SoSe 2021

AG Verteilte Systeme - Tag 1

Programmierpraktikum

Nikolaus Korfhage, Daniel Schneider

27.08.21

Ziel der folgenden Aufgaben ist es einerseits Bilderverarbeitung (Aufgabe 1) kennen zu lernen und sich andererseits mit Grundlagen der Netzwerkprogrammierung (Aufgabe 2) vertraut zu machen. Zu Aufgabe 1 finden Sie Vorgaben im Ilias.

Aufgabe 1 - Bildfilterung (10 Punkte)

Digitale Bilder können auf vielfältige Weise verarbeitet werden. Eine verbreitete Methode ist die Filterung. Sie erzeugt ein neues Bild basierend auf einer pixelbasierten Manipulation des Originalbildes. Jedes Pixel im neuen Bild wird als Funktion von Pixeln des Originalbildes, normalerweise aus der Umgebung des Zielpixels, berechnet. In Abbildung 1 ist die Anwendung eines Gauß-Filter auf das Originalbild zu sehen. Dabei wird für jedes Pixel im Ergebnisbild eine Filtermatrix auf einen Bereich um das entsprechende Pixel im Originalbild angewendet. Diese Operation wird als Faltung (engl. *convolution*) bezeichnet. Weitere Beispiele für Bildfilter sind Kantendetektion, Schärfung oder Weichzeichner.

In dieser Aufgabe soll zunächst ein Bild eingelesen werden und dieses anschließend in Graustufen konvertiert werden. Auf dem resultierenden 1-Kanal 2D-Array sollen dann Filter angewendet werden. Die entsprechenden Funktionen (Faltung und Konvertierung in Graustufen) sollen dabei selbst implementiert werden d.h. es sollen keine Libraries verwendet werden.

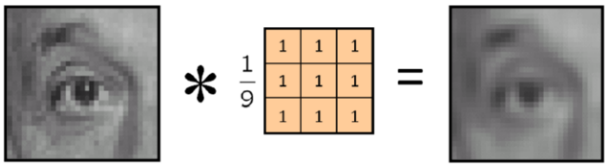

Abbildung 1: Glättung durch Gauß-Filter

- **(1 Punkt)** Das Eingabebild soll zunächst als `BufferedImage` eingelesen werden. Dies können Sie in der `main`-Methode der Klasse `ImageProcessing` erledigen. Jedes Bildpixel ist in `BufferedImage` als `int` gespeichert, hat also 32 Bit zur Verfügung. In Abbildung 2 ist die Aufteilung der Bit-Repräsentation eines RGB Pixel dargestellt. Jeweils ein Byte ist dem Alpha-, Rot-, Grün und Blaukanal zugeordnet und kann Werte zwischen 0 und 255 annehmen[1]. Mit `getRGB(x,y)` kann der RGB Wert eines Pixels im Bild ausgelesen werden. Z.B kann mit `getRGB(0,0)` auf den RGB Wert des Pixel links oben im Bild zugegriffen werden.

[1]Der Alphakanal, der die Transparenz des Pixel festlegt, soll hier nicht berücksichtig werden. D.h. $A = 11111111_2 = 255_{10}$.

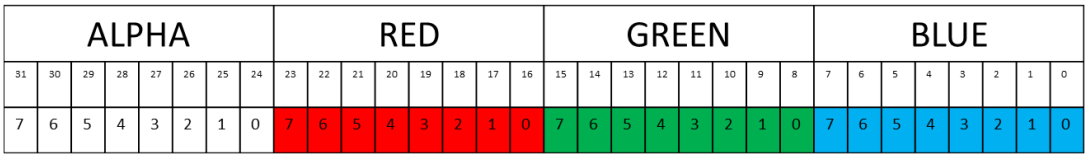

Abbildung 2: Zuordnung der Bit Positionen in `BufferedImage`

- **(1,5 Punkte)** Implementieren Sie die Methode `convertToGrayScaleArray`, die ein `BufferedImage` in ein 2-dimensionales `int`-Array konvertiert, wobei jeder Wert im Array einen Grauwert repräsentiert. Es gibt verschiedene Möglichkeiten, ein RGB-Bild in Graustufen zu konvertieren. Nutzen Sie hier folgende Formel: $G = 0,299 \times R + 0,587 \times G + 0,114 \times B$. Diese Verteilung der Farben spiegelt die entsprechende Farbempfindlichkeit der menschlichen Wahrnehmung wieder.

- **(1,5 Punkte)** Implementieren Sie dann die Methode `convertToBufferedImage`, die aus einem Integer-Array mit Werten für Graustufen ein `BufferedImage` vom Typ `BufferedImage.TYPE_INT_RGB` erzeugt. Schreiben Sie das konvertiere Bild in eine Datei.

Als nächstes sollen auf das Graustufen-Bild Filtermasken (engl. *kernel*) angewendet werden. Filtermasken sind meist quadratische Matrizen in unterschiedlichen Größen. Für digitale Bilder wird die Filtermaske K mit einer diskrete 2-dimensionale Faltung auf das Bild I angewendet[2]:

$$I^*(x,y) = \sum_{i=1}^{n}\sum_{j=1}^{n} I(x-i+a,\ y-j+a)K(i,j), \tag{1}$$

Dabei ist $I^*(x,y)$ das Ergebnispixel, a ist die Koordinate des Mittelpunktes in der quadratischen Filtermaske und $K(i,j)$ ist ein Element der Filtermaske. Um den Mittelpunkt eindeutig definieren zu können, sind ungerade Abmessungen der Filtermasken notwendig. Abbildung 3 veranschaulicht die Anwendung der Filtermaske auf eine bestimmte Position im Eingabebild.

