunkte. Die $N_{i,k}$ und die $M_{j,l}$ sind die nichtrationalen B-Spline-Basisfunktionen aus Kapitel 7. Um affine Koordinaten zu erreichen, wird wieder durch die homogenen Koordinaten dividiert:

$$\vec{q}(u,w) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} h_{i,j} P_{i,j} N_{i,k}(u) M_{j,l}(w)}{\sum_{i=0}^{n}\sum_{j=0}^{m} h_{i,j} N_{i,k}(u) M_{j,l}(w)} = \sum_{i=0}^{n}\sum_{j=0}^{m} P_{i,j} S_{i,k}(u,w)$$

wobei $P_{i,j}$ die dreidimensionalen Stützpunkte sind. Sie ergeben das *Kontrollnetz* für die Oberfläche. Bei den $S_{i,j}(u,w)$ handelt es sich um die biparametrisierten Flächenbasisfunktionen

$$S_{i,j}(u,w) = \frac{h_{i,j} N_{i,k}(u) M_{j,l}(w)}{\sum_{i1=0}^{n}\sum_{j1=0}^{m} h_{i1,j1} N_{i1,k}(u) M_{j1,l}(w)}$$

Die $S_{i,j}(u,w)$ sind nicht das Produkt der $R_{i,k}(u)$ und der $R_{j,l}(w)$ aus Kapitel 7. Sie haben aber ähnliche analytische Eigenschaften, so daß die NURBS-Fläche ähnliche analytische und geometrische Eigenschaften, wie die früher erwähnten NURBS-Kurven:

- Die $S_{i,j}(u,w)$ addieren für ein festes Paar von u und w zu 1.
- $S_{i,j}(u,w) \geq 0 \; \forall u,w$
- Der maximale Grad der Basispolygone ist gleich der Zahl der Kontrollpunkte -1 in der entsprechenden Dimension.
- Eine NURBS-Kurve der Ordnung k,l (Grad $k-1, l-1$) ist überall C^{k-2}, C^{l-2} stetig.
- NURBS-Flächen sind invariant bgzl. perspektivischer Projektion. Es genügt, die Stützpunkte mit der Viewing Pipeline zu transformieren und die eigentliche Interpolation im DC vorzunehmen.
- Wenn gilt $h_{i,j} \geq 0 \; \forall i,j$, liegt die Fläche innerhalb der konvexen Hülle des Kontrollnetzes.
- Der Einfluß eines Stützpunktes ist begrenzt auf $\pm k/2, \pm l/2$ Interpolationsabschnitte in jeder Parameterdimension.
- Um die Normale der Fläche in einem beliebigen Punkt zu berechnen, müssen die Richtungsableitungen bzgl. u und w in diesem Punkt gebildet werden.

16.11 CSG (constructive solid geometry)

Sollen die beschriebenen Objekte auch im physikalischen Sinne realisierbar sein (zum Beispiel über wohldefinierte Volumen verfügen), so müssen die gespeicherten Oberflächen zusätzliche Eigenschaften erfüllen. Die zugrundeliegende Theorie heißt CSG (constructive solid geometry). Um solche Objekte zu erzeugen, beginnt man mit wohldefinierten Objekten und erzeugt durch regularisierte Mengenoperationen $\cup*$ (Vereinigung), $\cap*$ (Durchschnitt), $\setminus*$ (Differenz) neue, wiederum physikalisch sinnvolle Objekte. Hierbei bedeutet für die mengentheoretische Operation op $\in \{\cup, \cap, \setminus\}$ die regularisierte Operation op* den Abschluss des Inneren der Verknüpfung von A mit B:

$$A\ op*\ B\ =\ closure(interior(A\ op\ B))$$

Die hierarchische Struktur wird beschrieben durch einen binären Baum, dessen Blätter beschriftet sind mit den Elementarobjekten und dessen innere Knoten beschriftet sind mit den regularisierten Mengenoperationen $\cup*$ (Vereinigung), $\cap*$ (Durchschnitt), $\setminus*$ (Differenz). Die Wurzel repräsentiert das resultierende Objekt, welches aus den Elementarobjekten an den Blättern unter Anwendung der Operationen an den inneren Knoten konstruiert werden kann.

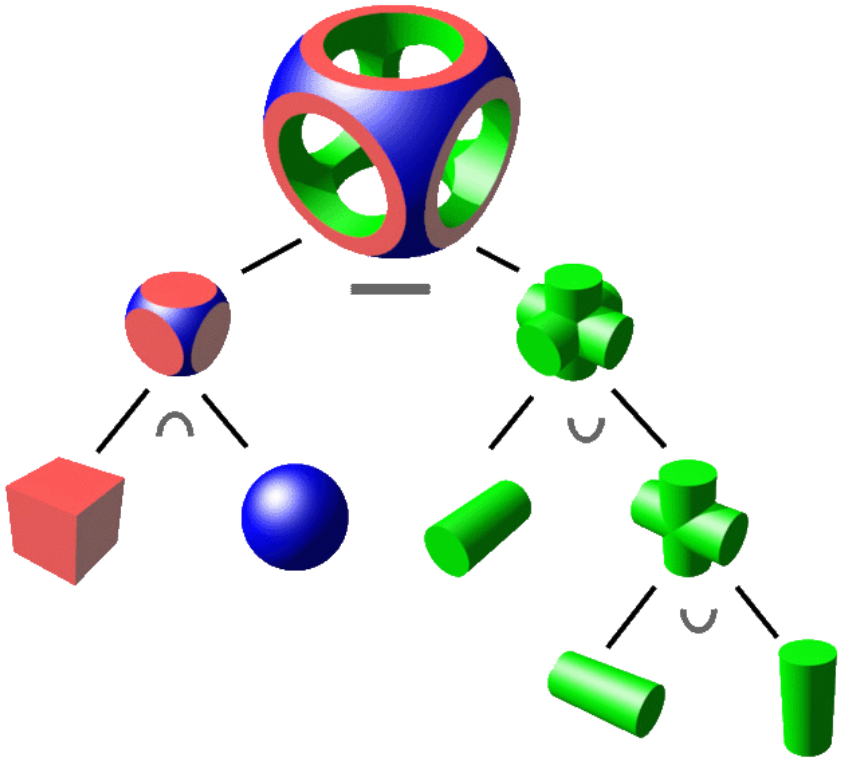


Abbildung 16.9: Darstellung der Verknüpfungshierarchie (Beispiel aus Wikipedia)

16.12 Octree

Zur Verwaltung der räumlichen Anordung von Objekten im dreidimensionalen Raum eignet sich der *Octree*. Es handelt sich um die Erweiterung um die dritte Dimension des 2-dimensionalen *Quadtree*.

Abbildung 16.10 zeigt die Platzierung von 4 schwarzen Rechtecken A,B,C,D in der Ebene. Der zugehörige Quadtree teilt rekursiv die repräsentierte Fläche in 4 Quadranten ein und notiert an den Knoten, ob er nichts enthält (weiss), teilweise Objekte enthält (grau) oder ein Rechteck repräsentiert (schwarz).

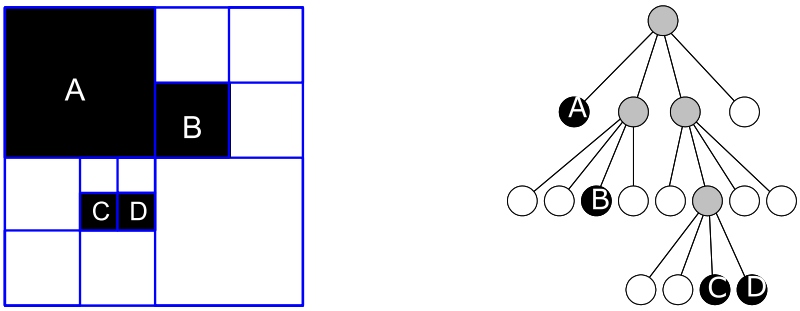


Abbildung 16.10: 4 Rechtecke in der Ebene mit zugehörigem Quadtree

Abbildung 16.11 zeigt die Platzierung von drei Würfeln A,B,C im Raum. Der zugehörige Octree teilt rekursiv den repräsentierten Raum in 8 Oktanten ein und notiert an einem Knoten, ob er nichts enthält (weiss), teilweise Objekte enthält (grau) oder einen Quader repräsentiert (farbig).

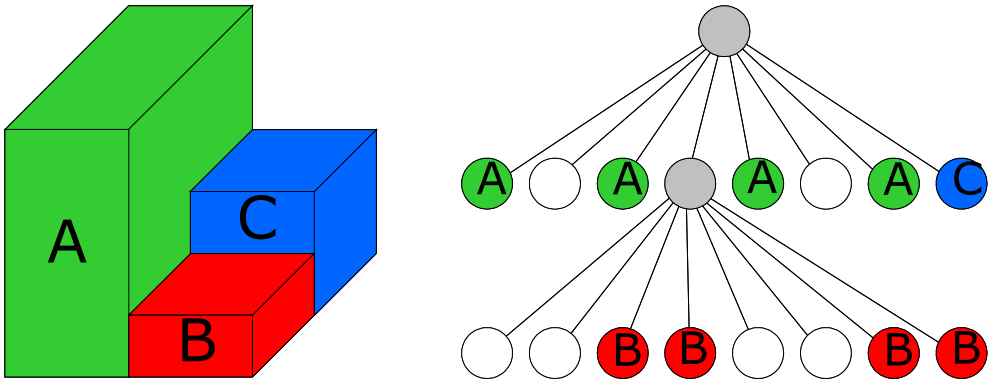


Abbildung 16.11: 3 Quader im Raum mit zugehörigem Octree

16.13 Java-Applet zur Wire-Frame-Projektion

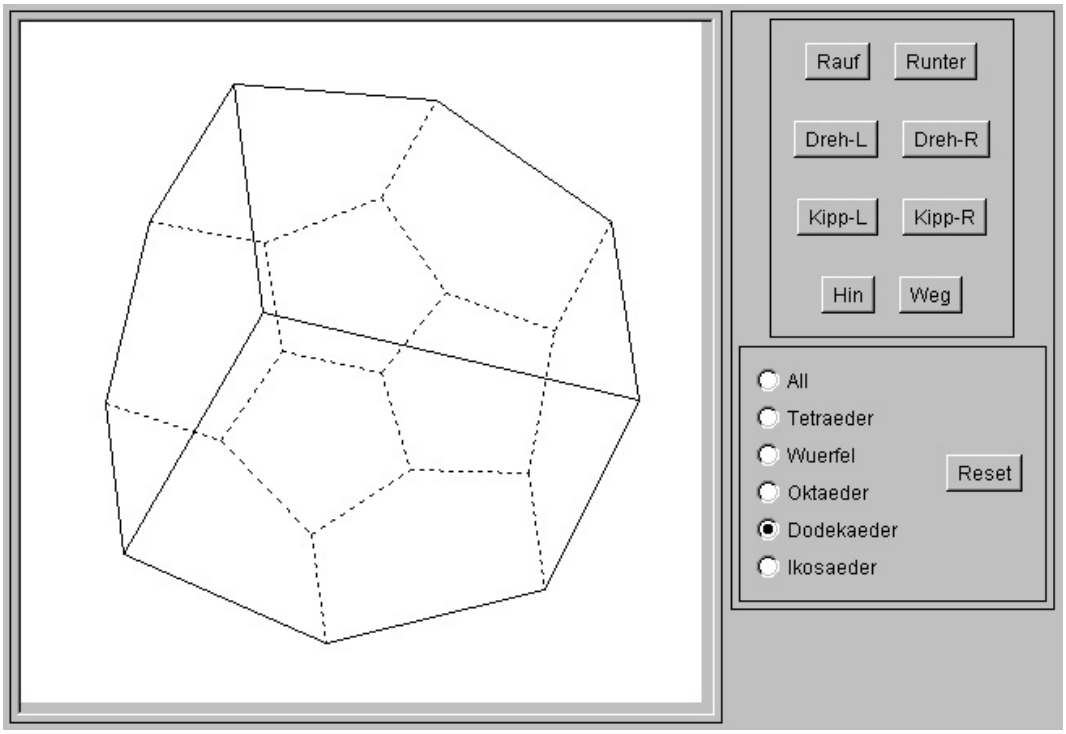


Abbildung 16.12: Screenshot vom 3D-wire-Applet

Kapitel 17

Culling

Unter dem Stichwort Culling werden alle Techniken zusammengefaßt, die zur Entfernung von unsichtbaren Kanten, Flächen und Objekten dienen.

Beim Culling werden unterschieden:

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

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

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

$$\vec{p} \cdot \vec{n} - \vec{a} \cdot \vec{n} = e.$$

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

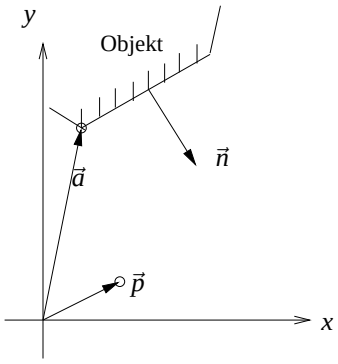


Abbildung 17.1: Back-Face Removal mit Hessescher Normalform

Um die Rückflächen zu erkennen, ist für alle Flächen die obige Ebenengleichung zu berechnen, in die der Betrachterstandpunkt eingesetzt wird. Ergibt sich $e \geq 0$, so handelt es sich um eine Vorderfläche, sonst um eine Rückfläche. Dabei ist zu beachten, daß ggf. für jeden Eckpunkt eine eigene Normale definiert ist. Dies ist zum Approximieren von gekrümmten Flächen notwendig. Bei einer Rückfläche sind definitionsgemäß alle Normalenvektoren abgewandt. Flächen, bei denen einige Normalen zum Betrachter und andere von ihm weg zeigen, werden teilweise dargestellt. Es ist Aufgabe des später vorzustellenden Shaders, die sichtbaren Pixel zu bestimmen. Bei der Liniendarstellung wird eine Kante gezeichnet, wenn mindestens ein Eckpunkt sichtbar ist.

Eine algorithmische Vereinfachung des Tests auf Sichtbarkeit ergibt sich durch die Transformation der Ebenengleichung ins NPC. Hierbei wird ausgenutzt, daß der Betrachterstandpunkt im NPC per Definition die homogenen Koordinaten $(0,0,1,0)$ besitzt. (Er liegt im Unendlichen auf der positiven z-Achse.) Daraus folgt, daß das Skalarprodukt zwischen dem Normalenvektor der Ebene und dem Vektor des Betrachterstandpunkts 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. Dabei können Objekte sich z.T. selbst verdecken oder zwischen Betrachter und anderen Objekten liegen.

Das Entfernen der nicht sichtbaren Flächen wird als *Hidden-Surface Removal (HSR)* bezeichnet.

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. Auch transparente Polygone sind realisierbar, allerdings nur mit großem Zusatzaufwand. Für den Fall, daß das transparente Polygon dem Betrachter am nächsten liegt, müssen Polygone, die *nach* dem transparenten Polygon gerendert werden und *hinter* diesem liegen, durch aufwändige Verknüpfungen eingeblendet werden.

Da der z-Buffer-Algorithmus am Ende der Transformationspipeline im DC arbeitet, wird er als Bildraumalgorithmus klassifiziert.

Der z-Buffer-Algorithmus wird vom Programm in der inneren Schleife des Scanline-Algorithmus aufgerufen und läßt sich wie folgt skizzieren:

```
Für jede Fläche F tue:
      Für jedes Pixel (x, y) auf dieser Fläche tue:
      berechne Farbe c und Tiefe z
      falls z > tiefe[x, y]:
      dann wird c an der Stelle x,y im Frame Buffer eingetragen
      und tiefe[x, y] auf z gesetzt.
```

Ist die Fläche durch $Ax+By+Cz+D=0$ gegeben, so ist die Tiefe im Punkt (x,y):

$$z = -\frac{Ax+By+D}{C}$$

Für auf einer Scanline benachbarte Punkte (x_i,y_j) und (x_{i+1},y_j) ergibt sich

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

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

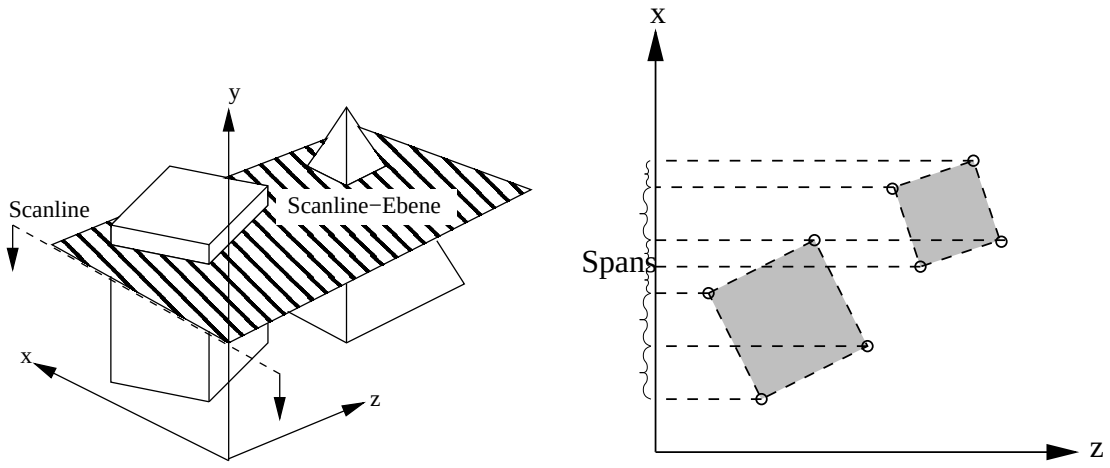
$$z_{j+1} = z_j - \frac{B}{C}$$

Eines der größten Probleme bei der Implementierung des z-Buffers ist sein Speicherplatzbedarf. Eine ausreichende Auflösung der Tiefeninformation ergibt sich erst mit einem 32-Bit z-Wert, da die z-Werte durch die Projektion in den Einheitswürfel nahe der Backplane (also bei $z = 0$) sehr dicht beieinander liegen. Für die RGB-Tripel und den Alpha-Kanal werden vier Byte benötigt, die in einem 32-Bit Integer kodiert werden. Pro Pixel belegen der z-Buffer und der Bildschirmspeicher also acht Byte. Bei einer Bildschirmauflösung von 1024×768 Pixeln ergeben sich folglich 6 MB.

17.2.2 Span-Buffer

Der z-Buffer-Algorithmus muß für jedes Pixel einer Scanline die z-Koordinate berechnen, um zu entscheiden, ob dieses Pixel gesetzt wird oder nicht. Selbst wenn es gesetzt wurde, kann es sein, daß das Pixel später von einem Polygon, das noch näher am Betrachter liegt, überschrieben wird.

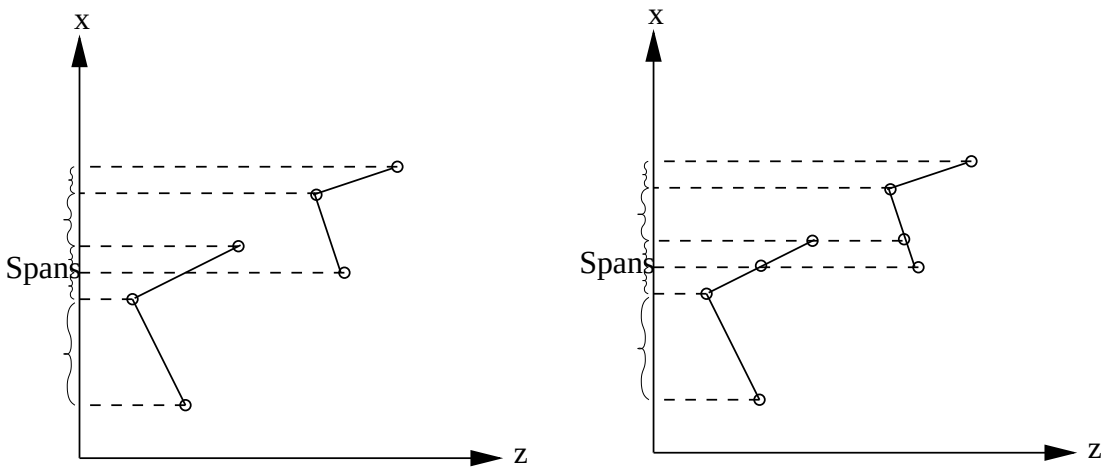
Der Span-Buffer-Algorithmus löst diese beiden Probleme. Normalerweise wird ein Polygon für mehrere aufeinanderfolgende Pixel einer Scanline vorherrschend sein. Ein solcher Scanline-Abschnitt, wird *Span* genannt. Der Algorithmus ermittelt für jede Scanline, welches Polygon in welchem Span vorherrschend ist. Dadurch können die z-Werte der Punkte innerhalb des Spans interpoliert werden und müssen nicht einzeln berechnet werden. Außerdem wird jeder Span genau einmal gezeichnet, es muß also kein Pixel mehrfach eingefärbt werden. Dieser Effizienzsteigerung steht zusätzlicher Aufwand zur Ermittlung der Spans entgegen. Der Abtastzeilenalgorithmus schiebt eine Ebene, die parallel



a) Scanline-Ebene schiebt sich durch die Szene b) Schnittfiguren innerhalb einer Scanline-Ebene

Abbildung 17.2: Pro Scanline wird der Span-Buffer einmal aufgebaut.

zur xz-Ebene ist, die y-Achse herunter (s. Abb. 17.2a). Diese Ebene schneidet ggf. einige Objekte. Die Schnitte mit den Flächen der Objekte sind Punkte oder Strecken (Span). Damit ist das Problem auf zwei Dimensionen (x und z) reduziert (s. Abb. 17.2b).



a) Nur die Spans der Vorderseiten werden betrachtet.

b) Teilung der Spans bzgl. der Endpunkte von anderen Spans.

Abbildung 17.3: Die Schnittfiguren bestimmen die Spans.

Dann werden die Spans der Vorderflächen nach ihren *x*-Werten sortiert (s. Abb. 17.3a) und anschließend miteinander verglichen. Jeder Span wird an allen - innerhalb seines Intervalls liegenden - *x*-Werten der anderen Spans in zwei neue Spans zerschnitten (s. Abb. 17.3b).

Hierbei wird klar, daß einander durchdringende Polygone (und damit sich schneidende Spans) *nicht* an ihrem Schnittpunkt in je zwei neue Spans zerlegt werden. Der Algorithmus versagt in diesem Fall und erzeugt keine korrekte Darstellung (s. Abb. 17.4b). Jetzt ist eine Liste von Gruppen mit deckungsgleichen Spans enstanden. Für jede Gruppe muß nur entschieden werden, welcher Span dem Betrachter am nächsten liegt. Dazu werden für die Anfangspunkte der Spans die *z*-Werte berechnet. Die *z*-Werte der Endpunkte müssen nicht berechnet werden, da ein Span, der am Anfangspunkt vorherrschend ist, auch am Endpunkt vorherrscht (weil keine Spans existieren, die sich schneiden) (s. Abb. 17.4a).
Die entstandene Liste von Spans wird anschließend noch danach untersucht, ob benachbarte Spans ursprünglich vom selben Polygon stammen. Wenn ja, werden sie wieder zu einem Span zusammengefaßt und erst dann auf dem Bildschirm dargestellt.

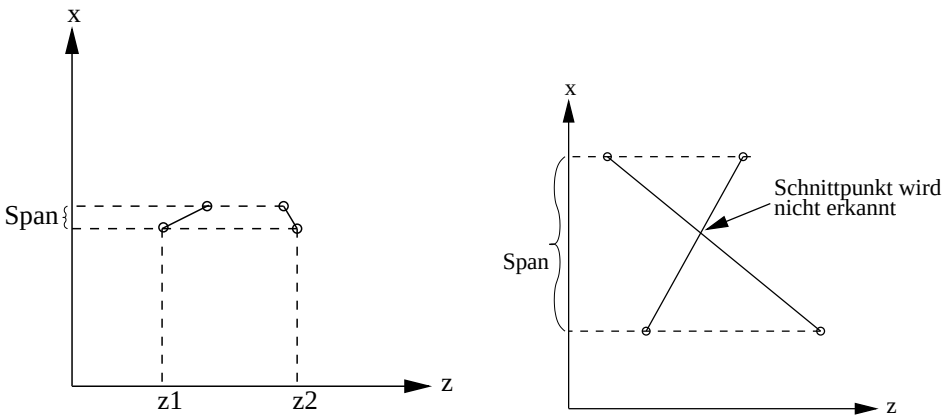
Der Aufwand zur Ermittlung der Spans ist allerdings so hoch, daß der Vorteil gegenüber dem z-Buffer mit steigender Polygonanzahl schwindet.

Falls die Polygone in korrekter *z*-Sortierung vorliegen, kann man mit Hilfe von einem SpanBuffer-Baum oder einem C-Buffer eine weitere Effizienzsteigerung erreichen.

17.2.3 Binary Space Partitioning

Unter Binary Space Partitioning (BSP) versteht man einen Objektraum-Algorithmus, der die Lage der Objekte untereinander berücksichtigt.

Zunächst wird im Vorhinein ein *BSP-Tree* konstruiert, in dem die räumliche Anordnung der Objekte repräsentiert ist. Dann wird für jedes Frame der Betrachterstandpunkt mit dieser Datenstruktur vergli-



a) Zwei Polygone im selben Span. Das linke Polygon ist vorherrschend.

b) Probleme bei sich durchdringenden Seiten.

Abbildung 17.4: Ermittlung des vorherrschenden Polygons.

chen, um die Sichtbarkeit der einzelnen Flächen zu ermitteln.

Die binären Knoten des BSP-Trees enthalten jeweils eine $n-1$-dimensionale Hyper-Ebene, die den n dimensionalen konvexen Gesamtraum (C_i in Abb. 17.5b) in zwei konvexe Halbräume (C_{i+1} und C_{i+2}) unterteilt. Zusätzlich enthalten sie zwei Verweise auf Unterbäume, die jeweils einen der beiden Halbräume repräsentieren. Diese rekursive Unterteilung führt im dreidimensionalen Fall dazu, daß der gesamte $\mathbb{R}^3$ durch die Knotenebenen in viele (kleine) konvexe Teilräume zerlegt wird (s. Abb 17.6).

Die Ebenen im Baum sind identisch mit den Flächen der Objekte, d.h. der aktuelle Teilraum des Vaterknotens wird bzgl. dieser Polygonfläche in einen Bereich zerlegt, der ”vor” der Fläche liegt und in einen Bereich, der ”hinter” der Fläche liegt. Alle Punkte im Raum vor der Fläche sind von ihr aus sichtbar (bzw. die Fläche ist von jedem Punkt dieses Raumes aus sichtbar) und alle Punkte im Raum hinter der Fläche sind von der Fläche aus unsichtbar (bzw. die Fläche ist von jedem Punkt des Raumes aus unsichtbar).

Alle Flächen aller Körper der Szene werden in den BSP-Tree eingefügt (s. Abb 17.6). Dabei kommt es vor, daß eine neu einzufügende Fläche eine (oder mehrere) der Knotenebenen schneidet. Die Fläche wird dann an der Knotenebene in zwei Teilflächen zerschnitten, die jeweils komplett in einem der beiden Teilräume liegen (s. Abb. 17.7). Da im average case $O(n \cdot \log n)$ und im worst case $O(n^2)$ Polygone durch Splits enstehen, ist die Konstruktion des BSP-Trees zu aufwändig, um in Echtzeit zu geschehen. Sie wird offline durchgeführt und daher eignet sich der BSP-Tree nur für statische Szenen.

Jedes Blatt des Baums repräsentiert eine (Teil-)Fläche der Szene.

Pro Frame wird der BSP-Tree bzgl. des aktuellen Betrachterstandpunkts traversiert, um die Tiefensortierung zu erreichen. In jedem inneren Knoten gilt:

> Die Flächen im Teilraum, in dem sich auch der Betrachter befindet, sind näher am Betrachter als die Flächen, die sich jenseits der Ebene (also im anderen Teilraum) befinden.

Ziel ist es, eine lineare Liste der Flächen aufzubauen, in der sich die am weitesten vom Betrachter

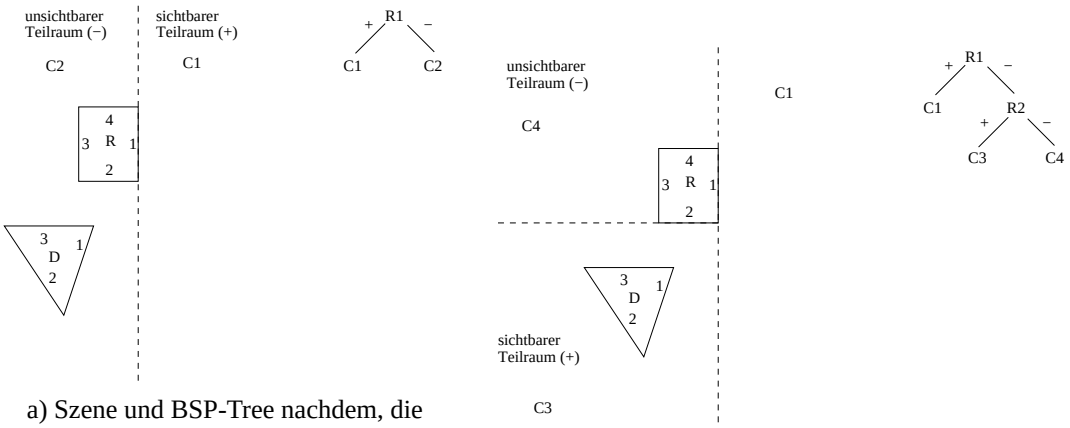


Abbildung 17.5: Einfache Szene aus einem Rechteck (R) und einem Dreieck (D). Erste Schritte beim Aufbau eines 2D-BSP-Trees. Die Knotenhyperebenen sind im 2D-Fall Geraden.

entfernte Fläche am Listenanfang befindet und die am wenigsten entfernte am Listenende (back to front order). Deswegen wird in jedem Knoten zunächst in den Teilraum abgestiegen, in dem sich der Betrachter *nicht* befindet. Erst, wenn alle Flächen aus diesem Raum in die Liste eingefügt wurden, wird der Teilraum, in dem sich der Betrachter befindet, rekursiv untersucht.

Im einfachsten Fall werden anschließend die Polygone gemäß der Liste auf den Bildschirm gezeichnet. Dabei muß beim Setzen eines Pixels kein Test bzgl. der z-Werte mehr durchgeführt werden. Allerdings werden weiterhin viele Pixel mehrfach gesetzt, da die Polygone sich zum Teil überdecken.

Dieser Nachteil kann beseitigt werden, wenn die Liste von hinten abgearbeitet wird. Dann wird das vorderste Polygon zuerst gezeichnet. Beim Zeichnen der weiteren Polygone muß darauf geachtet werden, daß deren verdeckte Teile nicht gezeichnet werden. Dies kann z.B. erreicht werden, indem man beim Zeichnen einen 2D-BSP-Tree aufbaut, indem man alle Polygone anhand der bereits gezeichneten in komplett sichtbare und komplett unsichtbare Teile zerlegt.

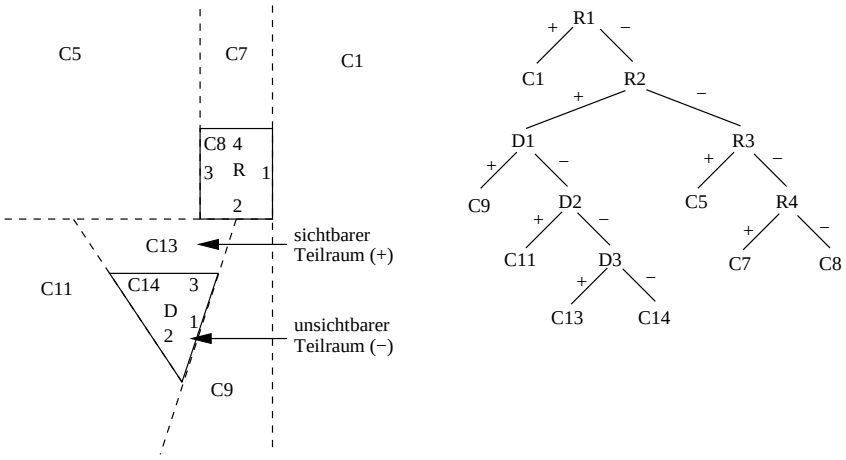


Abbildung 17.6: Gesamte Szene als BSP-Tree; zuletzt wurde Seite 3 des Dreiecks in den Baum aufgenommen. Dabei sind C13 und C14 aus C12 entstanden.

