Kapitel 1

Einführung

1.1 Motivation

- "Ein Bild sagt mehr als 1000 Worte."
- Das Auge erfaßt 40 Millionen Bits pro Sekunde.
- Lesegeschwindigkeit = 10 Worte pro Sekunde = $(10 \times 5 \times 8) = 400$ Bits pro Sekunde.
- $\Rightarrow$ Faktor 100.000.

1.2 Definition

Der Begriff *Grafische Datenverarbeitung* umfaßt

Computergrafik :
Beschreibung von Bildern
Eingabe der Beschreibung
Manipulation der Beschreibung
Ausgabe des zur Beschreibung gehörigen Bildes

Bildverarbeitung :
Verformung von Bilddaten (z.B. Drehung)
Verbesserung von Bilddaten (z.B. Kontrasterhöhung)
Vereinfachung von Bilddaten (z.B. Farbreduzierung)

Mustererkennung :
Analyse von Bilddaten
z.B. wo verläuft die Straße?
z.B. um welchen Buchstaben handelt es sich?

1.3 Anwendungen

Typische Anwendungsbereiche von Computergrafik:

- Grafische Benutzeroberflächen
  Point & Click statt Kommandos
- Business-Grafik
  Balken & Torten 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
- Medizin
  3D-Darstellung einer Computer-Tomographie
- Visualisierung im Forschungsbereich
  Darstellung von Funktionen, Darstellung von Molekülen
- 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

1.5 Ausgabegeräte

1.5.1 Soft Copy

- Refresh-Bildschirm
  Kathode emittiert Elektronen zur Anode (Phosphorschirm). Ablenkung erfolgt durch Magnetfeld. Auffrischung mit > 50 Hz.

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

- Farbbildröhre
  Schirm ist mit Tripeln von rotem, grünem, blauem Phosphor beschichtet, davor Lochplatte, 3 Kathoden

- Flach-Bildschirme

  - Plasma-Schirm
    2 Glasplatten mit horizontalen und vertikalen Leiterbahnen, dazwischen, an den Kreuzungspunkten der Leiterbahnen, Zellen mit Neongas.
  - LCD-Schirm (Liquid Crystal Display)
  - TFT-Schirm (Thin Film Transistor)

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

1.5.2 Random Scan versus Raster Scan

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

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

1.5.3 Hardcopy

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

1.6 Eingabegeräte

1.6.1 Technische Klassifizierung der Eingabegeräte

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

1.6.2 Logische Klassifizierung der Eingabegeräte

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

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

1.6.3 Eingabetechnik mit der Maus

- Objekte picken und platzieren
- Objekte picken und modifizieren (skalieren, dehnen, rotieren, ...)
- Objekte zeichnen
- eingeblendetes Gitter zur Orientierung
- *snap*: Absetzen nur an Gitterpunkten
- *Rubber Band*: Andeutung der Umrisse während des Bewegens
- horizontale oder vertikale Beschränkung beim Bewegen

Probleme bei Pick

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

1.6.4 Eingabe-Modi

Kommunikation zwischen Anwendungsprogramm und Eingabegeräten:

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

Kapitel 2

Grafische Oberflächen und ihre Programmierung

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

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

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

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

2.1 Der Window-Manager

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

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

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

Außerdem kann der Window-Manager neue Applikationen starten.

2.2 Eventhandling

Eine grafische Benutzeroberfläche muß in der Lage sein, auf zahlreiche verschiedene Ereignisse zu reagieren. Typische Events sind ein Mausklick, die Bewegung der Maus oder ein Tastendruck. Die Ereignisse betreffen aber nicht nur die Benutzereingaben, sondern auch die Interaktion zwischen Applikationen.

Z.B. ein Expose-Event wird beim Start einer neuen Applikation ausgelöst. Wird die Maus in ein Fenster gefahren, gibt es einen Enter-Event.

Die Zahl verschiedener Events wird durch Kombinationen multipliziert. Applikationen sollen häufig auf einen Doppelklick reagieren oder erst auf das Loslassen der Maustaste; die Bewegung der Maus bei gedrückter Taste wird oft zur Auswahl in Menüs verwendet.

Die unterschiedlichsten Ereignisse können zu beliebiger Zeit in beliebiger Reihenfolge auftreten. Ein gutes Anwendungsprogramm ist so geschrieben, daß es vom Benutzer gesteuert wird und nicht umgekehrt.

Das typische Schema für die Verarbeitung von Interaktionen in einem Anwendungsprogramm ist die Event-Loop. Nach dem Schema eines endlichen Automaten, der einen zentralen Wartezustand kennt, werden die Übergänge zu anderen Zuständen durch Benutzereingaben ausgelöst.

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

2.3 Eigene Applikationen

Während der Window-Manager mit einer Reihe von Events umgehen kann, die zu seiner Funktionalität gehören, muß sich der Entwickler einer grafischen Applikation mit den Ereignissen, die diese betreffen, selbst auseinandersetzen.

Soll eine Anwendung z.B. auf einen Knopfdruck reagieren, so muß der Event eines Mausklicks im Bereich der grafischen Darstellung des Buttons vom Entwickler abgehandelt werden. Dazu ist es notwendig, ein Applikationsprogramm zu schreiben, das auf dieses Ereignis hin die gewünschte Funktion ausführt, den sogenannten *Callback*.

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

die in einer Hochsprache alle notwendigen API-Vokabeln zur Beschreibung des Aufbaus und Ablaufs einer interaktiven grafischen Anwendung bereitstellt. Die Programmierung erfolgt zumeist in einer gängigen Programmiersprache, die GUI-Bibliotheken verwendet.

Unter UNIX ist die z.Zt. am meisten verbreitete Kombination X-Windows mit Xlib, Xt und dem Motif-Widget-Set. Die Programmierung erfolgt dabei meist mit Hilfe von C. Eine weitere Variante sind Bibliotheken, die auf diese Umgebung aufsetzen, wie z.B. Tk, das dann in Skript-Sprachen wie Tcl oder Perl5 verwendet werden kann.

