MULTIMEDIADATENFORMATE

Skript zum Seminar im WS02/03

SKRIPT ZUM SEMINAR

Multimediadatenformate

Herausgegeben von: Gerald Friedland
Freie Universität Berlin
Institut für Informatik
Takustr. 9
14195 Berlin
fland@inf.fu-berlin.de

Stand: April 2003

Inhaltsverzeichnis

Grundlagen der Entropiekodierung

von

Tomas Schackert

Einleitung

Wenn man Daten, wie zum Beispiel Bilder oder Musik, platzsparend speichern will, dann kann man versuchen diese Daten zu komprimieren. Man kann grundsätzlich zwei Arten von Kompressionsverfahren unterscheiden:

> **verlustbehaftete (lossy) Komprimierung**: Hier werden Daten unter einer bestimmten Fehlerquote abgebildet. So können zum Beispiel Daten mit geringem Informationsgehalt weggelassen werden, ohne die Gesamtinformation zu sehr zu entfremden. Dadurch wird die Menge der zu speichernden Daten kleiner. Allerdings bildet diese Art der Komprimierung verschiedene Eingangsdaten auf die selben Ausgabedaten ab und ist daher nicht umkehrbar (fehlerbehaftet). Ein praktisches Beispiel für die Verwendung verlustbehafteter Komprimierung ist das Jpeg-Format.
>
> **verlustfreie (loosless) Komprimierung**: Hierbei werden die Eingabedaten so auf die Ausgabedaten abgebildet, dass die Ausgabe möglichst häufig kürzer ist als die Eingabe. Allerdings wird jeder Algorithmus dieser Klasse zu bestimmten Eingaben auch immer Ausgaben produzieren, die länger sind als die Eingabe (siehe unten).[1]

Dieser Text befasst sich mit der Klasse der verlustfreien Komprimierung. Wenn nicht besonders darauf hingewiesen wird, beziehen sich Ausdrücke wie Komprimierung und Algorithmus immer auf verlustfreie Komprimierung. Die verlustbehaftete Komprimierung wird nur am Rande betrachtet.

Wir wollen uns zunächst der oben erwähnten Tatsache zuwenden, dass für jeden möglichen Algorithmus Ausgaben existieren, die länger sind, als die ursprüngliche Eingabe. Um dies zu verstehen, betrachte man folgende Fragen:

1) Kann man ein Programm entwickeln, das jede Eingabe um mindesten n Byte (fehlerfrei) verkleinert?
2) Kann man ein Programm entwickeln, das jede Eingabe zumindest verkürzt (fehlerfrei)?

Die Antwort auf Frage 1) findet man ziemlich leicht, man stelle sich lediglich eine Eingabe mit n-1 Byte vor. Die Antwort auf Frage zwei ist vergleichbar mit der Aussage *„Wenn 16 Gegenstände in 15 Räumen liegen, dann liegen in mindestens einem Raum 2 Gegenstände.“*
Für die Komprimierung heißt das: wenn ich 2^n Eingaben auf 2^m Ausgaben abbilde und m<n ist, so gibt es mindestens zwei Eingaben, die auf die gleiche Ausgabe abgebildet werden. Ein Algorithmus, der dies verspricht, kann also unmöglich verlustfrei arbeiten.

Wenn man also vollkommen zufällig Eingaben auswählt und diese komprimiert, so wird ein Algorithmus Ausgaben produzieren, die im Durchschnitt bestenfalls gleichlang mit den Eingaben sind. Wie kann man dann überhaupt fehlerfrei komprimieren? Die meisten Eingaben sind in der Regel nicht komplett chaotisch. Sie übertragen Information und Information enthält auch meistens ein bestimmtes System.
Zum Beispiel verteilen sich in Texten die Buchstaben nicht zufällig und bestimmte Lautkombinationen oder Wörter treten häufiger auf als andere. Wenn man den Text nun durch einen anderen ersetzt (kodiert) und dabei den häufigeren Elementen kürzere Symbole zuordnet und den selteneren längere kann man Platz sparen.

Beispiel für komprimieren durch ersetzen:

[1] unter der Voraussetzung, dass a) die Identitaetsabbildung kein Kompressionsalgorithmus ist, b) die Eingabedatei immer endlich gross ist und c) kein zusaetzliches Symbol ins Bildalphabet aufgenommen wird.

Der folgende Text stammt von der Homepage zum Seminar in dessen Rahmen diese Ausarbeitung entstand:

Das Seminar beschäftigt sich implementierungsnah mit Konzepten und Strukturen aktuell gebräuchlicher Multimediadatenformate. Zum Inhalt des Seminars gehören sowohl die zu Grunde liegenden mathematischen Konzepte und Algorithmen, als auch konkrete Implementationen gängiger Multimediaformate.

Dieser Text enthält 291 Zeichen. Speichert man ihn im ASCII-Format bedeutet das, er benötigt 291 Byte. Wenn man nun jedes Auftreten von „ö“ durch „\ö“, und jedes Auftreten von „me“ durch „ö“ ersetzt so erhält man:

Das Seminar beschäftigt sich impleöntierungsnah mit Konzepten und Strukturen aktuell gebräuchlicher Multiödiadatenformate. Zum Inhalt des Seminars gehören sowohl die zu Grunde liegenden mathematischen Konzepte und Algorithön, als auch konkrete Impleöntationen gängiger Multiödiaformate.

Im Gegensatz zum Original enthält dieser Text nur 287 Zeichen, es wurden also 4 Byte gespart. Das diese Umformung verlustfrei umkehrbar ist dürfte leicht ersichtlich sein.

Auch in Bildern lässt sich auf ähnliche Weise ein Algorithmus zum Speichern entwickeln. Hier liegen meist ähnliche Farben dicht bei einander. Vorrausgesetzt, es handelt sich um ein Bild mit hinreichend geringem Farbumfang (ideal sind Schwarz-Weis-Bilder), treten dann Ketten von gleichen Werten auf. Statt nun jedes Pixel einzeln zu speichern, kann man vor jedem Farbwert angeben, wie oft sich der Farbwert wiederholt. Dieses Verfahren wird als **Run-Length-Encoding (RLE)** bezeichnet.

Beispiel für Run-Length-Encoding:

W stehe für ein weißes Pixel, R für ein Rotes, r für ein rosa Pixel

lossless:	*WWRRRrrrRRW*	=	*2W3R3r2R1W*
lossy: (r=R)	*WWRRRrrrRRW*	=	*2W8R1W*

Auch hier liegt die eindeutige Umkehrung, zumindest im Fall der verlustfreien Kodierung klar auf der Hand.

Die Rate, mit der eine bestimmte Eingabe chaotische Züge aufweist, kann man durch den **Informationsgehalt** der Eingabe (**Entropie**) ausdrücken:

Wenn man mit sehr hoher Sicherheit von den vorrangegangenen Zeichen auf das nächste Zeichen schließen kann, dann ist die Information, dass das folgende Zeichen das erwartete ist, nur wenig Wert. Tritt hingegen ein unerwartetes Zeichen auf, so hat dieses einen hohen Informationswert.

Beispiel für Informationswert von Zeichen in Worten:

Es wird ein Wort aus der Menge {Bier, Biene, Biest, Biologe} gesucht. Die Beispielworte A und B werden nun zeichenweise von links nach rechts gelesen und die einzelnen Zeichen auf ihren Informationsgehalt hin untersucht.

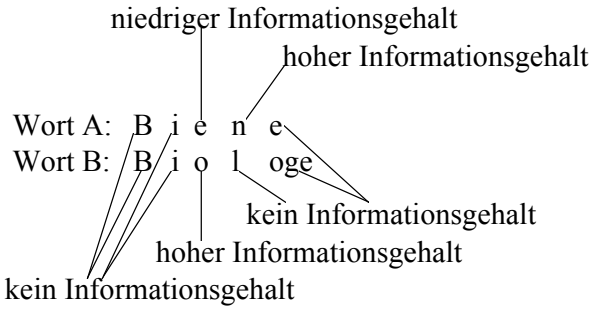


Abbildung 1: Beispiel für Zeichenentropie

Somit trägt „Bi“ keinerlei Information. Um das Wort zu erkennen, können die ersten beiden Zeichen ohne weiteres ignoriert werden.

Beim dritten Zeichen hat „e“ einen niedrigen Informationsgehalt. Es ist sehr wahrscheinlich, dass das dritte Zeichen ein „e“ ist, aber dadurch schränkt es die Wortmenge auch nicht besonders stark ein. Man braucht mehr Informationen (den 4. Buchstaben), um das Wort zu erkennen.

„o“ aber hat hier einen hohen Informationsgehalt. Weil „o“ nämlich so selten vorkommt, reicht alleine sein Auftreten aus, um das Ergebnis zu identifizieren.

Komprimierverfahren, die die Untersuchungen über Entropie nutzen, heißen **Entropiekodierung**.

Der Quellenkodierungssatz (Source Coding Theorem)

Wir wollen nun die Frage untersuchen, in wieweit man Texte komprimieren kann, ohne Informationen zu verlieren.

Basisformeln

Um der Quellenkodierungssatz zu verstehen, benötigen wir einige Variabeln, die hier mit ihren wichtigsten Eigenschaften zusammengefasst sind.

$P(x_i)$ -**Wahrscheinlichkeit** für x $P(x,y) = P(x) * P(y)$

X -Menge der möglichen Ergebnisse mit ihren dazugehörigen Wahrscheinlichkeiten

$h(x)$ -**Shannon information content** $h(x) = \log_2 \frac{1}{P(x)}$ (in Bit)

$$h(x,y) = \log_2 \frac{1}{P(x,y)} = \log_2 \frac{1}{P(x) * P(y)} = \log_2 \frac{1}{P(x)} + \log_2 \frac{1}{P(y)} = h(x) + h(y)$$

$H(X)$ -Entropie; Informationsgehalt für eine bestimmte Menge $H(X) = \sum_i P(x_i) * h(x_i)$

Der Informationsgehalt h(x)

Was bedeutet nun $h(x) = \log_2 \frac{1}{P(x)}$?

Beispiel für den Informationsgehalt:

Gegeben sei eine 64-Bit lange Kette, die nur Nullen enthält bis auf eine 1, die an beliebiger Stelle steht. Von einer Seite her werden nun die Bits auf ihren Informationsgehalt h hin untersucht.

i	1	2	3	4	....	32	33	...	62	63	64
$P(b_i = 0)$ *	$\frac{63}{64}$	$\frac{62}{63}$	$\frac{61}{62}$	$\frac{60}{61}$		$\frac{32}{33}$	$\frac{31}{32}$		$\frac{2}{3}$	$\frac{1}{2}$	0
$P(b_i = 1)$ *	$\frac{1}{64}$	$\frac{1}{63}$	$\frac{1}{62}$	$\frac{1}{61}$		$\frac{1}{33}$	$\frac{1}{32}$		$\frac{1}{3}$	$\frac{1}{2}$	1

$h(b_i = 0)$ *	0,0227	0,023	0,0235	0,0238		0,0444	0,0458		0,585	1	%
$h(b_i = 1)$ *	6	5,977	5,954	5,9307		5,0444	5		1,585	1	0

* unter der Vorraussetzung, dass noch keine 1 gefunden wurde

Man erkennt:

1) der Informationsgehalt eines sicheren Ereignisses ist 0 siehe $h(b_{64} = 1) = 0$

2) der Informationsgehalt von $h(b_1 = 1) = 6$

3) die Summe der Informationsgehalte der Bits 1 bis n ist immer genau dann 6, wenn das n-te Bit eine 1 ist, Beispiel: h(0001)= 0,0227+0,023+0,0235+5,9307=6. Logischerweise ist der Informationsgehalt aller weiteren Stellen 0 da, ja nie eine weitere 1 auftreten darf (nach Vorraussetzung) und es sich daher um sichere Ereignisse handelt:

$$\sum_{n=1}^{i} h(b_n) = \left(\sum_{m=66-i}^{64} \log_2 \frac{m}{m-1}\right) + \log_2 \frac{1}{1/(65-i)} = \log_2\left(\left(\prod_{m=66-i}^{64} \frac{m}{m-1}\right) * (65-i)\right) = 6$$

4) je unwahrscheinlicher ein Ereignis ist, desto höher ist sein Informationsgehalt wenn es eintrifft

1. Hauptsatz der Quellenkodierung (Shannon's Source Coding theorem)

Bisher haben wir im allgemeinen die Werte verlustfrei betrachtet. Wenn wir also ein Element aus 256 verschiedenen Ergebnissen speichern wollten, haben wir 8 Bit= 2^8 =256 verwendet. Dies bezeichnet man als den **raw bit content**: Sei A die Anzahl der möglichen Eingaben einer Variable X, dann ist $L_0(X) = \lceil \log_2 A \rceil$ die Anzahl der Bits, in denen sich ein Ergebnis immer speichern lässt. Die Länge mehrerer Variabeln ist additiv, also: $L_0(X,Y) = L_0(X) + H_0(Y)$.

Will man aber die Werte nicht verlustfrei betrachteten so schränkt das die Menge der möglichen Ergebnisse ein. Hierfür verwendet man dann den **essential bit content**: $L_\delta(X) = \lceil \log_2 |S_\delta| \rceil$ wobei $S_\delta = \{x \mid x \in X \wedge P(x) \geq 1-\delta\}$. Das bedeutet $L_\delta(X)$ ist die Anzahl von Bits die man benötigt, um ein Ergebnis zu speichern, wenn man alle Ergebnisse die unwahrscheinlicher als δ sind nicht berücksichtigt. Beispiel: In einem durchschnittlichen deutschen Text könnte man das Zeichen x entfernen ohne den Informationsgehalt des Gesamttextes zu sehr zu beschädigen. Es handelt sich dann aber nicht mehr um eine verlustfreie Komprimierung.

Betrachtet man nicht einzelne Elemente aus der Menge X aller Ereignisse, sondern Ketten solcher Elemente so folgt für ein gegebenes $0 < \delta < 1$ und $\varepsilon > 0$: Es existiert ein N_0 so das für alle $N > N_0$ gilt der 1. Hauptsatz der Quellenkodierung:

$$H(X) - \varepsilon < \frac{1}{N} L_\delta(X^N) < H(X) + \varepsilon \qquad \text{(also auch } \left| \frac{1}{N} L_\delta(X^N) - H(X) \right| < \varepsilon \text{)}$$

wobei X^N eine N-elementige Kette von Werten aus X ist und $H(X) = \sum_i P(x_i) * h(x_i)$ die Entropie.

Das heißt: für jede gewählte Fehlerquote nähert sich die durchschnittliche Bitlänge L je Element beliebig nah an H(X) an, wenn man sehr lange Ketten aus Elementen bildet. Für $N \to \infty$ kann man L genau auf H senken und das bei beliebigem δ, also auch annähernd verlustfrei. **Daher kann man davon ausgehen, dass jede Komprimierung die auf weniger als N*H(X) Bits komprimiert, verlustbehaftet ist.** Die Entropie bildet eine **untere Schranke** für die verlustfreie Komprimierung von Daten.

Eine wichtige Erkenntnis ist, dass für lange Ketten schon ein verschwindend kleines δ genügt, um L deutlich von $L_0(X)$ auf $H(X) + \varepsilon$ zu senken. Und dass auch bei sehr hohem δ nie weniger als $H(X) - \varepsilon$ Bits verbraucht werden.

Beispiel: Berechnung der Entropie für ein Alphabet mit 8 Zeichen

i	1	2	3	4	5	6	7	8
x_i	a	b	c	d	e	f	g	h
$P(x_i)$	1/4	1/4	1/4	3/16	1/64	1/64	1/64	1/64
$h(x_i)$	2	2	2	2,415	6	6	6	6

$$L_0(X) = \log_2 8 = 3$$

$$H = 3 * (\frac{1}{4} * 2) + (\frac{3}{16} * 2{,}415) + 4 * (\frac{1}{64} * 6) \approx 2{,}3278$$

Ich kann also, egal welchen Algorithmus ich verwende, eine Kette von Elementen aus X nie mit weniger als 2,3278 Bit pro Element verlustfrei komprimieren.

Das Wahrscheinlichkeitsmodell

Einige Kompressionsalgorithmen, wie die weiter unten vorgestellten Symbolcodes oder die arithmetische Kodierung, verwenden ein explizit anzugebendes Wahrscheinlichkeitsmodell. Die Darstellung hier ist auf das einfachste mögliche Modell beschränkt, jedem Zeichen eine bestimmte, feste Wahrscheinlichkeit zugeordnet ist.
Es ist jedoch durchaus möglich, andere, anwendungsspezifische Wahrscheinlichkeitsmodelle oder auch adaptive Verfahren zu verwenden, um eine bessere Komprimierung zu erreichen. Man muss lediglich sicherstellen, dass Komprimierung und Dekomprimierung das gleiche Wahrscheinlichkeitsmodell verwenden. Beispiele:

- adaptiv: Man möchte Eingaben kodieren, bei denen man davon ausgehen kann, dass bestimmte Zeichen wesentlich häufiger vorkommen als andere, ohne jedoch vorher zu wissen, ob ein Zeichen bei einer bestimmten Eingabe nun häufig vorkommt oder nicht. Hier wäre ein Modell möglich, das die Wahrscheinlichkeit für ein Zeichen proportional zu seiner Häufigkeit im bisherigen Text berechnet.

z.B.: „abab“ =>P(b)=0,5 aber „abbb“=>P(b)=0,75

- anwendungsspezifisch: Bei Texten für eine bestimmte Sprache können für die Wahrscheinlichkeit eines Zeichens vorrangegangene Zeichen berücksichtigt werden. Dabei könnten nicht nur häufige Folgezeichen berücksichtigt werden (z.B.: ch und ei), sondern mit Hilfe einer Wörterliste könnten ganze Wortanfänge verglichen werden.
  z.B. „Mod“
  - Wörter: Mode, Modell, Moderator, Modergeruch, moderieren, moderig, modern, modisch, Modul
  - e erhält höchste Wahrscheinlichkeit, alle außer i und u sehr geringe Wahrscheinlichkeit
  - Kodierung von e sehr effizient, Kodierung aller anderen Zeichen trotzdem möglich

Problem: Encoder/Decoder müssen die gleiche Wortliste verwenden, da sonst unterschiedliche Wahrscheinlichkeiten berechnet werden.

Symbol Codes

Eine Methode um eine Eingabe zu komprimieren ist, dass man jedem Zeichen des Quellalphabets mit konstanter Länge (z.B. 8 Bit) einen bestimmten Symbol Code zuordnet, der für häufige Zeichen kürzer, für seltenere Zeichen dafür länger ist. Für diese Ersetzung sollte gelten:

- Der erzeugte Code muss eindeutig decodierbar sein, dass heißt, zwei verschiedene Eingaben haben auch immer zwei verschiedene Ausgaben.
- Der Symbolcode sollte einfach zu decodieren sein, dass heißt, wenn man den zu decodierenden String Bit für Bit liest, muss man das Ende eines Symbol Codes erkennen können, sobald man das letzte Bit dieses Codes liest. (Das Problem besteht in der unterschiedlichen Länge der Symbol Codes im selben Alphabet). Jeder Präfixcode erfüllt diese Bedingung.
- Der Code sollte so kompakt wie möglich sein.

Kraft-McMellan-Ungleichung

Bei Codes mit konstanter Länge ist die Anzahl der möglichen Zeichen eines Alphabetes leicht aus der Anzahl der Bits abzuleiten. Wie viele Codes kann man aber mit einer variablen Länge $l(x_i) < n$ erzeugen?

Für alle Codes muss gelten $\sum_i \frac{1}{2^{l(x_i)}} \leq 1$

Das heißt, jeder Code hat ein bestimmtes Gewicht und je kürzer der Code desto höher das Gewicht. Ein Code der Länge 1 hat ein Gewicht von 1/2, während ein Code der Länge 3 nur ein Gewicht von 1/8 besitzt. Man kann also solange Codes zu seinem Alphabet hinzufügen wie die Summe aller Gewichte kleiner 1 ist. Ist dieses Maximum erreicht, so bezeichnet man den Symbol Code auch als **vollständig**.
Man kann sich das bildlich leicht an einem Präfixbaum verdeutlichen. Dabei bildet der Wert jedes Knotens den Präfix seiner Kinder. Verwendet man den Wert eines bestimmten Knotens als Code so bedeutet das:

1. keiner der Elternknoten kann noch als Code dienen, da dieser sonst ein Präfix des aktuellen wäre.
2. keiner der Kinderknoten kann noch als Code dienen, da diese den aktuellen Code als Präfix besitzen.

Mit der Kraft-McMellan-Ungleichung sind wir nun zwar in der Lage zu prüfen, ob ein Symbolcode vollständig ist und wenn nicht, welche Codelängen wir dem Symbolcode noch hinzufügen können. Wie bestimmt man nun aber die **optimale Bitlänge** für ein bestimmtes Zeichen?
Hauptsatz der Quellenkodierung sagt uns wie weit wir den Text höchstens verlustfrei komprimieren können. Es ist also nicht wirklich überraschend, dass die optimale Bitlänge für jeden Code durch den von Shannon definierten Informationsgehalt (Shannon information content: h(x)) gegeben ist. Das alleine impliziert allerdings noch keinen Algorithmus für einen optimalen Code, besonders da $h(x_i)$ auch eine gebrochene Zahl sein kann, wir aber nur vollständige Bits speichern können.

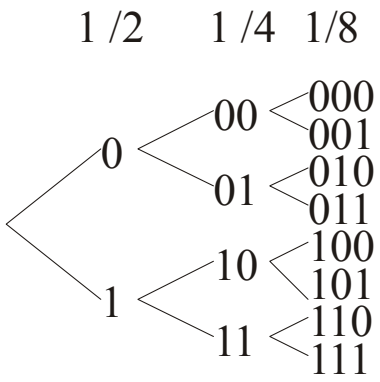


Abbildung 2: Codeslängen und ihre Gewichte

Huffman coding

Vorrausgesetzt man hat eine Tabelle mit den Wahrscheinlichkeiten für alle Zeichen des Eingabealphabetes. Wie erzeugt man nun den optimalen Symbol Code? Verwendet man viele kürzere Codes, dann entstehen zu viele sehr lange Codes, möglicherweise auch von Zeichen die häufiger vorkommen.
Der einfachste Weg, um einen annähernd optimalen Symbol Code zu erhalten, ist der Huffman-Algorithmus.
Dabei werden zunächst alle Zeichen als einfache Bäume repräsentiert, die nur aus einem Blatt bestehen. Der Knoten jedes dieser Bäume enthält außerdem den Wahrscheinlichkeitswert für das entsprechende Zeichen. Danach werden solange jeweils zwei Bäume zusammengefasst, bis nur noch ein einziger Baum existiert. Bei jedem Zusammenfassen erhält die neue Wurzel den Wert der der Summe der Teilbäume entspricht.
So entsteht für ein Alphabet

	a	b	c	d	e
P(x)	0,25	0,25	0,2	0,15	0,15

der rechts stehende binäre Baum. Die Binärdarstellung der Zeichen entspricht dann dem Weg von der Wurzel zu dem entsprechenden Blatt, wobei zum Beispiel der Schritt zum linken Kind mit einer 0 und der Schritt zum rechten Kind mit einer 1 notiert wird. Dann ergeben sich für den rechten Baum:

	a	b	c	d	e
Symbolcode	00	10	11	010	011

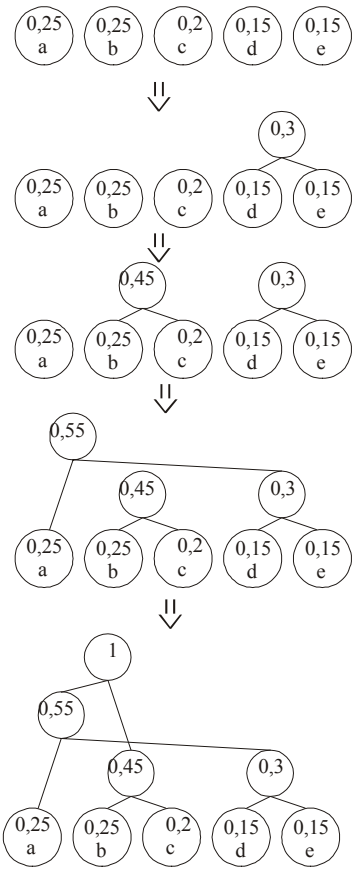


Abbildung 3:Erstellen eines Huffman-Tree

Effizienz des Huffman-Algorithmus

Es stellt sich natürlich die Frage, ob der so erzeugte Symbolcode wirklich der bestmögliche ist. Zum Beweis dazu muss man zeigen, dass das Zusammenfassen der beiden Teilbäume mit der geringsten Wahrscheinlichkeit zu einem Baum, und damit die Vergabe der gleichen Codelänge für die beiden Teilbäume, nicht zu einer höheren Codelänge führen kann, als sie bei irgendeinem anderen Symbolcode verwendet wird.
Seien a und b die Symbole mit den geringsten Wahrscheinlichkeit $P(a)$ und $P(b)$. Sie erhalten also beim Huffmancode die gleiche Codelänge. Angenommen es gäbe einen besseren (optimalen) Code der für diese beiden Symbole eine unterschiedliche Codelänge $len(a)$ und $len(b)$ enthält mit $len(a) < len(b)$. Da ein optimaler Code auf alle Fälle die Kraft-McMellan-Ungleichung erfüllt, muss es in dem optimalen Codebaum ein Symbol c geben, das mindestens die gleiche Länge wie b besitzt $len(c) \geq len(b)$. Aus der Bedingung, dass a und b die kleinsten möglichen Wahrscheinlichkeiten haben, folgt aber auch, dass $P(c) > P(a)$ gelten muss. Nun kann dieser Codebaum aber unmöglich optimal sein, da das unwahrscheinlichere Zeichen a eine kürzere Codelänge als das wahrscheinlichere Zeichen c hat. Würde man nämlich die Codes für a und c tauschen, würde die durchschnittliche Ausgabelänge um $(P(a) - P(c)) * (len(c) - len(a))$ sinken. Das wiederspricht der Annahme dass der zweite Code optimal ist, **daher gibt es keinen optimaleren Symbolcode als den von Huffman erzeugten**.

Effizienz von Symbolcodes

Für einen optimalen Symbolcode liegt die Codelänge für ein Zeichen bei

$$h(x) \leq l(x) \leq h(x) + 1$$

Daraus folgt für die Durchschnittslänge l_H der Zeichen eines codierten Textes:

$$H(X) \leq l_H \leq H(X) + \frac{1}{b}$$

wobei b die Konstante der Blockgröße ist. Im einfachsten, bis her verwendeten Fall ist b=1. Zur Blockgröße siehe auch III.5) Probleme von Huffman.

Probleme von Huffman (und Symbolcodes)

Ein Problem des Huffman-Algorithmus in der praktischen Anwendung ist, dass entweder für jede Eingabe ein extra Huffman Code erzeugt werden, und dieser dann natürlich auch mit der kodierten Ausgabe gespeichert werden muss, oder aber man verwendet einen festen, dafür aber nicht optimalen Code für jede Eingabe (Varianten wie adaptive Huffmancodes sind natürlich auch möglich).

Ein weiteres Problem besteht darin, dass Huffman nur die Wahrscheinlichkeit einzelner Zeichen verwendet, nicht aber die Wahrscheinlichkeit für Blöcke von Zeichen. So ist in einem Text ein i nach einem e wesentlich wahrscheinlicher als nach einem c. Umgekehrtes gilt für das h. Natürlich kann man auch den Huffman-Baum für Blöcke von 2 oder mehr Zeichen verwenden, jedoch erhält der Baum dadurch eine enorme Größe und man muss u.U. viele Kombinationen betrachten die überhaupt nicht vorkommen.

Nicht zuletzt muss man berücksichtigen, dass bei Huffman immer nur ganze Bits für einen Code verwendet werden können. Wenn nach Shannon ein Zeichen nur einen sehr geringen Informationswert hat (z.B. 0,2 Bit) wird trotzdem mindestens ein Bit verwendet, um die

Information zu speichern. Das Verwenden von Blöcken verschiebt das Problem zwar teilweise, aber es löst es nicht.

Arithmetic Coding

Bereits im Jahr 1948 wurden von Shannon die ersten Ansätze zur arithmetischen Kodierung veröffentlicht. 1963 wurde von Elias und Abramson die erste Implementierung veröffentlicht und erst 1976 wurden von Pasco und Rissanen praktische Lösungen dieses Algorithmus gefunden.
Die Arithmetische Kodierung umgeht die Beschränkung nur einzelne Symbole oder Symbol-Blöcke durch eine ganzzahlige Anzahl von Bits darzustellen, indem für den gesamten zu komprimierenden Datenstrom ein einziger Code berechnet wird (**Streaming Code**). Dadurch wird eine hohe Effizienz auch in den Fällen erreicht, die für die Huffman Kodierung ungünstig sind.

Bei der Arithmetischen Kodierung wird jede Eingabe auf eine Zahl im Intervall $[0,1) \subset \mathrm{R}$ abgebildet. Da das Intervall überabzählbar viele Elemente enthält, kann man jeder Sequenz eine reelle Zahl zuordnen. Aus dem selben Grund ist es notwendig das Ende der Eingabe durch ein besonderes Steuerzeichen (im folgenden #) zu signalisieren, da sonst der Decodierprozess nicht terminiert.

Kodierung/Dekodierung

Um eine Eingabe zu kodieren, betrachtet man zunächst das Startintervall $[0,1)$. Man zerlegt es in Partitionen deren Größe der Wahrscheinlichkeit für die einzelnen Zeichen entspricht. Wähle das Intervall, das dem ersten Zeichen entspricht und partitioniere diesen Bereich wieder entsprechend der Wahrscheinlichkeit für das nächste Zeichen. Wiederhole dies für alle übrigen Zeichen und wähle aus dem Endintervall die kürzeste Binäre Zahl.

Beispiel: Komprimierung von “bac#“ mit der arithmetischen Kodierung

x_i	a	b	c	#
$P(x_i)$	0.5	0.2	0.2	0.1

lies b

Partitionen (dezimal)	$[0,0.5)$	$[0.5,0.7)$	$[0.7,0.9)$	$[0.9,1)$
Partitionen (binär)	$[0,0.1)_2$	$[0.1,0.1\overline{0110})_2$	$[0.1\overline{0110},0.11\overline{1100})_2$	$[0.11\overline{1100},1)_2$

lies a

Partitionen (dezimal)	$[0.5,0.6)$	$[0.6,0.64)$	$[0.64,0.68)$	$[0.68,0.7)$
Partitionen (binär)	$[0.1,0.1\overline{1001})$	$[0.\overline{1001},0.1010001\ldots)_2$	$[0.1010001\ldots,0,1010111\ldots)_2$	$[0.1010111\ldots,0,\overline{1011}0)_2$

lies c

Partitionen (dezimal)	$[0.5,0.55)$	$[0.55,0.57)$	$[0.57,0.59)$ $\left[\begin{matrix}0.100100011\ldots,\\ 0.100101110\ldots\end{matrix}\right)_2$	$[0.59,0.6)$

lies #				
Partitionen (dezimal)	$[0.57,0.58)$	$[0.58,0.584)$	$[0.584,0.588)$	$[0.588,0.59)$ $=\begin{bmatrix}0.100101101...,\\0.100101110...\end{bmatrix}_2$

„bac#“ entsprich also einer Zahl im dezimalen Intervall $[0.588,0.59)$ oder binär $[0.100101101...,0.100101110...)$.
Genauer entspricht dieses Intervall gerade allen Zeichenketten die mit „bac#“ beginnen. Da nach dem Auslesen von # der Decoder das Ende erkennt, kann „bac#“ als eine beliebige Zahl im Intervall z.B. als 0,10010111 repräsentiert werden. Da das Startintervall die 1 nicht enthält, speichert man nur die Nachkommastellen: 10010111.

Bei der Decodierung erzeugt man ebenfalls vom Startintervall $[0,1)$ ausgehen die Partitionen und wählt dann jeweils das Unterintervall in dem der Code liegt. Das Verfahren wird wiederholt bis das Steuerzeichen für das Dateiende erreicht wird.

Grundlagen der Implementierung und Probleme

Das Hauptproblem bei der praktischen Umsetzung der arithmetischen Kodierung ist die endliche Genauigkeit von Computern. Wie man bereits im Beispiel weiter oben gesehen hat, werden die Intervalle sehr schnell sehr klein. Es ist daher nicht möglich, einfach Fließkommazahlen bei der Umsetzung zu verwenden, da die Intervall Grenzen genau beachtet werden müssen.

Um das Problem zu lösen, wird eine Skalierung des Intervalls verwendet (**Scaling Methode**). Mit Skalieren ist gemeint, dass das Intervall vergrößert wird. Das Vergrößern des Intervalls führt jedoch zu einem Verlust von Informationen über die bisher kodierte Sequenz.

Man kann die Gruppe der Ursprungsintervalle von $[0,1)$ in 2 Teile zerlegen: $[0,0.5)$ und $[0.5,1)$. Alle Zahlen größer oder gleich 0.5 haben, wenn man sie binär kodiert, eine führende 1. Dagegen haben alle Zahlen unterhalb von 0.5 eine führende 0 bei der binären Darstellung. Da jedes neue (Unter-)Intervall ein Teilbereich des vorherigen Intervalls ist, kann bei fortschreitender Kodierung eine Hälfte nicht mehr verlassen werden, sobald das Intervall völlig in einer Hälfte liegt. Zu diesem Zeitpunkt kann man mit Sicherheit das erste Zeichen des Codes bestimmen. Das heißt, man kann dieses Bit schon übertragen und das aktuelle Intervall skalieren. Skalieren bedeutet ein Vergrößern des aktuellen Intervalls auf das Ursprungsintervall. Dabei geht die Information über das höchstwertigste Bit zwar verloren, dies konnte, wie gezeigt, aber schon übertragen werden.
Für den Fall, dass das Intervall gegen 0.5 konvergiert, kann man jedoch zunächst kein führendes Bit festlegen. Man muss das Auftreten eines der beiden oberen Fälle abwarten und kann danach erst die Bits schreiben.
Die Skalierung des Intervall I sieht im Detail wie folgt aus:
für $I \subset [0,0.5)$ verwende $E_1 : [0,0.5) \to [0,1); E_1(x) = 2 * x$ (und schreibe eine 0 in die Ausgabe)
für $I \subset [0.5,1)$ verwende $E_2 : [0.5,1) \to [0,1); E_1(x) = 2 * (x - 0.5)$ (und schreibe eine 1 in die Ausgabe)

für $I \not\subset [0,0.5) \wedge I \not\subset [0.5,1) \wedge I \subset [0.25,0.75)$ verwende $E_3 : [0.25,0.75) \to [0,1); E_1(x) = 2*(x-0.25)$

Welche Bits müssen nun bei E3 geschrieben werden? Wie man leicht sieht, entspricht die Ausführung von E3 gefolgt von El der Ausführung von E1 gefolgt von E2.

Allgemeiner: $E_1 o (E_3)^n = (E_2)^n o E_1$. Äquivalent gilt $E_2 o (E_3)^n = (E_1)^n o E_2$. Nun ist klar welche Bits zu schreiben sind. Folgt nach n E3 mappings ein E1 schreibe eine 0 und n 1en (also 01^n). Folgt ein E2 schreibe 10^n.

Beispiel für E3-mapping:

Um diese Umformung etwas zu verdeutlichen, betrachte man das folgende Beispiel zweier E3-Umformungen gefolgt von einer E1 Umformung (von oben nach unten). Die grünen Linien markieren dabei den Übergang der Grenzen des Ursprungsintervall (oben) in das skalierten Intervall (unten):

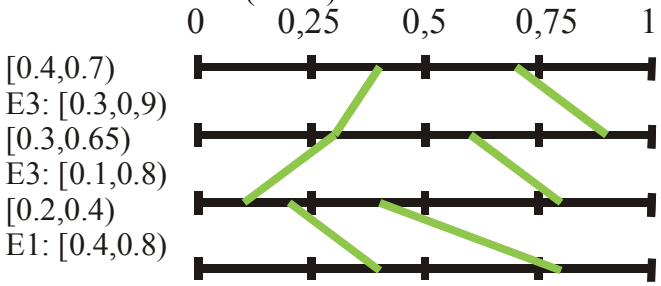


Abbildung 4: Beispiel für E3-mapping

Will man die E3-mappings ersetzen, so muss man die roten Intervallgrenzen benutzen. Es ist offensichtlich, dass diese das Intervall $[0,1)$ verlassen, aber nach der letzten Umformung sind die Intervallgrenzen wieder identisch:

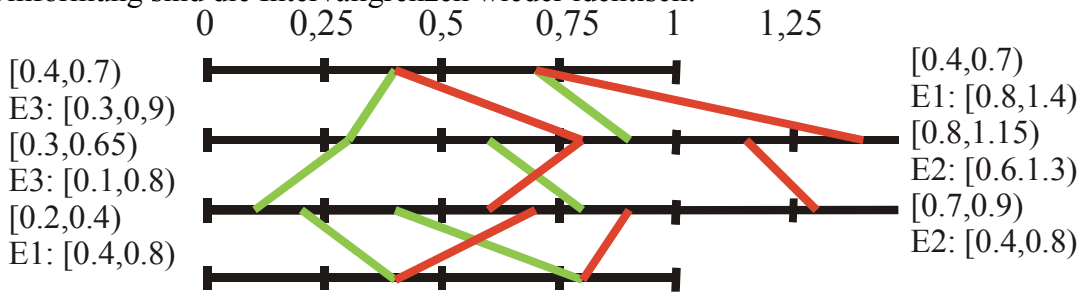


Abbildung 5: Interpretation des E3-mapping

Wichtig ist zu erkennen, dass ggf. auch mehrere Skalierungen pro gelesenem Zeichen durchgeführt werden müssen. Erst wenn das Intervall nicht mehr in einem der drei Bereiche liegt, kann man mit dem nächsten Zeichen fortfahren.

Effizienz der Arithmetischen Kodierung

Wie man sich überzeugen kann, ist die Länge für die Binärdarstellung eines einzelnen Zeichens in der Arithmetischen Kodierung

$$l(x) \le \lceil h(x) \rceil + 1 = \left\lceil \log_2 \frac{1}{P(x)} \right\rceil + 1 \le \log_2 \frac{1}{P(x)} + 2$$

Sei nun X^m eine Folge von m Zeichen aus X. Dann folgt für deren Länge:

$$\begin{aligned}
l(X^m) &= \sum_{x \in X} \left(P(x) * l(x) \right) \\
&\le \sum_{x \in X} \left(P(x) * \left(\log_2 \frac{1}{P(x)} + 2 \right) \right) \\
&= \sum_{x \in X} \left(P(x) * \left(\log_2 1 - \log_2 \left(P(x) \right) + 2 \right) \right) \\
&= \sum_{x \in X} \left(P(x) * \left(- \log_2 \left(P(x) \right) + 2 \right) \right) \\
&= \sum_{x \in X} \left(- \left(P(x) * \log_2 \left(P(x) \right) \right) + \left(P(x) * 2 \right) \right) \\
&= - \sum_{x \in X} \left(P(x) * \log_2 \left(P(x) \right) \right) + \sum_{x \in X} \left(P(x) * 2 \right) \\
&= - \sum_{x \in X} \left(P(x) * \log_2 \left(P(x) \right) \right) + 2 * \sum_{x \in S} \left(P(x) \right) \qquad \left(\text{mit} \sum_{x \in S} \left(P(x) \right) = 1 \right) \\
&= H(X^m) + 2
\end{aligned}$$

Da nach dem Hauptsatz der Quellenkodierung die komprimierten Daten mindestens eine Länge haben die der Entropie entspricht gilt:

$$H(X^m) \le l(X^m) \le H(X^m) + 2$$

Damit folgt für die im Durchschnitt für ein Symbol verwendete Bitlänge l_X:

$$\begin{aligned}
&\frac{1}{m} H(X^m) \le \frac{1}{m} l(X^m) \le \frac{1}{m} \left(H(X^m) + 2 \right) \\
&\Rightarrow \frac{H(X^m)}{m} \le l_x \le \frac{H(X^m)}{m} + \frac{2}{m} \\
&\Rightarrow H(X) \le l_x \le H(X) + \frac{2}{m}
\end{aligned}$$

Es ist sichergestellt, dass bei steigender Länge der Eingabe die Länge der Ausgabe sich immer mehr der Entropie nähert.

Wörterbuch-Techniken

Während die vorherigen Komprimierungstechniken ein Wahrscheinlichkeitsmodell zur Datenkompression heranziehen, sodass der Kompressionsgrad unmittelbar von der Güte des verwendeten statistischen Modells abhängt, machen wörterbuchbasierte Verfahren von der Redundanz der Eingabedaten Gebrauch, sie erzielen Kompression durch Substitution. Trotz dieses Unterschiedes erreichen sie eine asymptotische Annäherung an die Entropie. Daher werden sie auch als **Universalkomprimierungen** bezeichnet.
Die Wörterbuchtechniken fassen häufiger auftretende Zeichengruppen zusammen und vergeben an sie kürzere Codes.

Arten von Wörterbuchtechniken

Statische Verfahren

Es wird mit einem festen Wörterbuch gearbeitet, das neben den Grundzeichen auch häufige Zeichenpaare enthält. Dieses Verfahren ist sehr einfach und nur wenig effizient. Auch die Verwendung längerer Zeichenketten ist wenig effizient, da diese meist nur für eine kleine Gruppe von Eingaben häufiger vorkommen und somit die Menge der gut komprimierbaren Eingaben erheblich einschränken.

Semiadaptive Verfahren

Sie versuchen die Problematik eines ungeeigneten Wörterbuches zu umgehen, indem zu jeder gegebenen Eingabe ein neues Wörterbuch mit möglichst passenden Phrasen generiert wird. Der Ansatz ist recht vielversprechend, da so meist längere, für den Text spezifische Phrasen, in das Wörterbuch aufgenommen werden können, und eine bessere Kompression zu erwarten ist. Die zentrale Komponente von semiadaptiven Verfahren ist der *Parse*r, der die Eingabedaten in einem separaten Schritt analysiert. Mit Hilfe unterschiedlicher Strategien erzeugt dieser ein Wörterbuch mit einer meist beschränkten Anzahl von Phrasen, das bei der anschließenden Komprimierung verwendet wird.
Allerdings ist die Entscheidung, welche Phrasen das generierte Wörterbuch enthalten soll, um eine maximale Kompression zu erzielen, überraschend schwierig zu treffen. Ein zusätzlicher Overhead entsteht durch das Wörterbuch selbst, das nun zusammen mit den komprimierten Daten abgelegt werden muss.

Adaptive Verfahren

Diese Verfahren beginnen mit einem leeren Wörterbuch oder mit einem Wörterbuch, das nur die Grundzeichen enthält, die ja mit hoher Wahrscheinlichkeit in der Eingabe auftreten. Während des Komprimierens werden dem Wörterbuch nach einem bestimmten Schema neu auftretende Zeichenketten hinzugefügt, die dann an späterer Stelle wieder verwendet werden können. Einige Verfahren erlauben auch einzelne, wenig genutzte Zeichenketten wieder aus dem Wörterbuch zu entfernen.
Diese Art der Wörterbuch-Erzeugung ist ebenso einfach wie effizient. Die Wahrscheinlichkeit für das Auftreten eines Wortes steigt mit seiner erstmaligen Verwendung stark an. Zudem ist es bei geeigneten Algorithmen möglich das Wörterbuch nicht extra zu speichern sondern bei der Dekomprimierung selbst neu zu erzeugen.

Die bekanntesten dieser Verfahren gehören fast alle zur Gruppe der Lempel-Ziv-Kodierungen.

Die LZ-Algorithmen

Die Familie der LZ-Algorithmen teilt sich in 2 Gruppen, die Nachfolger des LZ77 und die Nachfolger des LZ78. Die beiden Basisalgorithmen wurden von Abraham Lempel und Jacob Ziv entwickelt, wobei 77 bzw. 78 das Veröffentlichungsjahr ist. Trotz ihrer ähnlichen Namen benutzen beide Algorithmen sehr verschiedene Ansätze zur Kodierung. Die meisten Variationen der LZ-Algorithmen kann man an dem LZ in ihrem Namen erkennen, wie zum Beispiel LZW oder LZH.

LZ77

Dieser Algorithmus betrachtet immer nur eine feste Zahl der folgenden Zeichen. Dieser Bereich der Eingabe wird als **Look-Ahead-Buffer** (LA) bezeichnet. Außerdem betrachtet er eine feste Zahl der zuletzt codierten Zeichen, dieser Bereich wird als **Search-Buffer** (SB) bezeichnet. Um die Zeichen im LA zu codieren wird der SB rückwärts nach der längsten Übereinstimmung durchsucht. Als Kodierung (Token) wird dann die Entfernung zum Anfang der Übereinstimmung (Offset), die Länge der Übereinstimmung und das folgende Zeichen im LA (Abschlusszeichen) gespeichert. Der jeweils betrachtete Teilbereich der Eingabe, also SB+LA, wird auch als **Sliding Window** bezeichnet.

Beispiel: Komprimierung von cabbdcbaacbdddabda mit LZ77

		Sliding Window			
Pos	bereits codiert	bereits codiert (SB)	LA	noch nicht verarbeitet	Token
				cabbdcbaacbdddabda	
1			cab	bdcbaacbdddabda	(0,0,c)
2		c	abb	dcbaacbdddabda	(0,0,a)
3		ca	bbd	cbaacbdddabda	(0,0,b)
4		cab	bdc	baacbdddabda	(1,1,d)
6		cabbd	cba	acbdddabda	(5,1,b)
8		cabbdcb	aac	bdddabda	(6,1,a)
10	ca	bbdcbaa	cbd	ddabda	(4,2,d)
13	cabbd	cbaacbd	dda	bda	(1,2,a)
16	cabbdcba	acbddda	bda		(5,2,a)
19	cabbdcbaacb	dddabda			

Interessant ist, dass der LA auch als SB betrachtet wird, wie im Beispiel an Position 13. Beim Original LZ77 ist die Größe des SB fest, daher verwendet man auch eine Konstante Bit-Länge für die Token.
Die Dekomprimierung ist verhältnismäßig einfach. Die Token haben konstante Länge und können daher problemlos ausgelesen werden. Offset und Länge beziehen sich immer auf bereits dekodierte Werte und das Abschlusszeichen ist nicht besonders codiert. Beachtet werden muss lediglich die oben erwähnte Überschneidung von SB und LA.

Variationsmöglichkeiten für LZ77:

- unbeschränkter Search-Buffer (**LZR**): variable Bitlänge der Tokens, keine höhere Kompressionsrate
- werden für die Zeichen im LA zu kurze Übereinstimmungen gefunden, werden sie als Klartext gespeichert (**LZSS**): vermeidet Aufblähen der Daten, benötigt zusätzliches Flag-Bit ob Daten oder Token folgt
- alle Teilstrings mit Länge von LA die sich z.Z. in SB befinden als Binärbaum zwischenspeichern (**LZSS**): beschleunigt Suche, beim Verschieben des Sliding Window muss der Baum korrekt aktualisiert werden
- Bit-Länge des Offset-Wertes im Token dynamisch verkürzen für den Fall, dass erste ein Teil des SB belegt ist (**LZB**): Beispiel: SB= 2^{16} Bit, bisher gelesene Zeichen=4 => Offset-Länge=2-Bit
- variable Bit-Länge für den Länge-Wert im Token (**LZB**)
- die Token-Werte zusätzlich mit Huffman-Code komprimieren (**LZH**): längere Laufzeit, keine höhere Kompressionsrate
- u.a.

Verwendung von LZ77-Varianten:

- gzip: Variante von LZ77, die mit der Huffman-Kodierung kombiniert wurde
- png: ähnelt gzip allerdings auf Bilder spezialisiert.

LZ78

Im Gegensatz zu LZ77 sucht LZ78 nicht im bereits kodierten Bereich nach einer Übereinstimmung. Es wird ein separates Wörterbuch benutzt, das zu Beginn keine Zeichenketten enthält.
Um eine Eingabe zu komprimieren, wird zunächst die längste Übereinstimmung mit einem Eintrag im Wörterbuch gesucht. Als Kodierung (Token) wird die Indexnummer der längsten Übereinstimmung und das Folgezeichen verwendet. Außerdem wird die Übereinstimmung mit Folgezeichen als neuer Eintrag in das Wörterbuch aufgenommen.

Beispiel: Komprimierung von „abacbabaccbabbaca“ mit LZ78

Schritt	Eingabe	Token	Neuer Wörterbucheintrag /Index
1	a	0,a	a,1
2	b	0,b	b,2
3	ac	1,c	ac,3
4	ba	2,a	ba,4
5	bac	4,c	bac,5
6	c	0,c	c,6
7	bab	4,b	bab,7
8	baca	5,a	baca,8

Die Dekomprimierung funktioniert äquivalent. Auch hier wird das Wörterbuch bei jedem Token um einen Eintrag erweitert, der sich aus dem über den Index gefundenen Wörterbucheintrag und dem explizit gespeicherten Folgezeichen ergibt.
Komprimierung und Dekomprimierung gemeinsam ist der Bedarf nach einem möglichst einfach zu verwaltenden Wörterbuch. In der Regel wird hier ein Baum verwendet, bei dem jeder Knoten genau so viele Kinder haben kann wie Eingabezeichen erlaubt sind.

Die Größe des Wörterbuches ist beim Original-Algorithmus nicht beschränkt. Daher müssen die Index-Werte in den Token mit variabler Bitlänge gespeichert werden. Eine explizite Angabe wie lang der im aktuellen Token verwendete Index ist, wird jedoch nicht gespeichert. Vielmehr wird die Bit-Länge des Index von der Größe des Wörterbuches bestimmt, der Index ist jeweils so lang, dass er auch den größten Wörterbuch-Index aufnehmen könnte. Beispiel: Enthält das Wörterbuch 24 Einträge so wird der Index mit $\lceil \log_2 24 \rceil = 5Bit$ gespeichert.

Variationsmöglichkeiten für LZ78:

- die Größe des Wörterbuches begrenzen: Ist das Wörterbuch voll, wird es gelöscht und von dieser Stelle der Eingabe an neu aufgebaut.
- die Größe des Wörterbuches begrenzen: Ist das Wörterbuch voll, wird es statisch weiterverwendet (ohne neue Einträge hinzuzufügen).
- eine Obergrenze für die Größe des Wörterbuches verwenden: Ist das Wörterbuch voll, dieses statisch weiter verwenden solange die Komprimierungsrate nicht spürbar absinkt, sonst neues Wörterbuch erstellen (**LZC**).
- das Folgezeichen nicht explizit sondern implizit als erstes Zeichen des Folge-Tokens speichern (**LZW**, patentiert): kompakterer Code, Wörterbuch enthält zu Beginn alle Eingabezeichen, bei der Dekomprimierung muss erst der Folge-Token gelesen werden um den Wörterbucheintrag zu erzeugen
- basierend auf LZW eine variable Bit-Länge für die Token verwenden (**LZC**).

Verwendung von LZ78-Varianten:

- compress unter Unix: benutzt LZC, benötigt daher Lizenz
- gif: LZC- ähnliche LZW Variante, benötigt daher Lizenz

Codes für Integerzahlen

Um Integerzahlen beliebiger aber endlicher Länge speichern zu können, wurden verschiedene Methoden entwickelt. Genau wie die arithmetische Codierung, sind aber auch hier viele Verfahren an ein bestimmtes Modell gebunden, das die Menge der optimalen Situationen einschränkt. Andere Verfahren sind universell und damit zwar nicht optimal für konkrete Situationen aber dafür immer anwendbar.
Man kann im wesentlichen folgende Vorgehensweisen unterscheiden:

1) Man überträgt die ganze Zahl im Unären-Code. Beispielsweise würde eine 10 als zehn Nullen gefolgt von einer 1 dargestellt.
2) Man übermittelt erst die Anzahl der Datenbits und danach die Daten selbst (self delimiting codes).
3) Man überträgt jeweils Blöcke von Datenbits, gefolgt von einem Flag-Bit das signalisiert ob dies der letzte Datenblock war oder nicht.

Self delimiting Integer codes

Es ist leicht ersichtlich, dass das erste Verfahren nicht besonders platzsparend ist. Möchte man andererseits das zweite Verfahren benutzen, so besteht die Notwendigkeit die Anzahl der Bits zu speichern, und da diese Anzahl beliebig groß sein kann, handelt es sich hierbei ebenfalls um einen Integer unbekannter Länge. Eine mögliche Lösung ist, die beiden ersten Verfahren zu kombinieren. Man überträgt zunächst die Anzahl der Datenbits im unären Code und überträgt dann die Daten selber. So ist zum Beispiel

```
00000110010
=000001   10010
=5        18
```

eine eindeutige Darstellung für die 18. Dieses Verfahren bedeutet, das die Darstellung der Daten etwa die doppelte Größe der Daten benötigt. Natürlich kann man dieses Verfahren auch rekursiv wiederholen und so zum Beispiel das oben angegebene Beispiel als die Repräsentation für Länge eines Integers mit 18 Bits zu verwenden.

Elias universal code

Beim Elias Code wird die Zahl in Blöcken gespeichert, er soll als Beispiel für die dritte Methode zum speichern von Integern unbekannter Größe dienen. Ist das niedrigste Bit eines Blockes 0 so handelt es sich um das Ende der Zahl. Ist es 1, so ergibt sich je nach Wert der höheren Bits dieses Blockes die Anzahl der Bits, die zum nächsten Block gehören. Natürlich gibt es für jeden Länge-Block einen Basiswert, ab dem der letzte Block aufwärts zählt. Dadurch wird nicht nur der Wert selber, sondern auch die Länge des Codes gespeichert.
Wie man sich leicht überzeugen kann, ist das Speichern von Codes auf diese Art relativ platzraubend.

beginnt mit	Codelänge	n	code(n)
0	1	1	0
1	3	2	100
		3	110
1,01	3+3=6	4	101000
		5	101010
		6	101100
		7	101110
1,11	3+4=7	8	1110000
		9	1110010
		10	1110100
		11	1110110
		12	1111000
		13	1111010
		14	1111100
		15	1111110
1,01,001	3+3+5=11	16	10100100000
		31	10100111110

beginnt mit	Codelänge	n	code(n)
1,01,011	3+3+6=12	32	101011000000
		45	101011011010
		63	101011111110
1,01,101	3+3+7=13	64	1011010000000
		127	1011011111110
1,01,111	3+3+8=14	128	10111100000000
		255	10111111111110
1,11,0001	3+4+9=16	256	1110001000000000
		365	1110001011011010
		511	1110001111111110
1,11,0011	3+4+10=17	512	11100110000000000
		719	11100110110011110
		1023	11100111111111110
1,11,0101	3+4+11=18	1024	111010100000000000
		1025	111010100000000010

Literatur

[McKay] David MacKay: Information Theory, Inference and Learning Algorithms
http://www.inference.phy.cam.ac.uk/mackay/Book.html

[ABD89] M. Atkinson, F. Bancilhon, D. De Witt, K. Dittrich, D. Maier, and S. Zdonik: The Object-Oriented Database Manifesto; in: Deducitve and Object-Oriented Databases; Proc. of the First International Conference on Deductive and Object-Oriented Databases (DOOD '89), Kyoto research Park, Kyoto, Japan, 4.-6. Dec. 1989; North-Holland/Elsevier Science Publishers; pp.223-240.

[DS00] D. Salomon. *Data compression – The complete reference*. Springer Verlag, 2. Auflage, 2000. ISBN 0-387-95045-1.

[KS00] K. Sayood. *Introduction to data compressio*n. Morgan Kaufmann, 2. Auflage, 2000. ISBN 1-55860-558-4.

[ccFAQ] comp.compression Frequently Asked Questions: http://www.faqs.org/faqs/compression-faq/.

Die folgende Literatur wurde nicht oder nur indirekt für dieses Kapitel verwendet, sie soll nur als Startpunkt für weitere Informationssuche angesehen werden.

allgemein: data-compression.com: http://www.data-compression.com/index.html

allgemein: C. E. Shannon, A Mathematical Theory of Communication (free pdf version) : http://cm.bell-labs.com/cm/ms/what/shannonday/shannon1948.pdf

allgemein: datacompression.info: http://www.datacompression.info/

allgemein: Lossless Image Compression : http://www.cs.technion.ac.il/Labs/Isl/Project/Projects_done/VisionClasses/DIP/Lossless_Compression/lossless.html

LZ78: Animation of Lempel-Ziv Encoding Algorithm: http://www.data-compression.com/lempelziv.html

LZ77,Huffman: RFC1951-DEFLATE Compressed Data Format Specification version 1.3: ftp://ftp.uu.net/graphics/png/documents/zlib/zdoc-index.html

LZ77: Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.

LZ78: J. Ziv, A. Lempel, Compression of Individual Sequences via Variable-Rate Coding, IEEE Trans. Inform. Theory, 1978, vol. 24, no. 5, pp. 530-536.

LZW: T.A. Welsh, A Technique for High-Performance Data Compression, Computer, 1984, vol. 17, no. 6, pp. 8-19

LZW-Patent: Nummer des US-Patentes: US4558302
Homepage von Unisys (Patentbesitzer für Gif)
http://www.unisys.com/about__unisys/lzw/
allgemeine US-Patentsuche: http://optics.org/research/patents.cfm
allgemeine US-Patentsuche: http://patft.uspto.gov/netahtml/srchnum.htm
Text-Kopien des LZW-Patentes:
ftp://ftp.std.com/obi/USPatents/lzw-patent.Z
ftp://ftp.uu.net/doc/literary/obi/USPatents/lzw-patent.Z
http://www.geocities.co.jp/SiliconValley/3453/gif_info/doc/LZWFIGs.lzh
Link zu USPTO

Aritmetic Coding: I.H. Witten, R.M. Neal, J.G. Cleary, Arithmetic Coding for Data Compression, Commun. ACM, 1987, no. 30, vol. 6, pp. 520-540.

Huffman: The Lossless Compression (Squeeze) Page: http://www.cs.sfu.ca/cs/CC/365/li/squeeze/

Huffman: D.A. Huffman, A Method for the Construction of Minimum-redundancy Codes, Proc. IRE, 1952, vol. 40, no. 10, pp. 1098-1101.

Analogtechnik und Digitalisierung von Audio und Video

von

Jirk Stolze

Einleitung

Wir nehmen unsere Umwelt durch unsere Sinne auf. Zwei der wichtigsten sind das Sehen und das Hören. Physikalische Größen, wie Lichtintensität oder Schall, wirken auf Rezeptoren in unseren Augen oder Ohren und wandeln diese Signale um, so dass unser Gehirn diese interpretieren kann. Dadurch können wir Töne, Geräusche, Musik und Bilder wahrnehmen.

Ein Ziel des Menschen ist es, seine Umwelt möglichst genau nachbilden zu können. Mit Hilfe der Analogtechnik ist es ihm gelungen, diese Signale festzuhalten, weiterzuleiten und auch zu speichern. Doch auch wenn sich die Analogtechnik weiter entwickelt hat, die Nachteile sind so gravierend, dass nach neuen Methoden gesucht wurde. Das Ergebnis ist die Digitaltechnik.

Die Umwandlung von natürlichen Größen bis hin zur digitalen Speicherung möchte ich hier nun näher erläutern. Ebenfalls sollen die Vor- und Nachteile dargestellt werden, die die Analogtechnik von der Digitaltechnik unterscheiden.

Der erste Teil wird sich mit der Digitalisierung von Audio-Signalen befassen, der zweite Teil geht dann auf die Digitalisierung von Bild- und Video-Signalen ein.

Von Vibrationen zur Audio-CD

Der Begriff *Audio* stammt von dem lateinischen Wort audire (hören) und dient als Sammelbegriff für akustisch wahrnehmbare Signale.

Entstehung und Wahrnehmung von Tönen

Es existieren viel mehr Töne, als wir wirklich wahrnehmen können.

Schallwellen werden von einer oder mehreren Tonquellen erzeugt. Tonquellen sind zum Beispiel alle vibrierenden Objekte. Beim Schlagen einer Trommel vibriert die Oberfläche, beim Reiben an einem Glas mit feuchten Fingern, beginnt das Glas zu vibrieren. Die menschliche Stimme ist eine komplexe Kombination aus verschiedenen Schwingungen: der Stimmbänder, der Luft in der Lunge, im Mund und Rachenraum sowie der Schwingungen in den Körperflüssigkeiten. Bei jeder Vibration wird die Luft stoßweise verdichtet, es entstehen Schallwellen. Diese brauchen, um sich fortzubewegen, ein Medium. Das kann Luft, Holz, Wasser oder ein anderes komprimierbares Medium sein.
Die Ausbreitungs- bzw. Schallgeschwindigkeit in der Luft beträgt 330m/s (1188km/h).

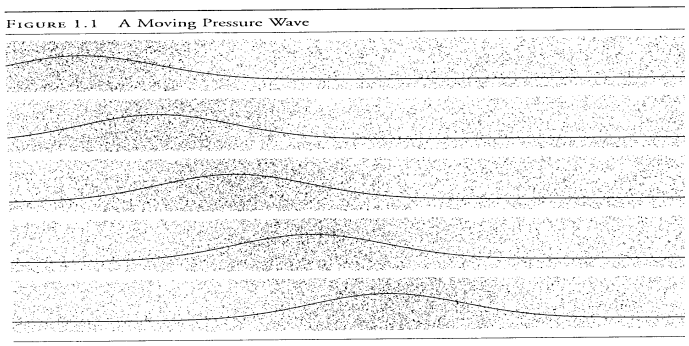


FIGURE 1.1 A Moving Pressure Wave

Abbildung 1: Eine sich fortsetzende Schallwelle [TM0]

Der Transport von Schallwellen ist im Prinzip bei jedem Medium gleich. Die Moleküle eines Mediums sind normaler Weise gleichweit voneinander entfernt. Nachdem sie durch Vibration verdichtet wurden, versuchen sie ihren alten Abstand wiederherzustellen. Dabei bewegen sie sich und verdichten das Gebiet ihrer Nachbarn. Die Abbildung 1 zeigt solch eine Schallwelle. Die dunkleren Bereiche sind die Verdichtungsbereiche. Jedes Molekül bewegt sich also nur ein kleines Stück vor und zurück, aber betrachtet man den gesamten Bereich entsteht eine fortlaufende Schallwelle.
Treffen diese Wellen dann auf unser Ohr, werden sie in Abhängigkeit von Stärke und Häufigkeit durch das Trommelfell in Reize umgewandelt, welche wir dann als Töne wahrnehmen.

Grundlagen zu Schwingungen

Die einfachste Schwingung stellt die Form einer Sinuskurve dar. Sie ist eine periodische Schwingung. Zu den periodischen Schwingungen gehören alle Töne, die man als Klänge bezeichnen kann (z.B. Klavier oder Glocke). Die nichtperiodischen Schwingungen erzeugen Geräusche (z.B. Wasserrauschen). Es gibt nur sehr wenige Töne, die rein periodisch sind. Jedoch kann man jeden Ton in seine Grundschwingungen zerlegen und aus verschiedenen Grundschwingungen kann ein neuer Ton erzeugt werden.

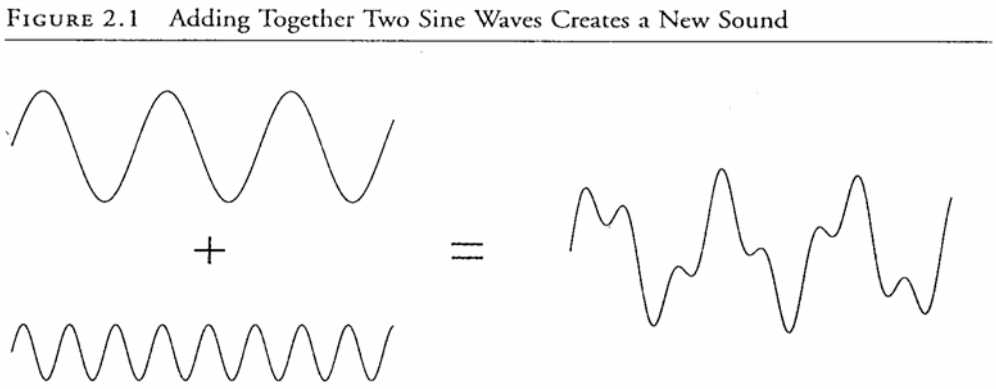


Abbildung 2: Komposition aus Sinusschwingungen [TM0]

Eigenschaften von Schallwellen

Durch bestimmte Eigenschaften von Schallwellen nehmen wir Töne unterschiedlich wahr. Die Lautstärke und die Höhe eines Tones werden durch diese bestimmt.

Lautstärke und Amplitude

Wie nehmen Töne in verschiedenen Lautstärken wahr. Verantwortlich dafür sind die Druckwellen, die unterschiedlich stark auf unser Ohr treffen. Die Stärke der Druckwelle ist

erkennbar an der Amplitude ihrer Schwingungskurve. Damit bestimmt die Amplitude einer Schwingung die Lautstärke eines Tones.
Als Maß für die Lautstärke wird die Schallintensität als Leistung / Fläche definiert, die Einheit ist Watt / m^2. Als Schallpegel bezeichnet man den 10fachen dekadischen Logarithmus vom Verhältnis zweier Schallintensitäten. Er ist daher dimensionslos. Als Bezeichnung verwendet man das *Dezibel* (dB, nach Alexander Bel). 0 dB ist der leiseste Ton, den ein Durchschnittsgehör noch wahrnehmen kann (10^{-12} Watt pro m^2). Technisch ist er definiert als ein 1000 Hz Ton – der einen Luftdruck von 20 µP (Mikropascal) erzeugt (gilt nur für Messungen in der Luft). Ein trainiertes Ohr kann eine Zunahme von 1 dB wahrnehmen. Den lautesten Ton den ein Mensch noch hören kann liegt bei ungefähr 120 dB. Das ist eine Billionen Mal stärker als der Referenz-Ton. Im Anhang sind verschiedene Töne aufgelistet.

Tonhöhe und Frequenz

Da jeder Ton in Sinusschwingungen zerlegt werden kann, existiert eine Möglichkeit ein Spektrum anzugeben, der einen Ton charakterisiert.

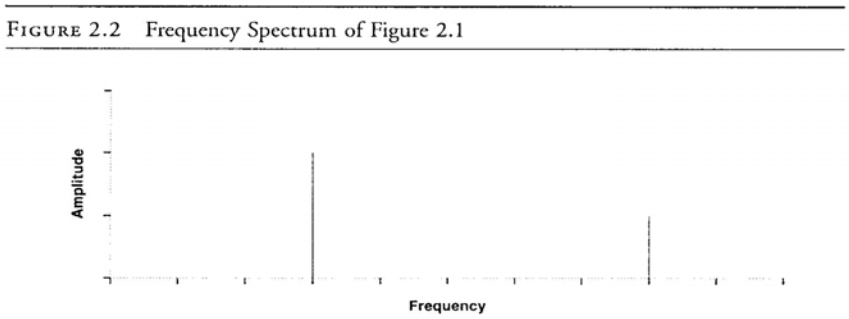


Abbildung 3: Frequenzspektrum von Abbildung 2 [TM0]

In Abbildung 3 erkennt man das Frequenzspektrum der erzeugten Schwingung aus Abbildung 2. Es ist zu sehen, dass die zweite Schwingung mit unterschiedlicher Frequenz diese Schwingung erzeugen. Es ist auch zu erkennen, dass die erste Schwingung eine größere Amplitude besitzt und daher auch lauter wahrgenommen wird als die zweite. Da die stärkste Frequenz im Allgemeinen die Tonhöhe bestimmt, empfindet man diese als reale Tonhöhe, was aber nicht wirklich stimmt.
Bei den meisten Instrumenten treten neben der Hauptfrequenz („Fundamental“) weitere Nebenfrequenzen auf („Harmonics“), welche die Tonhöhe beeinflussen.

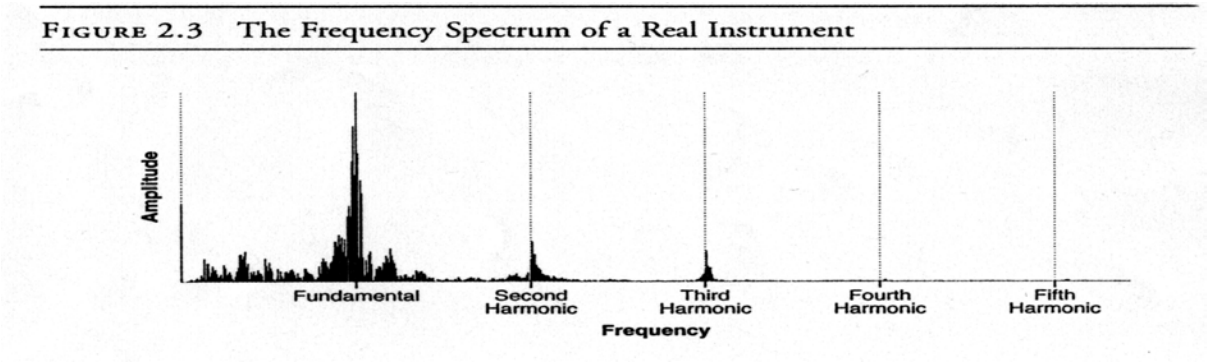


Abbildung 4: Frequenzspektrum eines Instrumentes [TM0]

Es ist sinnvoll, die Frequenz eines komplexen Tones, so zu definieren, dass sie der Frequenz der Sinusschwingung entspricht, die der empfundenen Tonhöhe gleicht.

Von Schallwellen zu analogen Signalen

Nachdem nun die Grundlagen von Schallwellen klar sind, stellt sich nun die Frage, wie man diese Wellen in ein leichter zu verarbeitendes Format bringen kann – zum Beispiel Elektrizität. Die Umwandlung von Schallwellen in elektrische Impulse ist die Basis von Radio, analoger Telekommunikation oder Kassettenrecordern. Doch wie funktioniert sie?

Aus einem Stück Papier mit einer Spule und einem Magnet wird ein Druckmesser gebaut.
Beim Auftreffen einer Schall(Druck-)welle auf das Papier wird die Spule je nach Stärke der Druckwelle in Richtung des Magneten gedrückt. Dabei wird ein Impuls in der Spule induziert. Je stärker die Druckwelle, desto stärker der Impuls. Dadurch werden die Schallwellen in elektrische Signale umgewandelt.

Umgekehrt funktioniert es ebenfalls. Durch Induktion von elektrischen Impulsen in die Spule wird das Blatt Papier vom Magneten angezogen oder abgestoßen. Es vibriert, dass heißt es erzeugt Schallwellen – also Töne.

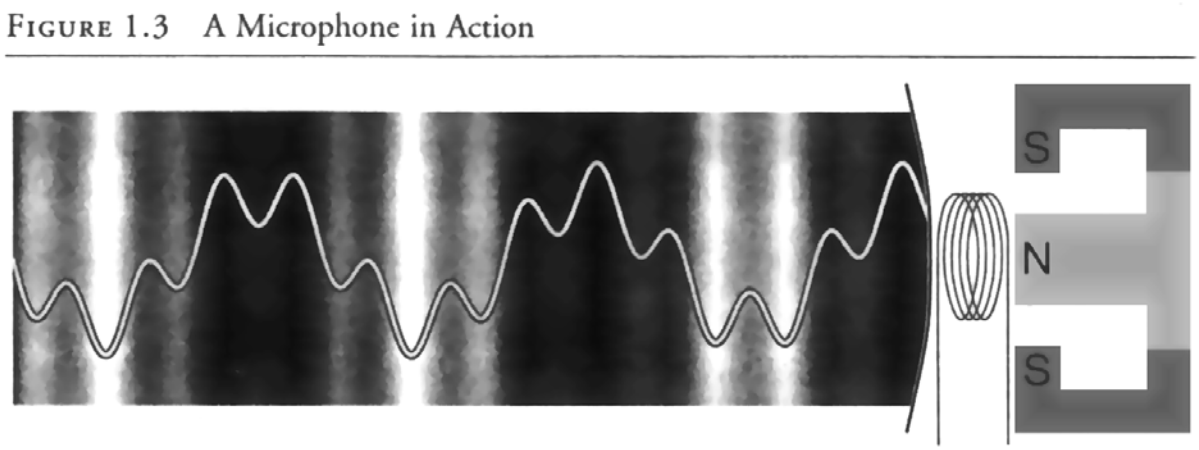


Abbildung 5: Funktionsweise eines Mikrophons [TM0]

Mikrophone und Lautsprecher funktionieren nach diesem Prinzip.

Digitalisierung von analogen Signalen

Der nächste Schritt ist nun die Konvertierung der analogen kontinuierlichen Signale in diskrete digitale Werte. Um dies zu erreichen sind zwei Prozesse erforderlich. Der erste ist das Abtasten des analogen Signals, welcher Messwerte, die so genannten „Samples“ liefert, welche dann im zweiten Schritt – der Quantisierung - gerundet werden, damit sie einem bestimmten Zahlenformat entsprechen.

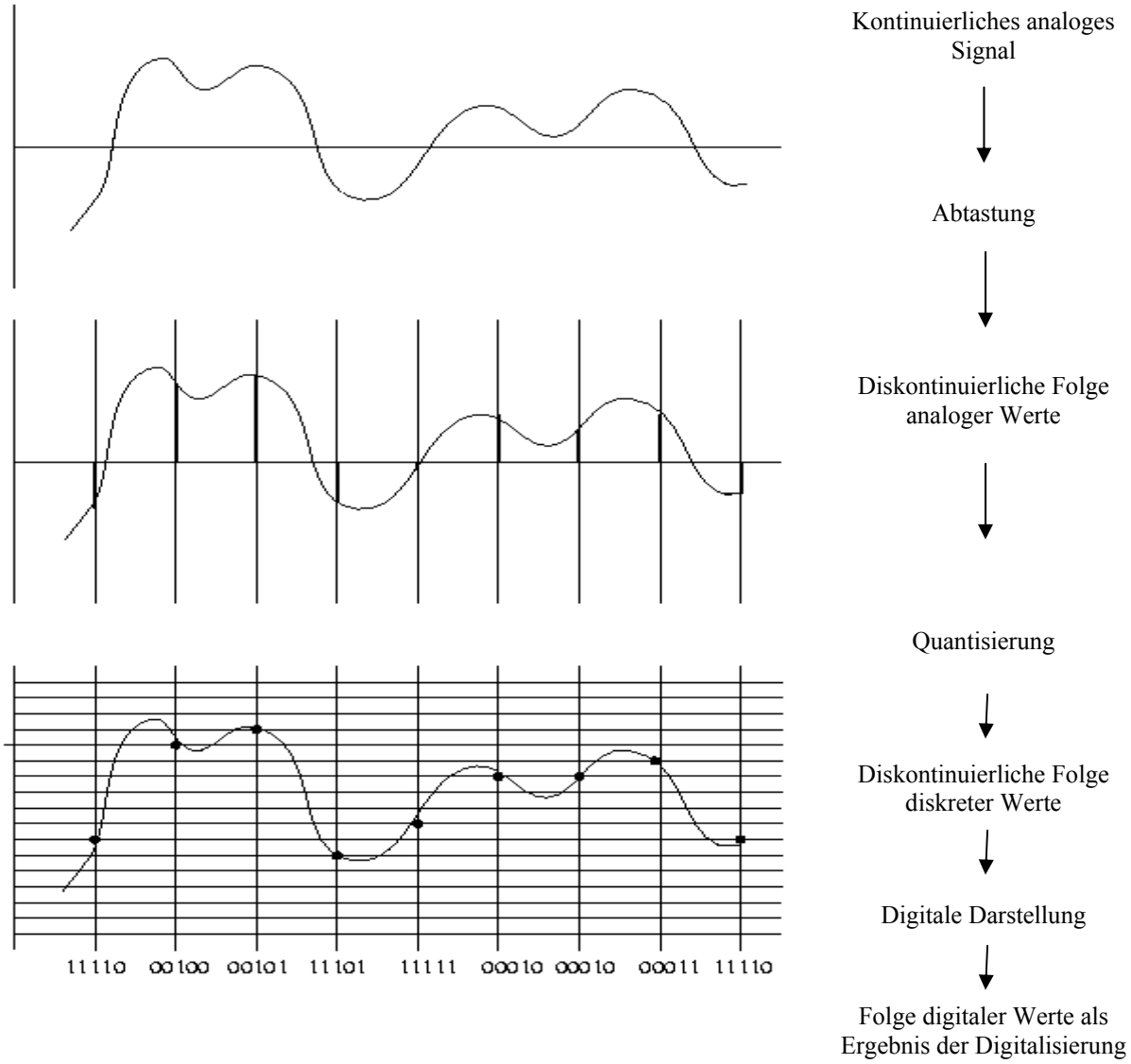


Abbildung 6: Der Gesamt-Digitalisierungsprozess [INT4]

Sampling

Wie oben schon erwähnt, steht der Begriff „Sampling“ oder „Samplen“ nicht ganz wie umgangssprachlich für das Aufnehmen eines Musikstückes, sondern nur für einen Teil dieses Vorgangs – des Messens(Abtastens) eines kontinuierlich analogen Signals in bestimmten Zeitabständen. Die Festlegung der Zeitabstände ist die so genannte Sampling-Rate und wird gebräuchlicher Weise in kHz angegeben. Sie bestimmt die Anzahl der Messungen des analogen Signals / pro Sekunde. In Abbildung 6 werden diese Messpunkte als senkrechte Striche dargestellt. Durch das Abtasten des Signals in gewissen Zeitabständen erhält man eine diskontinuierliche Folge von Werten, die aber noch unbegrenzt genau sind und so nicht in ein digitales Format passen. Sie dienen als Ausgangspunkt für die Quantisierung.
Bei gängigen Musik-CDs wird heute eine Sampling-Rate von 44,1 kHz angewendet, dass heißt, dass analoge Signal wird 44100mal pro Sekunde abgetastet. Bei digitalen Telefonsystemen reichen allerdings schon 8 kHz aus.