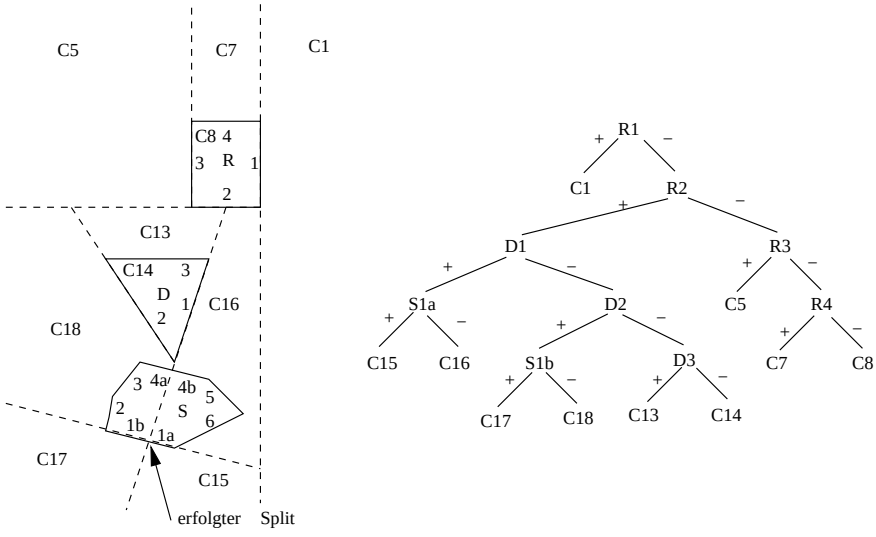


Abbildung 17.7: Neue Szene: Das Sechseck S kommt hinzu. Kante 1 und Kante 4 des Sechsecks wurde beim Aufnehmen geteilt.

Kapitel 18

Beleuchtung

Die reellwertigen Koordinaten der Objekte im DC am Ende der Transformationspipeline müssen in ganzzahlige Pixelkoordinaten gerundet werden. Die Rasterung von Flächen kann mit Hilfe des Scanline-Verfahrens auf die von Linien zurückgeführt werden. In einem zweistufigen Algorithmus wird die Fläche zunächst in eine Menge horizontaler Linien, die sogenannten *Spans*, zerlegt. Die Eckpunkte dieser Spans ergeben sich durch Rasterung der Kanten nach dem vorgestellten Verfahren. In der zweiten Stufe wird dann jeder der Spans gerastert.

Das *Scanline*-Verfahren wird zur Vereinfachung und Beschleunigung des Rendering-Programms nur auf Dreiecke angewandt, denn Dreiecke sind die einfachsten Polygone. Gegenüber allgemeinen Polygonen bieten sie den Vorteil, daß sie planar und konvex sind. Für das Scanline-Verfahren eignen sie sich ausgezeichnet, da für jede Bildschirmzeile (Scanline) maximal zwei Schnittpunkte mit den Dreieckskanten auftreten.

Ein konvexes Polygon läßt sich gemäß der Abbildung triangulisieren, indem von einem beliebigen Eckpunkt aus zu allen nicht benachbarten Eckpunkten Diagonalen gezogen werden.

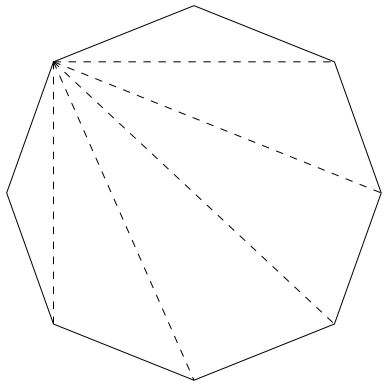


Abbildung 18.1: Triangulisierung eines konvexen Polygons

Unter Rasterung wird hier nicht nur das Füllen der Fläche in der Objektfarbe verstanden. Beim Einsatz des z-Buffers müssen neben den Pixelkoordinaten auch die z-Werte der Polygonpunkte interpoliert werden, um mit Hilfe dieser Tiefeninformation verdeckte Flächen zu unterdrücken.

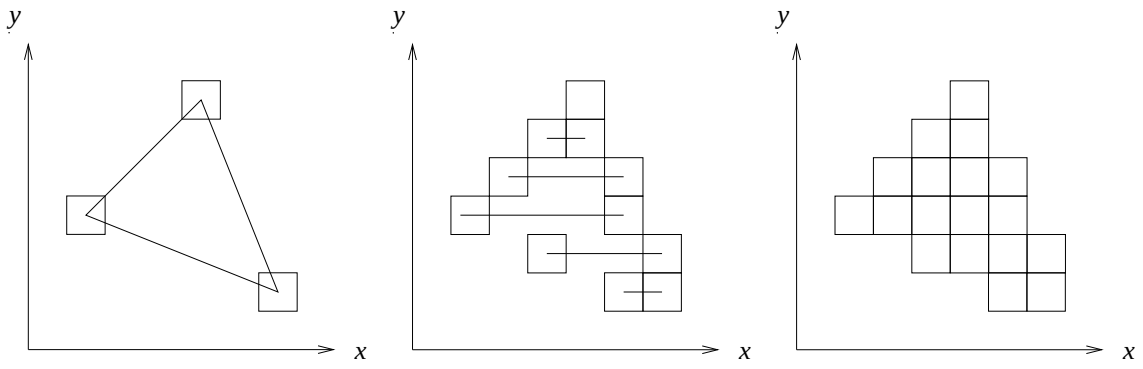


Abbildung 18.2: Scanline-Verfahren zur Rasterung von Dreiecken

Dabei ist zu beachten, daß die Anfangs- und Endpunkte der Spans sich mit einem von Swanson und Thayer 1986 entwickelten Algorithmus berechnen lassen, der dem Bresenham-Algorithmus für Linien ähnlich ist.

```
public void drawSpanStartPoints(int xa, int xe, int ya, int ye) {
  int dx, dy, x, y, error, mi, mf, schritt;

  dx = xe-xa;
  dy = ye-ya;

  error = -dy;
  mi = dx / dy;
  mf = 2*(dx%dy);
  schritt = -2*dy;

  for(y=ya; y<ye; y++) {
    setPixel(x, y);
    x += mi;
    error += mf;
    if(error > 0) {
      x++;
      error += schritt;
    }
  }
}
```

Außerdem sind je nach Schattierungs-Verfahren auch die Farbwerte bzw. die Normalenvektoren, die in den Eckpunkten gegeben sind, über die Fläche zu interpolieren. Verschiedene Farbwerte auf den Objektflächen ergeben sich durch die individuelle Beleuchtung der Punkte. Die Interpolation der Normalen simuliert dabei einen gekrümmten Flächenverlauf. Grundlegend für die Schattierungs-Verfahren ist die Beleuchtung einer Szene mit Hilfe von unterschiedlichen Lichtmodellen.

18.1 Bestandteile der 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

In den meisten Grafiksystemen werden vier Modelle von Lichtquellen definiert:

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

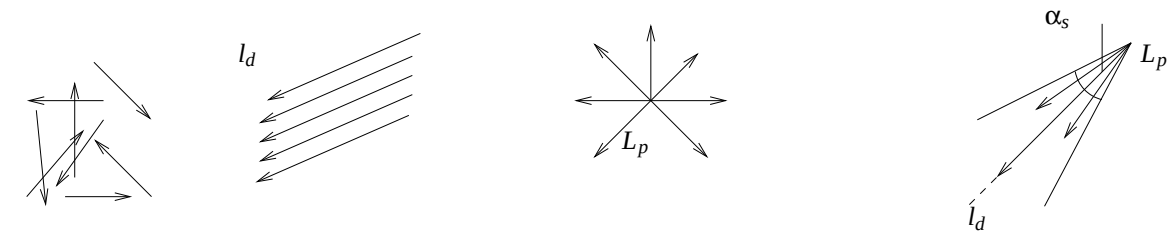


Abbildung 18.3: Vier Modelle von Lichtquellen (schematisch)

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

Gerichtetes Licht ist definiert durch eine Intensität I_g und eine Lichtrichtung $\mathbf{L}_g$. Es ist gekennzeichnet durch Lichtstrahlen, die parallel aus dem Unendlichen in eine bestimmte Richtung strahlen, vergleichbar mit dem Sonnenlicht.

Ein *Punktlicht* mit der Intensität I_0 hat eine Position $\mathbf{L}_p$ im Raum (WC), von der aus es in alle Richtungen gleichförmig abstrahlt wie eine Glühbirne. Die Intensität des Lichts nimmt mit zunehmender Entfernung r ab nach der Formel

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

Dabei bezeichnet r den Abstand von der Punktlichtquelle. C_1 und C_2 heißen Abschwächungskoeffizienten (*attenuation coefficients*, $C_1 \geq 1$, $C_2 \geq 0$).

Das aufwendigste Lichtmodell ist der *Strahler* (*Spot*). Dieser hat wie das Punktlicht eine Intensität I_0, eine Position $\mathbf{L}_p$ sowie die Abschwächungskoeffizienten C_1 und C_2. Dazu kommt eine bevorzugte Lichtrichtung $\mathbf{l}_d$, um die herum das Licht nur im Winkel α_s (*spread angle*) abgestrahlt wird. Außerhalb dieses Lichtkegels ist die Intensität Null. Der *concentration exponent* L_e definiert, wie stark die ausgesandte Lichtenergie mit zunehmendem Winkel zwischen $\mathbf{l}_d$ und der Strahlrichtung $\mathbf{r}$ abnimmt. Die Abnahme ist proportional zu

$$\cos(\mathbf{r}, \mathbf{l}_d)^{L_e}$$

innerhalb des durch α_s definierten Lichtkegels. Je größer L_e, um so stärker ist die Intensität im Zentrum des Lichtkegels gebündelt.

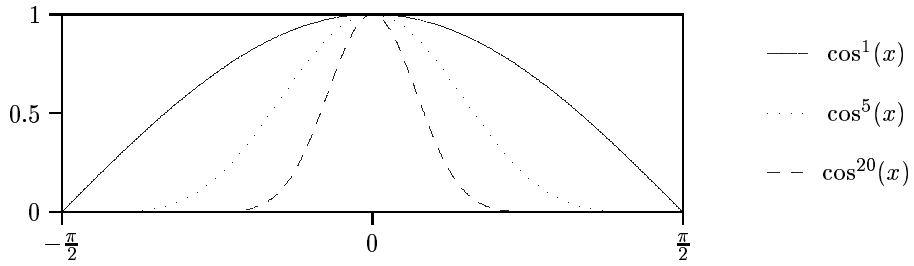


Abbildung 18.4: Lichtintensitätsverteilung beim Strahler (schematisch)

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:

$$\overline{C} = \overline{C_a} + \sum_{i=1}^{n} (\overline{C_{d_i}} + \overline{C_{s_i}}) \ .$$

Dabei ist $\overline{C}$ das gesamte von einem Punkt der Oberfläche reflektierte farbige Licht, n ist die Anzahl der Lichtquellen, $\overline{C_a}$ ist die an diesem Objekt beobachtete ambiente Hintergrundbeleuchtung und $\overline{C_{d_i}}$, $\overline{C_{s_i}}$ sind die 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.

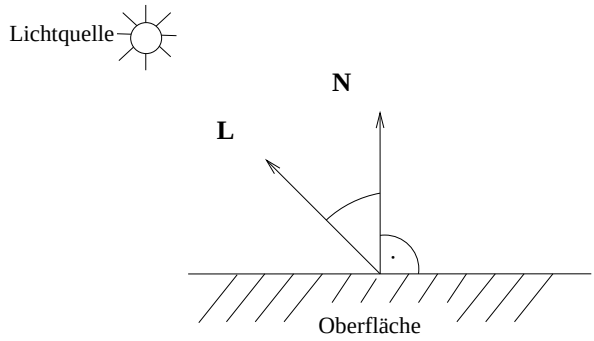


Abbildung 18.5: Cosinusgesetz von Lambert (für diffuse Reflexion)

Die *spekulare Reflexion* beruht auf dem Phänomen der Spiegelung. Eintreffendes Licht wird an der Oberfläche des Objekts gespiegelt (Einfallswinkel = Ausfallswinkel). Das reflektierte Licht hat die spekulare Objektfarbe $\overline{O_s}$, da keine tiefere Wechselwirkung mit dem Objekt stattfindet. Wird das Objekt aus der Reflexionsrichtung betrachtet, so hat es an dieser Stelle einen Glanzpunkt (*Highlight*) in der Farbe $\overline{O_s}$.

Natürliche Materialien sind zumeist keine perfekten Spiegel und strahlen deshalb nicht nur genau in die Reflexionsrichtung, sondern auch mit abnehmender Intensität in andere Richtungen. Von Phong stammt das Modell, daß die abgestrahlte Lichtmenge exponentiell mit dem Cosinus zwischen Reflexionsrichtung **R** und Betrachterrichtung **A** abnimmt. Der spekulare Exponent O_e bestimmt dabei den Öffnungswinkel des Streukegels um **R**, unter dem viel Licht reflektiert wird. Für $O_e \leq 5$ ergeben sich große, für $O_e \gg 10$ kleine Streukegel.

Für einen spekularen Reflektor gilt:

$$\overline{C_s} = k_s \cdot I_e \cdot \overline{O_s} \cdot \cos(\mathbf{R}, \mathbf{A})^{O_e} \ .$$

Dabei ist I_e die Intensität des einfallenden nicht-ambienten Lichts. k_s gibt an, wieviel von I_e gespiegelt wird.

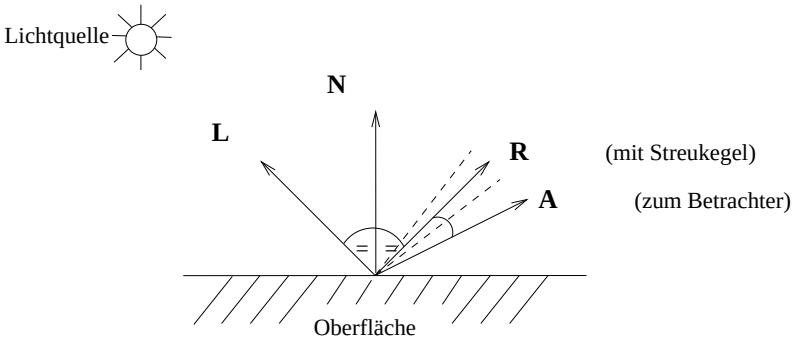


Abbildung 18.6: Spekulare Reflexion nach Phong

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 (wegen der erforderlichen Entfernungen) mit dem vorgestellten Phong-Modell beleuchtet. Nach der Abbildung in die Pixelkoordinaten (DC) werden mit dem Rasteralgorithmus alle Pixel der Dreiecksfläche in der Farbe des Mittelwertes der drei Eckfarbwerte gesetzt:

$$\overline{C_i} = \frac{\overline{C_A} + \overline{C_B} + \overline{C_C}}{3} \; \forall P_i \in \Delta(P_A, P_B, P_C) \; .$$

Dabei bezeichnet $\overline{C_i}$ den RGB-Farbwert für Pixel P_i (siehe Abbildung 18.7).

Beim Flat-Shading wird die Oberfläche des Objekts grob schattiert, die Helligkeitsübergänge sind nicht fließend. Die Qualität der erzeugten Bilder hängt vor allem von der Größe der Dreiecke ab. Das Hauptproblem dieses Verfahrens hängt mit der Eigenschaft des menschlichen Auges zusammen, sprunghafte Intensitätsunterschiede verstärkt wahrzunehmen. Der Randbereich einer helleren Fläche erscheint heller und der Randbereich einer angrenzenden dunkleren Fläche dunkler als der Rest des jeweiligen Polygons. Dieser Effekt wird *Mach-Band-Effekt* genannt.

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:

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

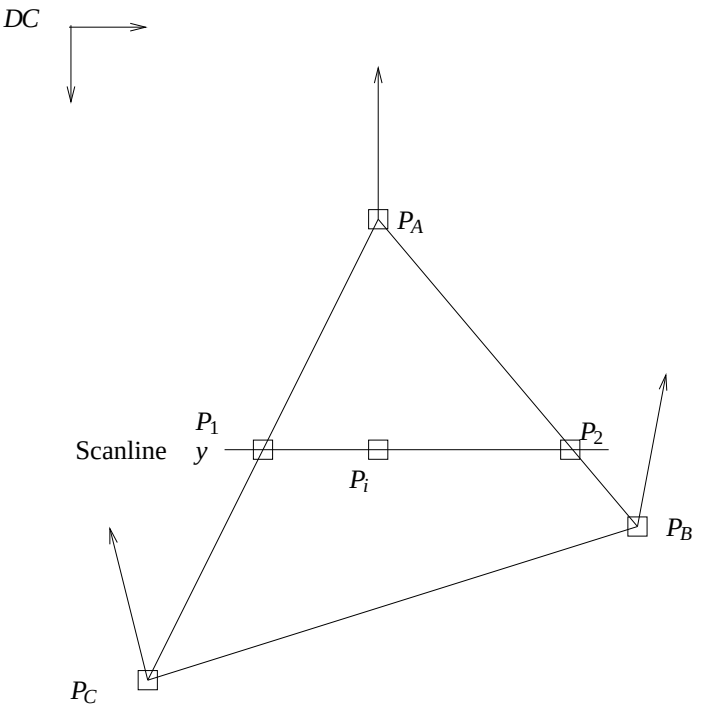


Abbildung 18.7: Dreieck-Shading

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

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

Die Interpolation der Farbwerte ist Teil des Rasteralgorithmus.

Das Gouraud-Shading beseitigt den Mach-Band-Effekt nur zum Teil, da das Auge sogar auf Unstetigkeiten in der zweiten Ableitung der Helligkeitskurve in der beschriebenen Art reagiert. Ein weiterer Nachteil liegt in den verformt dargestellten Spiegelungsflächen, die auf die Polygonaufteilung zurückzuführen sind. Dadurch erscheinen die Highlights der spekularen Reflexion auf großen ebenen Flächen evtl. gar nicht oder auf gekrümmten Flächen verzerrt. Eine Möglichkeit, diese unrealistischen Effekte zu begrenzen, besteht in der Verkleinerung der approximierenden Polygone, was einen höheren Rechenaufwand zur Folge hat.

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 (*per pixel shading*). Die dazu benötigten Normalenvektoren müssen zunächst aus den Normalenvektoren in den Eckpunkten berechnet werden. Für die Fläche in der Abbildung ergeben sich die Normalenvektoren in P_1 und P_2 durch lineare Interpolation der in P_A und P_C bzw. P_A und P_B und daraus wiederum die Normalenvektoren entlang der Scanline. Dabei ist zu beachten, daß für die Beleuchtung die Koordinaten und Normalen

im WC herangezogen werden.

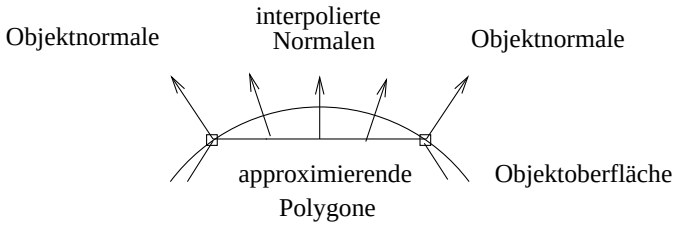


Abbildung 18.8: Interpolation der Normalen beim Phong-Shading

Durch die Interpolation der Normalen ist das Phong-Shading in der Lage, den ursprünglich gekrümmten Verlauf der Oberfläche wiederherzustellen, obwohl das Objekt durch planare Polygone approximiert wird. Dadurch ergibt sich eine fast natürliche spekulare Reflexion mit scharfen Highlights.

Auch mit weniger approximierenden Polygonen ergeben sich bessere Bilder als beim Gouraud-Shading; der Mach-Band-Effekt wird weitgehend unterdrückt. Diese hohe Qualität hat ihren Preis: Statt der Beleuchtung von drei Eckpunkten beim Flat- und Gouraud-Shading müssen beim Phong-Shading alle Pixel des Dreiecks beleuchtet werden,

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.
Schatten können auch mit Hilfe von Maps berechnet werden (s. folgendes Kapitel).

Kapitel 19

Texturing

Unter dem Begriff *Texturing* werden neben dem *Texture Mapping* auch alle anderen Verfahren zusammengefasst, bei denen das Aussehen einer Fläche an jedem Punkt mit Hilfe einer BitMap, einer Funktion oder sonstigen Daten verändert wird.

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

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

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

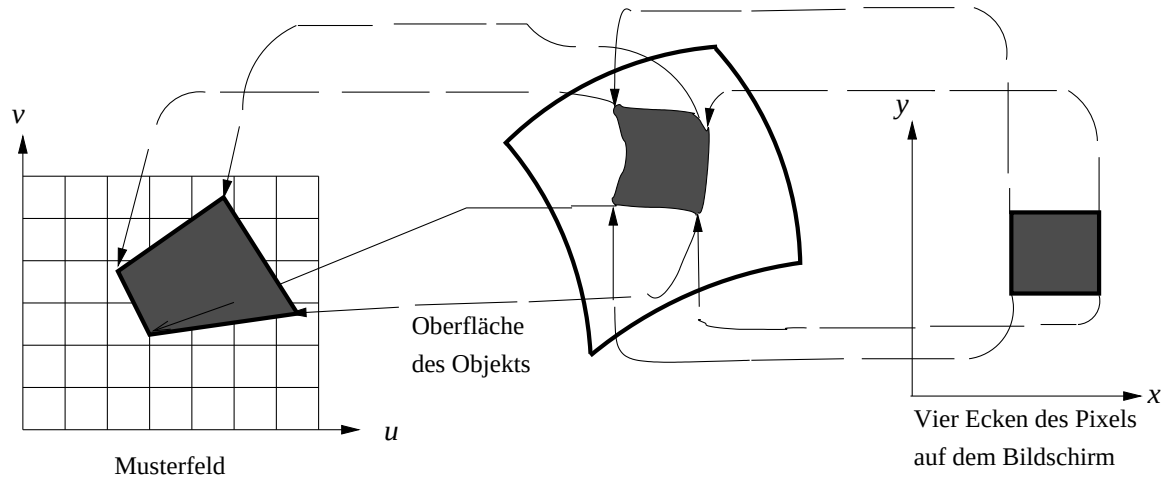


Abbildung 19.1: Transformationskette beim Texture Mapping

Wie Abbildung 19.1 zeigt, müsste eigentlich zum Einfärben eines Pixels im Bildraum die gewichtete Durchschnittsintensität aller überdeckten Pixel im Musterraum benutzt werden. Aus Effizienzgründen begnügt man sich jedoch mit dem Zugriff auf das Texel in der Mitte des zuständigen Texturbereichs.

Damit stellt sich der zugrunde liegende Prozess wie folgt dar:

- Raumkoordinaten des Flächenpunktes berechnen,
- Projektion in den Parameterraum durchführen,
- Texturkoordinaten berechnen
- Texturwerte ggf. anpassen
- Erscheinungsbild mit dem Texturwert modifizieren.

Nachdem zunächst das Texturing anhand der häufigsten Anwendung - dem *Texture Mapping* - beschrieben wird, werden eine Reihe weiterer Texturing-Techniken vorgestellt.

19.1 Texture Mapping

Beispiel: Eine Fläche der Szene soll den Eindruck erwecken als handle es sich um eine Mauer aus Steinen und Fugen. Dazu wird ein Foto von einer solchen Mauer auf die Fläche ”geklebt”. Immer, wenn die Fläche gerastert wird, werden die oben genannten Schritte für jedes Bildschirmpixel der Fläche durchgeführt.

- Die Koordinaten im DC werden für die Eckpunkte durch die Projektion WC_DC und für alle inneren Punkte durch Interpolation zwischen den Eckpunkten bestimmt. Es ergibt sich ein Tripel (x,y,z).
- Eine zweidimensionale Textur, wie sie fast immer vorliegt, hat zwei Parameter u und $v \in [0;1]$, die die Textur parametrisieren. Im zweiten Schritt wird die Projektion der DC-Koordinaten in den (u,v)-Raum berechnet. Im Mauer-Beispiel würde man eine perspektivische Projektion durchführen, um den korrekten perspektivischen Eindruck zu erhalten. Wenn z.B. eine Weltkarte um eine Kugel ”gewickelt” werden soll, um den Eindruck eines Globusses zu erwecken, müsste eine sphärische Projektion gewählt werden. Andere mögliche Projektionsarten sind zylindrisch, elliptisch oder kubisch. Alle Projektionen sind Abbildungen; daher das Wort ”Mapping” im Titel der diversen Techniken.
- Im darauffolgenden Schritt wird aus den Werten für u und v das Texture Element oder *Texel* für das aktuelle Pixel bestimmt. Diese Abbildung führt also aus dem Parameterraum in den Texturraum. Wenn die Mauer-Textur beispielsweise 256×256 Pixel groß ist, würden die Werte für u und v mit 256 multipliziert und der Nachkommaanteil abgeschnitten, um die ganzzahligen Koordinaten des Texels zu erhalten. Bei der Wahl dieser *Korrespondenzfunktion (corresponder function, siehe Abbildung 19.2)* stehen folgende Typen zur Verfügung:

  - *wrap, repeat, tile*: Das Bild wiederholt sich auf der Fläche. Dies wird erreicht, indem der Vorkommaanteil der Parameterwerte vernachlässigt wird. Diese Art des Texturing wird gewählt, wenn ein Material die ganze Fläche durch Wiederholung überziehen soll. Im Mauer-Beispiel wäre ein Bild ausreichend, das wenige Steine und Fugen enthält. Allerdings muß darauf geachtet werden, daß die Ränder des Bildes oben und unten bzw. rechts und links identisch sind; sonst sind die Nahstellen zwischen den Texturen erkennbar.
  - *mirror*: Wie wrap etc. allerdings wird die Textur abwechselnd im Original und in u- bzw. v-Richtung gespiegelt dargestellt. Dadurch läßt sich die Wiederholung schwerer feststellen.
  - *clamp*: Die Ränder der Textur fungieren als Klammer und werden für Werte > 1 wiederholt. Manche APIs erlauben die Einstellung clamp für einen Parameter und wrap für den anderen.
  - *border*: Werte > 1 werden mit einer extra festgelegten Farbe gerendert. Diese Einstellung läßt sich gut verwenden, wenn ein Aufkleber irgendwo auf einer Fläche plaziert werden soll.

- Die Werte, die bei diesen Koordinaten in der Textur hinterlegt sind, können auf verschiedene Weise genutzt werden, um das Erscheinungsbild der Fläche zu ändern. Im Mauer-Beispiel wären RGB-Werte in der Textur enhalten. Falls die Werte in der Textur für diese Fläche angepasst werden müssen, kann dies im nächsten Schritt geschehen. Wenn die RGB-Werte beispielsweise zu dunkel sind, könnten sie mit einem Faktor > 1 multipliziert werden, um so eine Helligkeitssteigerung zu erreichen.

- Abschließend wird der Texturwert interpretiert. RGB-Werte aus Bildtexturen ersetzen z.B. die diffuse Objektfarbe in der Beleuchtungsgleichung. Neben der harten Ersetzung der Ursprungswerte, können diese auch moduliert werden. Zu welchen Ergebnissen das führt, wird in den folgenden Abschnitten beschrieben.

Wenn in jedem neuen Frame, das gerendert werden soll, ein neues Bild aus einem Film als Textur benutzt wird, erscheint es so, als ob der laufende Film auf die Fläche projiziert wird, wie auf eine Kinoleinwand. Wenn die Texturkoordinaten in jedem Frame neu festgelegt werden, ”wandert” die Textur von Frame zu Frame über die Fläche.

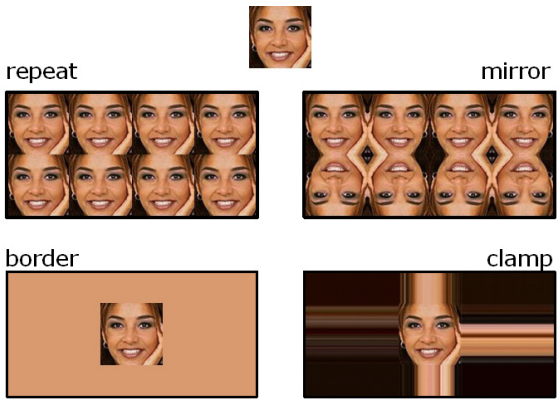


Abbildung 19.2: Auswirkungen der Korrespondenzfunktion

19.2 Mip Mapping

Wenn ein Polygon sich nahe beim Betrachter befindet bzw. weit vom Betrachter entfernt ist, stellt ein Pixel nur einen Teil eines Texels dar bzw. übedeckt ein Pixel mehr als ein Texel. Um den korrekten perspektivischen Eindruck zu erzeugen, muß diese Tatsache durch Antialiasing berücksichtigt werden. Eine Möglichkeit, dies zu tun ist das Mip Mapping (Multum in parvo: Großes im Kleinen). Damit nicht zur Laufzeit für jedes Pixel die Summe über alle überdeckten Texel gebildet werden muß, werden im Speicher schlechter aufgelöste Versionen der Textur vorgehalten. Jede Version hat ein Viertel der Texelzahl der nächstbesser aufgelösten Version (deshalb sind Texturen mit Zweierpotenzen als Kantenlänge ideal). Das Downsampling bei der Berechnung der Texturen kann mit einem Box-Filter - besser mit einem Gauss-Filter - geschehen.

Diese Texturen sind pyramidenförmig angeordnet; die beste Auflösung als Pyramidenboden; die schlechteste Auflösung (1 mal 1 Texel) als Pyramidenspitze. Die Koordinate vom Boden zur Spitze wird als Level of Detail (LOD) bezeichnet. Je nachdem, wie weit das Pixel vom Betrachter entfernt ist, wird es in die entsprechende Version der Textur projiziert, in der es ungefähr ein (oder zwei) Texel überdeckt.

19.3 Light, Gloss und Shadow Mapping

In einer statischen Szene kann die Auswirkung der Beleuchtung durch statische Lichtquellen (z.B.eine Wandlampe,die die Wand beleuchtet) vorab berechnet werden. Für jede Fläche wird eine Lightmap berechnet, die Werte zwischen 0 und 1 enthält. Mit ihr wird später die Beleuchtungsgleichung multipliziert:

$$C_{gesamt,diffus}[x,y] = C_{lighting,diffus}[x,y] * LightMap[u(x,y),v(x,y)]$$

Damit die Szene dabei nicht zuviel an Helligkeit verliert, kann anschließend mit einem Faktor zwischen 1 und 4 multipliziert werden, wobei maximal reines Weiß als Objektfarbe enstehen darf. Der Vorteil durch die Ersparnis an Rechenzeit ist erheblich.

Es besteht auch die Möglichkeit, eine LightMap in Scheinwerferform (weißer Kreis auf schwarzem Grund) zur Laufzeit über eine Fläche wandern zu lassen. Damit wird der Eindruck erzeugt, dass ein Scheinwerfer die Fläche überstreicht.

Gloss Mapping ist Light Mapping für die spekulare Objektfarbe. Z.B. könnten die Ziegel der Mauer glasiert sein. Dann würde die Fläche nur im Bereich der Ziegel spekulare Highlights zeigen.

Auch bei der Berechnung von Schatten kann eine Effizienzsteigerung erreicht werden, wenn die Szene statisch ist. Wie im vorangegangenen Kapitel beschrieben, wird die Szene aus der Sicht jeder Lichtquelle gerendert und die Schatten, die durch andere Flächen auf eine Fläche fallen, werden als Shadow Map bei dieser Fläche gespeichert.