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

Im folgenden werden anhand des X-Window-Systems typische Vokabeln der API/GUI-Programmierung dargestellt.

2.4 Das X-Window-System

2.4.1 Begriffe

Das X-Window-System wurde 1984 am MIT entworfen, seit 1987 ist es in der Version X11 verfügbar, mittlerweile als Release 6. Es werden Routinen bereitgestellt zur maschinenunabhängigen und netzwerktransparenten Formulierung von interaktiven grafischen Bedienoberflächen. Der Bildschirm wird aufgeteilt in mehrere, ggf. sich überlappende, *Fenster* (windows) mit jeweils eigenen Ein-/Ausgabefunktionalitäten. Die Fenster bilden eine baumartige Hierarchie (Kind-Fenster, Eltern-Fenster, Geschwister). Das oberste Fenster (root window) ist der Bildschirm selbst. Hierunter liegen die *Top-Level-Fenster* der einzelnen Anwendungen. Ihre Plazierung übernimmt der Window-Manager. Die Plazierung der Kinder und Enkel der Top-Level-Fenster regelt der Anwendungsprogrammierer.

X-Programme laufen auf verschiedenen Plattformen unter diversen Betriebssystemen. Das Anwendungsprogramm, genannt *X-Client*, schickt Aufträge an den *X-Server*, der die Hardware bedient.

Aus Performancegründen läuft die Client-Software häufig auf leistungsstarken Rechnern im Netz, während für den Server ein kleiner PC oder ein X-Terminal verwendet werden kann.

X-Server und X-Client können auf denselben oder auf verschiedenen Rechnern laufen. Sie kommunizieren mit Hilfe des *X-Protokolls* auf asynchrone Weise (gepuffert). Es gibt vier Arten von Nachrichten: *requests, replies, events* und *errors*.

Der X-Server hat exklusive Kontrolle über die grafischen Ausgabegeräte einschließlich Tastatur. Der Server verwaltet die *Ressourcen*, das sind komplexe Datenstrukturen für Windows, Pixmaps, Fonts, Grafik-Kontexte. Der Client referiert die Ressource über ihre ID. Dadurch wird der Kommunikationsbedarf reduziert; außerdem können mehrere Clients dieselben Ressourcen verwenden.

Der X-Client schickt Kommandos oder fragt Zustände ab. Ein Client kann Kontakt zu mehreren Servern halten. Der *Window-Manager* ist ein weiterer X-Client — und damit austauschbar. Er plaziert, verziert und ikonifiziert die Top-Level-Fenster.

Ein grafisches Ausgabegerät wird *display* genannt. Dazu gehören ein Grafik-Speicher und ggf. mehrere Bildschirme, genannt *screens*. Ein display-Name setzt sich zusammen aus

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

und kann über die Environmentvariable `DISPLAY` gesetzt werden:

```
sh% export DISPLAY=pan:0
sh% export DISPLAY=131.173.12.88:0
```

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

```
sh% xhost +kronos
```

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

2.4.2 Architektur

Das X-Window-System ist das Fenstersystem der UNIX-Welt. X-Anwendungen verwenden Routinen aus der Xlib-Bibliothek, aus den X Toolkit Intrinsics und aus einem Widget-Set.

`Xlib` enthält verschiedene Low-Level-Funktionen auf der Ebene des X-Protokolls. Z.B.

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

zeichnet auf dem Ausgabegerät `display` im Fenster `window` unter Verwendung des Grafik-Kontextes `gc` eine Linie von Punkt `x1,y1` zu Punkt `x2,y2`.

Die X Toolkit Intrinsics, kurz *Xt Intrinsics*, sind ein Aufsatz auf die Xlib. Hiermit lassen sich die grafischen Grundobjekte, die *Widgets*, kreieren und manipulieren. *Widget* ist ein aus den Begriffen *Window* und *Gadget* zusammengesetztes Kunstwort und bedeutet Dialogobjekt. Z.B.

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

besorgt von einem Widget die Breite und Höhe und überträgt diese Werte in die Variablen `breite` bzw. `hoehe`.

Das X-Window-System stellt nur die Technik zur Verwaltung von grafischen Oberflächen zur Verfügung, die Toolkits hingegen bringen Verhalten ins Spiel, ein spezielles *Look-And-Feel* der Benutzerschnittstelle.

Ein *Widget-Set* ist eine Ansammlung von aufeinander abgestimmten Widget-Klassen, die sich durch Design und Funktionalität charakterisieren. Bekannte Beispiele für Widget-Sets sind OSF/Motif, Open Look und Athena Widgets.

In Motif gibt es z.B. einen dreidimensional aussehenden Druckknopf in der Klasse `XmPushButton` oder ein Anzeigefeld mit verschiebbarem Inhalt in der Klasse `XmScrolledWindow`. Widgets werden in Motif wie folgt klassifiziert:

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

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

2.4.3 Ressourcen

Die wichtigsten Ressourcen:

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

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

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

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

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

2.5 Beispiele für Widget-Sets

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

Abbildung 2.1: Java typisches Look-And-Feel

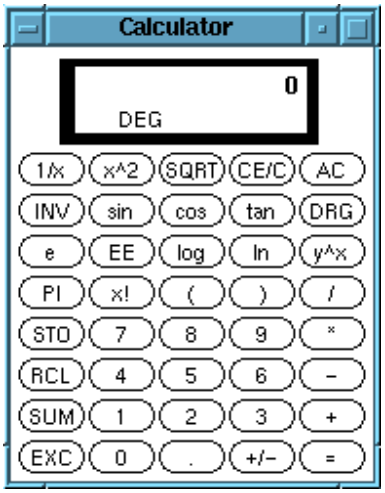

Abbildung 2.2: Athena: xcalc

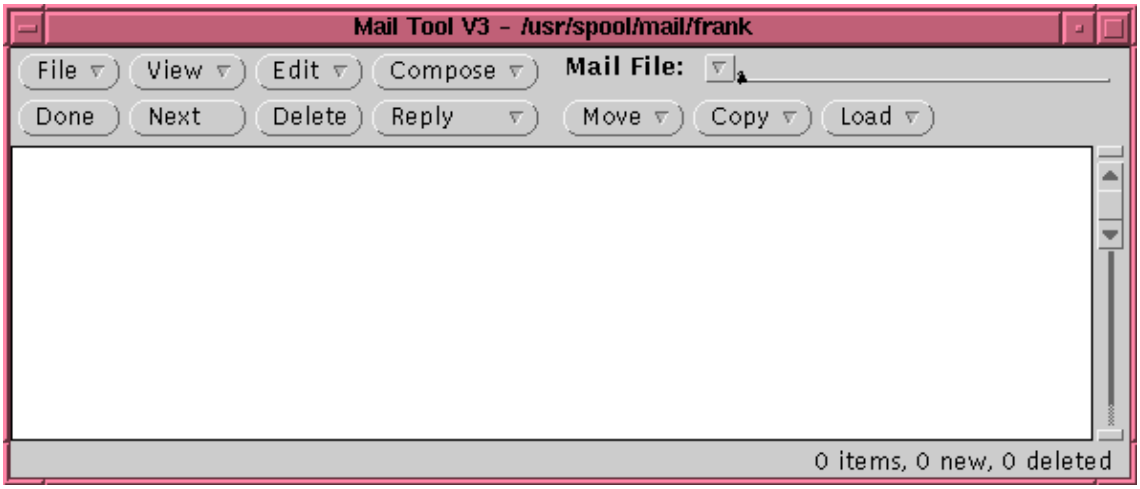

Abbildung 2.3: Open Look: mailtool

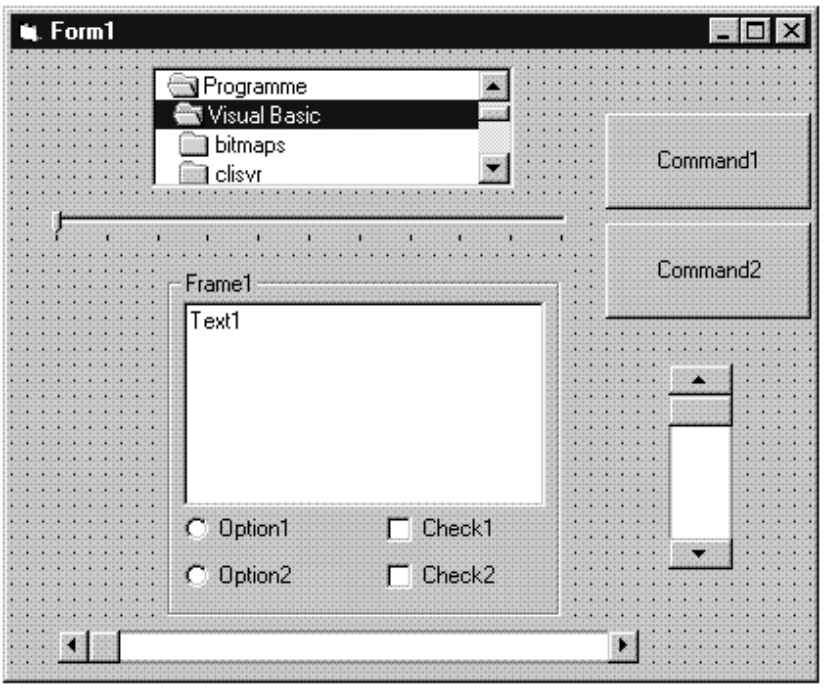

Abbildung 2.4: Windows-API mit VisualBasic

2.6 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 Benutzerschnittstelle 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 diees 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.

2.6.1 Java

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

Java wird von Sun beschrieben als

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

Java hat für die Basis-Befehle die Syntax von C übernommen und diese dann um Konstrukte zur objektorientierten Programmierung erweitert. Im Unterschied zu C++, wo ANSI-C als echte Teilmenge in der Sprache enthalten ist, wurden in Java Sprachelemente entfernt, hinzugefügt und geändert.

Ein weiterer, wesentlicher Unterschied ist, daß Java eine vollständig objektorientierte Sprache ist und keine Mechanismen besitzt, um z.B. Funktionen unabhängig von Klassen und Objekten zu implementieren.

Außerdem benutzt Java keine Zeiger, also insbesondere auch keine Funktionszeiger. Da Java keine Strukturen besitzt und Felder und Zeichenketten Objekte sind, besteht kein Bedarf für Zeiger. Java übernimmt automatisch das Referenzieren und Dereferenzieren von Objekten. Java implementiert zusätzlich eine Garbage Collection zur Speicherverwaltung.

Ein Beispiel

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

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

Dieses Programm besteht, wie jedes Java-Programm, aus einer öffentlichen Definition. Die Klasse enthält eine Methode mit dem Namen `main()`, in der der Interpreter mit der Programmausführung beginnt.

Dieses Programm muß in einer Datei gespeichert werden, die den Namen der öffentlichen Klasse, gefolgt von der Erweiterung `.java`, besitzt. Die Übersetzung geschieht mit:

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

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

```
sh% java HelloWorld
```

Ein Programm in Java besteht im allgemeinen aus einer oder mehreren Klassendefinitionen, von denen jede in ihre eigene `.class`-Datei in *Java Virtual Machine*-Code übersetzt wird. Eine dieser Klassen muß eine `main()`-Methode definieren, in der das Programm seine Ausführung beginnt.