Sampling-Methoden

Technisch gibt es mehrere Möglichkeiten ein analoges Signal abzutasten. Drei sollen hier kurz erläutert werden.

PAM (Puls-Amplitude-Modulation)

Die analogen Signale werden durch eine Serie von Impulsen, deren Amplitude die Soundstärke repräsentieren, übertragen.

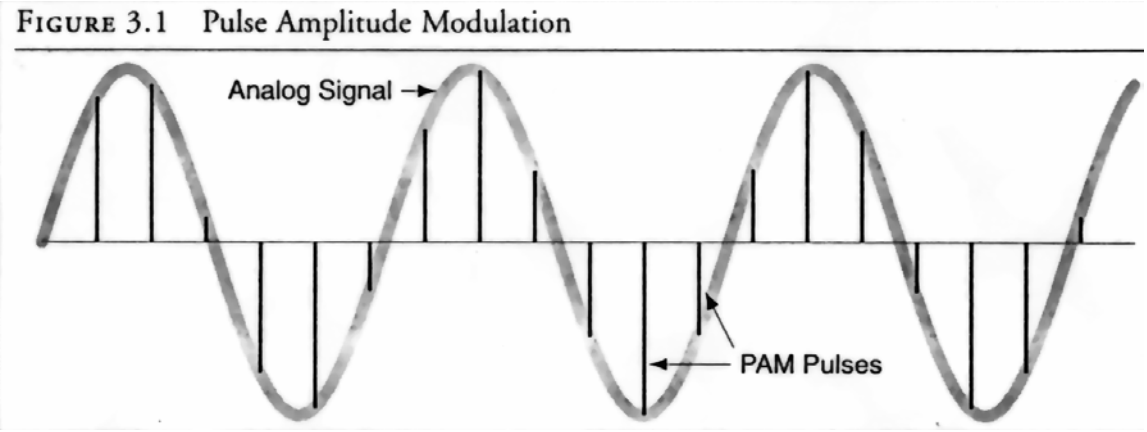


Abbildung 7: PAM [TM0]

Vorteil: Es ist einfach ein analoges Signal in ein PAM - Signal umzuwandeln und umgekehrt.
Folge: Die meisten ADCs und DACs benutzen PAM als ein Zwischenformat.

PWM (Puls-Width-Modulation)

Die analogen Signale werden durch eine Serie von Impulsen, deren Länge die Soundstärke repräsentieren, übertragen.

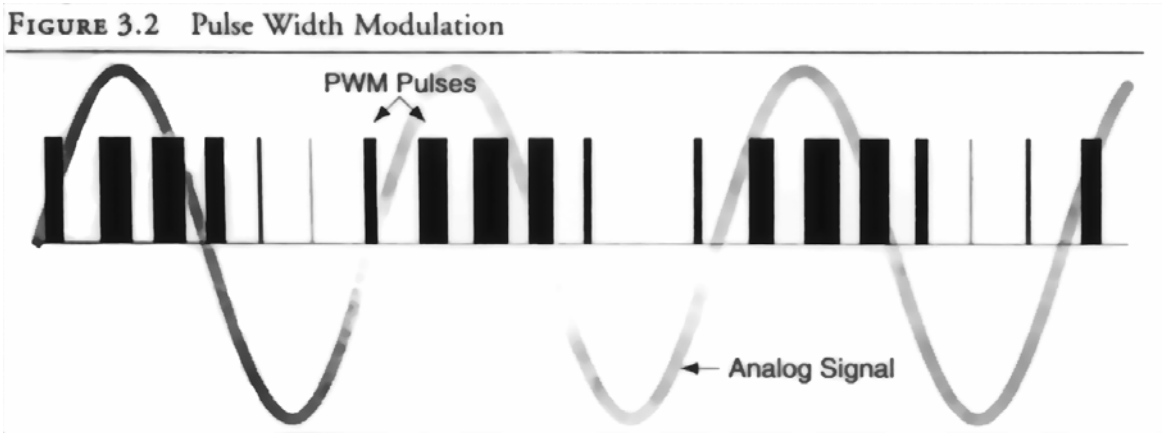


Abbildung 8: PWM [TM0]

Vorteil: In der Praxis werden die Amplituden der Signale oft zerstört. Durch Umwandlung eines analogen Signals vor dem Transport durch Kabel oder Radioverbindungen in PWM, wird die Schwächung des Signals vermindert.

PCM (Puls-Code-Modulation)

Die analogen Signale werden durch eine Serie von Impulsen, die den binären Daten des Samples entsprechen. übertragen. Ein Problem ist es, zu unterscheiden, wann ein Binärcode beginnt und endet. Eine Lösungsmöglichkeit liegt darin, ein zweites Signal wird mit zusenden, dass den Start- und Endpunkt eines Codes identifiziert. Eine weitere Möglichkeit besteht in der Verwendung eindeutiger Codes (Huffman-Kodierung o.ä.).

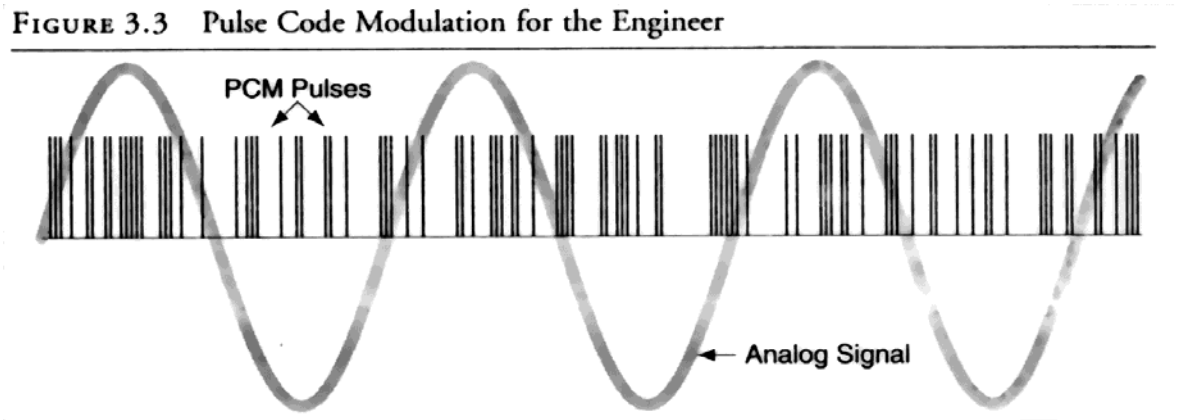


Abbildung 9: PCM [TM0]

Sampling - Mathematics

Eine wichtige Entdeckung zum Thema „Sampling“ hat der Wissenschaftler Harry Nyquist (1889 – 1976) gemacht. Er hat herausgefunden, dass ein analoges Signal exakt reproduziert werden kann, wenn die Abtastrate doppelt so hoch ist, wie die höchste Frequenz des analogen Signals. Damit hat er das Nyquist-Limit definiert, welches die Grenze der Frequenzen definiert, die noch verlustfrei aus einem analogen zeitlich lückenlosem in ein zeitdiskretes Signal überführt werden können. Die, auch als Abtasttheorem, bekannte Theorie wurde von dem Mathematiker Claude E. Shannon 1948/49 mathematisch bewiesen.
Dazu ein Beispiel. Möchte man ein Musikstück, das als höchste Frequenz 22kHz enthält, aufnehmen, sollte man mindestens eine Sampling-Rate von 44kHz benutzen. Anders ausgedrückt, benutzt man eine Sampling-Rate von 8kHz, so liegt das Nyquist-Limit bei 4kHz. Solange keine Frequenz im Originalsignal höher als 4kHz ist, findet die Überführung in zeitlich diskrete Werte verlustfrei statt.

Quantisierung

Quantisierung bezeichnet das Runden der ermittelten unbegrenzt genauen analogen Werte auf einen digitalen Wertebereich (z.B. 8bit Integer, 16bit Integer). Dies ist erforderlich, da digitale Werte diskret sind, dass heißt, sie liegen nicht dicht. In Abbildung 6 sind diese digitalen Werte als waagerechte Linien abgebildet. Das hat zur Folge, dass die gemessenen Werte an den Messpunkten auch zwischen zwei digitalen Werten liegen können. Hier wird dann gerundet. Man erhält so eine diskontinuierliche Folge diskreter Werte, die nun in die

Binärdarstellung umgewandelt werden. In Abbildung 6 wird das als Zweierkomplement dargestellt.

Probleme

Der Digitalisierungsprozess bringt Probleme mit sich. Zwei Hauptprobleme sind:

1) Da das analoge Signal nur in bestimmten Zeitabständen abgetastet wird, gehen Informationen verloren. Je geringer die Abtastrate ist, desto größer ist das Intervall welches nur durch einen digitalen Wert repräsentiert wird und umso größer ist der Informationsverlust. Nach Nyquist und Shannon kann man diesen Fehler vernachlässigen, solange die Voraussetzungen erfüllt werden. Dies führt zum nächsten Problem: Je höher die Sampling-Rate, desto höher ist auch der Platzbedarf, um diese Messdaten zu speichern.

2) Bei der Quantisierung werden die Originalinformationen durch das Runden der Messwerte verfälscht. Der entstehende Fehler ist zufällig und wird als eine Art Rauschen wahrgenommen. Dies wird als Quantisierungsrauschen bezeichnet.

Weitere Probleme, die auftreten können:

3) Aliasing - Ein digitales Sample kann durch mehrere Sinuswellen repräsentiert werden.

FIGURE 3.5 Aliasing: Many Sine Waves Can Generate the Same Samples

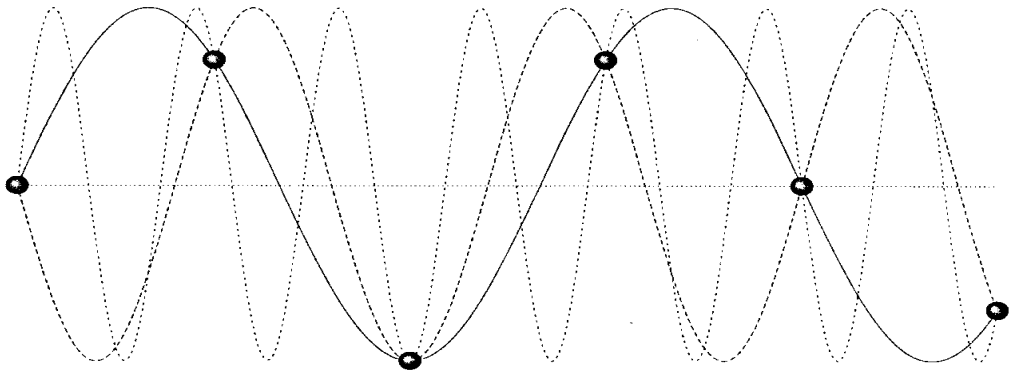


Abbildung 7: Aliasing [TM0]

Beim Abspielen kann es passieren, dass der DAC (Digital-Analog-Converter) nicht das Signal auswählt, dass man gerne möchte.

4) Clipping - Verzerrung. Die Spitzen der Sinuskurven werden abgeschnitten. Dieser Fehler tritt auf, wenn beispielsweise Verstärkerschaltkreise gesättigt sind oder es zu Variablen-überläufen kommt (Digitaltechnik).

Problembehandlung

Alle diese Probleme führen dazu, dass man Unterschiede zum Original wahrnimmt. Teils erkennt man Verzerrungen, teils ist ein Rauschen zu hören. Da das menschliche Gehör

Verzerrungen eher wahrnimmt, als ein gewisses Hintergrundrauschen, arbeiten manche Fehlerkorrekturen so, dass sie Verzerrungen teilweise in Hochfrequenzrauschen umwandeln. Eine Möglichkeit, dies zu tun, ist das Dithering. Die Idee dahinter ist die „error-diffusion“, wobei ein aufgetretener Fehler gespeichert und in die nächsten Quantisierungsschritte mit eingeht. Dadurch wird versucht, die Verzerrungen „zu vertuschen“.
Durch einhalten des Nyquist-Limits werden die Verzerrungen deutlich verringert.

Von Licht-Eindrücken zum Video

Der Begriff *Video* stammt von dem lateinischen Begriff *videre* (sehen) und bezeichnet Sequenzen von bewegten Bildern, die zur Bildschirmausgabe geeignet sind.

Wie sieht der Mensch?

Die 3-dimensionalen Welt wird auf die Netzhaut der Augen projiziert. Durch die unterschiedlich Perspektiven erkennen wir diese 2-dimensionalen Bilder als 3-dimensionales Bild. Das Auge hat Rezeptoren für farbiges Licht (rot, grün, blau). Die Fotorezeptoren registrieren Lichtwellen zwischen 400nm und 700nm. Ein Lichtstrahl mit einer Wellenlänge von 700nm erscheint rot, ein Lichtstrahl mit einer Wellenlänge von 500nm erscheint blau. Da die Empfindlichkeit dieser Rezeptoren für andere Farben genetisch kodiert ist, unterscheidet sie sich bei allen Menschen. Jeder Mensch hat also ein anderes Farbempfinden. Rot, Grün und Blau sind die Basis für alle anderen Farben, die durch Mischung aus diesen drei Grundfarben erzeugt werden.

Farbschemata

Es gibt verschiedene Möglichkeiten, Farben darzustellen. Als Ausgangsbasis benutzt man Grundfarben. Gebräuchliche sind:

RGB

Das RGB Modell (additive Farbmischung) benutzt die drei Grundfarben Rot, Grün, Blau. Eine Mischung aus den drei Komponenten ergibt bei gleicher Intensität weißes Licht.
Um verschiedene Farben zu mischen, variiert man die Intensitäten der drei Grundfarben. Mit dem RGB-Modell arbeiten VGA-Bildschirme.

CMYK

Die subtraktive Farbmischung bedient sich der Farben Cyan, Magenta und Gelb. Eine Mischung dieser drei Komponenten ergibt in der Theorie Schwarz, in der Praxis ein sehr dunkles Braun. Durch Zugabe von Schwarz (auch Tiefe genannt) enthält man auch im Bereich der unbunten Farben (Grau, Schwarz, Weiß) eine gute Reproduktionsqualität.
Dieses Modell wird allgemein CMYK genannt. Bei Druckern wird durch eine Patrone mit schwarzer Farbe die Lebenszeit der anderen 3 Patronen deutlich verlängert.

HSB - Hue (Farbton), Saturation (Sättigung), Britghtness (Helligkeit)

Das HSB-Modell entspricht unserer verbalen Farbbeschreibung am ehesten. Beschreibungen wie „ein kräftiges, helles Gelb“ lassen sich sofort umsetzen. Der Farbton beinhaltet die reine Farbinformation. Die Sättigung ist das Verhältnis von Stärke der reinen Farbe und den unbunten Anteilen. 0% ist immer ein Grauwert und 100 % die kräftigste, reine Farbe. Die Helligkeit entspricht der Helligkeit von 0% bis 100%. Dabei stellt 0% immer Schwarz dar, 100% immer Weiß.

YUV

Das YUV-Modell wird im analogen Fernsehen benutzt. Es besteht aus einem Helligkeitswert, der Luminanz (Y). Dieser ist das gewichtete Mittel der RGB- Anteile, wobei Rot zu 30%, Grün zu 59% und Blau zu 11% eingehen. Dies wird getan, um sich an die Empfindlichkeit des menschlichen Auges anzupassen. Die anderen beiden Komponenten sind zwei Farbanteile, die als Chrominanz U (oder Cr) und V (oder Cb) bezeichnet werden. Die U-Komponente ist die Differenz zwischen Weiß und Rot, die V-Komponente die Differenz zwischen Weiß und Blau.

Digitalisierung von Einzelbildern

Wie auch bei der Audio-Digitalisierung werden Bilder aus der physikalischen Welt in zwei Schritten digitalisiert.

Sampling

Die 2-dimensionale Fläche eines Bildes wird in kleine diskrete Regionen (Pixel oder auch Pel) unterteilt.

Quantisierung

Die Farbwerte der einzelnen diskreten Regionen (Pixel) müssen auf einen diskreten Wertebereich abgebildet werden. Dazu müssen die Werte gerundet werden. Dieser Prozess entspricht dem der Audio-Digitalisierung.
Abhängig von der Bildart müssen zu einem Pixel mehrere Werte abgespeichert werden. Sie erhöhen den möglichen Fehler – der auch hier als Quantisierungsrauschen bezeichnet wird.

Zweifarbtonbild

Bei einem Zweifarbtonbild reicht es ein Bit pro Pixel zu speichern, da die Helligkeit und der Farbton feststehen. Als Beispiel sei hier Text in einem Buch genannt.

Schwarzweißbild

Der Farbton ist für das ganze Bild konstant, es ändert sich lediglich der Helligkeitswert. Deshalb wird nur der Luminanzwert mit der entsprechenden Bit-Anzahl gespeichert.

Farbbild

Zu jedem Pixel werden 3 Komponenten gespeichert. Diese Komponenten sind in ihrer Bit-Anzahl und Art abhängig von der Größe und der Art des gewählten Farbschemas.

Digitalisierung von analogen Video-Signalen

Druckwellen nimmt man ab ca. 20 Hz als Ton wahr. Ebenso geschieht dies bei einer diskreten Folge von Einzelbildern, die als kontinuierliche Sequenz empfunden wird. Das menschliche Auge empfindet eine aufeinander folgende Darstellung von Einzelbildern ab einer Grenzfrequenz von etwa 16 Hz (16 Bilder pro Sekunde) als zusammenhängende Sequenz. Allerdings entsteht bis etwa 50 Hz ein Flimmereffekt durch die unvollkommene Speicherwirkung des Auges für optische Reize.

Bei Fernsehgeräten wird ein Vollbild in zwei zeilenweise ineinander geschachtelte Halbbilder geteilt. Es wird jeweils ein Halbbild nach dem anderen im Zeilensprungverfahren übertragen (*Interlace*-Verfahren). Zuerst werden alle ungeraden Zeilen übertragen, dann die geraden. Im PAL-Format wird jedes Halbbild 25 Mal pro Sekunde dargestellt, also beträgt die Vertikalfrequenz eines Vollbildes 50 Hz, wobei zwischen zwei Halbbildern 20 ms liegen.

Fernsehformate

PAL / SECAM

Das Fernsehformat PAL/SECAM wurde 1966/67 in Deutschland eingeführt. Es werden 25 Voll- bzw. 50 Halbbilder / Sekunde dargestellt. Das PAL-Signal wird durch die Quadratur-Amplituden-Modulation, das SECAM-Signal durch Frequenzmodulation umgesetzt. Es wird zusätzlich eine Synchronisation für Farbtreue durchgeführt. Das Fernsehbild besteht aus 625 Zeilen von denen aber nur 576 sichtbar sind. Das Bildformat hat eine Seitenverhältnis von 4:3. Die unterstützte Bandbreite ist 6.5 MHz und wird mit einer Frequenz von 13.5 MHz abgetastet. Also etwas größer als Nyquist-Limit.

NTSC

Die Einführung des NTSC – Formates in den USA war 1954. Auch in Kanada und Japan ist es das gängige TV-Format. Es werden 30 Voll- bzw. 60 Halbbilder / Sekunde ausgestrahlt.
Hier wird ebenfalls die Quadratur-Amplituden-Modulation angewandt, allerdings ohne Korrektur von Übertragungsfehler.
Das Seitenverhältnis der übertragenden Bilder ist ebenfalls 4:3, doch hat es nur 525 Zeilen von denen nur 480 sichtbar sind. Die Bandbreite des TV-Signals ist bei NTSC nur 5.5 MHz.

Abtastmethoden

4:1:1 – „Die Farbdifferenz-Signale U und V werden mit einem Viertel der Genauigkeit im Vergleich zum Luminanz-Signal Y dargestellt. Die Werte für Y, U und V werden für jede

Zeile ermittelt. Innerhalb einer Zeile wird ein Farbwert für jeweils 4 Bildpunkte (Pixels) verwendet. Dadurch bleibt die vertikale Auflösung der Farbinformation erhalten.“ [INT4]

4:2:0 – „Die Farbdifferenz-Signale U und V werden mit halber Genauigkeit im Vergleich zum Luminanz-Signal Y dargestellt. Allerdings werden U und V nur für jede zweite Zeile ermittelt. Es wird also abwechselnd eine Zeile im Verhältnis 4:2:2 und eine Zeile im Verhältnis 4:0:0, d. h. nur die Luminanz, kodiert. Dadurch wird ein Farbwert für je zwei Pixels neben- und untereinander verwendet. Die 4:2:0-Kodierung nutzt aus, dass das menschliche Auge die horizontale Auflösung besser als die vertikale wahrnimmt.“ [INT4]

4:2:2 – „Die Farbdifferenz-Signale U und V werden mit halber Genauigkeit im Vergleich zum Luminanz-Signal Y dargestellt. Beispielsweise zeichnet das analoge Betacam die Video-Signale in zwei Spuren auf: Eine Spur enthält das Luminanz-Signal Y, die andere Spur abwechselnd Blöcke für U und V. Dabei wird die Tatsache ausgenützt, dass das Auge weniger empfindlich auf die Farbe ist als auf die Helligkeit.“ [INT4]

4:4:4 – Hochqualitatives digitales Format: Jedes Pixel, sowohl in Luminanz wie in den Blau- und Rotdifferenzen wird abgetastet. „Alle drei Komponenten eines RGB- oder YUV-Signals werden in gleicher Qualität dargestellt.“ [INT4]

Quantisierung

Für ein einpoliges Signal, wie Luminanz, wird Quantisierung genutzt.

Für ein zweipoliges Signal, wie zum Beispiel für die Farbdifferenz (Cb, Cr), wird mid-tread Quantisierung angewandt. Damit wird sichergestellt, dass der Nullwert fehlerfrei ist. [CP1]

Auch hier treten durch das Runden der Werte bei de Quantisierung Fehler auf. Dieses Quantisierungsrauschen führt zu Verzerrungen der Farben und zu Ungenauigkeiten der Helligkeitswerte, was zur Folge hat, dass die digitalen Bilder im Gegensatz zu analogen Fernsehbildern nicht immer die gleiche Farbbrillianz haben.

Speicher-Probleme

Ein großes Problem bei der Digitalisierung von Video-Daten sind die entstehenden riesigen Datenmengen. Für ein PAL-Bild mit einer Größe von 768x576 Pixeln müssen pro Sekunde 33MB nur für die reinen Bildinformationen abgespeichert werden. Folgende kleine Rechnung erklärt den Wert:

PAL-Bild: 768x576 Bildpunkte bei 25 Bildern pro Sekunde
Für jeden Bildpunkt: 1 Helligkeitswert und 2 Farbwerte (jeweils 1 Byte = 8 Bit)

768 * 576 * 3 * 8 * 25 = 265420800 Bit = 33177600 Byte ≈ 33 MB

Vorteile und Nachteile der Digitaltechnik

Vorteile

Die Nachteile der analogen Bildaufzeichnung sind die Stärken der digitalen Speicherung. Bei mehrfachem Bearbeiten oder Kopieren verschlechtern sich digital gespeicherte Daten nicht, da sie nicht den Störeinflüssen, wie Pegelverschiebungen, Spannungsschwankung und auch der Abnutzung der Speichermaterialien unterliegen. Die Änderung in digital erfassten Daten ist wesentlich einfacher. Zeitlupenbilder und Standbilder sind kinderleicht zu realisieren.

Nachteile

Digital gespeicherte Daten haben einen wesentlich größeren Speicherplatzbedarf. Im Vergleich zu analogen System, die theoretisch einen unbegrenzten Farb- bzw. Grautonbereich besitzen, ist diese bei Digitalsystemen auf eine festgelegte Farbanzahl begrenzt.

Zusammenfassung

Töne und Bilder und der Weg in die digitale Form. Physikalische Größen werden zuerst in analoge Signale und dann durch Sampling und Quantisierung in digitale Formen gebracht. Es treten durch die Art des Abtastens des analogen Signals sowie durch das Runden der Werte auf einen diskreten Zieldatenbereich Fehler auf. Diese Fehler trüben die Qualität (Verzerrungen und Rauschen) der digitalen Ton- und Bilddaten.
Ein weiteres Problem ist die gewaltige Datenmenge, die bei der Digitalisierung entsteht. Um diese handhaben und bewältigen zu können, wurden verschiedene Komprimierungsverfahren entwickelt. Ein weiterer Weg die Datenmengen zu verkleinern, besteht darin, nur psychoakustisch bzw. psychooptisch relevante Daten zu speichern.

Abkürzungen

ADC	Analog-Digital-Converter
DAC	Digital-Analog-Converter
PAM	Puls-Amplitude-Modulation
PWM	Puls-Width-Modulation
PCM	Puls-Code-Modulation

Literatur

[TM0]	Tim Kientzle: „A Programmer's Guide to Sound“, Addison-Wesley, 1997
[CP1]	Charles Poynton: “A Technical Introduction to Digital Video”, J. Wiley & Sons, 1996
[AS2]	Axel Stolz: “Das große Soundblaster Buch”, Data Becker GmbH, 1992
[INT3]	http://cartoon.igwuw.tuwien.ac.at/fit/2001/fit09/1entstehung.html Entstehung von digitalem Video
[INT4]	http://www.uni-kiel.de/rz/video/video-sampling/ „Sampling-Verhältnisse bei Video Signalen“
[INT5]	http://www.lehre.informatik.uni-osnabrück.de/~mm/ Multimedia-Skript

Anhang

10^{-12}	0	Hörschwelle
10^{-11}	10	ruhiges Aufnahmestudio
10^{-10}	20	ruhiger Wohnraum
10^{-9}	30	ruhiges Büro
10^{-8}	40	gedämpfte Unterhaltung
10^{-7}	50	normales Büro
10^{-6}	60	gedämpfte, normale Unterhaltung
10^{-5}	70	kleines Orchester
10^{-4}	80	Geschäftsstraße, Fabrik
10^{-3}	90	Schwertransport
10^{-2}	100	U-Bahn
10^{-1}	110	laute Rockmusik, Donner
10^{0}	120	Flugzeug, Stadtbahn, Dampfhammer
10^{1}	130	Schmerzgrenze
10^{2}	140	
10^{3}	150	Flugzeugturbine in
10^{4}	160	abgeschlossenen Räumen

Tabelle 1: Zusammenhang Leistung in Watt/m^2 und Schallpegel in dB [INT5]

Quantisierung

von

Oliver Richter

Einleitung

Mit immer größeren Datenmengen, die über Netzwerke übertragen oder gesichert werden, gewinnt die Komprimierung dieser Daten immer mehr an Bedeutung. Bei einer

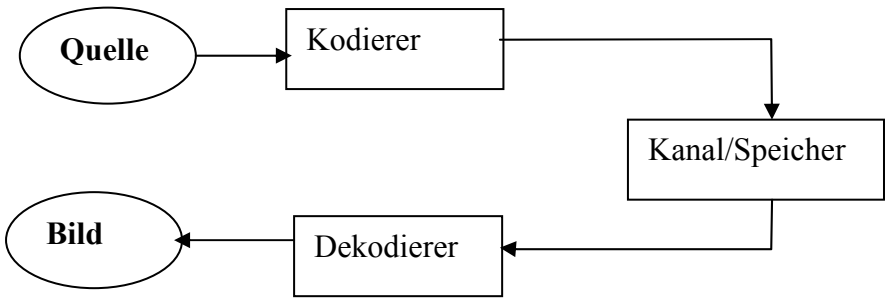


Abbildung 1: Struktur einer Datenkompression

Komprimierung wird die zu kodierende Datenmenge auf eine möglichst kleine Menge von Repräsentanten abgebildet. Abbildung 1 zeigt die Struktur einer Komprimierung. Eine Datenmenge durchläuft dabei folgende Schritte:

- Eine Datenquelle schickt einen Datenstrom in den Kodierer.
- Der Kodierer verarbeitet die eingehenden Daten und bildet sie auf eine Menge von Repräsentanten ab.
- Die kodierten Daten, in Form der Repräsentanten, können nun gespeichert oder auch über ein Netz übertragen werden.
- Der Dekodierer konstruiert aus den Repräsentanten die ursprüngliche Datenmenge oder eine Approximation davon.

Können die ursprünglichen Daten wieder vollständig hergestellt werden, spricht man von verlustfreier Komprimierung. Erhält man nach der Dekodierung nur eine Approximation der ursprünglichen Datenmenge spricht man von verlustbehafteter Komprimierung.
Die verlustfreien Komprimierungsmethoden werden bei Daten angewandt die keinen Datenverlust erlauben, zum Beispiel Text oder Programme.
Die verlustbehafteten Komprimierungsarten sind auf bestimmte Datenquellen, wie Bilder, Sprache oder Video, spezialisiert und können auf andere Datenquellen nicht angewendet werden. Diese Techniken bieten aber auch eine wesentlich höhere Kompressionsrate als die verlustfreien. Sie versuchen bei der Komprimierung die Informationen wegzustreichen die für die menschliche Wahrnehmung eine untergeordnete Rolle spielen.

Quantisierung allgemein

Die Quantisierung stellt eine der einfachsten Möglichkeiten dar, Daten verlustbehaftet Daten zu komprimieren.

Bei der Quantisierung wird eine große Menge A von Werten auf eine wesentlich kleinere Menge C von diskreten Werten abgebildet. Der Quantisierer stellt also eine Funktion dar, die die Definitionsmenge vollständig auf die Bildmenge abbildet.

$$Q : A \rightarrow C$$

Die Definitionsmenge kann überabzählbar groß und theoretisch auch unbeschränkt sein. Häufig wird aber ein Definitionsbereich betrachtet der in einem beschränkten Interwall liegt, wie zum Beispiel Farbwerte oder Lautstärken in einem bestimmten Interwall.

Anschaulich kann man sich einen Quantisierer als ein Verfahren vorstellen, das ein stetiges Eingangssignal auf eine Menge diskrete Werte „rundet".

Aufbau und Arbeitsweise eines Quantiesierers

Ein kompletter Quantisierer ist aus drei Komponenten aufgebaut, der Kodierungsfunktion, der Dekodierungsfunktion und dem Codebuch.

Die Kodierungsfunktion ist der Teil eines Quantisierers, der die verlustbehaftete Komprimierung durchführt. Sie teilt den Eingangsbereich des Quantiesierers in paarweise disjunkte Teilmengen und weist jeder Teilmenge einen Index zu.

$A = \bigcup_i A_i$ mit $A_i \neq A_J$ für alle i,j mit i $\neq$ j

Wird nun ein Wert der Kodierungsfunktion übergeben, überprüft sie in welchen Teilbereich der Wert liegt und gibt den Index des Teilbereiches zurück. Da die Teilmengen paarweise disjunkt sind, ist diese Zuweisung eindeutig und man kann sie als Funktion darstellen.

$Q_K(a) = i$ für $a \in A_i$

Das Codebuch ist eine Liste der Indizes und der Repräsentanten (Codewörter) der einzelnen Teilmengen. Die Repräsentanten einer Teilmenge sind die Werte auf denen die Elemente einer Teilmenge „gerundet" werden.

Codewort	1
Codewort	2
Codewort	n

Abbildung 2:Codebuch

Von der Art der Kodierungsfunktion sowie der Wahl der Repräsentanten innerhalb der Teilmengen hängt die Qualität des Quantiesierers ab. In den folgenden Kapiteln werden die einzelnen Möglichkeiten für die Einteilung und Wahl der Repräsentanten genauer beschrieben.

Die Dekodierungsfunktion wandelt mit Hilfe des Kodebuches einen Index in ein Element der Definitionsmenge zurück.

$$Q_D(i) = \begin{cases} Codewort_1, i = i_1 \\ Codewort_2, i = i_2 \\ ... \\ Codewort_n, i = i_n \end{cases},$$

Q_D ist die Umkehrung der Kodierungsfunktion aber keine Umkehrfunktion der Kodierungsfunktion Q_K, da

Q_D den Index nur auf eine Teilmenge des Eingangsbereiches abbildet.

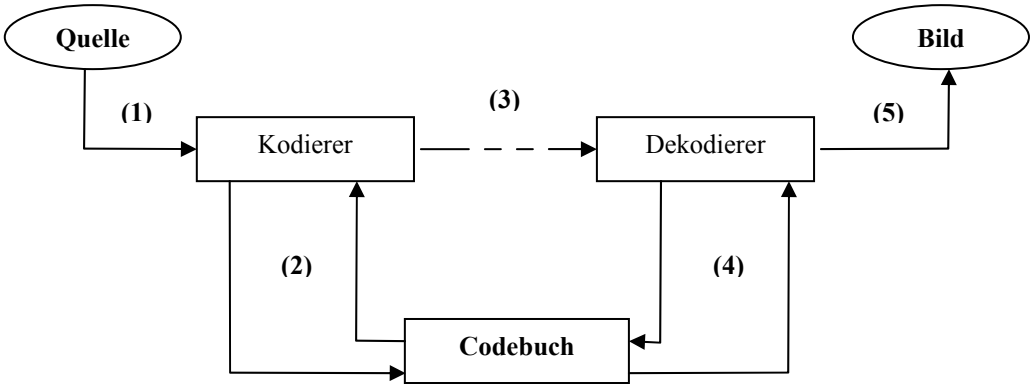


Abbildung 3 zeigt de **Abbildung 3: Struktur eines Quantisierers** enten eines Quantisierers.

1. Von der Quelle werden einzelne Werte nacheinander an den Kodierer übergeben. Je nach dem, welche Form diese Werte haben kann man zwei grundlegende Arten von Quantiesierern unterscheiden. Sind die Werte Skalare, spricht man von einem Skalarquantisierer, sind die Werte Vektoren spricht man von einem Vektorquantisierer.
2. Der Kodierer weist den eingehenden Wert einem Bereich zu. Theoretisch braucht der Kodierer das Codebuch für die Einteilung nicht, es genügt wenn dem Kodierer die Einteilung des Eingangsbereiches bekannt ist.
   In der Praxis wird meistens mit den Repräsentanten und einer Abstandsfunktion, in der Regel dem Euklidischen Abstand,

$$D(X,Y) = \sqrt{(X-Y)^2}$$

   Die Einteilung des Eingangsbereiches wird bestimmt indem man einen eingehenden Wert mit dem „Nearest Neighbor Algorithmus“ dem am nächsten liegenden Repräsentanten zuordnet. Hat der Kodierer den nächsten Nachbarn gefunden, gibt er den Index des Repräsentanten aus dem Codebuch zurück.

Nearest Neighbor Algorithmus

```
1. Setze        d = maximalen Wert;
           j = 1;
           i = 1;
```

```
2. Berechne  Dj(X,Yj);
3. Ist  Dj < d setze d = Dj;
                                  i = j;
4. Ist  j < n    setze j = j + 1;
                    gehe zu 2.;
5.  gebe i zurück;
```

3. Mit dem Index hat man eine verkleinerte Darstellung des ursprünglichen Wertes. Der Index kann dann übertragen oder gespeichert werden.

4. Damit man eine Approximation des kodierten Signals erhält müssen die einzelnen Werte noch dekodiert werden. Der Dekodierer gibt einfach den Eintrag des Codebuches an der Stelle i zurück.

Kompressionsrate und Fehler

Bei einer verlustbehafteten Kompression versucht man immer eine möglichst hohe Kompressionsrate mit einem möglichst geringen Fehler zu erreichen.

Die Kompressionsrate ist ein Maß für die Verkleinerung der Quelle. Sie hängt nur von der Größe des gewählten Codebuches ab. Ist n die Anzahl und k die Länge der Codewörter berechet sich die Kompressionsrate mit

$$r = \frac{\lceil \log_2 n \rceil}{k}$$

Daraus folgt, das die Kompression mit einem kleineren Codebuch Größer ist.

Mit der Abstandsfunktion kann man zu den einzelnen quantisierten Werten einen Fehler angeben. Der Fehler entspricht dem Abstand des quantisierten Wertes x zu dem Codewort c auf das der Wert x abgebildet wurde.

$$\Delta = D(x,c)$$

Als Maß für den Fehler der kompletten Quantisierung einer Quelle verwendet man den mittleren Quadratischen Fehler σ^2.

$$\sigma^2 = \frac{1}{i}\sum_i \Delta_i^2$$

und für eine Verteilung deren Funktion bekannt ist

$$\sigma^2 = \sum_M \int_{i-1}^{i} D(x,c) * f(x)$$

Da man den Fehler minimieren und die Kompression maximieren will, benötigt man ein Vergleichkriterium das beide Angaben der Quantisierung berücksichtigt. Es gibt zwei Vergleichskriterien mit denen Quantisierungsverfahren bewertet werden: Die „signal-to-noise-ratio“ SNR und die „peak-signal-to-noise-ratio“ PSNR.

Die SNR berechnet sich aus der mittleren Eingabe, dem Mittelwert der Elemente der Quelle

$$m = \frac{1}{n}\sum_i x_i$$

und dem mittleren quadratischen Fehler

$$SNR = \frac{m^2}{\sigma^2}$$

Die SNR wird verwendet um die Güte von Skalarquantisierern zu beurteilen und wird damit hauptsächlich bei Audiokomprimierung verwendet.

Die PSNR berechnet sich aus dem Maximal möglichen Wert M und dem mittleren Quadratischen Fehler.

$$PSNR = \frac{M^2}{\sigma^2}$$

Mit der PSNR bestimmt man die Güte von Vektorquantisierern und wird für die Beurteilung von Bildkompression verwendet. Der maximale Wert M ist dann der maximal mögliche Farbwert.

Skalarquantisierung

Die einfachste Möglichkeit eines Quantisierers ist die Quantisierung einzelner Quellenausgaben, die quantisierung von Skalaren. So ein Quantisierer wird Skalarquantisierer genant. SQ werden vor allem bei Analog-Digital-Wandlern und zeitlich abhängigen Quellen, wie bei der Audio und Video Komprimierung, verwendet.

Uniforme Quantisierer

Bei einem uniformen Quantisierer teilt man den Eingangsbereich in eine begrenzte Anzahl gleich großer Intervalle, sog. Zellen, ein. Wenn der Eingangsbereich nicht beschränkt ist, gibt es zwei Zellen die ins unendliche gehen. Die Zellen werden Überladungszellen genant. Die Repräsentanten der Zellen sind ihre Schwerpunkte, die Mittelwerte aller möglichen Werte der Zelle. Abbildung 4 veranschaulicht die Aufteilung einer skalaren Eingangsgröße.

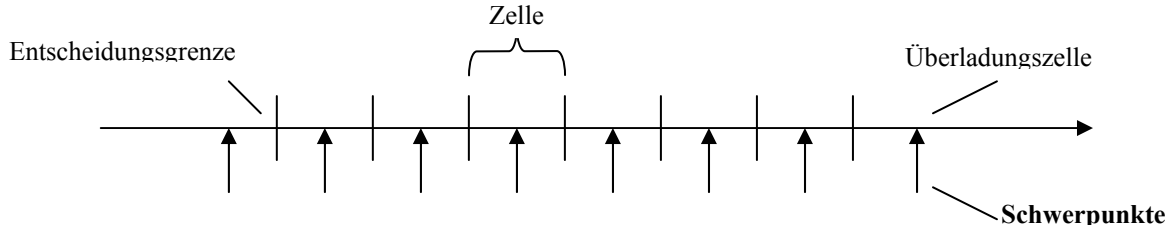


Abbildung 4: uniformer Quantisierer

Der maximale Fehler einer beschränkten Zelle ist durch die Interwallgröße Festgelegt, während der Fehler einer Überladungszelle ins unendliche gehen kann.

Nonuniforme Quantisierer

Sind die Werte der Quelle nicht gleichverteilt, sondern gewichtet, wie zum Beispiel bei einer Normalverteilung, werden einige Zellen häufiger getroffen als andere. Kennt man die Verteilungsfunktion der Quelle, kann man eine gewichtete Einteilung des Quellbereiches

Vornehmen und damit den MSE verkleinern und damit die SNR reduzieren und Qualität der Komprimierung verbessern.

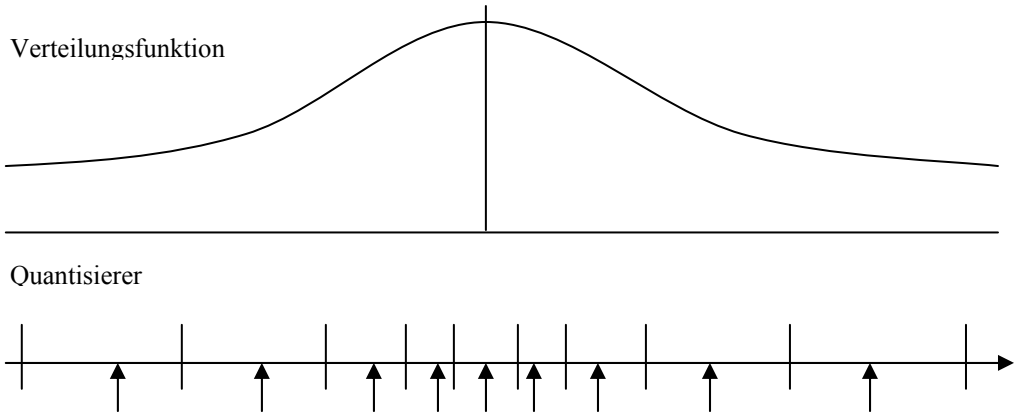


Abbildung 5:nonuniformer Skalarquantisierer

Abbildung 5 zeigt die Verteilungsfunktion einer Quelle und die angepasste nonuniforme Aufteilung des Skalarquantisierers.

Der Loyd-Algorithmus

Wenn man nun die Verteilungsfunktion kennt, wie kann man die Aufteilung des Quantisierers daran anpassen. Der Loyd-Algorithmus bietet die Möglichkeit, eine vorhandene Aufteilung zu verbessern so das sie beliebig genau an der idealen Aufteilung liegt.

Der Loyd-Algorithus beginnt mit einem Anfangscodebuch und verbessert mit Hilfe der Verteilungsfunktion iterativ das vorhandene Codebuch.

Loyd-Algorithmus

1. Wähle Initialcodebuch $\{y_i^k\}_{i=1}^M$ der Größe M;
   Genauigkeitsschranke ε;
   Setze k = 0;
   $d^{(0)} = 0$;

2. $$b_j^{(k)} = \frac{y_{j+1}^{(k)} + y_j^{(k)}}{2}; j = 1..M-1$$

3. $$d^{(k)} = \sum_{i=1}^{M} \int_{b_{i-1}^{(k)}}^{b_i^{(k)}} D(x, y^{(k)})^2 * f(x)$$

4. Ist $\frac{d^{(k)} - d^{(k-1)}}{d^{(k)}} < \varepsilon$ beende;

5. $y_i^{(k)} = \frac{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} x * f(x)dx}{\int_{b_{j-1}^{(k-1)}}^{b_j^{(k-1)}} f(x)dx}$;

Gehe zu 2;

Der Loyd-Algorithmus berechnet in jedem Durchlauf die Entscheidungsgrenzen in Abhängigkeit der Verteilungsfunktion und mit diesen Grenzen die neuen Schwerpunkte der neuen Interwalle. Die Neuberechnung der Interwallgrenzen und Schwerpunkte wird so lange wiederholt bis die Änderung des MSR die Genauigkeitsschranke unterschreitet.

Vektorquantisierung

Bei der Vektorquantisierung werden nicht nur einelementige Quellenausgaben kodiert sondern auch mehrdimensionale Elemente, Vektoren. So gesehen ist der Skalarquantisierer ein eindimensionaler Vektorquantisierer, also ein Spezialfall des Vektorquantisierers. Ein Vektorquantisierer wird meistens auf schon digitalisierte Werte angewendet, zum Beispiel zur Komprimierung von Bildern.

Vergleich von Skalar und Vektorquantisierer

Welchen Vorteil hat nun die Betrachtung mehrdimensionale Elemente gegenüber der Kodierung einzelner Skalare. Zu dieser Frage betrachten wir das Beispiel eines zweidimensionalen Quantisierers.

Betrachten wir die Größe und das Gewicht von Personen. Man kann davon ausgehen, das die Größe einer befragten Person zwischen 120 cm und 200 cm und ihr Gewicht zwischen 30 kg und 100 kg schwankt.
Beschreiben wir das Problem mit SQ erhalten wir zwei SQ's, einen für das Gewicht und einen für die Größe.
Teilen wir beide Quellenbereiche in Zellen der Größe 5 ein erhalten wir eine zweidimensionale Einteilung wie in Abbildung 6 beschrieben.

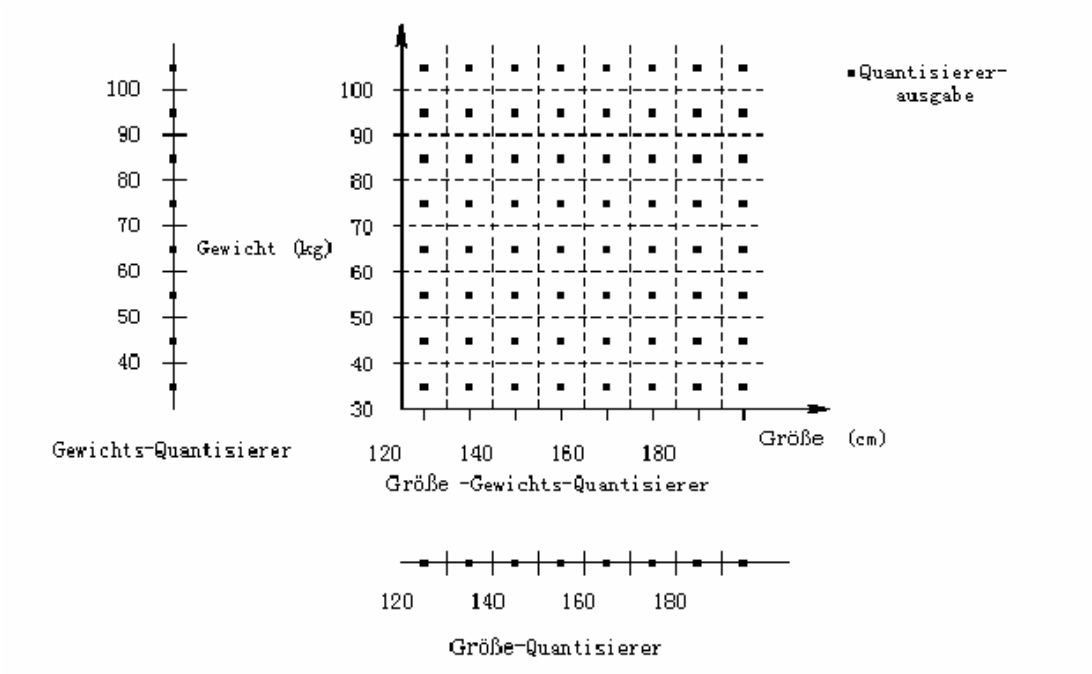


Abbildung 7: Skalarquantisierer

Nun kann man davon ausgehen, dass Größe und Gewicht einer Person nicht unabhängig voneinander sind, sondern in einem bestimmten Verhältnis zueinander stehen. Zum Beispiel wird es kaum Personen geben, die 120 cm groß sind und 100 kg wiegen. Bei dem Skalarquantisierer werden dann Bereiche abgedeckt die überhaupt nicht auftreten, oder zumindest so unwahrscheinlich sind, dass man sie vernachlässigen kann. Auch wenn man keine Gleichverteilung annimmt und den Größe und den Gewicht Quantisierer nonuniform aufteilt werden immer noch ungenutzte Bereiche beschrieben.

Teilen wir nun den zweidimensionalen Größe-Gewicht Raum mit Vektoren können wir die Schwerpunkte entlang der Linie verteilen die das Idealgewicht beschreibt. Abbildung 8 zeigt, wie eine solche Einteilung aussehen könnte.

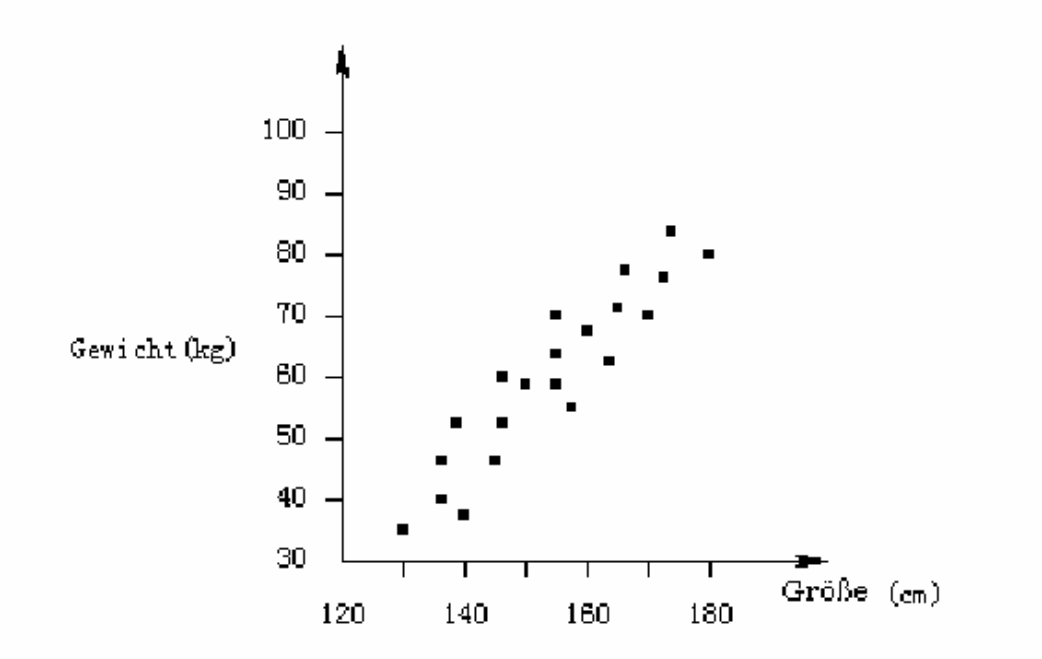


Abbildung 8: Vektorquantisierer 1

Man sieht, dass die Schwerpunkte sich um eine Geraden verteilen, die das Idealgewicht darstellt. Die unwahrscheinlichen Bereiche können mit dieser Einteilung noch mit einem großen Fehler dargestellt werden aber die Bereiche mit einer hohen Wahrscheinlichkeit sind jetzt mit einem wesentlich geringeren Fehler darstellbar.

Codebuchgenerierung

An dem Beispiel des Größe-Gewicht Quantisierers kann man erkennen, dass der Vektorquantisierers eine Verallgemeinerung des nonuniformen Skalarquantisierers ist. Bei einem nonuniformen Skalarquantisierer kann man mit Hilfe des Loyd-Algorithmus eine an die ideale Aufteilung angenäherte Einteilung des Quellbereiches erhalten.

Verallgemeinert man den Loyd-Algorithmus auf n Dimensionen kann man dieselbe Näherung mit einem Vektorquantisierer vornehmen.

Wenn man jetzt zum Beispiel ein Bild mithilfe der Vektorquantisierung komprimieren will hat man das Problem, das man die Verteilungsfunktion der Quelle nicht kennt. Dieses Problem kann man umgehen, wenn man den Loyd-Algorithmus so erweitert, dass das Codebuch mit Trainingsvektoren abgeglichen wird.

Der so erweiterte Algorithmus heißt Linde-Buzo-Grey-Algorithmus.

Linde-Buzo-Grey-Algorithmus

1. Wähle Initialcodebuch $\{y_i^k\}_{i=1}^M$ der Größe M;
   Genauigkeitsschranke ε;
   Setze k = 0;
   $d^{(0)} = 0$;

2. Setze Quantisierungsbereiche
   $$V_i^{(k)} = \{x_n : d(x_n, y_i) < d(x_n, y_j); \forall j \neq i\}$$

3. Berechne $D^{(k)}$

4. Ist $\frac{d^{(k)} - d^{(k-1)}}{d^{(k)}} < \varepsilon$ beende;

5. Setze k = k+1;
   $y_i^{(k)}$ = Mittelwert der Quantisierungsbereiche;
   Gehe zu 2.

Der LBG ordnet im jedem Durchgang die Trainingsvektoren neu den Quantisierungsbereichen zu. Aus den Trainingsvektoren der einzelnen Quantisierungsbereiche wird der Schwerpunkt berechnet und als neuer Schwerpunkt des Quantisierungsbereiches in das Codebuch eingetragen. Der LBG geht davon aus, das kein Quantisierungsbereiche leer ist.

Ist einer der Quantisierungsbereiche leer kann der LBG den Durchschnitt der Trainingsvektoren nicht berechnen, denn Division durch null ist nicht möglich. Man dieses Problem das Empty-Cell-Problem. Man kann dieses Problem umgehen indem man den Codebuchvektor nicht weiter verändert und darauf hofft, dass sich im nächsten Durchlauf wieder Vektoren in diesem Quantisierungsbereich befinden. Dann kann es aber vorkommen, dass ein Quantisierungsbereich einsteht, der nie genutzt wird. In der Praxis streicht man meistens diesen Vektor und ersetzt ihn durch einen neuen.

Betrachten wir den Algorithmus an dem Beispiel des Größe-Gewicht Quantisierers. Wir wählen als Initialcodebuch die Tabelle 2. Abbildung 9 zeigt die Aufteilung der Quantisierungsbereiche vor dem Durchlauf des LBG Algorithmus.

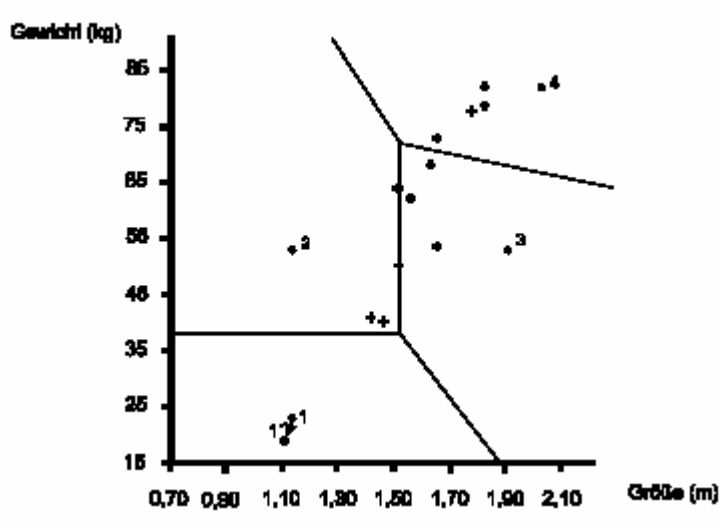


Abbildung 9: Startaufteilung

Der MSE vor dem Ausführen des Algorithmus ist d(1) = 76,35. Nach dem ersten Durchlauf d(2) = 53,01und nach dem dritten Durchlauf d(3) = 53,01. Der MSE hat sich vom Zweiten zum dritten Durchlauf nicht geändert und der Algorithmus terminiert. Abbildung 10 zeigt die endgültige Aufteilung der Quantisierungsbereiche.

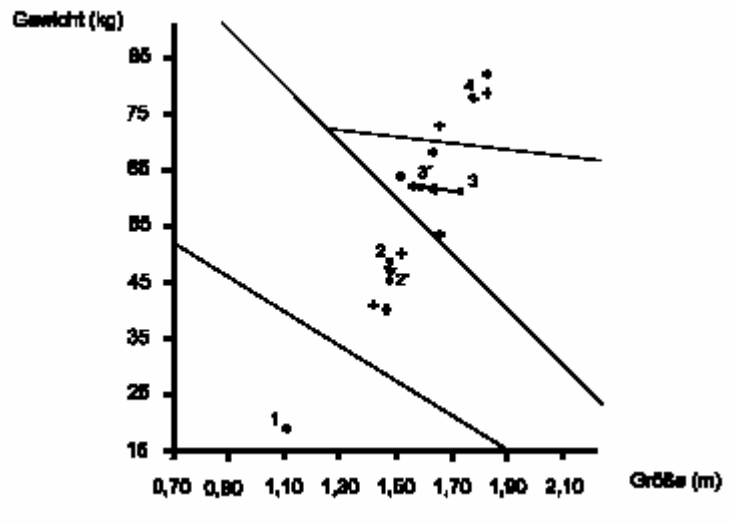


Abbildung 10: Endaufteilung

Was passiert mit der Aufteilung wenn man ein anderes Initialcodebuch wählt? Wählt man als Initialcodebuch die Tabelle 3 und führt den LBG durch erhält man die Start und Endaufteilung in den Abbildungen 11 und 12.

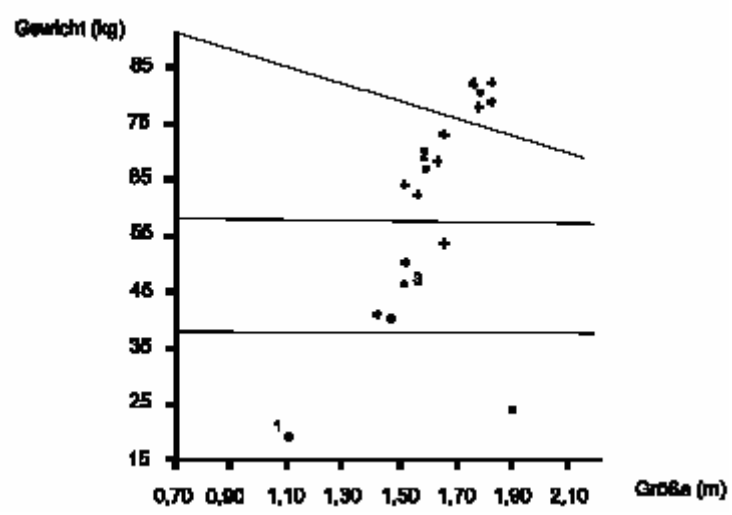


Abbildung 11: Startaufteilung

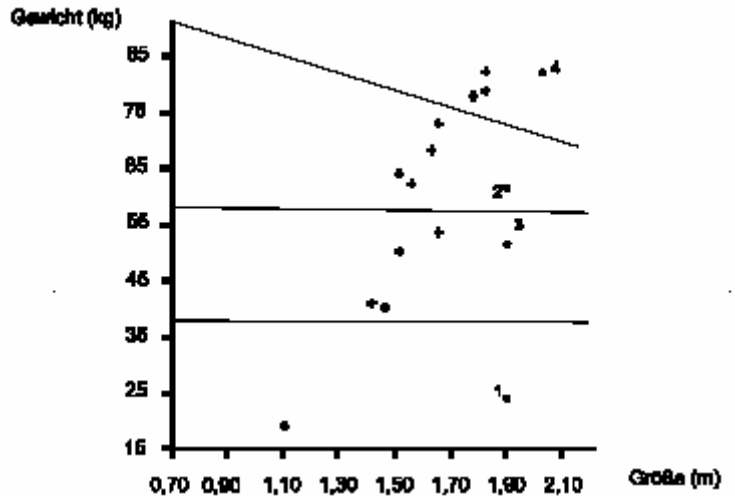


Abbildung 12: Endtaufteilung

Man kann an den Abbildungen 9 – 12 gut erkennen das die endgültige Einteilung stark von der Wahl des Initialcodebuches abhängt.

Wahl des Startcodebuches

Die einfachste Möglichkeit ein Initialcodebuches zu wählen ist zufällige Vektoren der Quelle zu wählen. Mit dieser Möglichkeit kann man die beste aber auch die schlechteste Möglichkeit wählen. Es gibt aber auch Möglichkeiten eine gezieltere Wahl zu treffen.

Die Splitting Methode fängt mit dem Schwerpunkt über alle Trainingsvektoren an und erweitert das Codebuch indem sie einen Vektor durch zwei andere ersetzt und dann den LBG durchführt

Splitting Algorithmus

1. Bilde Schwerpunkt aller Trainingsvektoren

2. Addiere / Subtrahiere eine Abweichung zu jedem Vektor und ersetze sie durch die neu entstandenen

3. Führe den LBG Algorithmus durch

4. Ist die gewünschte Codebuchgröße erreicht beende sonst gehe zu 2

Eine andere Möglichkeit ist paarweise nächste Nachbarn Methode. Man beginnt mit einem Initialen Codebuch das alle Trainingsvektoren enthält und sucht in jedem Schritt die nächsten Nachbarn. Hat man die nächsten Nachbarn gefunden entfernt man beide Vektoren und ersetz sie durch ihren Schwerpunkt.

PNN Algorithmus

1. Fülle Codebuch mit allen Trainingsvektoren

2. Finde nächste Nachbarn und berechne ihren Schwerpunkt

3. Entferne die n.N. und füge ihren Schwerpunkt in das Codebuch

4. Ist die gewünschte Codebuchgröße erreicht beende sonst gehe zu 2.

Man kann in der Praxis nicht genau sagen ob der PNN oder der Splitting Algorithmus das bessere Ergebnis liefert. Da aber die Laufzeit des Splitting Algorithmus besser ist, wird in den meisten Fällen die Splitting Methode verwendet.

Bildkomprimierung mit Vektorquantisierung

In der Praxis verwendet man die Vektorquantisierung zur Komprimierung für Bilder nur mit einer Vorverarbeitung, zum Beispiel der diskreten Kosinustransformation. Man kann aber auch nur mit der Vektorquantisierung eine Komprimierung durchführen.

Um mit Hilfe von Trainingsvektoren ein Codebuch zu erzeugen teilt man das Bild in Bereiche und löst diese in Vektoren auf. Abbildung 13 zeigt eine Möglichkeit vier mal vier Blöcke zu Vektoren umzuformen.

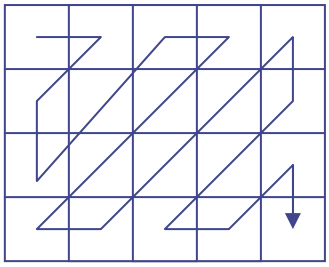


Abbildung 13

Bei einem 512*512 großen Bild erhält man so 1664 Trainingsvektoren. Man kann erkennen das bei einer so großen Menge die Wahl des Initialcodebuches mit PNN zu einer hohen Laufzeit führen würde, darum betrachten wir in diesem Beispiel nur die Splitting und Random Methode.

Die folgenden Bilder zeigen Images die mit dieser Methode komprimiert wurden. Die dazugehörigen Graphiken zigen die Güte der Komprimierung, PSNR, über die Bitrate von einer Randomisierten Wahl und einer Wahl nach der Splitting Methode.

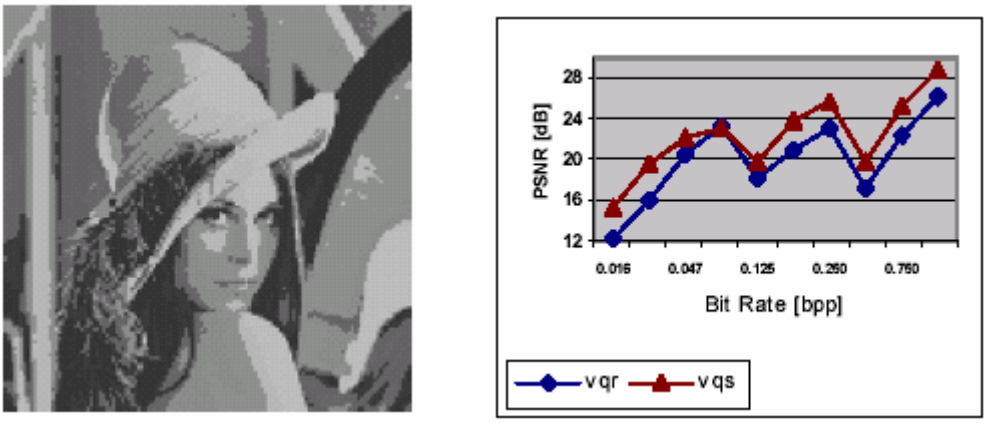


Abbildung 15

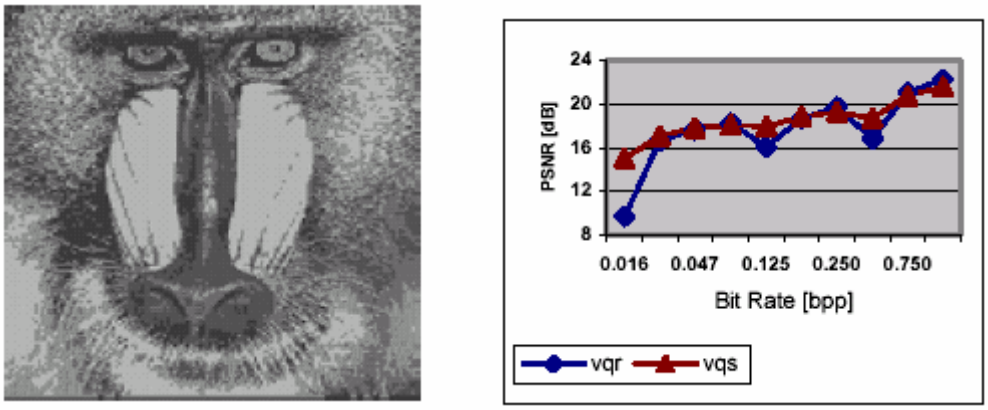


Abbildung 16

Man kann erkennen das bei jeder der dargestellten Bitraten die Splitting Methode ein genauso gutes oder besseres Ergebnis in der Güte der Komprimierung zeigt.

Zusammenfassung

Die Quantisierung ist die Grundlage für eine verlustbehaftete Komprimierung und man hat mit ihr eine Möglichkeit zwischen Genauigkeit und Kompressionsrate zu wählen um Quellen zu komprimieren.

Je nach Art der Quelle verwendet man die Kodierung einelementiger Samples, mit dem Skalarquantisierer, oder die Kodierung mehrdimensionaler Samples, mit einem Vektorquantisierer.

Der Skalarquantisierer bietet eine schnelle und einfache Kodierung und wird daher bei zeitlich abhängigen Signalen verwendet.

Der Vektorquantisierer ist in der Kodierung wesentlich langsamer da er für die Codebuchgenerierung aufwendige Algorithmen benutz und damit auf großen Mengen von Trainingsvektoren arbeitet. Er kann aber die Quelle in beliebig geformte Bereiche aufteilen und erreicht damit eine höhere Güte in der Komprimierung.

Abkürzungen

SQ	Skalarquantisierer
VQ	Vektorquantisierer
SNR	Signal to nois ratio
PSNR	Peak signal to noise ratio
MSE	mean sqare error

Literatur

[qu_1] Bastian Schoofs and Sebastian Reinartz , Scalar Quantization , TH Aachen http://www-i3.informatik.rwth-aachen.de/teaching/02/prosemhp/ausarbeitungen/07_ScalarQuantization.pdf

[qu_2] Yubo Qiu, Vektorquantisierung , TH Aachen http://www-i3.informatik.rwth-aachen.de/teaching/02/prosemhp/ausarbeitungen/08_VectorQuantization.pdf

[qu_3] Charniak , F. McDermott , Artificial Intelligence, Addison Wesley, 1. Mai 1985, ISBN: 0201119455

[qu_4] Nopparat Pantsaena, M. Sangworasil, C. Nantajiwakornchai and T. Phanprasit , Image Compression using Vector Quantisation,The Research Center for Communication and Information Technology (ReCCIT) Dept. of Electronics, Faculty of Engineering King Mongkut's Institute of Technology Ladkrabang , Department of Electronics Engineering
School of Engineering Bangkok University

Anhang

Größe (m)	Gewicht (kg)
1,83	82
1,83	79
1,65	54
1,12	19
1,50	64
1,57	52
1,63	68
1,52	50
1,65	73
1,42	41
1,45	40
1,77	78

Tabelle 1

i	Größe (m)	Gewicht (kg)
1	1,14	23
2	1,14	53
3	1,91	53
4	2,03	82

Tabelle 2

i	Größe (m)	Gewicht (kg)
1	1,91	23
2	1,91	58
3	1,91	53
4	2,03	82

Tabelle 3

Wichtige Transformationen: FFT, DCT und Wavelets

von

Yvonne Schindler

Einleitung

Das Ziel einer jeden Transformation ist es, Daten in eine andere Repräsentation zu bringen, die Vorteile für die anschließenden Operationen bringt. Durch eine Rücktransformation müssen die Daten wiederherstellbar sein. Durch Ungenauigkeiten bei der Berechnung ist dies aber nicht bei allen Transformationen immer möglich. Im Allgemeinen reicht aber die Genauigkeit für den Anwender aus [UniKL-www].
Ein Beispiel für eine einfache Transformation [Mannheim-www] ist die Berechnung des Logarithmus reeller Zahlen. Die Division reeller Zahlen ist aufwendiger, als die Berechnung der Logarithmen der zwei Zahlen, z.B. mit Hilfe von Logarithmentafeln, die Subtraktion der zwei Logarithmen und die Rückrechnung der Logarithmen (siehe Abbildung 1). Nach den Logarithmengesetzen erhält man auch auf diesem Weg die richtige Lösung.

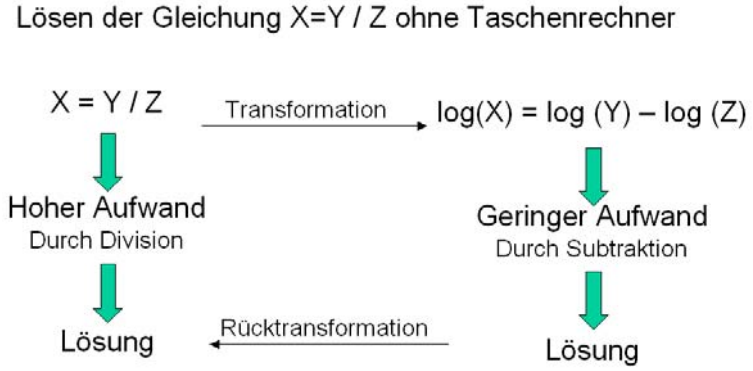


Abbildung 1: Transformationsbeispiel

Der vorliegende Text erläutert drei wichtige Transformationsverfahren für die Datenkompression.
Die Fourier Transformation und die schnellere Variante, die Fast Fourier Transformation. Die diskrete Cosinus Transformation, welche sich aus der Fourier-Transformation ableitet und die neueren Wavelets.

FFT - Fast Fourier Transformation

1822 verfasste Jean-Baptiste-Joseph Fourier sein Buch „Die analytische Theorie der Wärme“. Darin kam er zu der Erkenntnis, dass man Funktionen als Summe von unendlich vielen Sinus- und Kosinusfunktionen darstellen kann. Nichtperiodische Funktionen brauchen unendliche viele Sinus- und Kosinusfunktionen. Periodische brauchen endlich viele. Durch die unendliche Anzahl ist klar, dass eine gewisse Datenreduktion zustande kommt, da der Computer in der Praxis nicht mit unendlich vielen Werten rechnen kann.

1-dimensionale Fouriertransformation

Die 1-dimensionale Fouriertransformation geht von einem Vektor mit n Daten aus den reellen Zahlen in einen Vektor in den komplexen Zahlen [FhFlensburg-www].
Über die Umkehrfunktion kann man dann wieder den Vektor in den reellen Zahlen erzeugen. Damit ist die Fouriertransformation theoretisch verlustfrei. Praktisch hat der Rechner aber eine bestimmte Genauigkeit. Die reicht aber auch aus, denn es ist kein Unterschied für den

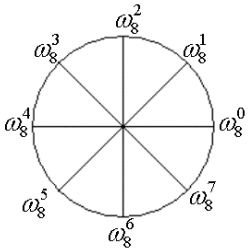
$$\begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_{n-1} \end{pmatrix} = \frac{1}{\sqrt{n}} \begin{pmatrix} \text{Fourie} \\ \text{r-} \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

Menschen sichtbar [Ekreide1-www].

n-te Einheitswurzel

ωn = n-te Einheitswurzel

$$\omega_n = \cos(\frac{2\Pi}{n}) + i * \sin(\frac{2\Pi}{n}) = e^{\frac{2\Pi i}{n}}$$



x^n hat in *C* n Lösungen

Bsp.: x^8 hat 8 Lösungen: $\omega_8^0, \omega_8^1, \omega_8^2, \omega_8^3, \omega_8^4, \omega_8^5, \omega_8^6, \omega_8^7$

Fouriermatrix

Def.: Sei $n \in N$ und w_n n-te Einheitswurzel in C. Die $n^x n$-Matrix F mit $F_{k,l} = w_n^{k*l}$ für alle k, l {0, ..., n-1}, heißt Fouriermatrix.
Demnach würde Fouriermatrix für n Daten diesen Aufbau haben:

$$\begin{pmatrix} f_0 \\ f_1 \\ F_{k,l}^{-1} \\ f_{n-1} \end{pmatrix} = \frac{1}{\omega_n^{-k*l}\sqrt{n}} \begin{pmatrix} \omega_n^o & \omega_n^o & \omega_n^o & \cdots & \omega_n^o \\ \omega_n^o & \omega_n^1 & \omega_n^2 & & \omega_n^{n-1} \\ \omega_n^o & \omega_n^2 & \omega_n^4 & & \omega_n^{2(n-1)} \\ \vdots & & & \ddots & \\ \omega_n^o & \omega_n^{n-1} & \omega_n^{2(n-1)} & & \omega_n^{(n-1)*(n-1)} \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}$$

Für eine Transformation muss es aber auch eine Rücktransformation geben. Diese wird durch eine inverse Fouriermatrix realisiert. Da F unitär ist, entspricht die inverse Fouriermatrix auch der transponierten Fouriermatrix, d.h. es müssen nur Spalten und Zeilen getauscht werden.

