Roland Holzer

Einführung in Java Advanced Imaging

für die Übungen
zur Vorlesung
Computer Vision 1
im
Wintersemester 2001/2002

Ulm 2001

Inhaltsverzeichnis

1 Was ist Java Advanced Imaging?

Java Advanced Imaging (JAI) ist ein Erweiterungspaket für Java. Es wurde von Sun entwickelt, um Javaprogrammierern die professionelle Bildverarbeitung zu erleichtern. Das Programmieren in JAI ist also reine Java–Programmierung, wobei man sich der zusätzlichen Klassen aus der JAI–Bibliothek bedient. Deshalb werden in den folgenden Abschnitten auch wichtige Java–Grundlagen besprochen. Um allerdings das Konzept der objektorientierten Programmierung vollständig zu erklären, fehlt hier der Platz. Auch eine vollständige Beschreibung von JAI würde den Rahmen dieser Einführung sprengen.

Um einen einfachen Einstieg in JAI zu ermöglichen, wird für die Übungen eine Codeschablone bereitgestellt. Diese Schablone bietet eine grafische Oberfläche mit einer Menustruktur, die Menüpunkte für das Laden von Bilddateien und deren Darstellung enthält. In den Übungen werden wir die Klasse *BildIntern* der Schablone Schritt für Schritt erweitern, so daß am Ende der Vorlesung jeder Teilnehmer sein eigenes kleines Bildbearbeitungsprogramm besitzt.

2 Ein wenig Java

Java ist eine von Sun Microsystems entwickelte objektorientierte Programmiersprache. Unter `http://www.java.sun.com` kann man sich das JDK (Java Developer Kit) sowie Zusatzbiblioteken kostenlos herunterladen. Das JDK beinhaltet unter anderem den Compiler javac.exe und den Interpreter java.exe. Java Quellcodedateien haben zwingend die Endung `.java`. Die Quellcodedateien übersetzt man dann mittels `javac <Dateiname>.java` in `.class`–Dateien. Ausführen kann man die `.class`–Dateien dann mittels `java <Dateiname>`.

Es gibt zu allen Java–Klassen ausführliche Online–Dokumentation, die sogennanten Java–Docs. Außerdem findet man auf dem Sun–Server auch Tutorials und Beispielprogramme für Java.

2.1 Variablen

In Java müssen alle Variablen deklariert und vor der ersten Verwendung initialisiert werden. Es gibt primitive Typen (Integertypen: byte, int, long; Gleitkommatypen: float, double; Wahrheitstyp: boolean) und Klassen. Eine Variable wird mit

Type bezeichner;

deklariert, eine Zuweisung erfolgt mit *bezeichner = wert;*. Deklaration und Initialisierung können auch in einem Schritt vollzogen werden: *Type bezeichner = wert;*. Beispiele: *double x = 3.4; float y; y=2.3; boolean wahr= true;*

Falls der Type einer Variable eine Klasse ist muß man (meistens) ein Objekt von dieser Klasse anlegen. Hierfür benutzt man *new Konstruktor(Parameter)*. Ein Konstruktor heißt immer wie seine Klasse (z.B. *String s = new String(); Hashtable h= new Hashtable(10);*. Die Beschreibung der Konstruktorten und ihrer Parameter findt man in den Java–Docs.

Arrays werden mit *Type* und je einem Paar eckige Klammern pro Dimension deklariert: z.B. *double[] einFeld, int[][] eineMatix*. In Java sind Arrays Klassen. Ein Array–Objekt wird — wie alle Objekte — mit *new* angelegt, z.B. *double[] fuenfElemente = new double[5]*. Jedes Array–Objekt „weiß" in Java, wie lang es ist. Dies kann man mit *length* abfragen. Der erste

Index eines Arrays ist immer 0; somit ist der höchste gültige Index *length-1*
Bsp: *double[] feld = new double[5]; for (int i=0; i<feld.length; i++) feld[i] = 5;*.