19.4 Alpha Mapping

Wenn eine Fläche Löcher haben soll (z.B. ein Baum, bei dem zwischen den Blättern der Hintergund zu sehen ist), kann eine Aplha Map darauf plaziert werden. Auch hier sind Werte zwischen 0 (die

Fläche ist bei diesem Texel völlig durchsichtig) und 1 (die Fläche ist völlig opak) abgelegt. Wieder wird die Beleuchtungsgleichung modifiziert:

$$C_{gesamt,alpha}[x,y] = C_{Baum,lighting}[x,y] * AlphaMap[u(x,y),v(x,y)] \\ +C_{Hintergrund,lighting} * (1 - AlphaMap[u(x,y),v(x,y)])$$

19.5 Environment oder Reflection Mapping

Im übernächsten Kapitel wird beschrieben, wie beim Raytraycing berechnet wird, welche Objekte sich in anderen spiegeln. Derselbe Effekt kann - etwas weniger akkurat - mit Hilfe von Environment Mapping erzeugt werden. Es wird angenommen, daß die Objekte, die sich in der Fläche spiegeln sollen, weit entfernt sind und daß sich das Objekt, zu dem die Fläche gehört, nicht in sich selber spiegelt. Es wird ein Strahl vom Betrachter zum Pixel der Fläche geschossen und dort reflektiert. Der reflektierte Strahl wird als Verweis in die Environment Map interpretiert und das dort gefundene Texel wird als Modulation der spektularen Objektfarbe benutzt. Die Interpretation des reflektierten Strahls kann z.B. sphärisch, kubisch oder parabolisch geschehen.

19.6 Bump Mapping

Beim Bump Mapping wird durch Veränderung der Flächennormalen anhand einer Bum Map der Eindruck einer rauhen Oberfläche erzeugt. D.h. es wird die Beleuchtungsgleichung modifiziert durch Manipulation der Normalen im Punkt (x,y) der Fläche. Diese Vorgehensweise hat keine physikalische Entsprechung.

Es gibt zwei Möglichkeiten, das Maß der Abweichung in einer Bump Map zu speichern:

- Die Bump Map besteht aus zwei Werten b_u und b_v in jedem Punkt (u,v), die zwei Vektoren skalieren, die senkrecht zueinander und zur Normalen sind und die zur Normalen addiert werden.
- Die Bump Map besteht aus einem Wert pro Texel und die Werte b_u bzw. b_v werden durch Differenzenbildung mit Werten in der Nachbarspalte bzw. Nachbarzeile errechnet.

Per-Pixel Bump Mapping wirkt sehr echt und ist nicht aufwändig zu berechnen. Allerdings werfen die Bumps keine Schatten und wenn der Betrachter flach über eine Fläche hinwegschaut, sind die durch das Bump Mapping suggerierten Höhendifferenzen nicht zu sehen.

19.7 Multitexturing

Jede der Texturing-Techniken für sich betrachtet, erhöht den Realismus der Darstellung. Noch mehr Realismus läßt sich durch die Kombination mehrerer Techniken erzielen. Beim *Multitexturing* werden dazu mehrere Texturen pro Fläche verknüpft. Die Reihenfolge der Verknüpfung und die Art der mathematischen Operationen spielt dabei eine wesentliche Rolle.

19.8 Displacement Mapping

Das Displacement Mapping fällt etwas aus dem Rahmen der anderen Techniken heraus, denn es modifiziert nicht das Erscheinungsbild der Fläche, sondern ihre Geometrie. Dies geschieht, um die Größe der Datei mit der Szenenbeschreibung klein zu halten. Als Basis dient ein Körper mit geringer Flächenanzahl. Jede Fläche des Körpers wird anhand der Displacement Map in mehrere kleine Flächen unterteilt. Dabei können die neuen Punkte nur entlang der Flächennormalen verschoben werden. Sie werden also entweder aus der Fläche herausgehoben oder in die Fläche hineingeschoben. Die Stärke der Verschiebung ist in der Displacement Map als Grauwert hinterlegt. Diese Technik eignet sich besonders gut, um Landschaften kompakt zu beschreiben (analog zu ”Elevation Grid” von X3D) oder um verschiedene Geometrien auf Basis eines Grundkörpers zu definieren.

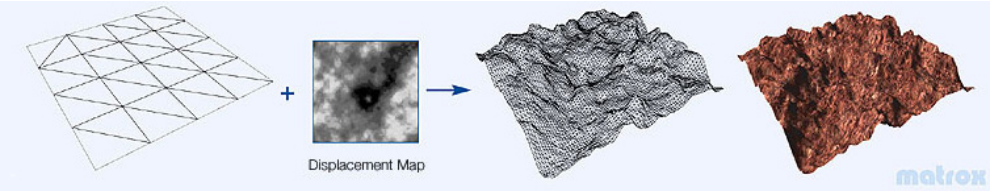


Abbildung 19.3: Displacement Mapping bei der Terraingenerierung (Courtesy by Matrox)

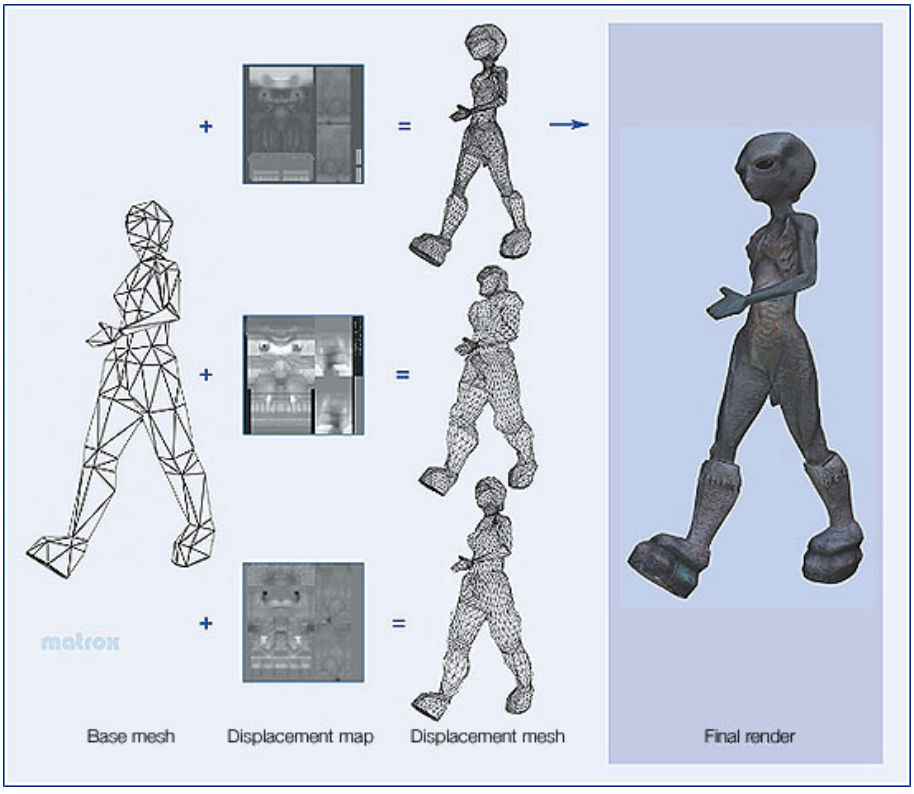


Abbildung 19.4: Generierung von drei Figuren aus einer Geometrie mit Hilfe von Displacement Mapping (Courtesy by Matrox)

19.9 Java-Applet zum Texture-Mapping

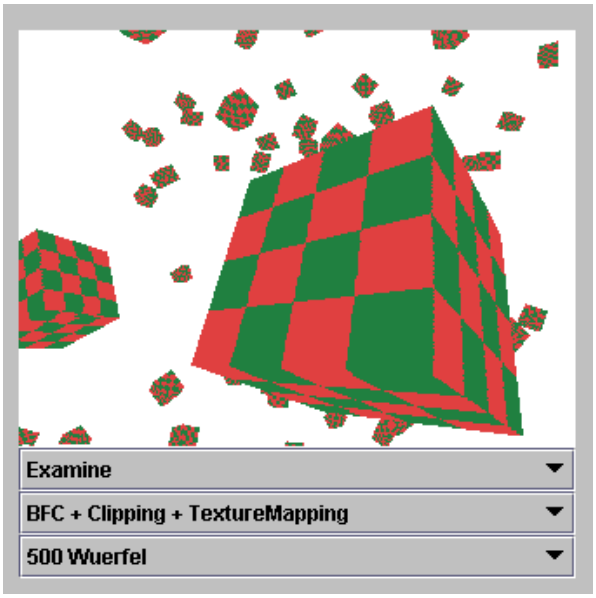


Abbildung 19.5: Screenshot vom Texture-Mapping-Applet

19.10 Java-Applet mit texturiertem Ikosaeder

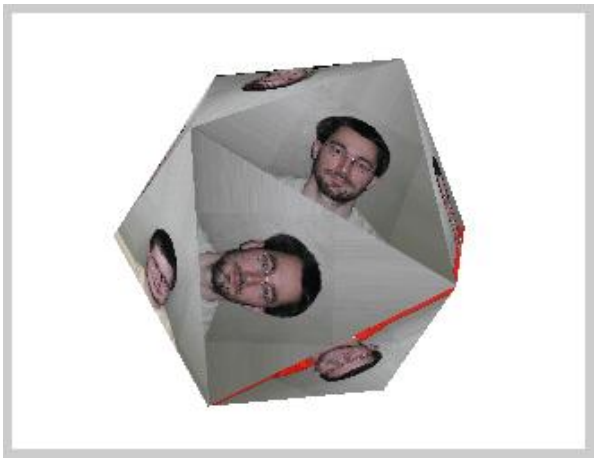


Abbildung 19.6: Screenshot vom Applet mit texturiertem Ikosaeder

Kapitel 20

VRML

VRML, sprich Wörmel, ist eine für das WWW entworfene Virtual Reality Modelling Language zur Beschreibung von 3-dimensionalen Szenen mit multimedialen Komponenten und Animation. Die gerenderte Projektion der Szene kann von jedem Web-Browser betrachtet werden, der über ein passendes Plugin verfügt.

20.1 Geschichte

Bei der Nutzung von Computer-Resourcen läßt sich ein weiterer Paradigmenwechsel beobachten: Während in der EDV-Gründerzeit speziell ausgebildete Techniker am Mainframe-Computer Befehle eintippten, danach einige Jahrzehnte später Kopfarbeiter per Drag & Drop Fensterelemente manipulierten, surft inzwischen jedermann und -frau in einer weltweit vernetzten multimedialen Landschaft. Der Kontext hat sich also von institutionell über persönlich zu sozial gewandelt.

Aus diesem Zeitgeist heraus diskutierten im April 1994 auf der 1st International WWW Conference in Genf Tim Berners-Lee, einer der Väter des WWW, mit den beiden Autoren des Systems Labyrinth, Mark Pesce und Tony Parisi. Es wurde eine Mailing List aufgesetzt, die schon nach wenigen Wochen mit mehr als 1000 Teilnehmern über Syntax für Strukturen, Verhalten und Kommunikation debattierte. Bereits im Oktober 1994 wurde auf der 2nd International WWW Conference in Chicago VRML 1.0 vorgestellt, ein Entwurf, der wesentlich vom Silicon Graphics System Open Inventor inspiriert war. VRML 1.0 konnte bereits geometrische Grundkörper und Polygone in einem Koordiantensystem plazieren und ihre Farbe und Materialeigenschaften spezifizieren. Auch ließen sich durch Hyperlinks Objekte beliebig im Web referieren. Abgesehen von dieser Möglichkeit der Interaktion handelte es sich allerdings um rein statische Szenen.

Diesem Manko sollte eine schnellstens eingerichtete VAG (VRML Architecture Group) abhelfen, welche Überlegungen zur Animation und zur Integration multimedialer Komponenten wie Sound und Video koordinierte. Zur 1st International VRML Conference im Dez. 1995 war es dann soweit: Als Sieger einer Ausschreibung für VRML 97 ging Moving Worlds von Silicon Graphics nach einer On-Line-Wahl. Überarbeitete Syntax sorgte für die Beschreibung statischer Szenen mit multimedialen Bestandteilen und ein neues Event-Handling-Konzept erlaubte Animation dieser Szenen sowie Interaktion mit dem Benutzer.

20.2 Einbettung

VRML-Szenen werden beschrieben in ASCII-Dateien mit der Endung *.wrl, welche innerhalb einer HTML-Seite mit dem EMBED-Kommando referiert werden, z.B.

```
<EMBED SRC=zimmer.wrl  WIDTH=600 HEIGHT=400>
```

Ein entsprechend konfigurierter Web-Server schickt dem anfordernden Clienten als Vorspann dieser Daten den Mime-Typ VRML, worauf das zur Betrachtung installierte Plugin, z.B. Cosmo Player 2.0 von Silicon Graphics, die eingehenden Daten in eine interne Datenstruktur einliest, von wo sie zur Berechnung einer fotorealistischen Projektion verwendet werden. In welcher Weise Blickwinkel und Orientierung in der Szene modifiziert werden können, bleibt dem Plugin überlassen: Mit Mauszeiger und Keyboard Shortcuts wandert der Benutzer durch eine virtuelle Welt, verkörpert im wahrsten Sinne des Wortes durch einen Avatar, seiner geometrischen Repräsentation, beschränkt in seiner Beweglichkeit durch physikalische Restriktionen und durch eine simulierte Schwerkraft.

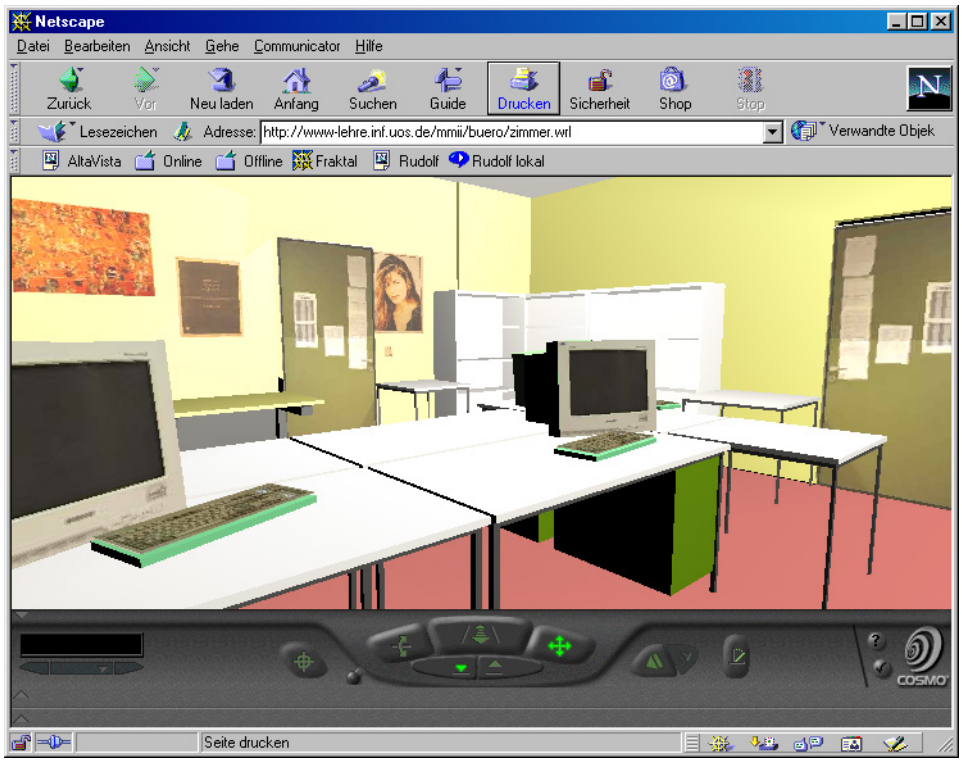


Abbildung 20.1: Screenshot vom Cosmo Player Plugin

20.3 Geometrie

Wichtigster Bestandteil von VRML-Szenen ist der sogenannte Knoten (meistens mit großem Anfangsbuchstaben geschrieben) der ähnlich eines Programmiersprachenrecords aus Feldern verschiedenen Typs besteht (meistens klein geschrieben). Diese Felder verweisen entweder auf nicht weiter strukturierte Objektknoten oder andere Gruppenknoten, die wiederum mittels ihrer Felder verzweigen können.

Beispiel 1 zeigt den Aufbau einer Szene, in der eine Kugel mit Radius 1.5 im Ursprung des Weltkoordinatensystems plaziert wird. Die x-Richtung entspricht der horizontalen Bewegung, y beschreibt die vertikale Richtung und z wächst auf den Betrachter zu. Der Sphere-Knoten hat dabei als einziges (optionales) Feld den Radius. Diese Kugel wird referiert über das geometry-Feld des Shape-Knotens, zuständig für die Gestaltung eines Objekts. Über das appearance-Feld wird die Materialbeschaffenheit in Form einer RGB-Farbe und eines Reflexions-Koeffizienten spezifiziert. Der Shape-Knoten wiederum ist als eins der Kinder im Transform-Knoten eingetragen, der über ein translation-Feld für die Verschiebung der Kugel sorgt.

```
# VRML V2.0 utf8
# kugel.wrl:
# gruene, stark reflektierende  Kugel mit Radius 1.5

Transform {                         # Plaziere
  translation 0 0 0                 # im Ursprung
  children [
    Shape {                         # eine Gestalt
      geometry Sphere {             # von der Form einer Kugel
        radius 1.5                  # mit Durchmesser 1.5
      }
      appearance Appearance {       # in der Erscheinung
        material Material {         # mit Materialbeschaffenheit
          diffuseColor 0 1 0        # gruene Farbe
          shininess 0.9             # stark reflektierend
        }
      }
    }
  ]
}
```

kugel.wrl

20.4 Polygone

Neben den Grundbausteinen Sphere (Kugel), Box (Quader), Cone (Kegel) und Cylinder (Zylinder) lassen sich eigene geometrische Gebilde konstruieren. Ausgehend von einer Liste von 3-D-Punkten im Raum werden jeweils gegen den Uhrzeigersinn alle Punkte durchlaufen, die ein Face (durch Polygon approximierte Körperfläche) aufspannen. Beispiel 2 zeigt die Definition einer 5-farbigen Pyramide mit quadratischer Grundfläche.

```
# VRML V2.0 utf8
# pyramide.wrl: selbstdefinierte 5-seitige Pyramide

Shape {
  geometry IndexedFaceSet {

    coord Coordinate {
      point [            # beteiligte Punkte
         0  3  0         # 0. Pyramidenpunkt (Spitze)
         0  0 -2         # 1. Pyramidenpunkt (Norden)
        -2  0  0         # 2. Pyramidenpunkt (Westen)
         0  0  2         # 3. Pyramidenpunkt (Sueden)
         2  0  0         # 4. Pyramidenpunkt (Osten )
      ]
    }

    coordIndex [         # Polygone gegen Uhrzeiger, Ende: -1
         4  3  2  1 -1   # 0. Face: Punkte 4 3 2 1 (Grundflaeche)
         0  1  2    -1   # 1. Face: Punkte 0 1 2   (Nordwesten)
         0  2  3    -1   # 2. Face: Punkte 0 2 3   (Suedwesten)
         0  3  4    -1   # 3. Face: Punkte 0 3 4   (Suedosten)
         0  4  1         # 4. Face: Punkte 0 4 1   (Nordosten)
    ]

    colorPerVertex FALSE
    color Color {
        color [          # pro Face eine Farbe benennen
          0 1 1          # 0. Face: Cyan
          1 0 0          # 1. Face: Rot
          1 1 0          # 2. Face: Gelb
          0 1 0          # 3. Face: Gruen
          0 0 1          # 4. Face: Blau
        ]
    }
  }
}
```

pyramide.wrl

20.5 Wiederverwendung

Zur Reduktion der Dateigrößen und zur besseren Lesbarkeit lassen sich einmal spezifizierte Welten wiederverwenden. Beispiel 3 zeigt die Kombination der beiden graphischen Objekte Kugel und Pyramide, wobei die Pyramide leicht nach hinten gekippt oberhalb der Kugel positioniert wird. Ferner wurde ein Hyperlink eingerichtet, der zu einer weiteren VRML-Welt führt, sobald der Mauszeiger auf der Kugel gedrückt wird.

```
# VRML V2.0 utf8
# gruppe.wrl:
# Kugel mit Hyperlink unter gekippter Pyramide

Transform{                              # Plaziere
  children[                             # die folgenden Objekte:

    Anchor {                            # Hyperlink
      url "multimedia.wrl"              # zur VRML-Datei textur.wrl
      description "Next world"          # Anzeige im Statusfenster
      children[                         # zugeordnet ist die Welt
        Inline {url "kugel.wrl"}        # aus Datei kugel.wrl
      ]
    }

    Transform {
      translation  0 1   0              # hebe 1 Einheit nach oben
      scale        1 0.5 1              # skaliere bzgl. y auf 50 %
      rotation     1 0   0 -0.523333    # kippe um 30 Grad nach hinten
      children[                         # verwende Welt
        Inline {url "pyramide.wrl"}     # aus Datei pyramide.wrl
      ]
    }

  ]
}
```

gruppe.wrl

20.6 Multimedia

Neben geometrischen Strukturen können VRML-Szenen auch multimediale Bestandteile wie Bilder, Audio und Video enthalten.

Beispiel 4 zeigt einen Würfel, auf den ein jpg-Bild als Textur aufgebracht wurde. Einem Sound-Knoten ist per URL eine Wave-Datei mit Position und Schallrichtung zugeordnet.

Sobald der Betrachter bei Annäherung an den Würfel eine gewissen Grenzwert überschritten hat, beginnt der Sound-Knoten mit dem Abspielen der Musik.

```
# VRML V2.0 utf8
# multimedia.wrl:
# Quader mit Bild-Textur und Soundquelle

#VIEWPOINT{0 0 20}                # 20 Einheiten vor dem Ursprung

Shape {                           # ein Gestaltknoten
  geometry Box {size 1 1 1}       # ein Quader der Kantenlaenge 1
  appearance Appearance {         # mit dem Aussehen
    texture ImageTexture {        # einer Bild-Textur
      url "posaune.jpg"           # aus der JPEG-Datei posaune.jpg
    }
  }
}

Sound {                           # ein Soundknoten
  source AudioClip {              # gespeist von einem Audio-Clip
    url  "party.wav"              # aus der Wave-Datei party.wav
    loop TRUE                     # in einer Endlosschleife
  }

  location  0 0 0                 # Schallquelle im Ursprung
  direction 0 0 1                 # dem Betrachter zugewandt
  minFront 1                      # Hoerbereichsanfang
  maxFront 8                      # Hoerbereichsende
}
```

multimedia.wrl

20.7 Interaktion

VRML97 bietet zahlreiche Möglichkeiten, mit denen einer Szene Dynamik und Interaktion verliehen werden kann. Die zentralen Bausteine für die hierzu erforderliche Ereignisbehandlung sind die EventIn- bzw. EventOut-Felder von Knoten, mit denen Meldungen empfangen und Zustandsänderungen weitergeschickt werden können. Es gibt Time,- Proximity-, Visibility-, Collision- und Touch-Sensoren, welche das Verstreichen einer Zeitspanne, das Annähern des Benutzers, die Sichtbarkeit von Objekten, das Zusammentreffen des Avatars mit einem Objekt und die Berührung mit dem Mauszeiger signalisieren. Verständlicherweise müssen Typ des verschickenden Ereignisfeldes und Typ des empfangenden Ereignisfeldes übereinstimmen.

Beispiel 5 zeigt die Kugel versehen mit einem Touch-Sensor, welcher bei Mausdruck eine Nachricht an den Soundknoten schickt, der auf diese Weise seinen Spielbeginnzeitpunkt erhält und die zugeordnete Wave-Datei startet.

```
# VRML V2.0 utf8
# interaktion.wrl:
# Kugel macht Geraeusch bei Beruehrung

Group {                              # plaziere Gruppenknoten
  children [                         # bestehend aus
    DEF Taste TouchSensor {}         # einem Touch-Sensor
    Inline { url "kugel.wrl" }       # und einer Kugel
  ]
}

Sound {                              # plaziere Soundknoten
  source DEF Tut AudioClip {         # gespeist von Audio-Clip
    url "tut.wav"                    # aus der Wave-Datei tut.wav
  }
  minFront  5                        # Anfang des Schallbereichs
  maxFront 50                        # Ende des Schallbereichs
}

ROUTE Taste.touchTime                # bei Beruehrung der Kugel
   TO Tut.set_startTime              # schicke Systemzeit an den Knoten Tut
```

interaktion.wrl

20.8 Animation

Die benutzergesteuerte oder automatische Bewegung von Objekten und Szenenteilen wird wiederum mit Hilfe der Ereignisbehandlung organisiert. Im Beispiel 6 wird die Ziehbewegung des gedrückten Mauszeigers zur Manipulation der lokalen Translation eines Objekts verwendet und das regelmäßige Verstreichen eines Zeitintervalls löst eine Nachricht an denselben geometrischen Knoten aus, der hierdurch seinen aktuellen Rotationsvektor erhält. Eine solche Konstruktion verlangt einen Orientation-Interpolator, welcher beliebige Bruchzahlen zwischen 0 und 1 auf die zugeordneten Werte seines Schlüsselintervalls abbildet, hier bestehend aus allen Drehwinkeln zwischen 0 und 3.14 (=180 Grad beschrieben in Bogenmaß), bezogen auf die y-Achse.

```
# VRML V2.0 utf8
# animation.wrl:
# selbstaendig sich drehende und interaktiv verschiebbare Pyramide

DEF Schieber PlaneSensor {}            # Sensor zum Melden einer Mausbewegung

DEF Timer TimeSensor {                 # Sensor zum Melden eines Zeitintervalls
  cycleInterval 5                      # Dauer 5 Sekunden
  loop TRUE                            # Endlosschleife
}

DEF Rotierer OrientationInterpolator{  # Interpolator fuer Rotation
  key      [0 , 1]                     # bilde Schluessel 0 und 1 ab auf
  keyValue [ 0  1  0  0                #   0 Grad Drehung bzgl. y
             0  1  0  3.14]            # 180 Grad Drehung bzgl. y
}

DEF Pyramide Transform {               # plaziere Objekt mit Namen Pyramide
      children [                       # bestehend aus
        Inline {url "pyramide.wrl"}    # VRML-Datei pyramide.wrl
      ]
}

ROUTE Timer.fraction_changed           # falls Zeitintervall sich aendert
   TO Rotierer.set_fraction            # schicke Bruchteil an Rotierer

ROUTE Rotierer.value_changed           # falls Drehung sich aendert
   TO Pyramide.set_rotation            # schicke Drehwert an Pyramide

ROUTE Schieber.translation_changed     # falls gedrueckter Mauszeiger bewegt wird
   TO Pyramide.set_translation         # schicke Translationswert an Pyramide
```

animation.wrl

20.9 Scripts

Manchmal reichen die in VRML angebotenen Funktionen wie Sensoren und Interpolatoren nicht aus, um ein spezielles situationsbedingtes Interaktionsverhalten zu erzeugen. Abhilfe schafft hier der sogenannte Script-Knoten, welcher Input empfangen, Daten verarbeiten und Output verschicken kann. Z.B. kann eine vom Touch-Sensor geschickte Nachricht eine Berechnung anstoßen, deren Ergebnis in Form einer Translations-Nachricht an ein bestimmtes Objekt geschickt und dort zur Neupositionierung genutzt wird.

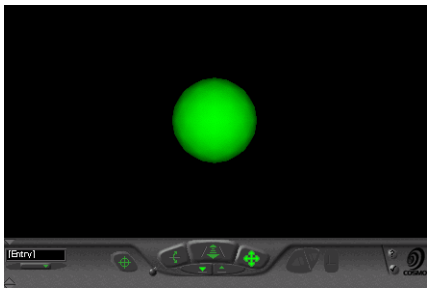
```
# VRML V2.0 utf8
# javascript.wrl: Rotation eines Objekts ueber Javascript

Viewpoint {position  0 2 8}             # Augenpunkt

Group {                                 # gruppiere
  children [                            # folgende Objekte
    DEF Taste TouchSensor{}             # Beruehrungssensor Taste
    DEF Pyramide Transform {            # Objekt Pyramide
      children[                         # bestehend aus
        Inline {url "pyramide.wrl"}     # VRML-Welt pyramide.wrl
      ]
    }
  ]
}

DEF Aktion Script {                     # Script mit Namen Aktion
  eventIn  SFBool     isActive          # Input-Parameter
  eventOut SFRotation drehung           # Output-Parameter
  url [                                 # gespeist von
     "javascript:                      // inline-Javascript
     function isActive(eventValue) {   // fuer eventIn zustaendig
       if (eventValue == true) {       // falls eventValue den Wert wahr hat
         drehung[0] = 0.0;             // drehe
         drehung[1] = 1.0;             // bzgl.
         drehung[2] = 0.0;             // der y-Achse
         drehung[3] += 0.174444;       // um weitere 10 Grad
       }
     }"
  ]
}

ROUTE Taste.isActive                    # bei Beruehren der Pyramide
   TO Aktion.isActive                   # sende Nachricht an das Script Aktion

ROUTE Aktion.drehung                    # vom Script Aktion erzeugter Dreh-Vektor
   TO Pyramide.set_rotation             # wird an die Pyramide geschickt
```

javascript.wrl

Die Formulierung des Berechnungsalgorithmus geschieht entweder durch ein Javascript-Programm, inline gelistet im Script-Knoten, oder durch eine assoziierte Java-Klasse, welche in übersetzter Form

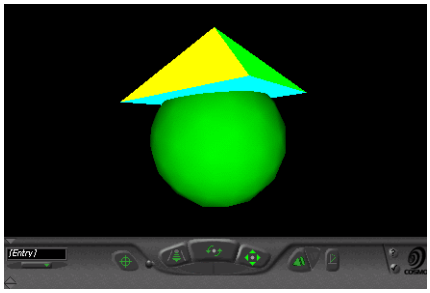
mit Dateiendung *.class lokal oder im Netz liegt. Zum Übersetzen der Java-Quelle ist das EAI (External Authoring Interface) erforderlich, welches in Form einiger Packages aus dem Verzeichnis importiert wird, in welches sie das VRML-Plugin bei der Installation deponiert hatte.

Beispiel 7 zeigt die Pyramide zusammen mit einem Java-Script, welches bei jedem Aufruf den Drehwinkel bzgl. der y-Achse um weitere 10 Grad erhöht.



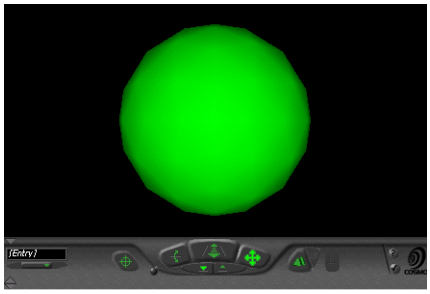
kugel.wrl

pyramide.wrl

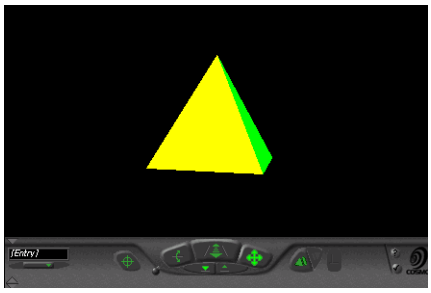


gruppe.wrl

multimedia.wrl



interaktion.wrl



animation.wrl

Abbildung 20.2: Screenshots der Beispiele 1 - 6

20.10 Multiuser

Eines der ursprünglichen Entwicklungsziele von VRML bleibt auch bei VRML 97 offen: es gibt noch keinen Standard für Multiuser-Welten. Hiermit sind Szenen gemeint, in denen mehrere Benutzer gleichzeitig herumwandern und interagieren können. Da jedes ausgelöste Ereignis von allen Beteiligten wahrgenommen werden soll, muß ein zentraler Server den jeweiligen Zustand der Welt verwalten und den anfragenden Klienten fortwährend Updates schicken. In diesem Zusammenhang erhält der Avatar eine aufgewertete Rolle: Zusätzlich zu seiner geometrischen Räumlichkeit, die schon zur Kollisionsdetektion in einer Single-User-Szenerie benötigt wurde, muß nun auch sein visuelles Äußeres spezifiziert werden, sicherlich ein weiterer wichtiger Schritt zur Verschmelzung eines real existierenden Benutzers mit der von ihm gespielten Rolle.

