Praktikum 5 - Bildbearbeitung

Rechnerarchitektur WS25/26 - Prof. Dr. Pascal Bormann

Praktikum 5 - Bildbearbeitung

Willkommen im Praktikum 5 von Rechnerarchitektur! In dieser Woche wartet eine reine ARM-Programmieraufgabe auf Sie, bei der Sie Bilder bearbeiten sollen. Machen Sie sich zuerst mit dem bereitgestellten Code-Gerüst vertraut und lesen Sie den Abschnitt über Bildformate, dann implementieren Sie die notwendigen Funktionen für Aufgabe 1 und Aufgabe 2 in den Dateien `to_grayscale.s` und `detect_edges.s`.

Das Code-Gerüst

Das bereitgestellte Projekt kompiliert eine Kommandozeilen-Anwendung zur Bildmanipulation. Nutzen Sie innerhalb der VM oder auf den Laborrechnern `make` um den Code zu kompilieren und führen Sie ihn von der Kommandozeile aus:

```
cd ${IHR_VERZEICHNIS}/aufgabe5 && make all
./bin/img_tool ./images/pride.jpg
```

`make all` kompiliert das Projekt, bestehend aus der C++-Datei `src/main.cpp` und den beiden ARM-Assembler Dateien `asm/to_grayscale.s` und `asm/detect_edges.s`. Innerhalb des C++-Codes sind zwei Funktionen (`to_grayscale` und `detect_edges`) definiert, die Sie in den Assembler-Dateien implementieren sollen. Das korrekte Linken der Dateien übernimmt `make` für Sie. Nach dem Kompilieren finden Sie das ausführbare Programm im `bin` Verzeichnis. Wenn Sie es ohne Argumente über die Kommandozeile starten, so erhalten Sie Informationen über die Parameter, die das Programm entgegennimmt:

```
./bin/img_tool

>> Usage: ./bin/img_tool <image-path>
```

Sie können es also mit einem Pfad auf ein beliebliges Bild ausführen. Für den Anfang nutzen Sie das bereitgestellte Bild im `images` Verzeichnis.

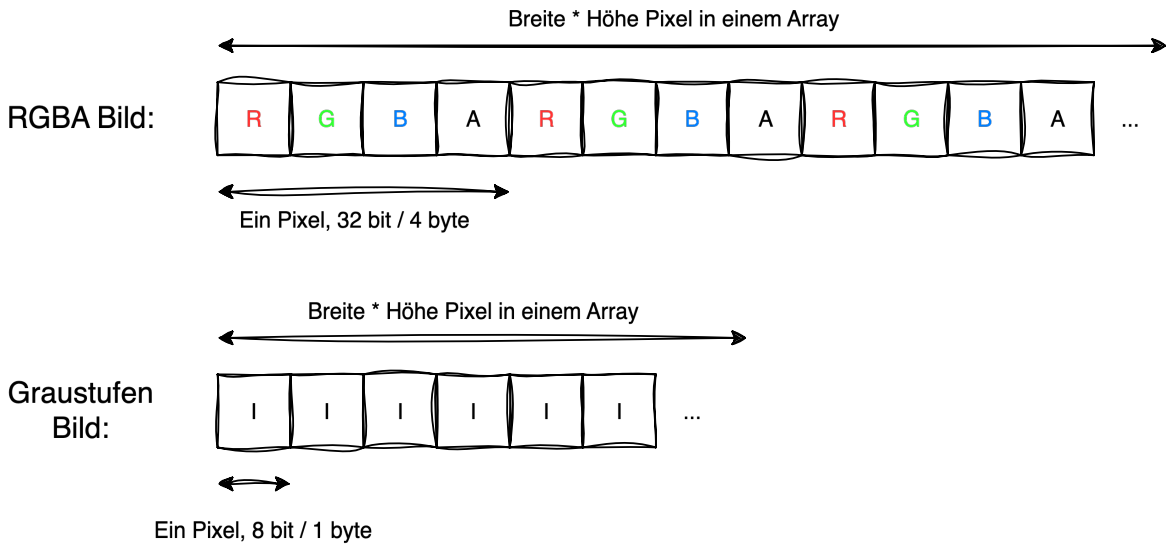
Bildbearbeitung

Das Programm soll einfache Routinen aus der Bildbearbeitung umsetzen: 1. Eine Umwandlung eines Farbbilds in ein Graustufen-Bild 2. Eine Kantenerkennung auf dem vorher umgewandelten Graustufen-Bild