Java hat in der Klasse Arrays einige nützliche Methoden für Arrays implementiert, hier ein paar davon:

- void fill(double[] feld, double wert) — setzt jedes Element des Arrays *feld* auf *wert*. Diese Methode gibt es auch für *int, long, float,* etc.
- void fill(double[] feld, int from, int to, double wert) — setzt jedes Element zwischen *from* und *to-1* auf wert.
- void sort(double[] feld) — sortiert *feld* aufsteigend
- void sort(double[] feld, int from, int to) — sortiert den Bereich *feld[from..to-1]* aufsteigend

Beispiel:
double[] feld = new double[10];
for (int i=0; i<feld.length; i++) feld[i] = feld.length-i;
Arrays.sort(feld); // ab hier ist feld sortiert

2.2 Methoden

In der objektorientierten Welt spricht man nicht von Funktionen oder Prozeduren, sondern von Methoden. Eine Methode schreibt man in Java allgemein so:

Type methodenName(Parameter){ ... }

Type spezifiziert den Datentyp des Rückgabewertes. Wenn nichts zurückgegeben werden soll wird hier der Typ *void* verwendet. Die Aufrufparameter werden mit *Type Bezeichner* aufgelistet und durch Kommata getrennt. Zwischen *{}* kommt dann die eigentliche Implementierung der Methode. Einen Wert liefert man mittels *return* zurück. Beispiel:

- int gibMirEineEins() return 1; — Parameterlose Methode mit dem Rückgabetyp *int*, die immer eine eins zurückgibt.
- void sagMir(String s){ System.out.println("ich sag Dir: " + s);} — Methode ohne Rückgabewert, welche einen *String* übergeben bekommt.
- double add(double x, double y) return x+y; — Methode mit zwei Parametern.

Der Methodenaufruf erfolt mittels *Objekt.MehtodenName(Parameter)*. Daraus wird ersichtlich, daß man zuerst ein Objekt von einer Klasse anlegen muß und über dieses die Methode aufruft. Z.B.: *String s="Hallo"; ...; s = s.toUpperCase();*.

Falls eine Methode keine Daten aus dem Objekt benötigt (z.B. *sin(x)*) kann man der Methodendefinition das Schlüsselwort *static* voranstellen. Hierdurch kann man die Methode auch über *Klassen.methodenName(Parameter)* aufrufen (man spricht von Klassen— bzw. statischen Methoden). Ein Beispiel hierzu findet sich im nächsten Abschnitt.

2.3 Mathematische Operatoren

Wie in allen Programmiersprachen stehen auch in Java die Operatoren +,-,*,/ und % (Modulo) zur Verfügung. Wie in C/C++ gibt es auch die Alternativen +=, -=, *= und /=, z.B. ist *s += 5;* das gleiche wie *s = s+5;*.

Desweiteren gibt es in der Klasse *Math* unter anderem die folgenden Klassenmethoden:

abs(x)	Absolutbetrag von x
sin(x)	Sinus von x
cos(x)	Cosinus von x
tan(x)	Tangens von x
exp(x)	e^x
log(x)	Logarithmus von x
sqrt(x)	$\sqrt{x}$

Beispiel: *double wurzel_von_25 = Math.sqrt(25);*

2.4 Programmflußsteuerung

Bedingte Anweisung:

```
if (Bedinung) Anweisung;
if (Bedinung) {Anweisung; ...}
if (Bedinung) {Anweisung; ...}else{Anweisungen;...}
```

While Schleife:

```
while (Bedinung) Anweisung;
while (Bedinung){ Anweisungen; ... }
```

For Schleife:

```
for ( Init; Bedinung; Fortschaltung ) Anweisung;
Bsp: for ( int i=0; i<25; i++ ) summe += i;
```

3 Raster und WritableRaster

Man kann zwar in JAI über die Imageklasse direkt auf die Pixelwerte des gekapselten Bildes zugreifen, allerdings ist es aus Performanzgründen empfehlenswert, sich das sogenannte „Raster" des Bildes zu holen und aus diesem dann die Pixelwerte auszulesen.