Der Java-Compiler erzeugt Bytecode statt direkten Maschinencode. Dieser Bytecode stellt ein architekturunabhängiges Objektdateiformat zur Verfügung, d.h., der Code ist dazu entworfen worden, Programme effizient auf verschiedene Plattformen zu transportieren. Ein Java-Programm kann auf jedem System laufen, das einen Java-Interpreter und ein Laufzeitsystem implementiert. Zusammengenommen implementieren beide die Java Virtual Machine.

2.6.2 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-On-Line-Dokumentation zu entnehmen.

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

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

Die Klassen des swing-Pakets lassen sich grob in drei Gruppen einteilen: Grafikklassen, in denen grafische Objekte wie Farben, Fonts, Bilder beschrieben werden; Komponenten, d.h. Klassen, die GUI-Komponenten wie z.B. Buttons, Menüs und Textfelder zur Verfügung stellen; sowie Layout-Manager, die die Anordnung von GUI-Komponenten in einer Applikation kontrollieren. Aus den verwandten Paketen sollen hier noch die Event-Listener besprochen werden, mit deren Hilfe die Interaktion zwischen User und Applikationen implementiert werden muß. Die folgende Übersicht erhebt keinen Anspruch auf Vollständigkeit, sondern beschreibt nur kurz die für Computergrafik-Programme wichtigen Klassen. Weitere Klassen sowie alle Details sind der entsprechenden On-Line-Dokumentation zu entnehmen.

Die Grafik-Klassen

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

Die GUI-Komponenten

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

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

- *Top-Level-Container*, wie `JFrame`, `JDialog` und `JApplet`, die jeweils ein eigenstaendiges "Fenster" erzeugen, das direkt in die grafische Benutztungsoberfläche eingeblendet und

vom Window-Manager verwaltet wird.

- *General-Purpose Container*, wie `JPanel` und `JScrollPane`, die in erster Linie beliebige andere Componenten aufnehmen und deshalb nur eingeschränkte eigene Funktionalität bieten.
- *Special-Purpose Container*, wie `JInternalFrame` und `JLayeredPane`, die eine festgelegte Rolle in der Benutzerschnittstelle spielen und deshalb nur eingeschränkt konfgurierbar sind.

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

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

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

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

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

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

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

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

Die Layout-Manager

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

- `FlowLayout`: Dies ist das einfachste Layout, welches bei `JPanel` voreingestellt ist. Jede Komponente wird in seiner bevorzugten Größe rechts von der vorigen Komponente eingefügt. Ist der rechte Rand des Containers erreicht, wird eine weitere Zeile angehängt, in der weitere Komponenten eingefügt werden können.

- `BorderLayout`: Dies ist ebenfalls ein einfaches Layout, das als Default-Layout bei `JFrame` verwendet wird. Es besteht aus den fünf Gebieten, Nord, Süd, Ost, West und Zentrum, in denen jeweils beliebige Komponenten plaziert werden können. Im Gegensatz zum `FlowLayout` werden hier die Komponenten an die Größe des Containers angepaßt: Die Nord- und Süd-Komponenten werden in X-Richtung gestreckt bzw. gestaucht; die Ost- und West-Komponenten in Y-Richtung und die Zentrums-Komponente in beiden Richtungen.

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

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

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

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

Event-Handling

Eine offene Frage ist nun noch, wie das Aktivieren einer GUI-Komponente eine entsprechende Aktion im Benutzer-Programm auslöst. Dazu gibt es in AWT folgendes Konzept: Eine Aktion des Benutzers — wie z.B. ein Mausklick — erzeugt in der Java-Maschine einen *Event*. Je nach Art der Aktion und des Kontextes, in dem sie stattfindet, ergeben sich verschiedene

Event-Typen, die wiederum verschiedene Argumente haben können. Eine Komponente kann für sie geeignete Events empfangen, indem ein sogenannter *Event-Listener* mit der Komponente verbunden wird. Für die verschiedenen Event-Typen gibt es verschiedene Listener, von denen einer oder mehrere bei einer Komponente eingetragen werden können. Ein Listener ist dabei ein Interface, in dem alle Methoden vorgegeben werden, die für eine Menge von zusammengehörigen Events notwendig sind. Tabelle 2.1 gibt eine Übersicht der verschiedenen Event-Listener:

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

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.

2.7 Swing-Beispielapplikation

In diesem Abschnitt wird eine einfache Applikation gezeigt, die die Worte "Hello World!" in einem eigenen Fenster darstellt (siehe Abbildung 2.1).

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

```
package HelloWorld;

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

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

    add(l, BorderLayout.CENTER);
  }
}
```

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

In `main` wird nur der Konstruktor aufgerufen. Dort wird zunächst versucht die Applikation im javatypischen Look-And-Feel darzustellen. Falls die Wahl des Look-And-Feel mißlingt, wird das Look-And-Feel der zugrundeliegenden Benutzungsoberfläche realisiert. Anschließend wird ein Fensterobjekt `JFrame` erzeugt und dem Fensterbereich, der für die Darstellung von GUI-Elementen zur Verfügung steht (*Content Pane*), eine Instanz der Klasse `Hello` hinzugefügt. Abschließend wird die Größe der Applikation mit der Methode `pack` berechnet und das Fenster mit der Methode `setVisible` auf der grafischen Benutzungsoberfläche sichtbar gemacht. Mit dem Aufruf der Methode `addWindowListener(this)` wird dem Fenster mitgeteilt, daß es selber auf Events reagieren soll, die mit dem Öffnen, Schließen, (De)Aktivieren und (De)Ikonifizieren des Fensters zu tun haben.