Um zu zeigen, dass diese Matrix, wirklich die inverse Fouriermatrix ist, muss man die beiden Matrizen, also die Fouriermatrix und die Inverse, miteinander multiplizieren.

$$F_n * F_n^t = \frac{1}{\sqrt{n}}\left[\omega_n^{kl}\right] * \frac{1}{\sqrt{n}}\left[\omega_n^{-kl}\right]$$
$$= \frac{1}{n}\left[\sum_{i=0}^{n-1} \omega_n^{ki}\omega_n^{-il}\right]$$
$$= \frac{1}{n}\left[\sum_{i=0}^{n-1} \omega_n^{i(k-l)}\right]$$
$$= \frac{1}{n}\left[\sum_{i=0}^{n-1} \omega_n^{ic}\right]$$
$$= \frac{1}{n}\left[\sum_{i=0}^{n-1} \left(\omega_n^{c}\right)^{i}\right]$$

$$\sum_{i=0}^{n-1}\left(\omega_n^c\right)^i = \frac{\left(\omega_n^c\right)^n - 1}{\omega_n - 1} = \frac{\left(\omega_n^n\right)^c - 1}{\omega_n - 1} = 0$$

$$= \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ \vdots & & \ddots & \\ 0 & 0 & & 1 \end{pmatrix}$$

$$\left(F_n\left(F_nB\right)^T\right)^T = F_nBF_n^T = F_nBF_n$$

=> $F^t = F^{-1}$

2-dimensionale Fouriertransformation

Bei der 2-dimensionalen Fouriertransformation wird die Fouriermatrix zweimal aud die Daten angewandt. Nach der ersten Anwendung muss das Ergebnis transponiert werden, damit die Transformation auf Spalten und auf Zeilen angewandt werden kann.

Zur Rücktransformation wird die inverse Fouriermatrix links- und rechtsseitig multipliziert. Auch hier kann man zeigen, dass man durch Anwendung der Fouriermatrix und danach

$$F_n^tBF_n^t \qquad F_n^t(F_nBF_n)F_n^t = IBI = B$$

Anwendung der inversen Fouriermatrix, die ursprünglichen Daten wiederherstellen kann.

Fast Fouriertransformation

Um die Fouriertransformation schneller zu machen nutzt man die Idee des dynamischen Programmierens. Einzelne Berechnungen der Matrix-Vektor-Multiplikation werden in einer

bestimmten Reihenfolge ausgeführt und Zwischenergebnisse gespeichert, die in folgenden Berechnungen genutzt werden. Das funktioniert aber nur bei einer Anzahl von Daten, die einer 2er-Potenz entsprechen.

Anwendungsbeispiel

Bei diesem Beispiel ist ein Bild gegeben, dass mit Streifen überlagert wurde. Die Streifen so zu entfernen ist relativ schwierig. Nach der Fouriertransformation sieht man eindeutig, wo die Streifen liegen. Wenn man nun diese Punkte entfernt und zurück transformiert, erhält man das Originalbild ohne überlagerte Streifen. An diesem Beispiel sieht man, dass sehr viel einfacher ist, eine Fouriertransformation zu machen und auf dem Ergebnis eine Operation auszuführen, als auf dem Originalbild eine entsprechende Operation auszuführen.

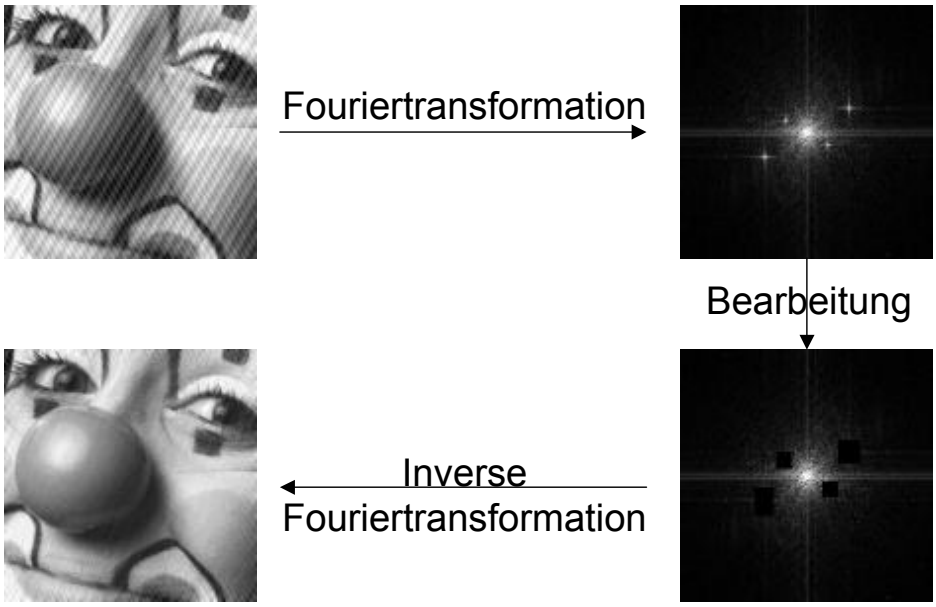


Abbildung 2: Beispiel für Anwendung der Fast Fourier Transformation

DCT – Diskrete Cosinus Transformation

Die Diskrete Cosinus Transformation findet in der Praxis sehr häufig Anwendung, z.B.: bei JPEG und MPEG.
Die DCT baut auf der Fourier-Transformation auf, benutzt aber nur die Kosinusanteile, rechnet daher nur mit reellen und braucht auch daher nur halb so viele Daten.
Die diskrete Cosinus Transformation hat unter anderem folgende Eigenschaften: sie ist verlustfrei umkehrbar.
sie ist separierbar, d.h. der mehrdimensionale Fall kann auf den 1-dimensionalen Fall zurückgeführt werden [UniSB-www].
Bei der DCT werden ähnlich wie bei der Fourier-Transformation die Ausgangswerte vom Zeit- in den Frequenzbereich umgewandelt. Es besteht nur ein wesentlicher Unterschied: Die DCT wandelt einen zweidimensionalen Bildbereich in einen zweidimensionalen Frequenzbereich um. Für die Größe der Bildbereiche gibt es bei dieser Transformation eigentlich keine Beschränkung, so könnte man zum Beispiel einen Bereich von 16x16 oder

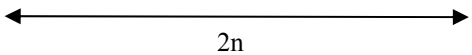
16x8 Werten umwandeln. Im JPEG-Standard werden aber immer Blöcke der Größe 8x8 transformiert. Das Ausgangsbild wird dabei in viele einzelne Blöcke unterteilt.
Die Zweidimensionale DCT läßt sich kombinieren aus der Durchführung der eindimensionalen DCT über die Zeilen und über die Spalten des Blocks.
Nach der Berechnung der DCT erhält man einen Bereich von 8x8 Frequenzen. Dabei steht eine niedrigste Frequenz 0, man spricht hier auch vom DC-Wert. Die anderen Felder enthalten die Amplituden der höheren Frequenzen, diese Werte nennt man auch die AC-Werte. Diese Koeffizienten repräsentieren mit steigenden Abstand zum DC-Wert höhere Frequenzen, wobei die höheren vertikalen Frequenzen durch höhere Zeilenindizes repräsentiert werden und die höheren Horizontalfrequenzen durch größere Spaltenindizes.
Die Transformation der Pixelintensitäten durch die DCT führt nicht zu einem Informationsverlust, abgesehen von den Rundungsfehlern und dem diskreten Charakter der Operation.
Die Rücktransformation mit Hilfe der inversen DCT ergibt somit wieder den Originalblock. Mit Hilfe der IDCT werden die Frequenzwerte wieder in Bildwerte umgewandelt.

Idee

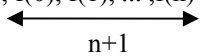
Definition: Eine Funktion f heißt genau dann gerade Funktion, wenn mit $x \in D_f$ stets auch gilt $-x \in D_f$ und wenn gilt $f(-x)=f(x)$, eine Funktion f heißt genau dann ungerade Funktion, wenn mit $x \in D_f$ stets auch gilt $-x \in D_f$ und wenn gilt $f(-x)=-f(x)$.

Zur Berechnung der n-ten Einheitswurzel wurden Sinus- und Kosinusfunktionen benutzt. Die Kosinusfunktion ist eine gerade Funktion, die Sinusfunktion ist eine ungerade Funktion.
Der imaginäre Anteil wird nur durch die Sinusfunktion berechnet. Es wäre ein Vorteil, wenn man diese Werte eliminieren könnte, so dass man nur noch Werte aus den reellen Zahlen hat.
Dies erreicht man in dem man die gegebenen Werte verdoppelt, wobei die neu hinzugekommenen Werte auf
linken Seite der Achse stehen [Ekreide2-www].

Gerade Funktion durch Verdoppelung der Werte



f(-n+1), f(-n+2), ... f(-1), f(0), f(1), ... ,f(n)



Dadurch heben sich die Sinuswerte auf und man behält nur noch Werte, die in den reellen Zahlen liegen.

Es wäre auch möglich das ganze analog für den Sinus zu machen, so dass die Kosinusanteile wegfallen. Es wurde aber festgestellt, dass die DCT vom menschlichen Auge als angenehmer betrachtet wird [Ekreide2-www 65. min].

Herleitung der Formel

Wie gerade beschrieben, wurden die gegebenen Daten verdoppelt. Daher geht die Summe der Formel für die Fourier-Transformation von –n+1 bis n.

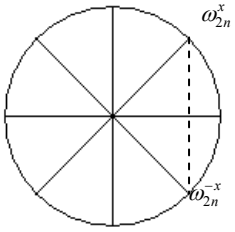
$$F_k = \frac{1}{2n}\left(\sum_{i=-n+1}^{n} f(i)\omega_{2n}^{-ik}\right)$$

$$F_k = \frac{1}{2n}\left(f(0) + \sum_{i=1}^{n-1}\left(f(i)\omega_{2n}^{ik} + f(-i)\omega_{2n}^{-ik}\right) + f(n) * \left(\cos\frac{2\Pi nk}{2n} + i\sin\frac{2\Pi n}{2n}\right)\right)$$

Wavelets

Eine weitere Transformation sind die Wavelets. Es handelt sich dabei um ein relativ neues Verfahren, welches Ende der 80er Jahre entwickelt wurde.
Die Idee ist, dass Funktionen auch durch die Summe von anderen Funktionen (Basisfunktionen) dargestellt werden können.

$$F_k = \frac{1}{2n}\left(\begin{array}{l} f(0) + \sum_{i=1}^{n-1} f(i)(\omega_{2n}^{ik} + \omega_{2n}^{-ik}) \\ + f(n) * (\cos \Pi k) \end{array} \right)$$



Die Transformation wird schrittweise vollzogen, d.h. man versucht die darzustellende

$$F_k = \frac{1}{2n}\left(\begin{array}{l} f(0) + 2\sum_{i=1}^{n-1} f(i)\cos\frac{\Pi ik}{n} \\ + f(n) * (\cos \Pi k) \end{array} \right)$$

Funktion in immer "glattere" Versionen aufzuspalten, die mit dem Grad der Iteration immer weniger Informationen enthalten. Dieses Verfahren nennt man auch Multi-Skalen-Analyse (englisch: Multiresolution Analysis) [UniKL-www].

Basisfunktionen

Als Basisfunktion kann jede orthogonale Funktion h genommen werden, für die gilt

$$\int_{-\infty}^{\infty} h(t)dt = 0$$

[Ekreide3-www]: Orthogonalität bedeutet die Vektoren der Funktionen stehen senkrecht zueinander, wie zum Beispiel bei Kosinus und Sinus ($\sin^2 x + \cos^2 x = 1$ auf dem Einheitskreis).
Von dieser Formel lässt sich auch der Name Wavelet ableiten, denn die Funktionswerten müssen sich sowohl oberhalb wie auch unterhalb der X-Achse befinden und erinnern daher mit etwas Phantasie an eine Welle (engl. wave).
Wie man sieht gibt es also nicht nur *eine* Wavelet-Transformation, sondern vielmehr wird mit diesem Begriff eine ganze Klasse von Transformationen beschrieben.
Einige für die Bildverarbeitung häufig genutzte Basisfunktionen werden hier vorgestellt:

Das Haar-Wavelet

Das Haar-Wavelet wird für die Erklärungen der Wavelet-Transformation sehr gern benutzt, da die Funktion recht einfach ist. Man erkennt in Abbildung 3 sehr schön, dass das Integral über diese Funktion gleich null ist. Auch der „Wellencharakter“ ist gut zu sehen [UniKL-www].

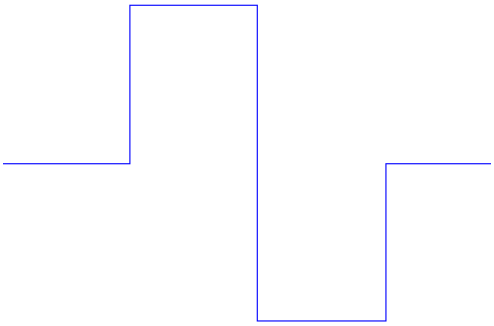


Abbildung 3: Haar-Wavelet

Daubechies-Wavelets

Die Klasse der Daubechies-Wavelets wurden nach Ingrid Daubechies benannt. Sie gilt als Mutter moderner Wavelets. Zusammen mit Stephane Mallat von der New York University entwickelte sie an den Bell-Laboratorien die Theorie weiter und stellte Zusammenhänge zur Signalverarbeitung her.
In Abbildung 4 und 5 sind zwei Vertreter dieser Klasse von Wavelets zu sehen. Auch bei diesen Wavelets kann man erahnen, dass das Integral gleich null ist [UniKL-www].

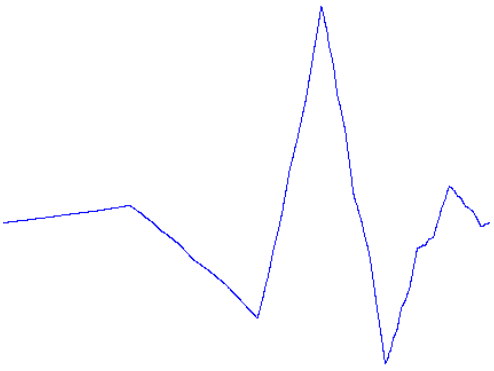


Abbildung 4: Daubechies 6-Wavelet

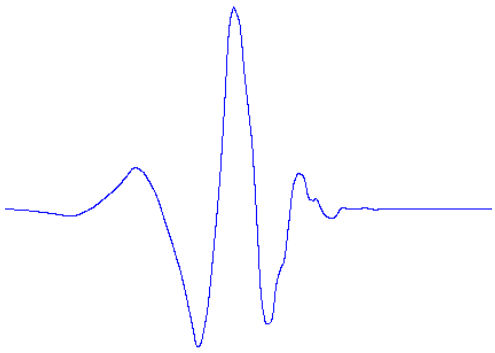


Abbildung 5: Daubechies 8-Wavelet

Grundprinzip

Die Grundidee ist die Berechnung des Mittelwertes und der Differenz zwischen zwei benachbarten Pixelwerten. Die Ergebnisse werden als Tiefpass und Hochpassanteile gespeichert. Der Tiefpassanteil wird mit den Haar-Funktionen weiter analysiert. Letztendlich ist das Wavelet-transformierte Bild durch

eine gewisse Anzahl immer kleiner werdender Hochpassanteile und einen einzigen Tiefpassanteil gekennzeichnet.

Ein-dimensionale Berechnung

In der Abbildung 6 sieht man diese Grundidee veranschaulich. Es werden links die Mittelwerte hingeschrieben und rechts die Differenzen. Danach werden rekursiv wieder die Mittelwerte und Differenzen berechnet, bis nur noch ein Tiefpassanteil vorhanden ist.

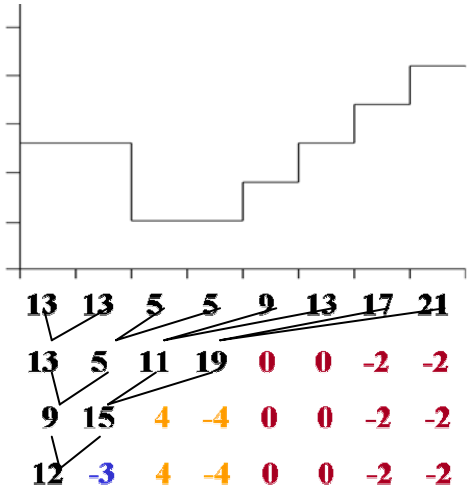
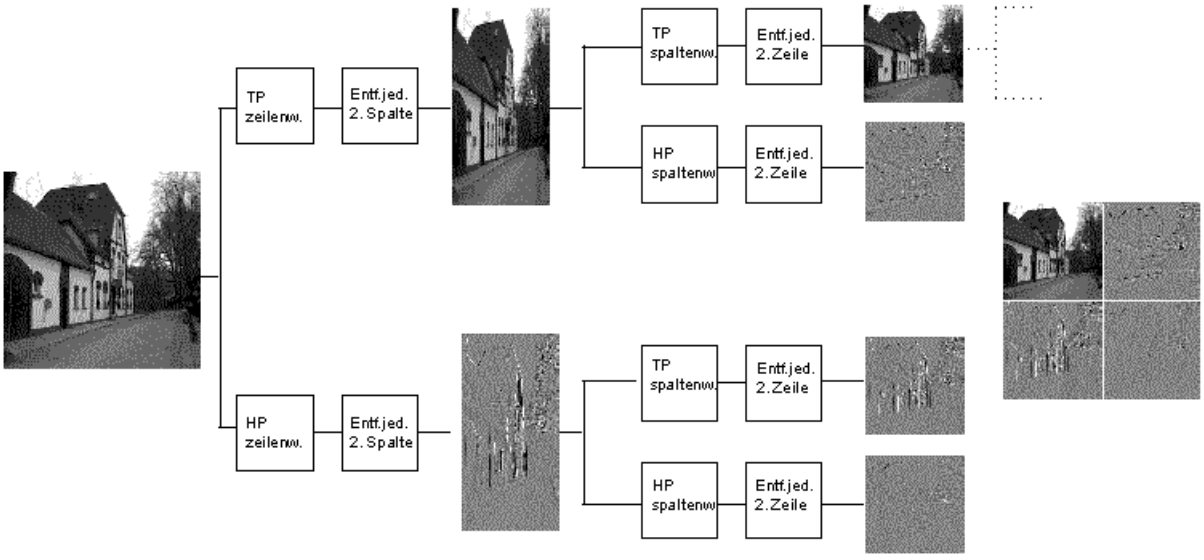


Abbildung 6: Beispiel Berechnung der Mittelwerte und Differenzen

Zwei-dimensionale Berechnung

Wie man in Abbildung 7 und 8 sieht, resultieren zwei Bilder, man hat also nun mehr Daten als vorher, die jedoch redundant sind. Man kann diese Bilder dann um den Faktor 2 verkleinern ohne Information zu verlieren. Der beschriebene Vorgang geschieht sowohl in horizontaler als auch in vertikaler Richtung. Die nachfolgenden Abbildungen zeigen Bilder vor und nach der Transformation.
In den Hochpassanteilen der ersten Transformationsstufe werden die feinen Bildstrukturen erfasst,
in den Hochpassanteilen der folgenden Transformationsstufen werden zunehmend gröbere Bildstrukturen erfasst [FhJena-www].



Abbildungen 7 und 8: Zwei-dimensionales Beispiel für Wavelet-Transformation

Die Kompression für Abbildung 9 ist jetzt wesentlich einfacher, da große Teile schwarz sind.

Abbildung 9: ein weiteres zwei-dimensionales Beispiel für Wavelet-Transformation

Vergleich zur DCT

Abbildung 10: Original Lena [TUChemnitz-www]

Abbildung 11: Kompressionsfaktor 1:25 links Wavelet, rechts DCT

Abbildung 12: Kompressionsfaktor 1:50 links Wavelet, rechts DCT

Zusammenfassung

In dieser Seminararbeit wurden drei wichtige Transformationen vorgestellt. Der Schwerpunkt wurde dabei auf die Anwendung für Bild- bzw. Videokompression gelegt.
Wie man gesehen hat, bringen die Transformationen Vorteile für spätere Kompressionsalgorithmen. Es ist aber auch möglich, die Daten wieder in den Originalzustand zurückzuführen, wobei allerdings Rundungsfehler auftreten können.
Die Idee der Fouriertransformation gibt es schon sehr lange, sie ist Grundlage für die zwei weiter vorgestellten Transformationen Diskrete Kosinus Transformation und Wavelet Transformation.
Für die Fouriertransformation gibt es auch eine schnelle Variante, die sogenannte Fast Fouriertransformation FFT. Diese nutzt das verfahren des dynamischen Programmierens und berechnet Teile von Matrizen in bestimmter Reihenfolge und speichert die Zwischenergebnisse.
Die Diskrete Kosinustransformation baut direkt auf der Fouriertransformation auf. Es werden die gegebenen Werte verdoppelt, so dass sich die Sinusanteile aufheben. So hat man nur noch die Kosinusanteile übrig und braucht nur noch mit reellen Werten rechnen.
Die Diskrete Kosinustransformation findet in der Bild- und Videobearbeitung, aber auch in der Audiobearbeitung sehr häufig Anwendung, so wird sie z.B. bei JPEG und MPEG genutzt.
Die Wavelet Transformation nutzt auch die Idee: die gegebenen Werte durch Basisfunktionen darzustellen, allerdings nicht wie bei der Fouriertransformation mit Sinus und Kosinus, oder bei der Diskreten Kosinus Transformation nur mit Kosinus, sondern mit verschiedenen Waveletklassen. Vorgestellt wurden die Haar- und die Daubechies-Wavelets. Die Bedingung für die Basisfunktionen ist die Orthogonalität und dass das Integral gleich Null sein muss. Durch die zweite Bedingung lässt sich auch der Name Wavelet herleiten, den die Funktion muss mit etwas Phantasie eine Wellenform haben. Es werden Mittelwerte und Differenzen zwischen einzelnen Pixel berechnet, wobei dann das Verfahren rekursiv mit den Mittelwerten wiederholt wird.
Wie man beim Vergleich der Wavelet Transformation mit der Diskreten Kosinustransformation sieht, sehen Bilder, die mit Wavelets transformiert worden sind und dann komprimiert häufig besser aus als Bilder, die mit der Diskreten Kosinustransformation transformiert worden sind.

Abkürzungen

FFT	Fast Fourier Transformation
DCT	Diskrete Cosinus Transformation

Links

[Ekreide1-www]	http://kazan.inf.fu-berlin.de/echalklectures/SS02/BV/fourier/
[Ekreide2-www]	http://kazan.inf.fu-berlin.de/echalklectures/SS02/BV/kosinus/
[Ekreide3-www]	http://kazan.inf.fu-berlin.de/echalklectures/SS02/BV/wavelets/
[FhFlensburg-www]	http://www.iti.fh-flensburg.de/lang/algorithmen/fft/fft.htm
[FhJena-www]	http://www.fh-jena.de/contrib/fb/et/personal/ansorg/ftp/wavelet/wavelet.htm
[FHWorms-www]	http://www.ztt.fh-worms.de/de/sem/ws95_96/kompressionsalgorithmen/node34.html#DCT

[Mannheim-www] http://www-mm.informatik.uni-mannheim.de/veranstaltungen/animation/multimedia/1d_dct_und_dft/documentation/Transformationen_Studienarbeit.pdf
[TUChemnitz-www] http://archiv.tu-chemnitz.de/pub/1998/0010/data/vortrag/dth/pictures.html
[UniKL-www] http://nt.eit.uni-kl.de/wavelet/einleitung.html
[UniSB-www] http://fsinfo.cs.uni-sb.de/~lynx/uni/fopra/node19.html

Postscript
&
Portable Document Format

von

Oliver Tenchio

Einleitung

PostScript und das Portable Document Format wurden bei Adobe entwickelt. Adobe wurde 1982 von Dr. John E. Warnock und Dr. Charles M. Geschke gegründet. Beide waren vorher bei Xerox beschäftigt und dort hauptsächlich an der Entwicklung von Interpress beteiligt. Interpress war eine Xerox-Implementierung von Konzepten, die bei Evans & Sutherland als Teil ihres „Design System“ entwickelt wurden. Da man bei Xerox Interpress nur für die eigenen Drucker entwickelte, und Warnock und Geschke aber gern ein eigenständiges Produkt wollten, verließen sie die Firma und gründeten Adobe[2]. Als erstes Produkt wurde 1984 PostScript auf den Markt gebracht. 1992 erschien die erste Version vom Portable Document Format, das sich durch eine geschickte Marketingstrategie Adobes zu einem Standard bei der systemübergreifenden Verbreitung von Dokumenten etablierte.

PostScript

Einführung

PostScript wird häufig als Seitenbeschreibungssprache bezeichnet, ist aber auch eine vollwertige, stackbasierte Programmiersprache ähnlich Forth. Es werden wichtige Punkte einer modernen Programmiersprache abgedeckt. So werden Datentypen unterschieden und der Programmfluss kann durch Schleifen und Bedingungen beeinflusst werden. PostScript stellt häufig die Verbindung zwischen einer Anwendungssoftware, wie Textverarbeitung oder DTP-Programmen und Ausgabegeräten, wie zum Beispiel Druckern oder Belichtern her.

Geschichte

Als erstes Produkt von Adobe hätte vielleicht niemand PostScript level 1 beachtet, wenn nicht Steve Jobs von Apple Warnock beauftragte hätte, einen PostScript-Kontroller für Apples LaserWriter zu entwickeln. Apple investierte 1985 dafür 2,5 Millionen Dollar in Adobe. Im Zusammenhang mit dem Programm PageMaker von Aldus konnte man erstmals ein erschwingliches System kaufen, mit dem man qualitativ hochwertige Dokumente erstellen und zu Papier bringen konnte. Die Kombination von LaserWriter, PostScript und PageMaker rettet damals Apple vor zurückgehenden Gewinnen bei ihren Computern und PostScript etablierte sich.

Mit den Jahren wurden immer wieder Erweiterungen zu PostScript hinzugefügt, die die Fähigkeiten an die steigenden Anforderungen anpassten. So kam 1991 PostScript level 2 auf den Markt, das sich aber lange Zeit nicht durchsetzen konnte. 1998 kam PostScript 3.

Interpreter

Ein PostScript-Programm wird von einem Interpreter verarbeitet. Dieser kann in einer Software zur Darstellung auf einem Bildschirm oder aber auch als Modul für Drucker oder ähnlichem vorhanden sein. Die Interpreter halten den Zustand der Programmabarbeitung in 5 Stacks fest. Dies sind ein Operanden-, Dictionary-, Execution-, Grafik- und der Clippingpathstack.

Ein Interpreter arbeitet die Programmdatei sequentiell ab. Dabei entscheidet die Art des gelesenen Objektes darüber, was geschieht. So wird ein Operandenobjekt einfach auf den Operandenstack geschoben. Bei einem Namensobjekt werden alle Dictionaries im

[2] Der Name Adobe stammt von einem kleinen Fluss hinter Warnocks Haus in Los Altos, Kalifornien.

Dictionarystack nach diesem Namen durchsucht, und die dazugehörende Prozedur ausgeführt. PostScript interne Operatoren befinden sich im Systemdict, das auch auf dem Dictionarystack liegt. Eigene Prozeduren werden im Userdict angelegt. Im Grafikstack werden die Eigenschaften, mit denen eine Linie, eine Schraffur oder Text ausgegeben wird gespeichert. Die eigentlichen Veränderungen, die beim Zeichnen stattfinden, werden in einem Seitenobjekt festgehalten, das auch auf dem Grafikstack liegt. Dieses Seitenobjekt ist am Anfang leer. Nach dem Aufruf von `showpage` wird diese Seite auf dem Ausgabemedium ausgegeben und wieder gelöscht. Der Clippingpathstack enthält Pfade, mit denen die Ausgabe auf das Seitenobjekt eingeschränkt wird. So befindet sich in der Regel ein Rechteck auf diesem Stack, das die nutzbare Fläche der Seite, also die Seitenränder, bestimmt. Es können aber beliebige Formen für diese Pfade bestimmt werden. Siehe dazu das Beispiel 0.

Es gibt verschiedene Operandentypen. Operanden sind zum Beispiel:

- Numerische Werte

```
1
-3
-.03
16#FFFF
```

- „literal name objects“ (der Schrägstrich gehört nicht zum eigentlichen Namen)

```
/box
/Times-Roman
```

- Zeichenketten

```
(Das ist ein String)
(oder dies)
```

- aber auch Prozedurrümpfe

```
{ 28.346 mul }
```

Dateiformat und Beispiele

Es gibt 3 PostScript-Dateiformate. Ein ASCII-Format und zwei Binärformate. Hier wird nur auf das Klartextformat eingegangen.

Hallo, Welt!

Als erstes ein kurzes Programm, das den Text „Hallo, Welt!“ in der unteren linken Ecke ausgibt.

```
%!

/Times-Roman findfont
40 scalefont
setfont

newpath
72 72 moveto
(Hallo, Welt!) show

showpage
```

Abbildung 1: Eine einfache PS-Datei

Eine PostScript-Datei beginnt in der ersten Zeile immer mit „`%!`“. Eventuell folgt noch die Versionsnummer, des verwendeten PostScripts.
In der 3. Zeile wird das Literal „`Times-Roman`“ auf den Operandenstack geschoben. Zum folgenden Namensobjekt „`findfont`“ wird die entsprechende Prozedur ausgeführt. Diese holt sich den Namen des zu suchenden Zeichensatzes vom Operandenstack und schiebt das passende Zeichensatzobjekt auf den Stack. Dieses wird in Zeile 4 auf 40 Punkte skaliert und in Zeile 5 wird dann das entstandene Objekt als aktueller Zeichensatz festgelegt.
Um den eigentlichen Text auszugeben, wird die aktuelle Schreibposition mit „`moveto`“ um 1 Zoll nach rechts und 1 Zoll nach oben verschoben. Anschließend kommt die Zeichenkette auf den Stack. Diese wird dann von „`show`“ auf das Seitenobjekt geschrieben. Mit „`showpage`“ wird dann die gesamte Seite ausgegeben.

Prozeduren, Grafik

Hier nun ein kurzer Ausschnitt eines PostScript-Programms, um die Definition von eigenen Prozeduren zu veranschaulichen. Das komplette Programm befindet sich im Anhang.

```
/cm { 28.346 mul } def

/box {
  newpath
  moveto
  2 cm 0 cm rlineto
  0 cm 2 cm rlineto
  -2 cm 0 cm rlineto
  closepath
} def
```

Abbildung 2: Prozedurdefinition

In der ersten Zeile wird das Literal „cm“ auf den Stack geschoben. Danach folgt ein Prozedurrumpf, welcher auch auf den Stack kommt. Der Operator „def“ holt sich dann die beiden obersten Objekte vom Stack, und trägt den Prozedurrumpf unter dem Namen „cm“ in das Userdict ein. Diese Prozedur dient dazu, einen cm-Wert in Punkte umzurechnen. Ein Punkt entspricht 1/72 Zoll und 1cm entspricht 28,346 Punkten. Diese Umrechnung wird in der zweiten Prozedur benutzt, um ein 2cm x 2cm großes Quadrat zu konstruieren. Wichtig ist hierbei zu beachten, dass in der zweiten Zeile durch „moveto“ die obersten zwei Elemente vom Stack geholt werden, damit dann an dieser Position das Quadrat erstellt werden kann. Diese zwei Elemente bilden also die Parameter für diese Prozedur. „box“ stellt aber nur einen Pfad zur Verfügung. Dieser muss dann zum Beispiel mit „stroke“ gezeichnet werden.

```
10 cm 10 cm box stroke
```

Erzeugt ein 2cm x 2cm Quadrat 10cm vom linken und 10cm vom unteren Seitenrand entfernt.

Clipping

Nun noch ein einfaches Beispiel für das Clipping.

```
%!

/Times-Roman findfont
300 scalefont setfont

newpath
100 100 moveto
(FU) false charpath clip

100 100 translate
0 1 360 {
  newpath
  gsave
    rotate
    0 0 moveto
    600 0 rlineto
    stroke
  grestore
} for

showpage
```

Abbildung 3: Clipping

Zuerst wird ein sehr großer Zeichensatz festgelegt. Der Text „FU“ wird dann mit dieser Schrift zu einem Pfad gemacht, der dann als Clippingpfad festgelegt wird. In der for-Schleife werden dann im Winkel von jeweils 1° lange Linien gezeichnet, aber eben nur innerhalb des Clippingpfades. Die Rotation wird dabei von „rotate“ übernommen, wobei das gesamte Koordinatensystem gedreht wird. Die Linie selbst wird dann einfach „waagerecht“ ausgegeben. Das Ergebnis ist im Anhang zu sehen.

Portable Document Format

Einführung

PDF ist im Gegensatz zu PostScript nicht als Seitenbeschreibungssprache, sondern als Dokumentenbeschreibungssprache gedacht. Aus der Sicht von PDF besteht ein Dokument aus einer Menge von verlinkten Objekten. Diese werden in einem Dokumentenkatalog zusammengefasst. Die Seiten werden über einen so genannten Seitenbaum aufgebaut. Die Struktur des Dokumentes, wie Kapitel und Unterkapitel, werden in der Outline-Hierarchie beschrieben. Diese wird häufig bei der Bildschirmdarstellung als Navigationshilfe angezeigt.

Da im Gegensatz zu PostScript das ganze Dokument und nicht nur eine Seite betrachtet wird, kann PDF zum Beispiel auch interne Links unterstützen. Des Weiteren wurde in PDF eine Art Versionsverwaltung integriert. Teile des Dokumentes können aktualisiert werden, ohne dass die alten Versionen gelöscht werden müssen. Durch Verschlüsselung und andere Mechanismen kann ein Autor auch Teile eines Dokumentes vor dem Ausdrucken schützen. Obwohl es mittlerweile auch Druckermodule gibt, die neben PostScript auch direkt PDF unterstützen, so dient dieses Format doch eigentlich eher dem Vermeiden von zu viel Papier, indem des eine Basis für eine Systemübergreifende Verfügbarkeit von Dokumenten aller Art bereitstellt.

Obwohl bei PDF das gleiche Abbildungsformat wie bei PostScript und auch sehr ähnliche Verfahren zur Darstellung verwendet werden, eignet es sich weniger zum Experimentieren an der Quelldatei. Dies liegt an dem von PDF vor geschriebenen Dateiformat, das an vielen Stellen Byteoffsets verlangt, die von Hand eben nicht so einfach zu bestimmen sind.

Geschichte

Das Portable Document Format wurde von Adobe erstmals 1992 auf der Comdex vorgestellt. Im September 1994 folgte PDF1.1, 1996 PDF1.2 und im April 1999 PDF1.3. Diese Versionen wurden immer mit einem hauseigenen Programm, dem Adobe Acrobat, auf den Markt gebracht. Dies sollte die schnelle Verbreitung und Etablierung sichern. Aber erst als Adobe die abgespeckte Version des Acrobat, den Acrobat Reader, frei zur Verfügung stellte[3] avancierte PDF schnell zum Quasistandard bei der Verbreitung von Dokumenten. Die letzte Version, PDF1.4, wurde erstmals nicht mit einem unterstützenden Programm vorgestellt, was dazu führte, dass sich diese Version nur sehr schleppend durchsetzen konnte.

Dateiformat und Beispiele

Eine PDF-Datei beginnt immer mit „`%PDF-x.y`“ im Header. Dem Kopf der Datei folgt der Hauptteil, dieser enthält einen Dokumentkatalog, den Seitenbaum und die Outline-Hierarchie. Am Ende stehen ein Cross-reference-table und ein Trailer. Diese enthalten auch die Byteoffsets, die die größte Hürde bei der Bearbeitung von Hand darstellen.

Der Dokumentkatalog, der Seitenbaum und die Outline-Hierarchie werden durch Objekte aufgebaut. Ein solches Objekt wird von einer Referenznummer und dem Schlüsselwort „`obj`“ eingeleitet. Innerhalb des Objektes werden Attribute gesetzt, und dann wird die Definition mit „`endobj`“ abgeschlossen. Als Beispiel hier ein Katalogobjekt aus einer einfachen PDF-Datei.

[3] Anfänglich kostete der Acrobat Reader 50$.

```
1 0 obj
<<
/Type /Catalog
/Pages 3 0 R
/Outlines 2 0 R
>>
endobj
```

Die erste Zeile enthält die Referenznummer des Objektes, wobei die zweite Zahl die Revisionsnummer darstellt. In der dritten Zeile wird festgelegt, dass es sich um einen Katalog handelt. Dieser enthält zwei Attribute, die jeweils durch eine Referenz definiert werden. Alle weiteren Objekte werden ähnlich aufgebaut, sie enthalten aber natürlich unterschiedliche Attribute.

Im Anhang befindet sich die vollständige Datei, die die selbe Ausgabe liefert, wie das „Hallo, Welt!“-Beispiel in PostScript (nur in englisch). Dort sind auch der cross-reference-table und der Trailer zu sehen.

Zusammenfassung

Zusammenfassend kann gesagt werden, dass es sich lohnt mal ein Bisschen mit PostScript zu experimentieren, da man auch rekursive Darstellungen leicht erzeugen kann. PDF ist aus dieser Sicht wegen der erwähnten Eigenschaften nicht so interessant, und man wird PDF-Dateien sicherlich auch in Zukunft eher durch ein entsprechendes Programm erzeugen.

Abkürzungen

Adobe	Adobe Systems Incorporated; San Jose, California, USA
DTP	Desktop Publishing
FU	Freie Universität Berlin
PDF	Portable Document Format
PS	PostScript

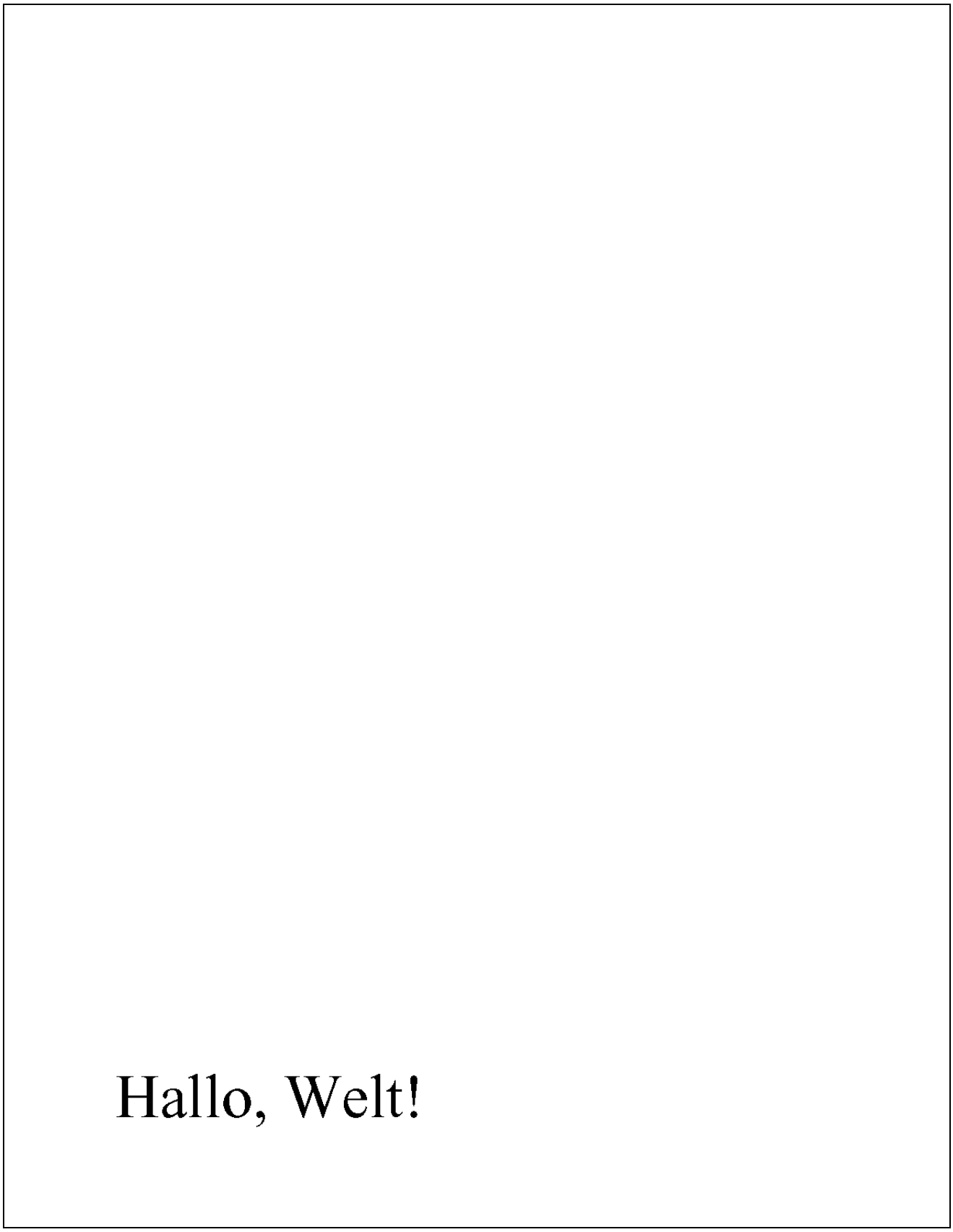
Literatur

[PDFR01] PDF Reference: Adobe portable document format version 1.4 / Adobe Systems Incorporated, third edition; 2001; Addison-Wesley Publishing Company

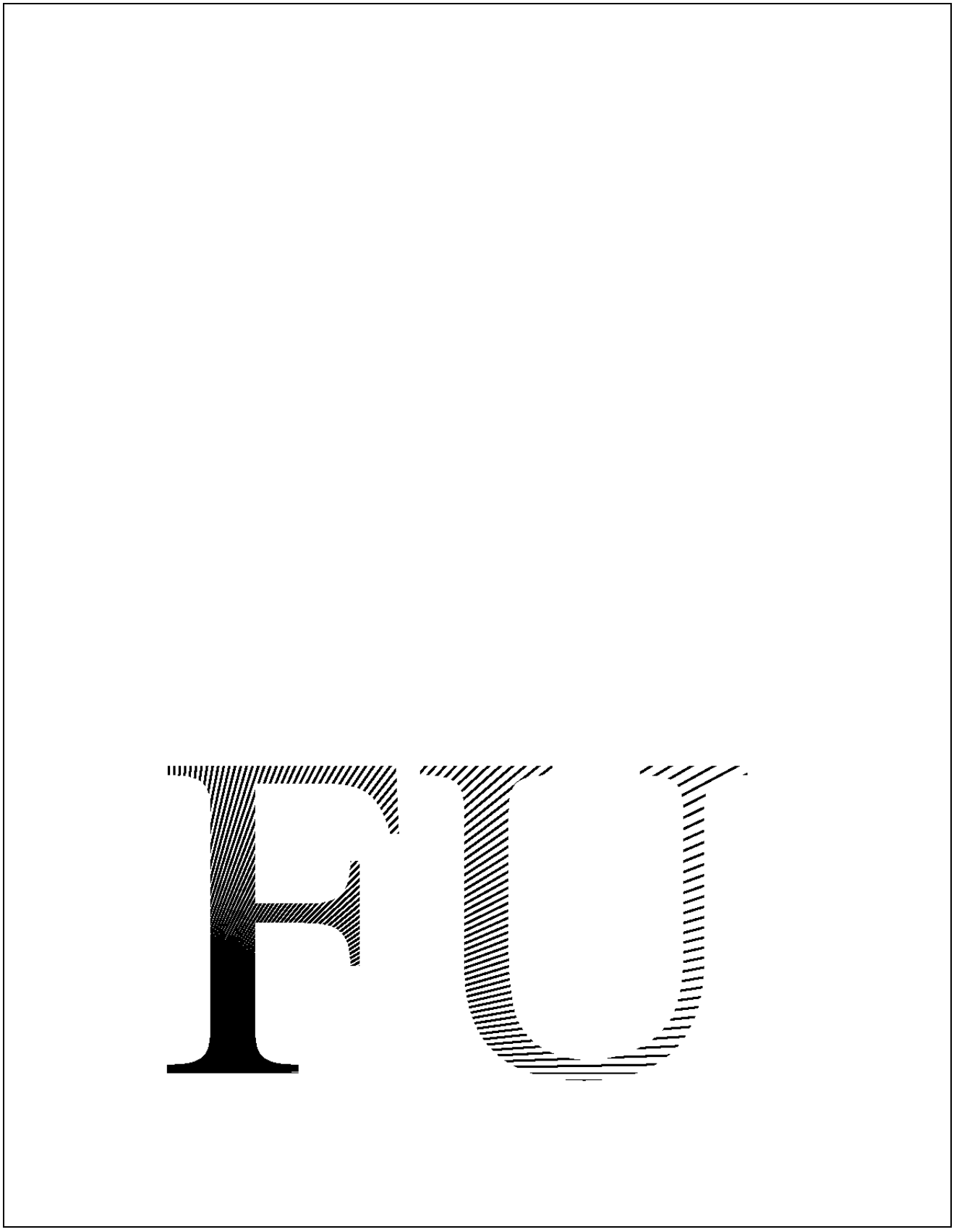
[PDFR99] PostScript language reference manual / Adobe Systems Incorporated, third edition; 1999; Addison-Wesley Publishing Company

Anhang

Ausgabe: PS - Hallo, Welt!



Ausgabe: PS - Clipping



Listing: PDF - Hello World

```
%PDF-1.0
1 0 obj
<<
/Type /Catalog
/Pages 3 0 R
/Outlines 2 0 R
>>
endobj

2 0 obj
<<
/Type /Outlines
/Count 0
>>
endobj

3 0 obj
<<
/Type /Pages
/Count 2
/Kids [4 0 R
       10 0 R]
>>
endobj

4 0 obj
<<
/Type /Page
/Parent 3 0 R
/Resources << /Font << /F1 7 0 R >> /ProcSet 6 0 R >>
/MediaBox [ 0 0 612 792 ]
/Contents 5 0 R
>>
endobj

10 0 obj
<<
/Type /Page
/Parent 3 0 R
/Resources << /Font << /F1 7 0 R >> /ProcSet 6 0 R >>
/MediaBox [ 0 0 612 792 ]
/Contents 5 0 R
>>
endobj

5 0 obj
<< /Length 44 >>
stream
```

```
BT
/F1 24 Tf
100 100 Td (Hello World!) Tj
ET
endstream
endobj

6 0 obj
[/PDF /Text]
endobj

7 0 obj
<<
/Type /Font
/Subtype /Type1
/Name /F1
/BaseFont /Helvetica
/Encoding /MacRomanEncoding
>>
endobj

xref
0 8
0000000000 65535 f
0000000009 00000 n
0000000034 00000 n
0000000120 00000 n
0000000179 00000 n
0000000322 00000 n
0000000415 00000 n
0000000445 00000 n

trailer
<<
/Size 8
/Root 1 0 R
>>
startxref
553
%%EOF
```

SVG
Scalable Vector Graphics

von

Thomas Reimann

Einleitung

SVG steht für Scalable Vektor Graphics. Es ist ein Grafikformat in XML. Dadurch ist es möglich, grafische Informationen in einer kompakten und portablen Form zu speichern. SVG ist aus PGML und VML entstanden. Im Oktober 1998 gab es den ersten Entwurf von SVG. SVG 1.0 wurde aber erst im September 2001 verabschiedet.

Grafiksysteme

Um Grafiken auf dem Computer darzustellen, verwendet man zwei unterschiedliche Grafiksysteme. Zum einem das Rastergrafiksystem und zum anderen das Vektorgrafiksystem.

Rastergrafiksystem

Bilder werden im Raster Grafik System durch *picture elements* oder kurz *pixel* dargestellt. Jedes *pixel* repräsentiert entweder eine Farbe im RGB Farbraum oder ist ein Index einer Farbpalette. Alle Pixel zusammen werden auch Bitmap genannt. Solch eine Bitmap wird oft in einem komprimierten Format gespeichert. Da die meisten Anzeigegeräte das Rastergrafiksystem (z.B. CRT Monitore) verwenden, ist es recht einfach, Rastergrafikbilder darzustellen. Rastergrafikbilder werden meist für Fotographien und gescannte Bilder (Fax, Kopierer) genutzt. Die gebräuchlichsten Rastergrafikformate sind JPEG, GIF und PNG.

Vektorgrafiksystem

Vektorgrafikbilder bestehen aus Aneinanderreihungen von geometrischen Formen. Diese Formen müssen von einem Programm in Rastergrafik konvertiert werden (scan conversion), wenn das entsprechende Bild auf einem Rastergrafik Anzeigegerät angezeigt werden soll. Es gibt allerdings auch Vektorgrafik Anzeigegeräte (z.B. Oszillograph), die diese Formen zwar in einem eigenen Format aber sonst direkt anzeigen können. Allerdings sind bei solchen Geräten die Bildwiederholraten abhängig von der Anzahl der darzustellenden Formen. Außerdem ist die Steuerungslogik teils sehr aufwendig und daher recht teuer. Deshalb werden heutzutage meist nur noch Rastergrafiksysteme verwendet. Weiterführende Informationen findet man in [FOL91]. Vektorgrafiken werden meist für CAD Programme benutzt. Aber auch PostScript und Flash sind Vektor orientiert.

Skalierbarkeit

Obwohl Vektorgrafiken nicht so verbreitet sind, wie Rastergrafiken, besitzen diese jedoch einen großen Vorteil gegenüber Rastergrafiken. Vektorgrafiken können ohne Verlust der Bildqualität beliebig skaliert werden. Dazu ein kleines Beispiel:

Abbildung 1: Original Bild (100%)

Abbildung 2: Raster Grafik (200%)

Abbildung 3: Vektor Grafik (200%)

Grundaufbau eines SVG

Eine SVG Datei beginnt grundsätzlich mit

```
<?xml version="1.0"?>
```

gefolgt von

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
      "http://www.w3c.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd>
```

Als nächstes kommt das <*svg*> Element(root tag) des Dokumentes

```
<svg width="140" height="170">
```

Optional können dann Titel und Beschreibung des Dokumentes angegeben werden

```
<title>Beispiel</title>
<desc>Beischreibung der SVG Grafik</desc>
```

Hier würden die eigentlichen Daten folgen

```
<!-- Zeichnung -->
```

Zum Schluss muss das <*svg*> Element

```
</svg>
```

wieder geschlossen werden.

Koordinaten

Eine SVG Grafik ist erstmal unendlich groß. Da man aber nur einen bestimmten Bereich der Grafik anzeigen möchte, muss man einen Viewport festlegen.

Viewport

Die Größe des Viewports legt man mit *width* und *height* im <*svg*> Element fest. Die Werte der beiden Attribute können einfach als Zahl angegeben werden. Dabei wird dann diese Zahl mit der Einheit Pixel interpretiert. Man kann allerdings auch Einheiten angeben:

- em Größe des Default Font
- ex Höhe des Buchstaben x
- px Pixel
- pt Point (1/72 Inch)
- pc Picas (1/6 Inch)
- cm Zentimeter
- mm Millimeter
- in Inch

Der Ursprung des Koordinatensystems liegt in der oberen linken Ecke. Wenn man eine andere Einheit anstelle einer vordefinierten benötigt, dann hat man die Möglichkeit mit dem Attribut *viewBox* sich eine Einheit selbst zu definieren. Dazu ein kleines Beispiel:

```
<svg width="4cm" height="5cm" viewBox="0 0 64 80">
```

width und *height* geben die Größe des Viewport an. *viewBox* bewirkt, dass auf 4cm 64 gleichgroße Abschnitte und auf 5cm 80 gleichgroße Abschnitte verteilt werden sollen. Danach werden Zahlen ohne Einheit mit der Einheit 1/16cm interpretiert.

Aspekt Ratio

Wenn man allerdings in x-Achsen Richtung eine andere Einheit wählt als in y-Achsen Richtung, dann sieht das Bild verzerrt aus.

Abbildung 4: Verzerrte Grafik

Um das zu verhindern, gibt es das *preserveAspectRatio* Attribut. Als Parameter gibt man einmal die Ausrichtung der *viewBox* an und zum anderen, ob die *viewBox* im Ganzen oder evtl. nur ein Teil angezeigt werden soll.

```
<svg width="45" height="135" viewBox="0 0 90 90"
      preserveAspectRatio="xMinYMin meet">
```

Dieses Beispiel bewirkt, dass die *viewBox* so verkleinert wird, so dass sie komplett in den Viewport passt und im oberen linken Bereich des Viewport angezeigt wird.

Abbildung 5: Original (100%)

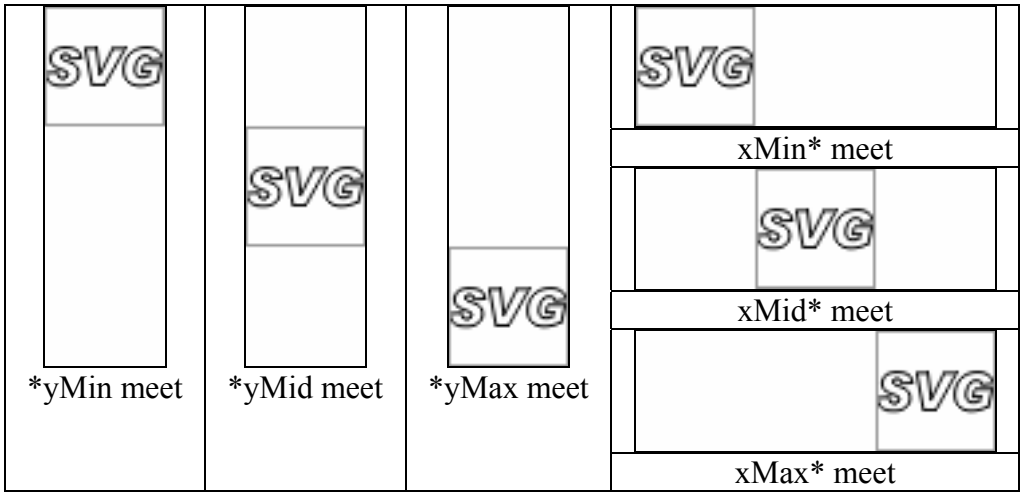


Abbildung 6: preserveAspectRatio meet

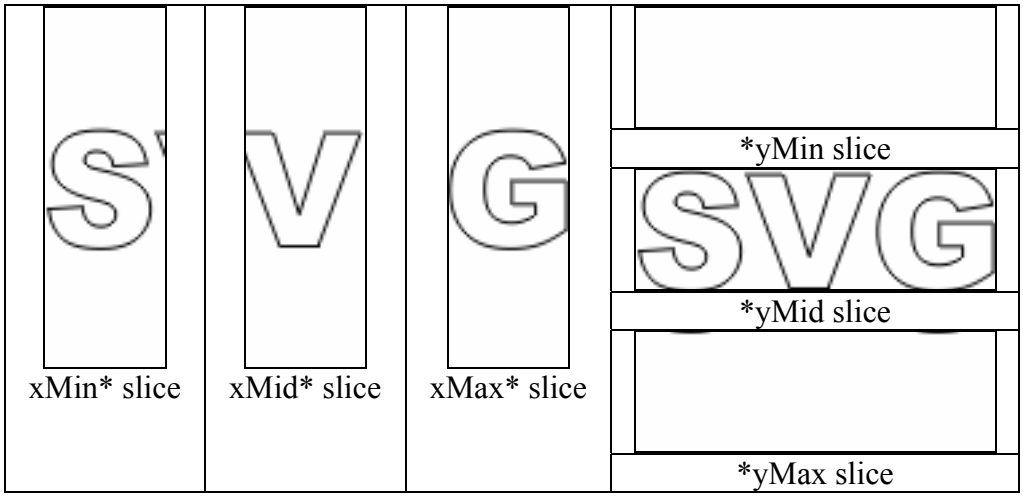


Abbildung 7: preserveAspectRatio slice

Grundformen

Nachdem das Koordinatensystem eingerichtet ist, kann mit der Zeichnung begonnen werden. Hier sollen die Grundelemente im Einzelnen erklärt werden.

Linien

Eine Linie wird durch das <*line*> Element angegeben.

```
<line x1="..." y1="..." x2="..." y2="..."/>
```

x1 und *y1* legen den Startpunkt und *x2* und *y2* den Endpunkt der Linie fest.

Pinselstrich (Stroke)

Linien oder andere geometrische Formen werden mit einem Pinsel gezeichnet. Mit dem *stroke-width* Attribut kann man die Pinselbreite und mit *stroke* die Farbe des Pinsels angeben. Ausserdem kann man mit *stroke-opacity* die Deckkraft einstellen und mit *stroke-dasharray* kann man den Pinselstrich stückeln. *stroke-width, stroke*, *stroke-opacity* und *stroke-dasharray* sind also Eigenschaften die das Aussehen des Pinsels verändern. Sie beeinflussen aber nicht die geometrische Form, die damit gezeichnet werden soll. Daher werden diese Attribute im *style* Attribut aufgelistet.

```
<line ... style="stroke-with: 10; stroke: red; stroke-opacity: 0.2;
stroke-dasharray: 5, 2;">
```

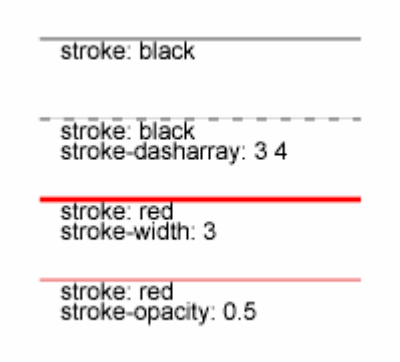


Abbildung 8: Linien

Rechtecke

Ein Rechteck wird durch das <*rect*> Element angegeben.

```
<rect x="..." y="..." width="..." height="..."/>
<rect x="..." y="..." width="..." height="..." rx="..." ry="..."/>
```

x und *y* geben die obere linke Ecke an, *width* die Breite und *height* die Höhe. Außerdem hat man die Möglichkeit, runde Ecken für Rechtecke mit den Attributen *rx* und *ry* anzugeben. Formen die eine Fläche einschließen, können mit Hilfe des *fill* Attributes ausgefüllt werden.

```
<rect ... style="fill: yellow;"/>
<rect ... style="fill: #000f66;"/>
```

Es gibt sowohl vordefinierte Farben

```
aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive,
purple, red, silver, teal, white und yellow
```

als auch selbst definierbare Farben. Das Format der eigenen Farben ist entweder in der Form #rgb oder #rrggbb, wobei r, g und b hexadezimale Werte sein müssen.

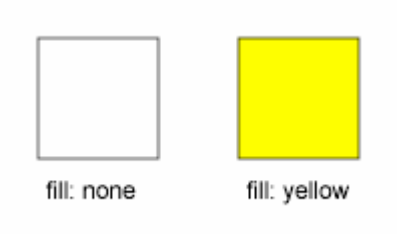


Abbildung 9: Füllung

Kreise and Ellipsen

Ein Kreis wird durch das <*circle*> Element angegeben.

```
<circle cx="..." cy="..." r="..."/>
```

cx und *cy* geben den Mittelpunkt und *r* den Radius an.

Eine Ellipse wird durch das <*ellipse*> Element angegeben.

```
<ellipse cx="..." cy="..." rx="..." ry="..."/>
```

cx und *cy* geben wieder den Mittelpunkt und *rx* und *ry* den Radius in x-Achsen und y-Achsen Richtung an.

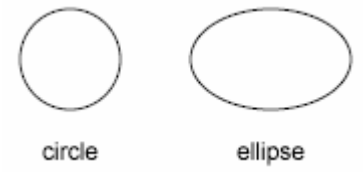


Abbildung 10: Kreis und Ellipse

Polygon

Wenn einem die vordefinierten Formen nicht genügen, so kann man mit Hilfe des <*polygon*> Elementes selber Formen beschreiben, z.B. einen Stern oder andere willkürliche Formen.

```
<polygon points="x1 y1, x2 y2, x3 y3, ..."/>
```

Die einzelnen Punkte des Polygons werden im *points* Attribut aufgelistet. Dabei bezeichnet *x1* und *y1* den Startpunkt. Von jedem Punkt wird eine Linie zum nächsten Punkt gezeichnet. Wenn es keine weiteren Punkte mehr in der Liste gibt, so wird einfach vom letzten Punkt eine Linie zum Startpunkt gezeichnet. Da es möglich ist, dass sich Linien schneiden, kann es zu Problemen mit dem *fill* Attribut kommen, da nicht immer eindeutig zwischen innerhalb und außerhalb unterschieden werden kann. Daher gibt man in diesem Fall zusätzlich das *fill-rule* Attribut an. Mit dem *fill-rule* Wert *nonzero* wird festgestellt, ob ein Punkt innerhalb oder außerhalb eines Polygons liegt, indem eine Linie von dem betrachteten Punkt unendlich weit weg gezogen wird. Wenn diese Linie von rechts nach links von einer Linie des Polygons geschnitten wird, wird um eins erhöht. Wenn die Linie von links nach rechts geschnitten wird, wird um eins erniedrigt. Wenn alle Schnittpunkte bearbeitet sind und das Ergebnis null ist, liegt der betrachtete Punkt außerhalb, ansonsten (nonzero) innerhalb. Der Wert *evenodd* arbeitet ähnlich. Allerdings wird jedes Mal wenn ein Schnittpunkt gefunden wurde, lediglich um eins erhöht. Wenn das Ergebnis gerade (even) ist, dann liegt der Punkt außerhalb, ansonsten (odd) innerhalb des Polygons.

Abbildung 11: Figure 3-12 [EIS02]

Polyline

Das Element <*polyline*> ist eigentlich nichts anderes als ein Polygon. Allerdings wird keine Linie zwischen den letzten und dem ersten Punkt gezogen.

```
<polyline points="x1 y1, x2 y2, x3 y3, ..."/>
```

Abbildung 12: Figure 3-13 [EIS02]

Line Caps und Joins

Mit dem Attribut *stroke-linecap* kann man die Form der Endpunkte beeinflussen. Der Wert *butt* bewirkt, dass die Linie direkt am Endpunkt abgeschnitten wird. Mit *round* wird ein ausgefüllter Halbkreis, mit einem Radius der *stroke-width* / 2 entspricht, ans Ende der Linie gezeichnet. Mit dem Attribut *square* wird die Linie um *stroke-width* / 2 verlängert.

Mit dem Attribut *stroke-linejoin* gibt man an, wie einzelne Linien verbunden werden sollen. Der Wert *miter* bewirkt, dass wenn die beiden Linien nicht in einem 180° Winkel aufeinander stehen, die Kanten der Linien so weit verlängert werden, bis diese sich schneiden. Wenn der Winkel zu spitz ist, kann man mit *stroke-miterlimit* verhindern, dass der Schnittpunkt nicht all zu weit weg liegt. Mit *round* werden Linien durch einen Kreis mit dem Radius *stroke-width* / 2 verbunden. Mit *bevel* werden lediglich die beiden äußeren Kanten der Linien mit einer Linie verbunden.

Abbildung 13: Figure 3-15 und 3-16 [EIS02]

Struktur des Dokumentes

Inline Styles

Attribute, die das Aussehen beeinflussen, werden im style Attribut aufgelistet.

```
<... style="..."/>
```

Man kann allerdings auch die Attribute direkt angeben.

```
<line stroke="red"/>
```

Man sollte aber die eigentlichen Daten von der Formatierung trennen.

Internal Stylesheets

Man kann aber auch ein <*style*> Element definieren und es dann beliebig oft im Dokument referenzieren. Dennoch kann am Element selber das <*style*> Attribut benutzt werden, um Stile zu überschreiben. Das Element <defs> dient lediglich dazu neue Elemente zu definieren.

```
<svg ...>
<defs>
<style type="text/css">
<![CDATA[circle {fill: #002ffc; stroke: blue; stroke-opacity: 0.3}]]>
</style>
</defs>
<cirlce cx="20" cy="20" r="30"/>
<cirlce cx="20" cy="60" r="30" style="fill: red"/>
</svg>
```

External Stylesheets

Ebenso ist es möglich einen Stylesheets für mehere SVG Dateien zu benutzen. Dazu wird lediglich eine CSS Datei angelegt, die von den SVG Dateien dann referenziert werden muss.

```
/* MyCSS.css */


* { fill:none; stroke: green } /* for all elements */
rect { stroke-dasharray: 7 2; }
circle.yellow { fill: yellow; }
.myclass { fill: blue }


/* SVG Datei */

<?xml version="1.0"?>
<?xml-stylesheet href="MyCSS.css" type="text/css"?>
<!DOCTYPE      svg      PUBLIC      "-//W3C//DTD      SVG      1.0//EN"
"http://www.w3c.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg ... >

/* fill: none; stroke: green */
<line .../>

/* fill: none; stroke: green; stroke-dasharray: 7 2 */
<rect .../>

/* fill: yellow; stroke: green */
<circle class="yellow" .../>

/* fill: blue; stroke: green */
<ellipse class="myclass" .../>

</svg>
```

Gruppen

Manchmal ist es einfacher Elemente zu einer Gruppe zusammenzufassen, um dann lediglich nur der Gruppe z.B. ein <*style*> Attribut zu zuweisen. Gruppen werden, sobald sie angegeben sind, auch gezeichnet. Wenn man dies nicht möchte, dann muss man sie innerhalb des <*defs*> Elements deklarieren.

```
<g style="...">
<line ...>
<rect ...>
</g>
```

Use

Um komplexere Grafiken, die öfter benutzt werden, nicht jedes Mal neu angeben zu müssen, kann man mit Hilfe des <*use*> Elements diese referenzieren und neu zeichnen lassen.

```
<g id="complex">
<line ...>
<line ...>
<rect ...>
```

```
</g>

<use xlink:href="#complex" x="..." y="..."/>
```

x und *y* geben die Position an, an der der Punkt (0,0) der Gruppe sein soll.

Symbol

Das <*symbol*> Element ist ebenso wie das <*g*> Element ein Container für komplexe Grafiken. <*symbol*> Elemente werden nicht gezeichnet, d.h. sie müssen nicht wie Gruppen innerhalb eines <*defs*> Elements stehen, damit sie nicht gezeichnet werden. Ausserdem kann man in einem <*symbol*> Element die Attribute *viewBox* und *preserveAspectRatio* benutzen.

Image

Um ein JPEG oder ein PNG Bild in einer SVG Grafik einzubinden, benötigt man das <*image*> Element.

```
<image xlink:href="Bild.jpg" x="..." y="..." width="..." height="..."/>
```

x und *y* geben die Position der oberen linken Ecke des Bildes an. *width* und *height* geben die Breite und Höhe des Bildes an. Das Bild wird, wenn es andere Werte für *width* und *height* hat, auf die neuen Werte skaliert.

Transformationen

Transformationen werden im *transform* Attribut eines Elementes aufgelistet. Man beachte, dass Transformationen nicht kommutativ sind.

```
<... transform="..."/>
```

Translate

Mit der Transformation *translate* verschiebt man das Element an den Punkt (x,y).

```
<... transform="translate(x,y)"/>
```

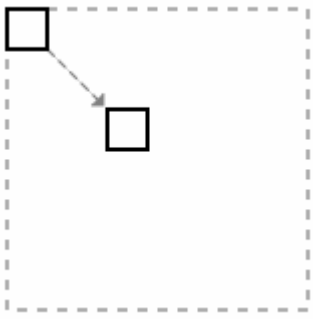


Abbildung 84: Figure 5-2 [EIS02]

Scale

Mit *scale* wird jeder Punkt des Elementes um den angegebenen Faktor multipliziert.

```
<... transform="scale(xy)"/>
<... transform="scale(x,y)"/>
```

Man kann zum einen nur einen Wert angeben, der dann sowohl für die x als auch die y Koordinate gleichermaßen benutzt wird, zum anderen kann man aber auch jeweils für die x Koordinate und die y Koordinate jeweils einen eigenen Wert angeben.

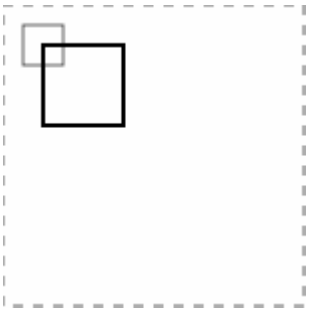


Abbildung 15: Figure 5-4 [EIS02]

Rotate

rotate dreht das Element um den angegebenen Winkel um den Punkt(0,0). Wenn zusätzlich zum Winkel auch noch x und y angegeben wird, wird das Element um den Punkt(x,y) gedreht.

```
<... transform="rotate(winkel, x, y)"/>
```

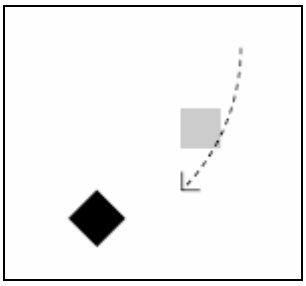


Abbildung 16: Figure 5-17 [EIS02]

Skew

skewX verschiebt alle x-Koordinaten um den angegebenen Winkel und lässt die y-Koordinaten unverändert. skewY verschiebt dagen alle y-Koordinaten um den angegebenen Winkel und lässt aber dir x-Koordinaten unverändert.

```
<... transform="skewX(winkel)"/>
<... transform="skewY(winkel)"/>
```

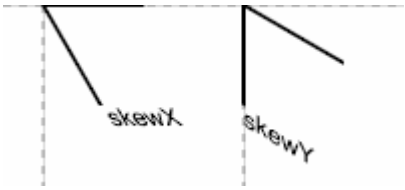


Abbildung 17: Figure 5-20 [EIS02]

Pfade

Das allgemeinste Element, ist das <*path*> Element. Mit ihm kann man Linien, Kreise und Kurven in einer Liste zusammenfassen und zeichnen lassen. Der Pfad funktioniert ähnlich einem Plotter, d.h. es gibt einen Stift, den man an einer beliebigen Stelle absetzen kann und mit dann mit dem Stift beginnt zu zeichnen. Man sollte jedoch die Grundformen dennoch in seiner SVG Datei benutzen, um eine bessere Lesbarkeit zu erreichen.

```
<path d="..."/>
```

d (data) enthält die einzelnen Anweisungen, die zusammen den Pfad bilden.

moveto, lineto und closepath

Die erste Anweisung eines Pfades ist stets der *moveto* Befehl. Damit legt man den Startpunkt des Pfades fest. Zu einem späteren Zeitpunkt im Pfad, bewirkt dieser Befehl, dass der Stift an einem neuen Punkt abgesetzt werden soll. Um sich viel Tipparbeit zu ersparen, wird der *moveto* Befehl durch m bzw. M abgekürzt, wobei ein kleiner Buchstabe bedeutet, dass der Punkt nach dem Befehl relative zum momentanen Punkt betrachtet werden soll und ein grosser Buchstabe bedeutet, dass der Punkt nach dem Befehls absolut betrachtet werden soll.
Ausserdem kann man nicht nur einen Punkt nach einem Befehl angeben, sondern mehrere Punkte, die dann alle so interpretiert werden, als ob jedes Mal der zuletzt angegebene Befehl davor stehen würde. Um eine Linie zu zeichnen, benutzt man den Befehl *lineto*. Auch dieser wird wieder durch l bzw. L angegeben. *lineto* zeichnet eine Linie vom momentanen Standpunkt des Stiftes zu dem Punkt hinter dem *lineto* Befehls. Mit dem Befehl *closepath* veranlasst man, dass der Stift eine Linie vom letzten Punkt eine Linie zum Startpunkt zeichnet. Der *closepath* Befehl wird durch Z abgekürzt.

```
<path d="M 10 10 L 50 10 L 10 50 Z"/>
```

Hier ein Beispiel eines Pfades:

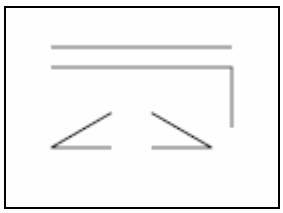


Abbildung 18: Figure 6-1 [EIS02]

Horizantale und vertikale Linien

Da man oft in einer Zeichnung horizontale und vertikale Linien benötigt, gibt es neben den *lineto* Befehlen (l bzw. L) auch noch die Befehle *horizontal lineto* (h bzw. H) und *vertical lineto* (v bzw. V) die lediglich nur noch eine Koordinate benötigen.

Bögen

Einen Bogen zeichnet man mit Hilfe des *elliptical arc* Befehls. Der *elliptical arc* Befehl wird mit a bzw A abgekürzt. Als Parameter erwartet er als erstes x- und y-radius des Bogens auf dem dann die einzelnen Punkte liegen. Dann wird *x-axis-rotation* angegeben. Dieser Wert gibt an, um wie viel Grad die x-Achse rotiert werden soll. Dann folgen zwei Flags. Das erste Flag, ist das *large-arc-flag*. Dieses Flag wird auf eins gesetzt, wenn der Bogen größer als 180° ist. Ist der zu zeichnende Bogen kleiner, so muss das Flag auf null gesetzt werden. Das zweite Flag, ist das *sweep-flag*. Das *sweep-flag* ist null, wenn der Bogen gegen den Uhrzeigersinn gezeichnet werden soll, ansonsten ist es auf eins gesetzt. Als letzten Parameter erwartet der *elliptical arc* Befehl den Endpunkt des Bogens. Der Startpunkt ist wieder der letzte Standpunkt vor diesem Befehl.

```
<path d="M 125,75 A 100,50 0 0 0 225,125"/>
```

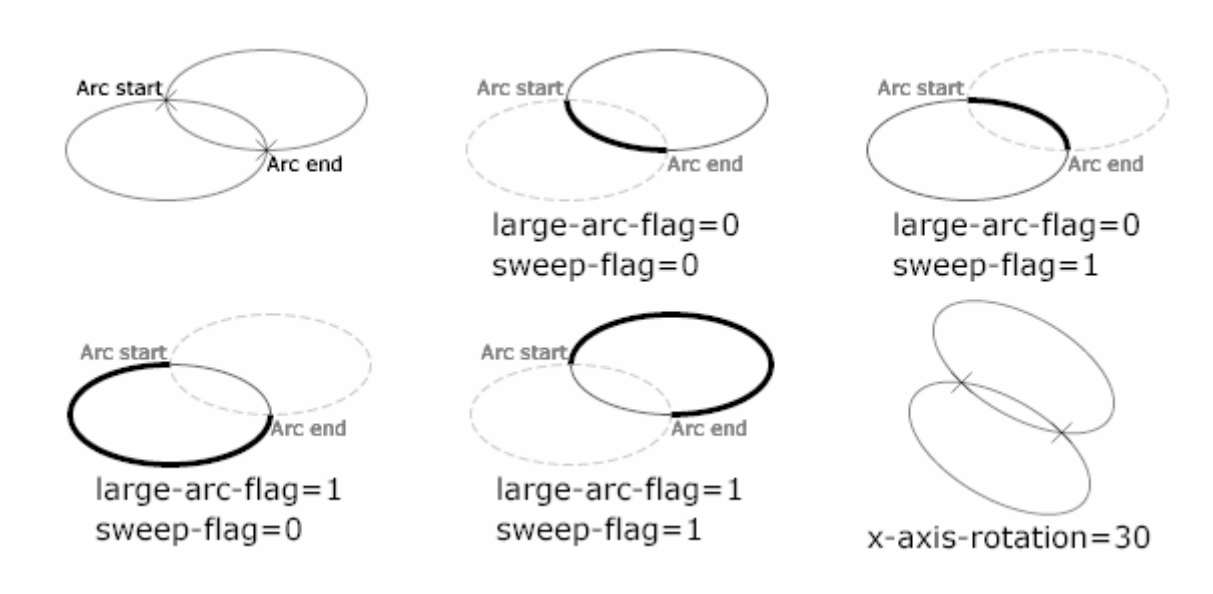


Abbildung 19: Figure 6-4 [EIS02]

Bezier Kurven

Es gibt zwei verschiedene Arten von Bezier Kurven in SVG.

Quadratische Bezier Kurven

Um eine quadratische Bezier Kurve zu zeichnen, benutzt man den q bzw. Q Befehl. Als ersten Parameter erwartet dieser Befehl die Koordinaten des Kontrollpunktes. Und als nächsten Parameter den Endpunkt der Kurve.

