

B3. Digitale Bildbearbeitung mit Java

B3.1 Bildbearbeitungsfunktionen in Java 2D

B3.2 Java Advanced Imaging API



Literatur:

J. Knudsen: Java 2D Graphics, O'Reilly 1999, Kap. 10
<http://java.sun.com/products/java-media/jai/>

Wozu Bildverarbeitung mit Programmen?

- Bildbearbeitung für Medienzwecke
 - Zeitungen, Zeitschriften, Werbeindustrie, Gebrauchsgrafik, ...
- Bildanalyse zur Objektverfolgung
 - Optische Maus, Kameranachführung, Objekt- und Werkzeugpositionierung in der Fertigung, ...
- Bildauswertung
 - Satellitenbildanalyse (Wetter, militärische Aufklärung, ...)
 - Geodatenverarbeitung (z.B. Luftbilder mit Landkarten abgleichen)
- Verkehrstelematik
 - Verkehrsüberwachung (z.B. Mautsysteme)
- Computerspiele
- ...

Digitale Bildbearbeitung

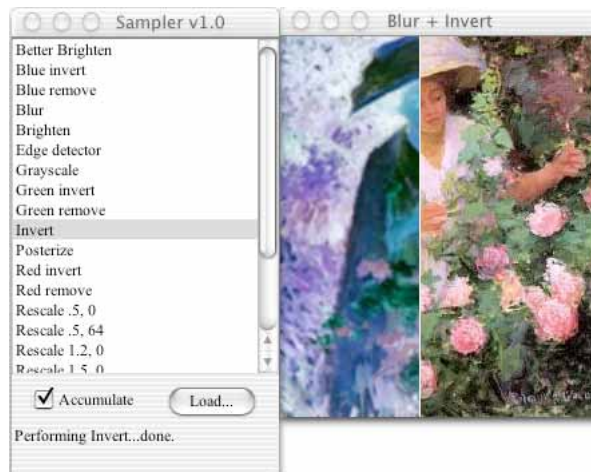
- Bilder aus der Sicht der Informatik:
 - spezielle Datenstruktur (insbesondere: 2-dimensionales Array)
 - Bearbeitung mit verschiedenen Algorithmen möglich
- *Filter*:
 - Ursprünglich Begriff aus der klassischen (analogen) Fotografie
 - Generell: Operation, die Bild in Bild transformiert
 - Klassische (physikalische) Filter:
 - » Polarisationsfilter, UV-Filter
 - » Weich-/Scharfzeichnung
 - » Helligkeits-, Farbfilter
 - » Effektfiler (z.B. Sterneffekt, Kachelung)
 - Bildbearbeitungsprogramme bieten Vielzahl von (Software-) "Filtern"
 - » Bsp. Adobe Photoshop, Gimp

Bildbearbeitung in Java

- Frühe Java-Versionen:
 - In AWT Einlesung und Anzeigen von Bildern unterstützt
 - Noch keine Funktionen zur Modifikation von Bildern
- Java 2D:
 - Bild als Bestandteil der Rendering-Kette
 - Begrenzter Satz von Bildbearbeitungsfunktionen
- Java Advanced Imaging (JAI):
 - Erste Version November 1999, Version 1.1: 2001, aktuell: 1.1.2 (Sept. 2004)
 - Ausgefeilte, hochleistungsfähige Bildbearbeitungsfunktionen
 - Folgt konsequent dem Java-Prinzip "Write once, run everywhere"
- Performance:
 - In diesem Bereich nach wie vor das Hauptproblem der Java-Plattform
 - C- und C++-Programme deutlich überlegen

Beispiel: Bildbearbeitung mit Java 2D

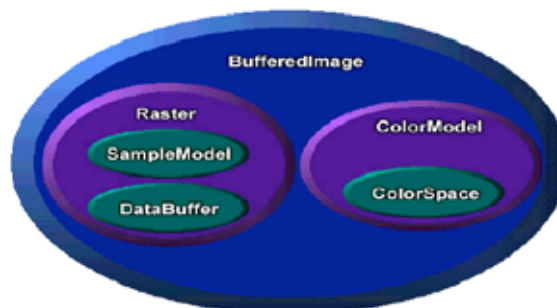
- aus: Knudsen, Kapitel 10



Ethel with Roses, Edmund Greacen, 1907

Java 2D: BufferedImage

- `java.awt.image.BufferedImage`:
 - Repräsentation eines Bildes
 - Verkapselt (d.h. versteckt Details von):
 - » Farbmodell
 - » Abtastung
 - » Datenpuffer