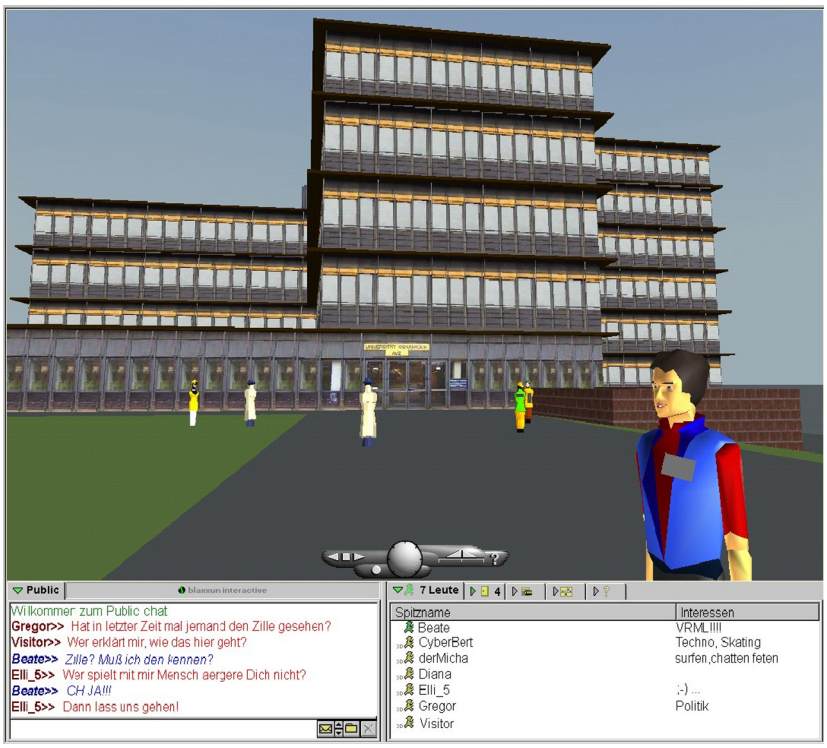


Abbildung 20.3: Screenshot vom Blaxxun Contact Plugin

Unter der Einstiegsseite `http://www-lehre.inf.uos.de/avz` kann das *Virtuelle AVZ* der Universität Osnabrück betreten werden, welches mit Hilfe eines Blaxxun-Community-Servers als Multiuser-Welt realisiert wurde.

Unter der Adresse `http://www-lehre.inf.uos.de/~cg/2008/VRML/uebersicht.html` sind die vorgenannten Dateien und weitere VRML-Beispiele abrufbar.

20.11 X3D

Als Nachfolger von VRML ist X3D vorgesehen, eine Beschreibungssprache für 3D-Welten auf Basis von XML. Ein Werkzeug zum Erstellen von X3D-Dateien bietet `http://www.vizx3d.com;` dort gibt es auch den Flux-Player zum Anzeigen von X3D-Welten im Web-Browser.

```
#VRML V2.0 utf8

Transform {
  translation  -0.03   0.00  -0.052
  rotation      0.82  -0.56  -0.039  2.10
  children [
    Shape {
      appearance Appearance {
        material Material {
          ambientIntensity 0.2
          shininess        0.2
          diffuseColor     1 0 0
        }
      }
      geometry Box {
        size 1 1 1
      }
    }
  ]
}
```

Beschreibung eines roten Würfel in VRML

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN"
  "http://www.web3d.org/specifications/x3d-3.0.dtd">
<X3D>
  <Scene>
    <Transform translation= "-0.03   0.00  -0.052"
               rotation=    " 0.82  -0.56  -0.039  2.10">
    <Shape>
      <Appearance>
        <Material ambientIntensity  ="0.2"
                  shininess         ="0.2"
                  diffuseColor      ="1 0 0" />
       </Appearance>
       <Box size="1 1 1"/>
    </Shape>
    </Transform>
  </Scene>
</X3D>
```

Beschreibung eines roten Würfel in X3D

Kapitel 21

OpenGL 3.1

21.1 Einordnung und Motivation

In der Rastergrafik, deren Grundlagen in den Kapiteln 13 bis 19 behandelt wurden, wird Materie als Geometrie modelliert und zusätzlich werden deren Oberflächeneigenschaften beschrieben. Dies geschieht, etwa im Falle der Auswertung des lokalen Beleuchtungsmodells, für jeden Punkt unabhängig vom Rest der Szene. Weil in das direkte Beleuchtungsmodell ausschließlich die Lichtquellen und Eigenschaften der Materie in diesem Punkt, nicht jedoch eventuell in der Verbindungsstrecke liegende Materie eingehen, existieren beispielsweise keine Schatten. Stattdessen muss der Programmierer eigene Routinen definieren, mit den Schatten explizit geworfen und nachträglich in die Szene eingefügt werden können. Weiterhin wurde, um nicht im direkten Einflussbereich einer Lichtquelle befindliche Materie überhaupt sichtbar erscheinen zu lassen, der Term des ambienten Lichts eingeführt. In Wirklichkeit erhalten diese Bereiche einer Szene durch indirektes Licht ihre Beleuchtung, welches ein oder mehrmals von anderer Materie in der Szene reflektiert wurde. Leuchtet ein Modell die Szene vollständig auf diese Weise aus und verzichtet auf einen ambienten Term, spricht man von Globaler Beleuchtung. Dementsprechend hat die beschriebene Vorgehensweise der Rastergrafik nahezu nichts mit den physikalischen Gesetzen, etwa der Lichtausbreitung, gemein.
Besonders deutlich zeigen sich die Unzulänglichkeiten des Modells anhand von Materialien, deren Eigenschaften nicht allein durch ihre Oberfläche beschrieben werden können. Dazu zählen Flüssigkeiten und Gase aber auch ein Beleuchtungsmodell der menschlichen Haut muss z.B. tiefer liegende Schichten berücksichtigen, um kein unnatürliches, „plastikartiges“, Aussehen zu erzeugen. Zwar existiert für die angesprochenen Probleme, gefördert u.a. durch zahlreiche Computerspiele, eine äußerst umfangreiche Sammlung von Algorithmen, welche oft als „Effekte“ bezeichnet werden. Jedoch simulieren diese „Hacks“ das Verhalten derartiger Materialien üblicherweise nicht physikalisch korrekt und liefern dementsprechend nur unter bestimmten Rahmenbedingungen glaubwürdige Ergebnisse. Beispielsweise kann ein 3D Oberflächenmodell einer Wolke, aus einer gewissen Entfernung betrachtet, einigermaßen glaubwürdig wirken. Nähert sich jedoch der Betrachter, werden die unrealistisch scharfen Kanten erkennbar. Ein denkbarer Trick, ohne physikalische Entsprechung, ist nun, diese Kanten nachträglich zu „verwischen“. Weiterhin genügt es gerade im Falle von Wolken nicht, lediglich ein lokales Beleuchtungsmodell für die Oberfläche auszuwerten. Stattdessen hängt die Beleuchtung in einem Punkt der Wolke u.a. von der Strecke ab, die Lichtstrahlen durch diese und andere Wolken auf dem Weg zu der Stelle durchdringen mussten, sowie Eigenschaften der Materie (Dichte, Konsistenz, ...) auf dem zurückgelegten Pfad. Zusätzlich werden die Teilchen auf diesem

Weg ebenfalls angeregt und strahlen ihrerseits Licht ab. Aufgrund der beschriebenen Funktionsweise der Rastergrafik kann eine allgemeingültige Nachbildung solcher optischer Erscheinungen beliebiger Materialien als sehr unwahrscheinlich eingestuft werden.

Im Gegensatz dazu steht Ray Tracing, in der Computergrafik ein Oberbegriff einer Klasse von Algorithmen, die Strahlen durch eine virtuelle Szene schicken, um ein Bild zu erzeugen. Diese Vorgehensweise ist weitaus näher an der realen Physik, weshalb viele der Algorithmen allgemeiner einsetzbar, vergleichsweise intuitiv verständlich und auch in der Computergrafik als Wissenschaft wesentlich weniger umstritten sind. Ein Beispiel eines einfachen Ray Tracers mit einem lokalen Beleuchtungsmodell folgt in Kapitel 23 dieses Skripts. Im Allgemeinen werden die Strahlen je nach Variante vom Betrachterstandpunkt und/oder von den einzelnen Lichtquellen ausgesandt. Die aufwändigeren Varianten können durchaus die Strahlen solange reflektieren und verfolgen, bis vom Betrachter ausgehende Srtahlen eine (nicht punktförmige) Lichtquelle oder von den Lichtquellen ausgesandte die Bildebene erreichen. Folglich ist Globale Beleuchtung umsetzbar und auch die anderen obengenannten Materialeigenschaften lassen sich, mit Varianten aus dem verwandten Bereich Volume Rendering, simulieren. Umgekehrt lässt sich nichts durch Rastergrafik darstellen, das nicht auch durch einen Raytracer erreichbar ist.
Aus diesen Gründen hat Rastergrafik im Bereich High Quality Graphics mittlerweile nahezu keinerlei Relevanz mehr. Ganz anders sieht es dagegen im Bereich Interactive Graphics aus. Denn dort ist nicht die bestmögliche Grafik, sondern die beste, noch in Echtzeit erzeugbare Darstellung, ausschlaggebend. Hier liegt die Rastergrafik derzeit weit in Führung, da der ihr zugrunde liegende Brute Force Ansatz hervorragend zur Ausführung auf moderner Grafikhardware geeignet ist. Nicht nur können Primitives, Vertices und Fragments völlig unabhängig voneinander verarbeitet werden, sondern zusätzlich werden identische Operationen auf großen Teilmengen davon ausgeführt, wovon die SIMD-artige Architektur moderner Grafikprozessoren (GPU) sehr stark profitiert. Zusätzlich können einige Teile der Berechnungen aufgrund ihrer Einfachheit sogar in fester Hardware umgesetzt werden, wodurch sich die Leistung weiter steigert. Dieser durch die Grafikhardware ermöglichte Performancevorteil von mehreren Größenordnungen im Vergleich mit einer Umsetzung auf CPUs sichert zumindest gegenwärtig die Daseinsberechtigung der Rastergrafik in Echtzeitanwendungen. Ansätze zur Beschleunigung der wesentlich flexibleren Ray Tracing Algorithmen sind bislang nicht von vergleichbarem Erfolg gekrönt.
Die bekanntesten APIs für Rastergrafik mit Hardwareunterstützung sind Microsofts DirectX unter Windows und die freie Bibliothek OpenGL. Ersteres zeichnet sich durch seine Leistungsfähigkeit, die breite Verfügbarkeit ausgereifter Implementierungen unter Windows sowie Hilfswerkzeuge für die Entwicklung aus. Das primäre Einsatzgebiet stellen traditionell Computerspiele dar. In letzter Zeit wird es jedoch auch vermehrt zur Forschung im Bereich der Computergrafik eingesetzt. OpenGL ist dagegen nicht nur unter Windows, sondern für die verschiedensten Betriebssysteme und Plattformen verfügbar. Aufgrund des daraus resultierenden breiten Anwendungsspektrums, etwa für Grafik im Web oder für Smartphones wie dem iPhone, liegt der Fokus dieser Veranstaltung auf OpenGL.
Gegenstand dieses Kapitels ist eine Einführung in OpenGL 3.1. Sofern nicht anders gekennzeichnet, beziehen sich alle Aussagen auf diese Version, auch wenn vereinfachend die Bezeichnungen OpenGL oder GL Verwendung finden. Vermittelt werden sollen dabei primär die zum Umsetzen einfacher Computergrafik Algorithmen dieser Veranstaltung mit Grafikhardwareunterstützung nötigen Kenntnisse.

21.2 Einleitung

OpenGL stellt eine Menge von Kommandos bereit, welche zum einen die Spezifikation von geometrischen Objekten in zwei oder drei Dimensionen ermöglichen und zum anderen solche die festlegen wie diese Objekte in einen Framebuffer gerendert werden. Die geometrischen Objekte setzen sich aus elementaren grafischen Grundformen (Primitives) zusammen. GL-Primitives sind ausschließlich: Punkt, Liniensegment und Dreieck. Nicht enthalten sind Routinen zum Beschreiben oder Verwalten komplexerer Objekte wie Kugel, Spline, Mesh, etc.. Deren Modellierung und Repräsentation durch die GL-Primitives bleibt der Applikation bzw. einer auf der GL aufsetzenden Hilfsbibliothek überlassen.

Die Zielsetzung von OpenGL ist, den Zugriff auf die Fähigkeiten der Grafikhardware auf der tiefsten Ebene zu gewährleisten, die noch hardwareunabhängig ist. Über ein einziges Interface soll der volle Funktionsumfang der GL auf verschiedenen Plattformen verfügbar sein, wobei ggf. fehlende Funktionalitäten in Software emuliert werden müssen. Zusätzlich vorhandene Funktionen können mithilfe sogenannter Extensions bereitgestellt werden, welche dann in einer Implementation berücksichtigt werden können aber nicht müssen. Auf diese Weise können zum einen neue Features sehr schnell integriert und zum anderen herstellerspezifische Eigenarten berücksichtigt werden. Aufgrund der Ausrichtung als procedurale API ist der Programmierer gezwungen, alle zum Rendern benötigten Schritte zu konfigurieren bzw. selbst im Detail festzulegen. Dies stellt einen fundamentalen Unterschied im Vergleich zu deskriptiven APIs (etwa Szenegraphen wie VRML, Kapitel 20) dar, bei denen der Programmierer das Aussehen einer Szene angibt und der Bibliothek die Details zu rendern überlässt. Dementsprechend benötigt der Programmierer sehr genaue Kenntnisse über die Graphics Pipeline, kann dafür jedoch eigene Algorithmen implementieren.

OpenGL ist, von der integrierten Shading Language GLSL abgesehen, eine API und keine Programmiersprache. Eine OpenGL Applikation ist demnach in einer anderen Sprache geschrieben und verwendet zusätzlich Befehle aus der OpenGL Bibliothek. Ein zentrales Merkmal ist die Auslegung zur Unterstützung verschiedener Plattformen (Smartphones, Spielkonsolen, PCs, Macs, etc.) und Betriebssysteme (Windows, MacOs, Linux, etc.). Weiterhin ist die API aus vielfältigen Sprachen wie etwa Fortran, C, C++, Java, Ruby, Php und sogar JavaScript heraus ansprechbar. In dieser Vorlesung wird OpenGL in Java Applets integriert, wobei die Bibliothek JOGL den Zugriff auf die nativen Funktionen ermöglicht. Auf diese Weise können die Beispielprogramme der Veranstaltung direkt in einem Webbrowser ausgeführt werden.

In der Spezifikation sind für eine Menge von ca. 250 Funktionen deren Verhalten, Wechselwirkungen mit anderen GL Funktionen sowie Auswirkungen auf den GL State nicht aber deren genaue Funktionsweise definiert. Eine Implementation dieser Spezifikation kann sowohl eine reine Software Bibliothek oder ein Treiber zusammen mit einer Grafikkarte sein. In dem Treiber ist dann festgelegt, welche Anteile einzelner Kommandos von der CPU oder der Grafikhardware übernommen werden. Wie groß der Anteil der Hardwarebeschleunigung ist, bleibt i.d.R. vor dem Programmierer verborgen. Da OpenGL auf sehr unterschiedlichen Zielplattformen implementiert werden können soll, legt die Spezifikation lediglich das ideale Verhalten fest. Teilweise ist eine bestimmte Abweichung von diesem Ideal erlaubt, sodass verschiedene OpenGL Implementationen bei der Ausführung identischer GL Operationen nicht zwingend für jeden Pixel übereinstimmende Ergebnisse liefern. Insgesamt legt

die Spezifikation die Funktionalität und das Ergebnis eines Rendervorgangs fest, allerdings ist dessen Qualität etwas und die zu erwartende Performance sehr stark implementationsabhängig.

21.3 Entwicklungsgeschichte

Die Entwicklung von OpenGL begann 1992 mit der Veröffentlichung der Spezifikation (Version 1.0) durch Silicon Graphics Inc. (SGI). Im selben Jahr wurde das OpenGL Architectural Review Board gegründet, dessen Mitglieder in den folgenden Jahren den Standard pflegen und erweitern sollten. Die Gründungsmitglieder der OpenGL ARB waren: SGI, Digital Equipment Corporation, IBM, Intel und Microsoft. Seit 2006 liegt die Kontrolle über den OpenGL Standard bei der Khronos Group, einem Konsortium zur Entwicklung offener Medienstandards auf verschiedenen Plattformen. Es besteht aus über hundert Mitgliedern, einige der Board Member sind: AMD/ATI, Apple, ARM, Imagination Technologies, Intel, Motorola, Nokia, nVidia, Samsung, Sony und Sun. Im Folgenden sollen einige wichtige Stationen von OpenGL hinsichtlich der Entwicklung eines Standards für hardwarebeschleunigte Grafik skizziert werden.

- **1.x**, ab 1992
  In frühen Versionen von OpenGL (1.X) bestand die Graphics Pipeline aus einer Folge fester, aber bis zu einem gewissen Grad konfigurierbarer Schritte (fixed Pipeline). So konnte beispielsweise die Beleuchtung mit Parametern für einzelne Bestandteile wie dem spekularen oder diffusen Anteil eingestellt werden. Nicht ohne weiteres möglich war dagegen die Implementierung eines eigenen Beleuchtungsmodells oder auch nur die Auswertung auf Fragment- statt Vertexebene. Das in diesem Skript als Phong Shading bezeichnete Beleuchtungsmodell war somit beispielsweise nicht verfügbar.

- **2.x**, ab 2004
  Um die stark eingeschränkte Flexibilität der festen Pipeline zu überwinden, wurde die programmierbare Pipeline eingeführt, bei der die starre Vertex- und Fragmentverarbeitung durch Vertex- und Fragment Shader ersetzt wurde. Dabei handelt es sich um praktisch beliebige auf der Grafikkarte ausführbare Programme, welche in die restliche weiterhin feste Pipeline integriert und anstelle der entsprechenden zuvor festen Schritte ausgeführt werden. Damit konnten zum einen sehr viel innovativere und oft auch effizientere Algorithmen implementiert werden und zum anderen war der Grundstein zum Nutzung der Grafikhardware für allgemeine Berechnungen (GPGPU) gelegt.

- **3.x**, ab 2008
  Über lange Zeit zeichnete sich OpenGL durch vollständige Abwärtskompatibilität aus. Die GL bestand in der Version 3.0 aus etwa 670 Kommandos, von denen viele redundant waren. Beispielsweise existierte die feste Pipeline weiterhin neben der Programmierbaren, mit der sich Erstere problemlos emulieren lässt. Ferner waren einzelne Funktionen sehr viel weniger effizient implementierbar als andere, die ein identisches Ergebnis lieferten. Infolgedessen wurden in der OpenGL Version 3.0 viele dieser redundanten Funktionen als deprecated ausgewiesen, um sie in zukünftigen Versionen entfernen zu können. Die im Mai 2009 veröffentlichte Version 3.1 entfernte die meisten der als deprecated markierten Funktionen aus dem OpenGL Kern und überführte sie in die neue ARB_compatibility Extension. Damit sind diese Funktionen nicht mehr zwingend Teil einer OpenGL Implementation und können, selbst wenn vorhanden, nicht

mit neuen Funktionen zusammen genutzt werden. Der Erhalt des mit OpenGL 1.x / 2.x kompatiblen Kontexts ist vor allem durch die CAD-Branche begründet, welche größtenteils weiterhin die bestehende Codebasis verwenden möchte. Ende 2009 folgte die OpenGL Version 3.2, welche unter anderem den bereits aus DirectX-10 bekannten Geometry Shader zur programmierbaren Verarbeitung von Primitives als Teil des Kerns einführte. Diese im Vergleich zu 3.0 deutlich schlankere API besteht aus ca. 250 Befehlen.

- **4.0**, ab 2010
  Im März 2010 wurden die Spezifikationen zu OpenGL 4.0 sowie zu GLSL Version 4.0 veröffentlicht. In OpenGL 4.0 hält die aus DirectX 11 bekannte Tessellation zur dynamischen Verfeinerung der Geometrie auf der GPU Einzug. Dazu werden zwei neue Shader, Tessellation Control und Tessellation Evaluation, sowie eine feste Pipeline Stage, der Primitive Generator, eingeführt. Weiterhin wurde die Möglichkeit zur Zusammenarbeit mit OpenCL, u.a. durch Einführung von 64 Bit Datentypen in GLSL, deutlich ausgebaut. Außerdem können mithilfe des Per-Sample-Fragment-Shaders flexiblere Anti Aliasing Techniken implementiert werden.

Bei der Entwicklung der API hat die Implementierbarkeit auf der zum jeweiligen Zeitpunkt aktuellen Grafikhardware immer eine zentrale Rolle gespielt. Anfangs wurden lediglich der Rasterizer und einige der anschließenden Operationen hardwarebeschleunigt (etwa Voodoo Graphics, 1996), später konnten zumindest teurere Grafikkarten auch die Transformation und Beleuchtung (T&L) der festen Pipeline übernehmen (etwa GeForce 256, 1999). Die später eingeführten Shaderprogramme wurden zunächst auf dedizierten Vertex- und Fragmentprozessoren ausgeführt (etwa GeForce 6800, 2004). Dies führte jedoch oft zu ungleichmäßiger Auslastung der Vertex- und Fragment- Prozessoren. Aus diesem Grund wurde das Unified Shader Modell eingeführt, bei dem Vertex-, Geometry- und Fragment Programme über praktisch identische Fähigkeiten verfügen und daher von der gleichen Hardware ausgeführt werden können. Verwendet wird dazu heute in der Regel an die bekannte SIMD-Architektur angelehnte Hardware (etwa GeForce 8800, 2006). Jene ist weiterhin sehr gut für parallele Computing Languages wie das 2008 ebenfalls von der Khronos Group veröffentlichte OpenCL geeignet. Dadurch sind prinzipiell beliebige Algorithmen auf dem Grafikprozessor (GPU) ausführbar und zudem ist eine direkte Zusammenarbeit mit OpenGL ist ebenfalls möglich. Zusammenfassend lässt sich ein Trend ausgehend von fester Hardware über dedizierte programmierbare Hardware hin zu Software, welche in massiv parallelen Umgebungen ausführbar ist, feststellen.

21.4 Spracheigenschaften und Syntax

OpenGL kann als Client Server Modell beschrieben werden. Eine Applikation (der Client) setzt OpenGL Befehle ab und die GL Implementation (der Server) führt diese aus. Typischerweise befinden sich Client und Server auf demselben Rechner, es besteht aber auch die Möglichkeit beide auf zwei Computer aufzuteilen und über Netzwerk miteinander kommunizieren zu lassen. OpenGL ist eine Zustandsmaschine, d.h. einmal gesetzte State Variablen, etwa die Hintergrundfarbe, behalten bis zu ihrem Widerruf ihre Gültigkeit. Auf diese Weise wird die Kommunikation zwischen Client und Server minimiert. Die Gesamtheit aller State Variablen zusammen mit den aktivierten Shadern legt genau fest, wie Primitives in den Framebuffer gerendert werden.

Neben der OpenGL Kern Bibliothek, deren Befehle mit **gl** beginnen, existiert eine Reihe weiterer Hilfsbibliotheken, von denen zwei sehr verbreitete hier kurz beschrieben werden:

- **OpenGL Utility Library (GLU)**
  Befehle dieser Bibliothek beginnen mit **glu** und stellen eine abstraktere Schicht über den Kommandos der OpenGL Library dar, um komplexere Operationen auszuführen. Darunter fallen unter anderem das vereinfachte Manipulieren von Projektionsmatrizen und das Modellieren von NURBS. Das OpenGL Utility Toolkit, dessen aktuelle Version 1.3 aus dem Jahre 1998 stammt, ist üblicherweise ebenfalls im Standard OpenGL Paket enthalten. Vorsicht: Auch wenn bislang keine der glu Befehle entfernt oder als deprecated markiert wurden, so basieren viele trotzdem auf Funktionen, welche ab der OpenGL Version 3.1 nicht mehr Teil des Kerns sind. Das Verhalten großer Teile dieser Bibliothek ist somit unklar.

- **OpenGL Utility Toolkit (GLUT)**
  OpenGL verfügt über keine Möglichkeit, die gerenderte Ausgabe an das jeweilige Fenstersystem anzubinden und kann keine Benutzereingaben verarbeiten. Diese Aufgaben kann für viele Plattformen das OpenGL Utility Toolkit übernehmen. Aufgrund dessen Einfachheit und eingeschränkter Funktionalität wird es primär in kleinen Demos zum Lernen von OpenGL eingesetzt. Das OpenGL Utility Toolkit gehört allerdings nicht zum Standard OpenGL Paket und wird seit einiger Zeit nicht mehr weiterentwickelt. Inzwischen gibt es unabhängige Hilfsbibliotheken mit gleicher Zielsetzung wie freeglut oder OpenGLUT. In dieser Veranstaltung wird GLUT nicht benötigt, da Java bzw. JOGL die Ein- und Ausgabe übernehmen.

Um die Portierung des OpenGL Codes zwischen verschiedenen Plattformen zu erleichtern, definiert OpenGL eigene Datentypen. Diese entsprechen gewöhnlichen C/C++ Datentypen, welche stattdessen verwendet werden können. Die folgende Tabelle führt die einzelnen Datentypen auf. Darin legt der ersten Spalte genannte Suffix als letzter Buchstabe eines Kommandos den erwarteten Datentyp an.

Suffix	Datentyp	C-Korrespondenz	OpenGl Name
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int	GLint
f	32-bit floating point	float	GLfloat, GLclampf
d	64-bit floating point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int	GLuint, GLenum, GLbitfield

Die Mehrheit der OpenGL Funktionen folgt einer Namenskonvention, deren erster Teil die Bibliothek angibt, aus der dieser Befehl stammt, gefolgt vom Namen der GL Wurzel Funktion. OpenGL kennt kein Überladen der Funktionen. Daher werden bei Kommandos mit gleicher Funktionalität aber anderen Parametern Befehlssuffixe hinzugefügt, welche diese Argumente festlegen. Die Form der OpenGL Befehle ist:

```
<Library prefix><Root command>[arg count][arg type][vektoriel]
```

Dabei kennzeichnen eckige Klammern optionale Suffixe. Ein Beispiel:

```
glUniform4fv(...)
```

Damit wird der Wurzelbefehl **Uniform** aus der Bibliothek **gl** aufgerufen und ihm 4 Float Werte in vektorieller Form (etwa als C-Pointer) übergeben. Eine weitere Spezialisierung vieler GL Funktionen ist durch die übergebenen GL Konstanten möglich. Beispielsweise hat der Befehl **glEnable(**`int cap`) in Abhängigkeit des übergebenen Parameters `cap` sehr unterschiedliche Auswirkungen auf den GL State. Ein Auszug:

GL_DEPTH_TEST	Aktiviert eine Vorschrift, gemäß der nur das vorderste oder das hinterste Fragment Einfluss auf den Framebuffer hat
GL_CULL_FACE	Aktiviert die drehsinnabhängige Entfernung der Vorder- und/oder Rückseite der Dreiecke (Culling)
GL_PRIMITIVE_RESTART	Aktiviert eine spezielle Indizierungstechnik für Primitives
GL_VERTEX_PROGRAM_POINT_SIZE	Ermöglicht dem Vertex Shader das Verändern der Größe von Point Primitives

Weitere Beispiele für OpenGL Funktionen:

```
// Setzen der Hintergrundfarbe
void glClearColor(float red, float green, float blue, float alpha);

// Setzen von Position und Größe eines rechteckigen Bereichs im Fenster,
// in den die Ausgabe gerendert wird
void glViewport(int x, int y, int width, int height);
```

21.5 JOGL und Codebeispiele

Wie bereits erwähnt, werden in dieser Veranstaltung native OpenGL Befehle aus Java Applikationen über die Java OpenGL Anbindung JOGL ausgeführt. JOGL ist verfügbar unter: `http://kenai.com/projects/jogl/pages/Home`. Zusammenfassend werden an dieser Stelle die für die Codebeispiele relevanten Unterschiede und Gemeinsamkeiten zu einer C-Anbindung aufgezählt:

- Namen und Bedeutung der OpenGL Funktionen und Konstanten sind identisch
- Ein- und Ausgabe erfolgt mithilfe von Java- und JOGL-Befehlen
- Da in Java keine Pointer existieren, werden in JOGL stattdessen vor allem von `java.nio.Buffer` abgeleitete Klassen für die Parameterübergabe verwendet
- Anstelle der GL Datentypen werden die entsprechenden aus Java eingesetzt
- Das JOGL Interface `GL3` stellt eine zum OpenGL 3.1+ Kontext kompatible Schnittstelle bereit und enthält alle Konstanten
- Der Zugriff auf OpenGL Befehle wird über die Methoden einer Instanz einer `GL3` implementierenden Klasse ermöglicht. Dieses Java Objekt hat in den folgenden Codebeispielen den Namen `gl`

Beispiel: Aus dem OpenGL Code einer C-Anbindung

```
glEnable(GL_DEPTH_TEST);
```

wird hier in JOGL:

```
gl.glEnable(GL3.GL_DEPTH_TEST);
```

Dementsprechend einfach ist die Portierung des OpenGL Codes aus einer C-Applikation und umgekehrt. Die in den folgenden Kapiteln enthaltenen Codebeispiele sind Auszüge eines größeren, ausführbaren OpenGL 3.1 Programms, welches auf der Webseite zu dieser Veranstaltung erreichbar ist. Das Programm kann eine mit einer Farbtextur versehene Geometrie rendern, wie in Abbildung 21.1 zu sehen. Da es sich um ein durchgehendes Beispiel handelt, werden gelegentlich Variablen vorheriger

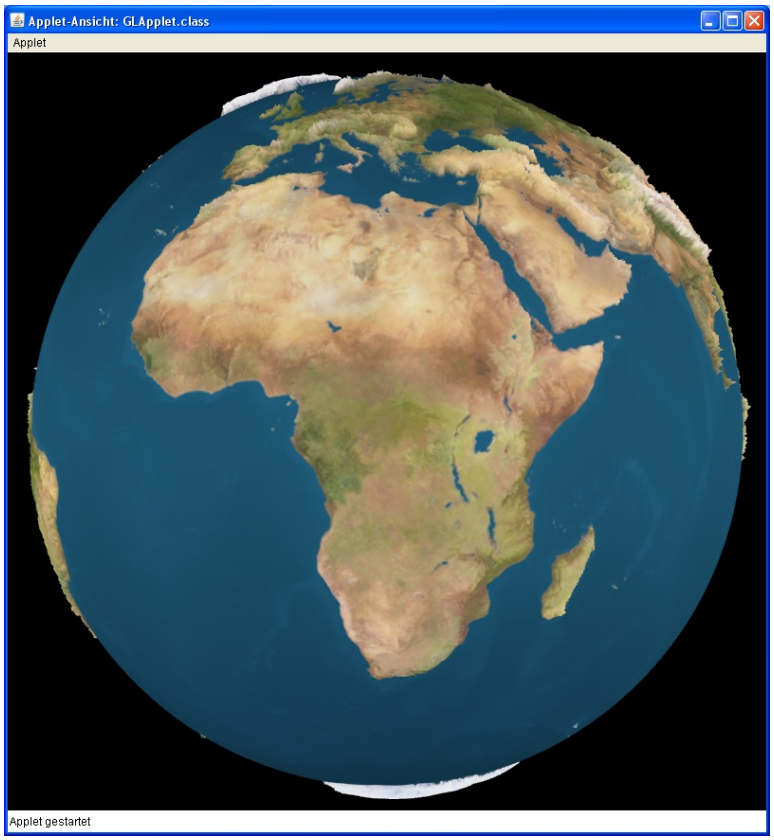


Abbildung 21.1: Darstellung einer Geometrie mit Farbtextur

Listings wiederverwendet. Es handelt sich bei den Beispielen um OpenGL (via JOGL) und GLSL Code. Dabei werden die jeweiligen Schlüsselbegriffe hervorgehoben und zur besseren Unterscheidung wird der OpenGL Code wie