Die Klasse `HelloApp` muß deshalb das Interface `WindowListener` adaptieren. Der Code enthält die sieben entsprechenden Methoden; die Rümpfe bleiben aber bis auf den von `windowClosing` leer, da sie hier nicht benötigt werden. Die Methode `windowClosing` wird aufgerufen, wenn

der User auf die "Fenster schließen"-Schaltfläche des Applikationsfensters geklickt hat. Durch Aufruf von `System.exit` wird die Applikation sofort beendet.

```
package HelloWorld;

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

public class HelloApp implements WindowListener {
  /** Erzeugt eine Instanz der Klasse Hello. */
  public HelloApp() {
    try {                                        // Versuche Look & Feel so zu
      UIManager.setLookAndFeel(                  // setzen, dass es auf allen
         // Systemen gleich aussieht
        UIManager.getCrossPlatformLookAndFeelClassName());
    } catch(Exception e) {};                     // sonst verwende default L&F

    JFrame frame = new JFrame("Hello");          // FensterObjekt erzeugen
                                                 // Instanz der Klasse Hello
                                                 // erzeugen und hinzufuegen
    frame.getContentPane().add(new Hello(), BorderLayout.CENTER);
    frame.addWindowListener(this);               // DIES Objekt ist fuer Window-
                                                 // Events zustaendig
    frame.pack();                                // anzeigbar machen und
                                                 // Fenstergroesse berechnen
    frame.setVisible(true);                      // Fenster sichtbar machen
  }

  /** Die folgenden Methoden gehoeren zum Interface WindowListener.
   *  Der Rumpf bleibt bei allen Methoden (bis auf windowClosing) leer. */
  public void windowActivated(WindowEvent e) { }
  public void windowClosed(WindowEvent e) { }
  public void windowDeactivated(WindowEvent e) { }
  public void windowDeiconified(WindowEvent e) { }
  public void windowIconified(WindowEvent e) { }
  public void windowOpened(WindowEvent e) { }
  /**
   * Sorgt dafuer, dass die Applikation beendet wird, wenn die entsprechende
   * Schaltflaeche im Rahmen des Fensters gedrueckt wurde.
   * @param e der ausgeloeste Event
   */
  public void windowClosing(WindowEvent e) {
    System.exit(0);                              // Programm sofort beenden
  }

  public static void main(String args[]) {
    new HelloApp();                              // Instanz erzeugen
  }
}
```

2.8 Java-Applet

Die Trennung zwischen Darstellung des Fensters und der Verwaltung/Darstellung des Fensterinhaltes in zwei Klassen bietet die Möglichkeit den Darstellungsrahmen zu verändern, z.B. indem die Klasse `HelloApp` durch eine dritte Klasse `HelloApplet` ersetzt wird. Diese erzeugt kein Fenster (`JFrame`) sondern ein `JApplet`, das in HTML-Dokumente eingebunden werden kann und wieder den Rahmen für die eigentlichen GUI-Elemente bildet:

```
package HelloWorld;

import java.awt.BorderLayout;                    // Importliste aller aus
import javax.swing.UIManager;                    // AWT u. Swing-Komponenten
import javax.swing.JApplet;                      // benoetigten Klassen

public class HelloApplet extends JApplet {
  /** Diese Methode wird vom Broswer beim Start des Applets aufgerufen */
  public void init() {
    try {                                        // versuche Look & Feel so zu
      UIManager.setLookAndFeel(                  // setzen, dass es auf allen
                                                 // Systemen gleich aussieht
        UIManager.getCrossPlatformLookAndFeelClassName());
    } catch(Exception e) {};                     // sonst verwende System L&F
                                                 // Instanz der Klasse Hello
                                                 // erzeugen und einfuegen
    getContentPane().add(new Hello(), BorderLayout.CENTER);
  }
}
```

Beim Start des Applets wird die Methode `init` aufgerufen, in der zunächst wieder versucht wird, das typische Java Look-And-Feel einzustellen. Anschließend wird eine Instanz der Klasse `Hello` zum GUI-Bereich des Applets hinzugefügt. Da die Größe des Applets im zugehörigen HTML-Dokument festgelegt und die Sichtbarkeit vom HTML-Betrachtungsprogramm geregelt wird sind keine weiteren Angaben erforderlich. Es muß kein WindowListener implementiert werden, da das Applet kein Fenster ist und ihm die entsprechenden Schaltflächen fehlen, wie im folgenden Beispiel zu sehen:

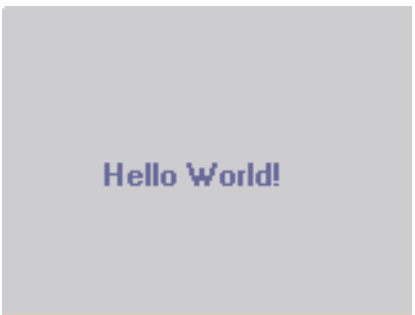

Abbildung 2.5: Screenshot des Hello-Applets

Kapitel 3

2D-Grundlagen

3.1 Koordinatensysteme

Die Angabe eines Punktes geschieht meistens durch kartesische Koordinaten x, y.

Eine Folge von Punkten kann absolut oder relativ spezifiziert werden:

Absolut: Jeder Punkt der Folge wird durch seine kartesischen Koordinaten beschrieben.

Relativ: Jeder Punkt der Folge wird unter Verwendung einer Windrose (Abbildung 3.1) relativ zum Vorgänger der Folge beschrieben, entweder mit absoluten Richtungscodes oder mit relativen Richtungscodes.

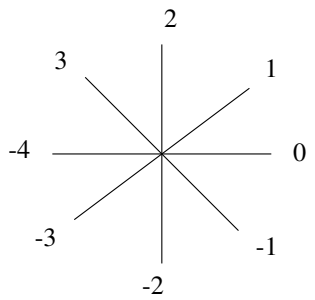

Abbildung 3.1: Windrose

Eine Verwendung von absoluten Richtungscodes ergibt für die Figur in Abbildung 3.2 die Folge $0, 2, 1, 2, 2, 0, -2, -2, -1, -2$. Der relative Richtungscode legt nach jedem Schritt die momentane Richtung als "0" fest. Es ergibt sich für die Figur in Abbildung 3.2 die Folge $0, 2, -1, 1, 0, -2, -2, 0, 1, -1$.