Hierfür stellen alle Imageklassen von JAI die Methode *getData()* zur Verfügung. Da JAI nicht auf Grauwertbilder beschränkt ist, wird jedes Pixel prinzipiell als Array dargestellt, welches z.B. die RGB–Werte enthält. Bei Grauwertbildern besteht dieses Array nur aus dem Wert für die Helligkeit und hat somit die Länge eins. Die Klasse *Raster* stellt eine Vielzahl von Methoden zur Verfügung. Die wichtigsten davon sollen hier kurz erklärt werden:

- *int getWidth()* — liefert die Breite des Rasters zurück.

- *int getHeight()* — liefert die Höhe zurück.
- *float[] getPixel(int x, int y, float[] pixel)* — liefert den Wert des Pixels an der Stelle x,y zurück. Mittels des Parameters *pixel* kann man der Methode ein bereits allokiertes float–Array übergeben, somit spart man sich das Anlegen eines neuen float–Arrays pro Aufruf. Dies sollte man immer ausnutzen, da sonst schon bei einem 512x512 großen Bild über eine viertel Million neue Arrays erzeugt werden, falls man jeden Pixel ausliest. Dieses übergebene Array muß bei Grauwertbildern mindestens die Länge eins haben.
- *double[] getPixel(int x, int y, double[] pixel)* — dito, außer daß das Pixel als double–Array behandelt wird.
- *float[] getPixels(int x, int y, int w, int h, float[] pixels)* — hier wird ein rechteckiger Bereich des Bildes ausgelesen, x und y spezifizieren die obere linke Ecke des Bereichs, w und h die Breite und Höhe. Der Parameter *pixels* kann wie in *getPixel* verwendet werden, allerdings muß er hier die Größe $w*h$ haben, da sonst ein Laufzeitfehler ausgelöst wird.
- *double[] getPixels(int x, int y, int w, int h, double[] pixels)* — dito, allerdings werden auch hier die Pixel wieder mit doppelter Genauigkeit behandelt.

Die Klasse *WritableRaster* ist von *Raster* abgeleitet und erweitert diese um Methoden, mit denen man Pixel setzen kann. Die Methoden zum Auslesen der Pixelwerte werden von *Raster* geerbt. Auch hier wieder nur die wichtigsten zusätzlichen Methoden:

- *void setPixel(int x, int y, float[] pixel)* — setzt den Wert des Pixels an der Stelle x, y auf den Wert in *pixel*. Diese Methode gibt es auch mit double–Array.
- *void setPixels(int x, int y, int w, int h, float[] pixels)* — diese Methode ist das schreibende Äqvivalent zu *getPixels*, und auch hier muß *pixels* mindestens $w*h$ viele Elemente enthalten.
- *void setRect(Raster source)* — Mit dieser Methode kann man ein *Raster* in ein *WritableRaster* zeichnen. Die obere linke Ecke des *Raster source* wird an der oberen linken Ecke des *WritableRasters* gezeichnet.
- *void setRect(int dx, int dy, Raster source)* — dito, hier wird der Ursprung von *source* an dx, dy angelegt. Somit kann man ein Raster z.B. in die Bildmitte eines anderen Bildes zeichnen.

4 Bilder in JAI

Nachdem wir die Rasterklassen betrachtet haben, stellt sich natürlich die Frage, woher wir diese eigentlich bekommen. Wie bereits erwähnt, werden wir die Klasse *BildIntern* der Codeschablone erweitern. Diese besitzt eine Variable namens *theImage* vom Typ *TiledImage*, welche das dargestellte Bild enthält. Über *theImage* kommt man also an die Pixelwerte.