Das sieht dann folgendermaßen aus:

Originalbild Graustufen Kantendetektion

Die konkreten Algorithmen sind in Aufgabe 1 und Aufgabe 2 erklärt, vorher müssen Sie noch wissen, wie die Bild-Daten im Speicher aussehen und behandelt werden. Das Programmgerüst übernimmt bereits eine Konvertierung des Eingabe-Bildes in ein einfaches Format, mit dem Sie arbeiten können. Das Format sieht folgendermaßen aus:



Ein Bild ist eine Ansammlung von Pixeln in einem 2D-Gitter der Maße `Breite` * `Höhe`. Aus Sicht des Computers können die Pixel jedoch auch in einem eindimensionalen Array gespeichert werden, jede Zeile im Bild beginnt dann an Offsets die ein ganzzahliges Vielfaches der `Breite` sind.

Ein Pixel selbst kann unterschiedliche *Formate* haben, wie hier dargestellt. Für Farbbilder sind die Formate `RGB` und `RGBA` üblich, da hier Farben aus den drei Grundkomponenten `Rot`, `Grün` und `Blau` aufgebaut werden können. `A` steht für `Alpha`, einen Transparenzwert, welcher halbdurchlässige Bilder ermöglicht. In unserem Programm verwenden wir das `RGBA` Format, wobei jede Komponente über ein Byte dargestellt wird. Jeder Pixel nimmt damit genau 4 Byte im Speicher ein, was die Verarbeitung einfacher macht als das `RGB`-Format, bei dem jeder Pixel 3 Byte einnimmt. Im Graustufen-Format gibt es nur einen einzigen Wert pro Pixel, die `I`-Komponente, die für `Intensität` steht, also die Helligkeit des Pixels darstellt. Hier ist jeder Pixel nur 1 Byte groß. Da die Pixel immer in einem Array liegen sollten Sie bereits Wege kennen, um die 4 Bytes eines `RGBA` Pixels und das eine Byte eines `I` Pixels in ein Register zur Verarbeitung zu laden.

Aufgabe 1 - Konvertierung zu Graustufen

In dieser Aufgabe sollen Sie einen Algorithmus implementieren, der ein `RGBA` Bild in ein Graustufen-Bild (`I`-Bild) konvertiert. Die dazugehörige Funktion soll in `asm/to_grayscale.s` implementiert werden und wird aus dem C++-Code wie folgt aufgerufen:

```
extern "C" void to_grayscale(
    const uint32_t *in, /* RGBA Pixel des Ausgangsbilds */
    uint8_t *out,       /* I Pixel des Graustufenbilds */
```

```
    uint32_t width, uint32_t height);

to_grayscale(rgbaPixels.data(), grayscalePixels.data(),
    uWidth, uHeight);
```

Die Arrays für das `RGBA` und `I` Bild werden vom C++-Code bereits angelegt, das `RGBA` Bild wird vom Dateisystem geladen und alles wird an die Funktion übergeben. Sie müssen also lediglich die Konvertierungsroutine vom `RGBA`-Array in das `I`-Array implementieren. Nach Aufruf des Programms wird im gleichen Verzeichnis neben dem Eingabe-Bild das Ergebnis der Graustufen-Konvertierung als neues Bild gespeichert, damit Sie ihre Resultate überprüfen können.

Implementieren Sie zuerst eine einfache Kopier-Routine, die lediglich ein spezifisches Byte aus dem `RGBA` Pixel als `I` Wert verwendet, also z.B. das `R` Byte. Nutzen Sie Shift- und Maskierungs-Operationen um den Wert zu extrahieren. Finden Sie heraus, welches Byte im `RGBA` Pixel welchem Farbkanal entspricht, also in welcher Endianness die Pixel gespeichert sind.

