Digitale Bilder

Reguieg, Ibrahim Khaled Döring, Jules

Abbildung 1: *Jeffrey Smith 2008*

Zusammenfassung

In dieser Mitschrift wird beschrieben, wie digitale Bilder im Rechner dargestellt werden. In der Realität werden Bilder durch reflektiertes Licht erzeugt. Das führt zu den folgenden Fragestellungen:

„Wie stellen wir Farben und Bilder am Computer dar?“

„Wie werden Bilder im Computer repräsentiert?“

Hier soll ein Weg hergeleitet werden, Bilder in einer Form zu abstrahieren, mit der Computerprogramme effizient arbeiten können.

1 Was ist ein Bild?

Bilder sind stetig. Stetige Bilder sind allerdings kaum im Computer darstellbar, da dieser mit diskreten Werten arbeitet. Also müssen wir die stetigen Bilder in diskrete Bilder umwandeln.

1.1 Formal Mathematisch

Am besten dafür eignet sich die folgende stetige mathematische Funktion, die die Menge der Koordinaten R auf die Menge der Farben V abbildet:

$$I(x, y) : R \to V \tag{1}$$

Die Funktion (1) bildet die Menge der Koordinaten auf die Menge der Farben ab.

$$R \subset \mathbb{R}^2 \tag{2}$$

(2) besagt, dass die Farben R eine Teilmenge des kartesischen Produktes $\mathbb{R}$ x $\mathbb{R}$

$$V = \mathbb{R}^+ \; Graustufenbild \tag{3}$$

(3) Der Farbraum für ein Fraustufenbild entspricht den positiven reellen Zahlen.

$$V = \left(\mathbb{R}^+\right)^3 \; Farbbild \tag{4}$$

(4) Für eine Farbbild wird der Farbraum dem dreidimensionalen positiven Raum zugewiesen.

Diese Art der Modellierung ist unnötig genau. Durch optische Ungenauigkeiten, denen die menschliche Wahrnehmung unterliegt, findet eine sehr viel gröbere Abtastung statt. Diese kann man zur Vereinfachung des Modells nutzen.

1.2 Wahrnehmung bei Menschen

Je weiter ein Objekt entfernt ist, um so kleiner ist die Abbildung auf die Stäbchen und Zapfen unserer Netzhaut, da das Licht von mehreren Punkten des gesehenen Objekts auf einen Punkt auf der Netzhaut projiziert wird. Ist ein Objekt sehr nahe, wie bei 2 wird es deutlich auf den Zellen der Netzhaut abgebildet.

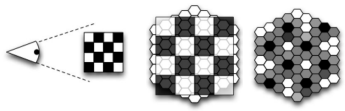

Abbildung 2: *Objekt nah am Auge.*
[0]

Somit hat das menschliche Auge, wie bei 3 keine Chance mehr die Unterschiede zwischen nahegelegenen Punkten wahrzunehmen.

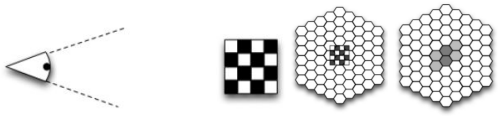

Abbildung 3: *Objekt fern vom Auge*
[0]

Auch beim Druck findet dieses Begebenheit Anwendung, beispielsweise druckt der Druckkopf ganz kleine Muster, um aus Gelb und Magenta Rot oder Orange zu mischen, wie auf Abbildung 4 zu sehen ist.

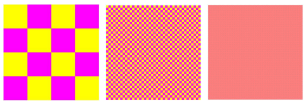

Abbildung 4: *Farben mischen aus CMYK.*
[0]

Ähnlich verhält es sich bei Bildschirmen, denn die mangelnde örtliche Auflösung führt zum Eindruck eines stetigen Bildes, wobei die Signale des Bildschirms eigentlich diskret sind.

2 Umsetzung für Computer

In diesem Abschnitt wird beschrieben, wie die Umsetzung der Bilder in er Programmierung aussehen. Naheliegend ist es, ein Bildgitter durch einen zweidimensionalen *byte* Array pro Farbkanal darzustellen. Nun kann jeder Subpixel bearbeitet werden, indem wir mit zwei geschachtelten For-Schleifen über die Koordinaten iterieren und dort die RGB-Farbwerte setzen.

Listing 1: *Bildgitter aus zweidimensionalen Byte Array pro Farbkanal*

```
final byte[][]pic = new byte[640][480][3];

for (int x = 0; x < width; ++x){
  for (int y = 0; y < height; ++y){
    byte r = pic [x][y][0];
    byte g = pic [x][y][1];
    byte b = pic [x][y][2];
    }
  }
```

2.1 Optimierung für CPU Caching

SRAM → ist *teuer* aber *schnell*
DRAM → ist *günstig* aber *langsam*

Auf der CPU befinden sich schnelle kleine SRAM basierte Speicher die als Caches benutzt werden. Cachecontroller versuchen zu errechnen auf welche Speicheradressen als nächstes zugegriffen wird und laden die Daten präventiv ins schnellere Cache. Dadurch muss die CPU seltener warten bis Daten aus dem langsamen Hauptspeicher auf DRAM Basis geladen sind. Modernen CPUs haben üblicherweise drei Cache-Level mit denen man versucht die *Cache Misses* zu minimieren.