```
gl.glBufferData(target, size, data, usage);
```

durch die Hintergrundfarbe deutlich von GLSL Code wie

```
uniform sampler2D colorTexture;
```

abgegrenzt.

21.6 Arten von Informationen

21.6.1 Vertices

Bei einem Vertex handelt es sich um einen mathematischen Punkt im Raum, der oft Eckpunkt einer geometrischen Figur ist. Neben der geometrischen Information, hinterlegt in der Position, enthält er häufig weitere Eigenschaften der Oberfläche in diesem Punkt. Verbreitete Beispiele sind Farbinformationen, Normalen und Texturkoordinaten aber auch andere Eigenschaften wie Materialdichte, Beschleunigung und Geschwindigkeit können zu den Vertex Attributen zählen. Vertexdaten können in OpenGL in Form von serverseitigen Datenstrukturen, den Buffer Objects, hinterlegt werden. Dabei handelt es sich um eine eindimensionale Folge der Attribute aller zu rendernder Vertices.
Sei `bufferNames` ein Java Objekt des Typs `IntBuffer`, so lautet der Code zum Erzeugen zweier (leerer) Buffer Objects mit **glGenBuffers**:

```
// Anzahl der zu erzeugenden Buffer Objects
int n = 2;

// Java Buffer, in dem die Indices der Buffer Objects hinterlegt werden
Java.nio.IntBuffer buffers = bufferNames;

gl.glGenBuffers(n, buffers);
```

Anschließend kann das zu einer Id gehörende Buffer Object zur Verwendung für Per Vertex Daten mit der Funktion **glBindBuffer** initialisiert werden:

```
int coordBuffer = bufferNames.get(0);
int buffer = coordBuffer;

// Gibt an, ob Vertex- oder Indexdaten enthalten sind. Hier: Vertexdaten
int target = GL3.GL_ARRAY_BUFFER;

gl.glBindBuffer(target, buffer);
```

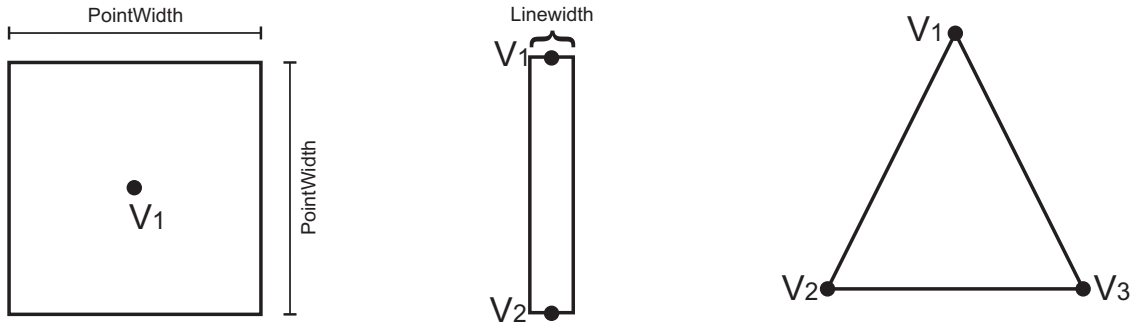
Dabei gibt die Konstante `GL_ARRAY_BUFFER` an, dass das Buffer Object Vertex Daten enthalten soll. Neben der Initialisierung, welche nur beim ersten Aufruf mit einer Vertex Buffer Id erfolgt, aktiviert der Befehl **glBindBuffer** das Buffer Object. Nachfolgende Befehle beziehen sich dann auf dieses Objekt. Anschließend kann der benötigte Speicher angefordert und optional mit Werten initialisiert werden. Vertex Daten, wie das Java `FloatBuffer` Objekt `vertexCoords`, können durch den Befehl **glBufferData** hinzugefügt werden:

```
// Benötigter Speicher in Byte
int size = vertexCoords.capacity * 4;

// Erwartete Art der Benutzung.
// Passende Wahl führt eventuell zu besserer Performance
int usage =  GL3.GL_STATIC_DRAW;

// Die Daten. Hier: FloatBuffer mit den Positionen aller Vertices.
Buffer data = vertexCoords;

target = GL3.GL_ARRAY_BUFFER;
gl.glBufferData(target, size, data, usage);
```

Hinweis: Mit allen anderen Vertex Daten, wie etwa Texturkoordinaten, wird analog verfahren.

21.6.2 Primitives

Die beschriebenen Vertices haben keinerlei Ausdehnung und können demnach nicht angezeigt werden. In OpenGL existieren drei Arten von elementaren geometrischen Grundformen, genannt Primitives, welche Gruppen aus ein bis drei Vertices eine räumliche Ausdehnung zuweisen können. Die erste sind Punkte, die über einen Vertex mit einer zusätzlichen Breite definiert werden. Die Breite wird in Pixeln angegeben und veranlasst den Rasterizer, für eine Breite · Breite große Fläche mit dem Vertex im Mittelpunkt Fragments zu erzeugen (siehe Abb. 21.2(a)).



(a) Ein Punkt, definiert über Mittelpunkt und Seitenlänge

(b) Eine Linie, definiert über Endpunkte und Breite

(c) Ein Dreieck, definiert über drei Eckpunkte

Abbildung 21.2: Primitives in OpenGL

Weiterhin gibt es mit Linien eindimensionale Primitives, die über zwei durch eine Gerade verbundene Vertices und einer zusätzlichen Linienbreite in Pixeln definiert werden (siehe Abb. 21.2(b)). Am häufigsten Verwendung finden jedoch zweidimensionale Primitives, in OpenGL ausschließlich Dreiecke. Sie werden durch drei Vertices repräsentiert und der Rasterizer erzeugt typischerweise Fragments für die gesamte Dreiecksfläche (siehe Abb. 21.2(c)).
Mit Ausnahme der Punkte benötigen alle Primitives zusätzlich zu den geometrischen auch topologische Informationen. Diese wird zum einen durch eine Sequenz von Indices definiert, wodurch die Reihenfolge, in der die Vertices zu Primitives zusammenzusetzen sind, festgelegt ist. Zum anderen können zum Indizieren von Linien und Dreiecken verschiedene Schemata eingesetzt werden. Einige sind für sechs Vertices und das Index Array {0, 1, 2, 3, 4, 5} in Abbildung 21.3 zu sehen.

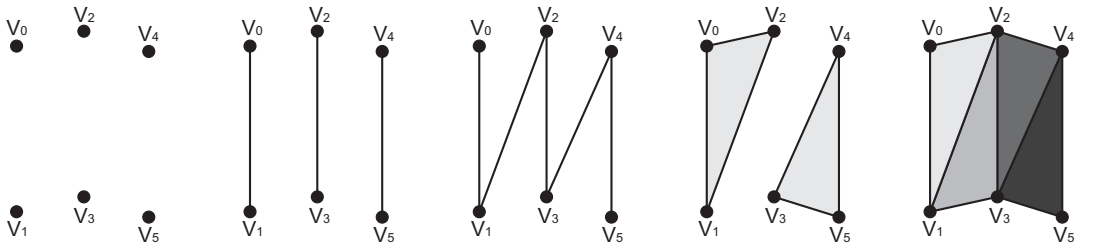


Abbildung 21.3: Primitive Indizierung

Im Einzelnen werden für diese Indexfolge die folgenden Primitives (in Klammern) erzeugt:

- `GL_POINTS`
  Ein Punkt besteht aus genau einem Vertex: (0), (1), (2), (3), (4), (5)
- `GL_LINES`
  Je zwei Vertices definieren eine Linie: (0, 1), (2, 3), (4, 5)
- `GL_LINE_STRIP`
  Eine Linienfolge wird definiert durch mindestens zwei Vertices, die der Reihe nach mit Linien verbunden werden: (0, 1), (1, 2), (2, 3), (3, 4), (4, 5)
- `GL_TRIANGLES`
  Je drei Vertices definieren ein Dreieck: (0, 1, 2), (3, 4, 5)
- `GL_TRIANGLE_STRIP`
  Eine Dreieckfolge wird definiert durch mindestens drei Vertices, die der Reihe nach zu Dreiecken verbunden werden: (0, 1, 2), (2, 1, 3), (3, 1, 4), (4, 1, 5)

Weil z. B. im Inneren geschlossener Flächen jeder Vertex Teil etlicher Primitives ist, können diese einfach durch Mehrfachindizierung verschiedenen Primitives zugewiesen werden. Veranschaulicht wird das Vorgehen anhand des in Abbildung 21.4 dargestellten Dreiecknetzes. Dort wird mithilfe zweier Index Arrays eine zusammenhängende Fläche definiert, ohne einen Vertex doppelt setzen zu müssen. Insbesondere ist der Vertex V4 Teil von sechs Dreiecken.

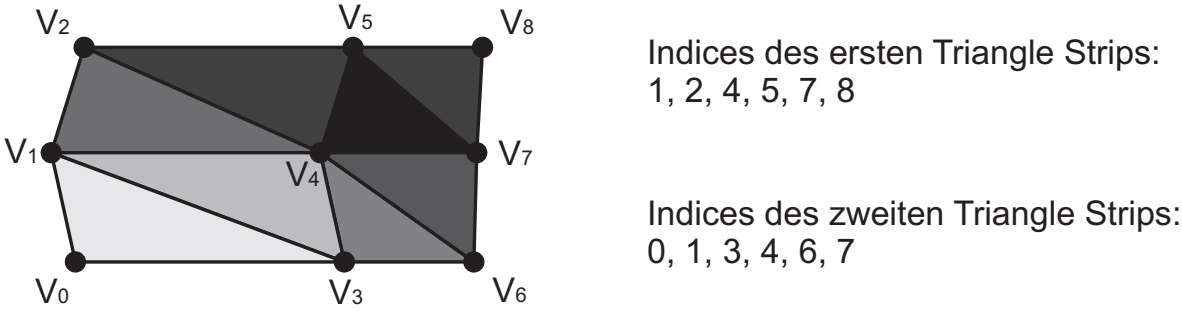


Abbildung 21.4: Ein Mesh bestehend aus zwei Triangle Strips

Die Indices werden ebenfalls in einem Buffer Object hinterlegt, dessen Erzeugung ähnlich wie bei den Vertex Daten abläuft. Sei `indices` ein Java `IntBuffer` mit der Folge der Indices. Dann lautet der Code zum Erzeugen des Index Buffer Objects:

```
int indexBuffer = bufferNames.get(1);
buffer = indexBuffer;

// Gibt an, ob Vertex- oder Indexdaten enthalten sind. Hier: Indexdaten
target = GL3.GL_ELEMENT_ARRAY_BUFFER;

// Benötigter Speicher in Byte
size = indices.capacity * 4;

data = indices;
gl.glBindBuffer(target, buffer);
gl.glBufferData(target, size, data, usage);
```

Dabei enthält das Java IntBuffer Objekt `indices` die Indexfolge und die Konstante GL_ELEMENT_ARRAY_BUFFER gibt an, dass das Buffer Object Indices enthält. Ist genau ein Index Array und ein oder mehrere Vertex Attribute Arrays aktiviert und letztere an Variablen des aktiven Shaders angebunden (siehe Abschnitt 21.10), können die Daten mithilfe des Befehls **glDrawElements** durch die Graphics Processing Pipeline verarbeitet werden:

```
// Anzahl der zu rendernden Elemente, hier: alle
int count = indices.capacity();

// Verweis auf das erste Element
int pointer = 0;

// Art des Primitives
int mode = GL3.GL_TRIANGLE_STRIP;

int type = GL3.GL_UNSIGNED_INT; // Datentyp

// Auslösen des Renderns
gl.glDrawElements(mode, count, type, pointer);
```

21.6.3 Globale Daten

In diese Kategorie gehören für die gesamte Geometrie eines Renderaufrufs identische Daten. Dazu gehören oft Projektionsmatrizen und Lichtquellen. Weiterhin zählen Texturen zu dieser Kategorie. Anders als die allgemeine Definition einer Textur, nämlich einer Vorschrift zum Versehen einer Oberfläche mit zusätzlichen Details, handelt es sich in OpenGL bei einer Textur schlicht um eine ein- bis dreidimensionale diskrete Datenstruktur zusammen mit einer Reihe von Funktionen zum i.d.R. kontinuierlichen Zugriff darauf. Die einzelnen Elemente einer Textur wiederum sind ein- bis vierdimensional und werden als Texel bezeichnet. Diese Datenstruktur kann beliebige numerische Informationen enthalten und ist aus allen Shadern heraus lesbar. So kann eine OpenGL Textur, insbesondere im Vertex Shader, auch für andere Aufgaben als zum Einfärben der Oberfläche verwendet werden. Nachfolgend wird ein Weg zum Erstellen einer gewöhnlichen 2D-Textur aus Rasterdaten und ohne Mipmapping beschrieben. Dafür muss zunächst mit dem Befehl **glGenTextures** ein leeres Texture Object erstellt werden:

```
// Anzahl der zu erzeugenden Texture Objects
int n = 1;

// Java Buffer, in dem die Indices der Texture Objects hinterlegt werden
IntBuffer textures = texNames;

gl.glGenTextures(n, textures);
```

Außerdem muss mit **glActiveTexture** eine der Textureinheiten, deren Anzahl implementationsabhängig ist, aktiviert werden:

```
// ID einer, hier der ersten, Textureinheit
int texUnit = GL3.GL_TEXTURE0;

gl.glActiveTexture(texUnit);
```

Nachfolgende Texturbefehle beziehen sich dann auf diese Textureinheit. Anmerkung: Gleichzeitig zu verwendende Texturen müssen verschiedenen Textureinheiten zugewiesen werden.

Anschließend kann das eingangs erzeugte Texture Object als 2D Textur initialisiert, aktiviert und mit **glBindTexture** an die zuvor aktivierte Textureinheit angebunden werden:

```
// Es soll eine 2D-Textur erzeugt werden
int target = GL3.GL_TEXTURE_2D;

// ID unserer Textur
textureName = texNames.get(0);

gl.glBindTexture(target, textureName);
```

Dem nun aktiven zweidimensionalen Texture Object können mithilfe der korrespondierenden Funktion **glTexImage2D** Daten, hier enthalten im Java `ByteBuffer image`, zusammen mit einer Beschreibung derselben übergeben werden:

```
target = GL3.GL_TEXTURE_2D;

// Mipmapping Level; Hier: deaktiviert, sonst 1...Max Level
int level = 0;

// Internes format der Texel
int internalFormat = GL3.GL_RGB;

// Breite der Textur in Texeln
int width = 503,

// Höhe der Textur
int height = 123,

// Zusätzlicher Rand; muss ab OpenGL 3.1 0 sein
int border = 0,

// Format der übergebenen Daten
int format = GL3.GL_RGB;

// Datentyp der übergebenen Daten
int type = GL3.GL_UNSIGNED_BYTE;

// Die Daten
Buffer pixels = image;

gl.glTexImage2D(target, level, internalFormat, width, height, border, format, type, pixels);
```

Soll Mipmapping eingesetzt werden, ist dieser Befehl für jede Mipmap zu wiederholen. Weitere Optionen, wie Interpolationsart, Interpretationsvorschrift für die Texturkoordinaten, etc. können mithilfe der Funktion **glTexParameter*** eingestellt werden.

21.6.4 Fragments

Die bisherigen Daten werden an die GL übergeben und können durch diese verarbeitet werden. Im Gegensatz dazu entstehen Fragments erst im Durchlauf der Graphics Pipeline aus der Geometrie. Der Rasterizer erzeugt aus den in den Bildraum projizierten Primitives für jeden überlappten Pixel eine Datenstruktur, genannt Fragment. Diese erhält neben der diskreten Pixelkoordinate eine Tiefeninformation und die für den Ort des Pixels interpolierten Daten der Vertices des Primitives. Im weiteren Verlauf der Graphics Pipeline (siehe Abschnitt 21.8) kann dem Fragment eine Farbe zugewiesen werden, welche eventuell Einfluss auf die Einfärbung des korrespondierenden Pixels hat. Im beschriebenen Spezialfall kann man sich das Fragment anschaulich als eine zum aktuellen Primitive gehörige Vorstufe zu diesem Pixel vorstellen.

21.7 Grober Ablauf eines Anwendungsbeispiels

In diesem Abschnitt soll anhand eines einfachen Beispiels ein Überblick über den gesamten Prozess von der Modellierung einer Szene bis zu einem gerenderten Bild beschrieben werden.

- **Modellierungssoftware (optional)**
  Komplexere Szenen, etwa in aktuellen Computerspielen, werden üblicherweise von Künstlern mithilfe spezieller Modellierungs- und Zeichenwerkzeuge erzeugt. Diese Programme verbergen die zugrundeliegende Computergrafik und performancekritische Details vor dem üblicherweise nicht in dieser Richtung ausgebildeten Künstler. Die auf diese Weise erzeugten 3D-Modelle, Texturen etc. sind i.d.R. bei weitem zu detailliert, als das sie direkt in der Praxis eingesetzt werden könnten. Deswegen werden die Modelle anschließend auf Detailstufen heruntergerechnet, welche den Anforderungen der einzelnen Zielplattformen entsprechen. Weiterhin können so auch für eine einzelne Plattform verschiedene Detailstufen (LOD) erzeugt werden, um beispielsweise für weiter vom Betrachter entfernte Figuren weniger Rechenleistung aufwenden zu müssen. Da dieser Teil nicht für das Laufzeitverhalten der Zielapplikation relevant ist, können hier sehr aufwändige Verfahren eingesetzt werden. Zuletzt werden die erzeugten Modelle in einem für die Zielapplikation lesbaren Format exportiert.

- **Applikation**
  Zunächst müssen die zuvor modellierten statischen Daten wie etwa Landschaften, 3D-Figuren-Modelle sowie zugehörige Texturen geladen und initial in der Szene angeordnet werden. Zur Laufzeit werden bewegliche Objekte ihre Position und i.d.R. auch geometrische Eigenschaften ändern (etwa Beine etc.). Die Aufstellung der zugehörigen Matrizen für diese dynamische Modellierung ist Aufgabe der Applikation. Weiterhin muss das Frustum samt zugehöriger Matrizen für die Betrachtungstransformation aufgestellt werden. Dieser Teil der Applikation, auch als 3D-Engine bezeichnet, testet weiterhin größere Objekte auf Sichtbarkeit, um diese gar nicht erst an den Renderer zu übergeben, legt die aktuelle LOD Stufe der Objekte fest und sortiert zumindest transparente Flächen vor. Außerdem sollte der Render State, etwa die gerade aktiven Texturen oder Shader, möglichst selten geändert werden müssen. Andernfalls müssten ständig andere Daten in den Grafikspeicher geladen und die Processing Pipeline reorganisiert werden. Eine in der Praxis schwierige Aufgabe für die 3D-Engine ist demnach, die Szene in einer Weise zu verwalten, sodass eine sinnvolle Balance aus räumlicher und Render State Kohärenz erreicht wird. Der in diesem Abschnitt beschriebene Teil kann direkt in der jeweiligen Programmiersprache (Java, C, C++, etc.) implementiert sein, alternativ existiert gerade für Standardaufgaben eine große Anzahl von Hilfsbibliotheken (u.a. Szenegraphen).

- **OpenGL**
  Am Ende des vorherigen Schrittes steht fest, welche Teile der Szene in welcher Reihenfolge und mit welchen Einstellungen an den Renderer zu übergeben sind. Diese Teile werden unabhängig voneinander in der zuvor festgelegten Reihenfolge gerendert. An dieser Stelle beginnt erst der Aufgabenbereich von OpenGL, der im Wesentlichen für jeden gleichzeitig zu rendernden Teil der Szene aus zwei Punkten besteht. Zum einen muss der feste Teil der Graphics Processing Pipeline konfiguriert sowie Shader aktiviert werden und zum anderen müssen die für diesen Teil benötigten Daten (Szenengeometrie, Texturen, etc.), aber auch Lichtquellen und Transformationsmatrizen von der Applikation an die GL Implementation übergeben werden (vergl. Abschnitt 21.10).

- **Graphics Processing Pipeline**
  Anschließend durchläuft die Geometrie die Graphics Processing Pipeline (Siehe Abschnitt 21.8). Diese übernimmt üblicherweise u.a. die Projektion der Geometrie, wertet die Beleuchtung aus, färbt die gerasterten Oberflächen ein und schreibt das Ergebnis in den Framebuffer. Dieser Prozess läuft idealerweise vollständig auf der GPU ab und wird durch ein OpenGL Kommando lediglich angestoßen. Dementsprechend findet sich der zugehörige Code, mit Ausnahme der in einer Shading Language, wie GLSL, verfassten Shader, nicht in der Applikation wieder.

Abschließend bleibt anzumerken, dass diese Vorgehensweise zwar nicht untypisch, jedoch keinesfalls zwingend ist. Gerade Multipass Renderer können deutlich davon abweichen und auch sonst kann es etwa sinnvoll sein, einzelne Aufgaben aus der Applikation in die Shader zu verlagern und umgekehrt. Weiterhin können Renderer implementiert werden, welche mit klassischer Rastergrafik praktisch gar nichts gemein haben.

21.8 Graphics Processing Pipeline

Aus Programmierersicht ist OpenGL eine feste Sequenz von Operationen, welche die Ausgangsdaten in ein Bild überführen kann. Diese wird als Graphics Processing Pipeline oder Graphics Pipeline bezeichnet und ist (in vereinfachter Form) in Abb. 21.5 zu erkennen.

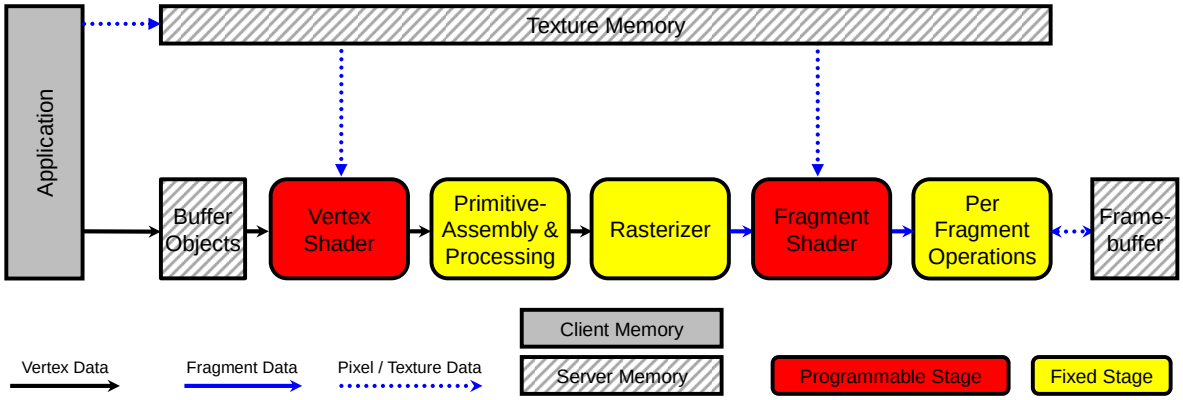


Abbildung 21.5: Vereinfachte Darstellung der Graphics Processing Pipeline in OpenGL

Tatsächlich muss eine GL Implementation lediglich mit dieser Pipeline identische Ergebnisse liefern, sich jedoch nicht im Detail an die Folge der einzelnen Schritte halten. Die wichtigsten Abschnitte sind:

- **Vertex Shader**
  Der Vertex Shader ist ein nahezu beliebiges Programm, welches Per Vertex Daten verarbeitet. Typische Aufgaben, die von ihm übernommen werden können, sind die Transformation von Vertices und Normalen oder auch die Beleuchtung einzelner Vertices (in diesem Skript als Gouraud Shading bezeichnet). Eine Instanz des Vertex Shaders verarbeitet immer nur die zu einem Vertex gehörenden Daten und hat keinerlei Lese- oder Schreibzugriff auf Daten anderer Vertices. Folglich kann eine beliebige Anzahl von Vertices ohne Synchronisationsaufwand parallel verarbeitet werden.

	Eingang	Ausgang
Art	Vertices	Vertices
Verhältnis pro Instanz	1	1

- **Primitive Assembly**
  Nachdem bislang die zu den einzelnen Vertices gehörigen Informationen völlig unabhängig voneinander verarbeitet wurden, werden in diesem Abschnitt mithilfe der topologischen Informationen die jeweiligen Primitives wiederhergestellt und die zugehörigen Daten gesammelt. Dementsprechend verlassen diese Stufe entweder Punkte, Linien oder Dreiecke samt der mit ihnen assoziierten Daten.

	Eingang	Ausgang
Art	Vertices	Primitives
Verhältnis pro Instanz	1, 2, 3	1

- **Primitive Processing**
  In diesem Abschnitt finden mit Clipping und Culling diejenigen Operationen statt, für die Kenntnisse über das gesamte Primitive nötig sind. Anschließend werden die Koordinaten in einem als Perspective Divide bezeichneten Schritt durch die homogene Koordinate geteilt.

	Eingang	Ausgang
Art	Primitives	Primitives
Verhältnis pro Instanz	1	≥ 0

- **Rasterizer**
  Aufgabe des Rasterizers ist die Überführung von Primitives in Fragments. Dazu wird für jedes von einem Primitive überlappte Pixel ein Fragment erzeugt, welches die 2d-Komponente des korrespondierenden Pixels sowie eine zusätzliche Tiefeninformation erhält. Weiterhin werden alle Daten der Eckpunkte des Primitives für diese Koordinate interpoliert und ebenfalls im Fragment gespeichert.

	Eingang	Ausgang
Art	Primitives	Fragments
Verhältnis pro Instanz	1	0 ... Fensterauflösung (i.d.R.)

- **Fragment Shader**
  Der Fragment Shader ist ein nahezu beliebiges Programm, welches die durch den Rasterizer erzeugten Fragments verarbeitet. Dabei werden auf Basis der Per Fragment Daten und globaler Daten wie etwa Texturen Berechnungen durchgeführt, um beispielsweise die Farbe des Fragments festzulegen. Oft finden hier die Texturierung der Oberflächen und die Auswertung eines Beleuchtungsmodells statt (in diesem Skript: Phong Shading). Ebenso wie im Vertex Shader gibt es keinerlei Möglichkeit Ergebnisse zwischen den Fragments auszutauschen, um die Parallelität der Berechnungen nicht einzuschränken.

	Eingang	Ausgang
Art	Fragments	Fragments
Verhältnis pro Instanz	1	0, 1

- **Per Fragment Operations**
  Die Per Fragment Operations regeln im Wesentlichen ob, und auf welche Weise die bereits verarbeiteten Fragments Einfluss auf den Framebuffer haben. Gerade in komplexen Szenen können Pixel von mehreren Primitives überlappt werden, sodass viele Fragments mit einem Pixel korrespondieren. Bei undurchsichtigen Oberflächen kann mithilfe des Tiefentests festgelegt werden, dass sich immer das zuvorderst liegende Fragment durchsetzt und einen ggf. bereits im Framebuffer hinterlegten Wert überschreibt. Alternativ besteht die Möglichkeit, den Wert des jeweils aktuellen Fragments und den bereits im Framebuffer eingetragenen als Eingaben für eine Blending Funktion zu verwenden und das Resultat wieder in den Framebuffer zu schreiben. Auf diese Weise lassen sich beispielsweise Transparenzeffekte realisieren.

	Eingang	Ausgang
Art	Fragments	Pixel
Verhältnis pro Instanz	1	0, 1
Verhältnis pro Pixel	0 ... N	1

21.9 OpenGL Shading Language

Der Begriff Shader wurde erstmals im Jahre 1984 von Cook in dessen Paper „Shade Trees“ eingeführt und meint ein Programm zur Beschreibung von Oberflächeneigenschaften. Im High-Quality-Rendering-Bereich existieren schon lange entsprechende Sprachen, wie die verbreitete und an C angelehnte Renderman Shading Language, an welcher sich auch aktuelle Shading Languages orientieren. Im Bereich der hardwarebeschleunigten 3D APIs, wie OpenGL oder DirectX, versteht man unter einem Shader dagegen ein in einer Shading Language (GLSL, HLSL, Cg, ...) geschriebenes Programm, welches auf einer GPU ausführbar ist. Sie waren nach Einführung von nVidias GeForce 3 im Jahre 2001 erstmalig einsetzbar. Es mussten allerdings zu ihrer Verwendung in OpenGL die entsprechende Extensions bemüht werden. Shader sind, wie der Vertex- und Fragment Shader (siehe 21.9), oft Teil der Graphics Processing Pipeline, werden jedoch nicht zwingend zum Berechnen der Oberflächeneigenschaften oder gar „Schattieren“ eingesetzt. Praktisch nichts mehr mit Cooks Definition gemein hat der Compute Shader, welcher in DirectX für allgemeine Berechnungen auf der GPU zuständig ist. Aufgrund des zu großen Umfangs der Thematik beschränkt sich dieses Skript auf eine exemplarische Behandlung von GLSL mit dem Fokus auf Vertex- und Fragment Shader.

Die Shading Language GLSL (auch: glSlang) ist seit der GL Version 2.0 (2004) Teil des OpenGL Kerns und ihre aktuelle Version ist 1.5 (2009). GLSL ist ebenso wie OpenGL Plattform und Betriebssystemunabhängig. Der GLSL Compiler ist Teil des Display Drivers und übersetzt die Shader erst zur Laufzeit, sodass optimierter Code für die jeweilige Hardware Architektur erzeugt werden kann. GLSL ist eine Hochsprache mit einer an C angelehnten Syntax. Es existieren allerdings eine Reihe Unterschiede, von denen einige auszugsweise im Folgenden erläutert werden. In diesem Skript kann lediglich ein kurzer und sehr unvollständiger Einblick in GLSL gegeben werden.

In GLSL existieren verschiedene Qualifier, welche Variablen vorangestellt sind und vor allem die Schnittstelle der Shader mit der restlichen Graphics Pipeline festlegen. Dies sind:

in Kennzeichnet im Shader lesbare Variablen, deren Wert für jeden Vertex bzw. jedes Fragment verschieden ist. Im Vertex Shader erhalten In-Variablen ihren Wert aus der Applikation und im Fragment Shader handelt es sich dabei um die interpolierten Vertexdaten nach dem Rastern der Primitives.

uniform Kennzeichnet im Shader lesbare globale Variablen, deren Wert für alle Vertices und Fragments während eines Durchlaufs der Graphics Pipeline konstant ist. Uniform-Variablen können ausschließlich durch die Applikation geschrieben werden und sind dann aus allen Shadern heraus in gleicher Weise lesbar.

out Kennzeichnet im Shader schreibbare Variablen, welchen die Ergebnisse der Berechnungen des jeweiligen Shaders zugewiesen werden. Diese Daten werden anschließend durch die restliche Graphics Pipeline verarbeitet.

const Kennzeichnet gewöhnliche Konstanten.

In GLSL existieren folgende skalare Datentypen: `float`, `int`, `uint` und `bool`. Beispiele:

```
float f = 2.5;
bool b = true;
```

Zusätzlich existieren vektorielle Datentypen, welche 2 bis 4 Komponenten, bestehend aus obengenannten Skalaren, besitzen:

vec2, vec3, vec4 Vektordatentypen, bestehend aus 2, 3 und 4 Floats.

ivec2, ivec3, ivec4 Vektordatentypen, bestehend aus 2, 3 und 4 Integers.

uvec2, uvec3, uvec4 Vektordatentypen, bestehend aus 2, 3 und 4 Unsigned Integers.

bvec2, bvec3, bvec4 Vektordatentypen, bestehend aus 2, 3 und 4 Booleans.

Auf die einzelnen Komponenten kann wahlweise über die Namensschemata x,y,z,w oder r,g,b,a oder s,t,p,q zugegriffen werden:

x, r, s Zugriff auf die erste Komponente eines Vektors

y, g, t Zugriff auf die zweite Komponente eines Vektors

z, b, p Zugriff auf die dritte Komponente eines Vektors

w, a, q Zugriff auf die vierte Komponente eines Vektors

Die Operatoren +, -, * und / sind für Vektortypen komponentenweise definiert. Einige Beispiele:

```
vec2 vector1 = vec2(1.0, 2.0);
vec2 vector2 = vec2(0.0, 1.0);
vec2 compWiseMul = vector1 * vector2;
```

Dementsprechend liefert `compWiseMul.x` den Wert 0.0 und `compWiseMul.t` den Wert 2.0. Zusätzlich existieren Matrizen aus Fließkommazahlen bis zu einer Größe von 4x4.

Sampler enthalten die zum Zugriff auf eine Textur benötigte Information und werden einem Shader von der Applikation übergeben. Beispielsweise ermöglicht sampler2D mithilfe der Funktion **texture** den Zugriff auf eine zweidimensionale Textur:

```
uniform sampler2D colorTexture;
vec2 texCoords = vec2(0.0, 0.0);
vec3 color = texture(colorTexture, texCoords);
```

Dabei stellt *texCoords* einen zweidimensionalen Index zum Zugriff auf eine Stelle in der Textur `colorTexture` dar und die Funktion **texture** liefert den entsprechenden Wert. In diesem Fall handelt es sich dabei um einen 3-Vektor der als RGB Tripel interpretiert wird.
Die Flusskontrolle ist ebenfalls an C angelehnt. So ist der Eintrittspunkt in einen Shader die Funktion main. Es existieren Schleifen (for, while, do-while) sowie die Schlüsselwörter break und continue. Verzweigung ist mit if und if-else möglich. Darüber hinaus besteht im Fragment Shader die Möglichkeit mit dem Schlüsselwort discard einen Beitrag des aktuellen Fragments zum Framebuffer zu verhindern. Funktionsaufrufe sind im Wesentlichen an C++ angelehnt, allerdings besteht keine Möglichkeit diese rekursiv aufzurufen. Anders als OpenGL kennt GLSL Überladung, sodass im obigen Beispiel nicht für jeden Texturtyp eine eigene Funktion bereitgestellt werden muss. Weiterhin enthält GLSL eine recht umfangreiche Bibliothek von mathematischen Standartfunktionen für Skalare, Vektoren und Matrizen. Dazu zählen Exponential-, Trigonometrische- und Interpolationsfunktionen, Kreuz- und Skalarprodukt, sowie Funktionen zum transponieren von Matrizen, etc.

21.9.1 Vertex Shader

Der Vertex Shader ist ein Programm, welches die Daten einzelner Vertices unabhängig voneinander verarbeitet. Vertexdaten haben für jeden Vertex einen anderen Wert, typische Beispiele dafür sind etwa die Position, die Normale und Texturkoordinaten. Abbildung 21.6 zeigt die in einen Vertex Shader eingehenden und die durch ihn schreibbaren Daten. Erstere sind im Wesentlichen:

- **User-defined In Variablen**
  Durch den Programmierer zu definierende Variablen, welche Per Vertex Daten enthalten. Auch gängige Vertexeigenschaften wie Position oder Normale müssen selbst definiert werden, da sie nicht zwingend zu einem Vertex gehören. Diese Variablen erhalten ihren Wert durch die Applikation und können im Vertex Shader gelesen werden.

- **User-defined Uniform Variablen**
  Durch den Programmierer zu definierende Variablen, welche globale Daten, etwa Projektionsmatrizen und Lichtquellen, enthalten. Sie erhalten ihren Wert durch die Applikation und können im Vertex- und Fragment Shader gelesen werden.

- **Build-in Uniform Variablen**
  Durch GLSL definierte Variablen, welche globale Daten enthalten.

- **Texture Maps**
  Texturen sind eine spezielle Form der User-defined Uniform Variablen, die ebenfalls globale Daten enthalten. Für einen Vertex wird jedoch i.d.R. nicht alles, sondern beispielsweise über eine Texturkoordinate nur ein bestimmter Wert ausgelesen.

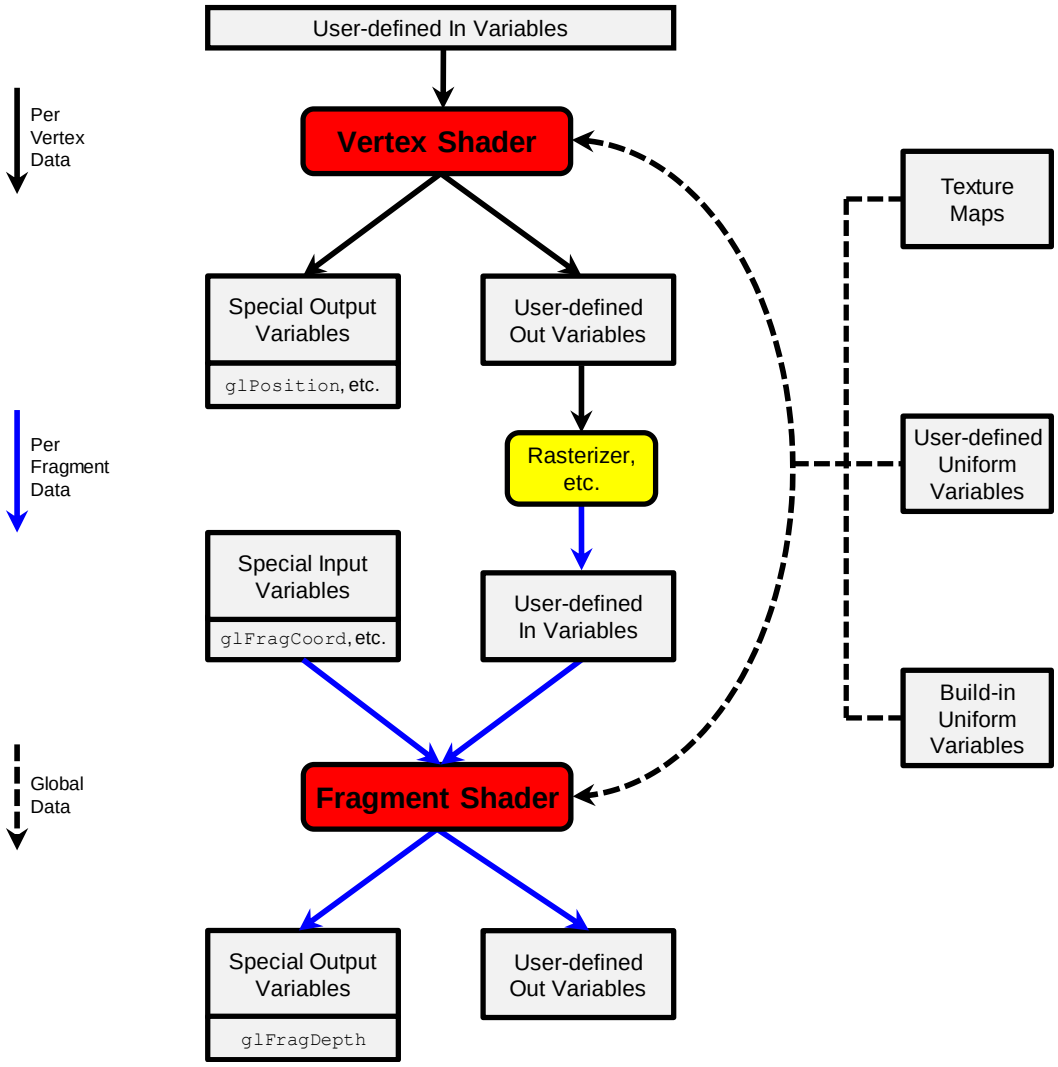


Abbildung 21.6: Datenstrom durch Vertex- und Fragment Shader

Basierend auf den obengenannten Daten kann der Vertex Shader praktisch beliebige Berechnungen durchführen, deren Ergebnisse mithilfe zweier Arten von schreibbaren Variablentypen ausgeben werden können:

- **User-defined Out Variablen**
  Durch den Programmierer definierte Variablen, mit deren Hilfe der Vertex Shader praktisch beliebige Informationen zur weiteren Verarbeitung durch den Fragment Shader ausgeben kann. Denkbare Beispiele sind: Transformierte Normalen oder Texturkoordinaten, Farbwerte, Krümmung, Geschwindigkeit, Gewicht, Temperatur, Druck, etc.

- **Special Output Variablen**
  Durch OpenGL definierte Variablen, deren Werte ggf. für Teile der auf den Vertex Shader folgenden Festen Pipeline benötigt werden. Ein Beispiel ist **gl_Position**, welche mit der in Clipping Koordinaten transformierten Vertexposition belegt werden muss, um eine sinnvolle Ausführung des Rasterizers zu ermöglichen.

Das folgende Listing zeigt einen sehr einfachen Vertex Shader zum Projizieren von Geometrie:

```
// GLSL Version 1.40
#version 140

// User-defined Uniform Variable des Typs 4x4-Matrix zum Transformieren
// der Vertexposition aus Object Coordinates in Clip Coordinates.
uniform mat4 mvpMatrix;

// User-defined In Variable des Typs 4-Vektor.
// Enthält Koordinaten des jeweiligen Vertex.
in vec4 myPosition;

// Texturkoordinaten
in vec2 myTexCoords;

// User-defined Out Variable für Texturkoordinaten
out vec2 texCoords;

void main() {

 // Texturkoordinaten werden (hier) einfach weitergereicht.
 texCoords = myTexCoords;

 // Transformieren der Koordinaten durch
 // Multiplikation mit der Transformationsmatrix
 vec4 vPositionCC =  mvpMatrix * myPosition;

 // Weise das Ergebnis der Special Output Variable gl_Position zu.
 gl_Position = vPositionCC;

}
```

21.9.2 Fragment Shader

Der Fragment Shader ist ein Programm, welches die Daten einzelner Fragments unabhängig voneinander verarbeitet. Abbildung 21.6 zeigt die in einen Fragment Shader eingehenden und die durch ihn schreibbaren Daten. Erstere sind im Wesentlichen:

- **User-defined In Variables**
  Hierbei handelt es sich um die durch den Rasterizer für das jeweilige Fragment interpolierten User-defined Out Variablen des Vertex Shaders. Ihr Name muss mit diesen übereinstimmen.

- **Special Input Variables**
  Durch GLSL definierte Variablen, welche Per Fragment Daten enthalten. Ein Beispiel ist **gl_FragCoord**, die Position des Fragments in Window Coordinates.

Zusätzlich existieren auch im Fragment Shader die Uniform Variablen und Texture Maps, welche sich identisch zu denen des Vertex Shaders verhalten. Basierend auf den obengenannten Daten kann der Fragment Shader praktisch beliebige Berechnungen durchführen, deren Ergebnisse mithilfe zweier Arten von schreibbaren Variablentypen ausgegeben werden können:

- **User-defined Out Variables**
  Der Programmierer kann für einen Fragment Shader eine Reihe von Ausgabevariablen definieren, welche in verschiedene Buffer geschrieben werden. Oft ist das Ergebnis der Berechnungen

des Fragment Shaders eine Farbe, welche eventuell Einfluss auf die Einfärbung des mit dem jeweiligen Fragment korrespondierenden Pixels hat.

- **Special Output Variables**
  Hierbei handelt es sich ausschließlich um die durch GLSL definierte Variable **gl_FragDepth**, welche die Tiefe des jeweiligen Fragments angibt. Wird sie nicht geschrieben, findet implizit die z-Komponente von **gl_FragCoord** Verwendung.

Das folgende Listing zeigt einen sehr einfachen Fragment Shader, der mit dem obigen Vertex Shader kombinierbar ist und eine Oberfläche gemäß einer Farbtextur einfärbt:

```
// GLSL Version 1.40
#version 140

// Handle zum Zugriff auf eine 2D-Textur, welche Farbinformationen
// enthält (hier: RGBA)
uniform sampler2D colors;

// Texturkoordinaten des jeweiligen Fragments
in vec2 texCoords;

// User-defined Out Variable zur Ausgabe der Farbe des Fragments
out vec4 myFragColor;

void main() {

 // Hole den RGBA Farbwert für die Texturkoordinaten des Fragments
 // aus der Textur und gib diesen als Ergebnis aus
 myFragColor = texture(colors, texCoords);

}
```

21.10 OpenGL Shading Language API

Unter der Bezeichnung OpenGL Shading Language API versteht man diejenige Teilmenge der OpenGL Funktionen, die das Erzeugen, Übersetzen, Linken und Aktivieren von Shadern übernehmen sowie diese mit Daten versorgen. Es folgt ein Überblick über einige zum Erzeugen und Benutzen von Shadern benötigte GL Funktionen.

Erzeugen eines Shaderprogramms

Zunächst muss ein Shader Programm, bestehend aus mindestens einem Vertex Shader sowie optional einem Fragment Shader, erstellt werden. Im folgenden Codebeispiel werden je ein leeres Vertex- und ein Fragment Shader Object mit **glCreateShader** erzeugt:

```
int myVs = gl.glCreateShader(GL3.GL_VERTEX_SHADER);
int myFs = gl.glCreateShader(GL3.GL_FRAGMENT_SHADER);
```

Dabei legt die übergebene Konstante die Art des zu erzeugenden Shaders fest und die Rückgabe ist ein Index zum weiteren Zugriff auf das serverseitig vorliegende Shader Objekt.

Im Anschluss daran wird den Shader Objekten der Sourcecode in Form eines Stringarrays mit der Funktion **glShaderSource** übergeben:

```
// Anzahl der Strings im Array
int count = 1;

// Längen der Strings im Array. In Java nicht unbedingt benötigt.
IntBuffer length = null;

// Id des Shaders
int shader = myVS;

// Der zugehörige Quellcode
String[] string =  {" Der VS-Quellcode (S. 250)... "};

// Erzeugen des Vertex Shaders
gl.glShaderSource(myVs, count, vsSource, length);

// Analog für den Fragment Shader...
shader = myFS;
string = {" Der FS-Quellcode (S. 251)... "};
gl.glShaderSource(shader, count, string, length);
```

In Java kann der Quellcode als einzelner String übergeben werden, dementsprechend genügt ein Array der Länge eins und weitere Angaben über die Längen der Arrays entfallen. Anschließend müssen die Shader mit **glCompileShader** übersetzt werden:

```
gl.glCompileShader(myVs);
gl.glCompileShader(myFs);
```

Weiterhin muss mit **glCreateProgram** ein (leeres) Program Object erzeugt werden:

```
int myShaderProgram = gl.glCreateProgram();
```

Diesem Program Object können dann mittels **glAttachShader** die zuvor kompilierten Shader Objects hinzugefügt werden:

```
int program = myShaderProgram;

// Füge den Vertex Shader ...
shader = myVS;
gl.glAttachShader(program, shader);

// und den Fragment Shader zum Shader Program hinzu
shader = myFS;
gl.glAttachShader(program, shader);
```

Zuletzt muss das Program Object noch mit **glLinkProgramm** gelinkt werden:

```
gl.glLinkProgram(myShaderProgram);
```

Falls beim Kompilieren der Shader Objects oder beim Linken des Shader Programs kein Fehler aufgetreten ist, kann das nun ausführbare Programm mit **glUseProgram** als Teil des aktuellen GL State gesetzt werden:

```
gl.glUseProgram(myShaderProgram);
```

Steuern des Datenflusses

Für das zuvor erzeugte Shader Object muss weiterhin der Datenfluss in dieses hinein und aus ihm heraus geregelt werden. Einige Varianten werden nachfolgend beschrieben.

In Variablen

Enthält ein Vertex Shader In-Variablen, welche die Per Vertex Attribute repräsentieren, muss die Applikation diesen Daten zuweisen. Hierzu müssen zunächst, wie in Abschnitt 21.6.1 beschrieben, die zugehörigen Buffer Objects erzeugt und das jeweils anzubindende aktiviert werden. Im Fall obiger Vertex Koordinaten somit:

```
gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, coordBuffer);
```

Ferner muss die Adresse der korrespondierenden Variable des Shaders abgefragt werden. Sei, wie im Vertex Shader aus Abschnitt 21.9.1, eine User-defined In-Variable `myPosition` deklariert als:

```
in vec4 myPosition;
```

Dann kann zunächst ihr Index mit **glGetAttribLocation** erfragt werden:

```
// Id des Program Objects
int program = myShaderProgram;

// Name der Variable des Vertex Shaders des Program Objects
String name = "myPosition";

int location = gl.glGetAttribLocation(program, name);
```

Anschließend kann das mit dieser Position zu assoziierende aktive Array mit Vertex Attributen aktiviert werden über:

```
int index = location;
gl.glEnableVertexAttribArray(index);
```

Zuletzt muss mit dem Befehl **glVertexAttribPointer** spezifiziert werden, wie die Daten im Buffer hinterlegt sind:

```
// Anzahl der Komponenten eines Vertex Attributs
int size = 3;

int type = GL3.GL_FLOAT; // Datentyp

// Normalisieren der Daten erforderlich?
boolean normalized = false;

//Abstand in Bytes zwischen konsekutiven Vertex Attributen
int stride = 0;

// Verweis auf das erste Element
int pointer = 0;

gl.glVertexAttribPointer(location, size, type, normalized, stride, pointer);
```

Uniform variablen

Enthält ein Shader Program Uniform Variablen, kann deren Position mit **glGetUniformLocation** erfragt und anschließend ihr Wert mit **glUniform*** gesetzt werden. Enthalte ein oder mehrere Shader eines Program Objects `myShaderProgram` eine uniform Variable `mvpMatrix`, welche folgendermaßen deklariert ist:

```
uniform mat4 mvpMatrix ;
```

so kann deren Adresse bestimmt werden mit:

```
int location = glGetUniformLocation(myShaderProgram, "mvpMatrix");
```

Anschließend können der Variable Daten, enthalten im Java `FloatBuffer mvpMatrix`, zugewiesen werden. Dies geschieht im vorliegenden Spezialfall einer 4x4 Matrix mittels **glUniformMatrix4fv**:

```
// Nur eine Matrix und kein Array
int count = 1;

// GL_TRUE: Werte in row major order angegeben
// GL_FALSE: Werte in column major order angegeben
boolean transpose = GL_FALSE;

// Die von der Applikation aufgestellte Transformationsmatrix
FloatBuffer value = mvpMatrix;

glUniformMatrix4fv(location, count, transpose, value)
```

Texture Maps sind spezielle Uniform Variablen, die zunächst Erzeugt und Aktiviert werden müssen (siehe Abschnitt 21.6.3). Der weitere Verlauf ist dem zuvor Beschriebenen recht ähnlich und beginnt ebenfalls mit dem Abfragen der Adresse der im Fragment Shader aus Abschnitt 21.9.2 über

```
uniform sampler2D colors;
```

deklarierten Textur, hier mit allerdings mit **glGetUniformLocation**:

```
location = glGetUniformLocation(myShaderProgram, "colors");
```

Dieser wird nun der Textur zugeordneten Textureinheit übergeben. In diesem Fall demnach:

```
// Der zu übergebende Wert. Hier die Id der Textureinheit
int x = texUnit;

gl.glUniform1i(location, x);
```

Out Variablen

Weiterhin können Out Variablen des Fragment Shaders an die einzelnen Render Buffer angebunden werden. Da lediglich der Buffer mit dem Index 0 angezeigt wird, müssen Farbwerte, welche nach dem aktuellen Rendervorgang sichtbar sein sollen, daran angebunden werden. Sei im Fragment Shader eine Out Variable `myFragColor` des Program Objekts `myShaderProgram` deklariert als:

```
out vec4 myFragColor;
```

dann kann sie vor dem Linken des Program Objects durch den Befehl **glBindFragDataLocation** an den anzeigbaren Buffer gebunden werden:

```
gl.glBindFragDataLocation(myShaderProgram, 0, "myFragColor");
```

21.11 Weitere Beispiele

In den vorherigen Abschnitten wurden die Konzepte einer GL Applikation beschrieben und an zentralen Stellen anhand von Codebeispielen erläutert. Nicht für OpenGL im Speziellen relevanter, reiner Java Applikationscode, etwa für Ein- und Ausgaben, die Routinen zum Aufstellen des Frustums sowie der zugehörigen Projektionsmatrizen und das Modellieren der Szene können dem auf der Webseite der Veranstaltung verfügbaren Quellcode entnommen werden. Weiterhin werden externe Daten benötigt, wie die Farbtextur, welche ebenfalls unter der angegebenen URL verfügbar ist. Das auf der nächsten Doppelseite folgende minimale Codebeispiel ist dagegen vollständig und benötigt keine externen Daten:

```
import javax.swing.JApplet; import java.nio.*; import com.sun.opengl.util.*;
import javax.media.opengl.*; import javax.media.opengl.awt.GLCanvas;

public class GLApplet extends JApplet implements GLEventListener {

    private GL3 gl; // Objekt zum Absetzen der GL-Befehle
    private int programObject, colorBuffer, coordBuffer;
    private GLCanvas canvas;

    public void init() {
        canvas = new GLCanvas(new GLCapabilities(GLProfile.get(GLProfile.GL3)));
        canvas.addGLEventListener(this);
        add(canvas);
    }

    // Erzeugt ein Shaderobjekt, siehe Abschnitt 21.10
    private int LoadShader(int type, String[] shaderSrc) {
        int shader = gl.glCreateShader(type);
        gl.glShaderSource(shader, 1, shaderSrc, null);
        gl.glCompileShader(shader);
        return shader;
    }

    // JOGL Methode, wird durch FPSAnimator aufgerufen
    public void display(GLAutoDrawable drawable) {
        gl.glClear(GL3.GL_COLOR_BUFFER_BIT); // Framebuffer loeschen

        // Per-Vertex Daten Re-Aktivieren und anbinden. (Abschnitt 21.10)
        gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, coordBuffer);
        int location = gl.glGetAttribLocation(programObject, "vPosition");
        gl.glVertexAttribPointer(location, 3, GL3.GL_FLOAT, false, 0, 0);
        gl.glEnableVertexAttribArray(location);
        gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, colorBuffer);
        location = gl.glGetAttribLocation(programObject , "vColor");
        gl.glVertexAttribPointer(location, 3, GL3.GL_FLOAT, false, 0, 0);
        gl.glEnableVertexAttribArray(location);

        // Zeichne aktivierte und gebundene Daten als Dreiecke
        gl.glDrawElements(GL3.GL_TRIANGLES, 3, GL3.GL_UNSIGNED_INT, 0);

        // Setze Backbuffer nach vorn
        canvas.swapBuffers();
    }

    // JOGL Methode, wird bei Initialisierung aufgerufen
    public void init(GLAutoDrawable drawable) {
        drawable.addGLEventListener(this);
        gl = drawable.getGL().getGL3();

        // Erzeuge serverseitige Datenstrukturen für Per-Vertex Daten,
        // hier Position und Farbe. (siehe Abschnitt 21.6.1)
        IntBuffer bufferNames = BufferUtil.newIntBuffer(3);
        gl.glGenBuffers(3, bufferNames);
        FloatBuffer vertexCoords = BufferUtil.newFloatBuffer(
                            new float[]{0, 1, 0, -1, -1, 0, 1, -1, 0});
        coordBuffer = bufferNames.get(0);
        gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, coordBuffer);
        gl.glBufferData(GL3.GL_ARRAY_BUFFER, 9*4, vertexCoords,
                        GL3.GL_STATIC_DRAW);
        FloatBuffer vertexColors = BufferUtil.newFloatBuffer(new float[]
                    {1, 0, 0, 0, 1, 0, 0, 0, 1});//rot, gruen, blau
        colorBuffer = bufferNames.get(1);
        gl.glBindBuffer(GL3.GL_ARRAY_BUFFER, colorBuffer);
        gl.glBufferData(GL3.GL_ARRAY_BUFFER, 9*4, vertexColors,
                        GL3.GL_STATIC_DRAW);
```

```
        // Erzeuge einen IndexBuffer.  (siehe Abschnitt 21.6.2)
        IntBuffer indices = BufferUtil.newIntBuffer(new int[] {0, 1, 2});
        int indexBuffer = bufferNames.get(2);
        gl.glBindBuffer(GL3.GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
        gl.glBufferData(GL3.GL_ELEMENT_ARRAY_BUFFER, 3*4, indices,
                        GL3.GL_STATIC_DRAW );

        // Vertex Shader zum Durchreichen von Position und Farbe
        String vsCode[] = {" #version 140                    \n"
                         + " in vec3 vPosition;              \n"
                         + " in vec3 vColor;                 \n"
                         + " out vec3 color;                 \n"
                         + " void main() {                   \n"
                         + "  color = vColor;                \n"
                         + "  gl_Position.xyz = vPosition;   \n"
                         + "  gl_Position.w = 1.0;           \n"
                         + " }                               \n"};

        // Fragment Shader zum Einfärben mit der interpolierten
        // Vertex Farbe
        String fsCode[] = {" #version 140                    \n"
                         + " in vec3 color;                  \n"
                         + " out vec4 fragColor;             \n"
                         + " void main(){                    \n"
                         + "  fragColor.rgb = color;         \n"
                         + "  fragColor.a = 1.0;             \n"
                         + " }                               \n"};

        // Erzeugen eines Program Objects bestehend aus obigen Shadern
        // Siehe Abschnitt 21.10
        int vertexShader = LoadShader(GL3.GL_VERTEX_SHADER, vsCode);
        int fragmentShader = LoadShader(GL3.GL_FRAGMENT_SHADER, fsCode);
        programObject = gl.glCreateProgram();
        gl.glAttachShader(programObject, vertexShader);
        gl.glAttachShader(programObject, fragmentShader);
        gl.glBindFragDataLocation(programObject, 0, "fragColor");
        gl.glLinkProgram(programObject);
        gl.glUseProgram(programObject);

        new FPSAnimator(canvas, 60).start();
    }

    // JOGL Methode, wird bei Veränderung des Fensters aufgerufen
    public void reshape(GLAutoDrawable drawable, int x, int y, int width,
            int height) {
        requestFocusInWindow();
        setSize(width, height);
    }

    public void dispose(GLAutoDrawable arg0) {}
}
```

Die Ausgabe des Dreiecks mit Farbinformationen für die Vertices ist in Abbildung 21.7 zu sehen. Einer der Gründe für die Kürze des Beispiels ist der Verzicht auf eine Projektion der Vertices. Dementsprechend müssen deren Koordinaten direkt im Bildraum angegeben werden.

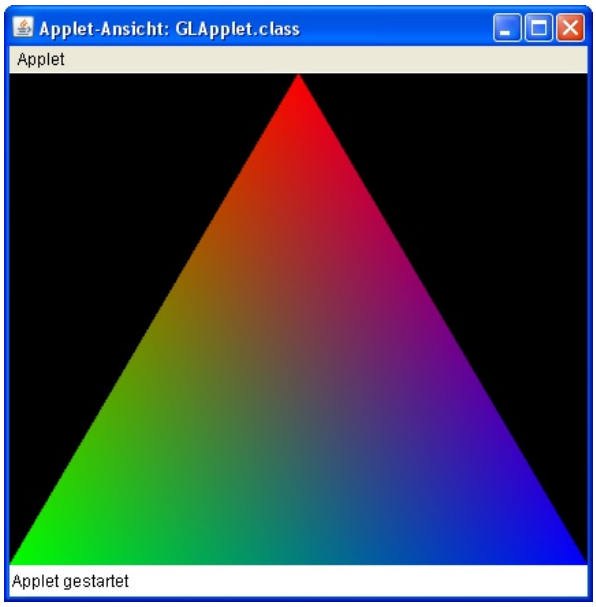


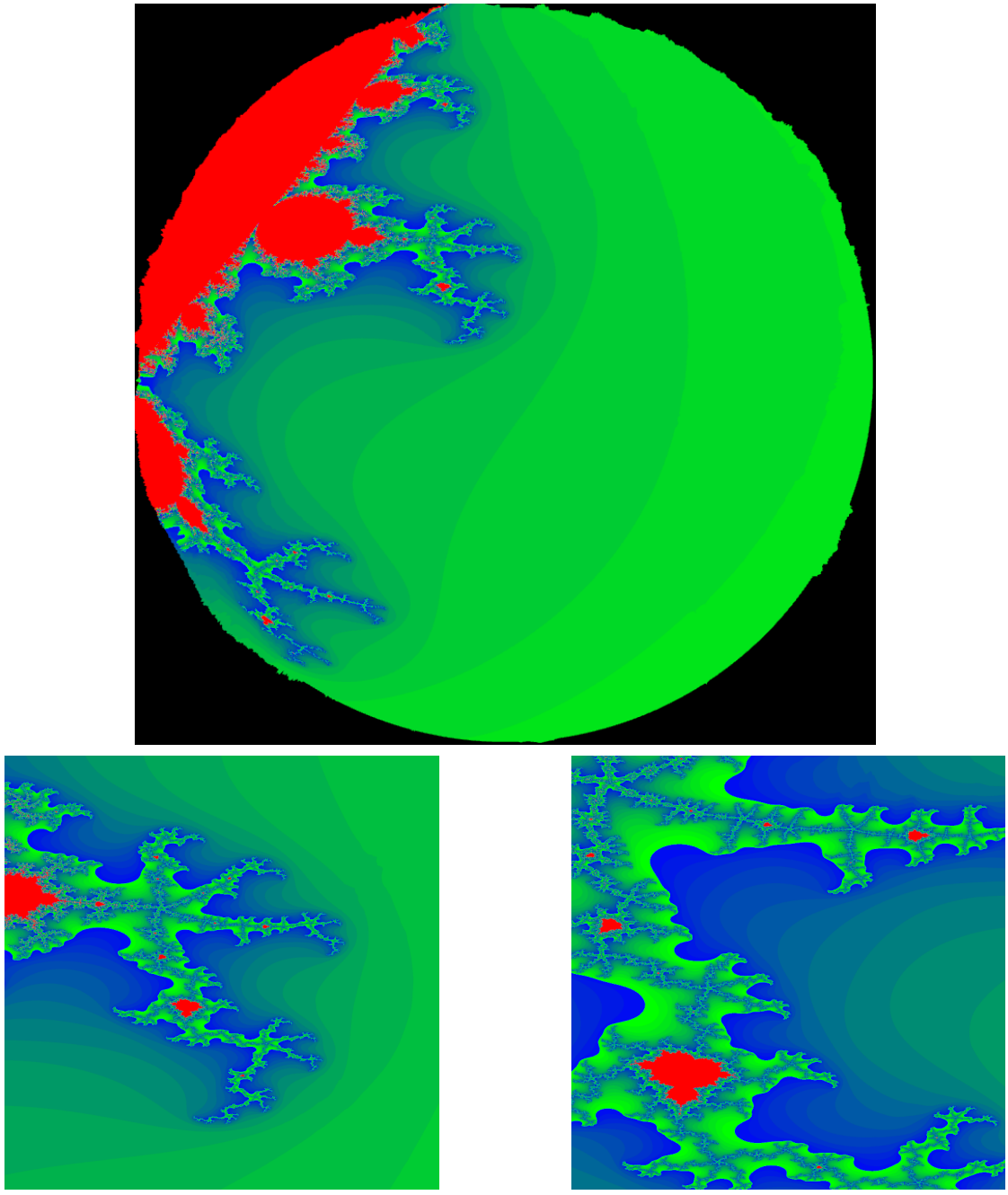
Abbildung 21.7: Ein Dreieck, eingefärbt mit den interpolierten Farben der Vertices

Als abschließendes Beispiel enthält das folgende Listing einen Fragment Shader zur Einfärbung einer Oberfläche mit einem Mandelbrot Fraktal:

```
#version 140
const float maxIterations = 100.0;
const vec3 innerColor  = vec3(1.0, 0.0, 0.0);
const vec3 outerColor1 = vec3(0.0, 1.0, 0.0);
const vec3 outerColor2 = vec3(0.0, 0.0, 1.0);