```
<path d="M 30 75 Q 240 30 300 120"/>
```

Man kann auch bei diesem Befehl wieder mehrere Kontrollpunkte und Endpunkte angeben, ohne jedes Mal wieder den q bzw. Q Befehl angeben zu müssen.

```
<path d="M 30 100 Q 80 30 100 100 130 65 200 80"/>
```

Man kann auch den letzten Kontrollpunkt spiegeln und diesen für die nächste Kurve weiterbenutzen. Dafür benötigt man den T Befehl. Dadurch wird der Übergang zur nächsten Kurve weicher.

```
<path d="M 30 100 Q 80 30 100 100 T 200 80"/>
```

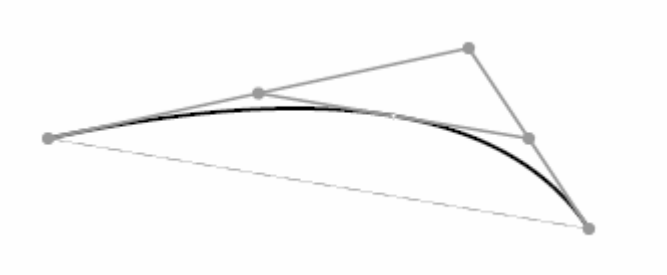


Abbildung 20: Figure 6-7 [EIS02]

Kubische Bezier Kurven

Kubische Bezier Kurven werden mit Hilfe des c bzw. C Befehls gezeichnet. Die ersten beiden Parameter geben diesmal zwei Kontrollpunkte an, der nächste Parameter gibt dann den Endpunkt an.

```
<path d="M 20 80 C 50 20 150 60 200 120"/>
```

Es können weitere Kontrollpunkte und Endpunkte angegeben werden, ohne den c bzw. C Befehl neu an zu geben.

```
<path d="M 30 100 C 50 50 70 20 100 100 110 130 45 150 65 100"/>
```

Auch hier kann der letzte Konrollpunkt der letzten Kurve gespiegelt werden und für die nächste Kurve als ersten Kontrollpunkt genutzt werden. Auch hierdurch wird der Übergang zur nächsten Kurve wieder weicher. Der Befehl hierfür ist der s bzw. S Befehl.

```
<path d="M 30 100 C 50 30 70 50 100 100  S 150 40 200 80"/>
```

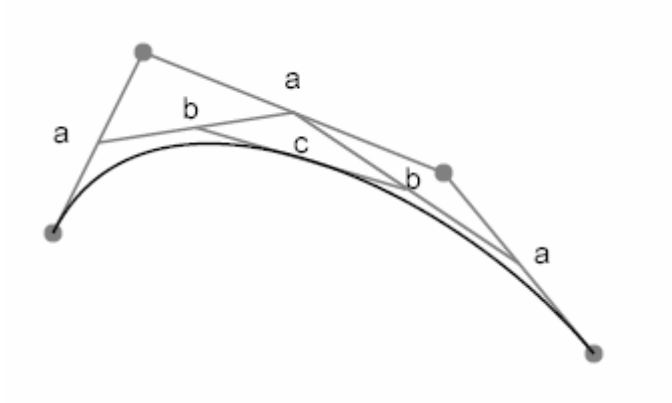


Abbildung 21: Figure 6-10 [EIS02]

Muster (Patterns) und Farbverläufe (Gradients)

Muster

Muster werden benutzt, um eine geometrische Form oder Pinselstriche mit sich wiederholenden Formen zu füllen. Um aus einer Form ein Muster (pattern) zu machen, muss die Form lediglich in das <*pattern*> Element eingebettet werden. Das <*pattern*> Element benötigt als Attribute *x* und *y*, welche die obere linke Ecke der Füllform festlegt. Dann noch die Attribute *width* und *height*, die die Breite und Höhe der Füllform festlegen. Diese werden entweder nur als Dezimalzahlen oder in Prozent angegeben. Mit dem Attribut *patternUnits* legt man fest, wie die Füllformen in der zu füllenden Form angeordnet werden sollen. Der Wert *objectBoundingBox* bewirkt, dass soviel Füllelemente, wie, durch *height* und *width* angegeben, in die zu füllende Form passen, gezeichnet werden. Wenn mehr Platz da ist, als gebraucht wird, wird einfach der übrige Platz zwischen den Füllformen aufgeteilt. Wenn weniger Platz da ist, als benötigt wird, wird einfach etwas von der Füllform abgeschnitten. Mit dem Wert *userSpaceOnUse* werden die Füllformen einfach nebeneinander gezeichnet und evtl. nur an den Grenzen der zu füllenden Form abgeschnitten. Mit dem Attribut *patternContentUnits* legt man fest, ob die Füllformen auf die Fläche, die sie füllen, skaliert werden sollen oder ihre Dimensionen beibehalten sollen. Der Defaultwert ist *userSpaceOnUse*. Wenn man stattdessen den Wert *objectBoundingBox* angibt, verhindert man, dass evtl. zusätzlicher Platz zwischen den Füllformen entsteht oder dass die Füllformen evtl. beschnitten werden. Ein Muster wird benutzt, indem man dem *fill* Attribut einen Verweis auf das Muster als Wert angibt. Außerdem können Muster auch geschachtelt werden, d.h. man kann in einer Füllform schon Flächen mit anderen Mustern füllen.

```
<pattern id="mypattern" x="0" y="0" width="20%" height="20%"
patternUnits="objectBoundingBox">
<line x1="0" y1="0" x2="20" y2="20"/>
<line x1="0" y1="20" x2="20" y2="0"/>
</pattern>

<rect x="0" y="0" width="100" height="100" style="fill:
url(#mypattern)"/>
```

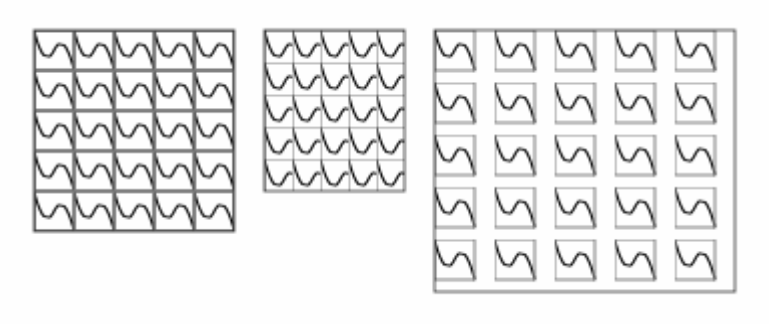


Abbildung 22: Figure 7-2 [EIS02]

Farbverläufe

Anstelle von Mustern, kann man Formen auch mit Farbverläufen füllen. Es gibt zwei verschiedene Farbverlaufsarten.

Linearer Farbverlauf

Ein linearer Farbverlauf besteht aus Farbübergängen entlang einer Linie. Ein <*linearGradient*> Element wird im <*defs*> Block angegeben. Im <*linearGradient*> Block gibt man nun mit Hilfe des <*stop*> Elements an, an welchen Stellen man welche Farbe haben möchte. Der Punkt x1,y1 gibt den Startpunkt und x2,y2 den Endpunkt der Farbverlauflinie an. Das Attribut *spreadMethod* kann drei Werte annehmen und ist dafür da, extra Platz zu behandeln. Einmal den Wert *pad*, der bewirkt, dass die Farben des Start- und Endpunktes, bis zu den Grenzen des zu füllenden Objektes ausgeweitet werden. Der Wert *repeat* bewirkt, dass der Farbverlauf (start-end) solange wiederholt wird, bis die Grenzen des zu füllenden Objektes erreicht sind. Der letzte Wert *reflect* bewirkt, dass der Farbverlauf abwechselnd (start-end, end-start, …) bis zu den Grenzen des Objektes wiederholt wird.

```
<linearGradient id="mygradient" x1="0%" y1="0%" x2="70%" y2="70%"
      spreadMethod="reflect">
<stop offset="0%" style="stop-color: red">
<stop offset="100%" style="stop-color: blue">
</linearGradient>

<rect x="0" y="0" width="100" height="100" style="fill:
      url(#mygradient)"/>
```

Abbildung 23: Figure 7-7 [EIS02]

Radialer Farbverlauf

Ein radialer Farbverlauf funktioniert genauso wie ein linearer Farbverlauf. Lediglich werden die Farbübergänge nicht an einer Linie entlang angeordnet, sondern an einem Kreis. D.h. anstelle von *x1,y1,x2* und *y2* gibt man den Mittelpunkt *cx* und *cy*, den Radius *r* und den Fokuspunkt *fx* und *fy* an. Der Fokuspunkt gibt den Punkt des Farbverlaufs, der später der Mittelpunkt des zu füllenden Objektes sein wird.

```
<radialGradient id="mygradient" cx"0%" cy="0%" r="70%" fx="0%" fy="0%"
  spreadMethod="reflect">
<stop offset="0%" style="stop-color: red">
<stop offset="100%" style="stop-color: blue">
</linearGradient>

<rect x="0" y="0" width="100" height="100" style="fill:
  url(#mygradient)"/>
```

Abbildung 24: Figure 7-11 [EIS02]

Text

Text wird in SVG im <*text*> Element angegeben. Die Position des Textes wird mit den Parameter *x* und *y* angegeben.

```
<text x="10" y="10">
      Text Text Text ...
</text>
```

Ohne weitere Angaben, wird der Text ab der angegebenen Position gerendert. Will man stattdessen den Text an den Punkt(x,y) horizontal zentrieren, dann muss man das *text-anchor* Attribut mit dem Wert *middle* mitangeben.

```
<text x="10" y="10" style="text-anchor: middle">
      Text Text Text ...
</text>
```

Die Schriftfamilie legt man mit *font-family* fest, die Grösse mit *font-size*. Mit *font-weight* stellt man ein, ob der Text fett oder normal gezeichnet werden soll. Des Weiteren kann man mit *font-style* festlegen, ob der Text kursiv oder normal dargestellt werden soll. Um Text unterstrichen darzustellen, bedient man sich dem *text-decoration* Attribut.
Man kann Text in SVG nicht nur von links nach rechts schreiben, sondern auch z.B. von oben nach unten.

```
<text  x="10"  y="10"  style="writing-mode:  tb;  glyph-orientation-
vertical: 0">
      Text Text Text ...
```

```
</text>
```

tb heißt hier top to bottom.
Text kann aber auch auf einem selbst definierten Pfad entlang gezeichnet werden. Dazu benutzt man das <*textPath*> Element.

```
<defs>
<path id="mypath" d="..."/>
</defs>

<text x="10" y="10">
<textPath xlink:href="#mypath">
Text Text Text ...
</textPath>
</text>
```

Schablonen (Clipping) und Masken (Masking)

Manchmal möchte man eine Grafik nicht einfach nur anzeigen, sondern eine Art Schablone darüber legen, durch die man nur die Grafik sehen kann. Dazu bedient man sich dem Clipping.

Schablonen

Um eine Schablone zu konstruieren, benutzt man das <*clipPath*> Element. Ebenso wie alle Elemente, die nicht direkt zeichnen, wird dieses Element in den <*defs*> Block geschrieben. Mit *clipPathUnits* wird eingestellt, ob die Schablone auf das zu betrachtende Objekt skaliert werden soll oder nicht.

```
<defs>
<clipPath id="myclippath" clipPathUnits="objectBoundingBox">
<circle cx="50%" cy="50%" r="30"/>
</clipPath>
</defs>

<g style="clip-path: url(#myclippath)">
<rect x="0" y="0" width="100" height="100"/>
</g>
```

Ein Beispiel:

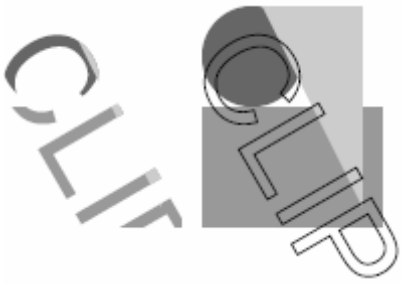


Abbildung 259: Figure 9-2 [EIS02]

Masken

Mit einer Maske kann deren Deckkraft auf andere Objekte übertragen werden. Dazu benutzt man das <*mask*> Element. Das <*mask*> Element sollte ebenfalls im <*defs*> Block angegeben werden. Die Maske kann sämtliche Formen enthalten, die man angibt.

```
<defs>
<mask id="mymask" x="0" y="0" width="1" height="1"
      maskContentUnits="objectBoundingBox">
<circle cx="50%" cy="50%" r="50" style="fill: red; fill-opacity:
      0.5;"/>
</mask>
</defs>

<g style="mask: url(#mymask)">
<rect x="0" y="0" width="100" height="100"/>
</g>
```

Hier ein feines Beispiel:

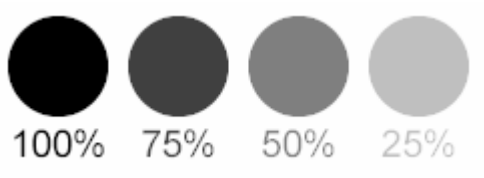


Abbildung 25: Figure 9-5 [EIS02]

Filter

Um Filter zu benutzen gibt es das <*filter*> Element. In diesem werden dann die eigentlichen Filteroperationen angegeben.

```
<defs>
<filter id="myfilter">
<!-- Filteroperationen -->
<filter/>
<defs/>

<g id="complex" style="filter: url(#myfilter);">
<!-- Zeichnung -->
</g>
```

Filteroperationen

Es gibt sehr viele Filteroperationen. Daher werden hier nur einige anhand eines kleinen Beispiels etwas näher erläutert werden können.

```
<filter id="myfilter">
<feGaussianBlur in="SourceAlpha" stdDeviation="2" result="blur"/>
<feOffset in="blur" dx="4" dy="4" result="offsetBlur"/>
<feMerge>
<feMergeNode in="offsetBlur"/>
```

```
<feMergeNode in="SourceGraphic"
</feMerge>
</filter>
```

feGaussianBlur bekommt als Eingabe die Alphawerte (*SourceAlpha*) der zu filternden Grafik und gibt als Ausgabe eine Grafik zurück, die über den Wert des *result* Attributs referenziert werden kann. Das Attribut *stdDeviation* gibt an, wie stark die Grafik verwischt werden soll. *feOffset* bekommt das Ergebnis der vorigen Filteroperation als Eingabe und verschiebt die eingegebene Grafik um 4 in x-Achsen und y-Achsen Richtung. Die Ausgabe kann wieder durch den Wert des *result* Attributs referenziert werden. Mit *feMerge* legt man einzelne Eingaben (*SourceGrafik*, *SourceAlpha*, oder Ausgabe einer Filteroperation) in einem Stack ab. Dieser sorgt dafür, dass alle Eingaben genauso, wie sie abgelegt wurden, auch wieder nacheinander gezeichnet werden.

Abbildung 26: Figure 10-3 [EIS02]

Animationen und Scripting

SMIL2

<animate>

Für animierte Bilder in SVG benutzt man SMIL2. SMIL2 ist ebenso wie SVG eine auf XML basierte Technologie. SMIL2 ermöglicht es einem, Animationen mit Hilfe des <*animate*> Elements zu erstellen.

```
<rect x="0" y="0" width="0" height="10"/>

<animate attributeName="width" attributeType="XML" from="0" to="100"
      begin="0s" dur="5s" fill="freeze" repeatCount="2"/>
</rect>
```

Das Attribut *attributeName* gibt an welches Attribut Veränderungen unterliegen soll. *attributeType* gibt an, um welchen Typ von Attribut es sich dabei handelt. Der Wert *XML* besagt, dass das Attribut Eigenschaften des Elements selber enthält. Den Wert *CSS* würde man also benutzen, wenn solch ein Attribut das Aussehen bzw. Formatierung eines solchen Elementes festlegt. *from* gibt den Anfangswert und *to* den Endwert des Attributs an. *begin* bestimmt den Zeitpunkt, an dem die Animation beginnen soll und *dur* wie lange die Animation laufen soll, bis der Endzustand erreicht ist. Wenn die Animation vorbei ist, bestimmt *fill* was mit der verbleibenden Zeit geschähen soll, wo die Animation nicht mehr

läuft. Der Wert *freeze* bewirkt, dass die Animation in ihrem Endzustand bleibt und nicht ihren Anfangszustand wieder einnimmt. Der Wert *remove* ist der Defaultwert. Das Attribut *repeatCount* gibt an, wie oft die Animation durchlaufen werden soll.

<set>

Bei einigen Attributen ist es nicht notwendig oder erforderlich, einen Wert über eine bestimmte Zeit sich ändern zu lassen. Daher gibt es das <set> Element.

```
<rect x="0" y="0" width="0" height="10" style="visibility: hidden"/>
<set attributName="visibility" attributeType="CSS" to="visible"
      begin="5min" dur="1s" fill="freeze"/>
</rect>
```

Hier wird wieder der Name des Attributs durch *attributeName* festgelegt. Der Wert *CSS* des *attributeType* Attributs besagt, dass es sich hier um ein Attribut handelt, welches das Aussehen bzw. die Formatierung der Daten, vornimmt. *to* gibt den neuen Wert an. Hier soll der Wert von *hidden* auf *visible* geändert werden.

<animateColor>

Das <animate> Element funktioniert nicht mit Farben, da diese nicht nur durch eine Zahl angegeben werden. Daher gibt es das <animateColor> Element.

```
<rect x="0" y="0" width="0" height="10" style="fill: yellow"/>
<animateColor attributName="fill" from="yellow" to="#00ff00"
      begin="5s" dur="15s" fill="freeze"/>
</rect>
```

<animateTransform>

Ebenso funktioniert das <*animate*> Element nicht mit Transformationen. Daher gibt es dafür ebenfalls ein eigenes Element, das <animateTransform> Element.

```
<rect x="0" y="0" width="10" height="10"/>
<animateTransform attributName="transform" attributeType="XML"
      type="scale" from="0" to="100 20" begin="5s" dur="15s"
      fill="freeze"/>
</rect>
```

type gibt den Typ der Transformation an. Wenn man mehrere Transformationen auf ein Objekt anwenden will, so muss man das Attribut *additive* mit dem Wert *sum* angeben. Dann werden alle Transformationen nacheinander ausgeführt, ohne die vorige Transformation zu ersetzen.

<animateMotion>

Um ein Objekt zu bewegen, konnte man bisher die Transformation *translate* benutzen. Wenn man aber nicht nur ein Objekt an einer Linie entlang bewegen möchte, dann muss man das <*animateMotion*> Element benutzen. Bei diesem Element kann ein eigener Pfad angegeben werden, an dem das Objekt entlang bewegt wird. Das Format des <*animateMotion*> Pfades ist das gleiche, wie das Format des SVG Pfades.

```
<rect x="0" y="0" width="10" height="10"/>
<animateMotion path="M 50 125 C 100 25 150 225 200 125" begin="5s"
      dur="15s" fill="freeze"/>
</rect>
```

ECMA (Javascript)

Um nicht nur Animationen sondern auch Interaktion in einem SVG zu ermöglichen, benutzt man ECMA Script.

Hier ein kleines Beispiel:

```
<script type="text/ecmascript">
<![CDATA[
function enlarge_circle(evt) {

var circle = evt.getTarget();

circle.setAttribute("r", 50);

}

function shrink_circle(evt) {

var circle = evt.getTarget();

circle.setAttribute("r", 25);
}

// ]]>
</script>

<circle cx="150" cy="100" r="25" fill="red"
        onmouseover="enlarge_circle(evt)"
        onmouseout="shrink_circle(evt)"/>
```

Abkürzungen

ECMA	European Computer Manufacturer's Association (Javascript)
SMIL2	Synchronized Multimedia Integration Language Level 2
SVG	Scalable Vector Graphics
PGML	Precision Graphics Markup Language
VML	Vector Markup Language
XML	eXtensible Markup Language

Literatur

[EIS02] J. D. Eisenberg, “SVG Essentials”, 2002
[FOL91] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes, “Computer Graphics: Principles and Practice”, 1991

Anhang

CSS Attribute

Name	Werte	Anwendung bei
alignment-baseline	auto | baseline | before-edge | text-before-edge | middle | after-edge | text-after-edge | ideographic | alphabetic | handing | mathematical	<tspan>, <tref>, <altGlyph>, <textPath> Elemente
baseline-shift	baseline | sub | super | *prozent* | *länge*	<tspan>, <tref>, <altGlyph>, <textPath> Elemente
Clip	*shape* | auto	Elemente, die einen Viewport einrichten und <pattern>, <marker> Elemente
Clip-path	*Uri*	Container Elemente und Grafik Elemente
Clip-rule	nonzero | evenodd | class=noxref	Grafikelemente innerhalb eines <clipPath> Elements
color	*Color*	Elemente mit den Eigenschaften fill, stroke, stop-color, flood-color, lighting-color
color-interpolation	auto | sRGB | linearRGB	Container Elemente, Grafikelemente und <animateColor>
color-interpolation-filters	auto | sRGB | linearRGB	Filter
color-profile	auto | sRGB | name | uri	<image> Elemente
color-rendering	auto | optimizeSpeed | optimizeQuality	Container Elemente, Grafikelemente und <animateColor>
cursor	*uri* | auto | crosshair | default | pointer | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize | text | wait | help	Container Elemente und Grafikelemente
direction	ltr | rtl	<text>, <tspan>, <tref>, und <textPath> Elemente
display	inline | block | list-item | run-in | compact | marker | table |	<svg>, <g>, <switch>, <a>, <foreignObject>,

	inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | none	Grafikelemente, <text>, <tspan>, <tref>, <altGlyph>, <textPath>
dominant-baseline	Auto | use-script | no-change | reset-size | alphabetic | hanging | ideographic | mathematical | central | middle | text-after-edge | text-before-edge | text-top | text-bottom	Text beinhaltende Elemente
enable-background	accumulate | *new* [*x y width height*]	Container Elemente
Fill	*paint*	Formen(shapes) und Text beinhaltende Elemente
fill-opacity	*opacity-value*	Formen(shapes) und Text beinhaltende Elemente
fill-rule	nonzero | evenodd	Formen(shapes) und Text beinhaltende Elemente
filter	*uri* | none	Container Elemente und Grafikelemente
flood-color	currentColor | *color specifier*	<feFlood> Elemente
flood-opacity	*alphavalue*	<feFlood> Elemente
Font	font-style, font-variant, font-weight, font-size line-height, font-familiy | caption | icon | menu | message-box | small-caption | status-bar	Text beinhaltende Elemente
Font-family	series of *familiy-name* oder *generic-familiy*	Text beinhaltende Elemente
Font-size	*absolute-size* | *relative-size* | *length* | *percentage*	Text beinhaltende Elemente
Font-size-adjust	*number* | none	Text beinhaltende Elemente
Font-stretch	normal | wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded	Text beinhaltende Elemente
Font-style	normal | italic | oblique	Text beinhaltende Elemente
Font-variant	normal | small-caps	Text beinhaltende Elemente
Font-weight	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900	Text beinhaltende Elemente
glyph-orientation-horizontal	*winkel*	Text beinhaltende Elemente
glyph-orientation-vertical	auto | *winkel*	Text beinhaltende Elemente
image-rendering	auto | optimizeSpeed |	Bilder

	optimizeQuality	
kerning	auto | *länge*	Text beinhaltende Elemente
letter-spacing	normal | *länge*	Text beinhaltende Elemente
lighting-color	currentColor | *farbe*	<feDiffuseLighting> und <feSpecularLighting> Elemente
marker, marker-start, marker-mid, marker-end	none | *uri*	<path>, <line>, <polyline> und <polygon> Elemente
mask	none | *uri*	Container Elemente und Grafikelemente
opacity	*alphawert*	Container Elemente und Grafikelemente
overflow	visible | hidden | scroll | auto	Elemente, die einen Viewport einrichten und <pattern>, <marker> Elemente
pointer-events	visiblePainted | visibleFill | visibleStroke | visible | painted | fill | stroke | all | none	Grafikelemente
shape-rendering	auto | optimizeSpeed | crispEdges | geometricPrecision	Formen(shapes)
Stop-color	currentColor | *farbe*	<stop> Elemente
Stop-opacity	*alphawert*	<stop> Elemente
stroke	*paint*	Formen(shapes) und Text beinhaltende Elemente
stroke-dasharray	none | *dasharray*	Formen(shapes) und Text beinhaltende Elemente
stroke-dashoffset	*dashoffset*	Formen(shapes) und Text beinhaltende Elemente
stroke-linecap	butt | round | square	Formen(shapes) und Text beinhaltende Elemente
stroke-linejoin	miter | round | bevel	Formen(shapes) und Text beinhaltende Elemente
stroke-miterlimit	*miterlimit*	Formen(shapes) und Text beinhaltende Elemente
stroke-opacity	*opacity wert*	Formen(shapes) und Text beinhaltende Elemente
stroke-width	*breite*	Formen(shapes) und Text beinhaltende Elemente
Text-anchor	start | middle | end	Text beinhaltende Elemente
Text-decoration	none | underline | overline | line-through | blink	Text beinhaltende Elemente
Text-rendering	auto | optimizeSpeed | optimizeLegibility | geometricPrecision	<text> Element
unicode-bidi	normal | embed | bidi-override	Text beinhaltende Elemente
visibility	visible | hidden | collapse	Grafikelemente, <text>,

		<tspan>, <tref>, <altGlyph>, <textPath> und <a>
word-spacing	normal | *länge*	Text beinhaltende Elemente
writing-mode	lr-tb | rl-tb | tb-rl | lr | rl | tb	<text> Elemente

Die Grafikformate
GIF und PNG

von

Annette Kaudel

Einleitung

Die Grafikformat GIF wurde Ende der 80er Jahre entwickelt, als der Bedarf für ein plattformunabhängiges Rastergrafikformat bestand, um Grafiken über Datennetze auszutauschen. So sollte es auch eine gute Kompression bieten, die mit Hilfe des LZW-Algorithmus erreicht wurde. Besonders im Internet hat sich GIF dadurch sehr stark durchgesetzt.
Nachdem die Firma Unisys 1994 Gebührenforderungen für den LZW-Kodieralgorithmus stellt, dessen Patent sie besitzt, entsteht die PNG Gruppe. Daraus geht das keine Patentrechte verletzende neue Grafikformat PNG hervor.

Historie

Ein paar wichtige Daten zur Geschichte der Grafikformate GIF und PNG:

1977	Abrahem Lempel und Jacop Ziv erfinden den Kompressionsalgorithmus LZ7	
1978	Der Algorithmus LZ78 wird als Variante von LZ77 entwickelt.	
1981	In den USA ist es nun möglich Software patentieren zu lassen. Abrahem Lempel und Jacop Ziv patentieren ihren LZ78 Algorithmus.	
1983	(20. Juni) Terry A. Welch (Sperry Corporation - später Unisys) patentiert das LZW-Kompressionsverfahren unter dem Titel „High speed data compression and decompression apparatus and method“, welches eine Variante des LZ78 Algorithmus ist. Die Ähnlichkeit, wird aber vom Patentamt nicht erkannt. (weitere Varianten von LZ77, LZW wurden entwickelt und teils auch patentiert)	
1987	(15. Juni) CompuServe veröffentlicht GIF (Version 87a) als freie und offene Spezifikation.	
1989	Die Version GIF 89a wird von CompuServe vorgestellt.	
1993	Unisys informiert CompuServe über die Verwendung ihres patentierten LZW-Algorithmus in GIF.	
1994	(29. Dez.)Unisys gibt öffentlich bekannt, Gebühren für die Verwendung des LZW-Algorithmus einzufordern.	
1995	(4. Jan.)	Die PNG Gruppe wird in einigen Diskussionsforen gegründet.
	(6. Jan.)	Der Name PNG steht fest.
	(7 Feb.)	CompuServe kündigt volle Unterstützung für PNG an.
	(7. März)	Die ersten PNG Bilder werden ins Netz gestellt..
	(8.Dez.)	Das *World Wide Web Consortium* (W3C) veröffentlicht die PNG-Spezifikation 0.92 als offizielles Arbeitsdokument.
1997	Die Internetbrowser Netscape 4.04 und Internet Explorer 4.0 erscheinen mit PNG Unterstützung.	
2003	Das GIF-Patent von Unisys läuft aus.	

GIF

Das Grafikformat GIF hat sich besonders im Bereich des Internets sehr stark durchgesetzt, wofür auch die folgenden Eigenschaften verantwortlich sind.

Eigenschaften

GIF ist ein Raster Grafikformat mit einer Farbtiefe 1 bis 8 Bit (256 Farben) pro Einzelbild. Zusätzlich kann eine Farbe als transparent angegeben werden. Es besteht die Möglichkeit mehrere Einzelbilder in eine Datei zu speichern, womit sich z.B. Animationen erzeugen lassen. Das verwendete LZW-Kompressionsverfahren bietet eine verlustfreie Kompression. Mit dem einstellbaren Interlaced Modus ist zudem möglich noch während der Übertragung ein grobe Vorschau des Bildes anzuzeigen.

Dateiaufbau

Eine GIF-Datei besteht aus 3 Blöcken, dem Header, dem Bilddatenblock, und dem Terminator. Der Bilddatenblock kann mehrfach vorkommen, falls die GIF Grafik mehrere Einzelbilder enthält, wie sie z.B. bei Animationen eingesetzt werden. Zusätzlich können auch noch Erweiterungsblöcke (ab Version 89a) vorkommen, die sich zwischen zwei Bilddatenblöcken oder hinter dem letzten Bilddatenblock befinden dürfen.

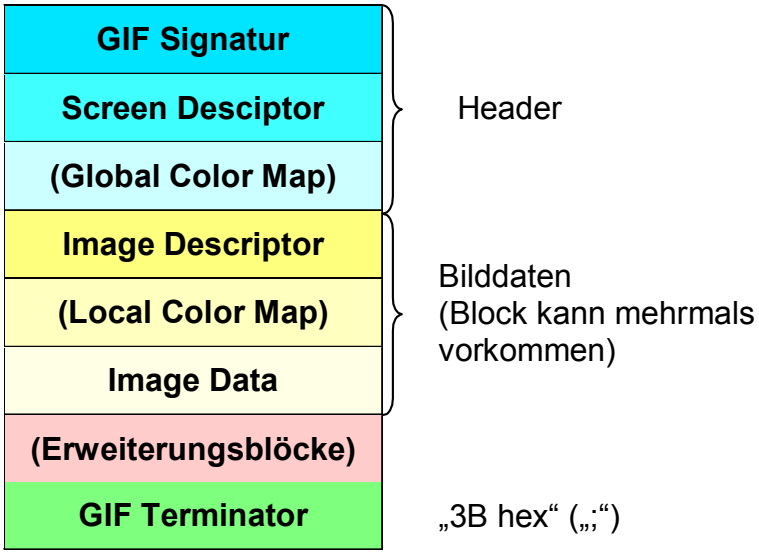


Abbildung 1: Dateiaufbau von GIF

Header

Der Header ist der Dateikopf. Hier werden alle grundsätzlichen Angaben zur Grafik gemacht, die für alle Teilbilder gelten sollen. Den Header besteht aus den Abschnitten: GIF-Signatur, Screen Descriptor und optionale globale Farbtabelle (Global Color Map).

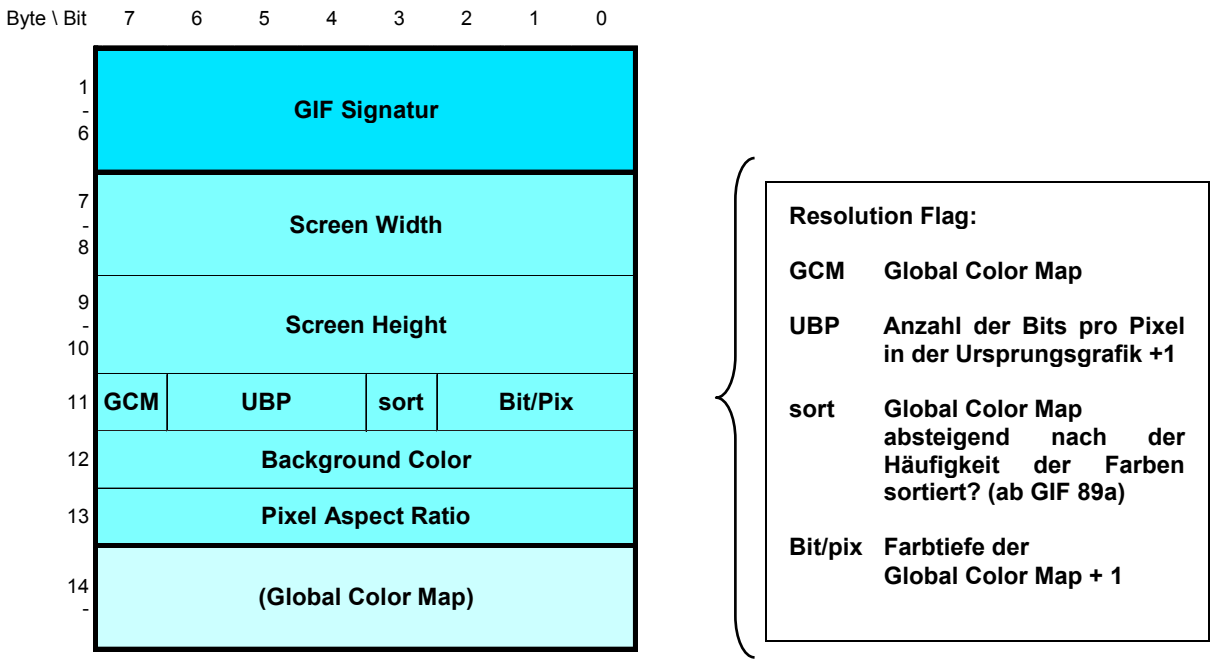


Abbildung 2: Aufbau des GIF Header

Jede GIF-Datei beginnt mit der **Signatur**. Diese besteht aus den ASCII Zeichen: „GIF87a“ oder „GIF89a“. die 87 bzw. 89 gibt dabei die verwendete GIF-Version an.

Nach der Signatur folgt der **Screen Descriptor**. Hier wird die Breite und Höhe des gesamten Bildschirms in Pixeln angegeben. Die Einzelbilder können innerhalb dieses definierten Bildschirms dargestellt werden.
Das nächste Byte ist das sogenannte Resolution Flag. Innerhalb dieses Bytes werden folgende Angaben gemacht:

1. ob eine globale Farbtabelle existiert
2. welche Farbtiefe die Ursprungsgrafik vor dem abspeichern als GIF hatte
   (Dieser Wert hat keine Wirkung auf die Darstellung.)
3. ob die globale Farbtabelle absteigend nach Häufigkeit der Farbe sortiert ist
   (Falls auf einem System weniger Farben als benötigt dargestellt werden können, ist es so möglich die am häufigsten benötigten Farben bevorzugt auszuwählen.)
4. Farbtiefe der globalen Farbtabelle in Bit/Pixel - 1

Anschließend wird der Index des Hintergrundfarbewertes für den Bildschirm in der globalen Farbtabelle angegeben. Falls keine globale Farbtabelle existiert, wird der Index in der Standard Farbtabelle verwendet.
Darauf folgt das Pixel-Abbildungs-Verhältnis (Pixel Aspect Ratio). Mit diesem Wert ist es möglich, das Verhältnis von Pixelhöhe und Pixelbreite nach folgender Formel anzugeben: Pixel-Abbildungs-Verhältnis = (Pixel Aspect Ratio + 15) / 64 . (Dieses Byte wird erst ab der Version 89a verwenden. Zuvor mussten alle Bits 0 sein.)

Falls vorhanden, folgt nun die **globale Fabtabelle**. Die einzelnen Farbwerte werden als 24-Bit-Farbwerte (RGB) hintereinander angegeben.

Bilddatenblock

Im Bilddatenblock werden die Angaben gespeichert, die nur das jeweilige Einzelbild betreffen. Auch dieser Block ist in drei Abschnitte gegliedert. Dem Image Descriptor, einer optionalen lokalen Farbtabelle (Local Color Map) und den eigentlichen kodierten Pixelwerten.

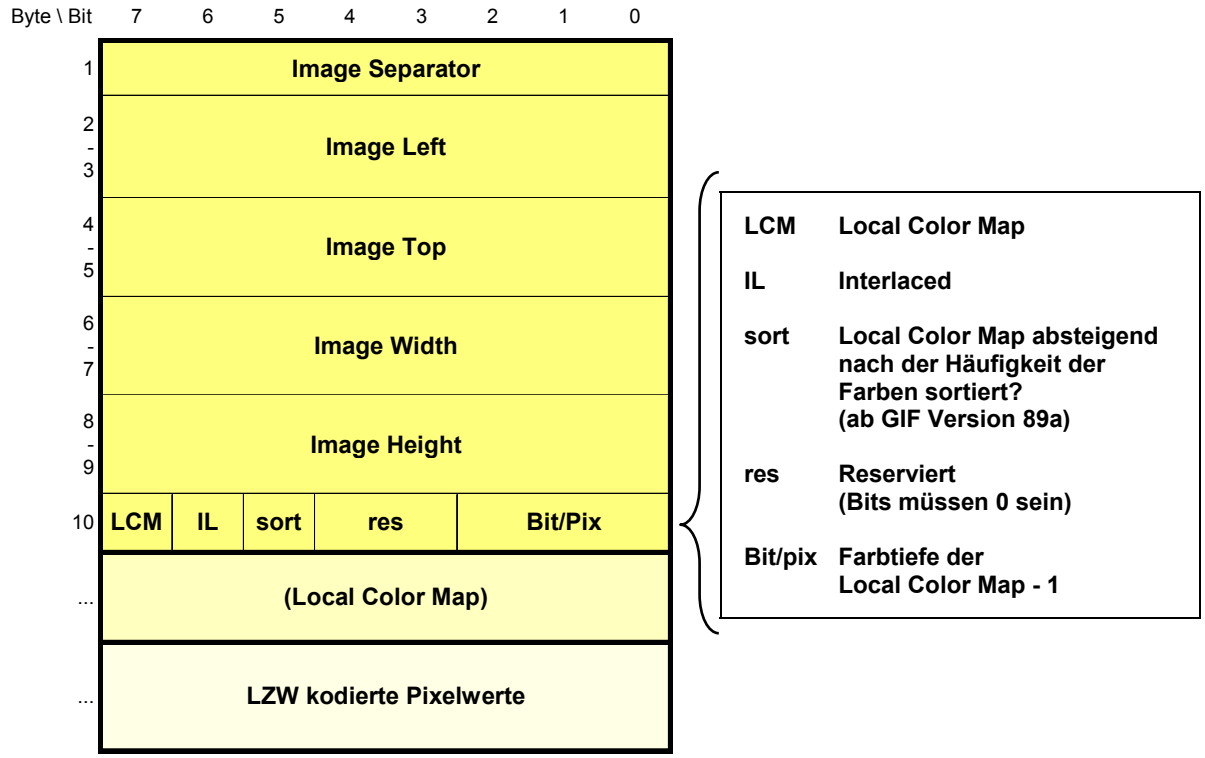


Abbildung 3: Aufbau des GIF Bilddatenblockes

Image Descriptor:

Jedes Einzelbild beginnt mit dem 1 Byte langen **Image Separator**, der aus dem hexadezimal Wert „2C“ (ASCII: „ , “) besteht. Dieses Byte gibt somit den Beginn eines neuen Einzelbildes an.

Danach folgen die Angaben Image Left und Image Top, die angeben, um wieviel Pixel das Einzelbild von der linken oberen Ecke auf dem GIF-Bildschirm verschoben angezeigt wird. Anschließend wird die Breite und Höhe des Einzelbildes definiert.

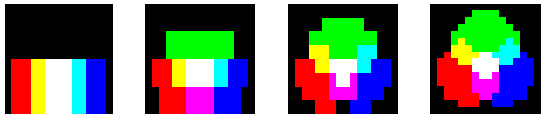
Das letzte Byte des Image Descriptors beinhaltet wiederum mehrere Parameter:

1. ob eine lokale Farbtabelle für dieses Einzelbild existiert
2. ob das Bild im Interlaced Modus abgespeichert wurde
3. ob die lokale Farbtabelle absteigend nach Häufigkeit der Farbe sortiert ist
4. zwei reservierte Bits (müssen 0 sein)
5. Farbtiefe der lokalen Farbtabelle in Bit/Pixel – 1

Falls vorhanden folgt nun eine **lokale Farbtabelle**, die nur für das jeweilige Einzelbild gilt, und eine eventuell vorhandene globale Farbtabelle ersetzt.

Interlaced Modus

Falls das Bild im Interlaced Modus abgespeichert wird, sind die Bildzeilen nach folgendem Schema umsortiert abgespeichert.



Zeile	Schritt 1	Schritt 2	Schritt 3	Schritt 4
0	1a			
1				4a
2			3a	
3				4b
4		2a		
5				4c
6			3b	
7				4d
8	1b			
9				4e
10			3c	
11				4f
12		2b		
13				4g
14			3d	
15				4h

Abbildung 10: die 4 Schritte des Interlaced Verfahrens bei GIF

Die Bildzeilen werden in 4 Schritten geladen:

1. jede 8. Zeile ab Zeile 0
2. jede 8. Zeile ab Zeile 4
3. jede 4. Zeile ab Zeile 2
4. jede 2. Zeile ab Zeile 1

So ist es möglich nach nur wenig übertragenen Daten schon eine grobe Vorschau des Bildes darstellen zu können.

Bilddaten

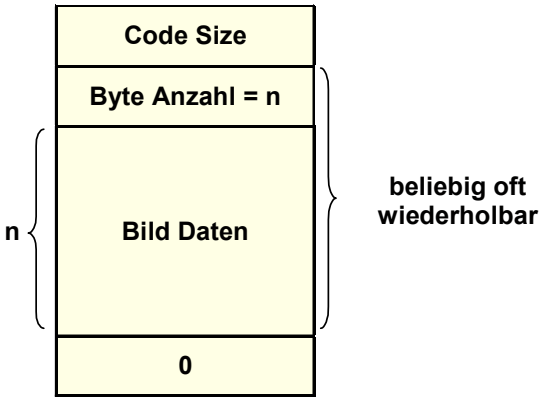


Abbildung 5: GIF Bilddaten (LZW komprimiert)

Das erste Byte der Bilddaten gibt die anfängliche Codelänge (Code Size) in Bits an. Normalerweise sind das die Anzahl der Bits pro Pixel (Farbtiefe) des Bildes. Außer bei Schwarz-Weiß Bildern, da muss der Wert 2 sein.
Danach folgen beliebig viele Datenblöcke mit den kodierten Pixelwerten beginnend von links oben. Jeder dieser Datenblöcke beginnt mit einem Byte welches die Länge der folgenden Daten in Byte (1-255) ohne dieses Byte angibt. Sind die kodierten Daten länger als 155 Byte wird ein neuer Datenblock nach dem selben Schema angehängt.
Nach dem letzten Datenblock folgt ein Nullbyte welches das Ende der Datenblöcke und somit auch des gesamten Einzelbildes anzeigt.

LZW-Kompression

Die Pixelwerte werden mit Hilfe des LZW-Algorithmus komprimiert. Die Grundidee der LZW-Kompression ist wiederkehrende Zeichenketten durch kürzere Codes zu ersetzen.

Besonderheiten bei GIF:

Codevergabe:

0 ... 2^Code Size – 1	jeder Paletteneintrag erhält einen Code
2^Code Size	„Clear Code“: Codetabelle wird auf den Anfangswert zurückgesetzt
2^Code Size + 1	„End of Information Code“: zeigt das Ende der LZW Daten an
2^Code Size + 2 ...	neu eingetragene Codes

Die Codes haben eine variable Codelänge von 3 bis 12 Bits. Sind alle Codes innerhalb der aktuellen Codelänge vergeben muss ein Clear Code ausgeführt werden und die neue Codelänge wird um eins erhöht. Zwischen den Clear Codes haben alle Code die selbe Länge. Um Füllbits zu vermeiden sind die Codes nacheinander in Bytes gepackt (siehe Abbildung 11).

	Byte 3	Byte 2	Byte 1
	76543210	76543210	76543210
...	eeeedddd	dcccccbb	bbbaaaaa

Abbildung 6: Codes in Bytes verpackt

Der eigentliche **LZW-Algorithmus** arbeitet wie folgt:

Encode:

```
Codetabelle initialisieren (jedes Zeichen erhält einen Code)
präfix = ” ”
while (Ende des Eingabedatenstroms noch nicht erreicht) {
     suffix := nächstes Zeichen aus dem Eingabedatenstrom
     muster := präfix + suffix
     if muster in Codetabelle then
          präfix := muster
     else
          muster in Codetabelle eintragen
          LZW-Code von präfix ausgeben
```

```
            präfix := suffix
}
if präfix nicht leer then
      LZW-Code von präfix ausgeben
```

	präfix	muster	suffix	Eingabedatenstrom	LZW Code	Ausgabe
0		A	A	ABCABCABCD		
1	A	AB	B	BCABCABCD	4:AB	0 (A)
2	B	BC	C	CABCABCD	5:BC	1 (B)
3	C	CA	A	ABCABCD	6:CA	2 (C)
4	A	AB	B	BCABCD		
5	AB	ABC	C	CABCD	7:ABC	4 (AB)
6	C	CA	A	ABCD		
7	CA	CAB	B	BCD	8:CAB	6 (CA)
8	B	BC	C	CD		
9	BC	BCD	D	D	9:BCD	5 (BC)
10	D					3 (D)

Initianlisierung:

Codetabelle: 0:A, 1:B, 2:C, 3:D

Abbildung 7: Beispiel für den LZW Encode Vorgang

Decode:

```
Codetabelle initialisieren (jedes Zeichen erhält einen Code)
präfix = " "
while (Ende der Daten noch nicht erreicht) {
      lese LZW-Code
      muster := dekodiere (LZW-Code)
      gebe muster aus
      neuer LZW-Code := präfix + erstes Zeichen von muster
      präfix := muster
}
```

Code	präfix	muster	neuer Code	Ausgabe
0		A		A
1	A	B	4 = AB	B
2	B	C	5 = BC	C
4	C	AB	6 = CA	AB
6	AB	CA	7 = ABC	CA
5	CA	ABC	8 = CAB	BC
3	ABC	D	9 = BCD	D

Initianlisierung:

Codetabelle: 0:A, 1:B, 2:C, 3:D

Abbildung 8: Beispiel für den LZW Decode Vorgang

Das Software-Patent der Firma Unisys beschränkt sich auf den Encode-Vorgang. Programme die diesen Algorithmus verwenden sind von den Gebührenforderungen betroffen. Der Decode-Vorgang bleibt Gebührenfrei.

Terminator

Dieses letzte Byte einer GIF Datei besteht aus dem hexadezimal Wert „3B“ (ASCII: „ ; “). Es zeigt das Ende der gesamten Datei an.

Erweiterungsblöcke

In der Version 89a wurden 4 verschiedene Erweiterungsblöcke definiert, die nach folgendem Schema aufgebaut sind (siehe Abbildung 12).

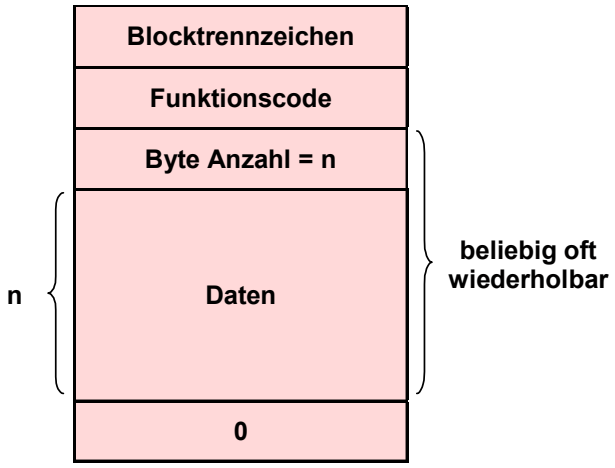


Abbildung 9: Aufbau der GIF Erweiterungsblöcke

Als Blocktrennzeichen wird der hexadezimal Wert „21“ (ASCII: „!“) verwendet. Anschließend folgt der Funktionscode, der die Art des jeweiligen Erweiterungsblockes bestimmt. Das nächste Byte gibt die Länge der folgenden Daten in Byte an. Falls diese länger als 255 Byte sind, wird ein weiterer Datenblock angehängt. Das Ende eines Erweiterungsblockes wird mit einem Nullbyte gekennzeichnet.

Folgende Erweiterungsblöcke wurden in der Version 89a definiert:

Plain Text Extension: (01 hex)	kann beliebigen ASCII Test als Grafik auf der Bildfläche darstellen
Comment Extension: (FE hex)	Kommentare (z.B.: Autor ...) als ASCII Text mitspeichern aber nicht im Bild anzeigen
Graphic Control Extension: (F9 hex)	gibt an wie nach der Anzeige des folgenden Bilddatenblocks weiter verfahren werden soll (z.B.: Animationsgeschwindigkeit)
Application Extension: (FF hex)	anwendungsspezifische Daten für eigene Erweiterungsblöcke

Innerhalb des Datenfeldes gibt es für die einzelnen Erweiterungsblöcke definierte Strukturen, die in der Spezifikation genauer nachgelesen werden können.

Beispiele

Abbildung 10: Beispiel einer GIF Datei

47 49 46 38 39 61	Gif Signatur = „GIF89a“	
10 00 10 00 A2 00 00	Gesamt Breite (hier 16) Gesamt Höhe (hier 16) = 1 010 0 010 Background Color Pixel Aspect Ratio	Farbtiefe (hier 3) Farbpalette sortiert (hier nein) Farbtiefe der Ursprungsgrafik (hier 3) globale Farbtabelle (hier ja)
00 00 00 00 00 FF 00 FF 00 00 FF FF FF 00 00 FF 00 FF FF FF 00 FF FF FF	globale Farbtabelle in RGB Werten	
2C 00 00 00 00 10 00 10 00 00	Image Seperator Image Left (hier 0) Image Top (hier 0) Image Width (hier 16) Image Height (hier 16) = 0 0 0 00 000	Farbtiefe der LCM Farben sortiert (hier nein) Interlaced abgespeichert reservierte Bits lokale Farbtabelle (hier nein)
04 46 10 C8 49 AB 05 2A AB 4B B5 E6 98 F7 59 A2 48 2E A5 B3 55 4B EB 39 70 95 B4 ED F3 C0 30 33 25 7C 6D DF 0E 86 50 C2 2B FE 1E C2 24 B1 98 68 D8 1A C9 21 80 D9 6C 58 A3 BA 25 CF 7A 55 EE 98 5D 2F A5 28 11 83 38 11 00	LZW komprimierte Bilddaten	
3B	Terminator	

PNG

PNG wurde in erster Linie entwickelt um ein Grafikformat zu haben, welches nicht von Patentrechten und den damit verbundenen Lizenzgebühren betroffen ist. So soll es auch eine Ausweichmöglichkeit für GIF bieten. Es wurden aber noch zahlreiche Features spezifiziert, die über die Spezifikation von GIF hinausreichen und den mit der Zeit veränderten Bedürfnissen und Möglichkeiten angepasst sein sollten.

Eigenschaften

Auch PNG ist ein für Datennetze optimiertes Raster-Grafikformat. Es deckt alle Eigenschaften von GIF ab, bis auf die Möglichkeit mehrere Einzelbilder (für Animationen) in einer Datei zu speichern. Dafür wurde das spezielle Format MNG (ehemals PNF) entwickelt welches PNG sehr ähnlich ist, jedoch zahlreiche Animationsfähigkeiten bietet.
Mit einer Farbtiefe von bis zu 48 Bit (RGB) soll es echte True Color Bilder ermöglichen. Ebenso sind aber auch Palletten-Bilder mit bis zu 8 Bit (256 Farben), sowie Graustufen-Bilder mit bis zu 16 Bit möglich.
Neben der transparenten Farbe wie in GIF ist bei PNG möglich echte Alphakanäle für eine Stufenlose Transparenz mitzuspeichern. Für jedes Pixel kann so ein Transparenzwert von 16 Bit gespeichert werden, der die Angabe über die Transparenz von komplett deckend bis komplett transparent macht.
Für die Komprimierung wird der LZ77 Algorithmus verwendet, welcher ebenfalls eine verlustfreie Komprimierung ermöglicht, und nicht von Lizenzforderungen betroffen ist.
Auch ein Interlaced Modus ist möglich, wird aber etwas anders als in GIF realisiert.
Das frei definierte Chunk Layout macht PNG leicht erweiterbar für individuelle Zwecke.

Dateiaufbau

Eine PNG Datei besteht aus mehreren einzelnen Chunks, von denen einige alle Decoder interpretieren können müssen („critical chunks“) und weitere die optional vorhanden sein können („ancillary chunks“) und falls sie nicht ausgewertet werden können eine Bildausgabe trotzdem möglich ist.

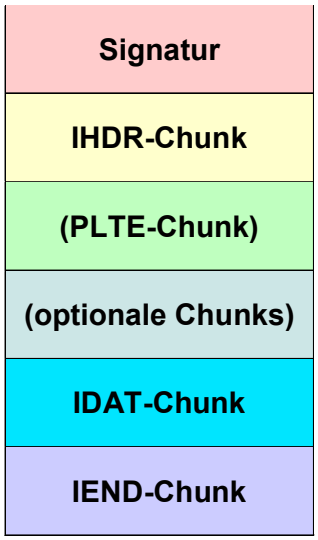


Abbildung 11: PNG Dateiaufbau

Auf jeden Fall vorhanden sein müssen, die Signatur, der Header Chunk (IHDR), der Data Chunk (IDAT) sowie der Trailer Chunk (IEND). Eventuell kann auch noch ein Palette Chunk (PLTE) sowie weitere optionale Chunks vorkommen. Diese optionalen Chunks müssen sich teilweise an bestimmten Stellen innerhalb der PNG-Datei befunden, in jedem Fall aber zwischen Header und Trailer.

Signatur

Jede PNG-Datei muss mit der 8 Byte langen Signatur beginnen. Anders als bei GIF dient diese nicht nur der Dateiart- und Versionsangabe, sondern hat ein paar Kontrollfunktionen integriert.

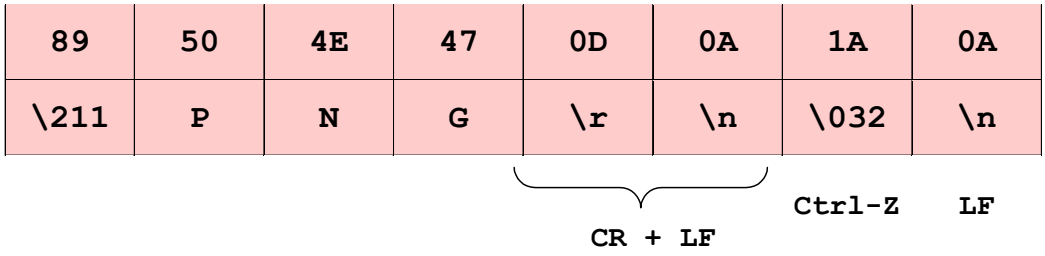


Abbildung 12: Aufbau der PNG Signatur

Das erste Byte testet, ob die Datei durch ein 8-Bit fähiges Datennetz gesendet wurde, ansonsten wäre schon dieses Byte falsch und die weitere Ausgabe kann abgebrochen werden. Anschließend folgen die drei ASCII Zeichen „PNG“, die die Dateiart klarstellen. Die nächsten beiden Bytes sind die Steuerzeichen Carriage Return (Wagenrücklaufsymbol) und Line Feed (Zeilenvorschubsymbol). Dieses Zeichen könnten auch in den Datenbytes vorkommen und müssen deshalb unverändert übertragen werden. Das 7. Byte stoppt die Ausgabe unter DOS, und das letzte Byte der Signatur überprüft nochmals ob auch einzelne Line Feed Symbole unverändert gesendet werden.

Chunk Layout

Bis auf die Signatur sind alle Blöcke (Chunks) nach der gleichen Grundstruktur aufgebaut.

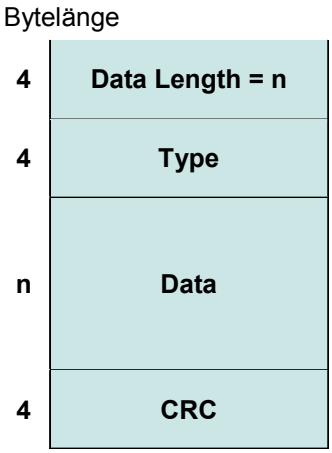


Abbildung 13: PNG Chunk Layout

Die ersten 4 Bytes geben die Länge des Chunks ohne Data Length, Typkennung und CRC-Prüfsumme in Bytes an. So kann ein eventuell für den Encoder unbekannter Block ohne Probleme übersprungen werden.
Anschließend folgt die ebenfalls 4 Byte lange Typkennung. Diese besteht in der Regel aus 4 ASCII Zeichen, wobei das 5. Bit jedes Zeichens für einige Zusatzparameter benutz wird, die nach folgender Konvention vergeben werden Dieses Bit entscheidet auch über die Groß-Kleinschreibung, was das Auswerten vereinfacht.

	Zeichen			
	1	**2**	**3**	**4**
Großbuchstabe	**critical**	**public**	**reserv.**	**not copy**
Kleinbuchstabe	**ancillary**	**privat**	**X**	**copy**

Abbildung 14: Konvention der Typkennung

Ist der erste Buchstabe ein Großbuchstabe, so handelt es sich um einen critical Chunk, ansonsten ist es ein anciallary Chunk. Der zweite Buchstabe gibt an ob der Block öffentlich oder privat ist. Das dritte Zeichen ist in der aktuellen Spezifikation noch unbenutzt und reserviert. Es sollte Momentan aber als Großbuchstabe vergeben werden. Der letzte Buchstabe gibt dem PNG Encoder darüber Auskunft, ob der Block vom Inhalt eines kritischen Blocks abhängt, und ohne Anpassung nicht einfach kopiert werden darf.
Nach der Typkennung folgen die eigentliche Daten, die in jedem Blocktyp anders definiert sein können.
Die letzten 4 Byte eines jeden Chunks, sind eine CRC-Prüfsumme über die Typkennung und die Datenbytes. Folgendes Prüfpolynom wird dabei verwendet:
$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$

Header Chunk IHDR

Der Start Block (Header Chunk) muss direkt an die Signatur anschließen. Er beinhaltet mehrere grundlegende Informationen über das PNG-Bild.

Bytelänge	
4	Data Length = 13
4	„IHDR“
4	Width
4	Height
1	Bit depth
1	Color Type
1	Compression method
1	Filter method
1	Interlace methode
4	CRC

Abbildung 15: Aufbau des Header Chunks (IHDR)

Die Typkennung „IHDR“ zeigt an, dass es sich um einen kritischen, öffentlichen Block handelt, der nicht ohne eventuelle Anpassung an veränderte Bilddaten kopiert werden darf.
Nach der Typkennung folgen Angaben über Bildbreite, Bildhöhe, Länge der Bitmuster (1,2,4,8 oder16 je nach Farbtyp) und Farbtyp. Der Farbtyp wird nach folgendem Muster angegeben:

	0	1
Bit 1	ohne Palette	mit Palette
Bit 2	Graustufen	RGB Farbe
Bit 3	kein Alpha Kanal	mit Alpha Kanal

Abbildung 16: Farbtyp-Byte Belegung

Desweiteren werden auch noch Angaben über die verwendeten Kompressionsmethode (bisher spezifiziert: 0 = Deflate-Algorithmus), über den verwendeten Filter (0 = adaptive Filterung) und die Interlaced Methode (0 = unverschachtelt; 1 = Adam 7).

Palette Chunk PLTE

Optional kann auch in PNG-Bildern eine Palette angegeben werden. Dieser Paletten Block (Palette Chunk) kann auch angegeben werden, wenn es sich um ein True Color Bild handelt. Die Palette kann damit als Angabe zur bestmöglichen Farbreduktion dienen, falls auf dem System nicht die erforderliche Farbtiefe zu Verfügung steht.

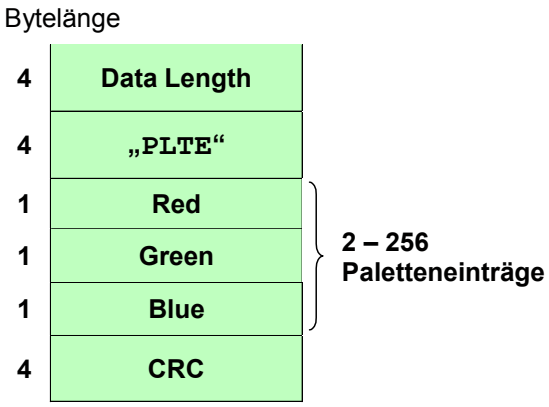


Abbildung 17: Aufbau des Palette Chunks (PLTE)

Innerhalb der Palette können 2 bis 256 RGB Farbwerte vorkommen. Reine Graustufenpaletten sind nicht möglich.

Data Chunk IDAT

Innerhalb des Daten Blocks (Data Chnuk) sind die eigentlichen Pixelwerte gespeichert. Bei Palettenbildern sind es Verweise auf den zugehörigen Paletteneintrag, ansonsten die tatsächlichen Farbwerte bzw. Graustufenwerte. Diese Pixelwerte sind mit dem LZ77-Algorithmus (siehe 3.2.10) gepackt.

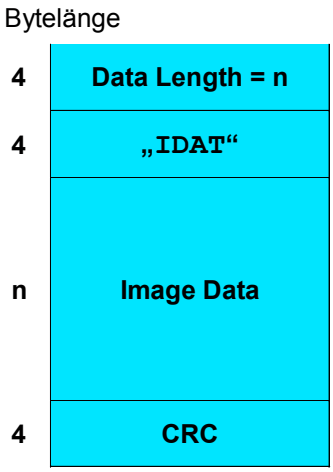


Abbildung 18: Aufbau des Data Chunks (IDAT)

Die einzelnen Pixelwerte werden in Bytes gepackt um keine Bits zu verschwenden.
Sind die Bilddaten länger als 2^32 Bytes werden die restlichen Bilddaten in einen weiteren Daten Block gepackt. Es können beliebig viele Daten Blöcke aufeinander folgen.

Trailer Chunk IEND

Der 12 Byte große Ende Block (Trailer Chunk) enthält keine Daten. Er dient nur dazu das Ende der PNG-Datei anzuzeigen.

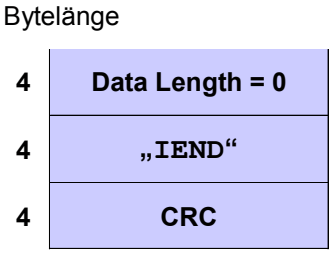


Abbildung 19: Aufbau des Trailer Chunks (IEND)

Ancillary Chunks

Neben den vier kritischen Blöcken gibt es auch noch eine Reihe von Zusatzblöcken (ancillary Chunks). Einige spezielle sind Teil der Spezifikation, aber es können nach dem bereits beschriebenen allgemeinen Chunk Layout auch beliebige eigene Chunks definiert und benutzt werden.

Folgende Liste zeigt ein paar der definierten ancillary Chunks:

• **Typkennung**	**Name**	**Beschreibung**
tIME	last modification	Datum der letzten Veränderung
bKGD	Background	Hintergrundfarbe (Paletteneintrag, Grau- oder RGB-Wert)
tRNS	Transparency	Alphawerte für einzelne Farben/Paletten
sBit	• Significant Bits	Bittiefe vor dem Speichern und nötige Umrechnung
gAMA	Image gama	Heiligkeitswerte beim Erstellen
cHRM	chromaticities	Chromaticity und White Point (RGB Spezifikation)
hIST	histogram	Häufigkeit der Farbwerte (nur bei Palette)
pHYS	physical pixel	• Abmessungen der Pixel
sCAL	Physical scale	Größe des abgebildeten Objekts (Karten, Medizin, ...)
tEXt	Textual Data	unkomprimierter Text (Autor, Copyright ...)
zTXt	compr. Text	komprimierter Text (für lange Texte)
gIFg	GIF Control Extension	GIF-Kontrollwerte

gIFx	GIF Aplication Extension	GIF-Erweiterungsblöcke

Weitere Zusatzblöcke wurden definiert und können genauer in der aktuellen Spezifikation nachgelesen werden.

Interlaced Modus

Für den Interlaced Modus ist momentan als einzige Methode das Adam 7 Verfahren spezifiziert.
Dieses Verfahren lädt das Bild in 7 Schritten nach folgenden Muster:

Das komplette Bild wird dazu in 8x8 Blöcke unterteilt. Innerhalb dieser Blöcke werden die Pixel in der nebenstehenden Reihenfolge geladen und gespeichert.
Das Adam 7 Verfahren liefert somit nach einer noch kürzeren Übertragungszeit wie bei dem Zeilenweisen Interlaced Modus aus GIF eine grobe Vorschau des Bildes.

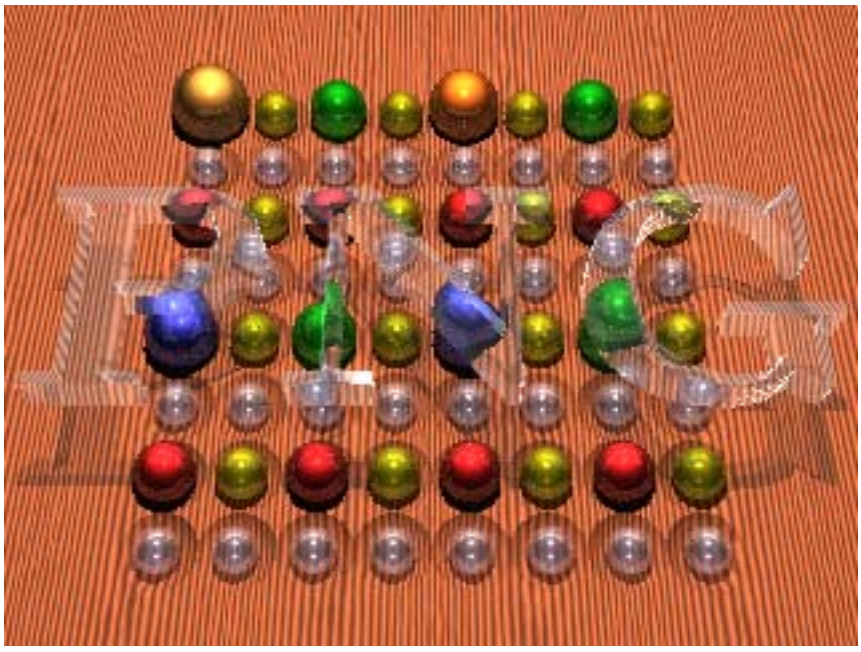


Abbildung 20: PNG Adam 7 Verfahren (Veranschaulichung aus der Spezifikation)

Filter

Noch vor dem Kodieren der Pixel mit dem LZ77-Algorithmus, können über die einzelnen Bildzeilen Filter angewendet werden, um die Kompressionsrate zu verbessern. Die Kompression wird effektiver wenn sie nicht über viele sehr unterschiedliche Pixelwerte angewendet wird, sondern statt dessen über die Differenz zu vorherigen Pixeln. So entstehen in vielen Fällen (z.B. bei Farbverläufen) viele gleiche Werte, In deren Folgen so auch größere wiederkehrende übereinstimmende Muster zu finden sind. Auch die Anwendung dieser Filter ist verlustfrei.
Welche Filtermethode in der Zeile verwendet wurde, wird in einem vor jeder Zeile mitgespeichertem Byte, dem Filtertypindikator angegeben.

In PNG sind folgende Filtermethoden definiert:

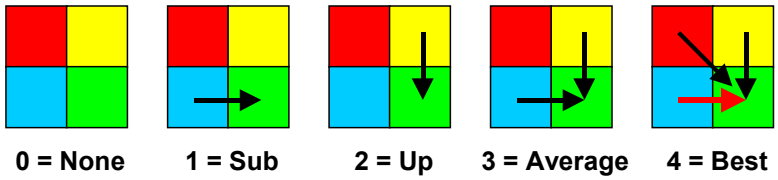


Abbildung 21: Filtermethoden

0 = keine Filterung
1 = Differenz zum links benachbarten Pixelwert
2 = Differenz zum darüber liegenden Pixelwert
3 = Durchschnitt aus dem linken und darüber liegenden Pixelwert
4 = kleinste Differenz zum linken, darüber oder links-darüber liegenden Pixelwert

LZ77-Algorithmus

In PNG wird der Deflate Algorithmus mit 32kBit Gleitfenster (Sliding Window) zur Komprimierung der Bilddaten verwendet. Dieser ist eine Variante des LZ77 Algorithmus, dem ältesten Kompressions Algorithmus der Lempel-Ziv Familie.
Er arbeitet ähnlich wie LZW nach dem Prinzip wiederkehrende Zeichenketten zu erkennen, ersetzt diese aber nicht durch Codes, sondern speichert einen Zeiger auf die zuletzt gefundenen Position dieser Übereinstimmung.

Encode:

```
Kodierungsposition := Anfang des Eingabedatenstroms
while (Ende des Eingabedatenstroms noch nicht erreicht) {

      finde die längste Übereinstimmung im Datenfenster mit
      der an der Kodierungsposition beginnenden Zeichenkette

      gebe den Zeiger auf die Übereinstimmung Datenfenster und
      das erste nicht passende Zeichen im Vorschaupuffer aus

