CG Übung 2: ImageFilter

Sascha Feldmann
sascha.feldmann@gmx.de

Inhaltsverzeichnis

1 Profil

1.1 Aufgabe

Erstellung einiger Filter mit ImageJ.

1.2 Abgabe

15.05.2012

1.3 Autor

Sascha Feldmann - 778455

1.4 Beginn der Bearbeitung

24.04.2012

2 Dokumentation zur Aufgabe

2.1 Methodik

Nachdem IDE-fähigen Setup von ImageJ schaute ich mir die importieren Beispiel-Filter ScaleImage.java und RGBComponentRemover.java an. Die Methodik der Speicherung von Farbwerten in den Pixeln war mir dank des aktuellen Vorlesungsinhalts klar.

Anschließend entwickelte ich das Histogramm, den Helligkeitsfilter, den Unschafzeichner und anschließend die Bilineare Interpolation.

2.2 Filter

2.2.1 Speicherung der Farbwerte

Ein Pixel speichert die Farbwerte folgendermaßen:

0x00RRGGBB

Dabei ist das Hexadezimalsystem Basis. Die ersten beiden Kanäle stehen für den Alpha-Kanal gefolgt von den einzelnen Farbkomponenten. Jede Farbkomponente kann einen Wert von 0 bis 256 haben, da jede Stelle 16 mögliche Zahlen verwaltet. Jede Stelle wird also durch 4 Bits ausgedrückt.

Folglich müßen die Farbkomponenten folgendermaßen extrahiert werden:

Listing 1: Code

```
1 int r = ( pixOld[i] & 0x00ff0000 ) >> 16;
2 int g = ( pixOld[i] & 0x0000ff00 ) >> 8;
3 int b = pixOld[i] & 0x000000ff;
```

Durch den logischen und-Operator wird die jeweilige Farbkomponente maksiert und durch Shiften um jeweils

- 16 Bits nach rechts (4 pro Bit) extrahiert den Rot-Anteil
- 8 Bits nach rechts (4 pro Bit) extrahiert den Grü-Anteil
- Blau muss nicht mehr geshifted werden, da keine Bits mehr folgen.

2.2.2 ComponentRemover

Die einzelnen Komponenten laßen sich sehr einfach entfernen. Mithilfe des Benutzerdialogs kann der favorisierte Farbraum ausgewählt werden. Ein weiterer Dialog bietet die Komponenten an. Die extrahierten RGB-Farbwerte werden mittels der ColorConverter-Klassen in einem RGB-Objekt gespeichert. Je nach ausgewählten Farbraum erfolgt eine Umwandlung in andere Fabraumobjekte. Nun werden die farbraumspezifischen Komponenten entfernt, indem sie einfach auf 0 gesetzt werden. Mit den entfernten und möglicherweise erhaltenen Werten wird ein neues Objekt des jeweiligen Farbraums gebildet. Durch Konvertierung in ein neues RGB-Objekt und anschließender Extrahierung der RGB-Komponenten haben wir die neuen Werte für den ImageProcessor und damit das resultierende Bild erhalten.

Listing 2: Beispiel für CMY

```
4  // Convert to CMY and
5  // save each component
6  RGB rgb =
7   new RGB(r, g, b );
8  CMY cmy =
9   rgb.returnCMY( );
10 int c = cmy.getCyan( );
11 int m = cmy.getMagenta( );
12 int y = cmy.getYellow( );
13
14 // Removing components
15 if ( cyan )
16 c = 0;
17 if ( magenta )
18 m = 0;
19 if ( yellow )
20 y = 0;
21
22 // Convert back to RGB
23 cmy = new CMY(c, m, y );
24 rgb = cmy.returnRGB( );
```

```
r = rgb.getRed( );
g = rgb.getGreen( );
b = rgb.getBlue( );
```

2.2.3 Histogram

Ein Histogramm zeigt Häufigkeiten von Werten an. Der häufigste Wert ist dabei das Maximum. Das Maximum der Helligkeitswerte eines Farbraums muss also festgestellt werden. Alle anderen Häufigkeitswerte sind relativ zu diesem Maximum. Zur Speicherung der Häufigkeitswerte bietet sich ein Array mit der Größe = Wertebereich an. Der Helligskeitswert jedes Pixels inkrementiert entsprechend jeden Wert des Histogramm-Arrays.