// Berechnet nur in Abhängigkeit von den Texturkoordinaten...
in vec2 texCoords;

// ...die Farbe des Fragments
out vec4 myFragColor;

void main() {
    float real  = texCoords.x; float imag = texCoords.y;
    float cReal = real; float cImag = imag;
    float r2 = 0.0;
    float iter;
    for(iter = 0.0; iter < maxIterations && r2 < 4.0; iter++) {
    float tempreal = real;
        real = (tempreal * tempreal) - ( imag * imag) + cReal;
        imag = 2.0 * tempreal * imag + cImag;
        r2   = (real * real) + (imag * imag);
    }
    vec3 color;
    if (r2 < 4.0)
        color = innerColor;
    else // mix ist eine GLSL-Funktion für lineare Interpolation
        color = mix(outerColor1, outerColor2, fract(iter * 0.05)) ;
     myFragColor = vec4(color, 1.0) ;
}
```

Dieser Shader berechnet die Oberflächenfarbe, wie der Fragment Shader aus Abschnitt 21.9.2, in Abhängigkeit der Texturkoordinate und kann folglich an dessen Stelle im Beispielprogramm eingesetzt werden. Die Oberfläche der auch in Abb. 21.1 dargestellten Erdkugel sieht dann, in verschiedenen

Zoomstufen, wie folgt aus:



21.12 Literatur

Der begrenzte Rahmen in dieser Veranstaltung ermöglicht leider keine vollständige Einführung in OpenGL und die Shaderprogrammierung. So mussten nicht nur viele wichtige Themengebiete ausgeklammert werden, auch die Beschreibung der einzelnen OpenGL Funktionen hat größtenteils Bei-

spielcharakter und beschreibt diese nicht vollständig. Nachfolgend einige Vorschläge zur weiteren Vertiefung:

- **Shreiner: „OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1“, ISBN-13: 978-0321552624**
  Dieses Werk, auch unter dem Namen Redbook bekannt, stellte in früheren Versionen eine sehr lesbare Einführung in OpenGL dar. Es ist für die nicht abwärtskompatible GL Version 3.1 leider nicht komplett überarbeitet, sondern nur an einigen Stellen erweitert worden. Weiterhin sind die entfernten Befehle noch enthalten und werden mit Erläuterungen zu existierenden Funktionen vermischt. Trotzdem kann es mangels verständlich formulierter Alternativen unter Vorbehalt empfohlen werden. Hinweis: Dieses Buch ist in Safari verfügbar.

- **OpenGL 4.0 Core Profile Specification**
  Sicherlich nicht so verständlich geschrieben und mit Beispielen versehen wie das Redbook, enthält die Spezifikation konsequent überarbeitete und detaillierte Ausführungen zur aktuellen OpenGL Version.

- **Rost: „OpenGL Shading Language, Third Edition“, ISBN-13: 978-0321637635**
  Sehr lesenswerte Einführung in GLSL 1.40, die außerdem einige interessante weiterführende Themen anspricht und viele Beispiele enthält. Ein OpenGL (3.1) Überblick ist in diesem auch als Orange Book bezeichneten Werk ebenfalls vorhanden, dieser bleibt jedoch für den Einstieg ohne vorherige Erfahrung etwas knapp und ist auch nicht vollständig. Hinweis: Dieses Buch ist in Safari verfügbar.

- **OpenGL Shading Language 4.00 Specification**
  Die Spezifikation der aktuellen GLSL Version 4.0 kann ergänzend zum Orange Book gelesen werden, insbesondere wenn neuere Features wie der Geometry Shader oder Tessellation eingesetzt werden sollen.

- **Munshi: „OpenGL ES 2.0 Programming Guide“, ISBN-13: 978-0321502797**
  Gute Einführung in die aktuelle Version von OpenGL for Embedded Systems, welche vor allem im mobilen Bereich zum Einsatz kommt. OpenGL ES 2.0 ist OpenGL 3.1 hinsichtlich des Konzepts, des Funktionsumfangs und des Codes sehr ähnlich. Hinweis: Dieses Buch ist in Safari verfügbar.

Die in Safari verfügbaren Bücher sind von einem Rechner des Netzes der Uni Osnabrück aus erreichbar unter unter:

```
http://proquest.safaribooksonline.com/?uicode=osnabrueck
```

Die Spezifikationen zu OpenGL und GLSL sind verfügbar unter:

```
http://www.opengl.org/documentation/current_version/
```

Abschließend muss noch eine Warnung vor Büchern sowie zahlreichen im Netz verfügbaren Beispielen, Texten und Tutorials ausgesprochen werden, die nicht mindesten OpenGL 3.1 und GLSL 1.4 thematisieren. Der Code ist nicht kompatibel und die beschriebenen Konzepte haben unter Umständen mit den Aktuellen wenig gemein.

Kapitel 22

Radiosity

22.1 Globale Beleuchtung

Ein Beleuchtungsmodell berücksichtigt bei der Berechnung der Farbe für einen Punkt das von den Lichtquellen direkt abgegebene Licht und das Licht, das den Punkt nach Reflexion und Transmission durch seine eigene und andere Flächen erreicht. Dieses indirekt reflektierte und hindurchgelassene Licht heißt *globale Beleuchtung*. Als *lokale Beleuchtung* bezeichnet man das Licht, das direkt von den Lichtquellen auf den schattierten Punkt fällt. Bisher wurde die globale Beleuchtung durch einen Term für ambiente Beleuchtung modelliert, der für alle Punkte auf allen Objekten konstant war. Dieser Term berücksichtigte weder die Positionen von Objekt und Betrachter noch Objekte, die das Umgebungslicht blockieren könnten.

In realen Szenen kommt nur ein geringer Teil des Lichts aus direkten Lichtquellen. Es gibt zwei Klassen von Algorithmen zur Erzeugung von Bildern, die die Bedeutung globaler Beleuchtung hervorheben. Das nächste Kapitel behandelt den Ray Tracing-Algorithmus, der neben der Ermittlung sichtbarer Flächen und deren Schattierung gleichzeitig Schatten, Reflexion und Brechung berechnet. Globale spiegelnde Reflexion und Transmission ergänzen dabei die für eine Fläche berechnete lokale spiegelnde, diffuse und ambiente Beleuchtung. Im Gegensatz dazu trennen die in diesem Kapitel behandelten Radiosity-Verfahren die Schattierung völlig von der Ermittlung sichtbarer Flächen. Sie berechnen erst in einem vom Blickpunkt unabhängigen Schritt alle Interaktionen einer Szene mit den Lichtquellen. Dann berechnen sie mit konventionellen Algorithmen zur Ermittlung sichtbarer Flächen und Schattierung durch Interpolation ein oder mehrere Bilder für die gewünschten Standpunkte des Betrachters.

Vom Blickpunkt abhängige Algorithmen (z.B. Ray Tracing) diskretisieren die Bildebene, um die Punkte zu ermitteln, an denen die Beleuchtungsgleichung für eine bestimmte Blickrichtung ausgewertet wird. Beleuchtungsalgorithmen, die vom Blickpunkt unabhängig sind (z.B. Radiosity), diskretisieren dagegen die Szene und verarbeiten sie weiter, um genügend Informationen zur Auswertung der Beleuchtungsgleichung an jedem beliebigen Punkt und für jede Blickrichtung zu erhalten. Die Algorithmen, die vom Blickpunkt abhängig sind, eignen sich gut zur Behandlung von Spiegelungen, die stark vom Standpunkt des Betrachters abhängen. Sie erfordern jedoch bei der Behandlung diffuser Phänomene zusätzlichen Aufwand, da sich diese über große Bildbereiche oder zwischen Bildern aus verschiedenen Blickpunkten wenig ändern. Die Algorithmen, die nicht vom Blickpunkt abhängen, modellieren diffuse Phänomene dagegen effizient, haben aber bei der Behandlung von Spiegelungen

einen enorm hohen Speicherbedarf.

22.2 Physikalische Ausgangslage

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. Die Terme für ambiente und diffuse Beleuchtung müssen dann beim Einfärben nicht mehr berechnet werden, da sie wesentlich realistischer bereits in den Radiosity-Werten enthalten sind. Dann werden eine oder mehrere (von unterschiedlichen Augenpunkten betrachtete) Darstellungen gerastert. Dabei fällt nur noch der Aufwand für die Ermittlung der sichtbaren Flächen und für die Schattierung durch Interpolation an.

22.3 Die Radiosity-Gleichung (Beleuchtungsgleichung)

Die bisher betrachteten Schattierungsalgorithmen behandeln die Lichtquellen immer unabhängig von den beleuchteten Flächen. Bei den Radiosity-Verfahren kann dagegen jede Fläche Licht abstrahlen. Daher werden alle Lichtquellen mit einem inhärenten Flächeninhalt modelliert. Die Szene wird in eine endliche Anzahl n diskreter Flächenelemente (*Patches*) zerlegt. Jedes dieser Elemente hat endliche Größe, strahlt über die gesamte Fläche gleichmäßig Licht ab und reflektiert Licht. Wenn man jedes der n Flächenelemente als opaken, diffusen Lambertschen Strahler und Reflektierer betrachtet, gilt für Fläche i

$$B_i \cdot A_i = E_i \cdot A_i + \rho_i \sum_{j=1}^{n} B_j \cdot F_{ji} \cdot A_j, 1 \leq i \leq n \; .$$

mit

E_i	=	von der Fläche i abgegebene Eigenstrahlung pro Flächeneinheit
ρ_i	=	Reflexionsvermögen der Fläche i
n	=	Anzahl der Flächen
A_i	=	Größe der Fläche i
F_{ji}	=	Anteil an der von Fläche j abgegebenen Energie, die auf Fläche i auftrifft, pro Flächeneinheit, genannt *Formfaktor*
B_i	=	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. $B_j \cdot F_{ji} \cdot A_j$ ist der Betrag des Lichts, das die ganze Fläche j verläßt und die Fläche i erreicht.

Zwischen den Formfaktoren in diffusen Szenen besteht eine nützliche Beziehung:

$$A_i \cdot F_{ij} = A_j \cdot F_{ji} \ ,$$

wobei A_i und A_j die Flächeninhalte sind. Daraus ergibt sich

$$F_{ij} = \frac{A_j}{A_i} \cdot F_{ji}$$

Umordnen der Ausdrücke liefert

$$B_i - \rho_i \sum_{1 \le j \le n} B_j \cdot F_{ij} = E_i \ .$$

Also kann der Austausch von Licht innerhalb der Flächenelemente der Szene durch ein Gleichungssystem ausgedrückt werden:

$$\begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \dots & -\rho_1 F_{1n} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \dots & -\rho_2 F_{2n} \\ . & . & . & \\ . & . & . & \\ . & . & . & \\ -\rho_n F_{n1} & -\rho_n F_{n2} & \dots & 1-\rho_n F_{nn} \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ B_2 \\ . \\ . \\ . \\ B_n \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ . \\ . \\ . \\ E_n \end{pmatrix}$$

Man beachte, daß der Beitrag eines Flächenelements zu seiner eigenen reflektierten Energie berücksichtigt werden muß (es könnte zum Beispiel konkav sein). Daher haben im allgemeinen nicht alle Terme auf der Diagonalen den Wert Eins.

Wenn man die Gleichung löst, erhält man für jedes Flächenelement einen Strahlungswert. Die Elemente können dann für jeden gewünschten Blickpunkt mit einem gewöhnlichen Algorithmus zur Ermittlung sichtbarer Flächen gerastert werden. Die Strahlungswerte sind die Intensitäten dieses Elements.

Gauß-Seidel-Iterations-Verfahren zur Lösung von linearen Gleichungssystemen

Die i-te Gleichung eines linearen Gleichungssystems $Ax = b$ lautet:

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

Wenn alle Diagonalelemente von A ungleich Null sind, gilt:

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

Das Iterations-Verfahren startet mit einer Schätzung x_1, die auf der rechten Seite eingesetzt wird zur Berechnung von $x_1[1]$. Im nächsten Schritt wird zur Berechnung von $x_1[2]$ bereits $x_1[1]$ benutzt. Ein Iterationsschritt lautet also:

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

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

22.4 Berechnung der Formfaktoren

Die Formfaktoren lassen sich definieren über die Gleichung

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

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.
A_i	=	Flächeninhalt von Fläche F_i

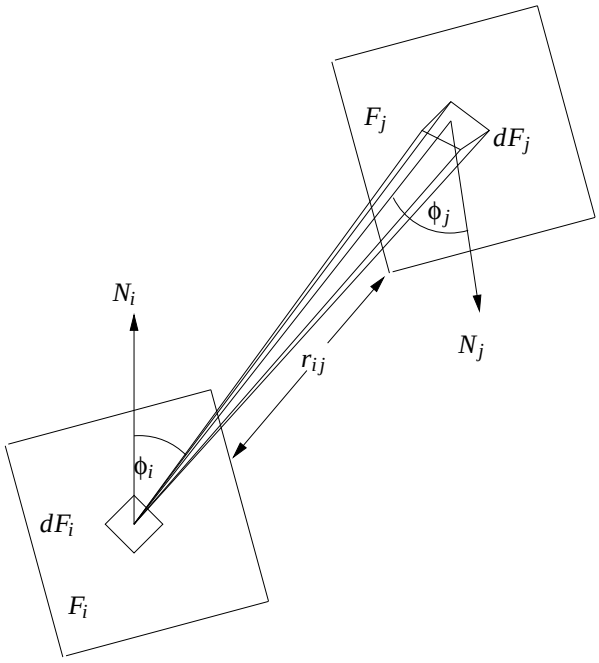


Abbildung 22.1: Der Berechnung von Formfaktoren zugrundeliegende Geometrie

Die geometrische Interpretation beschreibt den Formfaktor als das Verhältnis der Basisfläche einer Halbkugel zur Orthogonalprojektion der auf die Halbkugel projizierten Fläche: Erst projiziert man die von F_i aus sichtbaren Teile von F_j auf eine Halbkugel mit Radius 1 um dF_i, projiziert diese Projektion orthogonal auf die kreisförmige Grundfläche der Halbkugel und dividiert schließlich durch die Kreisfläche. Die Projektion auf die halbe Einheitskugel entspricht in der Gleichung dem Term $\cos\phi_j / r_{ij}^2$, die Projektion auf die Grundfläche entspricht der Multiplikation mit $\cos\phi_i$, und die Division durch

den Flächeninhalt des Einheitskreises liefert den Wert π im Nenner.

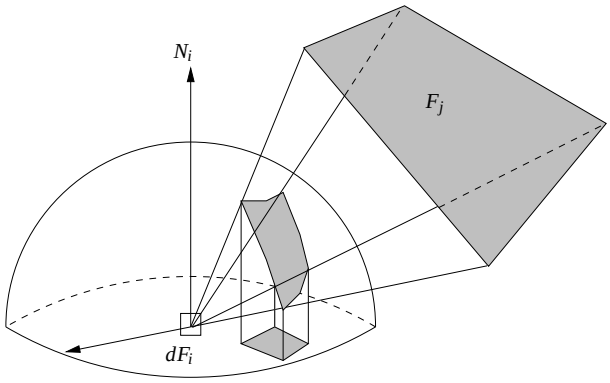


Abbildung 22.2: Geometrische Interpretation des Formfaktors

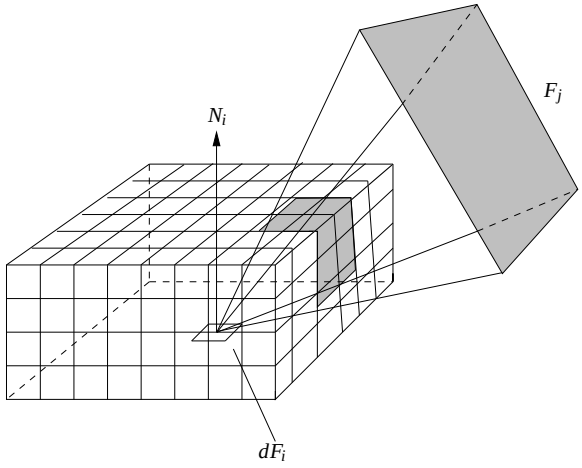


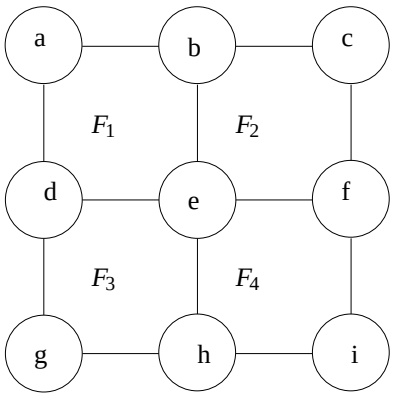
Abbildung 22.3: Simulation der Halbkugel durch Halbwürfel

Zur numerischen Berechnung wird die Halbkugel durch einen *Halbwürfel* (hemi-cube) mit dem Zentrum im Ursprung und dem Normalvektor in der z-Achse ersetzt. Die Oberseite des Würfels ist dabei parallel zur Fläche. Jede Seite des Halbwürfels wird in ein Raster gleich großer quadratischer Zellen aufgeteilt. (Die Auflösungen reichen von 50×50 bis zu mehreren Hundert pro Seite.) Dann wird jedes Flächenelement auf die Seiten des Halbwürfels projiziert. 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

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

22.5 Interpolation der Pixelfarben

Für jede Fläche i liege nun ihre *Radiosity* B_i vor. Hieraus interpoliert man die Strahlung für die Eckpunkte:



$$
\begin{array}{lcllcl}
B(e) & = & (B_1+B_2+B_3+B_4)/4 & & & \\
B(a) & = & B_1+(B_1-B(e)) & B(c) & = & B_2+(B_2-B(e)) \\
B(g) & = & B_3+(B_3-B(e)) & B(i) & = & B_4+(B_4-B(e)) \\
 & & & & & \\
B(b) & = & (B(a)+(B(c))/2 & B(d) & = & (B(a)+B(g))/2 \\
B(f) & = & (B(c)+(B(i))/2 & B(h) & = & (B(g)+B(i))/2
\end{array}
$$

Nun wird jedes Patch in zwei Dreiecke zerteilt, so daß nach der Projektion die Strahlungswerte an den Dreiecksecken zur zweifachen Interpolation verwendet werden können.

22.6 Schrittweise Verfeinerung

In Anbetracht des hohen Aufwands beim bisher beschriebenen Radiosity-Algorithmus stellt sich die Frage, ob man die Ergebnisse der Beleuchtungsberechnung inkrementell approximieren kann. Die Auswertung der i-ten Zeile des Gleichungssystems liefert eine Schätzung für die Strahlung B_i des Flächenelements, die auf den Schätzungen für die Strahlungswerte der anderen Flächenelemente basiert. Jeder Term in der Summe der Gleichung beschreibt die Auswirkung des Elements j auf die Strahlung des Elements i:

$$B_i \text{ updaten durch } \sum_j^n \rho_i B_j F i j$$

Diese Methode “sammelt” also das Licht der restlichen Szene ein.

Der Ansatz zur schrittweisen Verfeinerung verteilt dagegen die Strahlung eines Flächenelements auf die Szene, wobei jede Fläche j mithilfe von Fläche i aktualisiert wird:

$$B_j \text{ updaten durch } \rho_j B_i F ji$$

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.

$$\text{Beitrag von } B_i \text{ zu } B_j = \rho_j B_i F_{ij} \frac{A_i}{A_j} \text{ für alle } j \ .$$

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 $F_{ji} = (F_{ij} \cdot A_i)/A_j$.

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

repeat
   for {jede Fläche i, beginnend bei größter Ausstrahlung} do begin
     Plaziere Hemicube auf Fläche i
     Berechne Fij für alle 1 ≤ j ≤ n
     for j := 1 to n do begin              {für jede Fläche j tue}
        ΔR := ρj * ΔBi * Fij * Ai/Aj       {Strahlung von Fläche i}
        ΔBj := ΔBj + ΔR                    {Differenz erhöhen}
        Bj := Bj + ΔR                      {Strahlung erhöhen}
     end;
     ΔBi := 0;                             {Überschuß ist verteilt}
   end
until fertig                               {bis zur Konvergenz}
```

Bei jeder Ausführung der äußeren FOR-Schleife verteilt ein weiteres Flächenelement seine unver-

brauchte Strahlung auf die Szene. Daher werden nach der ersten Ausführung nur die Flächen beleuchtet, die selbst Lichtquellen sind sowie solche, die beim Verteilen der Strahlung des ersten Elements direkt beleuchtet werden. Rastert man am Ende jeder Ausführung des Codes ein neues Bild, so wird das erste Bild relativ dunkel und die nachfolgenden Bilder immer heller.

Abbildung 22.4 faßt den gesamten Ablauf zusammen.

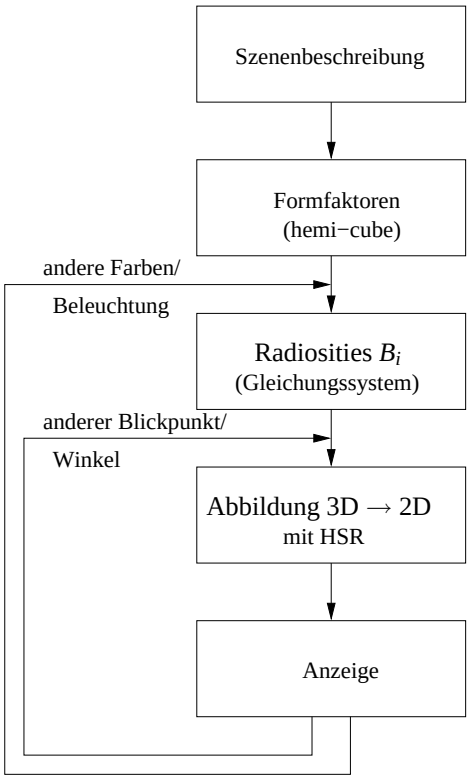
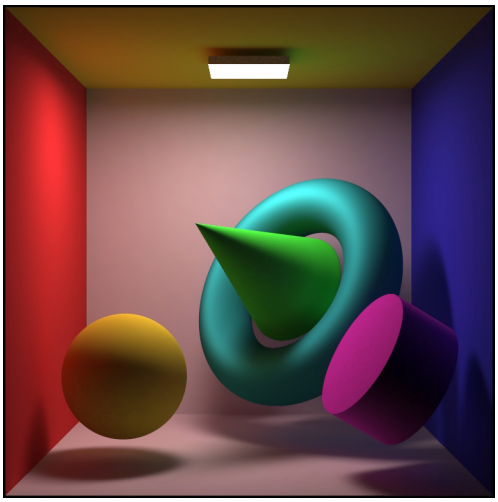


Abbildung 22.4: Ablaufschema beim Radiosityverfahren

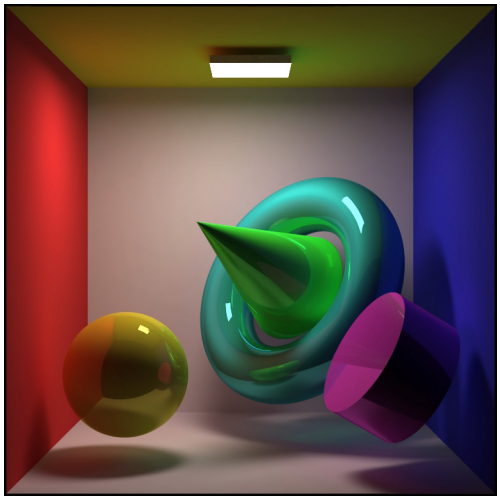
22.7 Screenshots

Auf der Seite `http://www.graphics.cornell.edu/online/research` gibt es einige Beispiele für Radiosity-Bilder.

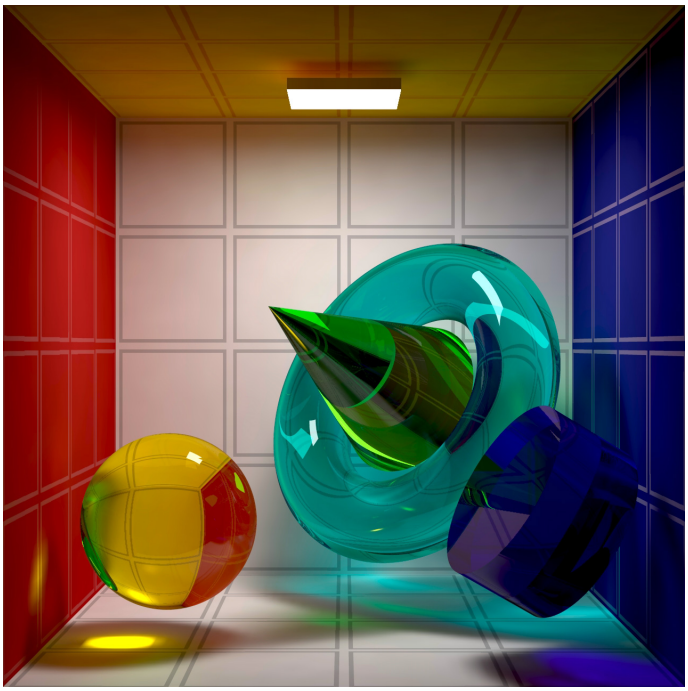
Die nächsten drei Bilder sind Screenshots vom Advanced Rendering Toolkit der Technischen Universität Wien (`http://www.cg.tuwien.ac.at/research/rendering/ART`). Hier wurde das klassische Radiosity-Verfahren um Ray-Tracing-Komponenten erweitert.



Diffuse Reflexion



Diffuse und spekulare Reflektion



Diffuse, spekulare und transparente Reflexion

Kapitel 23

Ray Tracing

23.1 Grundlagen

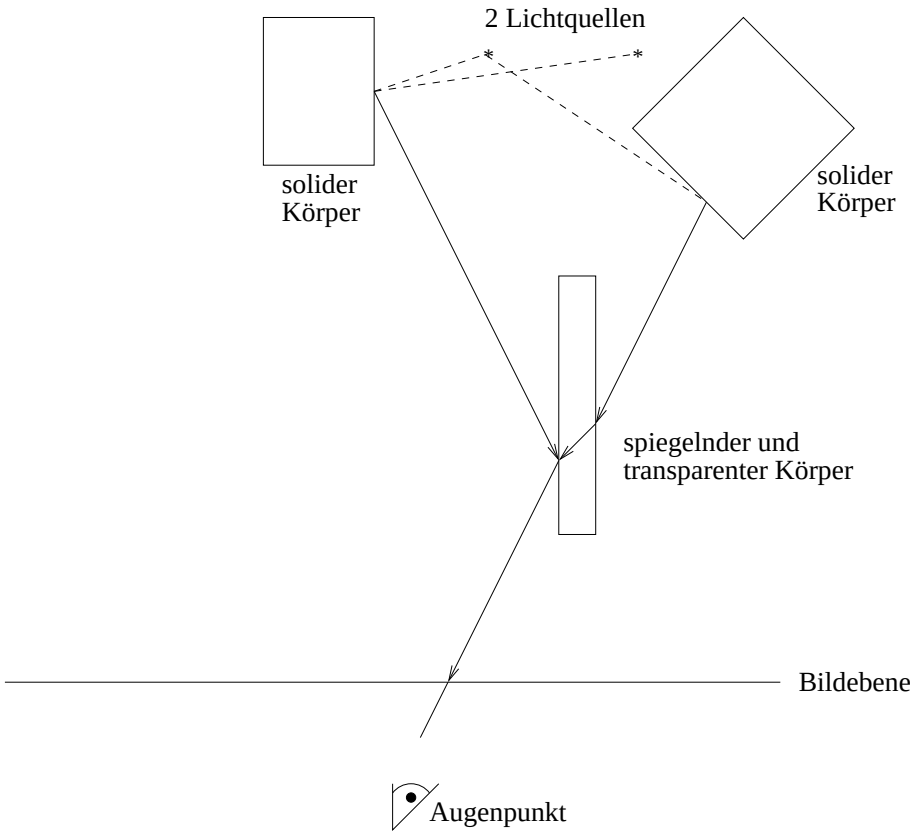


Abbildung 23.1: Prinzip der Strahlverfolgung

Verfahren mit *Ray Tracing* (Strahlverfolgung) eignen sich ausgezeichnet zur Modellierung von spiegelnden Reflexionen und von Transparenz mit Brechung (ohne Streuung). Globale Lichter werden mit einem ambienten Beleuchtungsterm ohne Richtung behandelt. Ausgehend vom Augenpunkt wird

durch jedes Pixel ein Strahl gelegt und der Schnittpunkt dieses Strahls mit dem ersten getroffenen Objekt bestimmt. Trifft der Strahl auf kein Objekt, so erhält das Pixel die Hintergrundfarbe. Ist das Objekt spiegelnd, so wird der Reflexionsstrahl berechnet und rekursiv weiterbehandelt. Ist das Objekt transparent, wird zusätzlich der gebrochene Strahl weiterbehandelt. Zur Berechnung von Schatten wird von jedem Schnittpunkt zwischen Strahl und Objekt zu jeder Lichtquelle ein zusätzlicher Strahl ausgesandt. Trifft dieser Strahl auf ein blockierendes Objekt, dann liegt der Schnittpunkt im Schatten dieser Lichtquelle, und das von ihr ausgestrahlte Licht geht in die Intensitätsberechnung des Punktes nicht ein.

23.2 Ermittlung sichtbarer Flächen durch Ray Tracing

Verfahren mit *Ray Tracing* ermitteln die Sichtbarkeit von Flächen, indem sie imaginäre Lichtstrahlen des Betrachters zu den Objekten der Szene verfolgen. Man wählt ein Projektionszentrum (das Auge des Betrachters) und ein Window in einer beliebigen Bildebene. Das Window wird durch ein regelmäßiges Gitter aufgeteilt, dessen Elemente den Pixeln in der gewünschten Auflösung entsprechen. Dann schickt man für jedes Pixel im Window einen *Augstrahl* vom Projektionszentrum durch den Mittelpunkt des Pixels auf die Szene. Das Pixel erhält die Farbe des ersten getroffenen Objekts. Das folgende Programm enthält den Pseudocode für diesen einfachen Ray Tracer.

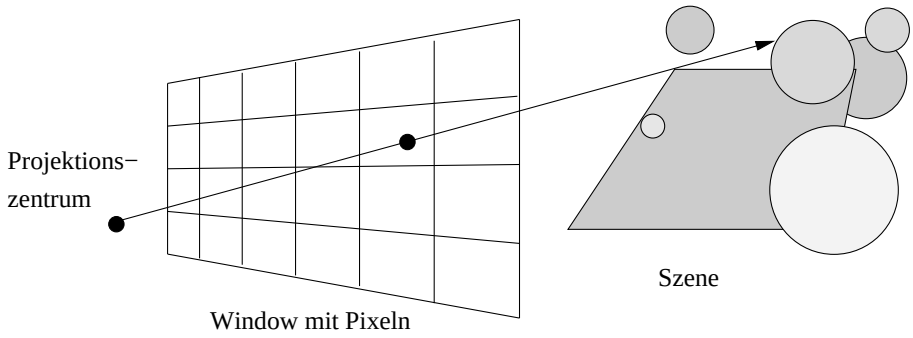


Abbildung 23.2: Strahl durch Bildebene

```
Wähle Projektionszentrum und Window in der Bildebene;
for(jede Rasterzeile des Bildes){
   for(jedes Pixel der Rasterzeile){
     Berechne den Strahl vom Projektionszentrum durch das Pixel;
     for(jedes Objekt der Szene){
        if(Objekt wird geschnitten und liegt bisher am nächsten)
           Speichere Schnittpunkt und Name des Objekts;
     }
     Setze das Pixel auf die Farbe des nächstliegenden Objektschnittpunkts;
   }
}
```

23.3 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 t 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.