Double Buffering

- Einfache Pufferung:
 - Verwendet einen internen Pufferspeicher in Größe des Bildes
 - Bei Änderung des Bildes wird Bild gelöscht und neues Bild angezeigt
 - kann zu flackerndem Bild führen
 - führt oft unnötig viele Anzeigevorgänge durch
- Doppel-Pufferung:
 - Verwendet zwei Speicher:
 - » *onScreen*- und *offScreen*-Fassung
 - Neues Bild wird zuerst *offScreen* aufgebaut
 - Angezeigtes Bild wird in einem Schritt durch neues Bild ersetzt
- Java 2D und Swing unterstützen an vielen Stellen automatisch Doppel-Pufferung

Einlesen von Bilddateien in Java

- Einlesen von Bilddateien umfasst komplexe Algorithmen
 - Decodieren des Bildformats
 - Einlesen lokal aus Datei oder über eine URL
 - Berücksichtigung von langsamen Festplatten- und Netzzugriffen
 - » Observer-Modell: Anzeigefunktion wird wieder aufgerufen, wenn Daten nachgeladen sind
- Java: Diverse Möglichkeiten zum Laden eines Bilds
 - Standard-AWT-Methode (*MediaTracker*)
 - Swing-Methode (*ImageIcon*)
 - Spezielle Codecs (Sun-JPEG-Codec meist in Standardinstallation enthalten)
 - Java Advanced Imaging (siehe unten)

Einlesen einer JPEG-Datei mit Spezial-Codec

```
import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.*;
import com.sun.image.codec.jpeg.*;

public class LoadJpeg extends Frame {
    private static BufferedImage mImage;

    public static void main(String[] args) throws Exception {
        Frame f = new LoadJpeg("yinmao.small.jpg");
        f.setSize(mImage.getWidth(), mImage.getHeight());
        f.setLocation(200, 200);
        f.setVisible(true);
    }

    public LoadJpeg(String filename)
        throws IOException, ImageFormatException {
        // Load the specified JPEG file.
        InputStream in = new FileInputStream(filename);
        JPEGImageDecoder decoder = JPEGCodec.createJPEGDecoder(in);
        mImage = decoder.decodeAsBufferedImage();
        in.close();
    }

    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.drawImage(mImage, 0, 0, null);
    }
}
```

Einlesen einer Bilddatei mit AWT MediaTracker

```
import java.awt.*;

public class LoadJpeg1 extends Frame {
    private static Image img;

    public static void main(String[] args) {
        Frame f = new LoadJpeg1();
        MediaTracker tracker = new MediaTracker(f);
        img = Toolkit.getDefaultToolkit().getImage("yinmao.small.jpg");
        tracker.addImage(img, 0);
        try {
            tracker.waitForAll();
        } catch (InterruptedException e) {
            throw new Error("Loading problem");
        }
        f.setSize(img.getWidth(null), img.getHeight(null));
        f.setLocation(200, 200);
        f.setVisible(true);
    }

    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.drawImage(img, 0, 0, null);
    }
}
```

Einlesen einer Bilddatei als Swing-Icon

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

public class LoadJpeg2 extends Frame {

    private static Image img;

    public static void main(String[] args) {
        Frame f = new LoadJpeg2();
        ImageIcon icon = new ImageIcon("yinmao.small.jpg");
        img = icon.getImage();
        f.setSize(img.getWidth(null),img.getHeight(null));
        f.setLocation(200, 200);
        f.setVisible(true);
    }

    public void paint(Graphics g) {
        Graphics2D g2 = (Graphics2D)g;
        g2.drawImage(img, 0, 0, null);
    }
}
```

Java 2D: Bildbearbeitungsfunktionen

- Bildbearbeitungsfunktionen (in Java 2D):
 - Schnittstelle `java.awt.image.BufferedImageOp`
- ```
public BufferedImage filter
(BufferedImage src, BufferedImage dst)
```
- Bearbeitet `src`, mit genauer zu definierendem Algorithmus
  - Liefert bearbeitetes Bild als Resultat
  - `dst` ermöglicht Angabe eines Speicherbereichs für das Ergebnis
    - » Falls `dst = null`: neues Bild erzeugt
    - » `dst = src`: Operation "auf der Stelle" ausgeführt (*in place*)
- Operationen werden als Objekte erzeugt
    - Entwurfsmuster "Strategy" (Gamma et al.)
    - Ausführung:
      - » Entweder bei Übergabe an `drawImage()`
      - » oder durch Aufruf von `filter()` aus dem Objekt

## Java 2D: Verwendung vordefinierter Operationen

- Beispiel: Konversion in Graustufen