Wenn das funktioniert sind Sie bereit eine richtige Graustufen-Konvertierung umzusetzen. Hierzu gibt es mehrere verschiedene Algorithmen, die in der Praxis verwendet werden. Der einfachste ist das *gewichtete Mittel* der `R`, `G`, und `B` Komponenten (Transparenz ist für uns uninteressant):

```
I = (R + G + B) / 3
```

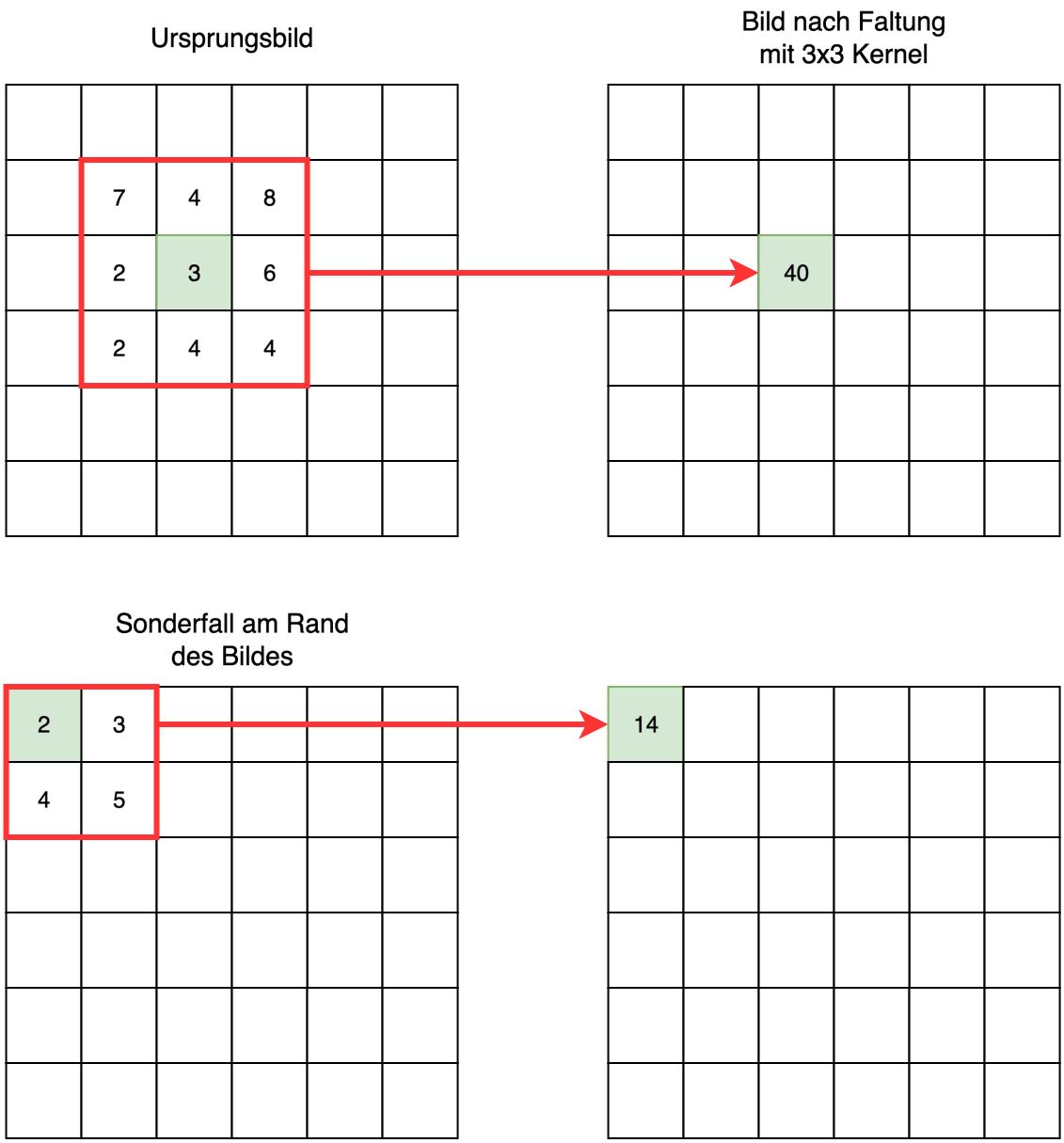
Da das menschliche Auge jedoch unterschiedlich empfindlich auf unterschiedliche Frequenzbänder reagiert erhält man hiermit nicht die besten Ergebnisse. Ein besseres Verfahren ist das *Luma Coding*, welches im TV-Bereich verwendet wurde. Hierbei werden die unterschiedlichen Farbkomponenten anders gewichtet:

```
I = 0.299*R + 0.587*G + 0.114*B
```

Implementieren Sie zuerst das gewichtete Mittel und diskutieren Sie dann (in Schrift-Form in ihrem Laborprotokoll), was die Berechnung des Luma Codings erschwert im Hinblick auf die ihnen bekannten ARM Befehle. (Hinweis: Wie geht die Berechnung mit Kommazahlen?)