23.4 Effizienzsteigerung zur Ermittlung sichtbarer Flächen

Die vorgestellte einfache, aber rechenintensive Version des Ray-Tracing-Algorithmus schneidet jeden Augstrahl mit jedem Objekt der Szene. Für ein Bild der Größe 1024×1024 mit 100 Objekten wären

daher 100 Mio. Schnittpunktberechnungen erforderlich. Ein System verbraucht bei typischen Szenen 75-95 Prozent der Rechenzeit für die Schnittpunktroutine. Daher konzentrieren sich die Ansätze zur Effizienzsteigerung auf die Beschleunigung der Schnittpunktberechnungen oder deren völlige Vermeidung.

Optimierung der Schnittpunktberechnungen

Die Formel für den Schnittpunkt mit einer Kugel läßt sich verbessern. Wenn man die Strahlen so transformiert, daß sie entlang der z-Achse verlaufen, kann man die gleiche Transformation auf die getesteten Objekte anwenden. Dann liegen alle Schnittpunkte bei $x = y = 0$. Dieser Schritt vereinfacht die Berechnung der Schnittpunkte. Das nächstliegende Objekt erhält man durch Sortieren der z-Werte. Mit der inversen Transformation kann man den Schnittpunkt dann für die Schattierungsberechnungen zurücktransformieren.

Begrenzungsvolumina eignen sich besonders gut dazu, die Zeit für die Berechnung der Schnittpunkte zu reduzieren. Man umgibt ein Objekt, für das die Schnittpunktberechnungen sehr aufwendig sind, mit einem Begrenzungsvolumen, dessen Schnittpunkte einfacher zu berechnen sind, z.B. Kugel, Ellipsoid oder Quader. Das Objekt wird nur dann getestet, wenn der Strahl das Begrenzungsvolumen schneidet.

Hierarchien

Begrenzungsvolumina legen zwar selbst noch keine Reihenfolge oder Häufigkeit der Schnittpunkttests fest. Sie können aber in verschachtelten Hierarchien organisiert werden. Dabei bilden die Objekte die Blätter, die inneren Knoten begrenzen ihre Söhne. Hat ein Strahl keinen Schnittpunkt mit einem Vaterknoten, dann gibt es garantiert auch keinen Schnittpunkt mit einem seiner Söhne. Beginnt der Schnittpunkttest also an der Wurzel, dann entfallen ggf. trivialerweise viele Zweige der Hierarchie (und damit viele Objekte).

Bereichsunterteilung

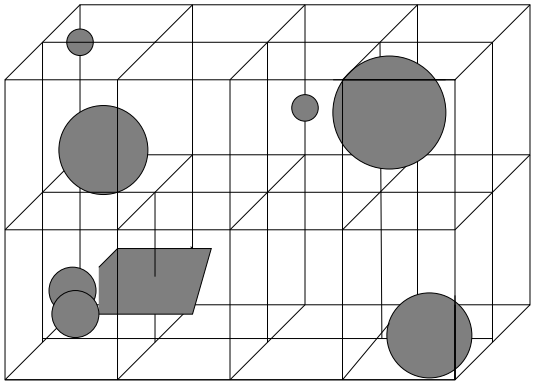


Abbildung 23.3: Unterteilung der Szene durch ein regelmäßiges Gitter

Die Hierarchie von Begrenzungsvolumina organisiert die Objekte von unten nach oben. Die Bereichsunterteilung teilt dagegen von oben nach unten auf. Zuerst berechnet man die *bounding box* der ganzen Szene. Bei einer Variante unterteilt man die *bounding box* dann in ein regelmäßiges Gitter gleich großer Bereiche. Jedem Bereich wird eine Liste mit den Objekten zugewiesen, die entweder ganz oder teilweise darin enthalten sind. Zur Erzeugung der Listen weist man jedes Objekt einem oder mehreren Bereichen zu, die dieses Objekt enthalten. Ein Strahl muß jetzt nur noch mit den Objekten in den durchlaufenen Bereichen geschnitten werden. Außerdem kann man die Bereiche in der Reihenfolge

untersuchen, in der sie vom Strahl durchlaufen werden. Sobald es in einem Bereich einen Schnittpunkt gibt, muß man keine weiteren Bereiche mehr testen. Man muß jedoch alle anderen Objekte des Bereichs untersuchen, um das Objekt mit dem nächstliegenden Schnittpunkt zu ermitteln.

23.5 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 23.4). Diese Spiegelungs-, Reflexions- und Brechungsstrahlen heißen *Sekundärstrahlen*, um sie von den *Primärstrahlen* zu unterscheiden, die vom Auge ausgehen. Gibt es bei dem Objekt spiegelnde Reflexion, dann wird ein Spiegelungsstrahl um die Flächennormale in Richtung r gespiegelt. Ist das Objekt transparent, und es gibt keine totale innere Reflexion, startet man einen Brechungsstrahl entlang t in das Objekt. Der Winkel wird nach dem Gesetz von Snellius bestimmt.

Jeder dieser Reflexions- und Brechungsstrahlen kann wiederum neue Spiegelungs-, Reflexions- und Brechungsstrahlen starten.

Typische Bestandteile eines *Ray-Tracing*-Programms sind die Prozeduren `shade` und `intersect`, die sich gegenseitig aufrufen. `shade` berechnet die Farbe eines Punktes auf der Oberfläche. Hierzu werden die Beiträge der reflektierten und gebrochenen Strahlen benötigt. Diese Beiträge erfordern die Bestimmung der nächstgelegenen Schnittpunkte, welche die Prozedur `intersect` liefert. Der Abbruch der Strahlverfolgung erfolgt bei Erreichen einer vorgegebenen Rekursionstiefe oder wenn ein Strahl kein Objekt trifft.

Seien v, N die normierten Vektoren für Sehstrahl und Normale. Sei $n = n_2/n_1$ das Verhältnis der beiden Brechungsindizes. Dann sind die Vektoren r und t, die im Schnittpunkt P zur Weiterverfolgung des reflektierten und gebrochenen Strahls benötigt werden, definiert durch

$$\begin{aligned} r &= 2\cos(\phi)N - v \\ t &= (-\cos(\phi) - q)N + v \\ \text{mit } q &= \sqrt{n^2 - 1 + \cos^2(\phi)} \end{aligned}$$

Ist q ein imaginärer Ausdruck, liegt Totalreflexion vor, und der Transmissionsanteil wird gleich Null gesetzt.

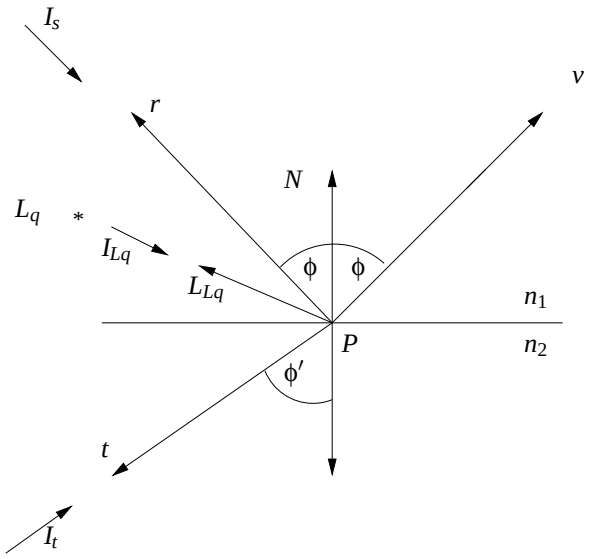


Abbildung 23.4: Einflußgrößen für Punkt P

In die Berechnung der Farbe im Punkt P geht die gewichtete Summe der in den einzelnen Rekursionsschritten berechneten Farbintensitäten ein. Unter Berücksichtigung des *Phong*-Modells ergibt sich

$$I = k_a I_a + k_d \sum_{Lq} I_{Lq}(\vec{N} \cdot \vec{L_{Lq}}) + k_s \sum_{Lq} I_{Lq}(\vec{v} \cdot \vec{R_{Lq}})^c + k_s I_s + k_t I_t$$

mit

k_a	ambienter Reflexionskoeffizient
I_a	Hintergrundbeleuchtung
k_d	diffuser Reflexionskoeffizient
I_{Lq}	Intensität der Lichtquelle
$\vec{N}$	Normale auf Ebene
$\vec{L_{Lq}}$	Vektor zur Lichtquelle
k_s	spiegelnder Reflexionskoeffizient
$\vec{v}$	Vektor zum Augenpunkt
$\vec{R_{Lq}}$	Reflexionsvektor für Lichtquelle L_q
c	spekularer Exponent
I_s	Reflexionssintensität (rekursiv ermittelt)
k_t	Transmissionskoeffizient (materialabhängig)
I_t	Transmissionsintensität (rekursiv ermittelt)

Pseudocode für einfaches rekursives Ray Tracing

```
Wähle Projektionszentrum und Window in der view plane;
for(jede Rasterzeile des Bildes){
   for(jedes Pixel P der Rasterzeile){
     Ermittle Strahl vom Projektionszentrum durch das Pixel;
     Color C = RT_intersect(ray, 1);
     Färbe Pixel P mit Farbe C;
   }
}

/* Schneide den Strahl mit Objekten, und berechne die Schattierung am nächsten */
/* Schnittpunkt. Der Parameter depth ist die aktuelle Tiefe im Strahlenbaum. */

RT_color RT_intersect(RT_ray ray, int depth)
{
   Ermittle den nächstliegenden Schnittpunkt mit einem Objekt;
   if(Objekt getroffen){
     Berechne die Normale im Schnittpunkt;
     return RT_shade(nächstliegendes getroffenes Objekt, Strahl, Schnittpunkt,
                     Normale, depth);
   }
   else
     return HINTERGRUND_FARBE;
}



/* Berechne Schattierung für einen Punkt durch Verfolgen der Schatten-, */
/* Reflexions- und Brechungsstrahlen */

RT_color RT_shade(
   RT_object object,          /* Geschnittenes Objekt */
   RT_ray ray,                /* Einfallender Strahl */
   RT_point point,            /* Schnittpunkt, der schattiert werden soll */
   RT_normal normal,          /* Normale in dem Punkt */
   int depth)                 /* Tiefe im Strahlenbaum */
{
RT_color color;               /* Farbe des Strahls */
RT_ray rRay, tRay, sRay;      /* Reflexions-, Brechungs- und Schattenstrahlen */
RT_color rColor, tColor;      /* Farbe des reflektierten und gebrochenen Strahls */

   color = Term für das ambiente Licht;


   for(jede Lichtquelle){
     sRay = Strahl von der Lichtquelle zum Punkt;
     if(Skalarprodukt der Normalen mit der Richtung zum Licht ist positiv){
        Berechne, wieviel Licht von opaken und transparenten Flächen blockiert 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.

23.6 Public Domain Ray Tracer Povray

Der “Persistence of Vision Ray Tracer” (POV-Ray) ist ein urheberrechtlich geschütztes Freeware-Programm zum Berechnen einer fotorealitischen Projektion aus einer 3-dimensionalen Szenenbeschreibung auf der Grundlage der Strahlverfolgung (`http://www.povray.org`). Seine Implementierung basiert auf DKBTrace 2.12 von David Buck & Aaron Collins. Zur Unterstützung des Anwenders gibt es einige include-Files mit vordefinierten Farben, Formen und Texturen.

```
#include "colors.inc"
#include "textures.inc"

camera {
       location <3, 3, -1>
       look_at  <0, 1,  2>
}
light_source { <0, 4, -3> color White}
light_source { <0, 6, -4> color White}
light_source { <6, 4, -3> color White}

plane {
       <0, 1, 0>, 0
       pigment {
               checker
                      color White
                      color Black
       }
}

sphere {
       <0, 2, 4>, 2
       texture {
               pigment {color Orange}
               normal  {bumps 0.4  scale 0.2}
               finish  {phong 1}
       }
}

box {
       <-1, 0, -0.5>,
       < 1, 2,  1.5>
       pigment {
               DMFWood4
               scale 2
       }
       rotate y*(-20)
}
```

Povray-Quelltext scene.pov

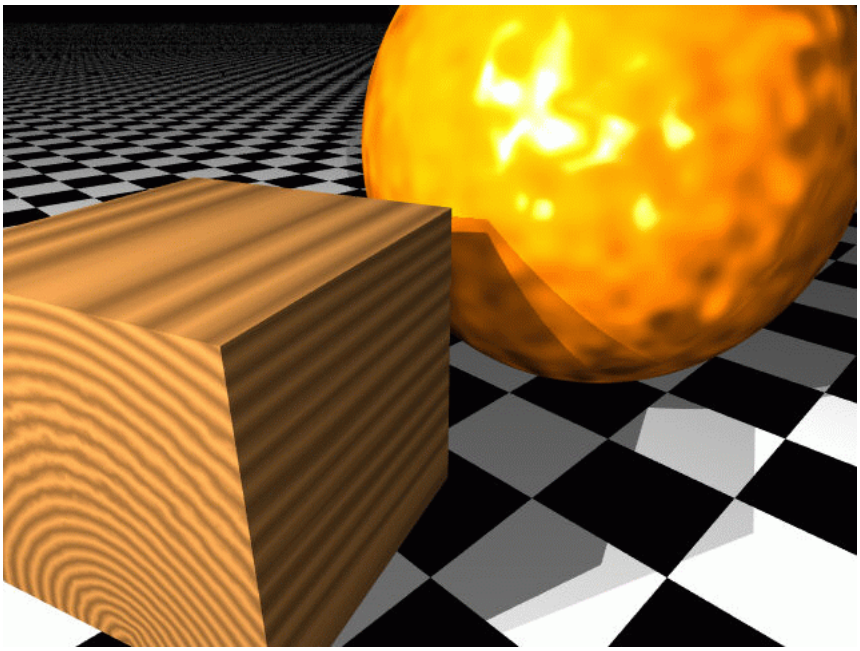


Abbildung 23.5: Povray-Bild zur Szenenbeschreibung

Der Aufruf von *povray* erfolgt zweckmäßigerweise mit einem Shell-Skript, dem der Name der zu bearbeitenden Datei als einziger Parameter übergeben wird:

```
#!/bin/sh

povray +I$1.pov +Oscene.tga +L/usr/lib/povray3/include +W320 +H200
```

Die vom Ray-Tracer erstellte Statistik lautet:

```
scene.pov Statistics, Resolution 320 x 200
-------------------------------------------------------------------------------
Pixels:           64000   Samples:           64000   Smpls/Pxl: 1.00
Rays:             64000   Saved:                 0   Max Level: 1/5
-------------------------------------------------------------------------------
Ray->Shape Intersection          Tests       Succeeded  Percentage
-------------------------------------------------------------------------------
Box                             238037           40342     16.95
Plane                           238037           62400     26.21
Sphere                          238037           50948     21.40
-------------------------------------------------------------------------------
Calls to Noise:                  0   Calls to DNoise:          83403
-------------------------------------------------------------------------------
Shadow Ray Tests:           522111   Succeeded:                24497
-------------------------------------------------------------------------------
Smallest Alloc:                 12 bytes   Largest:            12308
Peak memory used:           103504 bytes
-------------------------------------------------------------------------------
Time For Trace:    0 hours  0 minutes  18.0 seconds (18 seconds)
    Total Time:    0 hours  0 minutes  18.0 seconds (18 seconds)
```

Kapitel 24

Animation

In diesem Kapitel geht es um die Erzeugung von Animationseffekten in 3D-Szenen.

24.1 Key Frame Animation

Unter Key Frame Animation versteht man eine Sequenz von Transformationen, die für bestimmte Zeitpunkte (Key Frames) auf der Timeline definiert sind und die den Gesamtzustand eines Objekts verändern. Das System interpoliert für die Zeitpunkte zwischen den Key Frames die Effekte.

24.2 Forward Kinematics

3D-Figuren, bei denen gewisse Teile durch Gelenke gekoppelt sind, können gesteuert werden über die Angabe sämtlicher Winkelstellungen. Im realen Modell würden die Stellmotoren angesteuert und es resultiert daraus automatisch eine neue Position des Endeffektors $\vec{x}$. Im computergenerierten Film muss mithilfe von trigonometrischen Funktionen aus den Längen der Teilkörper und den zwischen ihnen geltenden Winkelstellungen Θ die neue Postion des Endeffektors $\vec{x} := f(\Theta)$ errechnet werden.

24.3 Inverse Kinematics

Ist von einem Roboterarm mit mehreren Gelenken die Zielstellung $\vec{x}$ bekannt und soll daraus die verursachende Winkelstellung Θ ermittelt werden, so spricht man von Inverse Kinematics. Da sich dieses Problem, also $\Theta = f^{-1}(\vec{x})$ nur in Spezialfällen lösen lässt, kommen meistens Iterationsverfahren zum Einsatz, bei denen aus dem schrittweisen Heranführen an die Zielposition die dazu assoziierten Winkelstellungen errechnet werden. Hierbei wird aus einer kleinen Raumänderung $\partial\vec{x}$ mithilfe der Jakobimatrix $J(\Theta)$ (alle partiellen Ableitungen) eine kleine Winkeländerung $\partial\Theta = J^{-1}(\Theta)(\partial\vec{x})$ bestimmt.

24.4 Particle Systems

Mit einem Particle System werden Tausende von kleinsten geometrischen Einheiten (Sandkörner, Wassertropen, Schneeflocken, Feuerfunken, ...) animiert. Jedes Partikel speichert einige charakteristische Daten (Position, Geschwindigkeit, Richtung, Lebenszeit, Größe, Farbe, ...), welche sich während der Simulation unter Anwendung physikalischer Gesetze verändern. Die Partikel interagieren weder mit der Umwelt noch untereinander.

Die Firma Scanline (`http://www.scanlinevfx.com`) verfügt über ein sehr leistungsfähiges System.

24.5 Verhaltensanimation

Bei einer Verhaltensanimation werden Hunderte von autonomen Agenten manipuliert, die durch einige charakteristische Daten (Position, Masse, Geschwindigkeit, Beschleunigung, ...) beschrieben werden. Jeder Agent verfolgt eine gewisse Strategie zum Erreichen seines Ziels (z.B. eine Türöffnung zu durchlaufen) und interagiert dabei sowohl mit der Umwelt als auch mit anderen Agenten.

Weiterführendes Material gibt es bei `http://www.red3d.com/cwr/steer`.

Kapitel 25

Maxon Cinema4D

3D-Modellierung unter Verwendung von

- Transformation (Translation, Skalierung, Rotation)
- Boole’schen Operatoren (Vereinigung, -schnitt, -komplement)
- Bezierkurven, NURBS
- Extrusion (Lathe, Sweep)
- Verformung (Biegen, Drehen, Schwerkraft, ...)

Projektion und Rendern (auch mit Radiosity) unter Berücksichtigung von

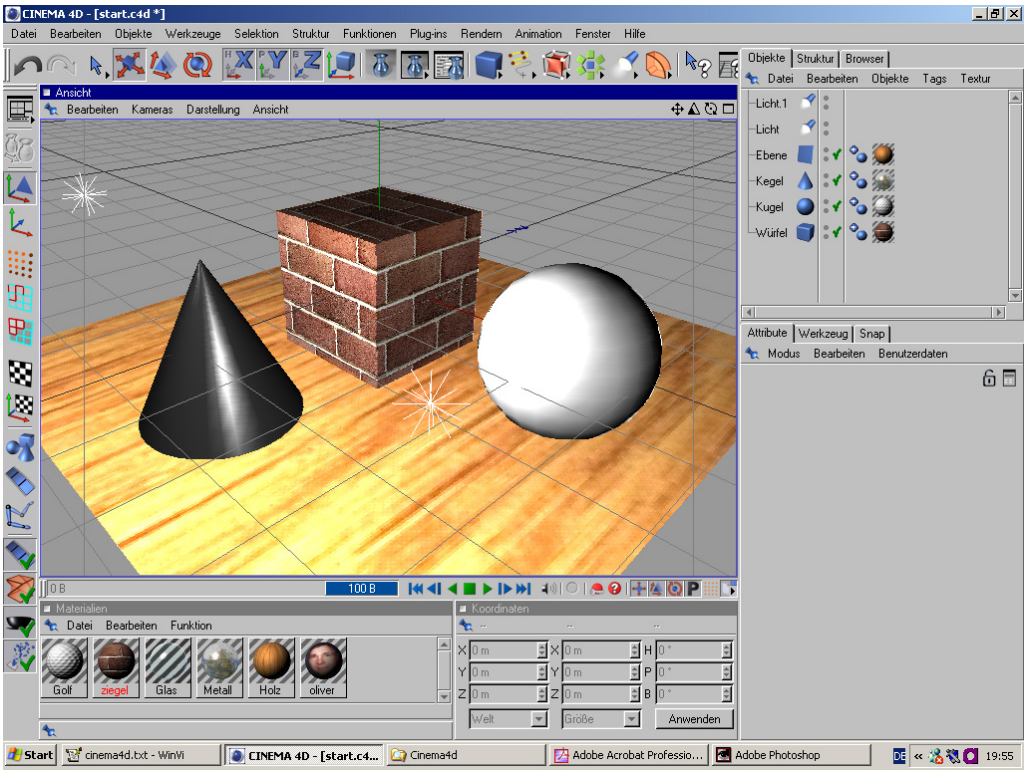
- Lichtquellen mit Schatten
- Spiegelung, Transparenz, Lichtbrechung
- Texture Mapping, Bump Mapping, Displacement Mapping

Animation durch

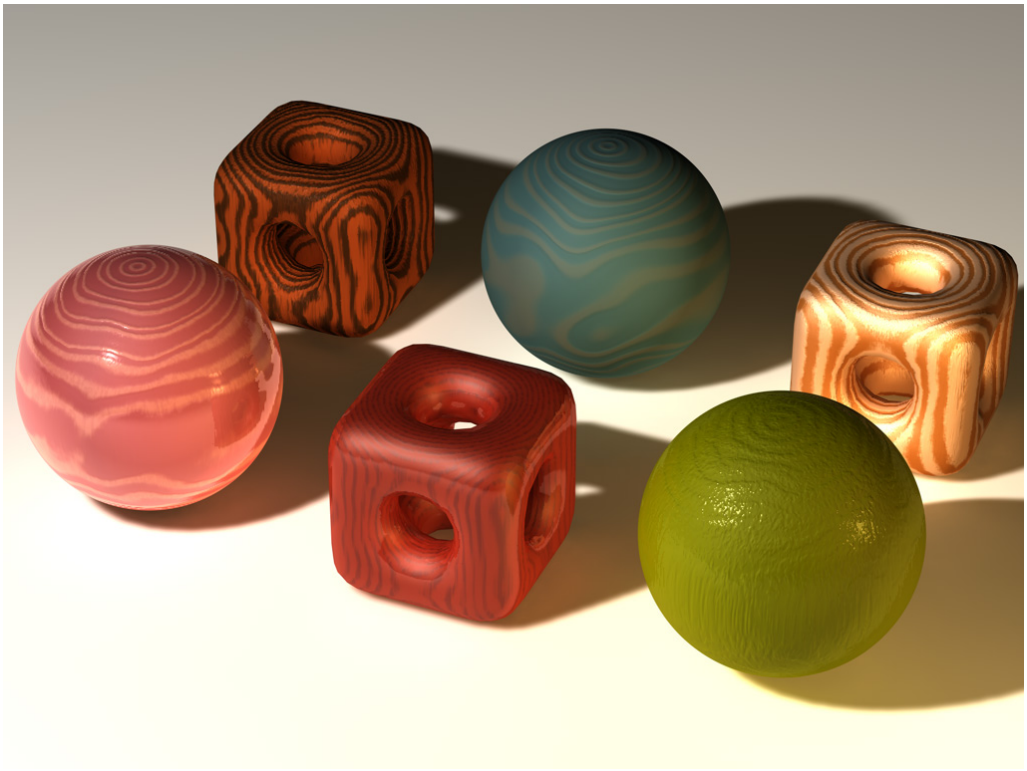
- Keyframes,
- Partikelsysteme
- Inverse Kinematics
- Bones (Skelett)

Ausgabe als

- Standbild (GIF, JPG, TIFF, ...)
- 3D-Format (VRML, DXF, Wavefront, ...)
- Filmformat (Flash, Quicktime, AVI, MPEG, ...)



Screenshot vom 3D-Werkzeug Maxon Cinema 4D



Anwendung von Texture Mapping

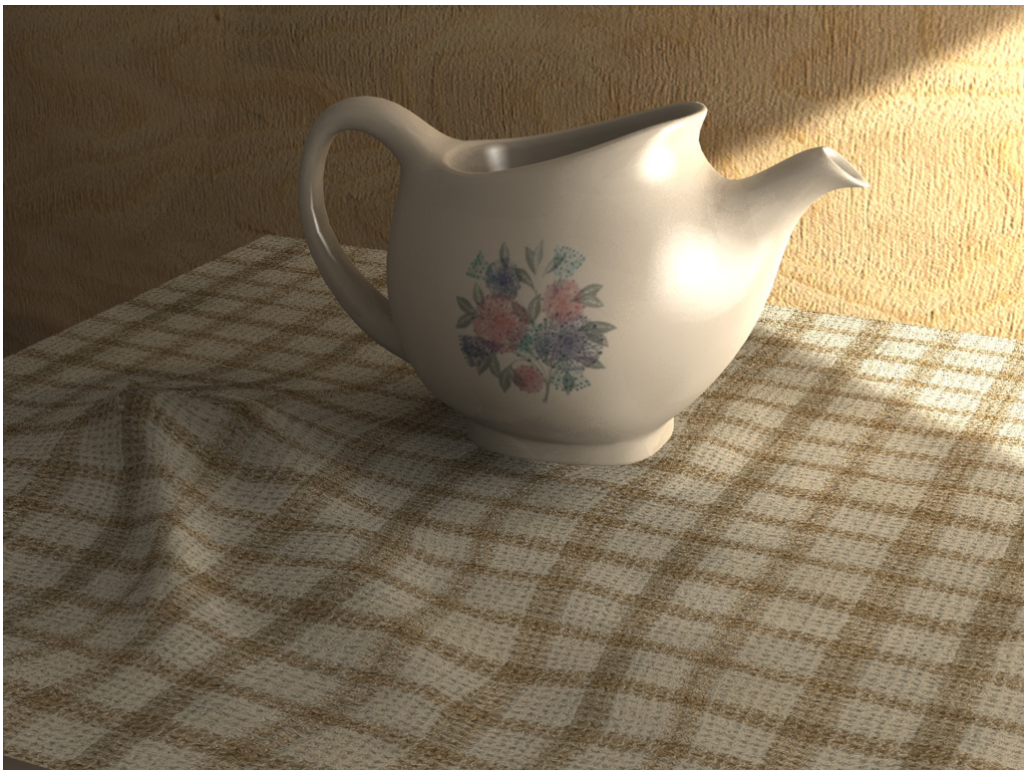


Bild gerendert mit Radiosity

Explosion erzeugt durch Deformator

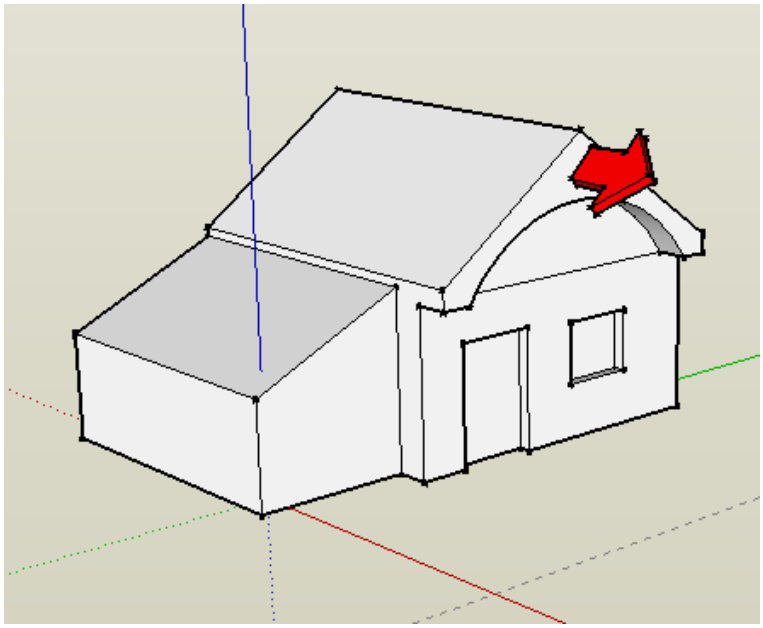
Einfluss der Schwerkraft auf ein Partikelsystems

Kapitel 26

3D im Web

26.1 Google SketchUp

Google SketchUp ist ein Werkzeug zum interaktiven Modellieren von 3D-Szenen, besonders aus dem Architekturumfeld. Die Standardversion kann kostenlos runtergeladen werden von `http://de.sketchup.com`.

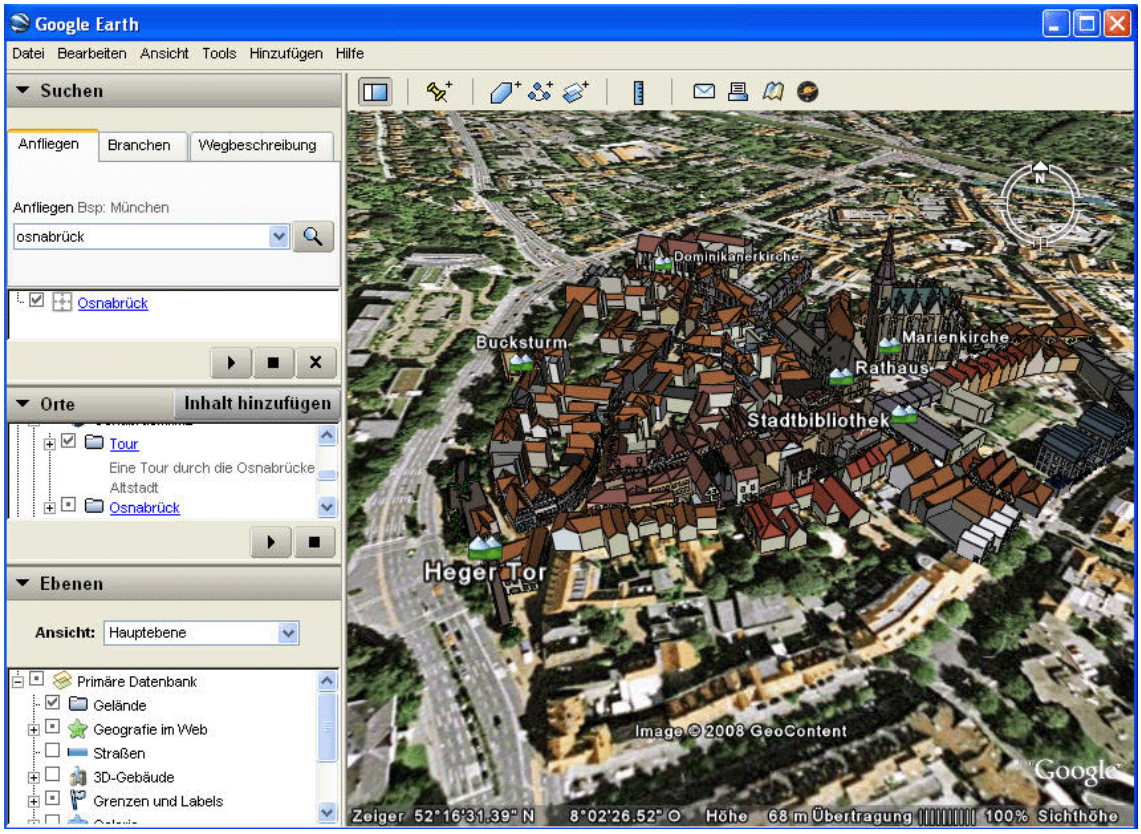


Modellierung mit Google SketchUp

Die mit SketchUp erstellten Gebäude können nach Google Earth exportiert werden, wobei die korrekte Georeferenzierung automatisch durch den von Google Earth zuvor importierten Ausschnitt der Weltkoordinaten bestimmt wird.

26.2 Google Earth

Google Earth ist ein Werkzeug zum Betrachten von Satellitenbildern mit georeferenzierten vektorisierten Straßen und 3D-Gebäuden. Die Standardversion kann kostenlos runtergeladen werden von `http://earth.google.com`.



3D-Modell der Osnabrücker Altstadt