```
public static BufferedImage convertToGrayscale
(BufferedImage source) {
 BufferedImageOp op =
 new ColorConvertOp(
 ColorSpace.getInstance(ColorSpace.CS_GRAY), null);
 return op.filter(source, null);
}
```



## Java 2D: Einfaches Rahmenprogramm für Operationen

```
import java.awt.*;
import java.awt.geom.*;
import java.awt.color.*;
import java.awt.image.*;
import java.io.*;
import com.sun.image.codec.jpeg.*;

public class GrayJpeg extends Frame {

 private static BufferedImage mImage;

 ... Einlesen wie in obigem Beispiel

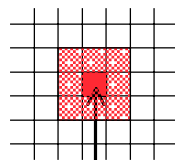
 public void paint(Graphics g) {
 Graphics2D g2 = (Graphics2D)g;
 BufferedImageOp op =
 new ColorConvertOp(ColorSpace.getInstance
 (ColorSpace.CS_GRAY), null);
 g2.drawImage(mImage, op, 0, 0);
 }
}
```

## Vordefinierte Operationen in Java 2D

| Klasse                          | Hilfsklassen                                                                                  | Effekte                                           | "in place"?<br>(src = dst) |
|---------------------------------|-----------------------------------------------------------------------------------------------|---------------------------------------------------|----------------------------|
| <code>ConvolveOp</code>         | <code>Kernel</code>                                                                           | Weich- und Scharfzeichnen, Kantenerkennung        | nein                       |
| <code>Affine TransformOp</code> | <code>java.awt.geom.AffineTransform</code>                                                    | Geometrische Transformationen                     | nein                       |
| <code>LookupOp</code>           | <code>LookupTable</code> ,<br><code>ByteLookupTable</code> ,<br><code>ShortLookupTable</code> | Inversion, Farbtrennung, Aufhellung, Thresholding | ja                         |
| <code>RescaleOp</code>          |                                                                                               | Aufhellen, Abdunkeln                              | ja                         |
| <code>Color ConvertOp</code>    | <code>java.awt.Color.ColorSpace</code>                                                        | Farbraumkonversion                                | ja                         |

## Faltung

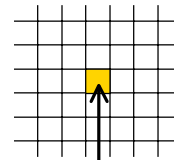
- Mathematisches Prinzip: Faltung (*spatial convolution*)
  - Berechnung der Farbe eines Zielpixels aus der Farbe des entsprechenden Quellpixels *und seiner Nachbarn*
  - Gewichtungsfaktoren gegeben durch Matrix: Konvolutionskern (*kernel*)
  - Summe der Matrixeinträge 1: Gesamthelligkeit unverändert



Quellbild

|             |            |             |
|-------------|------------|-------------|
| $k_{-1,-1}$ | $k_{0,-1}$ | $k_{+1,-1}$ |
| $k_{-1,0}$  | $k_{0,0}$  | $k_{+1,0}$  |
| $k_{-1,+1}$ | $k_{0,+1}$ | $k_{+1,+1}$ |

Matrix (*kernel*)



$$x'_{i,j} = \sum_{\substack{-1 \leq r \leq +1 \\ -1 \leq s \leq +1}} k_{r,s} \cdot x_{i+r,j+s}$$

Zielbild

Zusätzlich müssen die Zielwerte auf den zulässigen Wertebereich beschränkt (abgeschnitten) werden.



## Weichzeichnen

- Faltungsfiler, das Übergänge glättet ("verschmiert", *blur filter*)
  - Wertverteilung im Zielbild gleichmäßiger als im Quellbild
  - Gleichverteilung der Gewichte in der Matrix: bei 3x3-Matrix alle Einträge 1/9

Java-Quellcode dazu:

```
float ninth = 1.0f/9.0f;
float[] blurKernel = {
 ninth, ninth, ninth,
 ninth, ninth, ninth,
 ninth, ninth, ninth
};
ConvolveOp blurOp = new ConvolveOp
(new Kernel(3, 3, blurKernel),
ConvolveOp.EDGE_NO_OP, null);
```

|     |     |     |
|-----|-----|-----|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |



## Java 2D: ConvolveOp

- Klasse **Kernel**:
  - `public Kernel (int width, int height, float[] data)`
    - Konstruiert eine neue *kernel*-Matrix mit gegebenen Dimensionen
    - Das angegebene Array muss **width** x **height** viele Werte enthalten
- Erzeugung des Operators
  - `ConvolveOp` implementiert das Interface `BufferedImageOp`
    - `public ConvolveOp(Kernel kernel, int edgeHint)`
    - erzeugt einen Faltungsoperator mit gegebenem *kernel*
    - Zusatzangabe zur Behandlung der Pixel an Aussenkanten
      - » **EDGE\_ZERO\_FILL**: Randpixel des Zielbildes werden schwarz
      - » **EDGE\_NO\_OP**: Randpixel des Zielbildes bleiben unverändert

## Ortsfrequenz-Filter

- Mathematische Darstellung eines Bildes: (Orts-)Funktion  
 $f: X \times Y \rightarrow ColorValue$
- (Orts-)Frequenzraum:
  - Wie häufig wechselt der Farbwert (bzw. die Signalstärke) bei einer Veränderung auf der Ortsachse?
  - Fourier-Transformation: Zerlegung des Signals in Frequenzanteile
- Faltungen:
  - sind Ortsfrequenz-Filter
  - Weichzeichner:
    - » Entfernt hohe Ortsfrequenzen
  - Scharfzeichner:
    - » Betont hohe Ortsfrequenzen

## Schärfen

- Schärfung:
  - Filter, das jedes Pixel unverändert läßt, wenn seine Umgebung den gleichen Wert wie das Pixel selbst hat
  - Bei Änderungen in der Umgebung wird der Kontrast der Änderung verstärkt
- Idee:
  - Umgebungsgewichte negativ, kompensiert durch Gewicht des zentralen Pixels
- Beispiele:

|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 0  | -1 | 0  | -1 | -1 | -1 |
| -1 | 5  | -1 | -1 | 9  | -1 |
| 0  | -1 | 0  | -1 | -1 | -1 |

## Kantenerkennung

- Wie Schärfen, aber geringeres Gewicht für Originalinformation
- Summe der Matrix wesentlich kleiner als 1
  - Bild wird dunkel, fast schwarz
  - Kanten mit hohem Kontrast in weiß zu erkennen
- Beispiel:

|    |    |    |
|----|----|----|
| 0  | -1 | 0  |
| -1 | 4  | -1 |
| 0  | -1 | 0  |

## Lookup-Tabellen (1)

- Lookup-Tabellen erlauben eine direkte Umrechnung der Werte des Quellbildes in Werte des Zielbildes
  - Tabellierte Funktion:  
Quellwerte als Index für Tabelle benutzt, Zielwerte sind Einträge
  - Meist Werte zwischen 0 und 255, also 255 Tabelleneinträge
  - Verschiedene Varianten für Datentyp der Einträge (Byte, Short)
- Beispiel: Inversion

```
– Ähnlich zum fotografischen Negativbild
short[] invert = new short[256];
for (int i = 0; i < 256; i++)
 invert[i] = (short)(255 - i);
LookupTable =
 new ShortLookupTable(0, invert);
LookupOp invertOp =
 new LookupOp(table, null);
```



## Lookup-Tabellen (2)

- Es können auch verschiedene Lookup-Tabellen je Farbkanal angegeben werden
- Beispiel: Inversion des Rot-Wertes

```
short[] invert = new short[256];
short[] straight = new short[256];
for (int i = 0; i < 256; i++) {
 invert[i] = (short)(255 - i);
 straight[i] = (short)i;
}
short[][] redInvert = {invert, straight, straight};
LookupTable table =
 new ShortLookupTable(0, redInvert);
LookupOp redInvertOp =
 new LookupOp(table, null);
```

Zusatzparameter:

- bei Konstruktor für LookupTable: Angabe eines Offsets für Wertebereich (hier 0)
- bei Konstruktor für LookupOp: Angabe von RenderingHints (hier 0)

## Helligkeits-Skalierung

- Globale lineare Veränderung der Helligkeitswerte
- Zwei Einflussmöglichkeiten
  - Skalierungsfaktor (*scale factor*)
  - Verschiebung (*offset*)

- Beispiele:

- Helligkeit um 50% erhöhen