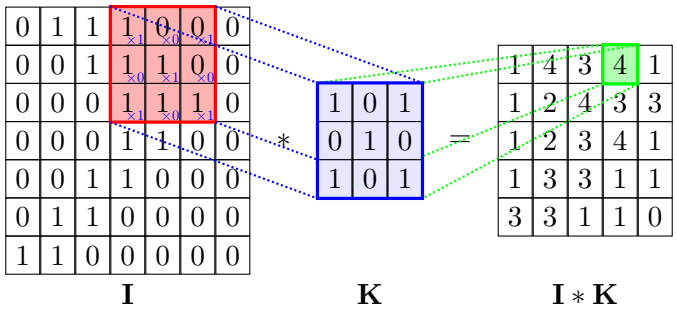

Abbildung 3: Faltung von Filtermaske K mit Eingabebild I

- **(2 Punkte)** Implementieren Sie in der Klasse `Kernel` die Funktion `convolve` die ein Eingabebild als 2D-Array (`int[][]`) entgegennimmt, die Faltung durchführt und das Ausgabebild ebenfalls als 2D-Array vom Typ `int[][]` zurückliefert. Beachten Sie, dass die Größe des Ausgabebildes nach der Faltung entsprechend der Filtergröße kleiner ist als die des Eingabebildes. Die Breite des Ausgabebildes ist dann $w_{I_{out}} = w_{I_{in}} - w_K + 1$, analoges gilt für die Höhe. Sowohl beim Array des Eingabebildes als auch beim Array des Kernel steht die erste Dimension für die `x`-Koordinate und die zweite für die `y`-Koordinate. Der Konstruktor der Klasse `Kernel` ist bereits implementiert: Um die Eingabe der Filtermatrix

[2] In der Praxis würde die Faltung durch eine Fourier-Transformation erfolgen, sodass sie in ein Produkt überführt wird. Dadurch reduziert sich der Aufwand zur Berechnung einer diskreten Faltung von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n \cdot \log n)$.

zu erleichtern wird die übergebene Filtermatrix dort transponiert. So ist es möglich einen Kernel, z.B. $k = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}$ mit

```
new Kernel ( new double[][] {{0,  0,  1},
                             {0,  1,  0},
                             {1,  0,  0}} );
```

zu initialisieren.

- **(1 Punkt)** Erstellen Sie mindestens einen JUnit Test für die Klasse `Kernel`. Testen Sie ob die Funktion `convolve` korrekte Ergebnisse liefert, indem Sie prüfen, ob die in Abbildung 3 dargestellte Faltung auf dem rot gekennzeichneten Bildausschnitt tatsächlich den Wert 4 liefert und ob die Breite und Höhe des Ausgabebildes korrekt sind. Erstellen Sie bei Bedarf weitere Tests.

- **(1 Punkt)** Wenden Sie verschiedene Filtermatrizen aus der Klasse `Kernels` auf das in Graustufen konvertierte Beispielbild an, konvertieren Sie es zurück in ein `BufferedImage` und schreiben das Ergebnisbild jeweils in eine Datei. Überlegen Sie sich weitere Filtermatrizen und wenden Sie sie auf das Bild an.

- **(2 Punkte)** Die Bilder sind nach der Faltung entsprechend der Kernelgröße kleiner als das Eingabebild. Überlegen Sie sich, wie dieses Problem behoben werden könnte. Implementieren Sie eine Lösung, bei der die Breite und Höhe des Eingabebildes erhalten bleiben.

Aufgabe 2 - Client für Sockets (5 Punkte)

Das Konzept eines Datenstroms erlaubt es Programmierern, sich bei der Programmierung auf die eigentliche Datenverarbeitung zu konzentrieren und von den Implementierungsdetails der Input-/Output-Operationen zu trennen. Dies erlaubt Programmierern, ihren Quellcode für unterschiedliche Arten von Datenströmen wiederzuverwenden, zum Beispiel für Lese- und Schreibzugriffe auf Dateien, aber auch für Netzwerkkommunikation.

Eine Verbindung über ein Netzwerk wird über so genannte Sockets hergestellt. Ein Socket stellt dabei ein Ende einer Verbindung zwischen zwei Endpunkten dar. Über den `InputStream` und den `OutputStream` des `Socket` werden Bytes von der Gegenstelle empfangen bzw. an diese gesendet.

Implementieren Sie mit Hilfe von Sockets ein einfaches, bidirektionales, interaktives und textorientiertes Netzwerk-Protokoll[3]. Hier soll zunächst nur der Client implementiert werden. Erstellen Sie eine Klasse `TextClient`, in der Sie in der `main`-Methode eine Verbindung zu einem Server aufbauen. Ihr Programm soll Benutzereingaben von der Konsole lesen, sie zum Server schicken und anschließend die Antwort des Servers auf der Konsole ausgeben.

Sie können zum Testen Ihres Clients unseren Echo-Server auf `dsgw.mathematik.uni-marburg.de` und Port 32823 verwenden. Der Echo-Server ist ein einfacher Server basierend auf Sockets, der Verbindungen akzeptiert und der die empfangenen Textnachrichten an den Sender zurückschickt.

[3] Kommunikation über Sockets sollte in der Praxis keinesfalls wie hier ungesichert verwendet werden und dient hier nur zur Veranschaulichung.