Zur Platzeinsparung werden die "wahrscheinlicheren" Richtungsänderungen mit weniger Bits codiert. Tabelle 3.1 zeigt eine mögliche Codierung, basierend auf den Wahrscheinlichkeiten in Spalte 1. Der zu diesen Wahrscheinlichkeiten konstruierte Huffman-Code wird mit Hilfe des Baumes in Abbildung 3.3 ermittelt. Für die Figur in Abbildung 3.2 entstehen dann statt $(10 \times 3) = 30$ Bits nur noch 26 Bits.

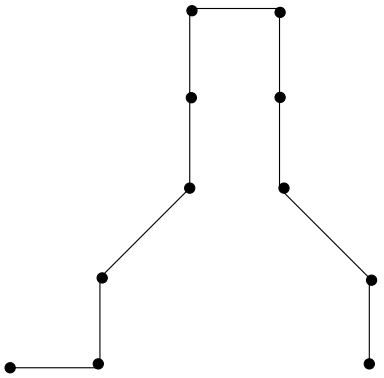

Abbildung 3.2: Beispiel für Streckenzug

%	Richtung	Code	Bit-Länge
28	0	00	2
26	1	01	2
24	-1	10	2
7	2	1100	4
6	-2	1101	4
4	3	1110	4
3	-3	11100	5
2	-4	11111	5

Tabelle 3.1: Relative Richtungscodes

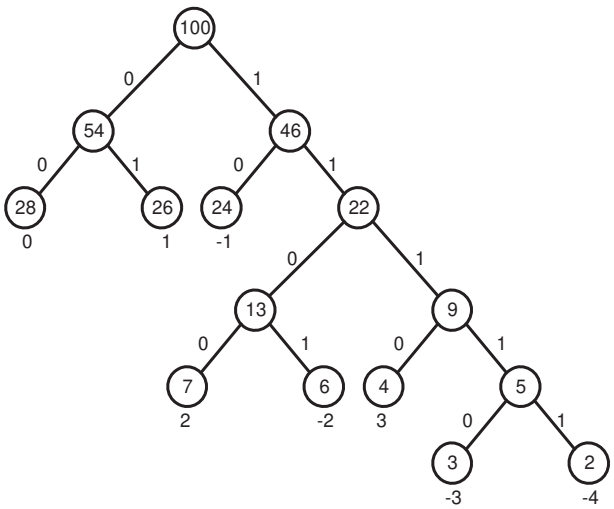

Abbildung 3.3: Code-Baum basierend auf Wahrscheinlichkeiten

3.2 Linie

- Gegeben sind Anfangs- und Endkoordinaten einer Linie.
- Zu berechnen sind die "anzuschaltenden" Pixel.

Eine erste, offensichtliche Möglichkeit verwendet die Geradengleichung $y = s \cdot x + c$:

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

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

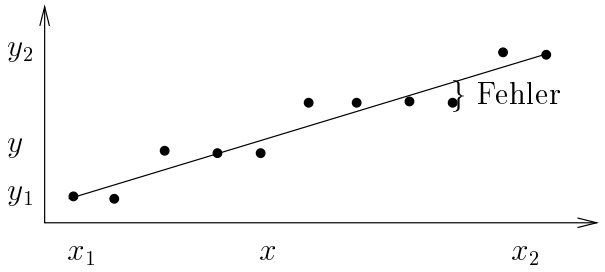

Abbildung 3.4: Fehler als Abweichung von der Ideal-Linie

Der resultierende Fehler pro Position (Abweichung des errechneten Punktes von der Ideal-Linie) wird in Abbildung 3.4 ersichtlich.

Unter Verwendung einer Klasse *Point* mit den Integer-Koordinaten `x` und `y` entsteht:

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

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

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

Das Verfahren hat zwei Nachteile:

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

Als Lösung bietet sich der Bresenham-Algorithmus an, der den jeweils nächsten y-Wert aus dem vorherigen berechnet und dabei den momentanen Fehler nachhält.

Zunächst betrachten wir Geraden im 1. Oktanten (d.h. Steigung < 1):

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

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

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

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

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

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

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

            }
    }
```

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

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

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

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

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

Abbildung 3.5: Vom Bresenham-Algorithmus erzeugte Linien

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

void bresenham_linie(Point P, Point q)                  // zeichnet Linie von P nach q
{
        Point p = new Point(P);
        int error, delta, schwelle, dx, dy, inc_x, inc_y;

        dx = q.x - p.x;
        dy = q.y - p.y;

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

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

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

        else                                // steil nach oben oder steil nach unten

        {

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

        set_pixel(q);

}
```

Bresenham-Algorithmus

3.2.1 Antialiasing

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

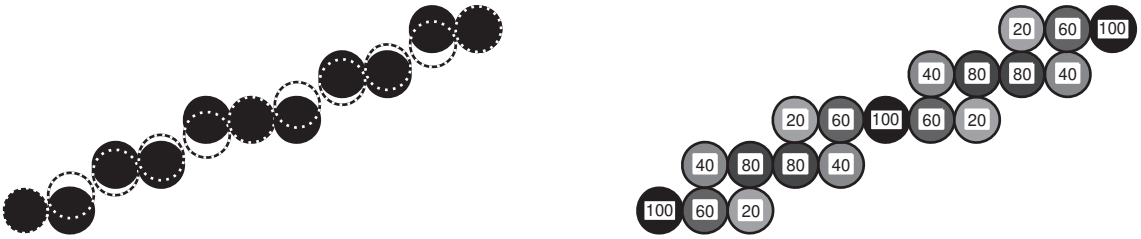

Abbildung 3.6: (a) Bresenham-Linie und Ideal-Linie (b) Resultierende Grauwerte

In Abbildung 3.6 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.