```
RescaleOp brighterOp =
 new RescaleOp(1.5f, 0, null);
```

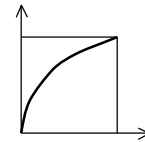
- Helligkeit um 50% reduzieren und absolute Korrektur um 64 Schritte

```
RescaleOp dimOffsetOp =
 new RescaleOp(0.5f, 64f, null);
```

## Verbesserte Helligkeitsskalierung

- Eine lineare Skalierung ist nicht optimal geeignet zur Helligkeitsanpassung (Farben wirken "ausgewaschen")
- Komplexere Funktionen in der Skalierung durch Lookup-Tabellen realisierbar
- Beispiel: Quadratische Skalierung

```
short rootBrighten = new short[256];
for (int i = 0; i < 256; i++)
 rootBrighten[i] = (short)
 (Math.sqrt((double)i/255.0) * 255.0);
```



linear



quadratisch

## Farbraumkonvertierung

- Bilder müssen oft gezielt auf einen anderen Farbraum angepasst werden
  - Die notwendigen Umrechnungen existieren vordefiniert, z.B. in Java 2D
- Drei Varianten:
  - Anpassung an Farbraum des Zielbildes
    - » nur sinnvoll, wenn Zielbild (in Java 2D: BufferedImage) bekannt
  - Anpassung an gegebenen Standard-Farbraum
    - » Beispiel (Graustufen):

```
BufferedImageOp op =
 new ColorConvertOp(
 ColorSpace.getInstance(ColorSpace.CS_GRAY), null);
```
  - Anpassung an gegebene (ICC-)Farbprofile
    - » Bestandteil eines potentiell komplexen Farbmanagement-Systems

## Java Advanced Imaging (JAI)

- Aktuelle Weiterentwicklung der Java-Bibliotheken zur Bildbearbeitung
- Wesentliche Zusatzeigenschaften:
  - Unterstützung verteilter Bildbearbeitung
    - » basiert auf Java Remote Method Invocation (RMI) und Internet Imaging Protocol (IIP) (siehe: [http://www.i3a.org/i\\_iip.html](http://www.i3a.org/i_iip.html))
  - Standardisierte Ein- und Ausgabe verschiedener Bildformate aus Dateien (BMP, GIF, FPX, JPEG, PNG, PNM, TIFF)
  - Optimierungen des Datenzugriffs:
    - » Zerlegung von Bildern in separat ladbare "Kacheln" (kompatibel mit "FlashPix"-Format)
  - Optimierung des Ausführungsmodells
    - » Zurückstellen von Teiloperationen
  - Durch Fremdhersteller erweiterbare Rahmenlösung
- Viele zusätzliche Bildbearbeitungsfunktionen
- Achtung: **Nicht** Bestandteil einer Standard-Java-Installation!

## JAI: Einfaches Beispielprogramm (1)

```
import java.awt.Frame;
import java.awt.image.renderable.ParameterBlock;
import java.io.IOException;
import javax.media.jai.Interpolation;
import javax.media.jai.JAI;
import javax.media.jai.RenderedOp;
import com.sun.media.jai.codec.FileSeekableStream;
import javax.media.jai.widget.ScrollingImagePanel;
/**
 * This program decodes an image file of any JAI supported
 * formats, such as GIF, JPEG, TIFF, BMP, PNM, PNG, into a
 * RenderedImage, scales the image by 2X with bilinear
 * interpolation, and then displays the result of the scale
 * operation.
 */
public class JAISampleProgram {
 public static void main(String[] args) {
 /* Validate input. */
 if (args.length != 1) {
 System.out.println("Usage: java JAISampleProgram " +
 "input_image_filename");
 System.exit(-1);
 }
 ...
 }
}
```

## JAI: Einfaches Beispielprogramm (2)

```
/*
 * Create an input stream from the specified file name
 * to be used with the file decoding operator.
 */
FileSeekableStream stream = null;
try {
 stream = new FileSeekableStream(args[0]);
} catch (IOException e) {
 e.printStackTrace();
 System.exit(0);
}

/* Create an operator to decode the image file. */
RenderedOp image1 = JAI.create("stream", stream);