Aufgabe 2 - Kantendetektion

Da Sie nun ein Graustufen-Bild generiert haben können Sie auf diesem Bild eine Kantendetektion durchführen. Hierfür gibt es ebenfalls viele verschiedene Algorithmen, die in der Regel sog. *Faltungen* (Eng. *Convolutions*) nutzen. Anders als bei der Graustufen-Berechnung, in der Sie jeden Pixel individuell betrachten konnten, benötigt die Kantendetektion bei jedem Pixel Informationen über die benachbarten Pixel. Das heißt wir bilden hier eine (gewichtete) Summe über eine Gruppe von Pixeln, zentriert um den aktuellen Pixel:



Die Gewichtung der Faltung wird über eine *Faltungsmaske* (Eng. *Kernel*) berechnet, in der Regel eine quadratische Matrix. Die Faltungsmaske im vorherigen Bild lautet:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Sie gewichtet jeden Pixel gleich und berechnet damit die Summe aller Pixel in einer 3x3

Nachbarschaft um den aktuellen (grün eingefärbten) Pixel. Für die Kantendetektion kann der sog. *Laplace-Operator* verwendet werden, die Faltungsmaske hierfür lautet:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Mathematisch gesehen berechnet Sie die Differenz zwischen dem aktuellen Pixel und seinen direkten Nachbarn. Da Kanten diskrete Sprünge im Bild darstellen, also Bereiche bei denen sich die Intensität sprunghaft ändert, ist dieser Differenzwert immer dann groß, wenn eine Kante vorliegt. Durch die negative Gewichtung kann es dabei auch zu negativen Resultaten kommen (das Vorzeichen gibt quasi die Richtung der Kante an). Da wir keine negativen Intensitätswerte darstellen können muss hier am Ende noch eine *Normalisierung* erfolgen. Diese können Sie umsetzen in dem Sie zuerst zum gefalteten Wert `128` dazuaddieren und den resultierenden Wert in den Wertebereich `[0; 255]` trimmen, z.B. durch eine bedingte Zuweisung von `0` für `x < 0` und `255` für `x > 255`.

Um auf die Pixel in der Nachbarschaft eines spezifischen Pixels `(x, y)` zuzugreifen, wenn alle Pixel in einem 1D-Array liegen, nutzen Sie folgende Identitäten:

- `Index(x, y) = y * Breite + x`
- `Index(x + n, y) = Index(x, y) + n` (auch für negative `n`)
  - **Achtung:** `x + n` darf nicht `< 0` oder `>= Breite` sein!
- `Index(x, y + m) = Index(x, y) + m * Breite` (auch für negative `m`)
  - **Achtung:** `y + m` darf nicht `< 0` oder `>= Höhe` sein!

Das schematische Bild der Faltung zeigt außerdem, dass es an den Rändern einer speziellen Behandlung bedarf, da es z.B. beim Pixel `(0,0)` (also dem Pixel in der oberen linken Ecke des Bildes) keine 8 benachbarten Pixel gibt, sondern nur 3. Dies müssen Sie in ihrem Code berücksichtigen, z.B. durch bedingte Befehlsausführung oder Sprünge. Summieren sie also am Rand nur die Pixel, auf die sie auch zugreifen können und ignorieren sie Pixel, die außerhalb des Bildes liegen würden.

Das Vorgehen ist also wie folgt:

- Berechnen Sie für jeden Pixel die 3x3 Nachbarschaft des Pixels, gewichtet mit der Laplace-Faltungsmaske
- Normalisieren Sie den resultierenden Wert in den Bereich `[0; 255]` wie oben beschrieben
- Schreiben Sie den normalisierten Wert in das Pixel-Array des Ausgabebilds

Zum Abschluss noch ein Blick auf die Funktionssignatur:

```
extern "C" void detect_edges(
    const uint8_t *in,  /* I Pixel des Graustufenbilds */
    uint8_t *out,       /* I Pixel des Kantenbilds */
    uint32_t width, uint32_t height);

detect_edges(grayscalePixels.data(), edgePixels.data(),
    uWidth, uHeight);
```