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

3.3 Polygon

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

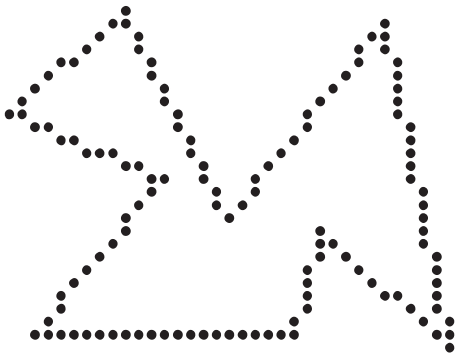

Abbildung 3.7: Polygon

3.4 Kreis

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

Betrachte zunächst den Verlauf eines Kreises um $(0, 0)$ im 2. Oktanten (Abbildung 3.8).
Für einen Punkt (x, y) sei $F(x, y) = x^2 + y^2 - r^2$

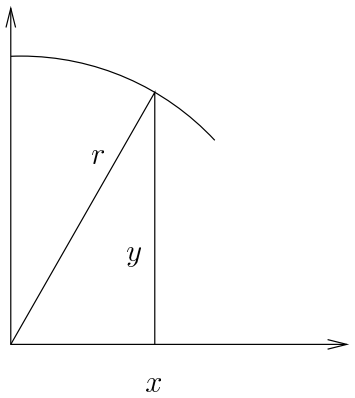

Abbildung 3.8: Verlauf des Kreises im 2. Oktanden

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 3.9).

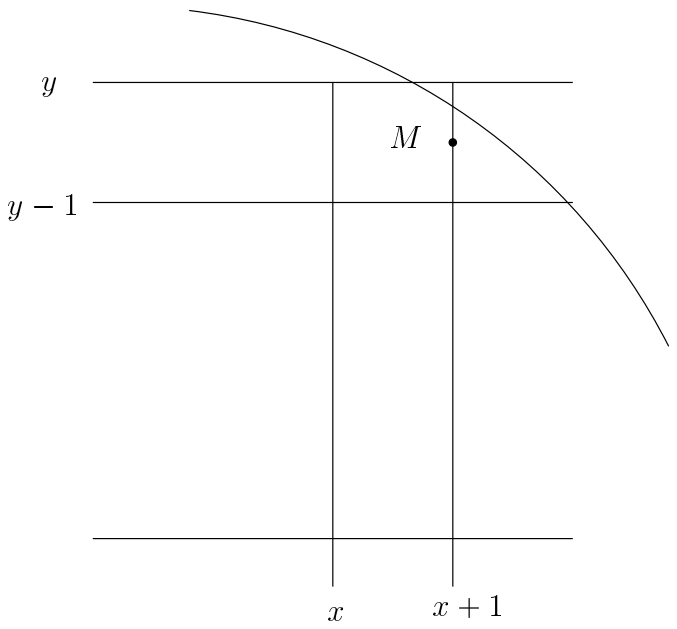

Abbildung 3.9: Testpunkt M für Entscheidungsvariable

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

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

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

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

Also ergibt sich:

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

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

        while (y>=x) {

                set_pixel (new Point(x,y));

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

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

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

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

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

                set_pixel (new Point(x,y));

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

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

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

tief.

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

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

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

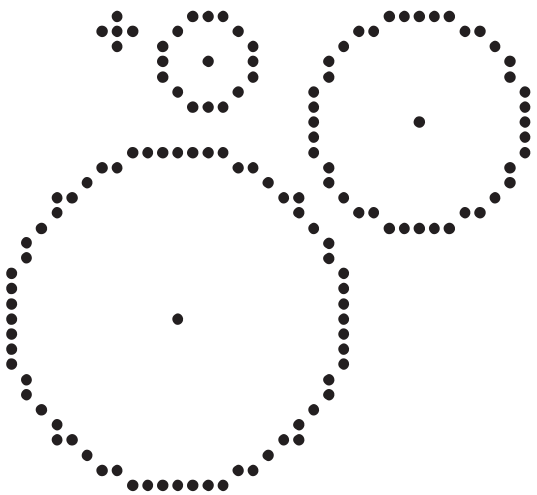

Abbildung 3.10: Vom Bresenham-Algorithmus erzeugte Kreise

Kapitel 4

2D-Füllen

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

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

4.1 Universelle Füll-Verfahren

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

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

Abbildung 4.1: Nachbarschaften bei universellen Füllverfahren

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

Vorteil: beliebige Umrandung möglich

Nachteil: hoher Rechen- und Speicherbedarf

Obacht:

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

Bild 4.2 zeigt beide Effekte.

Unerreichtes Gebiet beim 4-way-stepping

Farbauslauf beim 8-way-stepping

Abbildung 4.2: Probleme bei universellen Füllverfahren

Benötigt werden

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

und

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

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

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.

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

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

                set_pixel(p);                    // setze Vordergrundfarbe

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

        }
}




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

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

                del_pixel(p);                   // setze Hintergrundfarbe

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

        }
}
```

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

  private void fillRowByRow(int x, int y, Graphics g) {
    int lg;                                     // nicht gesetzte Pixel ganz
    int rg;                                     // links/rechts in dieser Z.
    Punkt hilf;                                 // Hilfpunkt
    int px = x;                                 // lokale Kopie

    while(!isPixelSet(x, y)) {                  // Solange Pixel ungesetzt
      setPixelInBuf(x, y);                      // Pixel merken und setzen
      hilf = new Punkt(x, y);
      hilf.paint(g);
      x--;                                      // naechstes Pixel links
    }
    lg = x+1;                                   // 1 zu weit gelaufen

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

    while(!isPixelSet(x, y)) {                  // Solange Pixel ungesetzt
      setPixelInBuf(x, y);                      // Pixel merken und setzen
      hilf = new Punkt(x, y);
      hilf.paint(g);
      x++;                                      // naechstes Pixel rechts
    }
    rg = x-1;                                   // 1 zu weit gelaufen

    for(int pos = rg; pos >= lg; pos--) {       // von rechts nach links
      if(!isPixelSet(pos, y - 1)) {             // falls Pixel in Reihe ueber
                                                // dieser nicht gesetzt:
        fillRowByRow(pos,y-1, g);               // Repraesentant! neuer Aufruf
      }
      if(!isPixelSet(pos, y + 1)) {             // falls Pixel in Reihe unter
                                                // dieser nicht gesetzt:
        fillRowByRow(pos,y+1, g);               // Repraesentant! neuer Aufruf
      }
    }
    return;
  }
```