/*
 * Create a standard bilinear interpolation object to be
 * used with the "scale" operator.
 */
Interpolation interp = Interpolation.getInstance(
 Interpolation.INTERP_BILINEAR);
...

```

## JAI: Einfaches Beispielprogramm (3)

```
/**
 * Stores the required input source and parameters in a
 * ParameterBlock to be sent to the operation registry,
 * and eventually to the "scale" operator.
 */
ParameterBlock params = new ParameterBlock();
params.addSource(image1);
params.add(2.0F); // x scale factor
params.add(2.0F); // y scale factor
params.add(0.0F); // x translate
params.add(0.0F); // y translate
params.add(interp); // interpolation method

/* Create an operator to scale image1. */
RenderedOp image2 = JAI.create("scale", params);

int width = image2.getWidth();
int height = image2.getHeight();
ScrollingImagePanel panel = new ScrollingImagePanel(
 image2, width, height);
Frame window = new Frame("JAI Sample Program");
window.add(panel);
window.pack();
window.show();
}

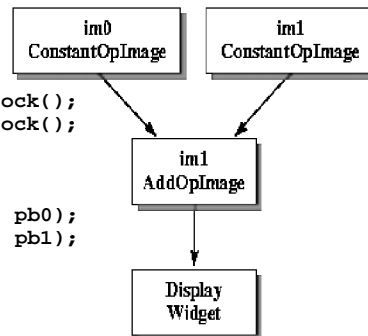
```

## JAI: Programmiermodell

- Operationen werden nicht direkt ausgeführt, sondern als Datenstruktur aufgebaut
  - Entwurfsmuster "*Command*" (Gamma et al.)
  - *Rendergraph* ist eine Baumstruktur mit den auszuführenden Operationen
  - *Parameterblöcke* werden in *Rendergraph* eingefügt

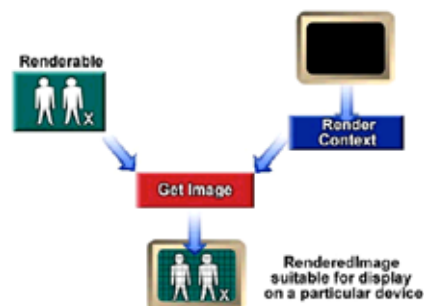
- Beispiel:

```
ParameterBlock pb0 = new ParameterBlock();
ParameterBlock pb1 = new ParameterBlock();
... // hier werden in pbX Daten
... // zur Manipulationen
... // für die Bilder geschrieben
RenderedOp im0 = JAI.create("const", pb0);
RenderedOp im1 = JAI.create("const", pb1);
im1 = JAI.create("add", im0, im1);
```



## Geräteunabhängiges Rendering

- JAI ermöglicht die Bereitstellung von *Rendergraphen*, die unabhängig vom gerätespezifischen Koordinatensystem sind
  - "*renderable*"-Objekte





## Operationen in JAI (1)

- Punkt-Operatoren
  - Eingangspixelwert durch Operation zu Ausgangspixelwert
  - Add, And, Or, Xor, Divide, Invert, Lookup, Composite, Constant, Threshold
- Flächen-Operatoren
  - Veränderung der Bildfläche durch Filtern bestimmter Pixels (Umgebungen)
  - Border, Convolve
- Geometrie-Operatoren
  - Geometrische Transformationen, die Umpositionierung der Pixel zur Folge haben
  - Lineare: Translation, Rotation, Skalierung
  - Nicht lineare: Warp-Transformationen
  - Ändern der Lage, Größe, Form eines Bildes
    - » Rotate, Scale, Translate, Warp

## Operatoren in JAI (2)

- Farbquantisierungs-Operatoren
  - Auch bekannt als Dithering
  - Quantisierungsfehler gering halten bei Operationen auf:  
Monochrombilder (Farbtiefe < 8 Bit), Farbbilder (Farbtiefe < 24 Bit)
  - ErrorDiffusion, OrderedDither
- Datei-Operatoren
  - Lesen und Schreiben
  - Fileload, Filestore, Encode, JPEG, usw.
- Frequenz-Operatoren
  - Räumlich orientierte- in frequenzorientierte Bilder übersetzen (Fourier Transformation)
  - DCT, Conjugate
- Statistik-Operatoren
  - Zum Analysieren des Inhalts von Bildern
  - Extrema, Histogram
- Kantenextraktions-Operatoren
  - Gradient, der Kanten im Bild zum Vorschein bringt
  - GradientMagnitude