Doch vorher ein wenig über Imageklassen in JAI. Es gibt etliche Imageklassen und Interfaces in JAI; wir wollen uns allerdings der Einfachheit halber auf zwei dieser Klassen beschränken: *PlanarImage* und *TiledImage*.

4.1 Die Klasse PlanarImage

Diese Klasse ist die Basisklasse aller Imageklassen in JAI. Sie stellt einige generelle Methoden zum Zugriff auf die Bilddaten zur Verfügung; hier die wichtigsten:

- *int getWidth()* — liefert die Breite des Bildes.
- *int getHeight()* — liefert die Höhe des Bildes.
- *Raster getData()* — liefert das *Raster* (s.o.) des Bildes zurück.
- *WritableRaster copyData()* — liefert eine Kopie des Bildes in einem *WritableRaster* zurück. Da *PlanarImage* aus Optimierungsgründen nicht direkt manipulierbar ist, werden die Daten hier kopiert. Man kann allerdings das *WritableRaster* manipulieren und dann ein neues *PlanarImage* daraus erzeugen mittels der Methode *BildIntern.getNewImage(WritableRaster)* aus der Codeschablone.

4.2 Die Klasse TiledImage

TiledImage ist eine von *PlanarImage* abgeleitete Klasse und erbt somit alle Methoden von derselben. Die für uns wichtigste Zusatzeigenschaft eines *TiledImage* ist, daß es sich direkt manipulieren läßt. Der Name tiled Image kommt von der Eigenschaft, daß man das Bild in Kacheln unterteilen kann, was bei großen Bildern sehr sinnvoll ist, da man so nur den aktuell betrachteten Bereich in den Speicher laden muß. Uns soll dies nicht weiter interessieren, da wir nur Bilder aus einer einzelnen Kachel benutzen wollen. Deshalb sind für uns primär nur zwei Funktionen dieser Klasse interessant:

- *Raster getTile(int tileX, int tileY)* — liefert das *Raster* einer Kachel zurück. Da wir nur eine Kachel pro Bild verwenden wollen, sind die Werte beider Parameter bei uns immer 0.
- *WritableRaster getWritableTile(int tileX, int tileY)* — dito, nur daß ein *WritableRaster* zurückgegeben wird. Manipulationen des *WritableRaster*s wirken sich hier direkt auf das Bild aus.

4.3 Zuweisungen zwischen Planar– und TiledImage

Da *TiledImage* eine Subklasse von *PlanarImage* ist kann man jeder Variable vom Type *PlanarImage* eine Variable vom Type *TiledImage* zuweisen. Das man einem Objekt immer Objekte seiner Subklassen zuweisen kann nennt man Polymorphismus:

```
TiledImage tilImage =...;
...
PlanarImage image =  tilImage;
```

Andersherum funktioniert dies nicht. Allerdings kann man, mit Hilfe eines speziellen Konstruktors, aus einem *PlanarImage* einfach ein *TiledImage* erzeugen:

```
PlanarImage image = ...;
TiledImage tilImage = new TiledImage( image, true );
```

Der zweite Parameter dieses Konstruktor legt fest ob das *TiledImage* den gleichen Speicherbereich wie das *PlanarImage* verwenden soll (true) oder nicht (false).

Für die Erzeugung neuer Bilder bietet die Klasse *BildIntern* aus der Codeschablone ein paar einfache Methoden:

- *TiledImage getNewImage(int width, int height)* — liefert ein neues Bild zurück, wobei alle Pixel auf schwarz gesetzt sind.
- *TiledImage getNewImage(float[] theData, int width, int heigth)* — dito, allerdings werden die Pixel auf die Werte aus *theData* gesetzt. Die Länge von *theData* muß *width*height* sein.
- *TiledImage getNewImage(WritableRaster raster)* — erzeugt ein Bild, dessen Inhalt durch *raster* festgelegt ist.