      Kodierungsposition += Länge der Übereinstimmung + 1
}
```

Pos	Sliding Window	Vorschaupuffer	Pos, ÜS	Ausgabe
0		AABCAAABC		(0,0)A
1	A	ABCAAABC	0,A	(0,1)B
3	AAB	CAAABC		(0,0)C
4	AABC	AAABC	0,AA	(0,2)A
7	AABCAAA	BC	3,BC	(2,1)C

Abbildung 22: Beispiel für den LZ77 Algorithmus

Beispiele

Abbildung 23: Beispiel einer PNG Datei

89 50 4E 47 0D 0A 1A 0A	• Signatur

00 00 00 0D 49 48 44 52 00 00 00 10 00 00 00 10 08 03 00 00 00 28 2D 0F 53	Data Length = 13 Byte “IHDR” Breite = 16 Pixel Höhe = 16 Pixel Länge der Bitmuster = 8 0011 → kein Alpha-Kanal, Farbbild, mit Palette 0: Deflate Komprimierung 0: adaptive Filterung 0: unverschachtelt CRC	IHDR
00 00 00 07 74 49 4D 45 07 D2 0C 0A 17 19 3B 0A 0D EF 28		tiME
00 00 00 09 70 48 59 73 00 00 0B 12 00 00 0B 12 01 D2 DD 7E FC		pHYs
00 00 00 04 67 41 4D 41 00 00 B1 8F 0B FC 61 05		gAMA
00 00 00 18 50 4C 54 45 00 00 00 FF 00 00 00 00 FF FF 00 FF FF FF FF FF FF 00 00 FF FF 00 FF 00 09 B4 2B E0		PLTE
00 00 00 4E 49 44 41 54 78 DA 6D CA 5B 0E 00 21 08 43 D1 82 3C F6 BF 63 51 E2 08 66 FA 77 4F 0A FC CF 63 6F 57 71 EF E2 FE 88 6B A6 1D 50 DD 62 66 D9 14 A0 22 D1 C6 BB 89 56 2F 61 E6 04 A2 05 CC 05 86 C8 E8 10 3B 80 0A C0 BD 7C 87 24 E0 66 DF 04 5A B2 02 38 83 63 D1 5B		IDAT
00 00 00 00 49 45 4E 44 AE 42 60 82		IEND

Browser Kompatibilität

PNG Bilder werden zwar von allen aktuellen Internet-Browsern dargestellt, aber diese Darstellung geschieht verschieden gut. Besonders der Alphakanal wird nicht immer richtig interpretiert.

Abbildung 24: MS Internet Explorer 5.5 und 6.0

Abbildung 2513: Netscape Navigator 4.75

Abbildung 26: Netscape Navigator 6.1 und 7.0, Opera 6.05, Mozilla1.1

Zusammenfassung

Das Grafikformat PNG sollte ein Ersatz für das von Lizenzgebühren betroffene GIF sein. Ersetzen konnte es das sehr weit verbreitete GIF nicht, aber es ist ein sehr gute Alternative, durch die zahlreichen Funktionen die über die Möglichkeiten von GIF hinausgehen. Besonders die verlustfreie Kompression auch von True Color Bildern sowie die Alphakanäle bieten eine gute Möglichkeit Fotos ohne Kompressionsverluste wie sie z.B. bei JPG auftreten abzuspeichern. Die Dateigröße ist dabei meistens größer als bei JPG aber doch geringer als TIF.
Dass PNG nicht GIF einfach ersetzten sollte, sondern ein neues leistungsfähigeres Grafikformat sein sollte wird auch durch die interne Namenserklärung „PiNG is not GIF“. Die Aussprache als „PiNG“ steht aber mittlerweile offiziell in der Spezifikation.

Abkürzungen

GIF	Graphics Interchange Format
LZ77	Lempel-Ziv-(Kompression) von 1977
LZW	Lempel-Ziv-Welch-(Kompression)
MNG	Multiple Network Graphics (ehemals PNF)
PNF	Portable Network Frame (später MNG)
PNG	Portable Network Graphics („PiNG“)
RGB	Rot-Grün-Blau Farbwerte

Literatur

[1] CompuServe Inc.: GIF 87a Specification; 1987; http://www.wotsit.org

[2] CompuServe Inc.: GIF 89a Specification; 1990; http://www.w3.org/Graphics/GIF/spec-gif89a.txt

[3] Thomas W. Lipp: Das Graphics Interchange Format (GIF); in: Grafikformate; 1997, Deutschland; S. 277 ff. (ISBN: 3-86063-391-0)

[4] Andreas Kunz: Das Grafikdateiformat GIF; in: Proseminar Redundanz; 1998/1999; Universität Karlsruhe; http://goethe.ira.uka.de/seminare/redundanz/vortrag10/

[5] Maximilian Hrabowski: Lempel Ziv; in: Proseminar Redundanz; 1998/1999; Universität Karlsruhe; http://goethe.ira.uka.de/seminare/redundanz/vortrag05/

[6] World Wide Web Consortium: PNG Specification Version 1.0; 1996; http://www.w3.org/Graphics/PNG/

[7] Greg Roelofs: PNG home page; 2003; http://www.libpng.org/pub/png/

[8] Max Völkel: Das Grafikdateiformat PNG; in: Proseminar Redundanz; 1998/1999; Universität Karlsruhe; http://goethe.ira.uka.de/seminare/redundanz/vortrag12/

JPEG (JFIF)

von

Robert Becker

Einleitung

JPEG ist die Abkürzung für „Joint Photograph Experts Group“, ein Komitee, das sich 1986 zum ersten Mal getroffen hat und sich aus Mitgliedern der Organisationen ISO und CCITT zusammensetzt. 1992 haben sie den ersten JPEG Standard veröffentlicht. Etwa vor einem Jahr wurde der neue Standard JPEG2000 vorgestellt. Auf diesen wird später noch eingegangen. Vorher wird grundlegend der JPEG Standard erklärt, auf seine Funktionsweise und seine Vorgaben eingegangen und dann eine, bzw. die Implementierung des JPEG Standards: JFIF erläutert. Am Ende folgen Beispiele zur Verdeutlichung des vorher theoretisch Erläuterten.

Der JPEG Standard

Die Ziele des JPEG Standards hat das Komitee wie folgt definiert. Es sollte ein Kompressionsverfahren ohne Datenverlust geben und eines mit Datenverlust, aber einstellbarer Kompressions- bzw. Verlustrate. Der Algorithmus sollte mit vertretbarer Komplexität arbeiten und auf alle unbewegten Farbbilder anwendbar sein, ohne Beschränkung der Farbtiefe.
Den Standard, der 1992 das Ergebnis dieser Arbeit darstellt, kann man wie folgt in 6 Unterpunkte aufteilen:

1. Konvertierung des Bildes in den YUV-Farbraum
2. Farb-Subsampling, Aufteilung in Blöcke
3. Diskrete Cosinustransformation (DCT)
4. Quantisierung der DCT-Koeffizienten
5. Serialisierung der Koeffizienten in ZickZack-Anordnung
6. Codierung der Koeffizienten
    a. Huffmancodierung
    b. Arithmetische Codierung

Der YUV-Farbraum

In einem Farbraum gibt es wie in einem dreidimensionalen Vektorraum 3 Basen die den Raum aufspannen. Im Vektorraum kann damit jeder Punkt erreicht werden, im Farbraum jede Farbe dargestellt. Die bekanntesten Farbräume sind RGB (Rot, Grün, Blau) und CMY(K) (Türkis, Magenta, Gelb, (Schwarz)). Diese Art der Darstellung wird Farbmodell genannt. Eine andere Möglichkeit ist das Helligkeit-Farbigkeit-Modell. Dieses Modell macht sich die

Fähigkeit des menschlichen Auges zu Nutze, besser Helligkeitsunterschiede wahrnehmen zu können als Farbunterschiede. So können wir als Menschen in Dunklen Räumen oder bei Nacht oft nur noch schemenhafte Konturen erkennen, aber nicht mehr spezielle Farben. Bei YUV steht das Y für die generelle Luminanz (Helligkeit des Bildes), U stellt die Farbabweichung in Richtung Rot dar, V die in Richtung Blau. Abbildung 1, 2 und 3 zeigen die Konversion von RGB in YUV und zurück. Bei den Formeln wird jeweils ein dreidimensionaler Vektor mit einer Konversionsmatrix multipliziert und als Ergebnis erhält man den Ergebnisvektor im anderen Modell.

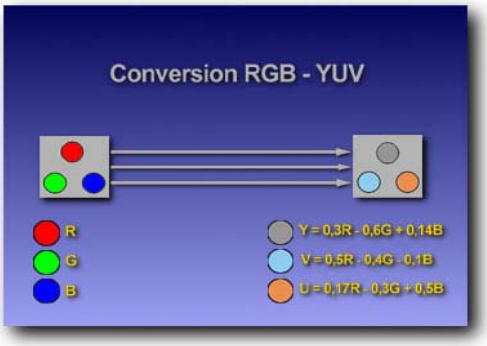


Abbildung 14

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{pmatrix} \times \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Abbildung 15: Formel RGB → YUV

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0.0 & 1.402 \\ 1.0 & -0.34414 & -0.71414 \\ 1.0 & 1.772 & 0.0 \end{pmatrix} \times \begin{pmatrix} Y \\ U \\ V \end{pmatrix}$$

Abbildung 3: Formel YUV → RGB

Der JPEG-Standard lässt zwar die Wahl des Farbmodells frei. Jedoch in der noch später beschriebenen und fast ausschliesslich verwendeten Implementierung JFIF hat man sich auf das Helligkeit-Farbigkeit-Modell YUV festgelegt.

Farbsubsampling / Aufteilung in Blöcke

Die Pixelwerte eines Bildes werden nun in 8x8 Blöcke unterteilt. Aufgrund der bereits angesprochenen besseren Helligkeitswahrnehmung des menschlichen Auges werden nun mehr Informationen zu der Helligkeit des Bildes als zu der Farbigkeit des Bildes gespeichert. So speichert man in diesem 8x8 Block alle 64 Helligkeitswerte, aber nur 16 Farbigkeitswerte. Dazu fasst man immer einen 2x2 Pixelblock zu einer Farbe zusammen.

Diskrete Cosinus Transformation (DCT)

Die diskrete Cosinus Transformation (DCT) ist mit der diskreten Fourier Transformation verwandt und stellt eine wesentliche Eigenschaft des JPEG-Standards dar. Die DCT dient zur Berechnung des Frequenzspektrums eines diskreten oder kontinuierlichen, zeitlichen oder räumlichen Signals. Sie lässt sich sowohl vorwärts (DCT zur Komprimierung) als auch rückwärts (IDCT zur Dekomprimierung) durchführen. Im Idealfall lautet die mathematische Formel hierfür: IDCT(DCT(x)) = x.

$$DCT(i,j) = \frac{1}{\sqrt{2N}} C(i) C(j) \sum_{x=0}^{N-1} \sum_{y=0}^{N-1} pixel(x,y) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

Abbildung 4: Diskrete Cosinus Transformation

$$Pixel(x,y) = \frac{1}{\sqrt{2N}} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} C(i) C(j) DCT(i,j) \cos\left[\frac{(2x+1)i\pi}{2N}\right] \cos\left[\frac{(2y+1)j\pi}{2N}\right]$$

$$C(t) = \frac{1}{\sqrt{2}} \quad wenn \quad t = 0 \quad sonst \quad 1$$

Abbildung 16: Inverse Diskrete Cosinus Transformation

Die 64 Werte der Eingangsmatrix M, die durch x und y räumlich angeordnet sind, werden durch DCT in 64 vorgegebene, orthogonale (vertikale) Basissignale zerlegt. Die einzelnen Basissignale haben alle eine eigene "räumliche Frequenz". Zusammengenommen ergeben die 64 Basissignale wieder das gesamte Frequenzband. Die Amplituden der Basissignale werden in Abhängigkeit von den Eingangssignalen berechnet, so dass die Multiplikation der Amplitude mit dem Basissignal wieder das Ausgangssignal der Matrix S ergibt. Die Amplituden werden beim JPEG-Standard auch DCT-Koeffizienten genannt. Sie beschreiben das Spektrum des Eingangssignals und sind dadurch eindeutig festgelegt. Das Element DCT(0,0) der Matrix beinhaltet den Koeffizienten, bei dem in beiden Richtungen die Frequenz gleich Null ist. Es wird DC-Koeffizient oder auch Gleichstromkoeffizient genannt, während die anderen 63 Amplituden als AC-Koeffizienten bzw. Wechselstromkoeffizienten bezeichnet werden.

Vorausgesetzt, daß die Eingangsdaten nicht stark und unregelmäßig schwanken, sind die meisten Informationen in den Koeffizienten der niedrigen Frequenzen enthalten. Daher

verfügt der DC-Koeffizient über den größten Informationsgehalt. Für Echtfarbenbilder ist diese Voraussetzung i.d.R. erfüllt, da hier häufig Farbverläufe und weiche Übergänge vorliegen. Hier sind relativ viele AC-Koeffizienten 0 oder annähernd 0 sind und können daher später leicht zusammengefasst werden.

Theoretisch ist es möglich, daß die Eingangsdaten der Matrix M exakt aus den Spektraldaten der Ausgangsmatrix S wiederhergestellt werden können, wenn die DCT und die IDCT mit entsprechend hoher Genauigkeit ausgeführt werden. Die DCT ist also mathematisch gesehen verlustfrei. Praktisch tritt dieser Idealfall jedoch nicht auf, da die DCT Gleichung transzendente Funktionen enthält, die nur mit endlicher Genauigkeit berechnet werden können.

Abbildung 17: Das Bild

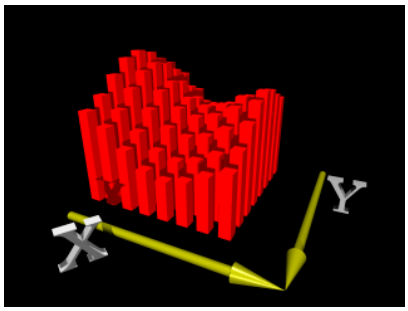


Abbildung 6: DCT Darstellung des Bildes

120	108	90	75	69	73	82	89
127	115	97	81	75	79	88	95
134	122	105	89	83	87	96	103
137	125	107	92	86	90	99	106
131	119	101	86	80	83	93	100
117	105	87	72	65	69	78	85
100	88	70	55	49	53	62	69
89	77	59	44	38	42	51	58

Abbildung 8: Pixelwerte des Bildes

700	90	100	0	0	0	0	0
90	0	0	0	0	0	0	0
-89	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Abbildung 9: DCT-Werte des Bildes

Quantisierung der DCT-Koeffizienten

Nach der DC Transformation werden die Koeffizienten quantisiert. Das heisst, dass die Signale nach ihrer Größe und Bedeutung in Stufen unterteilt werden. Je mehr Quantisierungsstufen dabei gewählt werden, umso weniger Quantisierungsverzerrungen entstehen hier. In einer Quantisierungsmatrix werden für jeden DCT Wert unterschiedlich starke Stufen eingestellt. Dabei wird der DC Wert oft am wenigsten und die unteren AC-Werte am

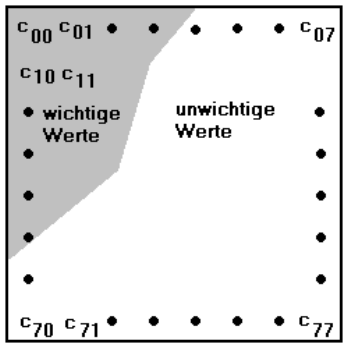


Abbildung 10: Quantisierungsmatrix

meisten verzerrt, da sie nicht so bedeutend sind und Veränderungen in diesem Bereich dem menschlichen Auge gar nicht oder kaum auffallen. Der quantisierte DCT Wert wird nun so erreicht indem man den DCT Wert durch den Wert in der Quantisierungsmatrix teilt und danach rundet. Um bei der Dekomprimierung wieder auf den DCT Wert zu kommen, muss einfach nur der quantisierte Wert mit dem entsprechenden Wert aus der Quantisierungsmatrix multipliziert werden.

$$\overline{DCT}(i,j) = Integer - Round\left(\frac{DCT(i,j)}{Q(i,j)}\right)$$

Abbildung 11: Quantisierung der DCT Werte

$$DCT(i,j) = \overline{DCT}(i,j) \cdot Q(i,j)$$

Abbildung 12: Rückwärts-Quantisierung

Hier entsteht Verlust. Durch ungenaue Quantisierungsstufen und das darauffolgende Runden werden die DCT Werte hier abgeschnitten und können i.d.R. nicht wieder zu ihrem ursprünglichen Wert zurückgeführt werden. Jedoch können die Quantisierungsmatrizen auch selbst individuell gewählt und erstellt werden, so dass man hier mehr oder weniger Verlust zulassen kann.

Serialisierung der Koeffizienten in ZickZack-Anordnung

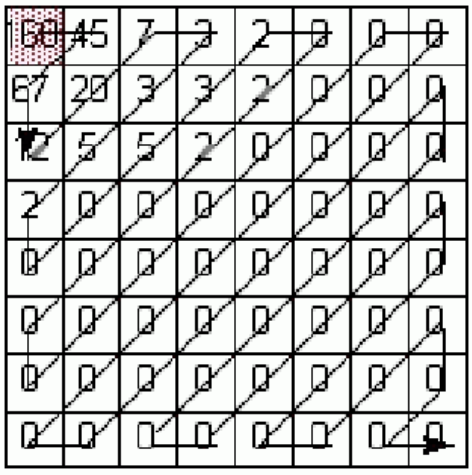


Abbildung 13: Serialisierung der Koeffizienten

Jetzt müssen die Daten in einen linearen Strom gebracht werden, d.h. die Werte aus der Matrix müssen serialisiert werden. Aufgrund der Gewichtung durch die Quatisierungsmatrix ist hier quasi schon eine Reihenfolge vorgegeben. Die wichtigen Werte stehen links oben, die unwichtigen rechts unten. Wir wollen um am Ende gut zusammenfassen zu können (RLE) möglichst alle 0 Werte in einer Reihe haben. Also fangen wir links oben an und gehen im ZickZack Muster durch die Matrix bis wir rechts unten angelangt sind. Der DC-Koeffizient wird bei dieser Serialisierung als Differenz zum DC-Wert des vorhergehenden 8x8 Blocks gespeichert.

Haben wir die Daten in einen linearen Strom gebracht, können wir diese mit RLE zusammenfassen. Dabei kommt uns nun die vermeintlich große Anzahl Nullen zu Gute.

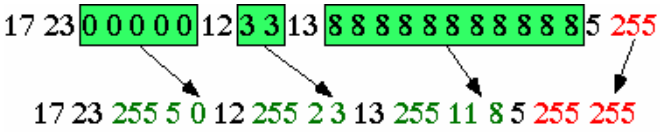


Abbildung 14: Run Length Encoding (RLE)

Codierung der Koeffizienten

Huffmancodierung

Die Huffmancodierung erstellt einen Huffman Baum und eine dazugehörige Huffmantabelle. Diese muss mit übertragen werden. In einem Huffman Baum werden Werte nach ihrer Häufigkeit sortiert abgespeichert. Je tiefer im Baum ein Wert steht, desto seltener kommt er vor. Das Blatt, das im Baum die geringste Tiefe hat, beinhaltet den am häufigsten vorkommenden Wert.

Arithmetische Codierung

Die Arithmetische Codierung erzielt im Gegensatz zur Huffmancodierung 5%-10% bessere Ergebnisse bei der Komprimierung. Jedoch ist die arithmetische Codierung auch wesentlich komplexer und benötigt somit wesentlich mehr Rechenleistung. Hinzu kommt, dass arithmetische Codierungsverfahren oft lizenzpflichtig sind. Besonders aufgrund dieses letzten Punktes hat man sich in der JFIF Implementierung für die Huffmancodierung entschieden. Der JPEG Standard lässt jedoch an dieser Stelle die Wahl des Codierungsverfahrens frei.

Operationsmodi

JPEG Dateien können in drei unterschiedlichen Modi abgespeichert werden. Dies beeinflusst hauptsächlich wie sich das Bild aufbaut und komprimiert wird.

Sequential Mode

In diesem Modus werden die Bilddaten einfach sequentiell von links oben bis rechts unten codiert. Dies liefert die beste Komprimierung und ist am einfachsten zu implementieren. Jedoch ist das ganze Bild erst sichtbar, wenn die gesamte Datei gelesen wurde.

Progressive Mode

In diesem Modus werden die Bilddaten in mehreren Durchgängen codiert. Die Koeffizienten mit den niedrigsten Frequenzen werden dabei zuerst codiert. Dadurch erhält man schon früh eine schemenhafte Vorschau auf das ganze Bild. Danach werden die Koeffizienten immer genauer codiert bis das ganze Bild komplett codiert wurde. Hierbei ist die Komprimierung etwas schlechter und die Implementierung auch schwerer, jedoch ermöglicht einem die „Vorschaufunktion“ besonders bei geringen Bandbreiten einen angenehmen Komfort.

Hierarchical Mode

In diesem Modus wird ein Bild in mehreren Auflösungen gespeichert. Es gibt sozusagen mehrere „Thumbnails“ eines Bildes die sich mit immer größeren Auflösungen dem Original annähern. Beim Laden eines Bildes wird mit der geringsten Auflösung angefangen und hierarchisch bis zum Original immer nachgeladen. Falls der Nutzer schon früh merkt, dass das Bild nicht das Gewünschte ist, kann er komfortabel abbrechen und zum Nächsten weitergehen.

Zusammenfassung

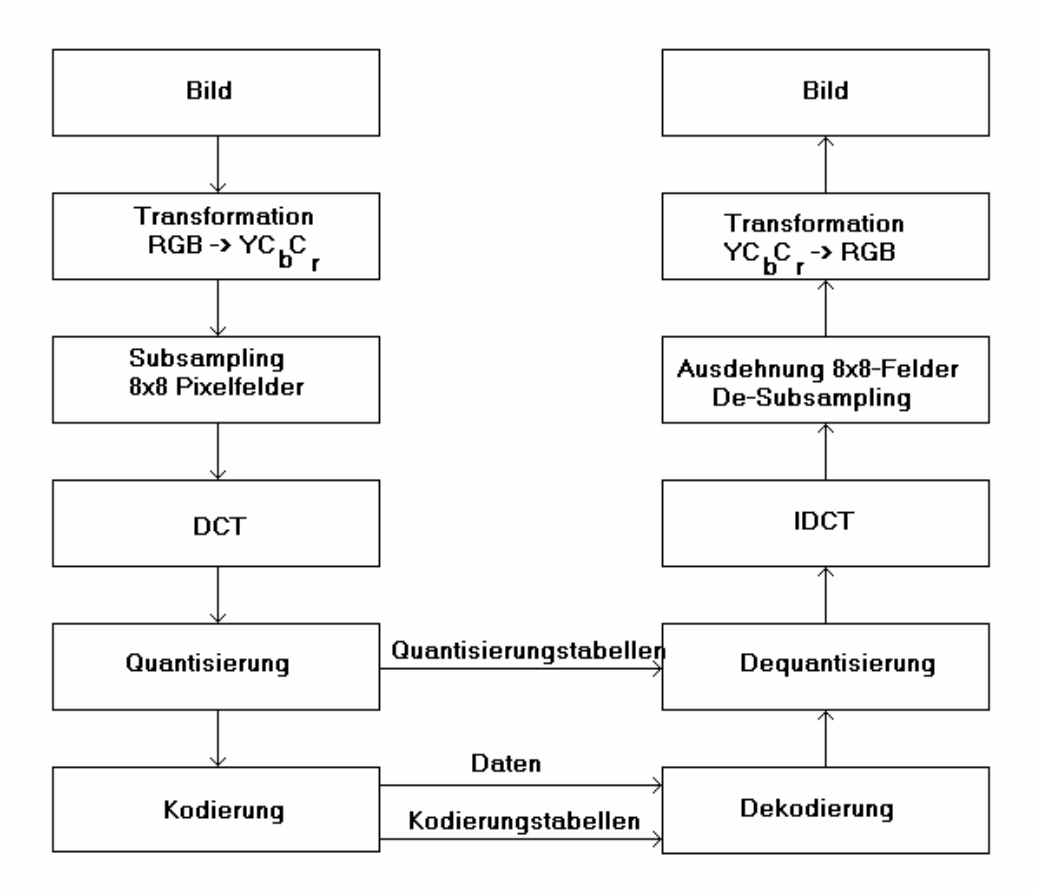


Abbildung 15: Zusammenfassung JPEG Kompression und Dekompression

Zusammengefasst sieht man hier noch einmal den kompletten Weg, den die Bilddaten bei der Komprimierung und Dekomprimierung durchlaufen. Dabei ist dieser auf den ersten Blick erst mal symmetrisch. Beachtet man jedoch dass bei der Kompression der Huffmanbaum, bzw. die Huffmantabelle erstellt werden muss, die dann bei der Dekompression nur gelesen werden muss, könnte man hier eine leichte Asymmetrie anmerken. Bei heutigen Rechenleistungen fällt dieses jedoch nicht mehr ins Gewicht und man kann mehr oder weniger von einem symmetrischen Kompressionsverfahren sprechen.

Die JFIF Implementierung

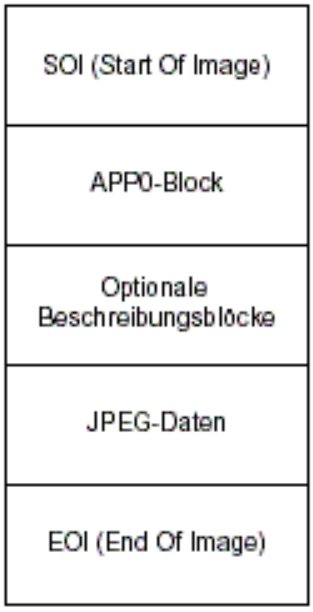


Abbildung 16: JFIF-Aufbau

JFIF bedeutet JPEG File Interchange Format. Als Implementierung des JPEG Standards muss diese sich in bestimmten Punkten festlegen. So hat man sich in der JFIF Implementierung für den YUV-Farbraum, sowie die Huffmancodierung entschieden. Eine solche Datei besteht aus den JPEG Daten, sowie den Informationen die zum Entpacken notwendig sind. Einzelne Teile der Datei sind in Blöcke unterteilt. Jeder Block hat eine bestimmte Anfangsmarkierung.

Start of Image → 0xFFD8

Application → 0xFFE0

Optionale Beschreibungsblöcke → 0xFF** (je nach Beschreibungsblock)

JPEG-Daten → 0xFFDA

End of Image → 0xFFD9

Jede Markierung hat 2 Byte (Hex) . Das Erste ist immer FF, das Zweite immer größer 0, da FF00 für den Wert FF steht.

Start of Image (0xFFD8)

Markiert den Anfang der Datei. Eine weitere Funktion hat dieser Block nicht.

Application (0xFFE0)

Enthält die Kennung (z.B. JFIF), sowie die Version, Punktdichte und ein Vorschaubild.

Anzahl Bytes	Inhalt
2	Länge der Markierung
1	4-Bit Präzision 4-Bit Nummer
64/128	Daten der Quantisierungstabelle

Anzahl Bytes	Inhalt
2	Länge der Markierung
1	Index
16	i.Byte = Anzahl Codes aus i Bits
N	Inhalte, n=Summe der vorhergehenden 16 Byte

Optionale Beschreibungsblöcke

Abbildung 17: Quantisierungstabelleneintrag

DQT - Define Quantisation Table (0xFFDB)

Hier werden die Quantisierungstabellen eingetragen werden. Für jede Quantisierungstabelle muss dabei ein eigener Eintrag erstellt werden. Diese wird mit der Präzision zusammen eingetragen.

Abbildung 18: Huffmantabelleneintrag

DHT - Define Huffman Table (0xFFC4)

Hier werden die Huffmancodierungstabelle eingetragen. Für jede Huffmanntabelle muss hier ebenfalls ein eigener Eintrag erstellt werden. Die Zählwerte geben an, wieviele Codes mit i Bit vorkommen.

SOF - Start Of Frame (0xFFC)

In diesem Block sind die Rahmendaten zu dem Bild gespeichert. Dazu zählen z.B. die Bildhöhe und die Bildbreite, sowie die einzelnen Komponenten und die dazugehörigen Quantisierungstabellen.

JPEG Daten (SOS - Start Of Scan (0xFFDA) **)**

Hier beginnt der Bitstrom.

Anzahl Bytes	Inhalt
2	Länge der Markierung
1	Anzahl der Komponenten
2	Daten der Komponenten (mehrfach)

Anzahl Bytes	Inhalt
1	Nummer der Komponente
1	4-Bit Nummer DC-Tabelle, 4-Bit AC-Tabelle

Abbildung 19: Blöcke für die JPEG-Daten

EOI - End Of Image (0xFFD9)

Hier endet die Datei.

Der Aufbau

1. SOI
2. APP0 Länge, Kennung, ...
3. DQT Länge, Präzision, ...
4. SOF0 Länge, Genauigkeit, ...
5. DHT Länge, Index, ...
6. SOS Länge, Bitstrom, ...
7. EOI

JPEG (JFIFs) erstellen

Beim Erstellen von JPEG(eigentlich JFIF)-Dateien kann man nun die Stärke der Quantisierung wählen, d.h. die Kompressionsstufe ist frei wählbar. Zudem kann man angeben, welcher Operationsmodus verwendet werden soll (z.B. Progressiver Bildaufbau). Manche Programme geben einem zudem die Möglichkeit zwischen unterschiedlichen YUV-Modellen zu wählen und bieten Vorschaubilder für die gewählte Option an.

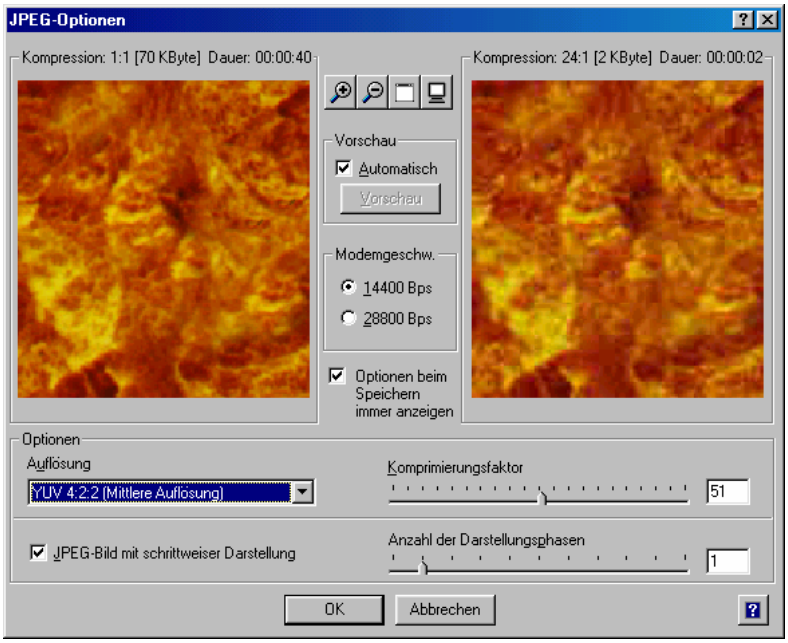


Abbildung 20: JPEG-Optionen für Komprimierung

JPEG2000 – Ein Ausblick

Das JPEG Komitee hat vor kurzem diesen neuen Standard vorgestellt. JPEG2000 soll dabei alte Probleme von JPEG beheben (wie z.B. die Artefaktbildung) und noch effizienter komprimieren.

Hier die Neuerungen zu JPEG:

- benutzt Diskrete Wavelet Transformation, in der ein Bild durch Tiefpassfilterung in ein Hochpassbild und mehrere Tiefpassbilder transformiert wird.

- kann 4 verschiedene „Progressive Modes“. Das Bild also progressiv nach Qualität, Auflösung, Position oder Farbe aufbauen.
- Benutzt arithmetische Codierung, anstatt wie bisher Huffmancodierung.
- Unterteilt das Bild nicht mehr in Blöcke, dadurch wurde der Artefaktbildung entgegengegangen. Das Bild wird jetzt im Ganzen codiert.
- Beinhaltet eine automatische Fehlerkorrektur für die Datenübertragung.

Der neue JPEG2000 Standard bringt eindeutig einige Neuerungen mit sich. Besonders da sich der alte JPEG Standard noch stark nach den damaligen schwachen Rechnerleistungen der Desktop-Computer gerichtet hat, bedurfte es eines neuen Standards, der veränderte Prioritäten setzt. JPEG2000 benutzt demzufolge komplexere Algorithmen, wie z.B. die arithmetische Codierung im Gegensatz zur Huffman Codierung bei JPEG, und erzielt damit bessere Erfolge bei der Kompression. Ebenso wird das Bild ab JPEG2000 im Ganzen kodiert. Das mag Rechnerseitig mehr Aufwand sein, bringt aber für die Bildqualität enorme Vorteile, da besonders bei hohen Kompressionsstufen die Artefaktbildung vermieden wird. Um bei geringer Datentransferrate schon frühzeitig einen Überblick über ein zu übertragenes großes

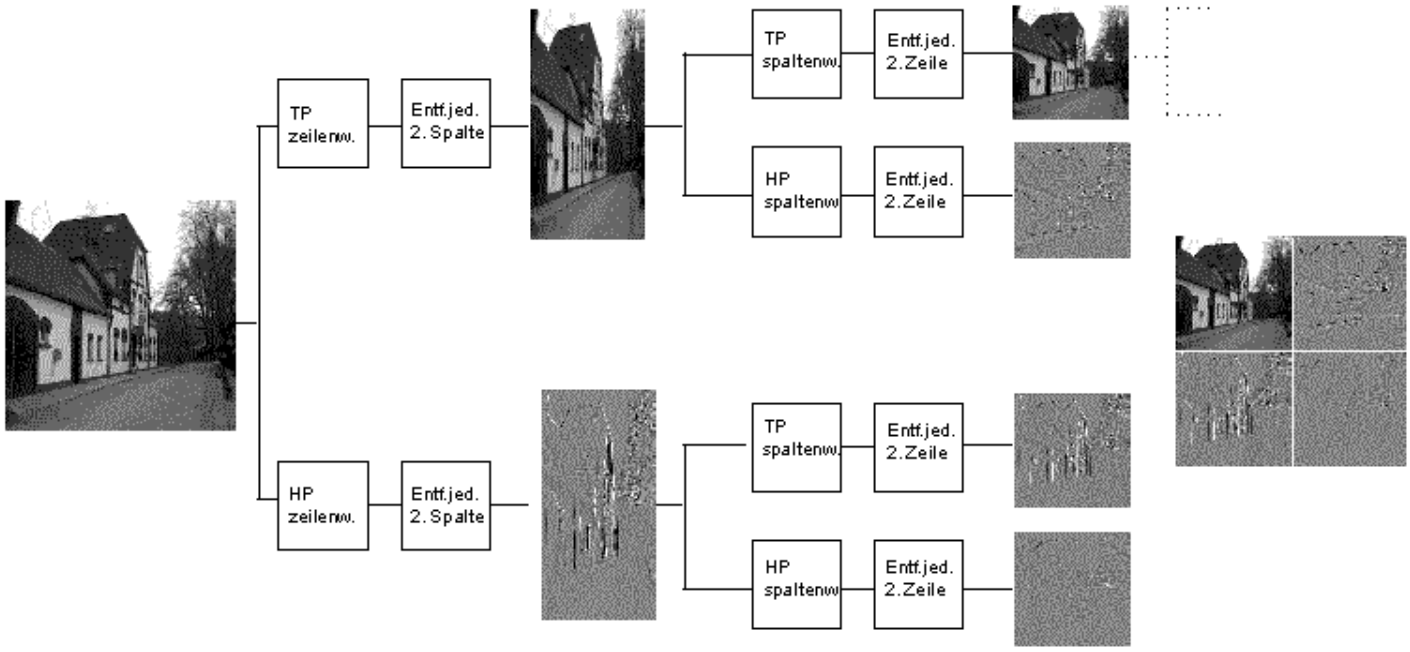


Abbildung 21: Wavelet Transformation mit einem Hochpassbild und mehreren Tiefpassbildern

Bild zu bekommen, hat JPEG2000 neben dem bekannten „Progressive Mode“ von JPEG noch 3 weitere „Progressive Modes“ die es möglichst komfortabel ermöglichen eine Vorschau zu

erhalten. Auch die automatische Fehlerkorrektur bei der Datenübertragung weist daraufhin, dass JPEG2000 eindeutig darauf zielt sich als Internet Bildformat zu etablieren.

Beispiele

JPEG Vergleich

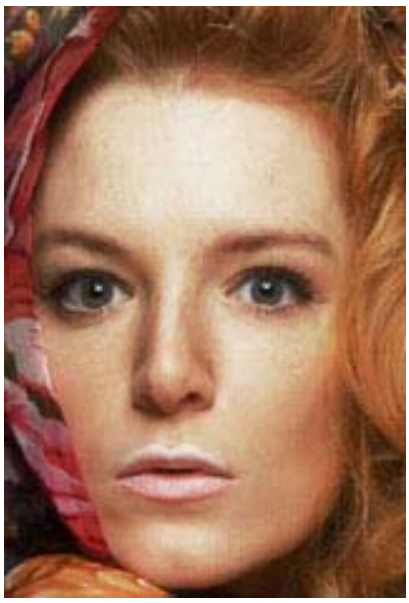


Abbildung 22:
JPEG niedrige Kompression (26 kb)

Abbildung 23:
JPEG hohe Kompression (11 kb)

Abbildung 24:
JPEG doppelte Kompression (12 kb)

JPEG-GIF Vergleich

Abbildung 25: JPEG (14 kb)

Abbildung 26: GIF (11 kb)

Abbildung 27: JPEG (6 kb)

Abbildung 28: GIF (8 kb)

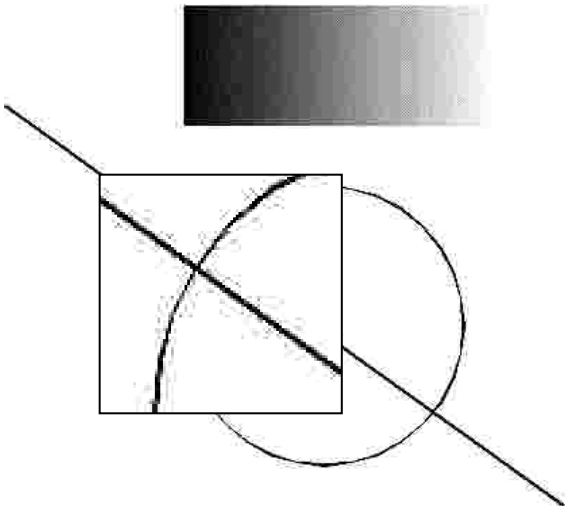


Abbildung 29: JPEG (7 kb)

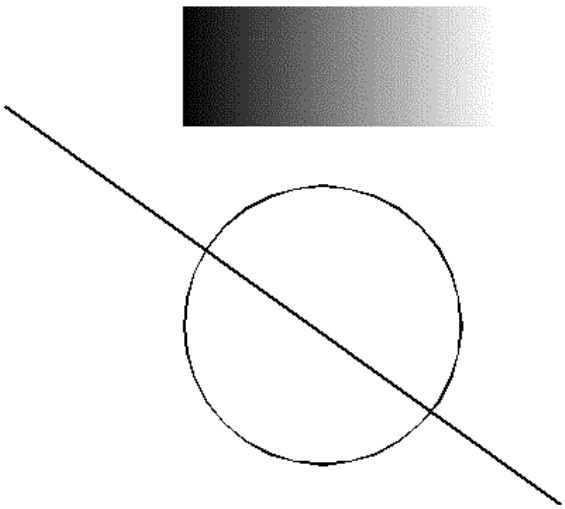


Abbildung 30: GIF (6 kb)

Diese Beispiele zeigen, dass JPEG wesentlich besser mit Farbverläufen umgehen kann als GIF. Besonders in Abb. 27 und 28 und sieht man diesen Unterschied im Farbverlauf im Himmel des Bildes. Im Gegensatz dazu kann GIF scharfe Kanten in Bildern besser komprimieren. Bei JPEG entstehen dagegen Artefakte, da durch die Komprimierung in Blöcken Farbwerte über die Kanten hinweg zusammengefasst und dann codiert werden. Generell lässt sich sagen, dass JPEG Bilder mit weichen Übergängen, wie z.B. Fotos besser komprimiert und GIF mit Bildern mit scharfen Übergängen, wie z.B. am Computer erstellte Bilder, Grafiken, Screenshots oder auch Textbilder besser umgehen kann.

JPEG-GIF-PNG Vergleich

Anzahl Bytes	Inhalt
2	Länge der Markierung
1	Index
16	i.Byte = Anzahl Codes aus i Bits
N	Inhalte, n=Summe der vorhergehenden 16 Byte

Anzahl Bytes	Inhalt
2	Länge der Markierung
1	Index
16	i.Byte = Anzahl Codes aus i Bits
N	Inhalte, n=Summe der vorhergehenden 16 Byte

Abbildung 32: PNG (6 kb)

Anzahl Bytes	Inhalt
2	Länge der Markierung
1	Index
16	i.Byte = Anzahl Codes aus i Bits
N	Inhalte, n=Summe der vorhergehenden 16 Byte

Abbildung 34: GIF (16 kb)

Abbildung 35: PNG (12 kb)

Abbildung 36: JPEG (8 kb)

Bei diesem Vergleich sieht man, wo PNG einzuordnen ist. Es komprimiert bei reinem Text schlechter als GIF, ist aber wesentlich besser als JPG. Besonders die Tatsache, dass keine Artefakte entstehen, ist angenehm. Bei den obigen fotorealistischen Bildern verhält sich PNG leicht schlechter als JPG, ist auch in den Dateigrößen größer, erreicht aber wesentlich bessere Ergebnisse als GIF. GIF schafft nur bedingt gute Qualität (dies variiert jedoch je nach Anpassung der 256 Farben an die Farben des Bildes).

JPEG-JPEG2000 Vergleich

Abbildung 37: JPEG (16 kb)

Abbildung 38: JPEG2000 (16 kb)

Abbildung 39: JPEG 1:20

Abbildung 40: JPEG 1:50

Abbildung 41: JPEG2000 1:20

Abbildung 42: JPEG2000 1:50

Besonders bei hohen Kompressionsraten merkt man, wo die Stärken von JPEG2000 liegen. Hier sind die fehlenden Artefakte angenehm und ermöglichen kleine Dateigrößen bei relativ guter Bildqualität. Dadurch dass das Bild in hohen Kompressionsstufen eher verschwimmt (unscharf wird) anstatt Artefakte zu bilden, sind Konturen noch eher erkennbar.

Zusammenfassung

Der JPEG Standard ist zu dem gängigen Bildformat für Fotos und Echtfarben-Bilder geworden. Maßgebend für diesen Erfolg ist nicht nur die gute Kompression durch DCT und Huffmancodierung sondern auch die gute Skalierbarkeit, die das Format für vielfältige Anwendungsgebiete interessant gemacht hat. Auf die Frage, ob ein Bild eher in JPEG oder GIF komprimiert werden sollte, muss immer die Frage folgen, ob ein Bild scharfe Kanten besitzt oder nicht. Computergrafiken, Text, Cartoons oder generell Bilder mit großen einfarbigen Flächen sollten immer in GIF komprimiert werden, da bei JPEG Artefakte

entstehen können. Bei Fotos hingegen oder Bildern mit mehr als 256 Farben reicht GIF nicht. JPEG ist bei farbenfreudigen Bildern weit im Vorteil und überzeugt durch eine hervorragende Farbbrillanz. Optimiert ist JPEG auf Farbübergänge (da diese in Fotos viel vorkommen). Technisch wird dies umgesetzt im Zusammenfassen der Farbblöcke und Komprimieren in Abhängigkeit von ihren Nachbarblöcken.

JPEG2000 ist der neue Standard dieses Komitees und man erkennt sofort in der Konstruktion dieses Standards dieselbe Professionalität die auch schon JPEG zu seinem Erfolg verholfen hat. Leider kommt er etwas spät und schon andere Formate haben den Platz eingenommen um den er jetzt noch kämpfen müsste. Zudem besteht die Notwendigkeit nicht unbedingt, bereits bestehende etablierte Formate, wie eben JPEG reichen noch aus und so ist mit einem Siegeszug des neuen JPEG2000 Standards nicht zu rechnen.

Abkürzungen

JPEG	Joint Photographic Experts Group
JFIF	JPEG File Interchange Format
DCT	Diskrete Cosinus Transformation
IDCT	Inverse Diskrete Cosinus Transformation
RLE	Run Length Encoding
RGB	Red-Green-Blue
CMYK	Cyan-Magenta-Yellow-Black
YUV	Luminance-Bandwidth-Chrominance
ISO	International Organization for Standardization
CCTT	Canadian Council of Technicians & Technologists

Literatur

Text

http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

http://www.weblearn.hs-bremen.de/risse/RST/SS98/compress/Kompression.htm

http://www.inf.fu-berlin.de/inst/zdm/lect/digvideo/BMP_JPEG_MPEG/sld001.htm

Bilder

Abb. 1: http://www.inf.fu-berlin.de/inst/zdm/lect/digvideo/BMP_JPEG_MPEG/sld001.htm

Abb. 2: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 3: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 4: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 5: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 6: http://www-i4.informatik.rwth-aachen.de/content/teaching/proseminars/sub/2002_2003_ws_docs/mpeg.pdf

Abb. 7: http://www-i4.informatik.rwth-aachen.de/content/teaching/proseminars/sub/2002_2003_ws_docs/mpeg.pdf

Abb. 8: http://www-i4.informatik.rwth-aachen.de/content/teaching/proseminars/sub/2002_2003_ws_docs/mpeg.pdf

Abb. 9: http://www-i4.informatik.rwth-aachen.de/content/teaching/proseminars/sub/2002_2003_ws_docs/mpeg.pdf

Abb. 10: http://www.inf.fu-berlin.de/inst/zdm/lect/digvideo/BMP_JPEG_MPEG/sld001.htm

Abb. 11: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 12: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 13: http://www.inf.fu-berlin.de/inst/zdm/lect/digvideo/BMP_JPEG_MPEG/sld001.htm

Abb. 14: http://www.inf.fu-berlin.de/inst/zdm/lect/digvideo/BMP_JPEG_MPEG/sld001.htm

Abb. 15: http://www.inf.fu-berlin.de/inst/zdm/lect/digvideo/BMP_JPEG_MPEG/sld001.htm

Abb. 16: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 17: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 18: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 19: http://141.22.17.5/~lehnart/Bildverarbeitung/tis_jpeg/html/tis_010.html

Abb. 20: http://www.inf.fu-berlin.de/inst/zdm/lect/digvideo/BMP_JPEG_MPEG/sld001.htm

Abb. 21: http://www.fh-jena.de/contrib/fb/et/personal/ansorg/ftp/wavelet/wavelet.htm

Abb. 22: http://www.peliworks.de/design/formateold/gif_und_jpeg1_druck.html

Abb. 23: http://www.peliworks.de/design/formateold/gif_und_jpeg1_druck.html

Abb. 24: http://www.peliworks.de/design/formateold/gif_und_jpeg1_druck.html

Abb. 25: http://www.peliworks.de/design/formateold/gif_und_jpeg1_druck.html

Abb. 26: http://www.peliworks.de/design/formateold/gif_und_jpeg1_druck.html

Abb. 27:http://www.handwerk-online.de/rubriken/handwerk_online/internet/bildbearbeitung-internet-gif-jpg.htm

Abb. 28:http://www.handwerk-online.de/rubriken/handwerk_online/internet/bildbearbeitung-internet-gif-jpg.htm

Abb. 29: http://www.informatik.tu-darmstadt.de/VS/Lehre/WS95-96/Proseminar/cunning/#5

Abb. 30: http://www.informatik.tu-darmstadt.de/VS/Lehre/WS95-96/Proseminar/cunning/#5

Abb. 31: selbsterstellt

Abb. 32: selbsterstellt

Abb. 33: selbsterstellt

Abb. 34: http://www.starhtml.de/online/grafik.htm

Abb. 35: http://www.starhtml.de/online/grafik.htm

Abb. 36: http://www.starhtml.de/online/grafik.htm

Abb. 37: http://nirvana.informatik.uni-halle.de/~ritter/Seminar2002/klapperstueck/

Abb. 38: http://nirvana.informatik.uni-halle.de/~ritter/Seminar2002/klapperstueck/

Abb. 39: http://www.fh-jena.de/contrib/fb/et/personal/ansorg/ftp/wavelet/wavelet.htm

Abb. 40: http://www.fh-jena.de/contrib/fb/et/personal/ansorg/ftp/wavelet/wavelet.htm

Abb. 41: http://www.fh-jena.de/contrib/fb/et/personal/ansorg/ftp/wavelet/wavelet.htm

Abb. 42: http://www.fh-jena.de/contrib/fb/et/personal/ansorg/ftp/wavelet/wavelet.htm

Grundlegende Audioformate

von

Stephan Lehmann

Einleitung

Dieses Kapitel beschäftigt sich mit grundlegenden elektronischen Kodierungen von Audioformaten. Um dieses Thema zu erörtern, muss anfangs auf spezifische Eigenschaften des menschlichen Gehörs, sowie auf Grundlagen digitaler Audio-Kodierung eingegangen werden. Anschließend werden Möglichkeiten verlustbehafteter sowie verlustfreier Audio-Kodierung betrachtet und verschiedene Datenformate vorgestellt. Es wird auf die verschiedenen Formen der Pulse Code Modulation eingegangen. Darauf folgt eine Beschreibung des WAV-Formats mit Schwerpunkt auf der Struktur dieses Formats. Ziel dieses Kapitels ist ein Überblick über die Funktionsweise gängiger Technologien, Kompressionsverfahren und einiger Dateiformate.

Grundlagen

Um sich mit konkreten Audiokompressionsverfahren auseinandersetzen zu können, ist es sinnvoll, zunächst einige grundlegende Begriffe und Verfahren zu erläutern, die in diesem Zusammenhang häufig Anwendung finden.

Eigenschaften des menschlichen Gehörs

Im Zusammenhang mit Audiosignalen macht es Sinn, sich mit den Besonderheiten des menschlichen Gehörs zu befassen. Speziell im Hinblick auf Kompressionsmöglichkeiten verspricht die Disziplin der Psychoakustik einen bemerkenswerten Nutzen. In diesem Abschnitt sollen jedoch nur einige grundlegende Eigenschaften erläutert werden, von denen einige in Zusammenhang mit den vorgestellten Kompressionsverfahren stehen.

Frequenzabhängiges Lautstärkempfinden

Die Wahrnehmung des menschlichen Ohres ist beschränkt auf Frequenzen zwischen 20 und 20000 Hertz. Frequenzen unterhalb von 20 Hertz (Infraschall) und oberhalb von 20000 Hertz (Ultraschall) sind nicht mehr wahrnehmbar. Der Frequenzbereich des hörbaren Schalls ist jedoch nur ein Mittelwert, da viele Faktoren wie zum Beispiel Alter, Training, Gewöhnung Einfluss darauf nehmen. Aus diesem Grund ist es auch sinnvoll, den zu verarbeitenden Frequenzbereich großzügig auszulegen.
Zudem werden Frequenzen innerhalb dieses Spektrums nicht gleich gut wahrgenommen. Das Hörvermögen des Ohrs ist im Bereich zwischen 2000 und 4000 Hertz am besten. Allgemein kann man sagen, dass das Auflösungsvermögen des Ohrs mit steigender Frequenz immer weiter abnimmt. Genauso verhält es sich mit sehr niedrigen Frequenzen unter 500 Hertz. Durch das Auflösungsvermögen wird die Hörschwelle bestimmt. Alle Geräusche die unter der Hörschwelle liegen werden nicht mehr wahrgenommen (blauer Bereich unten).

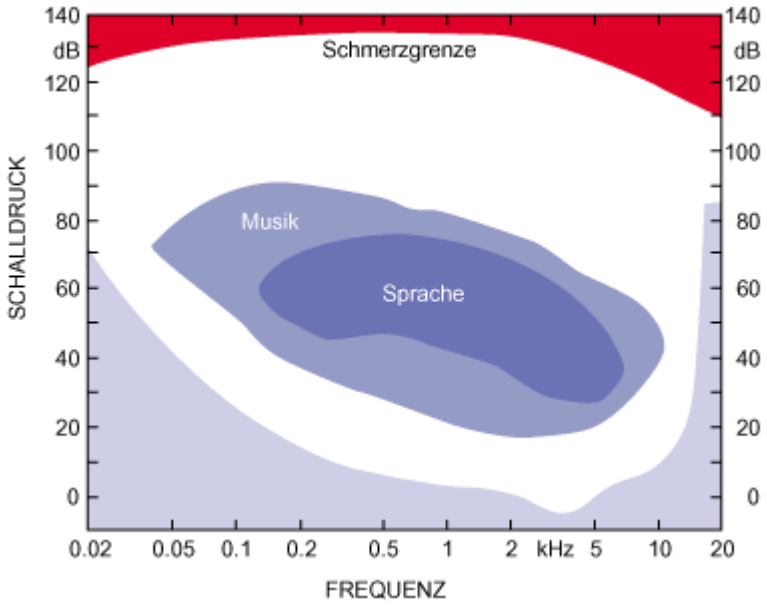


Abbildung 1: Hörempfinden

Gleichzeitig hängt die Wahrnehmung des Ohrs auch noch von der Lautstärke ab. Allgemein gilt, dass leise Geräusche besser aufgelöst werden als laute Geräusche. Das bedeutet, bei leisen Geräuschen werden auch Digitalisierungsfehler eher wahrgenommen.

Maskierungs- / Verdeckungseffekte

Eine weitere Eigenschaft des menschlichen Gehörs ist die Frequenzmaskierung. Dabei handelt es sich um folgendes Phänomen: ein sehr lauter Ton hebt die Hörbarkeitsschwelle in der unmittelbaren Umgebung seiner Frequenz an und überdeckt bzw. maskiert möglicherweise leisere Töne in diesem Bereich, die normalerweise hörbar wären. Die Stärke der Maskierungseffekte ist Abhängig von der Frequenz des verursachenden Tons. Ein Ton mit niedriger Frequenz hebt die Hörbarkeitsschwelle in einem Bereich von etwa 100 Hertz um die Tonfrequenz an. Töne höherer Frequenzen heben die Hörbarkeitsschwelle bis zu einem Bereich von 5000 Hertz um die Tonfrequenz an. Analog dazu gibt es die zeitliche Maskierung. Dabei wird ein Ton noch wahrgenommen, obwohl er schon beendet ist und kann nachfolgende Töne verdecken (Nachmaskierung). Gleiches gilt für vorhergehende Töne (Vormaskierung).

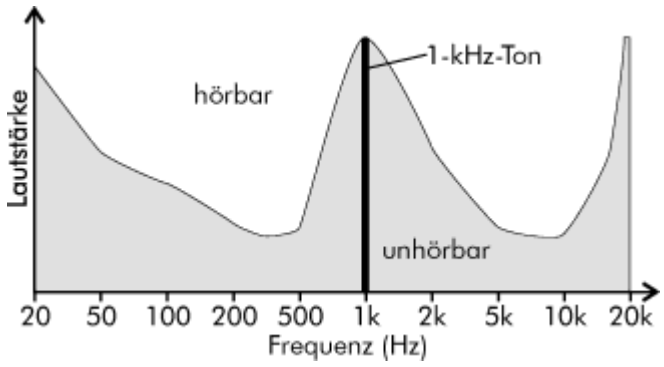


Abbildung 2 : Maskierung / Verdeckung

Digitalisierung von Audiodaten

Um die Verarbeitung von Audiodaten mit DV-Systemen zu ermöglichen ist es nötig, die Daten in eine digitale Form zu überführen. Dazu werden die kontinuierlichen analogen Audiosignale regelmäßig gemessen, in diskrete digitale Werte überführt und dann kodiert.

Abtastung

Das analoge Signal wird in gleich bleibenden Zeitintervallen abgetastet. Bildlich kann man es sich so vorstellen, dass ein senkrechtes Raster über das Ausgangsignal gelegt wird. Es entsteht ein diskontinuierliches Signal, welches aber immer noch analog ist, denn der Wert der Amplitude kann in bestimmten Grenzen immer noch jeden beliebigen Wert annehmen. Die Anzahl der Abtastpunkte pro Sekunde wird Abtastfrequenz oder Abtastrate genannt. Von der Wahl der Zeitintervalle hängt die Qualität des digitalen Signals ab. Je dichter die Zeitintervalle, desto besser die Qualität, aber desto mehr Daten fallen auch an.

Quantisierung

Nun werden nicht mehr alle Amplitudenwerte zugelassen, sondern nur noch bestimmte Werte. Bildlich vorgestellt: Es wird ein waagerechtes Raster über das Signal gelegt und die Signale können nur noch Werte annehmen, die an den Kreuzungspunkten beider Raster liegen. Normalerweise liegen die analogen Signalwerte nicht genau auf den Kreuzungspunkten. Deshalb muss man den realen Wert entsprechend auf- oder abrunden. Dadurch entstehen natürlich Abweichungen vom Originalton. Je gröber das waagerechte Raster ist, bzw. je weniger Werte das Signal annehmen kann, desto größer ist auch der Quantisierungsfehler.

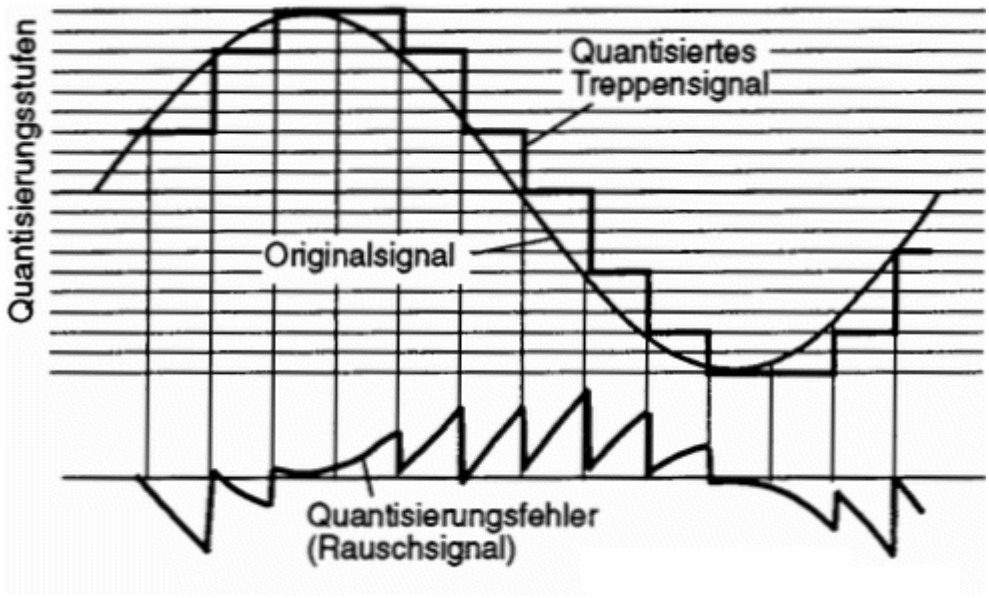


Abbildung 3: Digitalisierung

Den relativen Quantisierungsfehler, also das Verhältnis des Fehlers zum Signalwert, bezeichnet man als Rauschabstand. Die Anzahl der zugelassenen Werte nennt man Auflösung (üblicherweise in Bit). Außerdem ist, nach dem Abtasttheorem von Shannon, folgende

Bedingungen zu beachten: Die Abtastfrequenz muss mindestens doppelt so groß sein, wie die höchste Frequenz, die im abgetasteten Signal vorkommt. Bei Nichtbeachtung werden sonst hohe Frequenzen ausgelöscht bzw. auf niedrigere abgebildet.

Die bei der Digitalisierung anfallenden Datenmengen pro Sekunde errechnen sich nach folgender Formel :

Auflösung [Bit] / 8 * Abtastrate [1/s] * Kanäle

Bei einer Audio-CD mit 16 Bit Auflösung, einer Abtastfrequenz von 44,1 kHz und 2 Kanälen (Stereo) ergibt sich eine Datenmenge von 176 KByte/s

	44,1 kHz	**22 kHz**	**8 kHz**
16 Bit	**1411 kBit/s (Audio CD)**	**704 kBit/s**	**256 kBit/s**
12 Bit	**1056 kBit/s**	**528 kBit/s**	**192 kBit/s**
8 Bit	**704 kBit/s**	**352 kBit/s**	**128 kBit/s (ISDN)**

Tabelle 1: übliche Datenmengen

Die Kodierung kann in 2 Kategorien unterschieden werden, die verlustfreie Kodierung (Kompression) sowie die verlustbehaftete Kodierung (Reduktion).

Verlustfreie Kodierung

In vielen Anwendungen der digitalen Tontechnik werden sehr hohe Anforderungen an die Signalqualität gestellt. In Verbindung mit dem großen Vorteil der Digitaltechnik, ein Signal beliebig oft identisch reproduzieren zu können, liegt es nahe auf die verlustfreie Kodierung zurückzugreifen. Gleichzeitig soll das Datenaufkommen aber möglichst klein sein.

Die Verlustfreie Kodierung beseitigt nur so genannte Redundanzen, Informationen die schon in vorhergehenden Bereichen des Signals vorkamen. Verlustfreie Kodierungs-Algorithmen, wie Huffman oder Lempel-Ziv sind hervorragend geeignet um Texte oder Bilder zu komprimieren. Bei Audio-Daten stoßen sie jedoch schnell an ihre Grenzen. Huffman, sowie Lempel-Ziv sind auf die Komprimierung identischer Wiederholungen spezialisiert. Durch die große Anzahl verschiedener Samples und Geräusche allgemein kommen solche Wiederholungen bei Audio-Daten nur sehr selten vor. ZIP (Lempel-Ziv) erreicht bei höchster Kompressionsstufe nur etwa 10 % Kompression. Von einer 5-minütigen Audioaufnahme in CD-Qualität bleiben im günstigsten Fall immer noch 35 MByte übrig. Ein weiteres Problem entsteht dadurch, dass solchermaßen komprimierte Audio-Daten vor dem Abspielen oder Bearbeiten erst entpackt werden müssen.

Verlustlose Kodierungsverfahren werden überall dort angewendet, wo die Datenmenge von digitalen Signalen zum Zwecke der Übertragung oder Speicherung reduziert werden soll, ohne das Signal durch den Kodier- und Dekodiervorgang zu verändern. Dadurch ergeben sich folgende grundlegende Anwendungsmöglichkeiten:

- Archivierung von digitalem Audiomaterial
- Erhöhung der Aufzeichnungskapazität von digitalen Datenträgern
- Übertragung von Audiodaten in Netzwerken

Verlustbehaftete Kodierung

Im Unterschied zur verlustfreien Kodierung liegt das Hauptaugenmerk bei der verlustbehafteten Kodierung darauf, die Datenrate möglichst gering zu halten. Dabei finden meist die oben beschriebenen Eigenschaften des menschlichen Ohres Anwendung, indem zum Beispiel Töne, die unterhalb der Hörschwelle liegen, sowie maskierte Töne bei der Kodierung weggelassen werden. Durch solche Verfahren kann der Speicherplatz von Audiodaten beträchtlich reduziert werden. Im Folgenden sollen kurz einige dieser Verfahren vorgestellt werden.

Predictive Coding

Es wird eine Vorhersage über das nächste Sample auf Grundlage des vorangegangenen Samples gemacht. Gespeichert wird lediglich die Differenz zwischen dem echten Signal und dem Vorhersagewert. So muss nicht immer ein ganzes Signal gespeichert werden. Da die Differenzen zwischen Vorhersage und echtem Wert effektiver kodiert werden können als die eigentlichen Signale selbst, führt Predictive Coding zu einer Verringerung des Datenaufkommens.

Subband Coding

Beim Subband Coding wird das breitbandige Eingangssignal mit Hilfe einer so genannten Filterbank in mehrere schmalere Signale, Subbänder, zerlegt. Viele Subbänder enthalten unwichtigere Signale als andere laute Bänder. Für die wichtigen Bänder kann man nun mehr Platz aufwenden als für die unwichtigen Bänder. Mache kann man unter Umständen sogar weglassen (Maskierung). Diese Auswahl trifft der Encoder mit Hilfe des Psychoakustischen Modells. Zusätzlich zu den Audiodaten muss der Encoder auch Informationen über die Bitverteilung übertragen, so dass der Decoder das Signal aus den abgespeicherten Informationen synthetisieren kann.

Spectral- oder Transform Coding

Bei dieser Form der Kodierung wird auf das Eingangssignal eine Fast Fourier Transformation angewendet. Dabei wird das Signal in eine Überlagerung von Sinus- bzw. Cosinussignalen zerlegt. Solche Signale lassen sich besser kodieren als andere. Subband Coding kann man als Spezialfall von Transform Coding auffassen.

Zusammenfassend kann man sagen, dass bei der verlustbehafteten Kodierung das Signal nach der Frequenz aufgeschlüsselt wird und auf eine Form reduziert wird bei der sie mit verlustfreien Kodierungen gut komprimierbar sind. Damit kann eine deutliche Reduktion der Datenmenge erreicht werden wodurch sich neue Anwendungsgebiete eröffnen. Beispielhaft sei hier das Streaming über schmalbandige Netzwerkverbindungen (z.B. Internet) genannt. Bekannte Codecs sind MPEG Audio, AAC, WMA, OGG, Real Audio.

PCM (Pulse Code Modulation)

Die Pulse Code Modulation stellt die technische Realisierung der in Abschnitt 3 vorgestellten Digitalisierungsprinzipien dar. Das Signal wird abgetastet, quantisiert und kodiert. Man unterscheidet zwei wesentliche PCM-Verfahren in Bezug auf den Rauschabstand – lineares und dynamisches PCM. Zur Erinnerung: Der Rauschabstand ist der Quantisierungsfehler im Verhältnis zum Signalwert.

lineares PCM

Die bekannteste Anwendung des linearen PCM dürfte wohl die Audio-CD sein. Beim linearen PCM sind die Quantisierungsschritte immer gleich groß, der Rauschabstand ist somit nicht konstant und macht sich bei kleinen Signalpegeln stärker bemerkbar als bei großen Signalpegeln. Es stellt sich die Frage, wie klein man die Quantisierung wählen muß, um eine ausreichende Qualität zu erhalten. Dafür wird ein Signal zu Rauschabstand - Verhältnis (Signal-to-Noise-Ratio SNR) definiert durch :

$$SNR = 10 \cdot \lg(S/R)$$

Zum Beispiel wird ein Sprachsignal als akzeptabel definiert, wenn gilt : SNR < 40 dB. Um eine ausreichende Qualität zu erreichen muss das SNR immer unter einem gewissen Wert liegen, da sonst leise Geräusche nicht vom Quantisierungsrauschen zu unterscheiden sind. Um diese Eigenschaft garantieren zu können, müssen die Quantisierungswerte dicht beieinander liegen, was einer hohen Auflösung entspricht. Da Quantisierungsfehler laut Psychoakustik bei größeren Signalpegeln nicht mehr so stark wahrgenommen werden, verschwendet man viel Speicherplatz. Deshalb bietet es sich an, eine ungleichförmige Quantisierung vorzunehmen. Dazu dient die dynamische PCM.

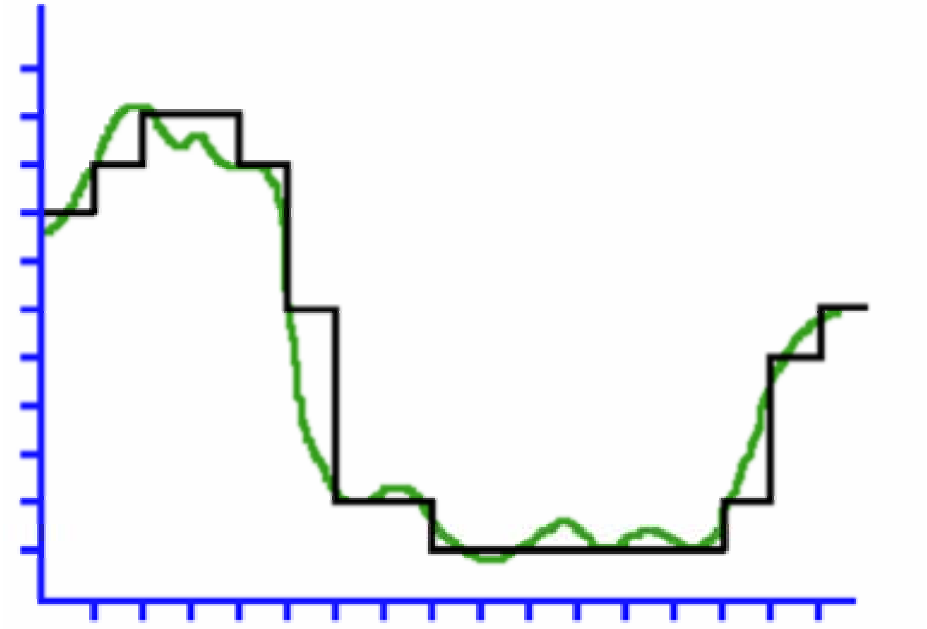


Abbildung 4 : Lineare Quantisierung

dynamisches PCM

Bei der dynamischen PCM werden die Quantisierungschritte unterschiedlich groß gewählt um den Rauschabstand konstant zu halten. Dafür existieren 2 Varianten.

Variante 1

Die Quantisierungschritte werden so angeordnet, dass bei kleinen Signalwerten kleine Quantisierungschritte gewählt werden und bei großen Werten große Quantisierungschritte (z.B. eine logarithmische Verteilung der Quantisierungschritte).

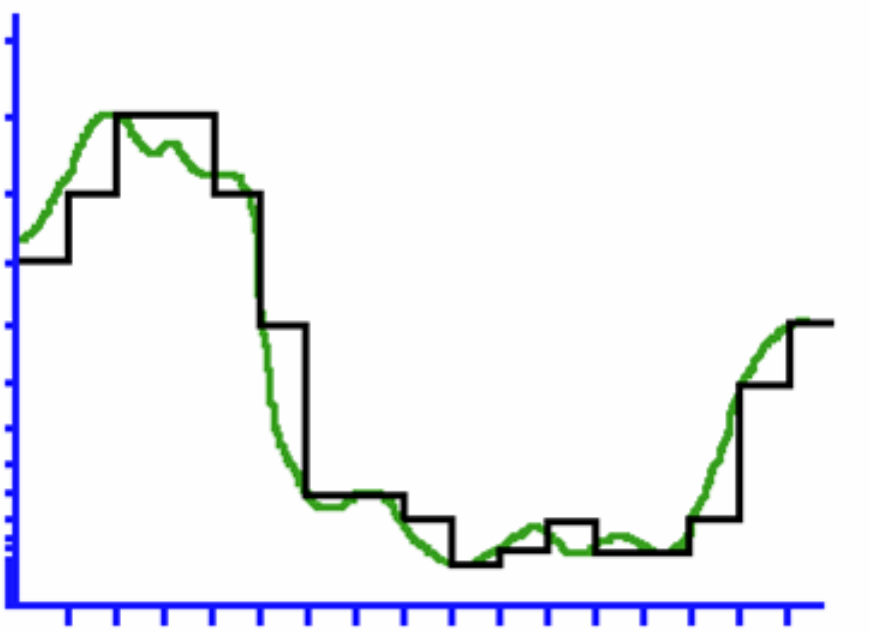


Abbildung 5: Logarithmische Quantisierung

Variante 2

Das analoge Signal wird vor der Quantisierung gestaucht und bei der Dequantisierung wieder expandiert. Die Stauchung erfolgt so, dass große Signalwerte stärker gestaucht werden als kleine Signalwerte. Die eigentliche Quantisierung erfolgt dann wieder in gleich großen Schritten. Baugruppen für die Stauchung und Expansion werden als Kompander (KOMpressor und exPANDER) bezeichnet. Standard-Kompander sind µ-law und A-law. Ursprünglich entwickelt für das Telefonsystem, werden sie auch in anderen Bereichen eingesetzt (z.B. µ-Law im au-Format von Sun). Während µ-law in Nordamerika und Japan eingesetzt wird, findet A-law vor allem in Europa Anwendung. Die Abtastwerte werden nach folgenden Formeln gestaucht:

µ-law

$$S\mu=\log(1+255*S)/\log(256)$$

S - linearer Abtastwert zwischen 0 und 1
Sµ - gestauchter Wert

A-law

$$S_A = A \cdot S/(1 + \ln A) \text{ wenn } S \leq 1/A$$
$$S_A = (1 + \ln (A \cdot S))/(1 + \ln A) \text{ wenn } 1/A \leq S \leq 1$$

A = 87,6 (andere Werte möglich)
S - linearer Abtastwert zwischen 0 und 1
Sμ - gestauchter Wert

Differential PCM (DPCM)

Beim Differential PCM wird der Umstand genutzt, dass die Unterschiede zwischen zwei aufeinander folgenden Abtastwerten in der Regel nur sehr gering sind. Da diese Differenzen auch im Verhältnis zu den ganzen Werten viel kleiner sind, lassen sie sich mit weniger Bits kodieren. In bestimmten Abständen werden ganze Abtastwerte als Referenzpunkte übertragen und dazwischen nur noch die Differenzen. Bei dieser Art der Kodierung können jedoch Probleme auftreten. Wenn sich das Eingangssignal sehr schnell ändert, dann kann es sein, das die reservierte Anzahl Bits für die Differenz nicht mehr ausreicht. Diesen Effekt bezeichnet man als „slope overload“. Abhilfe schafft die Zuweisung der Bits zu einer Tabelle, in der eine Menge von möglichen großen und kleinen Änderungen gespeichert ist.

Bitwert	**-8**	**-7**	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Änderung Fibonacci	**-34**	**-21**	**-13**	**-8**	**-5**	**-3**	**-2**	**-1**	**-0**	**1**	**2**	**3**	**5**	**8**	**13**	**21**
Änderung Exponential	**-128**	**-64**	**-32**	**-16**	**-8**	**-4**	**-2**	**-1**	**0**	**1**	**2**	**4**	**8**	**16**	**32**	**64**

Tabelle 2 : Schrittwerte DPCM

Beispielsweise würde bei exponentieller Verteilung eine Differenz zwischen zwei Abtastwerten von 64 mit einer 7 kodiert werden, eine Differenz von 8 mit einer 4. So können gleichzeitig viele kleine und auch einige große Differenzen kodiert werden. Doch auch diese Lösung kann noch verbessert werden. Dazu dient eine Weiterentwicklung der Differential Pulse Code Modulation, das ADPCM.

Adaptive Differential PCM (ADPCM)

Durch vorausschauende Betrachtung der Abtastwerte wird festgestellt, ob man sich auf einen krassen Übergang zu bewegt. Ist das der Fall, erhöht der Algorithmus schrittweise die für die Differenzkodierung verfügbare Bitanzahl. Nach dem krassen Wechsel wird dann die Anzahl der Bits für die Differenzkodierung schrittweise wieder verringert, bis ein erneuter Bedarf für eine Erhöhung besteht. In jedem Abtastschritt müssen nun 2 Werte berechnet werden, die Differenz zum Vorgängerwert und ein Wert der die aktuelle Änderungsgeschwindigkeit ausdrückt (Schrittweite). Im Folgenden wird das IMA-ADPCM (Interactive Multimedia Association) näher beschrieben. Beim IMA-ADPCM besteht jeder Samplewert aus 4 Bit,

Vorzeichenbit und drei Wertebits. Damit sind Werte von +7 bis -7 kodierbar. Nach jedem Samplewert wird die Schrittweite neu berechnet.

Kodierung

Das folgende Bild zeigt das Block Diagramm eines ADPCM Kodiervorgangs.

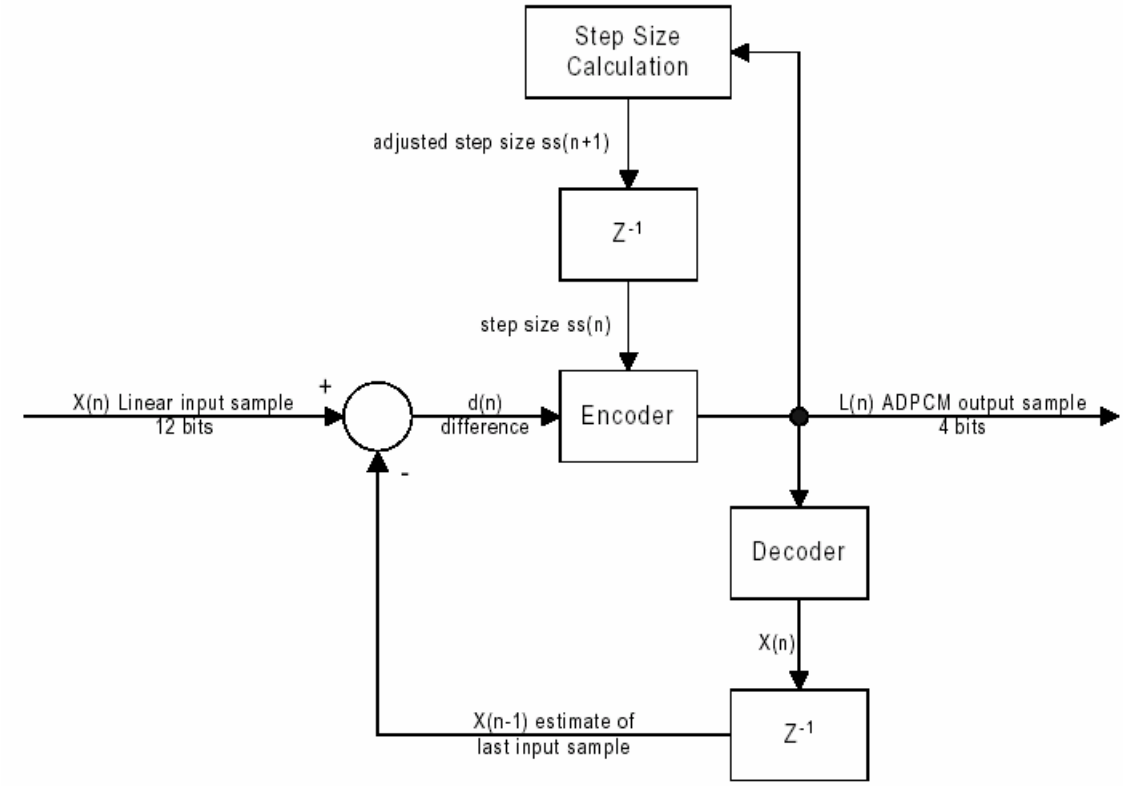


Abbildung 6 : ADPCM-Kodierer

Ein Eingangswert X(n) wird mit dem vorherigen Eingangswert X(n-1) verglichen. Die Differenz d(n), zusammen mit der aktuellen Schrittweite ss(n) geht durch den Kodierer, der den Ausgangswert L(n) erzeugt und die nächste Schrittweite ss(n+1) errechnet.

Die Berechnung des Ausgangswertes aus Differenz und Schrittweite dargestellt in Pseudocode :

```
B3 = B2 = B1 = B0 = 0
if (d(n) < 0) then B3 = 1
d(n) = ABS(d(n))
if (d(n) >= ss(n)) then B2 = 1 and d(n) = d(n) - ss(n)
if (d(n) >= ss(n) / 2) then B1 = 1 and d(n) = d(n) - ss(n) / 2
if (d(n) >= ss(n) / 4) then B0 = 1
L(n) = (10002 * B3) + (1002 * B2) + (102 * B1) + B0
```

Dekodierung

Das nächste Bild zeigt das Block Diagramm eines ADPCM-Dekodiervorgangs. Ein kodierter Wert L(n) geht in den Dekodierer, welcher die Differenz zum vorherigen Ausgabewert errechnet. Diese Differenz wird zum vorherigen Ausgabewert hinzuaddiert und ergibt den neuen Ausgabewert. Gleichzeitig wird aus dem Eingangswert auch die Schrittweite berechnet und dem Dekodierer zur Verfügung gestellt.

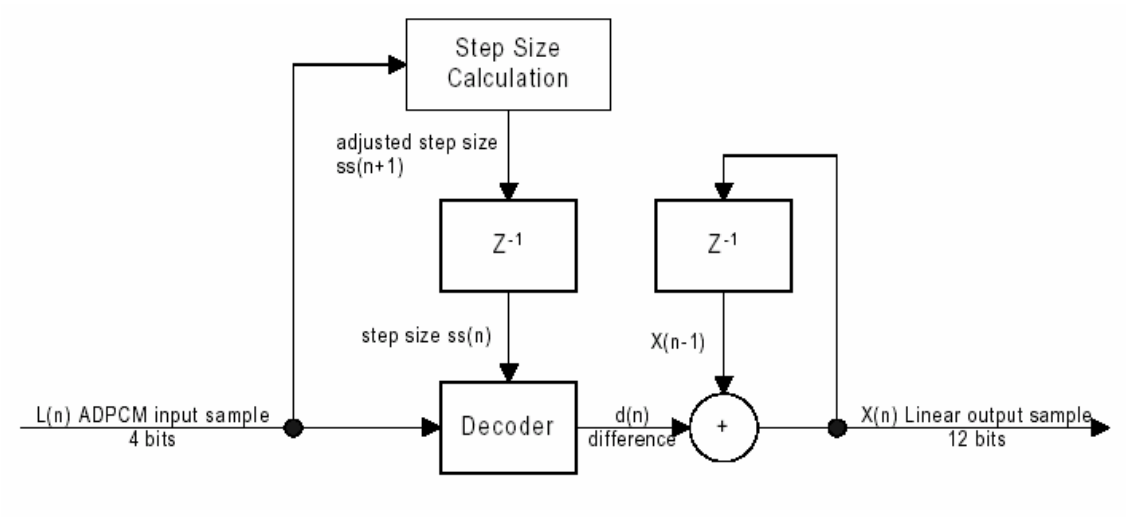


Abbildung 7 : ADPCM-Dekodierer

Pseudocode für die Berechnung des Ausgabewertes:

d(n) = (ss(n)*B2)+(ss(n)/2*B1)+(ss(n)/4*BO)+(ss(n)/8)
if (B3 = 1) then d(n) = d(n) * (-1)
X(n) = X(n-1) + d(n)

Berechnung der Schrittweite

Für Kodierung und Dekodierung muss der ADPCM-Algorithmus die Schrittweite berechnen. Die Schrittweite für den nächsten Wert wird durch folgende Gleichung beschrieben:

ss(n+1) = ss(n) * 1.1M(L(n))

Diese Gleichung kann als zweiteilige Tabelle implementiert werden. Zuerst wird der Wert des ADPCM-Samples als Index für eine Tabelle von Änderungsfaktoren benutzt (Tabelle 2). Dann wird der entsprechende Tabelleneintrag benutzt um den Index einer zweiten Tabelle zu verschieben (Tabelle 3). In dieser Tabelle stehen die echten Schrittweiten. Werte größer als 3 erhöhen die Schrittweite. Werte kleiner als 4 verringern die Schrittweite.

Bitwert	**-7**	**-6**	**-5**	**-4**	**-3**	**-2**	**-1**	**-0**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Indexänderung	**8**	**6**	**4**	**2**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**-1**	**2**	**4**	**6**	**8**

Tabelle 3 : M(L(n)) Werte

7	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**16**	**17**
19	**21**	**23**	**25**	**28**	**31**	**34**	**37**	**41**	**45**
50	**55**	**60**	**66**	**73**	**80**	**88**	**97**	**107**	**118**
130	**143**	**157**	**173**	**190**	**209**	**230**	**253**	**279**	**307**
337	**371**	**408**	**449**	**494**	**544**	**598**	**658**	**724**	**796**
876	**963**	**1060**	**1166**	**1282**	**1411**	**1552**	**1707**	**1878**	**2066**
2272	**2499**	**2749**	**3024**	**3327**	**3660**	**4026**	**4428**	**4871**	**5358**
5894	**6484**	**7132**	**7845**	**8630**	**9493**	**10442**	**11487**	**12635**	**13899**
15289	**16818**	**18500**	**20350**	**22385**	**24623**	**27086**	**29794**	**32767**	

Tabelle 4 : Schrittweiten (IMA-ADPCM)

Datenformat

Da beim ADPCM wie auch beim DPCM jeweils nur die Differenzen zwischen zwei Samplewerten übertragen werden, muss der Dekodierer auch einen Startwert kennen. Um den Dekodiervorgang an beliebigen Stellen des Datenstroms starten zu können, wird er in Pakete aufgeteilt. Am Anfang jeden Paketes steht der aktuelle Samplewert und die aktuelle Schrittweite. Die Pakete bestehen aus dem Paketkopf (2-8 Bytes) und den Sampledaten (je 4 Bit). Da durch die IMA keine genauere Spezifikation erfolgte, haben sich im Verlauf der Zeit unterschiedliche Implementierungen entwickelt. Zwei davon sind MS-ADPCM von Microsoft und AIFF-C von Apple (Quicktime).
Bei MS-ADPCM werden Mono- und Stereopakete unterschieden. Bei Monopaketen befindet sich am Anfang der Kopf und dahinter n Samples als Paketdaten. Bei Stereopaketen befinden sich am Anfang je ein Paketkopf für links und rechts gefolgt von abwechselnden rechts-links-Samples. Ein Paketkopf setzt sich aus Startsample (8 bzw. 16 Bit), der aktuellen Schrittweite (8 Bit) und einem 0 Byte zusammen.

Kopf	**Sample 1**	**Sample 2**	**Sample 3**	**Sample 4**	**Sample 5**	**…**

Tabelle 5 : Mono Paket (MS-ADPCM)

Kopf links	**Kopf rechts**	**Sample 1 links**	**Sample 1 rechts**	**Sample 2 links**	**Sample 2 rechts**	**…**

Tabelle 6 : Stereo Paket (MS-ADPCM)

Bei AIFF-C besteht der Paketkopf aus dem Startsample (obere 9 Bit) und der aktuellen Schrittweite (7 Bit). Die Paketdaten enthalten immer 64 Samples. Bei Stereosignalen werden extra Pakete für den rechten und linken Kanal angelegt.
Vorteile von MS-ADPCM ist seine Flexibilität. Es lassen sich eine beliebige Menge Samples in einem Paket unterbringen. Dadurch kann man bei großen Paketen den Overhead gering halten. Bei kleinen Paketen kann dagegen AIFF-C durch seinen kleineren Paketkopf Vorteile bringen.

WAV

Das WAV Format gehört zur großen Gruppe der Chunkformate. Der Ausgangspunkt dieser Entwicklung war das EA IFF 85-Format von Electronic Arts. Sowohl Microsoft als auch Apple entwickelten basierend auf dieser Vorlage eigene Soundformate. Microsoft schuf zunächst das RIFF (Ressource Interchange File Format) aus dem später das WAV-Format hervorging. Bei Apple entstand das AIFF Format (Audio Interchange File Format) und später das AIFF-C Format (Integration von Kompression). Von der Firma SUN wurde zeitgleich das SND Format entwickelt.

RIFF Format

Das Ressource Interchange File Format ist ein allgemeines Windows File Format zum Speichern von Multimediadaten, dazugehörigen Beschreibungen, Formaten, Wiedergabelisten, etc.. Im Prinzip besteht eine RIFF-Datei aus verschiedenen Teilpaketen, so genannten Chunks. Jeder dieser Chunks kann solche Daten enthalten aber auch aus anderen Chunks zusammengesetzt sein. So entsteht eine hierarchische Struktur aus Chunks und Subchunks. Außerdem besitzt jede RIFF-Datei einen 8 Byte RIFF-Header welcher die Datei identifiziert und die Dateigröße angibt.