Ich überlegte nun, wie man ein Histogramm grafisch mit ImageJ darstellen kann. Die ImageProcessor-Klasse bietet direkte Canvas-Methoden an. Dabei machte ich mir die drawDot()-Methode zunutze, da ich jedes einzelne Pixel zeichnen wollte. Da der Ursprung (0/0) oben links liegt, beim Diagramm aber unten links, muss folgende Bedingung für das Zeichnen des Pixels gelten:

Listing 3: Histogramm-Schleife

```
// Draw timber with
// current x - cord. here
for ( int y = 0; y <= ipNew.getHeight( ); ++y ) {
if ( y > ( ipNew.getHeight( ) - freq ) ) {
        ipNew.drawDot(x, y );
 }
}
```

Wenn das aktuelle y (im zu zeichnenden Histogramm-Image) also im Häufigkeitsbereich des aktuellen Histogrammwertes ($HoehedesHistogramms-Haeufigkeitvonx$)(freq impliziert frequency) liegt, wird an der aktuellen Stelle der Pixel gefüllt. Dabei wird das Histogramm also spaltenweise von oben nach unten durchiteriert.

Der Histogramm-Filter ist so konstruiert, dass er sämtliche Werte darstellen kann. Der Benutzer kann sich z.B. problemlos die Verteilung der Rot-, Grün- oder Blau-Werte anschauen.

2.2.4 LuminanceChanger

Das grundsätzliche Prinzip der Auswahl eines Farbraums vom Benutzer, der Konvertierung vom RGB-Farbraum tu diesem Farbraum und die anschließende Rückkonvertierung zu RGB bleiben hier erhalten.

Aus Bildbearbeitungssoftwares sind mehrere Helligkeitsmanipulationsmethoden bekannt: die Addition um den eingebenen Wert sowie die Multiplikation der Farbkomponenten mit diesem.

Beide Methoden sind sehr simpel.

- Bei RGB und CMY wird der Faktor gleichmäßig allen Komponenten hinzuaddiert. Bei Über -oder Unterschreitungen der Farblimits wird das jeweilige Maxi-/ Minima gesetzt.

- Bei YUV wird nur die Y-Komponente verändert, da diese ja eben die Helligkeit verwaltet.

- Bei HSV ist die Value-Komponente (in der Farbraum-Pyramide ist die Helligkeit V die vertikale Achse) die Helligkeit.

2.2.5 BlurrFilter

Blurring entsteht durch Berechnung von Mittelwerten für jeden Pixel und seinen Nachbarn. Ich wollte von Anfang an einen rekursiven Algorithmus entwickeln, der eine beliebige Anzahl von Nachbarpixeln in die Berechnung einbringen kann. Hierfür bietet sich eine Aufwandsbetrachtung an:

- Bei 4 Nachbarn sind für ein Bild mit einer Größe von 1000 x 1000 Pixeln $4 \cdot 10^6$ Rechnungen nötig.

- Möchte man nun von diesen 4 Nachbarn auch noch alle Nachbarn miteinbeziehen, dann steigt der Aufwand exponentiell: $4 \cdot 4 \cdot 10^6$

- In Stufe 3 hat man schon 64 Nachbarpixel.

Ein solcher Aufwand ist natürlich nicht benutzerfreundlich. Dennoch wollte ich den Blurr-Effekt auf diese Art und Weise implementieren.

Listing 4: Rekursiver Aufruf der Blurring-Methode

```
36 // Recursive, get middle
37 // values of each neighbour pixel here
38 final int[] middleTop = getMiddleValues(topX, topY, depth - 1, img );
39 final int[] middleBot = getMiddleValues(botX, botY, depth - 1, img );
40 final int[] middleLeft = getMiddleValues(leftX, leftY, depth - 1,
41                                           img );
42 final int[] middleRight = getMiddleValues(rightX, rightY,
43                                           depth - 1, img );
```

Die Benutzereingabe des Blurring-Faktors bedeutet also für die Anzahl der Nachbarn: $Anzahl = 4^F$

Das Erstaunliche: Das Blurring scheint selbst bei hohem Blurring-Faktor nicht so stark zuzunehmen wie beim Blurring auf dem geblurrten Bild.

Dennoch möchte ich aus Anschauungsgründen diese Lösung nicht verwerfen.

2.2.6 Scale-Filter

Im Filter sind zwei verbreitete Scale-Arten vorgesehen:

- NearestNeigbour: Der Wert des neuen Pixels ergibt sich aus dem relativen alten (Runden der RGB-Werte). Beim Upscalen werden also Pixel ganz einfach wiederholt, beim Downscalen gehen diese logischerweise verloren.

- Bilineare Interpolation: Diese setzt die mathematische Interpolation um. Bilinear heißt sie, da sie in unserem 2-dimensionalen Fall in x- und y-Richtung unterschiedlich gewichtet ist.