5 Ein paar Beispiele

Mit jeder Übungsaufgabe wird die Klasse *BildIntern* der Codeschablone um eine Methode reicher. Wir wollen nun exemplarisch eine Methode schreiben, welche den Mittelwert eines Bildes berechnet. In *BildIntern* findet man dann z.B. folgendes:

```
public double Mittelwert(){
  // Diese Methode soll den Mittelwert von theImage berechnen
}
```

Prinzipiell gehen wir wie folgt vor:

1. Wir holen uns das *Raster* von *theImage*.
2. Wir laufen über alle Bildpunkte und summieren sie auf.
3. Wir teilen das Ergebnis noch durch die Anzahl der Bildpunkte und geben dies zurück.

```
public double Mittelwert(){
   // Wir holen uns das Raster von theImage:
   Raster unserRaster = theImage.getData();

   // Wir legen eine Variable für die Summe der
   // Pixelwerte an
   double summe = 0.0;

   // Wir legen uns ein Array für die auszulesenden
   // Pixel an (s.o.):
   double[] pixel = new double[1];

   for ( int x=0; x<unserRaster.getWidth(); x++ )
       for ( int y=0; y<unserRaster.getHeight(); y++ ){
           // Pixel x,y auslesen und in pixel speichern ...
           pixel = unserRaster.getPixel( x, y, pixel );
           // ...und den Grauwert zu summe dazuaddieren
```

```
            summe += pixel[0];
        }

    // Es gibt Breite*Höhe viele Pixel,
    // durch diese Zahl müssen wir noch teilen:
    double teiler = unserRaster.getWidth()*unserRaster.getHeight();

    // return gibt das Ergebnis der Methode zurück:
    return (summe/teiler);
}
```

Als zweiters wollen wir ein neues Bild als Ergebnis unserer Methode. Hierzu schreiben wir eine Methode *invert()*, diese soll *theImage* invertieren zurückgeben. Die Methodensignatur sei gegeben durch:

```
public PlanarImage invert(){
  // Diese Methode soll das invertierte Bild von theImage zurückliefern
}
```

Prinzipiell gehen wir wie folgt vor:

1. Wir legen uns ein neues Bild an, in welchem wir das Ergebnis speichern
2. Holen uns das *WritableRaster* des neuen Bildes
3. Wir holen uns das *Raster* von *theImage*.
4. Wir laufen über alle Bildpunkte von *theImage*.
5. Setzen den invertierten Wert im Ergebnisbild

```
public PlanarImage invert(){
    // ein neues Bild erzeugen:
    TiledImage ergebnis = getNewImage( theImage.getWidth(), theImage.getHeight() );

    // Das WritableRaster (wir wollen ja die Pixel verändern)
    // des Ergebnisbides holen:
    WritableRaster ergRaster = ergebnis.getTile(0,0);

    // Das Raster des Orginalbildes holen:
    Raster sourceRaster = theImage.getData();

    // Wir legen uns ein Array für die auszulesenden
    // Pixel an (s.o.):
    double[] pixel = new double[1];

    for ( int x=0; x<sourceRaster.getWidth(); x++ )
        for ( int y=0; y<sourceRaster.getHeight(); y++ ){
            // Pixel x,y auslesen und in pixel speichern ...
            pixel = unserRaster.getPixel( x, y, pixel );
```

```
            // Wert invertieren:
            pixel[0] = 1-pixel[0];

            // Pixel im Ergebnisbild setzten:
            ergRaster.setPixel( x, y, pixel );
        }

    // Das Ergebnis Bild zurückgeben:
    return ergebnis;
}
```

6 Das Rad ist schon erfunden — die Klasse JAI

Die Klasse JAI stellt dem Programmierer Basisoperationen der Bildverarbeitung zur Verfügung. Da die meisten dieser Operationen unterschiedliche Parameter benötigen werden diese zuerst in einem *ParameterBlock* zusammengefaßt. Ein *ParameterBlock* mit dem Namen *pb* wird mit *ParameterBlock pb = new ParameterBlock();* erzeugt. Die zwei wichtigsten Methoden sind:

- *addSource(Object source)* — fügt das übergebene Bild als Quelle in den Parameterblock ein, da alle Klassen in Java von *Object* abgeleitet sind, kann man hier z.B. Objekte aller Bildklassen übergeben.
- *add(double wert)* — fügt einen Parameterwert hinzu. Diese Methode gibt es natürlich auch mit allen anderen Parametertypen.