```
struct {
        char        id[4];        // Identifizierungsstring = “RIFF”
        long        len;          // Dateilänge ohne den Header
} riff_header;
```

Direkt nach dem Header wird der Typ der RIFF-Datei (4 Byte) angegeben.

```
        char        id[4];        // z.B. Wave-Typ = “WAVE”
```

Dann folgen die einzelnen Chunks. Auch die Chunks beginnen wieder mit einen 8 Byte Header welcher den Chunk-Typ sowie die Chunkgröße angibt.

```
struct {
        char        id[4];        // Identifizierungsstring z.B. “DATA“ oder “FMT “
        long        len;          // Chunklänge ohne den Chunk-Header
} chunk_header;
```

WAV Format

Die wichtigsten Chunks für das WAV-Format sind im folgenden aufgelistet :

Format-Chunk (Enthält Informationen über die Soundformatierung)
Data-Chunk (enthält die Audiodaten)
Fact-Chunk (Angaben zum Aufzeichnungsformat)
Cue-Chunk (Liste von Markierungen)
Playlist-Chunk (Wiedergabeliste)
Associated Data List-Chunk (Liste anwendungsspezifischer Daten)
Label-Chunk
Note Chunk
Labeled Text-Chunk

Dabei wird zwischen notwendigen und optionalen Chunks unterschieden. 2 Chunks müssen in jeder WAV-Datei vorkommen: Format-Chunk und Data-Chunk. Sie bilden das Grundgerüst einer WAV-Datei. Alle anderen Chunks sind optional. Es gibt noch weitere Chunks, wie zum Beispiel Sampler- und Instrument-Chunk. Mit diesen beiden Chunks können Samples und Instrumente für MIDI Anwendungen beschrieben werden. Im Folgenden wird auf die oben genannten Chunks anhand eines C-ähnlichen Formats genauer eingegangen.

Format-Chunk

Der Format-Chunk beinhaltet wichtige Parameter der Sampledaten wie Samplerate, Auflösung und Anzahl der Kanäle.

```
struct {
        char[4]             “fmt “;
        long                chunkSize;
        short               wFormatTag;
        unsigned short      wChannels;
        unsigned long       dwSamplesPerSec;
        unsigned long       dwAvgBytesPerSec;
        unsigned short      wBlockAlign;
        unsigned short      wBitsPerSample;
        unsigned short      cbSize;              // nur bei Kompression
} Format-Chunk;
```

Der Wert von wFormatTag gibt an, um welches Format es sich handelt und damit ob die Audiodaten komprimiert sind oder nicht. Einige ausgewählte Formate finden sich im Anhang. Wenn hier ein anderer Wert als 1 steht, muss es zusätzlich einen Fact-Chunk geben. Im Format-Chunk wird dann noch der Parameter cbSize angegeben. Dieser Wert enthält die Länge der zusätzlichen Informationen im Format-Chunk. wChannels gibt die Anzahl der Kanäle an. 1 steht für Mono, 2 für Stereo, 4 für 4-Kanal-Sound usw.. dwSamplesPerSec gibt die Anzahl der Samples pro Sekunde an. Die drei Standardraten sind 11025, 22050 und 44100 kHz. In dwAvgBytesPerSec steht die Anzahl der Bytes die pro Sekunde abgespielt werden. Damit könnte eine Anwendung zum Beispiel die Größe des nötigen RAM-Puffers berechnen um die Datei fehlerfrei abzuspielen. Der Wert kann nach folgender Formel berechnet werden:

dwSamplesPerSec · wBlockAlign.

wBlockAlign gibt die Größe eines Samples in Byte an. 16 Bit Mono wären beispielsweise 2 Byte, 16 Bit Stereo 4 Byte.wBlockAlign errechnet sich nach der Formel:

wChannels · (wBitsPerSample % 8)

wBitsPerSample bezeichnet die Auflösung eines Samples. (z.B. 16 Bit Auflösung = 16)

Data-Chunk

Der Data-Chunk enthält die Sample-Daten aller Kanäle.

```
struct {
        char[4]             “data“;
```

```
        long            chunkSize;
        unsigned char   waveformData[];
} Data-Chunk;
```

Die Samples haben in waveformData[] folgende Anordnung :

Sample 1	Sample 2	Sample 3	Sample 4
Channel 0	Channel 0	Channel 0	Channel 0

Tabelle 7 : 8 Bit Mono PCM Anordnung

Sample 1		Sample 2	
Channel 0	Channel 1	Channel 0	Channel 1

Tabelle 8 : 8 Bit Stereo PCM Anordnung

Sample 1		Sample 2	
Channel 0 (loB)	Channel 0 (hiB)	Channel 0 (loB)	Channel 0 (hiB)

Tabelle 9 : 16 Bit Mono PCM Anordnung

Sample 1			
Channel 0 (loB)	Channel 0 (hiB)	Channel 1 (loB)	Channel 1 (hiB)

Tabelle 10 : 16 Bit Stereo PCM Anordnung

Jeder Samplewert besteht aus einem Integer. Das least significant byte wird zuerst gespeichert. Die Amplitudenwerte werden im most significant byte gespeichert, die restlichen Bits werden auf null gesetzt.

Fact-Chunk

Dieser Chunk wird nur benötigt, wenn kein lineares PCM verwendet wird. Abhängig von der verwendeten Komprimierung ist der Chunk unterschiedlich aufgebaut. Eine genaue Beschreibung würde an dieser Stelle zu weit führen.

Cue-Chunk

Der Cue-Chunk enthält ein oder mehrere Markierungspunkte, so genannte cue points. Jeder cue point zeigt auf einen bestimmten Offset in den Audiodaten. In Verbindung mit dem Playlist-Chunk können so zum Beispiel Wiederholungen beschrieben werden.

```
struct {
        char[4]         “cue “;
        long            chunkSize;
        long            dwCuePoints;
        CuePoint        points[];
} Cue-Chunk;
```

dwCuePoints gibt die Anzahl der cue points an. Dahinter folgen nacheinander die einzelnen cue points mit folgender Struktur :

```
struct {
```

```
        long            dwIdentifier;
        long            dwPosition;
        char[4]         fccChunk;
        long            dwChunkStart;
        long            dwBlockStart;
        long            dwSampleOffset;
} CuePoint;
```

dwIdentifier enthält eine individuelle Nummer, mit deren Hilfe der cue point referenziert werden kann. dwPosition legt die Reihenfolge der Wiedergabe fest. Je kleiner der Wert, desto eher wird er abgearbeitet. In fccChunk wird die ID des Chunks angegeben der die Audiodaten enthält auf die sich der cue point bezieht. Normalerweise wird hier der Wert “data“ eingetragen. dwChunkStart und dwBlockStart werden auf 0 gesetzt wenn es sich um unkomprimiertes PCM mit nur einem Data-Chunk handelt. Sie werden nur bei Kompression benutzt oder wenn es mehr als einen Data-Chunk gibt. dwSampleOffset gibt den Offset des cue points bezüglich der in dwChunkStart und dwBlockStart angegebenen Position an. Im Normalfall, also keine Komprimierung und nur ein Data-Chunk, gibt dwSampleOffset direkt den Offset im Data-Chunk an.

Playlist-Chunk

Der Playlist-Chunk beschreibt eine Widergabeliste anhand definierter cue points. Der Cue-Chunk enthält alle cue point und der Playlist-Chunk beschreibt, wie mit diesen cue points bei der Wiedergabe verfahren wird.

```
struct {
        char[4]         “plst”;
        long            chunkSize;
        long            dwSegments;
        Segment         Segments[];
} Playlist-Chunk;
```

dwSegments gibt die Anzahl der Segmente an, gefolgt von den einzelnen Segment Strukturen.

```
struct {
        long            dwIdentifier;
        long            dwLength;
        long            dwRepeats;
} Segment;
```

dwIdentifier identifiziert das Segment analog wie beim Cue-Chunk. Dieser Wert muss der gleiche Wert sein, wie der dwIdentifier des entsprechenden cue points. In dwLength verbirgt sich die Anzahl der Samples in diesem Segment. Zu Beachten ist dabei, dass dwSampleOffset des zugehörigen cue points die Startposition des Segmentes angibt. dwRepeats beschreibt, wie oft dieses Segment bei der Wiedergabe wiederholt werden soll. Ein Wert von eins bedeutet einmaliges Abspielen. Die Reihenfolge der Segmente in diesem Chunk ist beliebig, da die Wiedergabereihenfolge bereits in den cue points festgelegt wird.

Associated Data List-Chunk

Dieser Chunk enthält Namen oder Beschreibungen die bestimmten cue points zugeordnet werden können.

```
struct {
        char[4]             ”list”;
        long                chunkSize;
        char[4]             ”adtl”;
} ListHeader;
```

Der Associated Data List-Chunk enhält keine eigenen Daten, sondern nur Subchunks. Für Wav-Dateien sind drei Subchunks von Bedeutung: Label, Note und Labeled Text.

Label-Chunk

Mit dem Label-Chunk kann eine Bezeichnung zu einem cue point zugeordnet werden.

```
struct {
        char[4]             ”labl”;
        long                chunkSize;
        long                dwIdentifier;
        char                dwText[];
} LabelChunk;
```

dwIdentifier ist die Referenz auf den entsprechenden cue point. dwText enthält den null-terminierten String.

Note-Chunk

Der Note-Chunk dient dazu, einen cue point zu beschreiben. Er ist genau so aufgebaut wie der Label-Chunk, abgesehen davon, dass “labl“ durch “note“ ersetzt wird.

Labeled Text-Chunk

Mit dem Labeled Text-Chunk kann ein Abschnitt der Wiedergabeliste beschrieben werden.

```
struct {
        char[4]             ”ltxt”;
        long                chunkSize;
        long                dwIdentifier;
        long                dwSampleLength;
        long                dwPurpose;
        short               wCountry;
        short               wLanguage;
        short               wDialect;
        short               wCodePage;
        char                dwText[];
} LabelTextChunk;
```

dwIdentifier hat die gleiche Funktion wie beim Label- und Note-Chunk. dwSampleLength gibt die Länge des zu beschreibenden Segments an. Die weiteren Werte beschreiben den Typ der Beschreibung, Land, Sprache, Dialekt, Kodierung und schließlich auch den Text.

Zusammenfassung

Die zentralen Themen dieses Kapitels waren einerseits die Umsetzung von Prinzipien der Audiokodierung und andererseits die Vorstellung eines konkreten Audioformates, dem WAV-Format. Dabei wurde gezeigt, welche grundlegenden Verfahren zur Kodierung und Kompression angewendet werden. Ziel war dabei mehr das Verständnis der Verfahren als die genauen Details. Zwar gibt es inzwischen neuere und effektivere Kompressionsverfahren, wie MP3 und AAC, jedoch werden die grundlegenden Technologien auch in Zukunft noch Anwendung finden. Ein großer Abschnitt wurde dem WAV-Format gewidmet. Da es beim WAV-Format in erster Linie um möglichst einfache Speicherung und Audiobearbeitung geht, wurde hauptsächlich auf unkomprimierte WAV-Dateien eingegangen. Mit dem vermittelten Wissen sollte es möglich sein eigene WAV-Dateien zu erstellen.

Abkürzungen

PCM	Pulse Code Modulation
DPCM	Differential Pulse Code Modulation
ADPCM	Adaptive Differential Pulse Code Modulation
AIFF	Audio Interchange File Format
IMA	Interactive Multimedia Association
AAC	Advanced Audio Coding
WMA	Windows Media Audio
SNR	Signal To Noise Ratio
RIFF	Ressource Interchange File Format

Literatur

[1] Tim Kientzle: A Programmer's Guide to Sound, Addison Wesley,1998
[2] Horst Zander: Das PC- Tonstudio, Franzis, 1998
[3] Datenkompression allgemein: http://www.rasip.fer.hr/research/compress
[4] Psychoakustisches Modell:
http://www.physio.mu-luebeck.de/vorlesung/_private/sinne/sinnesphysiologie.pdf
[5] Kompression bei WAV-Dateien:
http://www.icculus.org/SDL_sound/downloads/external_documentation/wavecomp.htm

Anhang

Einige mögliche Codes für Datenformate in WAV-Dateien :

0x0001 = Microsoft Pulse Code Modulation (lineares PCM)
0x0002 = Microsoft ADPCM

0x0005 = IBM CVSD
0x0006 = Microsoft A-Law
0x0007 = Microsoft μ-Law
0x0010 = OKI_ADPCM
0x0011 = Intel DVI ADPCM
0x0012 = Videologic MEDIASPACE_ADPCM
0x0013 = SIERRA_ADPCM
0x0014 = Antex Electronics Corp G723 ADPCM
0x0015 = DSPGroup DIGISTD
0x0016 = DSPGroup DIGIFIX
0x0020 = Yamaha ADPCM
0x0021 = SONARC
0x0020 = DSPGroup Truespeech
0x0023 = Echo Speech Corp 1
0x0024 = Audiofile AF36
0x0025 = APTX
0x0026 = Audiofile AF10
0x0030 = Dolby AC2
0x0031 = Microsoft GSM610
0x0033 = Antex Electronics Corp ADPCME
0x0034 = Control Res VQLPC
0x0035 = DSPGROUP DIGIREAL
0x0036 = DSPGROUP DIGIADPCM
0x0037 = Control Res CR10
0x0038 = Natural MicroSystems VBXADPCM
0x0040 = Antex Electronics Corp G721 ADPCM
0x0050 = Microsoft MPEG
0x0101 = IBM μ-Law Format
0x0102 = IBM A-Law Format
0x0103 = IBM AVC ADPCM
0x0200 = Creative Labs ADPCM
0x0300 = Fujitsu FM Towns SND

Synthetisierung von Musik und MIDI

von

Sebastian Frielitz

Einleitung

Es gibt verschiedene Möglichkeiten Musikstücke auf digitalen Medien zu speichern. Einerseits kann man sie als Welle speichern, andererseits aber auch als computerisierte Instrumente.
Warum sind verschiedene Soundformate wichtig?
Zunächst einmal wegen der Größe der Musikdaten. Man möchte heutzutage mehr als nur ein paar Töne auf dem Computer speichern. Auch Computerspiele sind heutzutage so produziert, das man mehrere Stunden davor sitzt und durchgehend mit Musik versorgt wird. Diese Musik soll sich auf den Spielverlauf anpassen und soll abwechslungsreich gehalten werden. In diesem Falle ist Musik von computerisierten Instrumenten eine Möglichkeit, mit der mit wenig Aufwand die Musik variiert werden kann ohne viel Speicherplatz zu belegen. Es wird im allgemeinen weniger Speicher benötigt einzelne Noten für die computerisierten Instrumente zu speichern, als die Musik in Wellenform.
Andererseits soll vielleicht ein großes, oder sehr spezielles Klangspektrum an Musik abgedeckt werden. Dafür eignet sich die Musik in Wellenform besser, da dabei keine Beschränkung in der Wahl der Instrumente besteht, wie es im ersten Fall zu Gunsten des Speicherplatzes und der Bandbreite der Fall ist.

Im Weiteren möchte ich darauf eingehen, wie man Musik mit computerisierten Instrumenten synthetisiert. Daraus werde ich dann auf den MIDI Standard eingehen, seine Strukturierung und den Aufbau einer Standard MIDI Datei. Und im Zuge dessen ein Codebeispiel einer MIDI Datei heranziehen und erläutern.

Synthetisieren von Musik

Musikformate, bei denen keine Wellen, sondern einzelne Noten und Instrumente gespeichert werden, bezeichnet man als ‚notenbasierte' Musikformate im Gegensatz zu den ‚wellenbasierten' Musikformaten.

Es gibt zwei Techniken die notenbasierte Soundformate motivieren. Zum einen der Versuch den Klang eines physisch existierenden Instrumentes möglichst original zu duplizieren. Manche Synthesizerhersteller betreiben viel Aufwand, damit ihr Pianoklang genau so klingt wie ein bestimmtes physisches Piano.
Die zweite Technik ist Klänge zu produzieren, die musikalische Qualität haben, aber nicht als Instrument existieren.

Instrumente duplizieren

Um das zu bewerkstelligen werden zunächst mehrere Samples von dem zu simulierenden Instrument erstellt, etwa für jeden Ton eines. So sind alle Töne des Instrumentes erfasst und es kann jeder Ton des Instrumentes gespielt werden.
Würde tatsächlich jeder Ton als eigenes Sample erfasst werden, würde das zu einem sehr hohen Speicherbedarf führen. Um das zu vermeiden, wird ein Ton gesamplet und in der Frequenz modifiziert um die umliegenden Töne zu erzeugen. Diese Technik wird ‚Pitch Shifting' genannt.
Da jedoch reale Instrumente nicht den gleichen Klang für verschiedene Tonhöhen haben, wird für jede Oktave ein Sample angelegt und jeweils auf dieses das Pitch Shifting angewandt.

Spielt ein physisches Instrument einen Ton, so fängt dieser meistens mit einem lauten Teil an, hält sich dann eine Weile auf einer etwas niedrigeren Lautstärke und blendet dann langsam aus. Diese Art der Amplitudenvariation wird ‚envelope' genannt.
Um synthetisierte Töne realistisch klingen zu lassen, wird ein so genanntes ‚Envelope Control' eingesetzt.
Die verbreitetste Form des ‚Envelope Control' nennt sich ADSR, was ausgeschrieben *Attack, Decay, Sustain, Release* heißt. Siehe Abbildung 1.

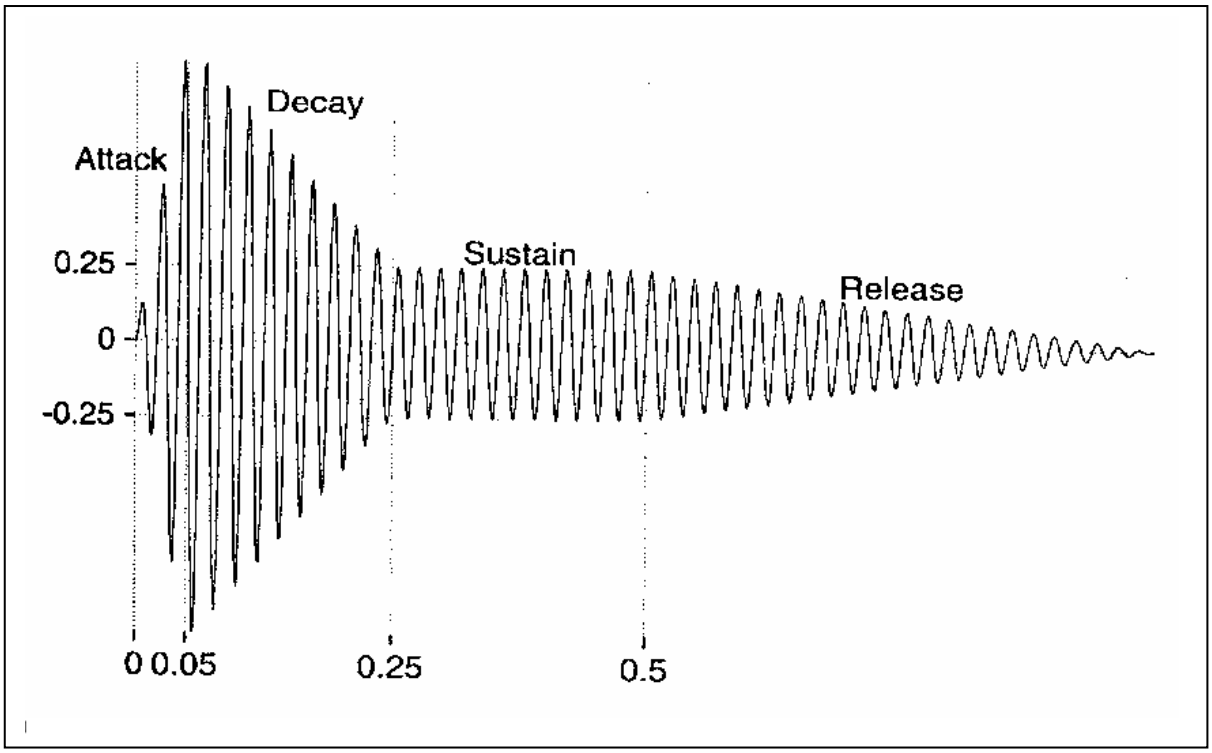


Abbildung 1 [T.K.98]

Die Umsetzung des ‚Envelope Control' kann auf verschiedene Weisen geschehen. Eine Mögliche ist, in hardware Synthesizern, die einen eigenen Lautstärken Controler haben, die Lautstärke beim Spielen der Note zu justieren.

Natürlich gibt es noch andere Formen des ‚Envelope Control'. Zum Beispiel der von Trompeten, oder Violinen, die ihren Ton über lange Zeit halten können. Oder ein Tremolo, bei dem schnell von einem Ton zum anderen hin und wieder zurück gewechselt wird.

Klänge erzeugen

Wenn Klänge künstlich erzeugt werden klingen sie sehr mechanisch, da sie von einem Computer so exakt gespielt werden, wie es kein physisches Instrument tut. Außerdem klingen künstliche Töne, die man lange spielt auch nicht besonders gut, da man, um Speicherplatz zu sparen, nur eine kurze Schleife des Tones oft wiederholt.
Ein Ansatz um den künstlichen Klang zu verbessern, nennt man ‚Plucked Strings'.

Plucked Strings Algorithmus:
Geräusche enthalten eine breite Varianz an Frequenzen. Es werden viele verschiedene zufällige Samples genommen und daraus ein Klang erzeugt.
Wird auf einer Samplingrate von 44.100 Hz gearbeitet, und werden 441 zufällig Samples in einer Tabelle erstellt und diese in einer Endlosschleife wiederholt, so ist das Ergebnis tatsächlich sehr musikalisch. Man erhält einen klaren 100 Hz Ton.
Die übliche Variante dieses Algorithmus ersetzt jedes Sample dieser Tabelle durch den Mittelwert aus seinem Vorgänger und dem Sample selber, damit der Ton sich angenehm entwickelt. Dadurch wird erreicht, das der Ton geglättet wird.

Dieser einfache Algorithmus braucht sehr wenig Speicher und ist relativ schnell. In der Praxis angewandte Varianten dieses Algorithmus verwenden zwar nicht nur zufällige Samples, aber ansonsten sind sie dem oben beschriebenen Algorithmus sehr ähnlich.

Ein Problem dieses Algorithmus ist, das Hohe Töne sehr kurz werden. Um dem entgegenzusteuern wird kein einfacher Mittelwert zur Entwicklung des Tones verwendet, sondern ein ausgewogener Mittelwert. Was auch noch zu bedenken ist: Wenn in den Zufallssamples kein Sample mit dem Wert 0 dabei ist kann es passieren das der Ton nie ausklingt. Um das zu verhindern wird das niedrigste Sample genommen und von allen Samples subtrahiert um einen Null Sample zu erhalten.

MIDI Standard

MIDI bedeutet ausgeschrieben: Musical Instrument Digital Interface.

Mit MIDI ist es möglich auf einfache Art und Weise mit einem einzigen Keyboard eine ganze Variation an Instrumenten zu kontrollieren.

Im Jahre 1988 wurde das Standard MIDI Datei Format von den Mitgliedern der *MIDI Manufacturers Assosiation* (MMA; http://www.midi.org) angenommen. Diese Spezifikation legte eine Vorgehensweise zur Speicherung von MIDI Ereignissen (im folgenden auch MIDI Events genannt) und Zeitinformationen (im folgenden auch Timing Informations genannt) in einer Datei fest.
Damit aber der Klang von einer Flöte auf dem einen Keyboard nicht wie ein Piano auf einem anderen klingt wurde zusätzlich *General MIDI* spezifiziert. General MIDI spezifiziert 175 Standard Instrumente und noch weitere Eigenschaften. Doch darauf werde ich weiter in Abschnitt 4.2 eingehen.

Organisation von MIDI Dateien

Eine MIDI Datei besteht aus einer Reihe von Chunks wie im RIFF Format festgelegt. Abbildung 2 gibt hier einen Überblick.

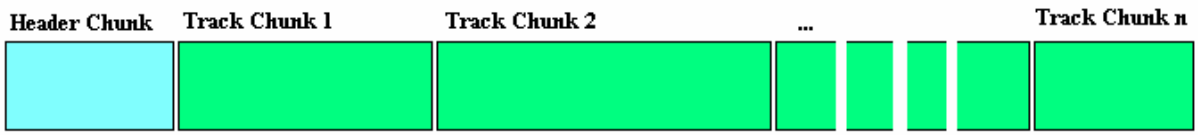


Abbildung 2

Jeder Chunk beginnt mit 4 Zeichen die ihn identifizieren. Im Standard MIDI Format gibt es zwei Arten von Chunks. Zum einen gibt es die Header Chunks, die jeweils am Anfang einer Standart MIDI Datei stehen und die Track Chunks, die dem Header Chunk folgen und die Musikdaten der Datei enthalten.

Der **Header Chunk** besteht aus folgendem:

4 Byte: MThd
2 Byte: Datei Typ
2 Byte: Anzahl der Tracks
2 Byte: Zeit Format

Der **Track Chunk** besteht aus folgendem:

4 Byte: MTrk
Danach folgen die Trackdaten

Ein Track Chunk enthält immer nur einen Track.
MIDI ist in mehreren Tracks organisiert, die jeweils die Daten für ein einzelnes Instrument enthalten.

Es gibt drei Typen von MIDI Dateien.
Typ 0 MIDI Dateien enthalten nur einen einzigen Track.
Typ 1 MIDI Dateien enthalten mehrere Tracks. Diese Tracks werden alle nebeneinander abgespielt. Das heißt, der MIDI Player muss alle Tracks in einen *Event Stream* (Ereignis Liste) bringen, um sie abzuspielen.
Der dritte und weniger häufigere Typ ist die Typ 2 MIDI Datei. Sie enthält auch mehrere Tracks, aber ohne Zusammenhang.

Beispiel eines MIDI Header Chunk

Um den Aufbau von Chunks in MIDI Dateien zu veranschaulichen ist in Abbildung 3 eine kleine MIDI Datei in HexaDezimal Code angegeben. Und in den folgenden die Erklärung.

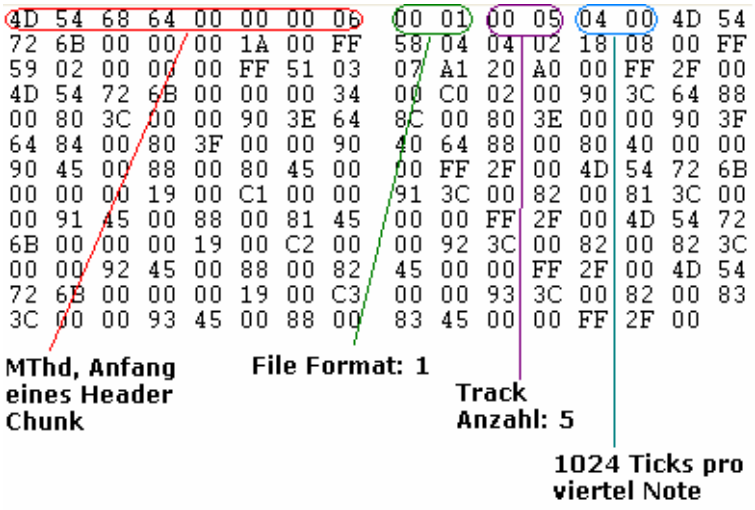


Abbildung 3

Zur Erklärung von Abbildung 3 sind in den Abbildungen 4 - 6 Auszüge aus dem MIDI Standard.
Abbildung 4 zeigt wie der Header Chunk einer MIDI Datei laut dem Standard auszusehen hat.

```
2. Header Chunk

      The header chunk appears at the beginning of the file, and describes the
file in three ways. The header chunk always looks like:

4D 54 68 64 00 00 00 06 ff ff nn nn dd dd
```

Abbildung 4

```
ff ff is the file format. There are 3 formats:

0 - single-track
1 - multiple tracks, synchronous
2 - multiple tracks, asynchronous
```

Abbildung 5

```
nn nn is the number of tracks in the midi file.

dd dd is the number of delta-time ticks per quarter note. (More about this
later)
```

Abbildung 6

In Abbildung 5 und 6 sind die Platzhalten aus Abbildung 4 beschrieben. Hier stehen die vier ‚f' für 2 Byte, die das MIDI Datei Format beschreiben.
In Abbildung 6 stehen die vier ‚n' für 2 Byte, die die Anzahl der Tracks angeben und die vier ‚d' für 2 Byte, die das Zeitformat festlegen.
In Abbildung 3 sind diese 4 Elemente (farblich hervorgehoben).

MIDI Track

Ein MIDI Track ist etwas anderes als ein MIDI Channel, auch wenn es für Multi Track MIDI Dateien üblich ist jeden Track auf einem anderen Channel zu spielen.
Ein MIDI Track ist einfach eine Liste von Ereignissen (MIDI Events) mit jeweils einem Zeitintervall vorangestellt.
Das Zeitintervall des jeweiligen Ereignisses gibt an, wieviel Zeit verstreichen muss, bis das Ereignis ausgeführt werden darf. Eine andere Möglichkeit wäre den absoluten Zeitpunkt, an dem das Ereignis auftreten soll, einzutragen. Standard MIDI jedoch legt sich auf den einen Zeitintervall zwischen den Ereignissen fest.

Abarbeitung von MIDI Tracks

Wie oben gesagt, besteht jeder MIDI Track aus mehreren aneinandergereihten MIDI Ereignissen, die jeweils durch ein Zeitintervall getrennt sind. Jeder Chunk in einer MIDI Datei hat eine explizite Länge, auf die genau geachtet werden muß, um zu wissen, wann die Trackdaten zuende sind.

Wenn es sich nicht um den ersten Track in der Datei handelt, dann müssen die MIDI Ereignisse an der richtigen Stelle in der MIDI Ereignisliste eingetragen werden. Das ist etwas kompliziert, da man anhand der Zeitintervalle der Ereignisse die richtige zeitliche Stelle in der Ereignisliste finden muß.

MIDI Ereignisse (MIDI Events)

MIDI Ereignisse sind Datenpakete, die musikalische Aktionen definieren. Das erste Byte gibt den Ereignis Typen an, das höchste Bit (das linkeste im Byte) ist bei Status Bits ist immer 1.
Darauf folgt eventuell, wenn es sich um ein bereitzustellendes Ereignis handelt, zwei Channelbytes, welche den Kanal angeben, auf dem das Ereignis ausgeführt werden soll.
Danach kommen nur noch Daten Bytes, die jeweils immer mit einer 0 im ersten Bit anfangen.

In Abbildung 7 ist eine kleine Übersicht über MIDI Ereignisse, wo auch gut zu erkennen ist, dass die Status Bytes, die hier immer die ersten beiden Ziffern im Hex Code bilden, wenn man sie Binär schreibt mit einer 1 im ersten Bit beginnen.

Als Beispiel: Note Spielen
$(8c)_{16} = (\underline{1}000\ 1100)_2$

TABLE 22.2 MIDI Events as Used in Standard MIDI Files

Code (hex)	Description
`8c nn vv`	Note *nn* off with velocity *vv* on channel *c*
`9c nn vv`	Note *nn* on with velocity *vv* on channel *c*
`Ac nn vv`	*Polyphonic key pressure aftertouch* Change the pressure (usually vibrato) of note *nn* (that's already playing) on channel *c* to *vv*
`Bc mm ss`	Change mode *mm* on channel *c* to *ss*
`Cc ii`	*Program change* Select instrument sound *ii* for channel *c*
`Dc vv`	*Channel pressure aftertouch* Change the pressure of all notes playing on channel *c* to *vv*
`Ec ff cc`	*Pitch wheel change* Change the pitch of all notes playing on channel *c* by a certain proportion; *ff* holds the least significant 7 bits, *cc* the most significant
`F0 length data`	*System exclusive* (SOX) The *length* is a variable-length integer indicating the length of the following *data*.
`F7 length data`	*Special system exclusive* The *length* is a variable-length integer indicating the length of the following *data*.
`FF tt length data`	*Meta event of type* tt The *length* is a variable-length integer indicating the length of the following *data*.

Abbildung 7 [T.M.98]

Die Kanäle sind von 0 – 15 und die Instrumente von 0 – 127 durchnummeriert

Angemerkt hierzu sei nur noch, das die oben aufgelisteten Ereignisse nicht mit denen im weiter unten beschriebenen MIDI ‚Wire Protocol' übereinstimmen.

Um das Beispiel weiterzuführen, kommt nun ein Track Chunk nach dem Header Chunk. Er beginnt wie in der Standard MIDI Spezifikation festgelegt (Abbildung 8), markiert mit dem roten Kasten in Abbildung 9.

```
3. Track Chunks

The remainder of the file after the header chunk consists of track chunks.
Each track has one header and may contain as many midi commands as you like.
The header for a track is very similar to the one for the file:

4D 54 72 6B xx xx xx xx

As with the header, the first 4 bytes has an ascii equivalent. This one is
MTrk. The 4 bytes after MTrk give the length of the track (not including the
track header) in bytes.
```

Abbildung 8

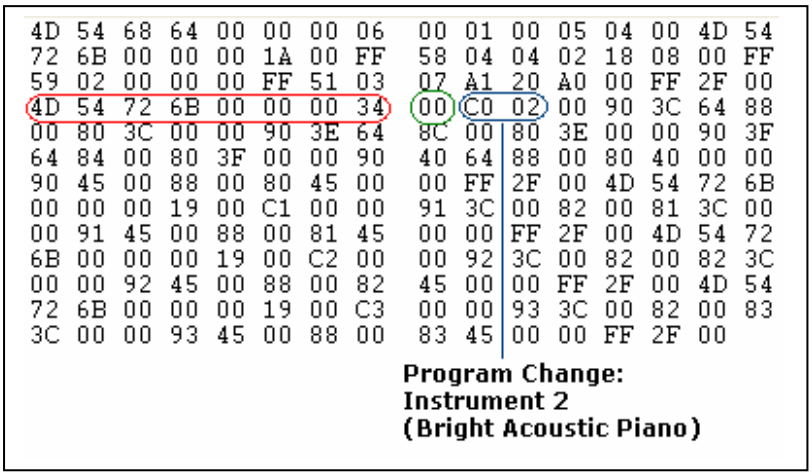


Abbildung 9

Die grüne Markierung ist das Zeitintervall, das vor jedem Event steht, und in diesem Falle Null ist, da es keinen Sinn macht nach einem Chunk Header zu warten.
Blau markiert ist dann der Aufruf des Ereignisses, wie anhand der Tabelle aus Abbildung 7 zu ersehen ist. Hier wird also ein Instrumentenwechsel vorgenommen, und zwar wird das Instrument Nummer 2, im MIDI Standard als Bright Acoustic Piano festgelgt, gewählt.

In Abbildung 10 wird auf das nächste Ereignis eingegangen. Zunächst ist wieder ein Zeitintervall (grün markiert) mit Länge Null zu finden, und dann blau markiert das Spielen einer Note.
Danach folgt wieder ein Zeitintervall, jetzt mit rot markiert, das diesmal eine Länge von 1024 Ticks aufweist (Zahlenformat, siehe Integer variabler Länge für Zeitintervalle, Abschnitt 4.1.3.1), damit die Note auch ein Weile gespielt wird. Darauf folgt das Note-ausschalten-Ereignis und das Spielen des Tones ist beendet.

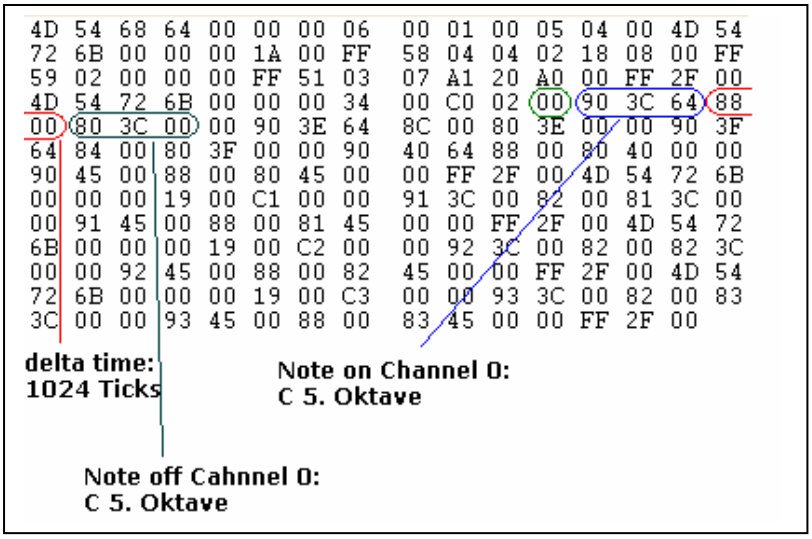


Abbildung 10

Integer variabler Länge für die Zeitintervalle

Um Speicherplatz in MIDI Dateien zu sparen wird in denselben ein schlankeres Zahlenformat für die Zeitintervalle definiert.
Bei diesem Zahlenformat können kleine Zahlen in einem Byte gespeichert werden und größere Zahlen, bis zu 32 Bit Zahlen, in mehreren Bytes.

In jedem Byte werden nur 7 Bit zur Zahlencodierung verwendet. Das ‚most significant bit' (msb), also das linkeste Bit eines Bytes wird dazu verwendet um zu überprüfen ob noch ein weiteres Byte zu der Zahl gehört, oder ob es sich um das letzte Byte der Zahl handelt. Enthält das msb eine 1 so folgt noch ein weiteres Byte, enthält es eine 0, so ist es das letzte Byte. Zur Veranschaulichung befindet sich ein Beispiel in Abbildung 11.

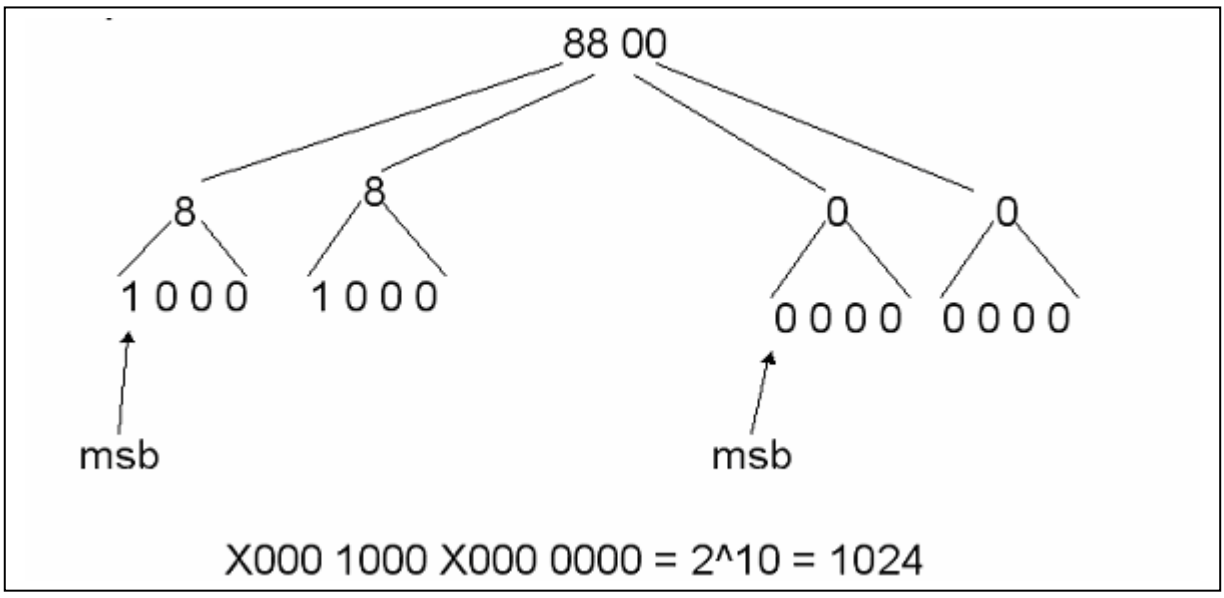


Abbildung 11

Meta Ereignisse (Meta Events)

In einer MIDI Datei sind noch mehr Daten enthalten als zeitgesteuerte Klänge. Diese Ereignisse werden als Meta Ereignisse (Meta Events) bezeichnet. Meta Ereignisse befinden sich meistens am Anfang eines Tracks, doch sie können an einer beliebigen Stelle im Track auftreten.

Meta Ereignisse können Copyright Bemerkungen enthalten, aber auch Daten, die für das Abspielen der MIDI Datei wichtig sind.
In einer Typ 1 MIDI Datei ist der erste Track nur für bestimmte Meta Ereignisse reserviert, wie Vorzeichen, Takt und Tempo Informationen. Weitere Meta Ereignisse sind am Anfang eines Tracks zu finden, man findet den Track Titel und die Track Sequenznummer hier.

Ein Meta Ereignis beginnt stets mit dem Byte $FF_{(16)}$, gefolgt vom Typ Byte und dem Längen Byte, danach folgen die Daten Bytes.

Die MIDI Meta Ereignisse können in der Standard MIDI Spezifikation nachgelesen werden. Hier wird jetzt nur auf zwei besondere Meta Ereignisse eingegangen. Zum einen das ‚Time Signature Event', und das ‚Sequenz-Specific Meta Event'.

Time Signature

Dieses Ereignis legt Takt und Geschwindigkeit des MIDI fest. Es besteht aus 4 Byte. Die ersten zwei Byte geben den Takt an, zum Beispiel 3/4 oder 6/8 Takt. Um diesen Takt zu bekommen wird das erste Byte direkt ausgelesen, also 3 bei einem 3/4 Takt oder 6 bei einem 6/8 Takt. Das zweite Byte gibt den Exponenten von 2 an. Das bedeutet also, das bei einem 3/4 Takt das zweite Byte eine 2 wäre, da 2^2 gleich 4 ist, bei einem 6/8 Takt wäre also das zweite Byte eine 3.
Das dritte Byte im Time Signature Ereignis gibt die Anzahl der MIDI ‚clocks' pro Metronomklick an. Ein MIDI ‚clock' ist 1/24 einer MIDI viertel Note, und die Dauer einer MIDI viertel Note ist wiederum durch das MIDI Tempo Meta Ereignis festgelegt.
Das vierte und letzte Byte des Time Signature Meta Ereignisses: Da eine viertel Note auf Papier nicht unbedingt der viertel Note in einem MIDI entspricht, kann man in diesem Byte angeben, wie viele notierte 32stel Noten in einer MIDI viertel Note gespielt werden.

Sequenz-Specific Meta Event

Mit diesem Ereignis können einer MIDI Datei zusätzliche Daten zugeordnet werden. Um Verwirrungen vorzubeugen, werden dieses Ereignis in den ersten 4 Bytes signiert. Das erste Byte enthält immer eine 0 und die drei folgenden Bytes die ‚Manufacturer ID'. So würde ein MIDI von Microsoft zum Beispiel die Signatur 0 0 6 5 enthalten.

MIDI Timing

Bevor eine MIDI Datei abgespielt werden kann, müssen die ‚delta time tick counts', in denen die Zeitintervalle zwischen den MIDI Ereignissen notiert sind, in eine nutzvolle Zeiteinheit umgewandelt werden. MIDI verwendet zwei Techniken um die Dauer eines Ticks zu spezifizieren.

Ist der ‚Time Format' Code im Header Chunk negativ, also ist die Zahl aus Byte 5 und 6 im Header Chunk negativ, so liegt in der MIDI Datei das Zeitformat SMPTE vor

Ist der ‚Time Format' Code positiv, dann ist das Zeitformat nach ‚Musical Tempo' zu bestimmen.

SMPTE (Society of Motion Picture and Television Engineers)

Bei dieser Technik werden die Teile von Stunden, Minuten, Sekunden und ‚Frames' gezählt.
Ein ‚Frame' ist hier die Zeitdauer eines einzelnen Bildschirmaufbaus in einem Kino-, oder Fernsehdisplay und reicht von einem 1/24stel bis zu einem 1/30stel einer Sekunde.

Diese Technik wird weithin von Video und Audio Entwicklern verwendet um, Ton und Bild präzise zu spezifizieren und synchronisieren.

Hierbei müssen zunächst die Anzahl der Frames pro Sekunde festgelegt werden.
SMPTE wird dann durch vier Werte festgelegt, die die Stunden, Minuten, Sekunden und Frames angeben, die bis dahin verstrichen sind. Zum Beispiel: 01:23:43:21 würde eine Zeit von einer Stunde, 23 Minuten, 43 Sekunden und 21 Frames bedeuten.

Tempo-Based Timing

Beim Tempo-Based Timing handelt es sich um ein etwas intuitiveres Zeitformat. Bei diesem Zeitformat werden die Zeitintervalle in Schläge pro Minute, genannt ‚beats per minute' und im folgenden abgekürzt durch bpm, angegeben.
Ein beat entspricht meist einer viertel Note und die meisten MIDI Dateien haben ein Tempo von 80 bis 200 bpm. Der Standardwert von MIDI liegt bei 120 bpm.
Zu diesem Zeitformat wird im Header wie oben beschrieben eine positive Zahl ausgelesen. Diese Zahl steht für die Ticks pro beat, in den meisten Fällen also die Ticks pro viertel Note.
Bei diesem Format kann das Tempo während des Stücks verändert werden, was durch das ‚Tempo Meta Event' geschieht.

General MIDI

General MIDI ist ein Standard, der festlegt was Geräte, die MIDI tauglich sind leisten müssen.
Diese Geräte müssen 16 Kanäle haben, die Kanäle 1 bis 9 und 11 bis 16 sind Melodie Kanäle, der Kanal 10 ist der Rhythmus Kanal, dieser muß die 47 Rhythmusinstrumente unterstützen.
MIDI Produkte müssen 16 simultane Noten auf den Melodiekanälen und 8 simultane Noten auf den Rythmuskanälen spielen können.
MIDI Prokukten müssen auch noch eine Latte andere Eigenschaften aufweisen, wie Controler für die Modulation und die Lautstärke nur als einzelne Beispiele genannt.
Der General MIDI Standard legt 175 Instrumente fest, doch er legt nicht fest, wie diese funktionieren müssen. So könnte zum Beispiel ein Produkt die Lautstärke eines Tones linear ansteigen lassen und ein anderes logarithmisch.

Weitere MIDI Standards sind:

Downloadable Samples

Downloadable Samples ermöglichen es, ein beliebiges MIDI von einem Netzwerk herunterzuladen. Der *Downloadable Samples Standard Level 1.0* (DLS-1). Hier speichert eine DLS-Datei einen odere mehrere Intrument Samples zusammen mit einer Anzahl an Angaben, wie diese Samples gespielt werden. Diese Samples liegen im WAVE Format gespeichert vor.

Base / Extended MIDI

Um es für PC Hardware Händler einfacher zu machen, hat Microsoft die Multimedia PC Spezifikation, abgekürzt MPC, veröffentlicht. Nach dieser Spezifikation gibt es zwei Level, zum einen die Low-Cost Systeme, die einen sogenannten ‚Base Multitimbral Synthesizer' verwenden und die High-End Systeme, die einen sogenannten ‚Extended Multitimbral Synthesizer' verwenden.
Diese beiden Level sind bekannt als Base MIDI und Extended MIDI.

Base MIDI bietet 6 simultane Melodie Noten auf 3 Instrumenten und 3 simultane Rhythmus Noten auf 3 Instrumenten.
Extended MIDI bietet 16 simultane Melodie Noten auf 9 Instrumenten und 16 simultane Rhythmus Noten auf 9 Instrumenten.

Zum Vergleich:
General MIDI bietet 16 simultane Melodie Noten auf 16 Instrumenten und 8 simultane Rhythmus Noten auf 8 Instrumenten.

Zudem muss man sagen, das Base- und Extended MIDI viel an Bedeutung verloren haben und heutzutage eher General MIDI verwendet wird.

Die beiden Standardlevel Based- und Extended MIDI verwenden verschiedene Kanäle, so das ein Stück sowohl als Base MIDI, als auch als Extended MIDI in einer Datei untergebracht werden kann. So ist es möglich, ein Stück in beiden Versionen abzulegen, so das jemand der Extended MIDI hat in den vollen Genuß seiner 16 Kanäle kommt und jemand der nur Base MIDI hat, das Stück auch abspielen kann.
Das Problem dabei ist, das General MIDI alle Kanäle benutzt, und somit die beiden Versionen gleichzeitig abspielen würde.
Um dem vorzubeugen, hat Microsoft einen Standard entwickelt, mit dem es möglich ist die MPC MIDI Dateien zu erkennen. Danach müssen diese Dateien unter anderem mit dem ‚Sequenz-Specific-Meta Event' markiert werden.

MIDI Wire Protocol

Wenn MIDI über ein Netz übertragen werden soll, möchte man nicht erst die gesamte Datei herunterladen müssen um sie abzuspielen. So müssen die Ereignisse also schon in der richtigen Reihenfolge vom Netz geladen werden um sie spielen zu könne wenn sie an der Reihe sind.
Beim MIDI Wire Protocol sieht die MIDI Datei etwas anders aus, als eine Standard MIDI Datei.
Es gibt eine Anzahl von System Ereignissen, die es bei Standard MIDI nicht gibt. Außerdem verwendet das MIDI Wire Protocol den ‚Running Status' um effizienter zu sein. Dieser

‚Running Status' verwendet ein gelesenes Status Byte weiter, wenn nach dem folgenden Ereignis kein neues Status Byte kommt, sondern sofort wieder Datenbytes.
Im MIDI Wire Protocol werden Ereignisse in ‚real-time' und ‚non-real-time' Ereignisse unterteilt. ‚Real-time' Ereignisse müssen an einer bestimmten Stelle in der Ereignisseliste erscheinen um den gewünschten Effekt zu erzielen. Beispiele für ‚real-time' Ereignisse sind Synchronisationsereignisse wie Start und Continue. Die ‚non-real-time' Ereignisse müssen nicht unbedingt an einer bestimmten Stelle auftreten. Sie können irgendwann geladen werden und später interpretiert.

Zusammenfassung

Im Großen und Ganzen kann man sagen, daß MIDI ein interessantes Konzept ist, Musik auf MIDI Geräten Digital zu speichern und zu kontrollieren. MIDI hat eine sehr intuitive Herangehensweise zur Synthetisierung von Musik, die dem Komponieren von Musik auf Papier sehr nahe kommt. Das einmalige speichern von Instrumenten und dann lediglich durch Ereignisse gesteuertes Spielen dieser Instrumente ist sehr effizient. Doch ist es ein wenig unkomfortabel, das die Musikinstrumente bei MIDI nicht wirklich auf allen Systemen die gleichen sind. Und das selbst wenn dies der Fall ist, der Standard kleine Lücken hat, durch die die Stücke nicht immer genau gleich klingen. Wenn man MIDIs auf verschiedenen MIDI Produkten abspielt, würde man sich als Komponist eines Stückes doch wünschen, das es genau so klingt, wie man es komponiert hat, egal wo man es abspielt. Außerdem schränkt der MIDI Standard sehr ein. Es stehen einem zwar 175 Instrumente zur Verfügung, doch möchte man über diesen Standard hinaus, ist das mit Schwierigkeiten verbunden, da über den Standard hinaus gehende Instrumente natürlich nicht auf fremden MIDI Produkten zu finden sind und eventuell erst nachgerüstet werden müssten. Gesang und Sprache ist mit MIDI auch nicht möglich. Zudem ist die kleine Dateigröße nicht mehr so entscheidend, so das man diese Mankos nicht mehr so gerne in Kauf nimmt.

Abkürzungen

MIDI	Musical Instrument Digital Interface
MMA	MIDI Manufacturer's Association
msb	most significant bit (das höchste Bit eines Bytes)
bpm	beats per minute (Schläge pro Minute)
MPC	Multimedia PC Spezifikation

Literatur

MIDI Standard Anhang

[T.K.98] Tim Kientzle, Aprogrammer's Guide to Sound, Addison Wesley, 1998

Videokodierung mit H261 / H263.

von

Lars Wolter

Einleitung

Videocodecs gibt es recht viele, und die meisten die so bekannt sind wie DivX oder MPEG dienen dem kodieren von Filmen in Dateien. Hier wird ein Videostandard behandelt, der hauptsächlich für Videokonferenzen eingesetzt wird. Während die Codecs für Videodateien meist als fertige Kodier- und Dekodierprogramme daherkommen, existiert für H.26x eine Definition des Formats, so wie ein Referenzdekoder. Welche Features eine Implementierung eines Enkoders oder Dekoders unterstützt, liegt in den Händen der Entwickler. Dies macht es möglich dass es viele verschiedene Programme gibt, die miteinander kommunizieren können.
Um diese wichtige Stellung des Videostandards bei Videokonferenzsystemen, werde ich am Ende auch noch auf das Konferenzprotokoll H.323 eingehen.

Geschichte und Hintergrund

Beide Standards wurden von der ITU (International Telecommunication Union) entwickelt.
Die ITU ist eine Organisation die in Genf sitzt und zu der „United Nations System of Organizations“ gehört. Die ITU gibt es bereits seit dem 17. Mai 1865. Sie wurde gegründet um die Telegraphie international zu regeln. Im laufe der Zeit legte sie diverse Standards im Bereich der Telekommunikation fest.
Die ITU gründet für verschiedene Projekte kleinere Forschungsgruppen. Diese beschäftigen sich mit einem Thema wie die Videokodierung über mehrere Jahre. Im Anschluss legen sie die Ergebnisse der World Telecommunication Standardization Conference (WTSC) vor. Diese stimmt dann über einen neuen Standard ab. Die Entwicklung ist im Vergleich zu anderen Bereichen in der Telekommunkation und Informatik recht langwierig. Die WTSC tagt nur alle vier Jahre. Abstimmungen und Beschlüsse außerhalb der Konferenzen sind selten, aber auch möglich.
Die Entwicklung von H.261 begann bereits 1988 und wurde dann im März 1993 von der WTSC abgesegnet. Im Anschluss begann man mit der Entwicklung von H.263 (Anhänge bis H). Seit 1997 gibt es einen erweiterten Standard, der als H.263+ bezeichnet wird (Anhänge bis T). Aktuell ist H.263++ seit 2000 (Anhänge U,V,W). Danach kam noch Anhang X (2001). Mittlerweile beschäftigt man sich mit dem Nachfolger H.264 der sehr stark auf den Vorgänger aufbaut, aber auch einige Grundlegende Neuerungen enthält. „Die Kodiereffizienz von H.264/AVC übertrifft alle existierenden Standards etwa um den Faktor zwei“ [CT6/03].

Die ersten Gedanken zur Entwicklung dieser Videostandards galten den Bildtelefonen, die durch die Verwendung digitaler Telefonverbindungen (ISDN) zu dieser Zeit möglich wurden. Diese damaligen Systeme mussten mit wenig Bandbreite zurechtkommen. Es standen pro Leitung 64 kBit zur Verfügung, es konnten jedoch mehrere Leitungen zusammengeschaltet werden. Doch nur größere Firmen konnten es sich leisten 10 oder gar 30 Leitungen für eine Videokonferenz zu blockieren. Außerdem war die damalige Rechenkraft der meisten Geräte stark eingeschränkt. Auch Hochspezialisierte DSPs die einen kontinuierlichen Videostrom kodieren konnten waren selten und teuer. Somit war die erste Version H.261 auch sehr sparsam ausgelegt. Erst als man mit H.263 auch gezielt die Videokonferenzen über Netzwerke mit ihren hohen Bandbreiten einbezog, gestaltete sich der Standard flexibler. Die Rechner in einem Netzwerk haben zusätzlich die Möglichkeit wesentlich aufwendigere Berechnungen zur Kodierung des Videostroms durchzuführen.

H.261

Übersicht

Dieser Videostandard wurde entwickelt um Videokonferenzen über ISDN Leitungen durchzuführen. Dies heißt dass er darauf ausgelegt wurde bei einer niedrigen Datenrate von 64kBit noch Video zu Übertragen. Da es auch die Möglichkeit gibt, mehrere ISDN-Leitungen zusammenzuschalten, lässt sich der Standard auch nach oben skalieren. Es werden allerdings nur zwei Bildformate unterstützt. Der erste ist CIF mit 352 x 288 Pixeln. Dieser ist für Datenraten bis 256kBit geeignet. Das zweite Format ist QCIF mit 176x144 Pixeln. Das zweite begnügt sich auch mit 64kBit. Die Beschränkung der Datenrate auf die Werte ist nötig um Audio und Videosynchronisation zu garantieren. Der Codec speichert die Bilddaten in Form von Luminanz- (Helligkeit) und Chrominanzwerten (Farbwerte) auch als YUV bekannt. Wie bei den meisten Codecs arbeitet dieser auch auf Pixelblöcken mit 8x8 Pixeln. Diese Blöcke werden wiederum zu Makroblöcken zusammengefasst. Ein Makroblock besteht demnach aus vier Y-Blöcken und den zwei dazugehörigen Chrominanz-Blöcken (C_B und C_R).
Das man zu vier Helligkeitswerten nur ein paar Farbwerte speichert ist eine gängige Vorgehensweise, da das menschliche Auge Farbveränderungen nicht so stark wahrnimmt wie eine Änderung der Helligkeit.
Die Komprimierung bei H.261 erfolgt durch Quantisierung. Dazu wird jeder Block einer Diskreten Kosinus Transformation unterzogen und anschließend mit einer festen Quantisierungsmatrix komponentenweise multipliziert.
Die anschließend anfallenden Daten werden dann im ZigZag Verfahren angeordnet und Verlustfrei komprimiert. Die stärkste Komprimierung erfolgt durch Motion Kompensation, wobei für jeden Makroblock ein Vektor gespeichert werden kann mit Werten zwischen -15 und +15. So kann ein entsprechender Encoder im vorherigen Bild um die Position des aktuellen Makroblocks nach einem ähnlichen Block von Pixeln suchen. Findet er einen, werden nur die Differenzen quantisiert, oder im optimalen Fall nur der Offsetvektor gespeichert.

Details

Fehlerkorrektur

Bei H.261 gibt es noch eine fest integrierte Fehlerkorrektur. Beim späteren H.263 ist diese nur noch optional. Die Fehlerkorrektur arbeitet auf festen Blöcken des kodierten Bitstroms. Sie fasst 492 Datenbits zusammen und berechnet dazu 18 Paritätsbits, das ganze wird dann noch mit einem Framebit und einem Füllindikator Bit zu einem 512 Bit Block komplettiert. Der verwendete Algorithmus zur Fehlerbehandlung ist der BCH (Bose, Chaudhuri and Hocquenghem[INFCOD]).

Anhänge

Anhang A spezifiziert Genauigkeit der IDCT.

Die Genauigkeit der DCT ist ein sehr wichtiger Faktor bei der Videokodierung, da es der jeweiligen Implementierung überlassen wird, wie der genaue Kodier- und Dekodier-Algorithmus aufgebaut ist. Unter Umständen wird auch Hardware zur Erledigung dieser Aufgaben herangezogen. Diese ist dann evtl. beschränkt auf eine geringe Präzision.
In diesem Anhang wird eine Referenzimplementierung der IDCT angegeben. Wobei eine Präzision von 64 Bit Floatingpoint Werten (Double) bei der gesamten Berechnung verwendet

wird. Dazu werden maximale Fehlertoleranzen spezifiziert, um die die eigene Implementierung von dieser Referenzimplementierung abweichen darf.

Anhang B spezifiziert einen hypothetischen Referenzdecoder.

Dieser Anhang beschreibt nur die Puffereigenschaften eines Decoders. Dieses kann genutzt werden, um die Bandbreite eines Stromes zu analysieren, und wie die Puffer des Decoders diese Daten verarbeiten.

Anhang C Codec Verzögerungsmessung

Dieser Anhang beschreibt wie man Probleme bei Verzögerungen zwischen Encoder und Decoder entgegenwirken kann. Dies ist besonders wichtig um den Ton zum Bild zu synchronisieren, da verschiedene Implementierungen auch verschiedene Verzögerungen aufweisen können.

Anhang D Standbildübertragung

Hier gibt es eine Erweiterung, die es ermöglicht Bilder in einer Auflösung von 704 x 576 zu Übertragen. Dabei wird das Bild auf vier Bilder verteilt. Dieser Modus ist sehr nützlich wenn man in einen Videostrom z.B. eine Folie oder die Abbildung eines Tafelbild einbinden will.

H.263

Übersicht

H.263 ist eine Weiterentwicklung von H.261 und ist die Konsequenz aus immer mehr Bandbreite und der großen Verfügbarkeit des Internets. Deswegen ist H.263 hauptsächlich für Videokonferenzen per Standleitungen oder über das Internet gedacht. Im Gegensatz zu H.261 ist es wesentlich flexibler geworden. Dazu unterstützt es mehrere Auflösungen.

Bildformat	Waagerechte Auflösung	Senkrechte Auflösung
sub-QCIF	128	96
QCIF	176	144
CIF	352	288
4CIF	704	576
16CIF	1408	1152

Als weitere Neuerung gegenüber H.261 haben wir eine genauere Motion Compensation. Diese arbeitet mit Vektoren die Werte zwischen -16 und 15.5 annimmt. Dabei werden Halbpixel verwendet. Dazu gibt es neue Bildtypen die es erlauben, dass mit den Vektoren Bilder referenziert werden, die in der Zukunft und/oder Vergangenheit liegen (PB Frames).
Ein weiterer Punkt ist die Struktur der Daten. Zusätzlich zu den Blöcken und Makroblöcken kommen die GOBs dazu (GOB = Group of Blocks). Diese beinhalten eine Zeile von Makroblocks und erlaubt die Angabe von Parametern die für alle enthaltenen Makroblocks gilt. Später wird auch noch eine Alternative zu den GOBs definiert, die Slices.

Die Fehlerkorrektur wurde in einen Anhang verlegt (Anhang H) da bei der Verwendung von H.263 meistens darunter liegende Protokolle verwendet werden, die eine Fehlerkorrektur enthalten, wie TCP/IP.
Mithilfe der ganzen Extras die in den Anhängen (von A bis X) definiert werden, erreicht man schon fast die Funktionalität von MPEG4 (bezogen auf die reine Videokodierung). Die ITU hat mittlerweile einen MPEG4 Standard spezifiziert der auf H.263 aufbaut. Dieser heißt H.264.

Details

Struktur der Bilddaten

Picture

PSC	TR	PTYPE	PQUANT	CPM	PSBI	TRB	DBQUANT	PEI	PSUPP	PEI	GOBs	ESTUF	EOS	PSTUF

Jedes Bild in dem Videostrom verfügt über einen Header. Dieser enthält einen Startcode (PSC) mit 22 Bit, so dass es möglich ist aus einem Beliebigen Bitstrom auch mithilfe des ebenfalls 22Bit großen EOS den Anfang eines Bildes zu erkennen. Es folgen Angaben über den Bildtyp I-,P-,PB-,B-,EI-,EP-Frame und das Bildformat (PTYPE)
Dazu kommen einige Daten die erkennen lassen, welche Extras bei der Kodierung des Bildes verwendet wurden Anhänge: D,E,F,G,I,J,K,N,P,Q,R,S und T diese Angaben sind im PTYPE angegeben, bzw. in dem alternativen PLUSPTYPE.
Zusätzlich stehen hier Quantisierungsinformationen (DQUANT, DBQUANT) die für das gesamte Bild verwendet werden. Nun folgen die eigentlichen Daten, die GOBs. Das Bild wird beendet durch den End of Sequence Code (EOS). Das ganze kann dann noch mit Hilfe von Stopfbits (ESTUF und PSTUF) ausgerichtet werden.

GOB

GSTUF	GBSC	GN	GSBI	GFID	GQUANT	Makroblock Daten

Der erste GOB darf keinen Header besitzen, da für diesen die Werte im Bild Header verwendet werden. Bei den anderen GOBs wird auf jeden Fall ein Header mit Stopfbits (GSTUF) und dem 17Bit Startcode (GBSC) verwendet. Dieser kann für spezielle Fälle mit einem neuen Quantisierer Wert der für alle nachfolgenden Blöcke des aktuellen Bildes gilt (GQUANT) belegt werden. Die weiteren Werte in diesem Header gelten zur Identifikation eines Substreams (Anhang C). Während die GOB Nummer (GN) die GOBs durchzählt.
Alternativ zu den GOBs können auch Slices verwendet werden (Anhang K).

Makroblock

Wie auch bei H.261 setzt sich ein Block aus vier Werten für den Luminanzwert und je einem Wert für die Chrominanzwerte zusammen.
Auf der Ebene des Makroblocks wird entschieden wie ein Bild kodiert wird. Es gibt zwei Grundsätzliche Möglichkeiten INTRA oder INTER. Bei der INTRA Kodierung werden einfach die Bildinformationen innerhalb des Blocks kodiert. Bei der INTER-Kodierung benutzt man einen Bereich aus einem anderen Bild als Referenz. Dazu gibt es die Möglichkeit einen (oder unter Nutzung der Anhänge D oder F auch mehrere) Offset-Vektor anzugeben.

Dieser Vektor wird verwendet um nur die Differenzen zwischen den referenzierten Bilddaten und denen des aktuellen Makroblocks zu kodieren.

Picture type	MB type	Name	COD	MCBPC	CBPY	DQUANT	MVD	MVD2-4
INTER	not coded	-	X					
INTER	0	INTER	X	X	X		X	
INTER	1	INTER+Q	X	X	X	X	X	
INTER	2	INTER4V	X	X	X		X	X
INTER	3	INTRA	X	X	X			
INTER	4	INTRA+Q	X	X	X	X		
INTER	5	INTER4V+Q	X	X	X	X	X	X
INTER	stuffing	-	X	X				
INTRA	3	INTRA		X	X			
INTRA	4	INTRA+Q		X	X	X		
INTRA	stuffing	-		X				

Die Tabelle zeigt welche verschiedenen Möglichkeiten es für Makroblöcke gibt. Der oberste ist ein spezieller INTER-Block der verwendet wird, wenn sich der Bildausschnitt den der Makroblock enthält sich nicht geändert hat. Dieser enthält dann auch keine Blockdaten.
Die Werte in der Tabelle beschreiben den Makroblockheader, ein X kennzeichnet, ob der Eintrag im Header vorhanden ist.
Dabei beschreiben COD und MCBPC den Typ des MB. DQUANT modifiziert den momentanen Quantisierungswert für diesen Makroblock und MVD ist ein Motionvektor.

Block

Der kodierte Block besteht aus 8 Bit DC und den AC Werten kodiert mit Codes variabler Länge (VLC). Die ACs werden mithilfe der ZigZag Methode angeordnet. Die folgende Tabelle zeigt einen Teil der VLC Tabelle die für Blöcke verwendet wird.

INDEX	LAST	RUN	LEVEL	BITS	VLC CODE
0	0	0	1	3	10s
1	0	0	2	5	1111 s
2	0	0	3	7	010101s
…					
100	1	39	1	13	000001011110s
101	1		1	13	000001011111s
102	ESCAPE			7	0000011

Die AC Werte werden auf eine spezielle Art und Weise kodiert. Man verwendet ein Tripel bestehend aus einem LAST, einen RUN und einem LEVEL Wert. LAST beschreibt dabei ob dies der letzte Wert ist der nicht Null ist. RUN beschreibt die Anzahl Nullen vor diesem Wert. LEVEL ist der Wert des ACs. So werden also nur ACs größer als Null gespeichert. Man erinnere sich, bei der Quantisierung werden die meisten Werte auf Null reduziert.

Die Tabelle enthält nun eine Reihe von Kombinationen die besonders häufig auftreten und ordnet ihnen kurze Bitsequenzen zu. Das letzte Bit dieser Sequenz (durch s gekennzeichnet) gibt immer das Vorzeichen des Levels an.
Alle nicht vorhandenen Kombinationen werden mit 22 Bit als Escape(7), LAST(1), RUN(6) und LEVEL(8) kodiert.

Anhänge

Für die Anhänge A und B siehe H.261

Anhang C Considerations for Multipoint

Hier werden Möglichkeiten definiert, die es erlauben, mehrere parallele H.263 Verbindungen zu synchronisieren. Es gibt auch Kommandos um einzelne Ströme anzuhalten oder wieder fortfahren zu lassen.

Anhang D Unrestricted Motion Vector mode

Erlaubt es die Vektoren bei der Motioncompensation auch außerhalb von Bildbereiche zeigen zu lassen. Außerdem erhält man die Möglichkeit längere Vektoren anzugeben. Die maximale Länge hängt nun von dem Bildformat ab. Bei CIF16 liegen die Werte zwischen -256 und 255,5.

Anhang E Syntax-based Arithmetic Coding mode

Hier wird eine alternative Methode zu den Codes variabler Länge angegeben. Die Arithmetische Kodierung ist eine sehr effiziente Art der Datenkomprimierung, so dass man eine Verringerung der Datenrate um 5% annehmen kann. Allerdings zum Preis eines höheren Kodierungsaufwands. Im Anhang ist der Code für die Kodierung als auch für die Dekodierung angegeben. Außerdem die kleinen Änderungen in der Struktur der Blockdaten.

Anhang F Advanced Prediction mode

Dieser spezielle Modus erlaubt vier Verschiebungsvektoren pro Makroblock. Er aktiviert auch gleichzeitig die Möglichkeit Vektoren zu definieren die über Bildgrenzen hinausgehen, ohne den Modus aus Anhang D zu verwenden.
Für die Luminanzwerte werden drei der vier Vektoren verwendet, wobei die folgende Formel zeigt wie:
$P(x,y) = (q(x,y) \cdot H0(i,j) + r(x,y) \cdot H1(i,j) + s(x,y) \cdot H2(i,j) + 4)/8$
q,r,s sind die drei referenzierten Pixel.
Hx(i,j) ist eine vorgegebene Gewichtungsmatrix, wobei verschiedene Matrizen gewählt werden, je nachdem von wo referenziert wird. Für die Chrominanzdaten wird der vierte Vektor verwendet, wobei der normale Algorithmus verwendet wird.

Anhang G PB-frames mode

Dieser Modus spezifiziert einen neuen Bildtyp. Diese PB-Kombination beschreiben zwei Bilder, die zusammen kodiert werden. Dabei wird die Struktur der Makroblöcke geändert. Diese enthalten jetzt 12 Blöcke, zuerst die 6 P-Blöcke anschließend die 6 B-Blöcke. Die P-Blöcke können sowohl INTRA-kodiert als auch INTER-kodiert sein, während die B-Blöcke immer INTER-kodiert werden. Das besondere ist nun, das die B-Blöcke sowohl das vorherige Bild

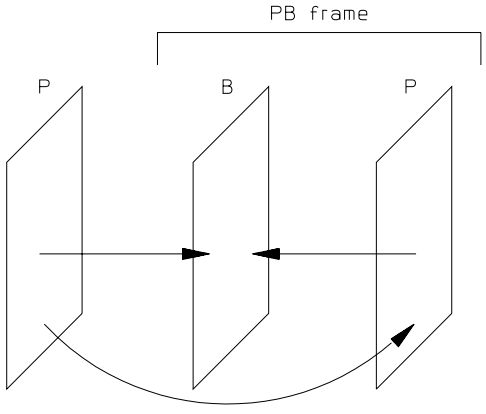


Abbildung 1 PB-Frame

referenzieren können. Als auch das P-Bild in der PB-Kombination also ein zukünftiges Bild. Aber nur Pixel die innerhalb des Makroblocks referenziert werden, werden bidirektional bestimmt. Es wird also der Durchschnitt der beiden P-Bilder-Pixel bestimmt. Pixel außerhalb des Makroblocks werden weiterhin nur normal vorhergesagt.
Das P-Bild der PB-Kombination kann nur aus dem Bild vorher referenzieren.

Anhang H Forward Error Correction for coded video signal

Dieser Anhang beschreibt die optionale Verwendung einer Fehlerkorrektur. Diese arbeitet nach dem gleichen Prinzip wie die in H.261 verwendete Fehlerkorrektur.

Anhang I Advanced INTRA Coding mode

Dieser Modus ist eine Alternative zum normalen Kodiermodus eines INTRA Makroblocks. Hierbei kann man die Informationen aus benachbarten Blöcken innerhalb desselben Makroblockes verwenden. Weiterhin wird eine alternative VLC Tabelle angegeben, und zwei weitere ZigZag Modi.

1	2	3	4	11	12	13	14
5	6	9	10	18	17	16	15
7	8	20	19	27	28	29	30
21	22	25	26	31	32	33	34
23	24	35	36	43	44	45	46
37	38	41	42	47	48	49	50
39	40	51	52	57	58	59	60
53	54	55	56	61	62	63	64

1	5	7	21	23	37	39	53
2	6	8	22	24	38	40	54
3	9	20	25	35	41	51	55
4	10	19	26	36	42	52	56
11	18	27	31	43	47	57	61
12	17	28	32	44	48	58	62
13	16	29	33	45	49	59	63
14	15	30	34	46	50	60	64

Anhang J Deblocking Filter mode

In diesem Modus verwendet der Kodierer einen Deblocking Filter um die Blockartefakte zu verhindern die oft auftreten. Dieser Filter wird auf das kodierte und wieder dekodierte Bild angewandt welches der Kodierer für die Vorhersagen verwendet. Die Bildqualität erhöht sich am besten wenn dieser Modus zusammen mit denen in Anhang D und F verwendet wird.

Anhang K Slice Structured mode

Dieser Modus ersetzt die Struktur der GOBs durch Slices. Slices erlauben es dem Kodierer die Makroblöcke nicht zeilenweise zu gruppieren, sondern sie in einem Rechteck oder sogar beliebig zu gruppieren. Die Slices sind ähnlich aufgebaut wie die GOBs. Sie enthalten nur zusätzliche Informationen darüber welche Makroblöcke darin enthalten sind.

Anhang L Supplemental Enhancement Information Specification

Dieser Anhang erlaubt dem Kodierer zusätzliche Informationen zu den Bildern anzugeben. Das sind Informationen, die z.B. dafür sorgen dass das momentane Bild oder ein Bildteil angehalten wird (freeze request). Außerdem werden so Bilder oder auch Bildabschnitte zur externen Speicherung markiert.
Man kann auch so genannte Chroma-Keying Informationen speichern. Das erlaubt es das bestimmte Bildbereiche anhand ihrer Farbe Transparenz geschaltet werden, wobei auch Halbtransparenz unterstützt wird. Man kann so z.B. einen Hintergrund festlegen, und dann den Vordergrund animieren. Diese werden dann mit Hilfe des Chroma Keys zusammengemischt.

Anhang M Improved PB-frames mode

Hier wird der Modus, der PB-Bildkombinationen verwendet erweitert, indem es möglich ist sich zu entscheiden in welche Richtung man die Bildvorhersage verwendet. So kann man entweder nur das vorherige referenzieren, nur das in der PB-Kombination enthaltene oder eben beide.

Anhang N Reference Picture Selection mode

Dieser Modus erlaubt es dem Dekoder die erfolgreiche Dekodierung zu bestätigen oder den Fehlschlag der Dekodierung an den Enkoder zu senden. Dieser kann dann einzelne nötige Informationen an den Dekoder schicken. Dazu werden zwei Kanäle eingerichtet über die diese Kommunikation erfolgt. Referenzwerte in den Bildern und Blöcken werden benutzt um sich zu verständigen.

Anhang O Temporal, SNR, and Spatial Scalability mode

Diese Skalierungsmodi erlauben es die Komprimierung der Daten weiter zu skalieren. Dazu werden neue Bildtypen eingeführt. B-Bilder EP-Bilder und EI-Bilder. Die B-Bilder kennen wir bereits aus den beiden PB Modi. Sie können jetzt alleine Auftreten, und können auch in beide Richtungen referenzieren. Sie können allerdings nicht selbst als Referenz verwendet werden. Dabei werden die B Bilder im Datenstrom so angeordnet, dass sie Datentechnisch hinter allen Bildern liegen die sie referenzieren. Dies ermöglicht die zeitliche Skalierung.
Die andere Methode der Skalierung arbeitet mit mehreren Schichten. Die Basisschicht enthält nur die nötigsten Informationen. Eine weitere Schicht, die Enhancement Schicht enthält dann weitere Daten um die Bildqualität zu erhöhen. Diese Bilder sind dann die EI und EP Bilder die analog zu den I und P Bildern vorhergesagt sind oder absolute Daten enthalten. Sie verwenden jedoch beide die Informationen aus der Basisschicht. Das ganze kann noch erweitert werden, indem mehr als nur zwei Schichten verwendet werden.
Weiterhin wird definiert auf welche Art und Weise die verschiedenen Daten der Schichten in dem Datenstrom übertragen werden.

Anhang P Reference Picture Resampling

Das Resamplen von Refrenzbildern erlaubt es ein Bild zu Transformieren und so Verzerrungen von Bildern effizient zu kodieren. So könnte ein Zoom beispielsweise kodiert werden. Man kann so auch den Wechsel von Auflösungen kodieren.

Anhang Q Reduced-Resolution Update mode

Hierbei überträgt man für das aktuelle Bild nur Bilddaten in niedriger Auflösung, während die Referenzdaten in hoher Auflösung vorliegen. Es kommt also nur bei INTER Kodierten Blöcken zum Einsatz. Die niedrig aufgelösten Daten werden Supersampled (in der Auflösung hochgerechnet) und dann mit den Vollaufgelösten Referenzdaten verrechnet.

Anhang R Independent Segment Decoding mode

Bitte in [ITU263] nachlesen.

Anhang S Alternative INTER VLC mode

Bietet eine alternative VLC Tabelle für die INTER-Kodierung.

Anhang T Modified Quantization mode

Bitte in [ITU263] nachlesen.

Anhang U specifying an optional Enhanced Reference Picture Selection (ERPS) mode

Bitte in [ITU263] nachlesen.

Anhang V specifying an optional Data Partitioned Slice (DPS) mode

Bitte in [ITU263] nachlesen.

Anhang W specifying optional Additional Supplemental Enhancement Information

Bitte in [ITU263] nachlesen.

Anhang X Profiles and Levels Definition

Definiert Profile die es erlauben Standardisierte Kodierer und Dekodierer zu definieren.
Die folgenden Profile existieren.

Kodieren eines INTRA Bildes

Um nun nicht nur Die Struktur eines Datenstromes im H.26X Format zu beschreiben gibt es diesen Abschnitt. Er beschreibt die Vorgehensweise wie man den Videostrom kodiert. Dabei entwickeln wir hier keinen kompletten Encoder sondern betrachten nur die wichtigsten Elemente. So kann man diese Übersicht aber auch genauso für andere Videostandards verwenden die ähnlich arbeiten, wie z.B. MPEG.
Wenn man allerdings einen voll funktionsfähigen Kodierer und Dekodierer schreiben will wird man nicht drum herum kommen die volle Spezifikation des jeweiligen Standards zu lesen. Die vorgehende Übersicht reicht dazu in keinem Fall aus.

Bits schreiben

Etwas das man die ganze Zeit über benötigt um einen Videostrom zu schreiben ist eine bequeme Methode einzelne Bits zu schreiben. Diese Funktionalität braucht man um die diversen Header zu schreiben und auch um die VLCs zu schreiben die bei der Kodierung der eigentlichen Bilddaten anfallen.