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 4.3 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 $(x_1, x_2), (x_3, x_4), \ldots$

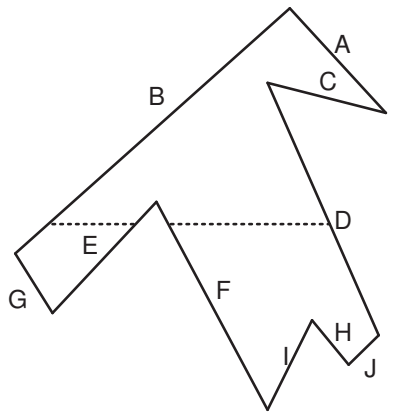

Abbildung 4.3: Polygon mit Scanline

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.

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

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 4.4).

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.

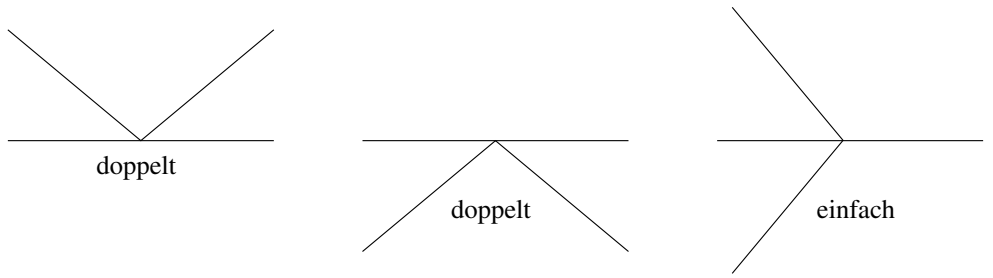

Abbildung 4.4: Fallunterscheidungen beim Berechnen der Schnittpunkte

Kohärenz-Eigenschaft

Abbildung 4.5 zeigt, wie die Schnittpunkte für Scan-Line y_{i+1} sich mit Hilfe der Schnittpunkte von Scan-Line y_i bestimmen lassen.

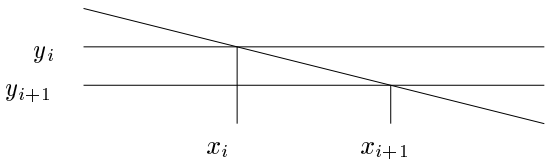

Abbildung 4.5: Fortschreiben der errechneten Schnittpunkte

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

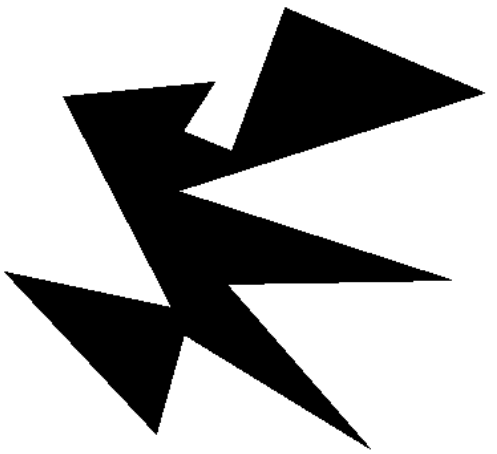

Abbildung 4.6: Vom Scanline-Algorithmus gefülltes Polygon

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

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

public class Edge {

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

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

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

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

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

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

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

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

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

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

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

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

        public Edge() {

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

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

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

        Point P2 = new Point(p2);

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

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

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

        Edge edge1 = edges;                                           // Hilfsobjekt

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

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

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

        compare_y = Points[k].y;

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

        return(new_y);
}

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

        Edge edge1 = new Edge();

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

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

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

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

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

        return(l_act_edge);
}


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

        edge2 = edges.next;

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

        } while (edge2 != l_act_edge);

        return (l_act_edge);
}
```

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

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

        } while (edges != l_act_edge);
}

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

        prev_edge = edges;

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

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

        Edge edges = new Edge();

        bottom_y = Edge_Sort(NumPoints, Points, edges);

        l_act_edge = edges.next;

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

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

4.3 Dithering

Eine $n \times n$-Dithermatrix M ist mit den n^2 Zahlen zwischen 0 und $n^2 - 1$ besetzt. Zum Färben einer Fläche mit Grauwert $k, 0 \leq k \leq n^2$ werden alle Pixel (i, j) gesetzt mit $M[i \mod n, j \mod n] < k$. Abbildung 4.7 zeigt das Füllmuster zum Schwellwert 7.

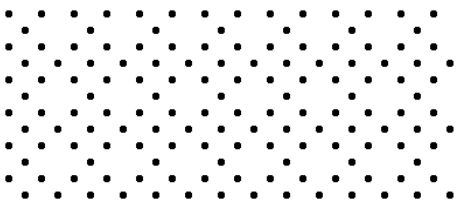

Abbildung 4.7: Zum Schwellwert 7 gehörender Grauwert

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

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

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

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

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

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

int grauwert(                                   // liefert den Grauwert
             int i, int j )                     // an Position i, j
{
  return(dit_mat[i][j]);
}
```