Mittels der Klasse *JAI* kann man nun ein bearbeites Bild mit der *create*–Methode erzeugen. Diese Methode ist überladen, hier einige Signaturen:

- *PlanarImage create(String op, PlanarImage img)* — *op* bezeichnet den Operator und *img* das zu bearbeitende Bild.
- *PlanarImage create(String op, PlanarImage img1, PlanarImage img2)* — Falls der Operator *op* zwei Bilder benötigt, kann man diese Version von *create* verwenden.
- *PlanarImage create(String op, ParameterBlock pb)* — Hier sind die Parameter für die Operation *op* in dem ParameterBlock *pb* gesammelt.

Hier werden nun ein paar ausgewählte Operationen aufgelistet:

Pixeloperatoren:	
Operator	**Beschreibung**
Add	Addiert zwei Bilder pixelweise
Subtrat	Subtrahiert zwei Bilder pixelweise
Multiply	Multipliziert zwei Bilder pixelweise
AddConst	Addiert zu jedem Pixel einen konstanten Wert
DivideByConst	analog
MultiplyConst	analog
SubtractConst	analog
LookUp	ändert die Pixelwerte nach einem LookUp–Table
Absolute	Berechnet von jedem Pixel den Absolutebetrag
Exp	Berechnet von jedem Pixel den Logarithmus
Log	Berechnet von jedem Pixel den e^{Pixel}
Invert	Invertiert das Bild

Bereichsoperatoren:	
Operator	**Beschreibung**
Convolve	Faltet das Bild mit dem *JAIKernel* aus dem Parameterblock
BoxFilter	s. Vorlesung
MedianFilter	s. Vorlesung

Frequenzoperatoren:	
Operator	**Beschreibung**
DFT	berechnet die Fouriertransformierte
IDFT	die inverse Fouriertransformation
Magnitude	berechnet den Betrag von complexwertigen Bildern
Phase	berechnet den Phasenwinkel jedes Pixels

Beispiele: (seien img und img2 vom Type *PlanarImage*)

1. Bei manchen Bildmanipulationen können negative Werte auftreten. Allerdings interessiert hier oft nur der Betrag der Werte. Diesen errechnet man wie folgt:

```
ParameterBlock pb = new ParameterBlock();
pb.addSource( img );
PlanarImage betrag = JAI.create( "Absolute", pb );
```

2. Manchmal kann man das Ergebnis eines Filtervorgangs besser beurteilen wenn man den Unterschied zwischen Orginal und bearbeitetem Bild betrachtet:

```
ParameterBlock pb = new ParameterBlock();
pb.addSource( img );
pb.addSource( img2 );
PlanarImage differenz = JAI.create( "subtract", pb );
```

   oder:

```
PlanarImage differenz = JAI.create( "subtract", img1, img2 );
```

3. In der Vorlesung werden wir die Fourier–Transformation kennenlernen und welchen Nutzen sie hat. Die Transformierte eines Bildes erhält man mittels...

```
ParameterBlock pb = new ParameterBlock();
pb.addSource( img );
PlanarImage fourier = JAI.create( "DFT", pb );
```

...und wenn man gleich den Betrag der Fourier–Koeffiziente will:

```
pb = new ParameterBlock();
pb.addSource( fourier );
PlanarImage fourierBetrag = JAI.create( "magnitude", pb);
```