```
#define PUT_BITS(bits, nBits, nTempBits, tempBits, bitStream) { \
    nTempBits += (nBits);\ // gezählte Bits erhöhen
    if (nTempBits > 32)  {\ // wenn sie nicht mehr ins INT passen
      u_int extra = (nTempBits) - 32;\ // Anzahl extrabits
      tempBits |= (u_int)(bits) >> extra;\ // Bits dazupacken
      *(u_int*)bitStream = tempBits;\ // Alle bits speichern
      bitStream += sizeof(u_int);\ // Stream weitersetzen
      tempBits = (u_int)(bits) << (32 - extra);\ // extrabits…
      nTempBits = extra;\ // …ins temp peichern und anzahl setzen
    } else\ // ansonsten alle bits den temporären hinzufügen
      tempBits |= (u_int)(bits) << (32 - (nTempBits)); \
}
```

Dieses Define ist eine effiziente Methode wenn man C verwendet. Man kann das natürlich auch in Java implementieren. Für alle die nicht Wissen was ein Define in C ist, gibt es den folgenden kleine Einschub.
Ein Define ist eine Möglichkeit innerhalb von Sourcecode ein Textstück durch ein anderes zu ersetzen. Das ganze bietet aber noch mehr Möglichkeiten. In diesem Fall ist es so, dass an der Stelle im Code wo steht PUT_BITS(a,5,32,b,c) der komplette Block eingesetzt wird wobei alle vorkommen von bits im Block durch a, nBits durch 5 usw. ersetzt werden. Man könnte das ganze auch als Funktion implementieren, doch so verzichtet man auf den

Funktionsoverhead, der bei dieser recht wichtigen Funktion einiges Aufwand für den Prozessor bedeuten würde.

Bildformat umwandeln

Jetzt kann angefangen werden das erste Bild zu kodieren. Dazu muss das Bild erstmal umgewandelt, transformiert und quantisiert werden. Die Bilder die man bekommt können in unterschiedlichen Formen vorliegen. Am einfachsten wäre es, wenn sie bereits im YUV Format vorliegen. Ist das nicht der Fall muss jeder Pixel umgewandelt werden. Das andere häufige Format indem Bilder vorliegen können ist das RGB Format. Die Umwandlung von RGB nach YUV wird an dem nebenstehenden Bild von Bernie und Ert gezeigt.
Für die Umrechnung gibt es einen einfachen Algorithmus:

Abbildung 18 RGB Bild

```
Y=(0.257*R)+(0.504*G)+(0.098*B)+16
C_R=(0.439*R)-(0.368*G)-(0.071*B)+128
C_B=-(0.148*R)-(0.291*G)+(0.439*B)+128
```

R, G, B sind die Pixelwerte der RGB Darstellung, Y der Luminanzwert und C_R und C_B die Chrominanzwerte. Nach der Umwandlung erhält man drei Bilder. Die zwei Chrominanzbilder werden nur mit einem viertel der Auflösung benötigt.

Abbildung 21 Luminanzdaten

Abbildung 20 C_R

Abbildung 19 C_B

DCT anwenden

Das sind die Daten die jetzt kodiert werden sollen. Das erste was gemacht wird ist die Transformation mit der DCT, der Diskreten Kosinus Transformation. Es muss also die folgende Funktion implementiert werden:

$$f(x,y) = \tfrac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C(u)\,C(v)\,F(u,v)\cos[\pi(2x+1)u/16]\cos[\pi(2y+1)v/16]$$

Über die Implementierungsmöglichkeiten der DCT und auch der iDCT könnte man ein Buch schreiben. Die effizientesten Implementierungen sind direkt auf spezielle Hardware angepasst und verwenden dort spezielle Funktionen um z.B. mehrere Werte parallel zu berechnen. Solche für 3dNow oder SSE angepassten Funktionen sollen hier nicht gezeigt werden. Die direkteste Implementierung würde wohl so aussehen:

```
for(int i=0;i< 8;++i) {
    for(int j=0;j< 8;++j) {
        float koeffizient=0.0f;
        for(int k=0;k< 8;++k){
            for(int l=0;l< 8;++l){
                koeffizient = koeffizient + bildBlock[k*8+l]*
                    cos(i*PI*((2*k)+1)/16)*
                    cos(j*PI*((2*l)+1)/16);
            }}
        if ((i==0)&&(j!=0))koeffizient=koeffizient*1/sqrt(2);
        if ((j==0)&&(i!=0))koeffizient=koeffizient*1/sqrt(2);
        if ((j==0)&&(i==0))koeffizient=0.5*koeffizient;
        dctBlock[i*8+j]=0.25*koeffizient;
}}
```

Diese verschachtelten Schleifen sind sehr ineffizient, die Transformation eines kompletten Bildes liegt im Sekundenbereich (auf einem 800MHz Pentium III) das ist natürlich für Echtzeit Videokompression ungeeignet. Doch man kann schon durch einfache Umstellung des Algorithmus eine Effizienzsteigerung erhalten.