[0]

2.2 Graustufenbild

Um *Cacheprediction* besser aus zu nutzen ist es also sinnvoll, Daten hintereinander abzulegen. Der folgende Code ist ein Beispiel für ein Graustufenbild als eindimensionales Array.

Listing 2: *Graustufenbild*

```
final byte[][] pic = new byte [width * height];

for (int x = 0; x < width; ++x) {
  for (int y = 0; y < height; ++y) {
    byte g = pic [y * width +x];
  }
}
```

2.3 Farbbild mit separierten Kanälen

Ähnlich kann man ein Farbbild modellieren, indem man die Farbkanäle einen nach dem anderen,wie in Abbildung XXX ablegt.

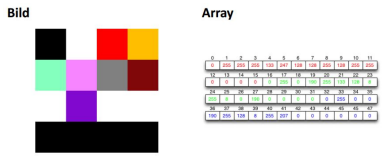

Abbildung 5: *Farbbild mit separierten Farbkanälen.*
[0]

Listing 3: *Farbbild mit separierten Kanälen*

```
final byte[][] pic = new byte [640 * 480 * 3];
SomeImageLoader.loadInto(pic);
for (int x = 0; x < width; ++x) {
  for (int y = 0; y < height; ++y) {
    final byte red = pic[0 * width * height
    + y * width + x ];

    final byte green = pic[1 * width * height
    + y * width + x ];

    final byte blue = pic[2 * width * height
    + y * width + x ];
    }
  }
```

2.4 Farbbild Pixelweise

Eine weitere Möglichkeit besteht darin, jeden Pixel durch 3 aufeinander folgende Subpixel darzustellen.

Listing 4: *Farbbild Pixelweise*

```
final byte[][] pic = new byte [640 * 480 * 3];

for (int x = 0; x < width; ++x){
  for (int y = 0 ; y < height; ++y){
    byte r = pic [y * width *3 + x*3 +0];
    byte g = pic [y * width *3 + x*3 +3];
    byte b = pic [y * width *3 + x*3 +2];
    }
  }
```

2.5 Farbbild Pixelweise als Integer

Die häufigste Darstellung ist allerdings die pixelweise Darstellung in einem Array vom Typ Integer, da sie mehrere Vorteile in sich vereint. Die Performanz wird einerseits durch *Data structure alignment* auf Speicherseite und andererseits durch Ausnutzen hochperformanter Shiftoperationen für Subpixelzugriffe gewährleistet. Zudem erhält man für Menschen verhältnismäßig gut interpretierbare Werte. [0]

Der Zugriff auf Subpixel findet mittels Bitmasken und Shiftoperationen statt.

Listing 5: *Farbbild Pixelweise als Integer*

```
final int height = 480;
final int width = 640;
final int[] pic = new int[height * width ];
SomeImageLoader.loadInto( pic );
for(int y = 0; y < height; ++y ) {
  for(int x = 0; x < width; ++x ) {
    int red = (pic[ y * width + x ] &
    0xff0000) >> 16;
    int green = (pic[ y * width + x ] &
    0xff00) >> 8;
    int blue = pic[ y * width + x ] & 0xff;

    pic[y * width + x ] =
    ((red & 0xff) << 16) |
    ((green & 0xff) << 8) |
    (blue & 0xff);
  }
}
```

3 Koordinatensystem

Bei allen beschriebenen Vorgängen ist zu beachten, dass das in der Computergrafik übliche Koordinatensystem mathematisch korrekt unten links beginnt und sich nach oben und rechts ausbreitet.

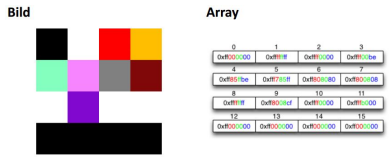

Abbildung 6: *Farbbild in pixelweiser Darstellung im Integer Array.*
[0]

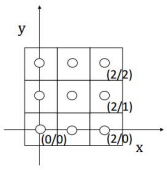

Abbildung 7: *Bilder in der Computergrafik.*
[0]

In der Bildverarbeitung hingegen beginnt es gemäß der Leserichtung der westlichen Kultur sowie der Richtung in CRT Monitoren, oben links und breitet sich nach rechts und unten aus.

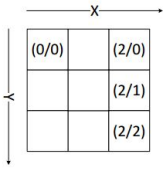

Abbildung 8: *Bilder in der Bildverarbeitung.*
[0]

Literatur

Ulrich Drepper. What every programmer should know about memory. *Red Hat, Inc*, 11, 2007.

Stephan Rehfeld. Digitale Bilder. `https://public.beuth-hochschule.de/~rehfeld/lehre/2015/ss/digitale-bilder.pdf`, 2015.

Eric Rowell. Know thy complexities! `http://bigocheatsheet.com/`, 2013.