```
void init_dct(){
    int i, j;
    double s;
    for (i=0; i<8; i++){
        s = (i==0) ? sqrt(0.125) : 0.5;
        for (j=0; j<8; j++)
            c[i][j] = s * cos((PI/8.0)*i*(j+0.5));
    }
}
```

Dazu verwendet man einen einmaligen Setupschritt, indem eine Matrix erzeugt wird in der vorberechnete Kosinus-Werte gespeichert werden. Auch die teuere Wurzeloperation wird in diesen Setupschritt ausgelagert. Im Anschluss muss man dann nicht mehr die komplette Rechnung in der inneren Schleife vornehmen, sondern kann einfach mit den Werten der hier erzeugten Matrix multiplizieren.
Diese Optimierung steigert die Effizienz bereits um ein vielfaches, so dass man wieder an Echtzeitbetrieb denken kann.
Die folgenden Operationen laufen jetzt auf der Blockebene ab. Danach werden sie zu Makroblöcken zusammengefasst.

Quantisierung

Nachdem die Daten transformiert wurden, müssen diese jetzt quantisiert werden. Durch diesen Schritt erhält man einen Großteil der Komprimierung, da hier Informationen weggeworfen werden.

Der DC Wert wird in 8er Schritten Quantisiert, die AC Werte werden auf Werte zwischen -255 und +255 begrenzt. Die Schrittgröße wird definiert durch QUANT + DQUANT (Diese werden durch den Picture, GOB bzw. Makroblockheader definiert.
Man sollte hier einen Analyse Schritt einbauen der alle Quantisierten Blöcke in diesem Makroblock betrachtet, der den optimalen Wert für diesen Makroblock herausfindet. Man kann das dann auch noch ausdehnen um den besten Wert für den aktuellen GOB bzw. den aktuellen Slice zu bestimmen.
Es kann auch einer der erweiterten Modi benutzt werden. Der in Anhang T definierte modifizierte Quantisierungsmodus erlaubt es Werte bis 2040 als Resultat der Quantisierung zu verwenden.
Die Formel für die Quantisierung lautet:
Level = (Coeff/QUANT-1) /2 wenn QUANT ungerade
Level = ((Coeff+1)/QUANT-1) /2 wenn QUANT gerade
Diese Level müssen nun angeordnet werden. Dazu bediet man sich des ZigZag Verfahrens, oder einer der Varianten die einem der erweiterte INTRA Kodierungsmodus gibt der in Anhang I spezifiziert ist. Diese Umordnung der Daten sorgt dafür, dass die weniger wichtigen Werte als letztes aufgeführt werden. Dadurch fallen am Ende einige Nullen an die dann wegkomprimiert werden können.

Kodierung

Dies geschieht durch die VLCs die die Koeffizienten als Tripel der Form (LAST,RUN,LEVEL) speichert. Falls Anhang T verwendet wird muss auch die für diesen Fall definierte Escapesequenz verwendet werden, um die Werte bis 2040 in konstante 11 Bit Level zu verpacken.
Weiterhin ändert sich das Kodierverhalten falls Anhang I oder Anhang E verwendet wird.
Für die Implementierung greift man auf das PUT_BITS define zurück. Dieses schreibt die VLCs in den Datenstrom. Um die VLC Tabelle zu verwalten legt man am besten einen Hash an, der unter Eingabe des Tripels den VLC Code liefert. Ist der Code nicht vorhanden muss das Tripel direkt gespeichert werden.

Kodieren eines INTER-Bildes

Bisher haben wurde nur ein komplettes Bild INTRA kodiert. Doch den größten Gewinn in der Datenrate erreicht man mit Motion Compensation. Denn dadurch werden Informationen aus einem vorherigen Bild benutzt um das aktuelle Aufzubauen. Das funktioniert, weil es in den meisten Filmen einen unbewegten Hintergrund gibt, und sich so nur ein Teil des Bildes ändert. Aber auch bei einem langsamen Kameraschwenk über eine Szenerie verschieben sich die Bilddaten nur berechenbar.
Dazu werden in H.263 die P-Bilder verwendet. In ihnen kann es einige Makroblöcke geben, deren Inhalt nicht INTRA-kodiert wird sondern INTER-kodiert. Für die INTER-kodierung muss man einen Bildteil im vorherigen Bild finden der dem aktuellen sehr ähnlich ist. Dazu sucht man in der näheren Umgebung des aktuellen Blocks zuerst. Hier gibt es viele Ansätze sehr schnell einen Bereich zu finden, doch dieses sind gut gehütete Geheimnisse der Firmen die solche Enkoder schreiben. Hat man ein Bildausschnitt gefunden, bildet man einen Vektor der den Offset zwischen der Position des aktuellen Makroblocks und des gefundenen darstellt. Diesen Vektor speichert man im Header des Makroblocks. In den einzelnen Blocks transformiert man nun nur noch die Differenz zwischen dem eigentlichen Block und dem gefundenen Offsetblock. Da die Differenzen wesentlich kleiner sind als die eigentlichen Werte erzeugt die DCT auch wesentlich mehr Nullen bzw. kleinere Werte die sich besser

quantisieren lassen. Wenn die Differenz sehr klein ist, kann man auch einen Makroblock ohne Daten speichern, dann wird einfach nur der Block aus dem vorherigen Bild kopiert.
Damit man nicht allzu große Überraschungen erlebt, sollte man als Referenzbild ein Bild nehmen, das man selbst durch den Dekoder geschickt hat, den nur so kann man dasselbe Ergebnis erzielen wie am Ende beim Dekodieren.
Diverse Anhänge geben einem hier weitere Möglichkeiten, wie größere Suchradien für Referenzbereiche, da man längere Offsetvektoren speichern kann. Außerdem die Modi die weitere Bildtypen definieren. So kann man z.B. die bidirektionalen B-Frames verwenden.

H.323

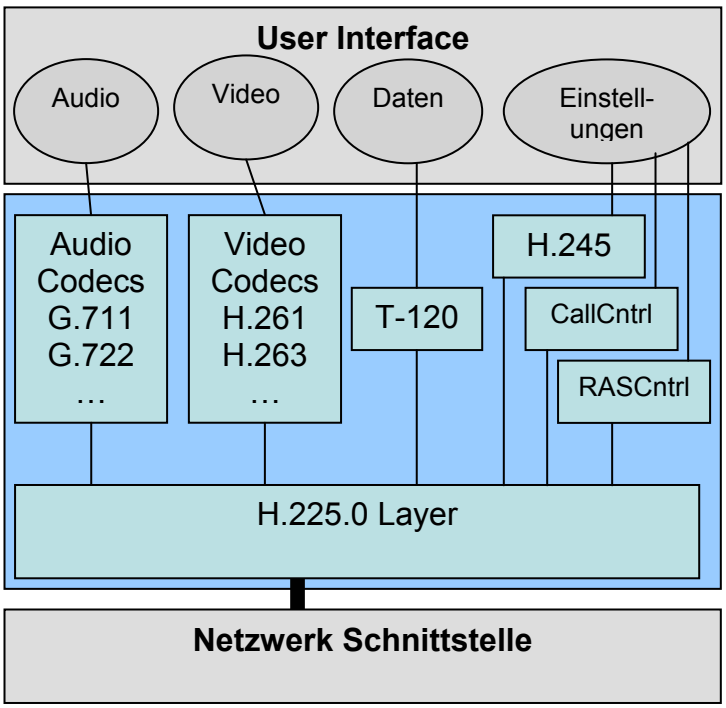
Für ein komplettes Videokonferenzsystem benötigt man nicht nur einen Standard um Videodaten zu übertragen. Man benötigt auch Möglichkeiten um die Videoverbindung zu steuern, Audiodaten zu übertragen, die Verbindung auszuhandeln und evtl. für andere Datendienste wie einen Chat, ein Whiteboard oder ähnliches.
Für diese Aufgaben gibt es H.323 und einige weitere Standards die von diesem verwendet werden.

Übersicht

H.323 Entstand 1996 als Nachfolger von H.320 welches für Videokonferenzen über ISDN gedacht ist. Die aktuelle Version die mir bekannt ist, ist die Version 4 von 2000. Es wurde ebenfalls von einer Forschungsgruppe bei der ITU entwickelt und findet heutzutage weite Verbreitung. Es erlaubt die Kopplung verschiedener Videokonferenzsysteme, und dies funktioniert sehr effektiv auch Plattformübergreifend.
H.323 muss von unterschiedlichen Komponenten beherrscht werden. Neben dem Client Programm gehören dazu die folgenden:

1. **Gatekeeper**
   Dieser stellt Adressübersetzungen und Bandbreitenkontrolle zur Verfügung.
2. **Multipoint Controller und Processor**
   Koordiniert mehrere Terminals, so dass diese gemeinsam an einer Konferenz teilnehmen können.
3. **Proxy**
   Dieser kanalisiert Verbindungen zwischen Terminals. Damit wird es möglich eine Konferenz zu führen mit zwei Terminals die sich jeweils hinter einer Firewall verbergen (unter der Vorraussetzung das wenigstens irgendeine Kommunikation gestattet ist).

Alle diese Programme müssen den H.323 Standard interpretieren können.



Komponenten

H.323 ist ein Standard der viele verschiedene andere miteinander kombiniert um ein System zu schaffen. An dieses System fügt man ein User Interface an, welches einem Benutzer erlaubt die Videokonferenz zu führen. H.323 befasst sich nicht damit

was dem Benutzer für Möglichkeiten gegeben werden. Man könnte also auch eine reine Audiokommunikation mit H.323 realisieren. Oder mithilfe des T-120 Standards ganz eigene Informationen schicken, und nur die Vermittlungs- und Transportfähigkeiten von H.323 nutzen.

Hier eine Übersicht über die Standards die H.323 verwendet:

- **H.225.0**
  verpackt Daten fürs LAN, initiiert und beendet Verbindungen mit Terminals, Proxys, etc. über multiple Kanäle.
- **H.245**
  Steuerung der Medienströme, Austausch von Kompatibilitäts-Informationen, Steuerung der Datenkanäle und andere Kommandos.
- **H.450**
  Aushandlung weiterer Kommunikationsmittel
- **H.235**
  Sicherheitsprotokolle zur Authentifizierung und Verschlüsselung.
- **H.332**
  Für die Koordination von größeren Konferenzen
- **G.711, G.722, G.723, G.728 und G.729**
  Audio Codecs mit Datenraten von 5.3 Kbps bis 64 Kbps
- **H.263 und H.261**
  Die Videostandards die oben besprochen wurden. Dabei kann man sich die speziellen Fähigkeiten dieser zunutze machen, wie Verwendung von H.263 Substreams (Anhang C), Und die vielfältigen Möglichkeiten zum anhalten und markieren von Bildbereichen.

Das bekannteste Tool was den H.323 Standard verwendet ist wohl Netmeeting, ein Videokonferenztool von Microsoft. Dieses kleine, simple Tool ist Bestandteil des Microsoft Betriebssystems Windows und Quasi-Standard für Videokonferenzen mit Standardhardware. Doch auch im Linuxbereich gibt es viele Anwendungen wie OpenH323 die kostenlos inklusive Sourcecode eine komplette Videokonferenzsuite enthalten. Auch setzen viele Hersteller von Kompletten Konferenzsystemen (Die Hardware und Software zusammen vertreiben) auf H.323. Das ermöglicht all diesen Systemen miteinander zu konferieren.

Zusammenfassung

In diesem Kapitel werden die Videostandards H.261 und H.263 besprochen. Diese Standards werden für Videokonferenzsysteme verwendet. Also ein Bereich bei dem es auf Echtzeitkodierung ankommt. Außerdem darf nur eine geringe Verzögerung vorhanden sein. Diese Standards beschreiben nur das Format der Daten, Außerdem sind die Formate recht flexibel, so dass auch ein Enkoder von einer Firma mit dem Dekoder einer anderen zusammenarbeitet.
Im Anschluss an die Beschreibung der Standards wird noch eine Übersicht über die Implementierung eines Enkoders gegeben. Dies ist jedoch keine konkrete Implementierung, wer eine solche machen will kommt nicht umher die Dokumente der ITU zu konsultieren.
Am Ende wird noch kurz auf den H.323 Standard eingegangen der für die Abhandlung von Konferenzen verwendet wird, und der die Videostandards verwendet um Videodaten zu übertragen.

Abkürzungen

ITU	International Telecomunication Union.
ITU-T	Telecomunication Standardization Sector of ITU
GOB	Group of Blocks
VLC	Variable Length Code (Ein Kode variabler länge)

Literatur

[CT6/03] Effizienter kodieren, Karsten Sühring, Dr. Heiko Schwarz, Dr. Thomas Wiegand, c't 6/2003 Seite 266, 10.03.2003; Heise Zeitschriften Verlag GmbH & Co. KG.

[IFCOD] W. Heise und P. Quattrocchi. Informations- und Codierungstheorie. Springer-Verlag, 1989.

[ITU261] Recommendation H.261,Video Codec for Audiovisual Services at $p \times 64$ kbit/s, März 1993; ITU-T.

[ITU263] Draft H.263, Video coding for low bit rate communication, 27 Januar 1998; ITU-T.

[ITU323] Draft H.323v4 (Including Editorial Corrections - February 2001), Study Group 16, Q.13-14/16 Rapporteur Meeting Geneva, 9 – 10 November 2000; ITU-T.

MPEG 1|2 Video

von

Klaus Brügmann

Zielsetzungen der Motion Picture Experts Group

Als sich 1988 die Moving Pictures Experts Group (MPEG) als eine Arbeitsgruppe innerhalb der ISO und dem IEC formierte, war die Entwicklung der Computerhardware so weit fortgeschritten, dass der multimediale Einsatz von Rechnern auch im Heimbereich realisierbar wurde. Eine zentrale Frage war und ist bis heute die der Komprimierung der speicherintensiven Videodaten. Die Einbindung von Videos in Computeranwendungen, aber auch einfach die digitale Archivierung von Filmen auf dem sich damals schnell verbreitenden Medium CD machten es zu dem erklärten Ziel der MPEG, einen Standard für die Repräsentation von Videodaten in komprimierter Form zu entwickeln, der die Speicherung von Bilddaten in VHS-Videorekorder-Qualität bei Bitraten von bis zu 1.5 Mb/s unterstützte. Der fertige Standard mit dem Spitznamen MPEG-1 wurde schließlich 1993 als ISO/IEC 11172 veröffentlicht. 1990 schon erkannte man bei der MPEG den Bedarf nach einer Erweiterung des Standards, der Fernsehsendern die Kodierung von Videoaufzeichnungen in Sendequalität bei Bitraten zwischen 4 und 6 Mb/s ermöglichte sowie Unterstützung weiterer Funktionen bot. Ein zweiter Standard wurde in Angriff genommen und 1994 als ISO/IEC 13818 oder auch MPEG-2-Standard verabschiedet. Eingesetzt wurde diese Erweiterung in einer großen Vielfalt von Anwendungen, später auch bei der Einführung von hochauflösendem Fernsehen (High Definition TV), für welches der Standard sich ebenfalls als geeignet erwies.

Hauptteil

Visuelle Wahrnehmung und analoge Medien

Etwa um eine Geschichte zu erzählen, wird in der Verwendung von Videobildern der Versuch unternommen, dem Menschen - und speziell seinen Sinnesorganen Auge und Ohr - eine visuelle Umgebung vorzutäuschen. Mit den heutigen technischen Möglichkeiten ist eine solche Vortäuschung in der Qualität, in der wir die Wirklichkeit erleben, nicht möglich. Dennoch genügt dem Zuschauer in der Regel der Fernseher-große Ausschnitt eines visuellen Erlebnisses, um sich in das betrachtete hineinzuversetzen. Der Zuschauende bringt dabei den Willen zur Adaption der Bilder auf. Damit das Seherlebnis darüber hinaus auch angenehm ist, muss das Videobild aber eine gewisse Qualität besitzen:

Das Auge ist in der „schillernden“ Wirklichkeit einem kontinuierlichen Strom von Wellen ausgesetzt. Im Sehnerv werden diese „Einflüsse“ in Form von Impulsen als Momentaufnahmen an das Sehzentrum weitergeleitet. Kontinuierliche Bewegungen der gesehenen Dinge kommen also als Einzelbilder im Gehirn an. Das Gehirn, bestrebt, Bewegung als solche wahrzunehmen, rekonstruiert diese aus den ihm zur Verfügung stehenden Einzelbildern. Videobilder dürfen sich auch auf diese Fähigkeit des Gehirns verlassen, dürfen dabei aber gewisse Toleranzgrenzen nicht unterschreiten. So gilt eine Bildfrequenz von 16Hz als Grenze, unterhalb der eine Bewegung als ruckartig empfunden wird.

Auch was Farb- und Helligkeitsübergänge angeht muss das Videobild eine ausreichende Qualität haben, um nicht als unscharf oder farb-unecht zu wirken. Diese Eigenschaften

drücken sich in der Bild-Auflösung und in der Genauigkeit der Farbwiedergabe oder auch Farbtiefe aus.

Abbildung 1: Unzureichende Bild-Auflösung

Das analoge Fernsehen erreicht seine Qualität durch 25-30 Einzelbilder pro Sekunde, einer Auflösung von 625 Zeilen (PAL), auf denen ein Elektronenstrahl den kontinuierlichen Helligkeits- und Farbverlauf der Aufnahme nachzeichnet. Um den Flimmereffekt zu vermeiden, der sich bei dieser Frequenz aus der zeitlich versetzten Bestrahlung der Bildschirmbereiche ergibt, setzt die klassische Fernsehtechnik Interlacing ein. Dabei wird ein Bild in zwei Halbbilder zerlegt gesendet, wobei sich jedes der beiden über die gesamte Bildschirmfläche erstreckt, jedoch jeweils nur jede zweite (bzw jede erste) der Zeilen beinhaltet.

Wozu digitales Video?

Die analoge Übertragung und Speicherung von Videobildern ist jedoch mit gewissen Nachteilen behaftet. So können Verfälschungen eines Signals, die sich etwa während der Übertragung ereignen, nicht erkannt, geschweige denn rückgängig gemacht werden. Die Editierung von solchen Signalen gestaltet sich hardwaretechnisch als aufwendig und bringt zudem weitere Qualitätsverluste mit sich. Die technische Repräsentation von visuellen Aufzeichnungen in digitaler Form jedoch verhindert den Qualitätsverlust bei allen Operationen weitgehend. Paritätsbits gleichen den Abbau der Signalqualität durch Fehlererkennung aus. Systemintern können digitale Videodaten Verlustfrei dupliziert und editiert werden. Der grundlegende Unterschied zu analoger Repräsentation liegt im Arbeiten mit diskreten Werten. Mit begrenzter Anzahl verwendbarer Bits für einen Pixel sind natürlich auch nur begrenzt viele Farben kodierbar. Eine Hauptanforderung an digitale Videodaten ist deswegen die Bereitstellung eines ausreichenden Spektrums an Farben und einer hohen Dichte von Pixeln, so dass im Auge des Betrachters der Eindruck eines Kontinuums (der Farben und Flächen) entsteht, wie es die reale Welt und bedingt auch das analoge Video bieten.

Die MPEG-Kodierung

Überblick

Die beiden hier besprochenen Standards zur Kodierung von digitalem Video umfassen neben der Beschreibung von algorithmischen Werkzeugen zur Komprimierung von Bilddaten auch solche zur Kodierung von Audio- und Benutzerdaten und zur Integration von mehreren solcher Ströme zum komplett kodierten, audiovisuellen Film. Diese Arbeit befasst sich ausschließlich mit den zur Bildverarbeitung bereitgestellten Werkzeugen.

Der MPEG-Standard schreibt dabei lediglich vor, wie ein kodiertes Signal auszusehen hat. Damit ist der Dekodiervorgang eindeutig festgelegt. Um jedoch einen besseren Zugang zu den einzelnen Kompressionsschritten zu finden, soll hier der Verschlüsselungsvorgang betrachtet werden.

Zunächst einmal können MPEG-Daten nach wie vor als eine Folge von Einzelbildern, den Frames, begriffen werden. Obwohl diese nicht unabhängig voneinander betrachtet werden können, steht grundsätzlich ein Frame für ein aufgezeichnetes Bild, die Frame-Wiedergaberate entspricht also der Frequenz des resultierenden Videobildes.

Die eine grundlegende Vorgehensweise, die Bilddaten zu komprimieren, besteht in der Verringerung von örtlicher Redundanz, die in Sampling-Werten dadurch enthalten ist, dass sich die Färbungen benachbarter Pixel in der Regel nur sehr wenig unterscheiden.

Die zweite große Einsparung an Bandbreite, die MPEG-Datenströme gegenüber einer Serie von „JPEG-Frames“ macht, rührt aus der zeitlichen Redundanz, der großen Ähnlichkeit, die aufeinanderfolgende Bilder eines Videos in der Regel haben. Ein MPEG-Kodierprogramm findet ähnliche Bereiche in verschiedenen Frames und setzt statt expliziter Neukodierung der aktuellen Sampling-Werte einfach Referenzen zu anderen Frames in den Code ein. Er lässt Teile eines Frames sozusagen durch andere Frames „voraussagen“. Eine solche Voraussage muß nicht fehlerfrei sein. Der Unterschied zwischen Vorlage und Kopie kann zusammen mit der Referenz kodiert werden, ohne dass dabei die Datenmenge der expliziten Kodierung erreicht würde.

Örtliche Redundanz

Um örtliche Redundanz der Bildinformationen zu verringern, verwendet der Codec die JPEG-Kodierung. Das als zweidimensionales Array von Farbwerten vorliegende, digitalisierte Bild wird zu Beginn mittels entsprechender Formeln in einen angepassten Farbraum übertragen, den YCrCb– oder auch YUV- Farbraum, in dem eine Farbe durch einen Helligkeits- und zwei Chrominanzwerte repräsentiert ist. Dies bietet eine erste Kompressionsmöglichkeit, nämlich das Subsampling der beiden Farbwerte Cr und Cb. Jeweils nur der Mittelwert vierer Pixel wird dabei übernommen. Dieser Schritt trägt der Tatsache Rechnung, dass das menschliche Auge für Helligkeitsunterschiede sensitiver ist als für Farbwechsel. Nun werden diese drei Bitmaps in eine Rasterstruktur von Blöcken mit je 8x8 Pixeln eingeteilt. Ein Block von Farbwerten ist demnach vier Blöcken von Helligkeitswerten zugeordnet. Jeder dieser Blöcke wird hierauf durch Diskrete Kosinus-Transformation (DCT) in den Frequenzraum übertragen.

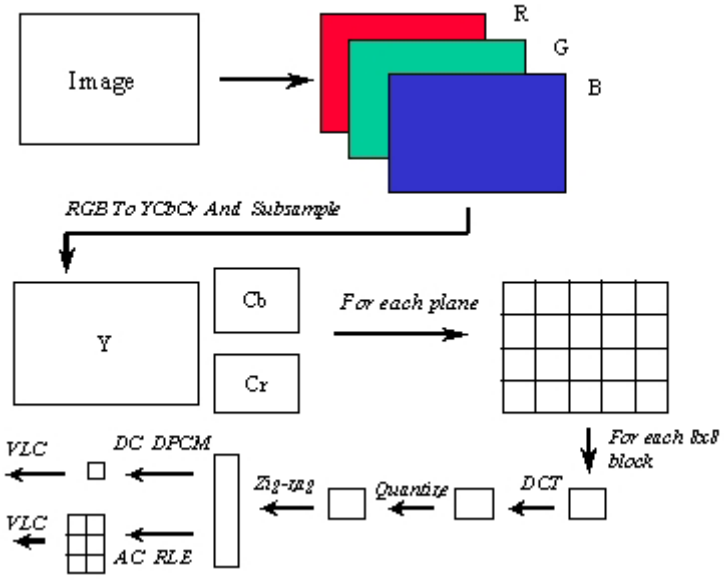


Abbildung 2: Komprimierungsschritte (intra-kodiert, JPEG-ähnlich)

Die aus der DCT hervorgehende Koeffizientenmatrix jedes Blockes wird einer gezielten Quantisierung unterzogen, welche die Genauigkeit, mit der die Frequenzwerte kodiert werden, in Abhängigkeit von deren Relevanz für das sichtbare Bild einstellt. Hier findet nun eine Trennung der DC-Koeffizienten von den AC-Werten statt. Während die DC-Koeffizienten nach einer differenziellen PCM-Kodierung abgespeichert werden, müssen die Werte der AC-Koeffizienten zunächst serialisiert werden.

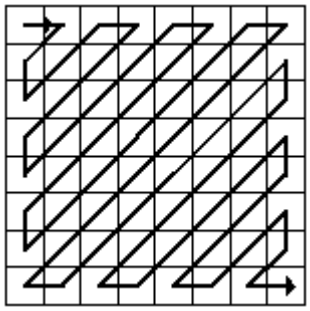


Abbildung 3: ZickZack-Muster

Das hierzu verwendete ZickZack-Muster begünstigt eine Häufung der Null-Koeffizienten zum Ende der Serie hin, so dass diese durch einen spezifizierten „EndOfBlock"-Code ersetzt werden können, da die Anzahl der Koeffizienten ja bekannt ist. Die Serie von Koeffizienten vor dieser Markierung wird zerlegt und in Tokens umgewandelt, die nach einer im Standard spezifizierten Huffman-Tabelle kodiert werden.

Block-Zerlegung der Bilder

Die zu kodierenden Videobilder werden also in Blöcke von 8x8 Pixeln eingeteilt, wobei zu vier Blöcken der Helligkeits-Map jeweils ein Block jeder der beiden Farb-Maps gehört. Ein solches 16x16 Pixel großes Quadrat des ursprünglichen Bildes wird **Makroblock** genannt. Jeder dieser Makroblöcke eines Frames kann nun auf verschiedene Arten kodiert werden:

Zum einen kann ein Kodierer die Bildinformation direkt angeben, indem er die Sample-Werte gemäß dem oben beschriebenen Schema der JPEG-Codierung verschlüsselt. Nach Anwendung der DCT liegen diese als Koeffizienten der Frequenzen, die die Farbübergänge des betreffenden Blockes repräsentieren, vor. Diese Blöcke nennt man **intra-kodiert** (intra-coded), da sie die vollständige Information der repräsentierten Samples in ihrem Code enthalten.

Die zweite Möglichkeit besteht darin, für einen Block lediglich einen Verweis auf einen anderen Block eines anderen Frames zu setzen, dessen Aussehen mit dem aktuellen weitgehend übereinstimmt. Diese Referenz besteht aus einer Angabe eines Frames sowie aus einem Vektor, der die Verschiebung des ähnlichen Bereiches zum aktuellen beschreibt, einem Bewegungsvektor. Man lässt den aktuellen Makroblock also durch einen anderen Frame voraussagen, wonach diese Vorgehensweise auch **Bewegungs-kompensierte Voraussage** (engl. **Motion-compensated Prediction, MCP**) genannt wird.

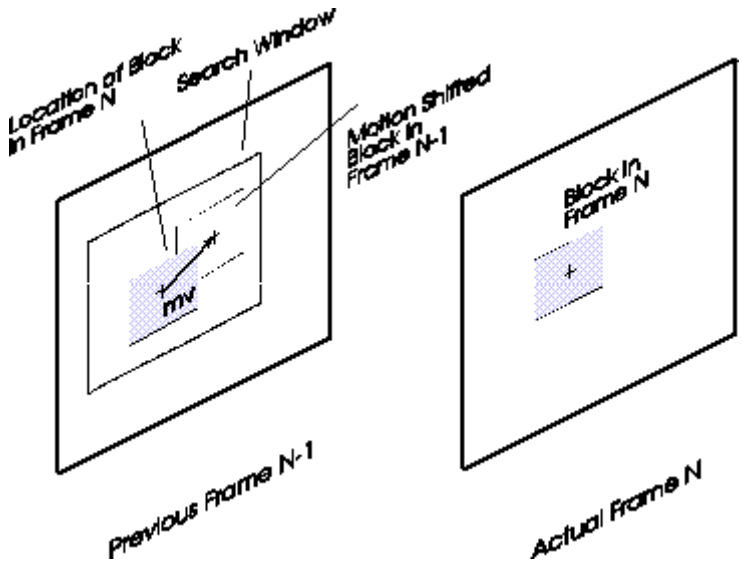


Abbildung 4: Voraussage und Bewegungsvektor

Eine solche Referenz ist natürlich weitaus kürzer zu kodieren als die tatsächliche Bildinformation, woraus sich die hohen Kompressionsraten eines MPEG-Videos gegenüber etwa sequentiellen JPEG-Bildern ergeben. Die auf diese Weise im Bitstrom beschriebenen Blöcke nennt man **inter-kodiert** (inter-coded), da sie eine Beziehung zwischen Frames herstellen, welche dann später auch nicht mehr einzeln dekodiert werden können. Die Referenzierungen können dabei zu Frames vor dem aktuellen weisen, also in die „Vergangenheit“, aber auch in die „Zukunft“ zu Frames, die eigentlich noch gar nicht angezeigt wurden, die aber trotzdem schon dekodiert im Speicher vorliegen müssen, um als Vorlage oder auch Vorhersage (Prediction) zu dienen. Natürlich ist es oftmals unmöglich, eine genaue Übereinstimmung zu finden, weshalb der Standard vorsieht, den referenzierten Bildbereich nicht nur zu nennen, sondern auch den Fehler der Voraussage, also der Unterschied zwischen einer Vorlage und dem aktuellem Block zu kodieren. Dazu werden die Differenzen der einzelnen Sampling-Werte berechnet und darauf nach gleichem Schema wie auch bei Inter-Kodierung komprimiert. Die Vektoren beschreiben eine Pixelweise Bewegung. Referenzierbar sind demnach Bereiche gleicher Größe (16x16 Pixel), die nicht unbedingt im Blockraster, welches für ein gegebenes Bildformat festgelegt ist, liegen müssen. Die Referenzierung für den aktuellen Block kann auch in beide Zeitrichtungen gleichzeitig

erfolgen. Aus den beiden Bildausschnitten, einem zukünftigen und einem vergangenen wird dann der Mittelwert berechnet, dessen Vorhersage-Fehler wiederum in den Code einfließt.

Die Suche nach geeigneten Bildteilen, die zu diesem Zweck im Zuge des Kodierens stattfindet, stellt eine Herausforderung an Programmierer dar, da sie die Kompressionsrate wie auch die Kodierungsgeschwindigkeit entscheidend beeinflusst. Darauf wird unten noch genauer eingegangen.

Die dritte „Kodierungsweise“ für einen Makroblock ergibt durch das Auslassen (**skipped** Macroblocks). Dies bietet sich vor allem bei Videosequenzen an, in denen Teile des Bildes über mehrere Frames hinweg unverändert bleiben. Ausgelassene Blöcke eines Frames werden einfach mit der Bildinformation des vorangegangenen Frames gefüllt. Szenen, die mit fester Kamera gedreht werden, aber auch Zeichentrickfilme enthalten gewöhnlich viele gleichbleibende Blöcke, letztere wegen der vielen einheitlich gefärbten Flächen.

Frames

Die einzelnen Frames eines MPEG-Videos bestehen also aus kodierten Makroblöcken. Welche Art der Kodierung in einem Frame zum Einsatz kommt, beeinflusst allerdings die Möglichkeiten der Dekodierung. Das MPEG-Format wurde entwickelt, um Data-Streaming zu unterstützen, weshalb es für einen Dekodierer möglich sein soll, an beliebiger Stelle in einen MPEG-Strom einzusteigen. Wird in einer Frame-Sequenz jedoch die Kenntnis des Sequenzbeginns vorausgesetzt, um einen Frame zu entschlüsseln, so ist dies nicht möglich. Unter anderem deswegen werden im Standard verschiedene Frame-Typen definiert, sowie Aussagen über erlaubte Referenzierung in der MCP gemacht.

I-Frames (**i**ntra-kodierte Frames) unterliegen der Beschränkung, dass sie nur intra-kodierte Blöcke enthalten dürfen. Diese bieten einem Dekoder sichere Einstiegspunkte in den Bitstream, da sie die komplette Bildinformation selbst enthalten. Auch dürfen nachfolgende Frames keine „älteren“ als den letzten I-Frame referenzieren, so dass der Dekodierer also alle nachfolgenden Frames entschlüsseln kann.

P-Frames (**p**rädiktiv kodierte Frames) enthalten neben intra-kodierten Blöcken auch solche inter-kodierten, deren MCP-Vektoren in die Vergangenheit, und zwar zum letzten vorhergehenden I- oder P-Frame weisen. Um einen P-Frame zu entschlüsseln ist also nur die Pufferung eines weiteren dekodierten Frames nötig.

B-Frames (**b**idirektional prädiktiv kodierte Frames) können nun jede Art von Blöcken enthalten. MCP-Referenzierungen weisen dann auf den letzten vorhergehenden und/oder auf den nächsten folgenden I- oder P-Frame. Um Komplikationen zu vermeiden und um die Zahl der zu puffernden Frames bei zweien zu belassen dürfen B-Frames selbst nicht referenziert werden.

Aufgrund dieser Bedingungen wird beim Kodieren einer Sequenz meist eine bestimmte Frame-Abfolge eingehalten und nur missachtet, wenn das Videomaterial - etwa bei Schnitten - dies notwendig macht. Die Auswahl eines bestimmten Musters kann bei vielen Kodierprogrammen vom Benutzer getroffen werden. Typische Muster enthalten regelmäßig I-Frames, zwischen denen eine konstante Anzahl P-Frames liegt, zwischen denen wiederum eine geringe Zahl B-Frames liegen (z.B. IBBBPBBBPBBBPBBB, IBBPBB oder IBBPBBPBBPBB).

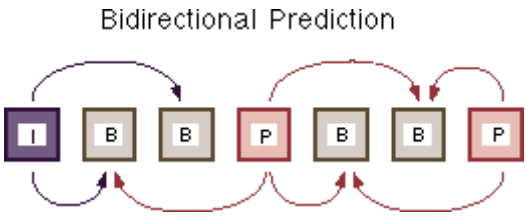


Abbildung 5: Mögliche MCP-Referenzierung

Constrained Parameters Bitstream

Um die Hardware-technischen Anforderungen des Dekodiervorganges eines MPEG-Stromes zu kontrollieren, wird innerhalb des Standards unter dem Namen „Constrained Parameters Bitstream“(CPB) ein Satz von Beschränkungen der Parameter eines kodierten Videos definiert. Entwickler von MPEG-unterstützenden Anwendungen nahmen diese als Orientierungshilfe für die erforderliche Leistungsfähigkeit ihrer Produkte. Die CPB beschränken die kodierten Makroblöcke eines Frames auf 396 bei einer Bildrate bis zu 25 Hz und 330 bei bis zu 30 Hz. Daraus resultierte eine weite Verbreitung von MPEG-Video im SIF-Format mit 352x240 Pixeln bei 30 Hz beziehungsweise 352x288 Pixel bei 25 Hz. Die Begrenzung gilt dabei nur als obere Schranke.

Standard-Erweiterungen durch MPEG2

Interlacing

Eine grundlegende Erweiterung durch den MPEG2-Standard ist die Möglichkeit Halbbilder zu kodieren. Dies ist eine zwingende Notwendigkeit für die Verwendung des MPEG-Videoformates zur Aufzeichnung und Ausstrahlung von Fernsehbildern. Die Bilder werden dabei in zwei Halbbilder zerlegt gesendet um von den empfangenden Geräten auch als solche auf den Schirm gezeichnet zu werden. Ob der hohen Bildfrequenz fällt das dem Betrachter nicht auf, ergibt aber eine gleichmäßigere Ausleuchtung und damit eine Reduktion des Flimmereffektes bei Röhrengeräten.

Level

Die in MPEG1 als Constrained Parameters Bitstream (CPB) bekannte Definierung eines gebräuchlichen Bildformates wird bei MPEG2 auf insgesamt vier sogenannte Level ausgeweitet (Low, Main, High-1440, High). Die einzelnen Level beschreiben jeweils Parameterbeschränkungen der Bildauflösung, der Bildfrequenz, der Bitrate des Datenstroms sowie der notwendigen Puffergröße des verwendeten Dekoders. Die Beschränkungen geben dabei Maximalwerte an, ein High-Level Dekodierer wird also beispielsweise auch Main-Level Videodaten anzeigen können.

Level	Max. frame, width, pixels	Max. frame, height, lines	Max. frame, rate, Hz	Max. bit rate, Mbit/s	Buffer size, bits
Low	352	288	30	4	475136
Main	720	576	30	15	1835008
High-1440	1440	1152	60	60	7340032
High	1920	1152	60	80	9781248

Abbildung 6: Tabelle: Level (MPEG2)

Der Low-Level entspricht qualitativ den CPB des MPEG1-Standards. Standard-Fernsehsignale erfordern die Auflösung des Main-Level, während hochauflösendes Fernsehen High-1440-Level verwendet.

Profile

MPEG2 bietet die Option, Videos in mehreren Qualitäts-Stufen gleichzeitig zu übertragen. Man spricht von der Skalierbarkeit der Videodaten. Das auf mehrere Kanäle verteilte Signal liegt dabei zum einen auf einer Basis-Ebene als ein unabhängig dekodierbarer Datenstrom vor. Besitzt ein Dekodierer über die nötige Hardware, so kann er zusätzlich Datenströme einer (oder mehrerer) Erweiterungs-Ebenen desselben Videos in den Entschlüsselungsprozess mit einbeziehen und so die Darstellungsqualität verbessern. Alle Möglichkeiten der Skalierbarkeit simultan einzusetzen, wäre allerdings wenig sinnvoll, weswegen MPEG, um die Art der verwendeten Skaliermethodik zu kontrollieren mehrere Profile definiert:

Das **Main-** und das **Simple-Profil** sind beide nicht skalierbar, wobei das Simple-Profil noch der Einschränkung unterliegt, keine B-Frames zu verwenden, was an die Komplexität des Dekodierers geringere Ansprüche stellt (z.B. bezüglich der Puffergröße). Diese Profile entsprechen im wesentlichen einer Kodierung nach dem MPEG1-Standard (bis auf die Verwendung von Interlacing).

Als skalierbare Profile gelten das SNR-, das Spacial- und das High-Profil:

Das **Signal-to-Noise-Ratio(SNR)-Profil** bietet dabei auf der höheren Ebene Informationen, die den Qualitätsverlust der Komprimierung verringern. Der Enhancement-Layer-Bitstream enthält die bei der Quantisierung der DCT-Koeffizienten gemachten Rundungsfehler in kodierter Form. Dadurch kann das entschlüsselte Base-Layer-Datenmaterial noch einmal im Hinblick auf die Genauigkeit der DCT-Koeffizienten aufgewertet werden. Bewegungsvektoren der MCP werden dabei nur im Basis-Ebenen-Signal übertragen.

Im **Spatial-Profil** kodierte Übertragungen stellen auf der Erweiterungs-Ebene eine höhere Auflösung des Bildmaterials zur Verfügung. Alternativ können für die Vorhersage von Makroblöcken des Enhancement-Layer-Datenstromes die Frames des Base-Layer-Datenstromes in skalierter Form verwendet werden.

Das **High-Profil** schließlich vereint die Werkzeuge des SNR- und des Spatial-Profils.

Eine weitere Form der Skalierbarkeit, die nicht als Profil definiert ist, ist die **temporale Skalierbarkeit**. Dabei werden auf der Erweiterungs-Ebene weitere Frames der gleichen Auflösung wie der des Base-Layers übertragen, durch welche sich die Bildfrequenz des resultierenden Videos erhöhen lässt.

Die MPEG-Syntax

MPEG1-Syntax

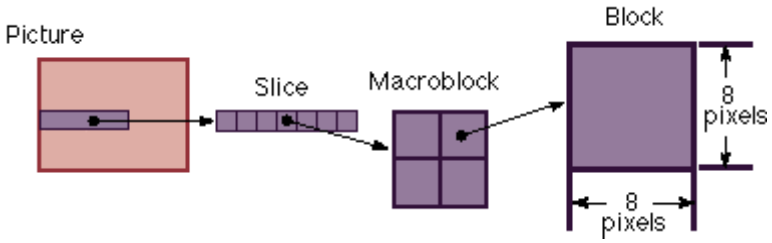


Abbildung 7: Zerlegung eines Bildes

Die höchste syntaktische Struktur eines MPEG-Video-Datenstromes ist die Videosequenz. Sie ist nicht identisch mit einem kompletten Film oder einer Sendung, sondern beschreibt einen Teilausschnitt. Der Standard sieht vor, einen Videodaten-Strom in Sequenzen einzuteilen, da deren Header alle erforderlichen Daten über das verwendete Bildformat, die Bildfrequenz, die angestrebte Bitrate, Level, Profil usw enthalten. Ein Dekodierer, der an beliebiger Stelle des Stromes zu lesen beginnt, erhält so alle nötigen Informationen. Die im Sequenz-Header gegebenen Informationen sollen nicht variieren, sondern lediglich eine Wiederholung der zu Beginn der Übertragung gemachten Angaben darstellen. Eine Ausnahme bilden dabei die Quantifizierungs-Matrizen, welche natürlich frei spezifizierbar bleiben.

Der Sequenz-Header wird von den Datenblöcken der Bilder gefolgt. Dabei entspricht die Reihenfolge der Frames im Bitstrom nicht unbedingt der tatsächlichen Darstellungsreihenfolge. Kodiert wird viel mehr die Reihenfolge, in welcher die Frames zur Dekodierung benötigt werden. So verringert sich die benötigte Puffergröße, wenn etwa zur Dekodierung eines B-Frames ein zukünftiger P-Frame benötigt wird.
Die Bild-Header enthalten u.a. Information über die Art der Framekodierung (z.B. I-Frame, Halbbild).

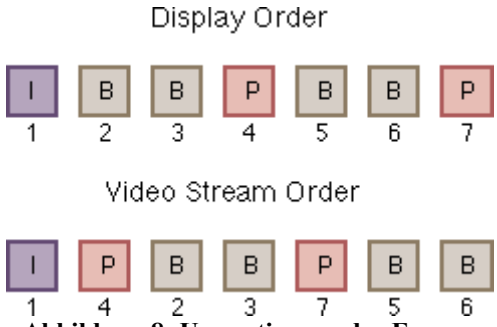


Abbildung 8: Umsortierung der Frames

Zwischen Bild und Makroblock existiert eine weitere Ebene in der Hierarchie: Die **Slices** beschreiben Horizontale zusammenhängende Reihen von Makroblocks derselben Zeile des Bildes. Im Header der Slice-Datenstruktur ist die vertikale Positionierung kodiert, welche zur genauen Lokalisierung der enthaltenen Makroblöcke mit verwendet wird. Mit Hilfe der Slices ist es möglich, Blöcke eines Frames unkodiert zu lassen, ohne dies noch explizit im Code erwähnen zu müssen. Eine Slice-Struktur enthält dann kodierte Makroblock-Datenstrukturen. Diese wiederum haben Header-Angaben über horizontale Position, über die Bewegungsvektoren bei Interkodierung und weiteres über die Art der Kodierung. Letztlich enthalten die Makroblock-Datenstrukturen die einzelnen VLC-Codes der Quantisierten DCT-Koeffizienten, und damit die eigentliche Sample-Information, also die Pixelfarbe.

Die Strukturen des MPEG-Datenstromes beginnen jeweils mit einem eindeutigen 32 Bit langen **Startcode**, dessen erste 23 Stellen Nullen enthalten. Um die Emulation einer solchen Kombination zu verhindern werden in den Code Markerbits eingesetzt.

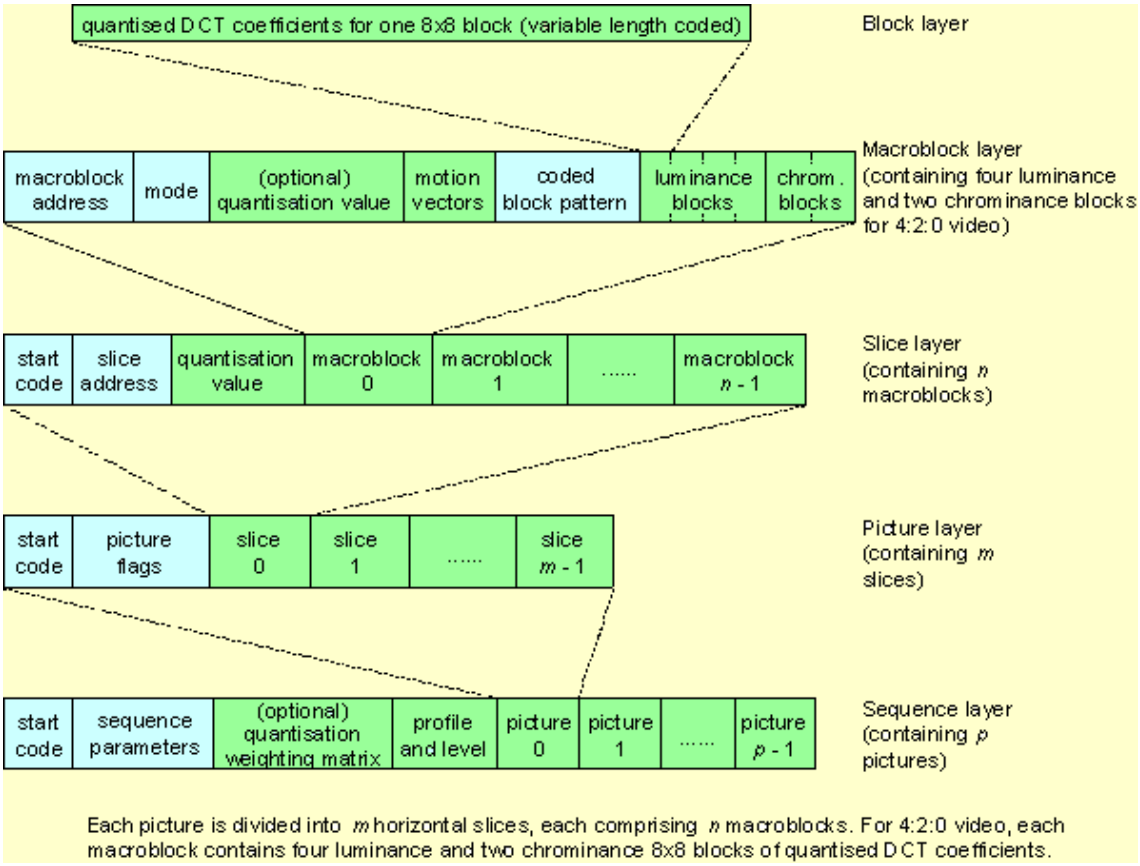


Abbildung 9: Hierarchischer Aufbau einer Video-Sequenz

Syntax-Erweiterung durch MPEG2

Die MPEG2-Syntax lässt den MPEG1-typischen Sequenz-Headern sogenannte Extensions, Erweiterungen folgen, welche die Verwendung von MPEG2-Funktionen wie etwa skalierbaren Profilen regeln. Verhält sich der Basis-Ebenen-Datenstrom einer MPEG2-Übertragung an die MPEG1-Spezifikation, könnte also von einem entsprechenden Dekodierer angezeigt werden, so kann im Sequenz-Header ein entsprechendes Flag gesetzt werden.

Grundsätzlich wurde auf eine **Abwärtskompatibilität** zu MPEG1-Code geachtet, so dass dieser auch von MPEG2-Wiedergabeprogrammen angezeigt werden kann.

Bitrate: konstant oder variabel?

Bei der Kodierung eines digitalen Videos können bezüglich der Kompression verschiedene Ziele gesetzt sein. So kann zum einen die gleichbleibende Qualität des Bildes an erster Stelle stehen. Da sich das dargestellte aber fortwährend verändert und die Komprimierungsmöglichkeiten von Frame zu Frame oder auch von Szene zu Szene stark differieren können, besteht die Gefahr einer höchst variablen Bitrate des kodierten Video-Datenstromes. Bei Videos, die auf lokalem Datenträger vorliegen, ist das eine übliche Vorgehensweise, weil die mögliche Durchsatzrate systemintern in der Regel weit über den im

Datenstrom auftretenden Spitzenwerten der Bitrate liegen. Wird ein Video allerdings über einen Kanal mit begrenzter Bitrate gesendet (Video-Streaming über Telefonleitung, digitales Fernsehen), so ist eine eher konstante Bitrate erforderlich. Um dies zu gewährleisten bietet der MPEG-Standard die Möglichkeit, im Code Skalierungsfaktoren für die Einträge der Quantisierungsmatrizen anzugeben. Somit kann ein Kodierer, der einen Datenstrom mit konstanter Bitrate erzeugen soll, die Zahl der für die Kodierung eines Frames aufgewendeten Bits steuern und einen vorbestimmten Durchschnittswert anstreben. Konsequenz daraus sind sichtbare Qualitätsverluste, die sich in Szenen einstellen, die zum Beispiel aufgrund hoher Aktivität zu geringe Komprimierungsmöglichkeiten bieten.

MCP-Algorithmen

Die intelligente Suche nach guten Vorhersagen für Makroblöcke im Videomaterial ist eine zentrale Aufgabe für Entwickler von MPEG-Kodierprogrammen. Sie nimmt großen Einfluss auf die Kodiergeschwindigkeit bzw die erreichte Kompression. Besonders für Live-Videoübertragungen, die in Echtzeit kodiert werden müssen, sind leistungsfähige Such-Algorithmen unabdingbar. Einige Ansätze sollen hier vorgestellt werden.

Die optimale Kodierung für einen gegebenen Makroblock findet man stets, indem man alle möglichen (referenzierbaren) Bereiche betrachtet, jeweils den Prädiktions-Fehler berechnet (und komprimiert) und von allen Möglichkeiten diejenige auswählt, welche die beste Kompression liefert. Die Intra-Kodierung eines Blockes verwendet man dann, wenn man so eine noch bessere Kompression erreicht, wenn also keine brauchbare Voraussage gefunden werden konnte.

Ist die Kodierzeit unbegrenzt, so stellt dieses Vorgehen sicher das Optimum dar. Ist sie es nicht, so muss ein Kompromiss zwischen kurzer Suchdauer und hoher Kompression gesucht werden. Man will also in möglichst kurzer Zeit den möglichst besten Kandidaten für die MCP finden und führt dazu eine schnelle **Vorauswahl** unter allen Möglichkeiten durch. Ungeachtet der getroffenen Vorauswahl lässt sich zudem eine **Toleranz für den erzielten Fehler** festlegen. Die Suche wird dann abgebrochen, sobald sich ein Kandidat mit einem Prädiktionsfehler unterhalb dieser Grenze findet. Welche Möglichkeiten der Vorauswahl bieten sich nun:

Möglichkeiten der Vorauswahl

Fenstersuche

Man beschränkt sich auf das **Abtasten eines Fensters** um die Position des betreffenden Makroblockes herum, in welchem sich mit hoher Wahrscheinlichkeit die besten Treffer finden. Liegen außerhalb noch bessere Treffer, so findet man sie nicht, hat aber wenigstens nicht lange gesucht. Bei Verwendung einer Toleranzgrenze bietet sich dabei ein Spiralsuchmuster an, welches die vermutlich besten Kandidaten im Fenster zuerst betrachtet.

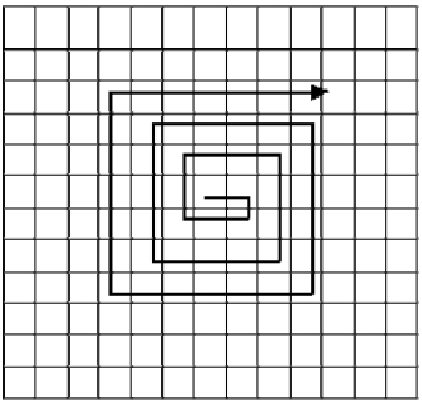


Abbildung 10: Spiralsuchmuster

Eine bessere Platzierung des Suchfensters könnte man außerdem erreichen, wenn man am Bewegungsvektor desselben Blockes des vorhergehenden Frames eine erwartete Bewegung des aktuellen Blockes berechnet, und das Suchfenster um diesen Vektor verschiebt.

Mittelwertbildung

Hierbei Berechnet man zunächst den Mittelwert der Sample-Werte für jeden Kandidaten einzeln. Nun berechnet man von jedem die Differenz zum Mittelwert des aktuellen Makroblocks. Diese Differenz kann wesentlich schneller berechnet werden und liegt rechnerisch noch unterhalb des tatsächlichen Fehlers. So können viele Kandidaten schnell verworfen werden, bevor eine zeitaufwendigere Berechnung des tatsächlichen Fehlers stattfindet. Die zeitaufwendige Mittelwertberechnung aller Bildbereiche eines referenzierten Frames relativiert sich dadurch, dass sie nur einmal durchgeführt werden muss und die Werte für alle in Kodierung befindlichen Makroblöcke des aktuellen Frames verwendet werden können.

Subsampling

Bei diesem Ansatz betrachtet man zuerst ein grobes Raster von Kandidaten aus dem gewählten Suchbereich und konzentriert sich beim weiteren Suchen auf die direkte Umgebung der besten Stichproben.

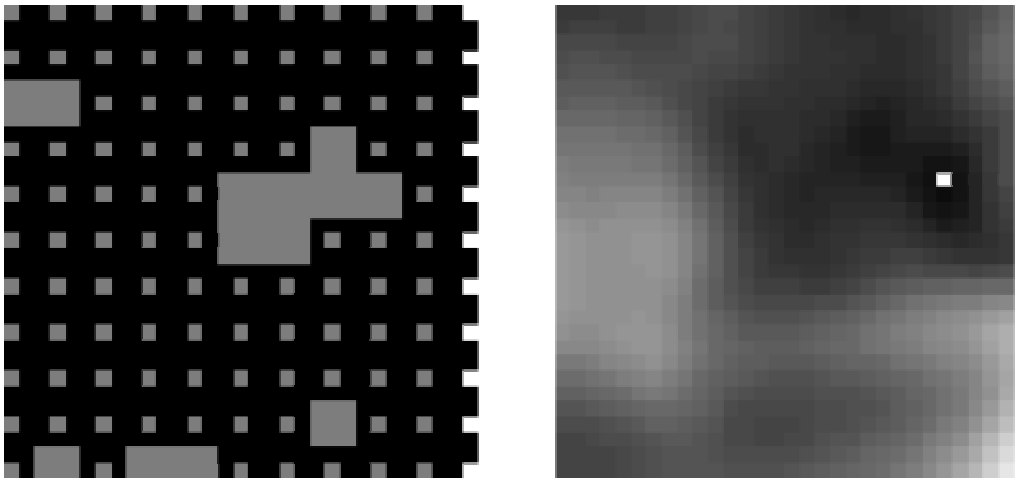


Abbildung 11: Suche nach grobem Raster, bester Treffer

Fourier-Abschätzung

Statt der Mittelwertbildung führt auch eine Fouriertransformation nach den niedrigsten Frequenzen der verglichenen Quadrate zu einer aussagekräftigen Abschätzung des Fehlers.

Fazit

Die Standards MPEG1 und MPEG2 beschreiben ein für digitales Video blockbasiertes Kodierverfahren, welches zwar verlustbehaftet ist, aber hohe Komprimierungsraten gegenüber unkomprimiertem Video oder auch sequentiellen JPEG-Bildern hat.

Durch die in der Regel höheren Quantisierungsfaktoren für höhere Frequenz-Koeffizienten geht leicht die Qualität harter Kanten verloren, wodurch z.B. Texte oder auch Linien (Zeichentrick) zur Artefaktbildung neigen. Für Photos jedoch, und damit auch für Film, eignen sich die Kompressionsschritte von MPEG sehr gut. Diese Tatsache und seine breite Anwendbarkeit bedingen, dass MPEG zehn Jahre nach seiner Veröffentlichung noch immer der am weitesten verbreitete Standard zur Kodierung digitalen Videos ist.

Literatur

[Shanawaz 1996] Shanawaz Basith, 1996, MPEG : Standards, Technology and Applications, http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol2/sab/article2.html, download 15.1.2003

[Tsang] George Tsang, -, MPEG-1 Video Codec, http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/mpeg1/, download 15.1.2003

[Tudor 1995] P.N. Tudor, 1995, MPEG-2 VIDEO COMPRESSION, http://www.bbc.co.uk/rd/pubs/papers/paper_14/paper_14.html, download 15.1.2003

[Fogg 1996] Chad Fogg, 1996, MPEG-2 FAQ, http://bmrc.berkeley.edu/frame/research/mpeg/mpeg2faq.html, download 15.1.2003

[Lincoln 1997] B. Lincoln, M. T. Malkin, 1997, Speed-up of Block Motion Estimation in Long-Term Memory Motion-Compensation Prediction, http://ise0.stanford.edu/class/ee392c/demos/lincoln_malkin/projhtml.html, download 15.1.2003

[Sikora 1997] Thomas Sikora, 1997, MPEG-1 and MPEG-2 Digital Video Coding Standards, http://wwwam.hhi.de/mpeg-video/papers/sikora/mpeg1_2/mpeg1_2.htm, download 15.1.2003

[Tanenbaum 2001] Andrew S. Tanenbaum, 2001, Computer Networks, 2nd edition, Prentice Hall

Bild-Quellen

Abbildung 2	[Tsang]
Abbildung 3	[Fogg 1996]
Abbildung 4	[Sikora 1997]
Abbildung 5	[Shanawaz 1996]
Abbildung 6	[Tudor 1995]
Abbildung 7	[Shanawaz 1996]
Abbildung 8	[Shanawaz 1996]
Abbildung 9	[Tudor 1995]
Abbildung 10	[Lincoln 1997]
Abbildung 11	[Lincoln 1997]

Eine Beschreibung von MPEG Audio

Von

Jörg Rebenstorf

Einleitung

MPEG-1/2 Audio Standard Part 3

Vorab ein Hinweis: Wenn im folgenden von MPEG-1 bzw. MPEG-2 geschrieben wird, bezieht sich das immer auf die Standards ISO 11172-3 bzw. ISO 13818-3, Second Edition 1998-04-15, also nur jeweils den Teil 3, in denen drei Audio Dekoder spezifiziert werden.
Generell beschreiben die entsprechenden Teile der Standards ein Verfahren zum Dekodieren eines bestimmten Datenstroms, dem MPEG Audio Bitstream.
Das Verfahren zum Kodieren ist nicht komplett spezifiziert, denn man erhofft sich so eine höhere Flexibilität für Implementierungen und Änderungen am Verfahren einen solchen MPEG Audio Bitstream aus PCM Daten zu erzeugen. So ist ein MPEG Audio Encoder definiert als jeder beliebige Encoder, der ein für ein MPEG Audio Dekoder dekodierbaren Audiostream produzieren kann. Bis heute ist jedoch am Encoder keine wesentliche Verbesserung veröffentlicht worden. Es existiert ein Encoder vom Fraunhofer Institut, einem der an der Entwicklung des Standards wesentlich beteiligten Organisationen und Patenthalter von MPEG Audio. Dieser Encoder gilt als der beste der verfügbaren Kodierer für den MPEG Audio Standard. Nicht fest vorgegeben sind vor allem die Psychoakoustischen Modelle des Encoders. Im Standard sind zwei verschiedene Psychoakustische Modelle vorgeschlagen, die jedoch nicht patentfrei implementierbar sind. Der Dekoder ist fest spezifiziert und immer patentfrei implementierbar. Der ISO Standard kostet jedoch Geld, so daß effektiv Kosten entstehen, wenn man einen MPEG Audio Dekodierer implementieren möchte. Der bekannte MPEG Audio Encoder Lame unter Unix hat ein eigenes Psychoakustisches Modell und kommt so ohne eine Lizenz vom Patent aus.
Das MPEG Audio Verfahren ist vom Berechnungsaufwand her asymetrisch, d.h. beim dekodieren fallen wesentlich weniger Berechnungen als beim umgekehrten Fall an. Dies ist erwünscht, da normalerweise deutlich öfter dekodiert wird als kodiert wird, ein dekodieren in Echtzeit wichtiger ist als ein kodieren in Echtzeit und die Dekodierer typischerweise beim Endverbraucher benutzt werden und in großer Stückzahl mit möglichst wenig Hardware- und Softwareresourcen implementierbar sein sollen, damit sich das MPEG Verfahren gegenüber anderen in der Massenanwendung durchsetzt.
Der MPEG-2 Standard stellt lediglich eine Erweiterung zu MPEG-1 da. Es ist kein neuer Audio Dekoder definiert worden. MPEG-1 und MPEG-2 definieren die gleichen Dekoder, genannt Layer I bis Layer III. Ein Layer-x Dekoder muss auch alle Bitstreams von den darunter liegenden Layern dekodieren können.
Ein Layer III Decoder muss also auch Layer II und Layer I Bitstreams dekodieren können, um dem Standard zu entsprechen. MPEG-2 erweitert den MPEG-1 Standard in Bereichen wie beispielsweise unterstütze Bitraten, 5-Kanal-Audio und unterstütze Sampling-Frequenzen. Der MPEG-2 Audio Standard ist rückwärtskompatibel zum MPEG-1 Audio Standard. Ein MPEG-2 Dekoder kann alles das dekodieren, was ein MPEG-1 Dekoder dekodieren kann, aber zusätzlich die Erweiterungen von MPEG-2. Ein MPEG-1 Dekoder, der einen MPEG-2 Audio-Bitstream bekommt, ignoriert hingegen die zusätzlichen MPEG-2 Daten automatisch, da die Erweiterungen in von MPEG-1 Dekodern ungenutzten Bereichen

Code	MPEG-1 Sampling Rate	MPEG-2 Sampling Rate
00	44,100	22,050
01	48,000	24,000
10	32,000	16,000
11	reserved	reserved

des MPEG-Bitstreams definiert sind (vorwärtskompatibel).
Bei allen MPEG-Layern liegen wichtige Schlüsselpunkte im Bitstream an Bytegrenzen, um ein möglichst effektive Berechnungen in Software zu ermöglichen. Die Polyphase Filter Bank ist statisch konfiguriert und kann in Hardware parallel durchgeführt werden. Vor- und Zurückspulen im MPEG Bitstream ist dadurch erleichtert, dass die Einheiten der Daten, die nur komplett dekodierbar sind, klein gehalten wurden und das springen zwischen solchen Blöcken einfach zu berechnen ist. Mehrfach kodieren und dekodieren (Kaskadierung) von MPEG Audio Daten ist nicht verlustfrei. Es kommen zusätzliche Fehler, sogenannte Artifakte, hinzu. Dieser Effekt ist jedoch laut Tests erst bei einer längeren Kette hörbar. Trotzdem sollte eine MPEG Audio Aufnahme, die evtl. nachbearbeitet werden soll, mit einer höheren Qualität kodiert werden als die, die nötig wäre um ein MPEG Audio Datenstrom, der nicht mehr verändert wird, zu kodieren, was in der Praxis nur durch eine höhere oder die variable Bitrate beim Encoder eingestellt werden kann. Wenn man beim ersten kodieren die eingeführten Fehler weit unter der Kurve der Hörschwelle und der der Maskierungseffekte hält ist es wahrscheinlicher, dass trotz Kaskadierung die nach und nach eingeführten Fehler noch nach vielen Stufen nicht hörbar bleiben. Fehler werden im MPEG Audio Verfahren durch die Quantisierung und die Rechenungenauigkeiten d.h. letztendlich auch durch die Bitzuweisung (bit allocation), die Skalierung und die Transformation durch die Filter verursacht. Filter, Skalierung und Bit-Allocation sind nur dann verlustfrei, wenn beliebig genaue Auflösungen für die Zahlendarstellungen zur Verfügung stehen würden.

Vergleich von MPEG Audio Layer III mit Ogg Vorbis

Ein Vergleich der Audioqualität mit konkurrierenden Verfahren wie OggVorbis ist nur schwer möglich. Zum einen kommt es bei perzeptionellen Verfahren immer auf die Audiodaten selbst an, wie gut ein Verfahren arbeitet. So gibt es z.B. Codecs die speziell für Sprache entwickelt werden und für andere Samples wenig geeignet sind (z.B. bei GSM). Auch ein spezieller Sprach-Encoder für MPEG Audio wäre denkbar. Ein direkter Vergleich mit OggVorbis ist ein deshalb streng genommen unmöglich, da es auf den bei MPEG nicht fest vorgegebenen Encoder entscheidend ankommt. Trotzdem gibt es Personen, die einen Vergleich zwischen der höchsten Qualitätsstufe eines OggVorbis Codecs und eines MP3 Codecs mit fester relativ hoher Bitrate (160 kbit/s) versuchen, wobei kein Vergleich von Messwerten betrieben wird, sondern lediglich der perzeptionelle Eindruck grob verglichen wird.
Dazu ist jedoch zu sagen, dass sich ein solcher Vergleich aus mehreren Gründen so wie beschrieben verbietet. Zum einen wird nicht der orginal PCM Datenstrom abgespielt, um den Teilnehmern zu ermöglichen festzustellen, wie sich die Daten optimal anhören sollen. So empfinden viele Menschen Musik, die mit sog. Loundness Funktion abgespielt wird, als „besser“ als das Orginal, obwohl die Musik lediglich in bestimmten Frequenzen verändert wird. Ein ähnlicher Effekt könnte, vorausgesetzt das Orginal wird als Referenz nicht abgespielt worden, auch bei diesem Vergleich zum tragen kommen. Es geht bei einem Vergleich von Codecs jedoch nicht darum, was sich „besser“ anhört, sondern welches Verfahren Orginal getreuer arbeitet. Zum anderen darf ein MPEG Audio Codec nicht „ausgebremst“ werden, indem man eine feste Bitrate einstellt, denn dann wird im Normalfall nicht das optimale Ergebnis entstehen, siehe Beschreibung des Verfahrens. Hier hätte der MPEG Audio Modus der variablen Bitrate gewählt werden müssen. Üblicherweise wären dann als Ergebnis der Kodierungen Dateien unterschiedlicher Länge entstanden, was den Vergleich wiederum nicht besonders fair erscheinen läßt, da mehr Bits Natur gemäß mehr Informationen darstellen *können* und somit eine schwächere Kodierung darstellen. Dies ist jedoch bei unserem Vergleich nicht zu vermeiden, da OggVorbis keine Kodierung mit fester

Bitrate also fester Dateilänge bei vorgegebener Playback-Zeit kennt.

MPEG Audio En-/Decoder

Allgemeiner Teil

Dieser Teil ist bei allen drei Layern gleich.
Als erster Schritt werden die sog. Side Information Daten, d.h. die Daten die den Dekoder steuern aus dem Bitstream gelesen. Dies sind unter anderem die Sampling-Frequenz, die Bitrate und der Modus (dual channel, verschiedene Stereo Modi oder Mono). Sie stehen im Header eines jeden Frames. Bitrate und Frequenz dürfen sich nur bei Layer III bei jedem Frame ändern. Dies wird Variable Bitrate genannt.
Die einzelnen Kanäle der Musik werden immer einzeln dekodiert, außer beim Modus „intensity stereo“ für Layer I und Layer II und bei „joint stereo“ für Layer III. Bei „intensity stereo“ werden beide Kanäle für bestimmte obere Subbands zusammen in einem Kanal kodiert.

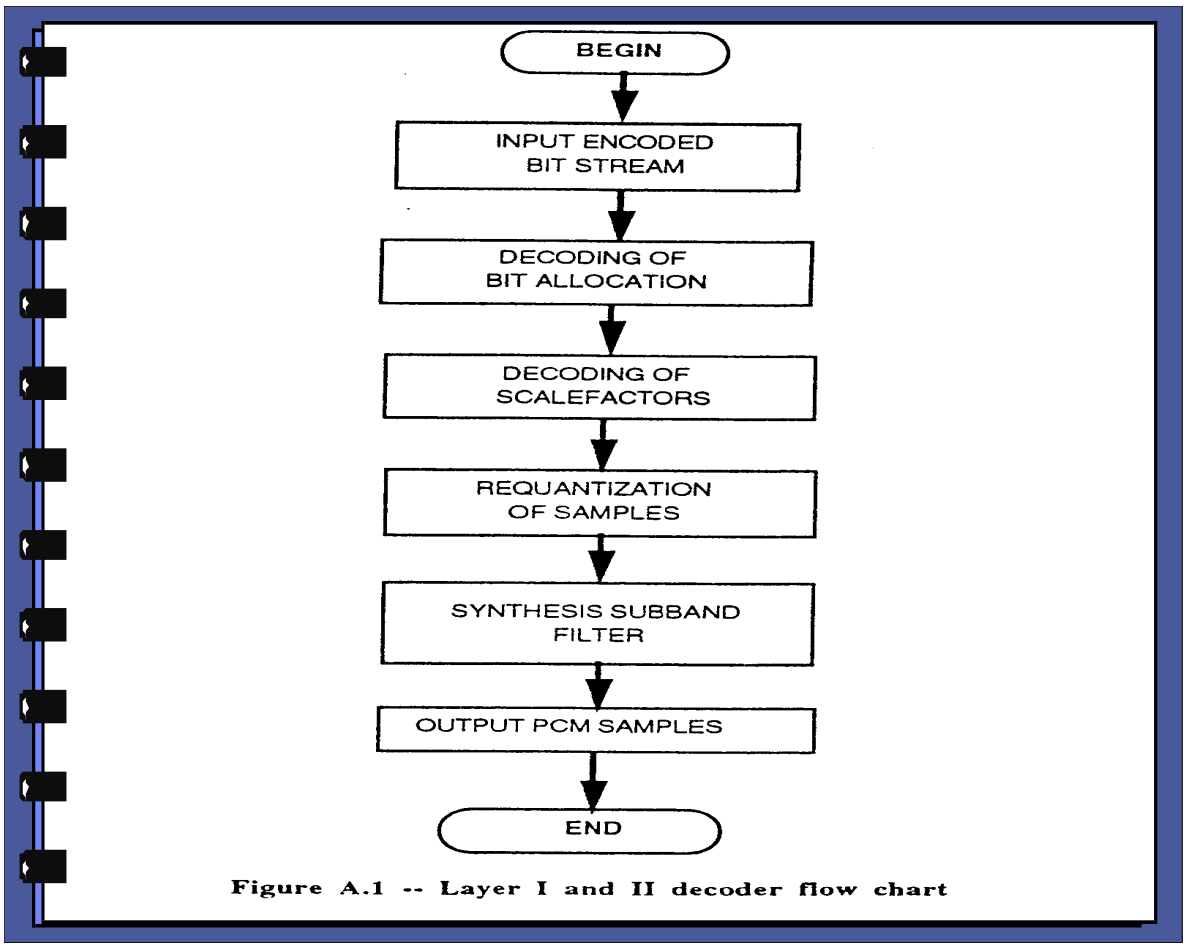


Figure A.1 -- Layer I and II decoder flow chart

Audio Layer I

Als zweiter Schritt wird das Breitbandsignal mit Hilfe einer Polyphase Filter Bank (in Software realisiert durch mathematische Funktionen; in Hardware parallel ausführbar aber i.A. auch mathematisch berechnet, um hohe Genauigkeit zu erreichen) in 32 gleich große Frequenzbänder (Subbands) aufgeteilt. Aus 32 PCM Werten, also Sample-Daten in der

zeitlichen Darstellung, werden dabei 32 Werte in den Subbands erzeugt, also Daten in einer Frequenzdarstellung; je ein Wert pro Subband. Diese Werte werden im folgenden Text „Subsamples“ genannt. Jeder dieser Subsample-Werte der Frequenzdarstellung beinhaltet Informationen zu allen 32 PCM-Daten; aber nur jeweils Informationen eines kleinen Teils eines jeden PCM-Werts. Die 32 Subsample-Daten sind also nur komplett transformierbar, d.h. um einen einzigen PCM-Wert zu erhalten braucht man Informationen aus allen 32 Subband-Daten.
Zusätzlich sind die 32 Subband-Werte im Filter je zur Hälfte zeitlich überlappt mit davor und dahinter liegenden Subband-Werten, so dass sukzessiv je zwei Frames zusammen dekodierbar sind.

Qualität der Filterfunktion

Nachteilig für die Qualität des Verfahrens ist, dass die Filterfunktion die möglichen Frequenzen nicht völlig getrennt voneinander aufspaltet. Statt dessen überlappen sich benachbarte Frequenzbänder. Eine einzige Frequenz kann daher auch Werte in zwei Subbands beeinflussen. Außerdem sind der Filter und sein Inverses nicht verlustfrei. Der Fehler des Filters und seines Inversen ist jedoch klein. Der Vorteil ist die geringe Komplexität der Berechnung der Filterfunktion und seiner Inversen. Den Nachteil, dass die Subbands nicht mit den kritischen Bändern des menschlichen Gehörs übereinstimmen, wird dadurch ausgeglichen, dass den unteren Bändern vom Encoder mehr Bits zugewiesen werden als den hohen. Die unteren Bänder sind beim menschlichen Gehör schmaler also genauer als die kritischen Bänder in denen die hohen Frequenzen wahrgenommen werden. Mehr Bits sind gleichbedeutend mit genauerer Auflösung. Die kritischen Bänder des Gehörs sind definiert als die Frequenzbereiche in denen sich Frequenzen gegenseitig beeinflussen, also z.B. gegenseitig maskieren können.

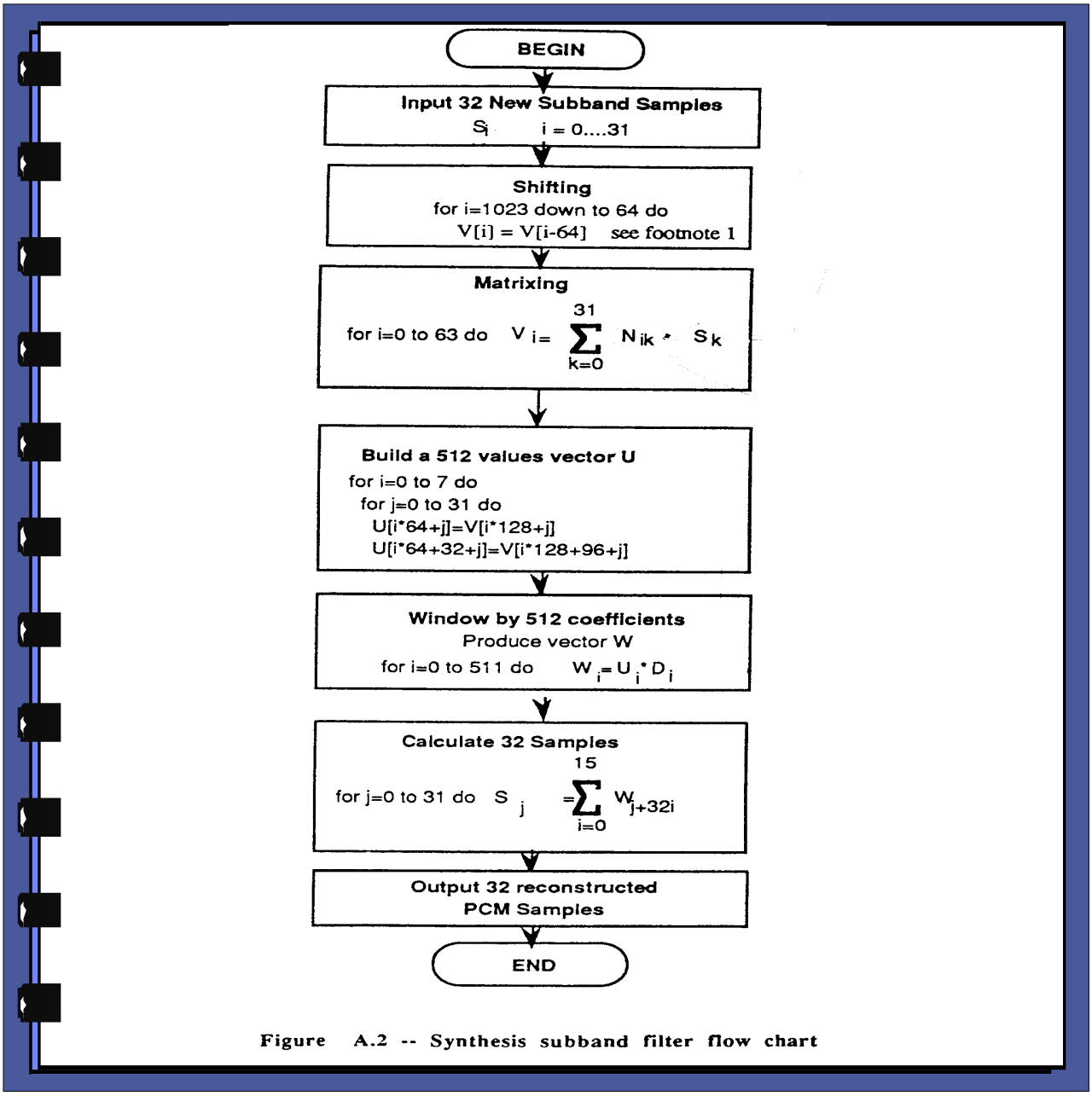


Figure A.2 -- Synthesis subband filter flow chart

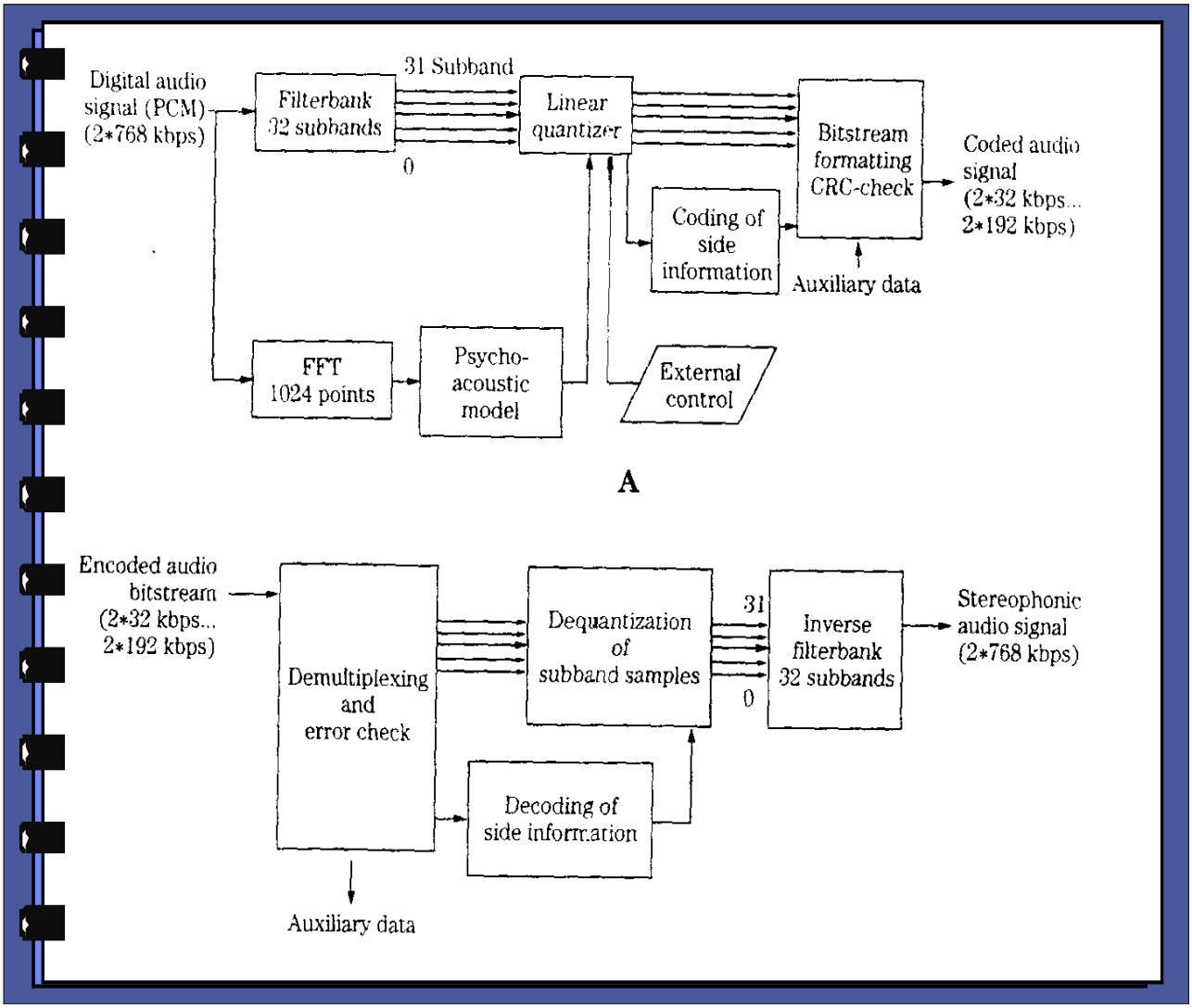
Filter, Gruppierungen, Framing

Zur Wiederholung: Der Filter gibt aus 32 PCM Samples je einen Wert pro Subband aus. Da es 32 Subbands sind, sind es insgesamt auch wieder 32 Werte. Bei Layer I werden 12 Subband-Samples in jedem der 32 Subbands gruppiert. Angewandt auf alle 32 Subbands jedes Kanals ergibt sich ein Frame, der also Informationen zu 384 Breitband-Samples pro Kanal enthält. Es gehören 12 Subsamples aus Subband Nummer Null zusammen und 12 Samples aus Subband Nummer Eins usw. Sie stehen jedoch nicht in dieser Reihenfolge im Bitstream. Die Reihenfolge der Daten kann man Punkt 2.4.1.6 des Standards entnehmen. Zusammengehörig sind sie in so fern, dass die Anzahl der Bits mit denen diese 12 gruppierten Subsamplewerte gespeichert sind zusammen bestimmt werden durch einen einzigen Bit-Allocation-Wert, d.h. sie haben immer die gleiche Bitlänge. Diese Bitlänge („allocation[ch][sb]“) steht zwar nicht

direkt als Wert an der entsprechenden Stelle im Bitstream aber die dazugehörige Tabelle tut fast das gleiche, nämlich einen 4-Bit Wert auf einen 4-Bit Wert abbilden. Dies bewirkt keinerlei Datenreduktion. In erster Linie ist es gedacht, um den Wert 1111b zu vermeiden, damit der Startcode nicht aus Versehen im Bitstream auftauchen kann. Die einzelnen Sample-Daten haben eine Bitlänge zwischen [2,15], sofern Bits alloziiert wurden. Die Subband-Werte einer Gruppe enthalten zeitlich aufeinander folgende Informationen über ein bestimmtes Frequenzband. Der Skalierungsfaktor („scalefactor[ch][sb]“) steht ebenfalls nicht direkt im Bitstream sondern statt dessen ein Index auf diesen Wert. Die zugehörige Tabelle, um den echten Skalierungsfaktor zu finden, muss im Dekoder erzeugt oder fest abgespeichert werden. Diese Tabelle bildet einen 6 Bit-Wert auf einen Gleitkommawert zwischen [2.0, 0] mit 14 Nachkommastellen ab, was eine echte Datenreduktion bewirkt.

(De-) Kodierverfahren

Beim Kodieren wird für jede Gruppe bestehend aus 12 Subband-Samples eine Bit-Allocation größer Null vergeben, sofern nicht das ganze Subband als nicht hörbar eingestuft wird, in welchem Fall der Wert Null vergeben wird. Dieser zugeteilte Wert ist die Bitlänge jedes der zwölf Werte dieses Subbands. Je höher also diese ganzzahlige positive Zahl ist, um so genauer können die 12 aufeinander folgenden Subband-Werte einer Gruppe (eines jeweiligen Subbands) kodiert werden.
Auch die (Re-)Quantisierungsfunktion bestimmt sich aus dem Bit-Allocation-Wert. Nachdem der Subsample-Wert mit Hilfe der Skalierungsfaktors beim Dekodieren (durch Multiplikation) normalisiert wurde, wird er mit der entsprechenden Funktion requantisiert. Von dem so gebildeten Wert werden die höchstwertigsten N Bits genommen, wobei N der Bitlänge des Bit-Allocation-Werts entspricht. Schließlich wird noch das MSB invertiert, um wiederum auszuschließen, dass der Startcode aus versehen erzeugt wird. Der so erzeugte Wert ist das Ergebnis der Dekodierung; ein PCM Wert.
Der Bit-Allocation Wert wird je nach Psychoakustischem Modell beispielsweise basierend auf dem berechneten Masking-Threshold bestimmt, also der Hörschwelle einschließlich Maskierungseffekte. Um diese Kurve zu berechnen wird eine 512-Subsamples FFT-Weitband-Transformation berechnet. Da der Encoder jedoch mit Absicht nicht im Standard komplett spezifiziert ist, sondern lediglich zwei Psychoakustische Modelle vorgeschlagen werden, ist diese Methode zum Zuteilen des Bit-Allocation-Werts nicht fest vorgegeben. In jedem Fall wird die FFT Kurve nicht mit im Bitstream abgespeichert sondern lediglich die Subsamples. Für jede Gruppe wird ein Scalefactor berechnet, z.B. basierend auf dem größten Subsample-Wert in der jeweiligen Gruppe eines jeden Subbands. Jeder der Subband-Samples einer Gruppe wird mit dem gleichen Scalefactor dividiert und dadurch normalisiert. Die Normalisierung führt dazu, dass die Auflösung des Quantisierers optimiert wird. Die Quantisierung wird iterativ vorgeschlagen. Wenn noch Bits in einem Frame Platz haben werden sie nach und nach zu den Subbands basierend auf dem SMR (signal-to-mask ratio) zugeteilt, womit sich das jeweilige S/N (signal-to-noise) Verhältnis über das Minimum erhöhen kann. Subbands mit einem großen SMR bekommen bevorzugt Bits im Vergleich zu Subbands mit kleineren SMRs. Die SMRs geben jeweils einen Wert an, der aussagt bei welcher Granularität der Quantisierer Rauschen als Fehler einfügt, also Rauschen, das nicht maskiert wird.



Audio Layer II

Bei Layer II ist das Dekodierverfahren sehr ähnlich zu dem von Layer I. 36 Subsamples werden hier statt 12 logisch gruppiert und in drei Teile unterteilt, die jeweils wieder die 12 Subsamples einer Layer I Gruppe sind. Diese 36 Subsamples werden „Granule“ genannt. Sie sind zeitlich aufeinander folgende Werte eines Subbands. Es gibt ein Granule pro Subband pro Kanal pro Frame.

Die physikalische Verschaltelung im Bitstream unterscheidet sich von der logischen. Im Bitstream stehen immer Triplets von Subsamples (bzw. ein Subsamplecode). Ein Triplet bei Layer II sind drei Subsample-Werte (oder ein Subsamplecode) pro Kanal (ein oder zwei Kanäle) eines Subbands. Zwölf dieser Triplets ergeben einen Granule. Die Triplets werden sukzessiv gespeichert für alle Kanäle, dann für jedes Subband und schließlich stehen in der äußersten Ebene der physikalischen Struktur zwölf mal alle Werte von allen Subbands. Da zwölf mal drei gleich 36 sind, werden so 36 Subsamples pro Kanal für jedes Subband logisch gruppiert und Granule genannt. Siehe auch Punkt 2.4.1.6 des Standards.

Um ein Granule aus dem Bitstream auszulesen muss also im Bitstream mehrfach vorwärts gesprungen werden.

Viele Informationen sind durch zusätzliche Tabellen bei Layer II indirekt im Bitstream

gespeichert, was die Datenreduktion erhöht. So wird in bestimmten (festen) Subbands bei bestimmten starr festgelegten Bit-Allocation-Werten für dieses Subband eine Kodierung benutzt, bei der der Subsample-Wert nur einmal statt dreimal abgespeichert wird und dann als „samplecode“ statt als „sample“ bezeichnet wird, siehe auch Tabelle B.4 des Standards. Oder anders ausgedrückt der Samplecode gilt in diesem Fall für drei Subsample-Werte. Diese Methode wird „grouping“ genannt. Die drei Subsample-Werte werden durch einen speziellen Algorithmus aus dem Samplecode berechnet.
Auch der Skalierungsfaktor kann mehrfach (für mehrere Subsample-Werte) verwendet werden. Ob und wie dies geschieht bestimmt sich aus einem zusätzlichen Wert im Frame, dem Scalefactor-Selection-Information-Wert („scfsi[ch][sb]“). An der Schreibweise „scfsi[ch][sb]“ ist auch zu erkennen, das der „scfsi“-Wert pro Kanal und pro Subband in einem Frame unterschiedlich sein kann. Der Skalierungsfaktor selbst steht wie gehabt, als Index in eine Tabelle im Datenstrom. Bei Layer II sind es bis zu drei Skalierungsfaktoren pro Kanal und pro Subband, die hintereinander im Frame gespeichert sind. Die Tabelle ist die selbe wie bei Layer I.
Die Länge des Bit-Allocation-Werts ist bei Layer II innerhalb eines Frames nicht immer gleich, sondern je nach Sampling-Frequenz, Bitrate und Subband eine andere (Table B.2 ff), wird allerdings nicht vom Encoder bestimmt sondern ist für jede Kombination von Sampling-Frequenz und Bitrate fest vorgegeben. Der Wert selbst ist wiederum ein Index in eine Tabelle, die einen zwei bis vier Bit langen Wert auf einen Wert zwischen drei und 65535 abbildet, oder auf den Wert „nicht alloziiert“. Der zugehörige Wert in der Tabelle ist die Anzahl der benutzten Quantisierungsstufen für jeden der 36 Subsamples dieses Subbands dieses Kanals des Frame. Daraus bestimmt sich die benutzte Quantisierungsfunktion. Die Subsample-Daten stehen ansonsten, wie bei Layer I, „raw“ im Bitstream, sind also nicht weiter kodiert, es sei denn „grouping“ wird benutzt. Man erkennt an der Tabelle, dass manche der höchsten Subbands nie Bits zugewiesen bekommen. Für niedrige Subbands gibt es mehr Auswahlmöglichkeiten an Quantisierungsstufen als bei höheren.
Der Filter ist identisch mit dem von Layer I. Es werden wiederum 32 Subband-Werte, je einer aus einem Subband, pro Kanal zusammen dekodiert und man erhält 32 PCM Werte.
Im Vergleich zu Layer I ist festzustellen, dass gerade bei den Bit-Allocation-Werten weniger Flexibilität bei der Auswahl der Werte möglich ist, was jedoch die Kodierung verstärkt. Die Quantisierungsfunktionen sind sehr ähnlich. Durch „grouping“ und „scfsi“ sind Zusammenlegungen von Subsample- und Skalierungsfaktor-Werten möglich. Im Falle von „grouping“ für Subsample-Triplets zu einem Subsamplecode und im Falle von „scfsi“ für Skalierungsfaktoren, die für Untermengen von den drei Teilen des Granule gelten.

Audio Layer III

Die berühmten MP3 Dateien aus dem Internet sind Bitstreams, die mit MPEG-1 Audio Layer III dekodiert werden können und damit auch mit einem MPEG-2 Audio Layer III Dekoder.
Layer III benutzt ebenfalls die gleichen Filter wie die beiden vorher beschriebenen Layer. Zusätzlich ist hier jedoch eine IMDCT beim dekodieren dem Synthesefilter vorgeschaltet. Beides zusammen wird Hybrid Filterbank genannt. Jedes Subband wird mit Hilfe der MDCT beim kodieren zusätzlich in 18 Frequenzlinien aufgespalten. Dies erhöht die Frequenzauflösung. Die IMDCT erzeugt aus 18 Werten 36 Werte. Da die 36 Werte genau zur Hälfte zeitlich überlappt sind mit den davor liegenden und den danach liegenden 18 Werten, entstehen in jedem Schritt effektiv nur 18 Werte nicht 36. Die ersten 18 werden mit den letzten 18 des davor liegenden Transformationsblocks überlappt und sind die neu berechneten

Werte. Die zweiten 18 neuen Werte werden gespeichert für den folgenden Block. Sogenanntes „adaptive window switching“ wird benutzt, um Pre-Echoes (Zeit-Artifacte) zu reduzieren. Es gibt dabei kurze und lange Fenster. Die Frequenz oberhalb derer kurze Blöcke verwand werden, um eine bessere Zeitauflösung zu erreichen, kann ausgewählt werden.
Es wird ein nicht linearer (non-uniform) Quantisierer benutzt, um die Frequenzkomponenten zu quantisieren. Der Betrag der Werts vom Huffman Dekoder wird exponentziert mit 4/3.
Außerdem sind die Werte Huffman kodiert. Dazu gibt es 18 verschiedene Tabellen. Die Tabelle wird durch den Wert „table_select“ bestimmt, der direkt im Bitstream steht und für bestimmte Regionen eines Granule eines Frame gilt. Außer der Auswahl der Tabelle legt „table_select“ auch die Anzahl der ESC-Bits fest, „linbits“ genannt. Zusätzlich gibt es einen Wert „big_values“ im Bitstream. Dieser ist ein Zähler von Frequenzlinienpaaren in Region Null, Eins und Zwei. Die verbleibenden Huffmancodebits werden als Quadrupel anhand einer Tabelle, die durch „count1table_select“ ausgewählt wird dekodiert. „count1table_select“ steht direkt im Bitstream. Es wird solange Huffman dekodiert bis entweder 576 quantisierte Werte von Frequenzlinien berechnet sind oder bis keine Huffmancodes mehr übrig sind, je nach dem was zuerst eintritt. Wenn es mehr Huffman Codes gibt als für 576 Frequenzlinien, dann wird der Rest als Stopfbits betrachtet und ignoriert.
Es gibt einen main_data_begin Pointer, der einen Byteoffset darstellt und den Anfang des sogenannten „main data stream“ kennzeichnet. Dies ist der Teil des Bitstroms, in dem die Scalefactors und die Huffman Codes stehen, sowie frei wählbare Zusatzdaten („ancillary_bit“).
Ein Layer III Frame besteht aus dem üblichen Header, dann folgt der Side-Information-Stream und der Main-Data-Stream beginnt irgendwo davor („main_data_begin“ dieses Frame), beginnt spätestens genau hinter dem Side-Information-Stream dieses Frame (wenn Pointer gleich Null) und endet spätestens genau vor dem Anfang des folgenden Frame Header, endet evtl. jedoch auch schon früher, was bestimmt wird durch den main_data_begin Pointer des folgenden Frame. Der Pointer ist 9 Bit lang und adressiert Bytes, daher ist der maximale Unterschied in der Länge zweier aufeinander folgender Frames gleich 4096 Bits.
Die Scalefactors werden aus einem Wert im Bitstream namens „scalefac_compress“ dekodiert. Zunächst werden zwei Werte nämlich slen1 und slen2 direkt daraus berechnet. Dann werden diese Werte als Indizes in eine Tabelle, in der die Scalefactors selbst stehen, betrachtet. Die Tabelle ist als Alternative auch berechenbar. Je nach Wert von „scfsi“ (scalefactor selection information) sind die so bestimmten Scalefactors auch für den zweiten Granule gültig. Die Scalefactors berechnen sich unterschiedlich je nach Window Typ. Bei „short windows“ fließen „global_gain“, „subblock_gain“, „scalefac_multiplier“ und „scalefac_s“ Werte aus dem Bitstream in die Berechnung ein. Bei „long windows“ sind es „global_gain“, „scalefac_multiplier“, „preflag“ und „scalefac_l“, sowie „pretab“, der sich aus einer Tabelle bestimmt.

Anwendungsgebiete

MP3-Player findet man mittlerweile in vielen Geräten. Dazu gehören (tragbare) CD-Player, USB Memory Sticks, Autoradios mit CD-Player, Mini-CD-Player, Handys, PDAs. Wobei in den PDAs und Handys meist keine spezielle Hardware für MP3 Dekodierung eingebaut ist, sondern ein reiner Software-Player benutzt wird. Außerdem kann man natürlich auch in jedem Desktop oder Laptop einen Software-Player einsetzen. Selten haben Soundkarten für Computer Hardware-Dekoder für MPEG Audio Layer III.
Layer II Dekoder findet man in Set-Top-Boxen für DVB-T, DVB-C und DVB-S also im Fernsehbereich von Europa und im Digital Tape System, einem vor allem in den USA

bekannten digitalen Nachfolger der klassischen analogen Audio Tape Technik. Auf DVDs ist kein MPEG Audio Layer II Sound gespeichert, hier wird das „Dolby Digital“ System benutzt. Auch bei Videospeicherformaten wie MPEG Video und AVI kommt für den Soundteil MPEG Audio Layer II zum Einsatz.

Zusammenfassung

MPEG Audio sind verschiedene Audio Dekoder, die für eine generelle Audioanwendung gedacht sind. Sie sind nicht für ein spezielles Anwendungsgebiet, wie z.B. Sprachkodierung konzipiert. Layer I und II sind relativ simpel zu implementieren. Hauptschwierigkeit ist hier die effiziente Implementierung des Synthesefilters. Audio Layer III ist deutlich komplexer durch die Verwendung einer IMDCT und eines dynamischen Bit-reservoirs, das die Pufferverwaltung komplexer macht. Auch die Verschachtelungen im Bitstream sind komplexer als bei den anderen Layern. Bei gleicher Bitrate sollte ein Layer III Bitstream regelmäßig bessere Ergebnisse im Sinne von perzeptioneller Unverfälschtheit erzielen als ein Layer II oder I Bitstream. In erster Linie bedingt durch Verwendung von IMDCT und Huffman Kodierung (bei „perzeptionell gleich gutem“ Encoder).

Literatur

MPEG-1 Standard ISO/IEC 11172-3 First edition 1993-08-01 Part 3: Audio
MPEG-2 Standard ISO/IEC 13818-3 Second edition 1998-04-15 Part 3: Audio
Tim Kientzle: A Programmer´s Guide to Sound, Addison Wesley 1998.
Grill et al.: United States Patent 5,579,430 Nov. 26, 1996
Ken C. Pohlmann: Principles of Digital Audio 3rd Edition

MPEG-4 - Ein Überblick

von

Martin Goralczyk

Einleitung

Was ist MPEG-4?

Der Bereich der bewegten Videobilder auf dem Computer hat sich in der letzten Zeit sehr stark verändert. Waren die immensen Daten noch vor wenigen Jahren kaum für PCs zu bewältigen, wird es heute immer selbstverständlicher selbst auf kleinen tragbaren Geräten, beispielsweise Mobiltelefonen, bewegte Bilder, wie Animationen und Musikvideos, zu betrachten, an andere zu verschicken oder sogar selbst zu kreieren.
Durch diese neuen Möglichkeiten sind Fernsehen und Internet nun näher aneinandergerückt und nur noch als Medien unterscheidbar, aber nicht unbedingt was die Informationen oder Unterhaltung betrifft. Die Informationen sollen auf allen Medien und Endgeräten nutzbar und nicht von technischen Barrieren verstellt sein. Das Fernsehen wird in Zukunft immer interaktiver werden und sich dem Internet annähern, und andererseits wird das Internet immer selbstverständlicher und überall zugreifbar, wie es das Fernsehen heute schon ist.
So wie es in Europa für das Fernsehen den PAL Standard gibt, musste nun ein Videostandard für die Computerwelt und die Schnittmenge zwischen Internet und Fernsehen geschaffen werden, der folgende Anforderungen erfüllen sollte:
Er sollte natürlich möglichst gut komprimieren, ohne auffällig sichtbar an der Qualität zu sparen.
Er sollte gut skalierbar sein; derselbe Videostrom sollte also die verschiedensten Geräte bedienen, vom Handy bis zum Großbildfernseher. Dabei ist zu beachten, dass die verschiedenen Geräte einerseits verschiedenbreite Anbindungen haben, und andererseits aufgrund von Display oder Rechenleistung verschieden gut darstellen können.
Er sollte Interaktivität gewährleisten und damit inhaltsbasiert auf die Videodaten zugreifen können; nicht mehr wie bisher nur zeit- und ortsbezogen. Also müssen einzelne Objekte im Video, wie zum Beispiel Personen auch als solche abgespeichert werden, damit man auch auf sie zugreifen kann.
Er sollte sowohl mit natürlichen als auch mit synthetischen Quelldaten gut zurechtkommen, was nicht selbstverständlich ist, da beide verschiedene Strategien verlangen.
Die Vorgänger MPEG-1 und MPEG-2 haben all diese Punkte nicht erfüllt, obwohl sie schon seit Jahren in kommerziellen Produkten für das Wohnzimmer wie CD-i (MPEG-1) und DVD (MPEG-2) verarbeitet waren. Grund dafür ist sicher die feste Struktur, die den beiden Formaten zugrunde liegt. Sie nutzen ein festes Kompressionsverfahren, das sich auf einzelne Situationen nicht anpassen lässt. Im Gegensatz dazu nutzt MPEG-4 keinen festen Kompressionsalgorithmus, sondern ist eine Toolsammlung, die für viele verschiedene Situationen Speziallösungen bietet. Desweitren sind die beiden Vorgängerstandards auch in MPEG-4 implementiert, so dass es abwärtskompatibel ist.

Mythos DIVX

Nicht immer sind die Ambitionen für etwas neues so edel wie im letzten Abschnitt beschrieben. Viele Nutzer haben auf MPEG-4 gewartet, um Filme in guter Qualität mit vertretbarem Aufwand übers Internet zu tauschen. Die Vorgänger Codecs waren nicht so leistungsfähig, so dass gute Qualität riesige Datenmengen hervorrief, die wiederum nur langsam über ein Netz wie das Internet zu verschicken waren. In erster Linie gilt das Interesse natürlich den teuren Hollywood Produktionen und nicht den selbstproduzierten Privataufnahmen, also DVD Material kostenlos über das Internet zu beschaffen. Die beiden Probleme waren das *Content Scrambling System* und die riesige Datenmenge.

Hinter dem CSS steckt ein Verschlüsselungssystem dessen Schlüssel im DVD-Player vorhanden und geheim war, und der deswegen auch verschlüsselt werden musste. Dies wurde allerdings im Xing DVD-Player versäumt, so dass findige Linux Entwickler ihn zuerst für eigene Player benutzten, und er heute in vielen Tools steckt, mit denen man DVDs auslesen kann.
Dem Problem der großen Datenmengen konnte man nun mit MPEG-4 begegnen. Da dies aber ein ISO-Standard (ISO 14496) ist, und damit nicht kostenlos, passierte folgendes: Jerome Rota hackte 1999 den damaligen von Microsoft entwickelten MPEG-4 Codec und nannte ihn DIVX, ein Name für ein System, dass digitale Inhalte schützte. MPEG-4 wurde im Oktober 1998 als Standard definiert und im Dezember 1999 nochmals überarbeitet und eine Version zwei standardisiert. Dieser Microsoft Codec war nicht ISO 14496 kompatibel und damit auch nicht DIVX. So kann man abschließend dazu sagen, dass DIVX ein kostenloser leistungsfähiger Codec auf MPEG-4-basis ist, der mit dem ISO-Standard nicht konform geht, aber Filme in ausreichend guter Qualität komprimiert.

Ein erster Eindruck

Um mir einen ersten subjektiven Eindruck von MPEG-4 zu verschaffen, verglich ich die MPEG Codecs miteinander. Ich codierte eine 30-sekündige Sequenz mit den drei verschiedenen Standards und verglich Rechenzeit, Dateigröße und Qualität des resultierenden Videos. Um möglichst gleiche Vorraussetzungen zu schaffen verwendete ich dasselbe Quellmaterial für alle drei mit folgenden Eckdaten:
640 x 480 Pixel Auflösung
25 Bilder pro Sekunde
500 KBit/s für Video
128 KBit/s für Audio
beste Qualität
Als MPEG-1/2 Encoder benutzte ich TMPGEnc, als MPEG-4 Encoder das Div-X Derivat 3ivX.
Die Ergebnisse waren eindeutig:
Der MPEG-4 Encoder war doppelt so schnell wie der MPEG-1/2 Encoder, die MPEG-4 Datei war etwa 3,5% kleiner als die beiden anderen und die Qualität bei MPEG-4 war beeindruckend viel besser als bei den beiden Vorgängern. Während bei MPEG-1/2 das Bild nur aus groben Blöcken bestand, wenn die Szene bewegungsintensiv war, blieb das Bild bei MPEG-4 fein aufgelöst.
Ein Bildbeispiel aus der Videosequenz die als MPEG-2 codiert worden ist zeigt Abbildung 1, das entsprechende Gegenstück aus der DIVX kodierten Sequenz zeigt Abbildung 2.

Abbildung 1: MPEG 2

Abbildung 22: DIVX

Im weiteren Verlauf werde ich in Kapitel 2 auf die Grundstrukturen solch eines Videos eingehen, in Kapitel 3 und 4 auf die verschiedenen Tools für die Audio- und Videokompression. In Kapitel 5 werden noch einige Extras angesprochen, für die ein eigenes Kapitel zuviel gewesen wäre und zum Abschluss wird in Kapitel 6 die Seminararbeit kurz zusammengefasst und ausgewertet.

Strukturen

Struktur einer Szene

In diesem Kapitel werden die Grundstrukturen von MPEG-4 anhand eines einfachen Beispiels erläutert. Wie in der Einleitung schon erwähnt, sollen bei MPEG-4 einzelne Objekte aus denen sich das Video zusammensetzt abgespeichert werden, und nicht das Video als eine Menge von Pixeln. Dabei ist es unerheblich ob diese einzelnen Szeneninhalte natürlichen oder synthetischen Ursprungs sind. Sie werden ohnehin separat kodiert und können dann zusammengesetzt in einem Videostrom gespeichert werden. Der Zusammenhang in der Szene entsteht durch den Szenengraphen, einen gerichteten zyklenfreien Graphen, also einen Baum, der die einzelnen Objekte vierdimensional abspeichert. Das bedeutet er speichert die Koordinaten wo das Objekt im Filmausschnitt zu sehen sein soll, und zu welchem Zeitpunkt. Die einzelnen Objekte werden auch primitive Objekte genannt, und teilen sich in Video und Audio Objekte. Diese beiden wiederum können natürlicher oder synthetischer Herkunft sein.

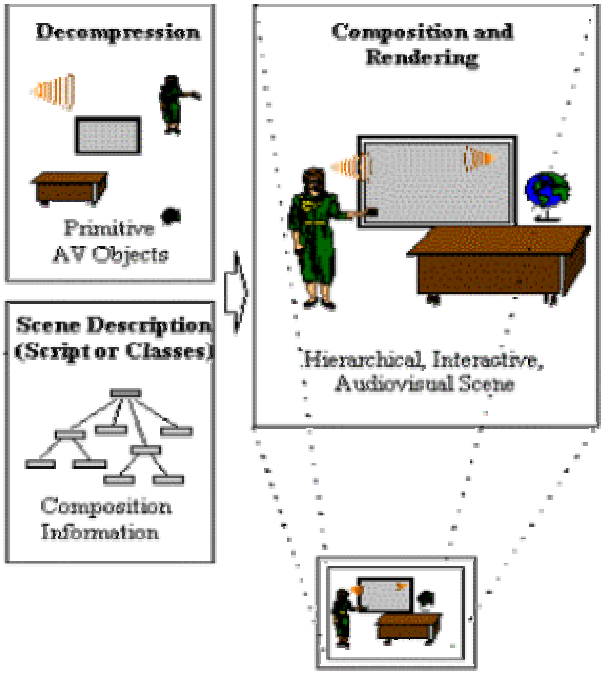


Abbildung 3: Aufbau einer MPEG-4 Szene (Bild aus [6])

Abbildung 3 zeigt eben solch eine zusammengesetzte Szene. Der Kasten oben links zeigt die einzeln dekomprimierten Objekte Sprecherin, Schreibtisch, Globus, Leinwand und Sprache. Hier ist es unwichtig, ob die Objekte beispielsweise synthetisch dreidimensional oder natürlich zweidimensional sind. Es könnten all diese Videoobjekte dreidimensional sein, und auf der Leinwand würde ein zweidimensionaler Film ablaufen. Der Szenengraph aus dem Kasten unten links fügt die Objekte zu dem Gesamtbild zusammen.
Weiterhin verfügt der Szenengraph noch über ein weiteres Objekt, wobei es sich um die virtuelle Kamera handelt. Die bestimmt den gezeigten Bildausschnitt den dann auch der Zuschauer sehen kann. So kann, wie im wirklichen Leben auch, die Szene im Dekoder viel größer sein als es der Bildausschnitt zeigt, und erst ein Schwenk des Bildausschnittes über alle Objekte offenbart alle Details. In Abbildung 3 ist das durch die gepunkteten Linien dargestellt.
Die Vorteile dieser separaten Kodierung sind die Manipulierbarkeit, und damit die gewünschte Interaktivität. Es lassen sich so zum Beispiel Sprachen auswählen (Austausch der Audio-Objekte), oder auch andere Objekte löschen oder anfügen. Wenn die Verbindung zum Sender des Videos bidirektional ist, lassen sich die benötigten Objekte auswählen, so dass nur diese übertragen werden müssen und nicht mehr der gesamte Strom.
Wenn die gesamte Szene dreidimensional gespeichert ist, ließe sich auch die gewünschte Perspektive einstellen, beziehungsweise ein gewünschter Bildausschnitt.

Szenengraph

Um dem Beispiel aus Abbildung 3 treu zu bleiben, zeigt Abbildung 4 den passenden Szenengraphen.

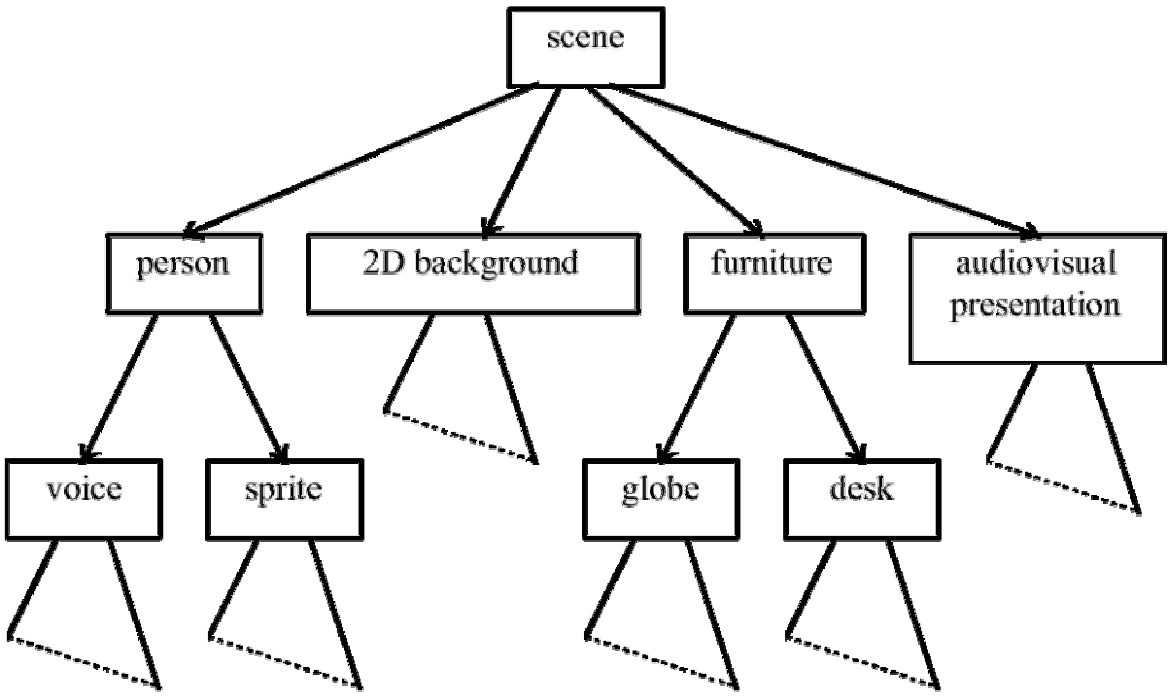


Abbildung 4: Szenengraph (Bild aus [1])

Der Wurzelknoten in diesem Graphen ist eine *grouping node*, ein Knoten der die Objekte einer gesamten Szene zusammenfasst. Die Kinder der *grouping nodes* heißen *children nodes*, sind also Knoten die ein bestimmtes Objekt repräsentieren. Es gibt auch noch den Typus der *bindable children nodes*, Knoten die in jeder Szene nur eine Instanz haben dürfen, wie etwa die Sprache oder der Bildausschnitt. Interessant sind auch die *sensor nodes*, knoten die mit Sensoren (mouseover, etc.) ausgestattet sind, um Benutzereingaben zu ermöglichen.
Eine größtmögliche Interaktivität ist mit der Möglichkeit Java beziehungsweise Javascript zu nutzen gegeben.
MPEG-4 basiert auf dem Binary Format for Scene Description (BIFS), welches den kompletten Virtual Reality Modeling Language Standard (VRML) beinhaltet. VRML ist eine Beschreibungssprache von 3D-Modellen, wie Kugeln, Quadern oder ähnlichem. Bei VRML können Objekte eben mit diesen Sensoren oder Skript Knoten versehen werden, in denen der Java-Code ausgeführt wird. Somit ist die Interaktivität in den Videos praktisch unbeschränkt. Es gibt beispielsweise Videos, bei denen der Mausklick auf bewegte Objekte auf die passende Homepage verlinkt, oder Videos, in denen sich die Objekte, verschieben, rotieren oder ausblenden lassen.

Schichtenmodell

Ein MPEG-4 Decoder besteht im wesentlichen aus den drei Schichten Delivery Layer, Synchronisation Layer, Decompression Layer und ist auf Abbildung 5 dargestellt.
Die Delivery Layer hat im wesentlichen folgende Aufgaben: Empfangen des Datenstroms und demultiplexen der einzelnen Elementaren Ströme. Für das Empfangen wird ein eigenes Transportprotokoll, das Delivery Multimedia Integration Framework (DMIF) benutzt, das

ähnlich wie FTP arbeitet. Allerdings werden von der Decoderseite nur Pointer auf die Objekte an den Server gesendet, die auch übertragen werden sollen. Dieser Datenstrom wird dann mit dem Demultiplexer in die einzelnen Elementaren Ströme (ES) zerlegt, wobei Elementare Ströme normalerweise einem primitiven Objekt wie zum Beispiel dem Szenengraph entsprechen.
Die Synchronisation Layer versieht die Objekte mit Zeitstempeln, ist also für die Synchronisation der Objekte zuständig, des weiteren auch für die Fehlererkennung und -korrektur.
Die Decompression Layer dekomprimiert die Elementaren Ströme und gibt diese an den Renderer der die Objekte in der Szene zusammenstellt den sichtbaren Ausschnitt wählt und das Bild dann ausgibt.
Selbstverständlich besteht auch eine Rückverbindung, so dass der Benutzer noch Einfluss nehmen kann wie etwa spulen, Sprache wechseln etc.

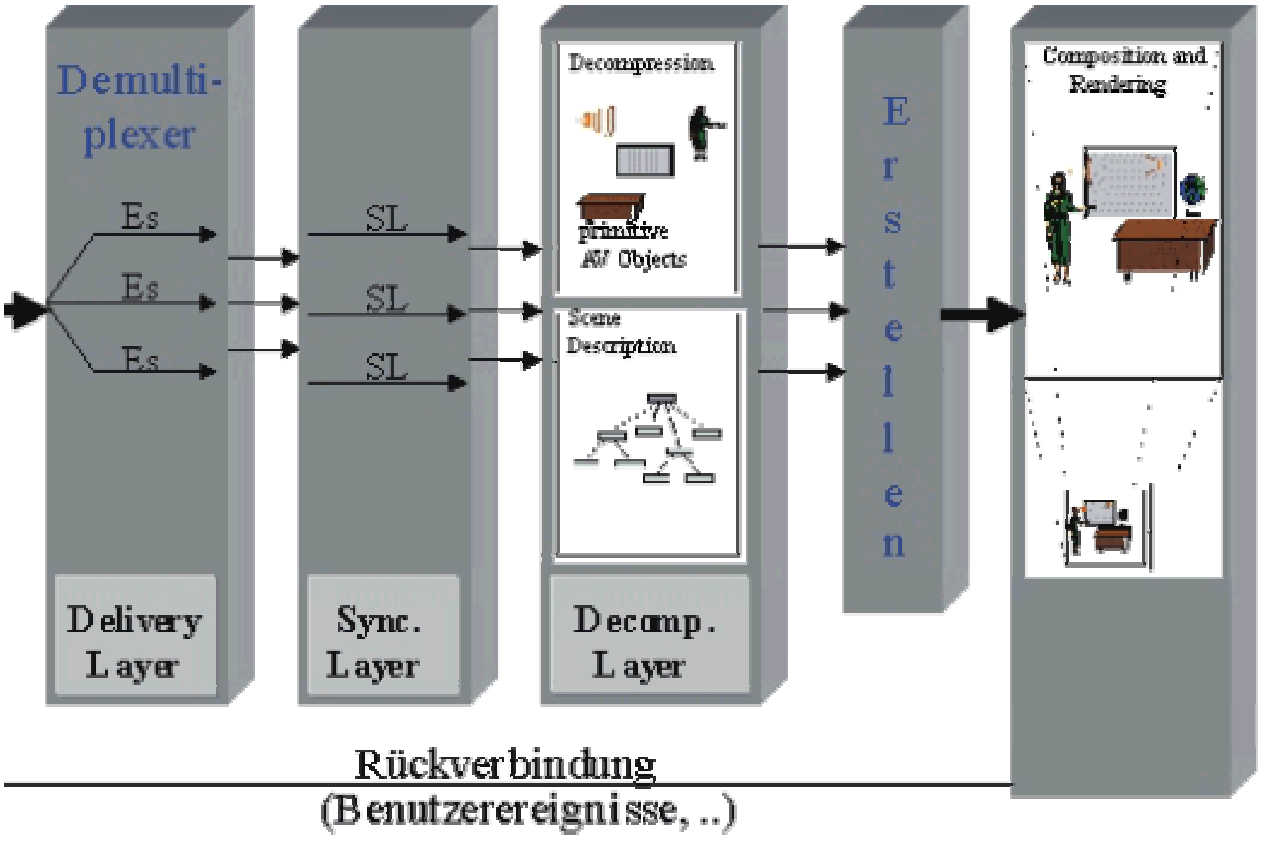


Abbildung 5: Schichtenmodell (Bild aus [6])

Audio

In diesem Kapitel wird ein Überblick über die differenzierten Möglichkeiten gegeben, natürliches und synthetisches Audiomaterial in MPEG-4 zu verwenden.

Natürliches Audiomaterial

Natürliches Audiomaterial setzt sich im wesentlichen aus zwei Komponenten zusammen: Sprache und Musik. Da von beiden verschiedene Anforderungen an den Encoder ausgehen, werden sie im MPEG-4 Standard getrennt behandelt. So verfahre ich hier auch.

Natürliche Sprache

Die Sprachkodierung hat zwei Seiten, die zu beachten sind. Einerseits ist das menschliche Ohr auf Sprache sehr gut trainiert und erkennt Unterschiede, Betonungen und ähnliches sehr genau, geht also kritischer mit den Informationen um. Andererseits ist Sprache meist Mittel zur Kommunikation, und so verstehen wir auch Sprache, die stark verfremdet wurde, beispielsweise beim Telefonieren. Hier fällt auf, dass wir nicht alle Toninformationen der Sprache brauchen, um zu verstehen, Sprache also stärker komprimieren können. Genau das ist das Ziel der beiden Sprach-Codecs aus MPEG-4: *Harmonic Vector eXcitation Code* (HVXC) und *Code Excitat Linear Prediction* (CELP). Beide benutzen eine spezielle Form der Vektorquantisierung, die auf menschliche Sprache optimiert worden ist, um Sprache verständlich bei geringen Bitraten zu komprimieren.
HVXC kodiert mit 2 bis 6 KBit/s bei 8 kHz und CELP kodiert mit 6 bis 24 KBit/s bei 8 bis 16 kHz. Damit sind diese Modelle ideal für Internettelefonie oder Mobilfunk, aber nicht für Sprache in einem Hollywoodfilm. Man könnte selbstverständlich auch Musik damit kodieren, aber der geringe Frequenzbereich und die geringen Bitraten wirkein einer akzeptablen Qualität entgegen.

Natürliche Musik

Wenn es um Codecs geht, die natürliche Musik kodieren, dann müssen sie im direkten Vergleich mit dem zehn Jahre alten MP3 bestehen, der ja Quasi-Standard in diesem Bereich ist, und ebenfalls im MPEG-$ Standard enthalten ist.
Im MPEG-4 Standard sind dafür *MPEG-2 Advanced Audio Coding* (MPEG2 AAC) vom Fraunhofer Institut und *Transform-domain Weighted Interleave Vector Quantisation* (Twin VQ) von den NTT Human Interface Laboratories in Japan vorgesehen.
Zu beiden lässt sich sagen, dass sie mit 96 KBit/s mit zeitgemäßen Encodern klingen wie MP3 mit 128 KBit/s und auf dem Prinzip der Vektorquantisierung beruhen, die Komprimierung jedoch aufwendiger als bei MP3 ist.

Synthetisches Audiomaterial

Auch Synthetisches Audiomaterial lässt sich in Sprache und Musik unterteilen, da auch hier die Ansätze völlig verschieden sind.

Synthetische Sprache

In MPEG-4 ist ein sogenanntes Text To Speech (TTS) System integriert, also eine generierte Computerstimme, die Texte vorlesen kann. Damit es sich nicht so hölzern anhört wie vor einigen Jahren auf Computern üblich, besteht die Möglichkeit eigene Phoneme, also Bestandteile der Sprache, die aus einigen Buchstaben bestehen, einzubinden. Weiterhin ist es möglich Parameter zu übertragen, die auf animierten Gesichtermodellen für Lippensynchronität sorgen. Solche Gesichtsmodelle sind in MPEG-4 auch implementiert und kommen im nächsten Kapitel unter anderem zur Sprache.
Durch solch ein TTS System kann auch mit wenig fließenden Daten ein sehr lebendiger Eindruck eines virtuellen Gegenübers entstehen. Im Internet könnten solche Systeme bei der Bestellannahme oder im Kundenservice hilfreiche Dienste leisten.

Synthetische Musik ist schon seit vielen Jahren ein eigener Bereich in der Musikbranche. Ein Standard aus dieser Zeit wird ebenfalls unterstützt: das Musical Instrument Digital Interface (MIDI). Im Gegensatz zu Audiokompressionsverfahren wird bei MIDI nur angegeben welches Instrument wann welchen Ton spielt. Es werden also Informationen über Instrument, Note, Lautstärke, Dauer und ähnliches gespeichert. Daraus resultiert die geringe Dateigröße und der einschlägige Sound, denn der Ton wird aus diesen Informationen generiert, und je nach den gespeicherten Samples auf der Soundkarte hört sich das dann mehr oder weniger gut an. Dieses Verfahren ist dadurch recht unflexibel, weil nur bestehende Instrumente benutzt werden können, und kann nur für Musik eingesetzt werden.

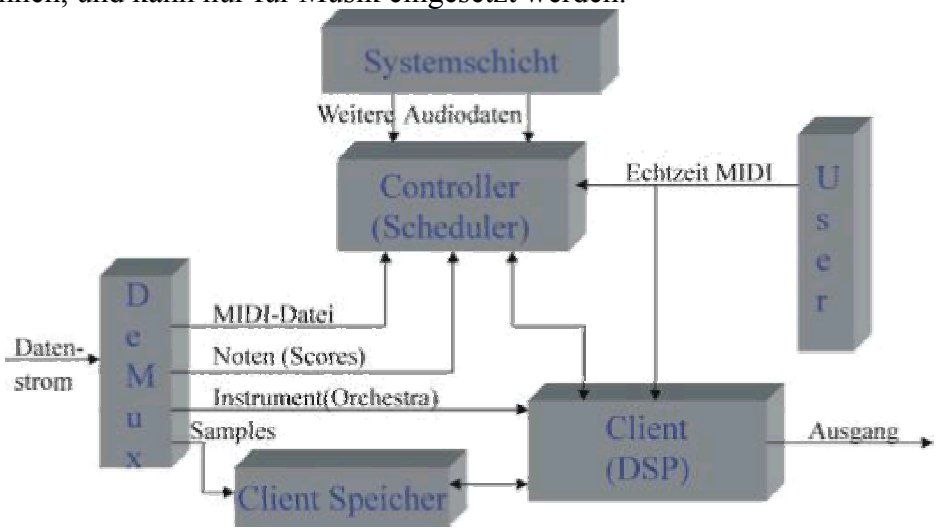


Abbildung 6: SAOL Decoder (Bild aus [6])

Eine Verbesserung von MIDI ist die Structured Audio Orchestra Language (SAOL), vom MIT Media Laboratory entwickelt. Die Vorteile von MIDI bleiben hier erhalten und die Nachteile fallen weg. Es werden immer noch Töne durch einige, wenige Parameter definiert, es können aber zusätzlich noch eigene Instrumente oder Samples mit übertragen und abgespielt werden. Der Decoder für dieses flexiblere Format ist in Abbildung 6 zu sehen.
Der Datenstrom wird zuerst im Demultiplexer in seine Elemente zerlegt, diese werden dann vom Scheduler mit Zeitstempeln versehen und angeordnet. Neue Instrumente oder Samples können auch geladen werden. All diese Informationen werden nun dem Digitalen Soundprozessor zur Verfügung gestellt, der daraus die Musik berechnet.

Video

Wie schon bei der Audiocodierung kann auch bei der Videocodierung zwischen natürlichen und synthetischen Objekten unterschieden werden.

Natürliches Videomaterial

Für die Kodierung von natürlichem Videomaterial existiert in MPEG-4 ein Modell das aus drei Komponenten besteht: *Shape Coding*, *Motion Coding* und *Texture Coding*. Wie in Abbildung 7 zu sehen werden alle drei Verfahren separat ausgeführt und dann vermultiplext.

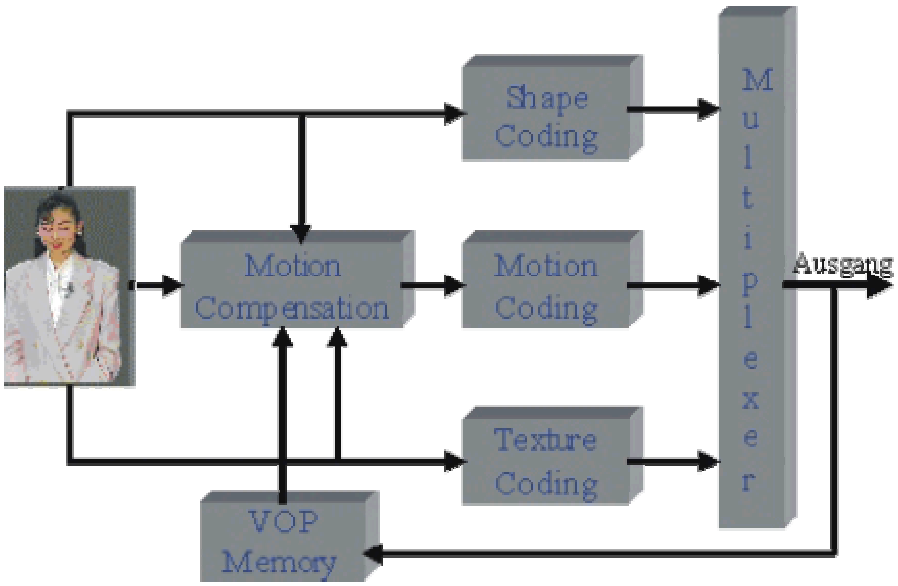


Abbildung 7: Kodierung von natürlichem Videomaterial (Bild aus [6])

Shape Coding

MPEG-4 ist der erste Videostandard der auch *Shape Coding* zulässt. Das bedeutet, dass die Videoobjekte nicht rechteckig sein müssen, sondern die natürliche Form haben, die die Objekte im Video repräsentieren. Es ist bis heute schwierig Algorithmen zu finden, die in einem Video Objekte wie etwa Personen herausfinden können, aber die Umsetzung dieser Objekte ist schon in MPEG-4 implementiert. Es gibt zwei Möglichkeiten, um zu speichern wie die Objekte definiert werden.

Einmal wird in einer Matrix (Bitmap) binär gespeichert, ob ein Pixel zum Objekt gehört oder nicht. Zur effektiven Speicherung wird diese in 16x16 Blöcke geteilt, die verlustfrei oder -behaftet komprimiert werden können.

Die andere Möglichkeit ist ähnlich, nur können 256 verschiedene Werte abgespeichert werden, um ebenso viele verschiedene Transparenzstufen zu bilden (*Grey Scale Shape Coding*).

Es ist klar, dass eine Matrix mit nur Nullen und Einsen sehr effizient gespeichert werden kann. Da es beim *Grey Scale Shape Coding* 256 verschiedene Werte gibt, wird noch eine Diskrete Cosinus Transformation (DCT) angewendet, um die Datenmenge zu verringern. Auf die DCT werde ich in diesem Kapitel nicht weiter eingehen, da andere Kapitel dieses Thema schon erschöpfend behandeln. Abbildung 8 zeigt ein Beispiel zum *Shape Coding*.

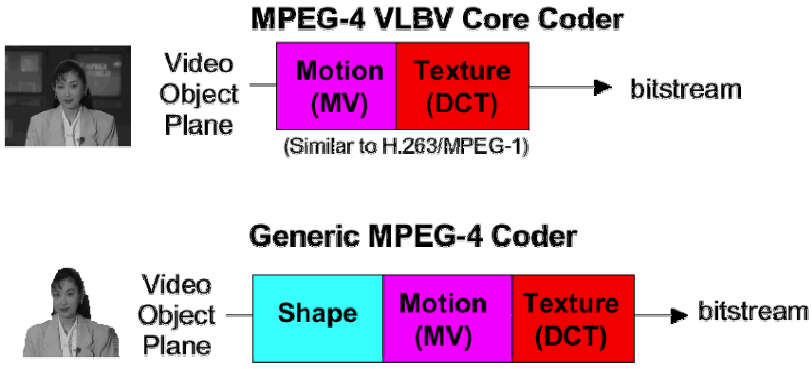


Abbildung 8: Shape Coding (Bild aus [1])

Motion Coding

Beim *Motion Coding* ist die Idee, die zeitliche Redundanz innerhalb eines Videos zu verringern. Die Erfahrung zeigt, dass in Videos häufig Bilder den Vorgängern und Nachfolgern sehr ähnlich sind. Wenn also in einer Sekunde auf der Leinwand nichts passiert, das Video aber 25 Bilder pro Sekunde anzeigt, dann wurden 24 Bilder zuviel abgespeichert.
Beim *Motion Coding* von MPEG-4 werden die Videoobjekte in Abhängigkeit von den zeitlichen Nachfolgern und Vorgängern kodiert. Dabei gibt es, genau wie bei MPEG-2 drei verschiedene Bildtypen, die auch Abbildung 9 zeigt:
I-Videoobjekte (*Intracoded*) werden völlig unabhängig von anderen kodiert.
P-Videoobjekte (*Predicted*) werden nur in Abhängigkeit von Vorgängern kodiert.
B-Videoobjekte (*Bidirectional Interpolated*) können in Abhängigkeit von Vorgängern und Nachfolgern kodiert werden.

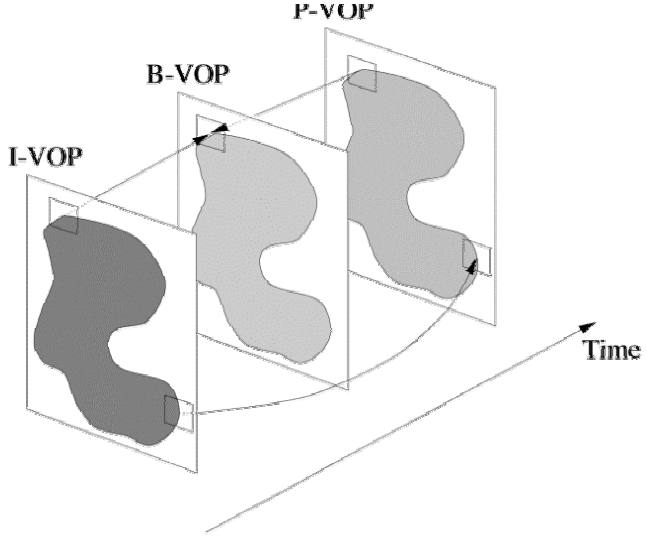


Abbildung 9: Motion Coding (Bild aus [6])

Dabei wird das Bild in Blöcke aufgeteilt und jeder Block wird in den möglichen abhängigen Objekten gesucht. Wenn er gefunden wird, wird auf diesen Block verwiesen anstatt ihn nochmals zu kodieren, wenn nicht dann wird er intrakodiert. Dieses Verfahren wird in der Seminararbeit „MPEG-1/2“ ebenfalls beschrieben.

Texture Coding

Auch beim Texture Coding gibt es kaum Unterschiede zu MPEG-2. Bei bewegten Texturen wird eine 8x8 blockbasierte DCT mit anschließender Quantisierung benutzt um die Daten zu reduzieren. Neu ist eine Möglichkeit der DCT-Koeffizienten Vorhersage, bei der der Bildgradient entscheidet ob Zeilen oder Spalten vorhergesagt werden können, je nachdem wie das jeweilige Bild gerade aussieht. Der Artikel [7] befasst sich ausführlicher mit dem Thema.
Statische Texturen werden mit einer Wavelet Transformation Komprimiert, und lassen sich dann auf Polygone abbilden, zum Beispiel um dreidimensionale Modelle realistisch darzustellen. All das ist Vergleichbar mit der Komprimierung bei JPEGs, sieht aber Qualitativ besser aus

Sprite Coding

Eine Möglichkeit Bandbreite bei Videos zu sparen, in denen sich wenig bewegt, ist das *Sprite Coding*. Dabei wird ein Hintergrundbild einmalig übertragen, und ansonsten nur Videoobjekte die sich darauf bewegen. Dem Vorteil der guten Komprimierung stehen die tote Darstellung des Hintergrundes entgegen, die das Gehirn sicher als unwirklich empfindet. Das Beispiel aus Abbildung 10 zeigt ein Tennismatch, bei dem nur der Spieler sich bewegt, und der Hintergrund nur einmalig übertragen wird. Dies wirkt sicher unwirklich und hat bei Nachrichtensendungen oder der Wettervorhersage eine bessere Wirkung.

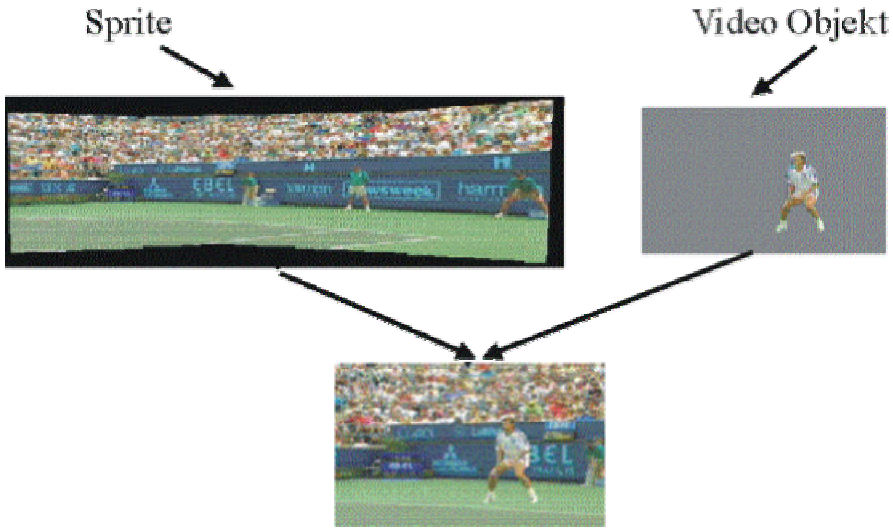


Abbildung 10: Sprite Coding (Bild aus [1])

Synthetisches Videomaterial

Bei der Kodierung von synthetischem Videomaterial gibt es einige erwähnenswerte Besonderheiten, die außer VRML in MPEG-4 implementiert sind. VRML habe ich als Darstellungssprache von 3D Modellen schon erwähnt und gehe nicht weiter darauf ein

Facial and Body Animation

Bei *Facial and Body Animation* geht es um Tools, die es möglich machen, Gesichter beziehungsweise Körper zu definieren, ohne die Bandbreite zu belasten. Bei der *Facial Animation* gibt es beispielsweise 84 Punkte, sogenannte *Facial Definition Parameters* (FDP), die man in Abhängigkeit voneinander definiert um ein neutrales Gesicht zu beschreiben.

Einige von diesen Punkten, zum Beispiel an Lippen und Augen lassen sich dann animieren um Bewegungen im Gesicht zu erzeugen. Ein Beispiel dafür zeigt Abbildung 11.

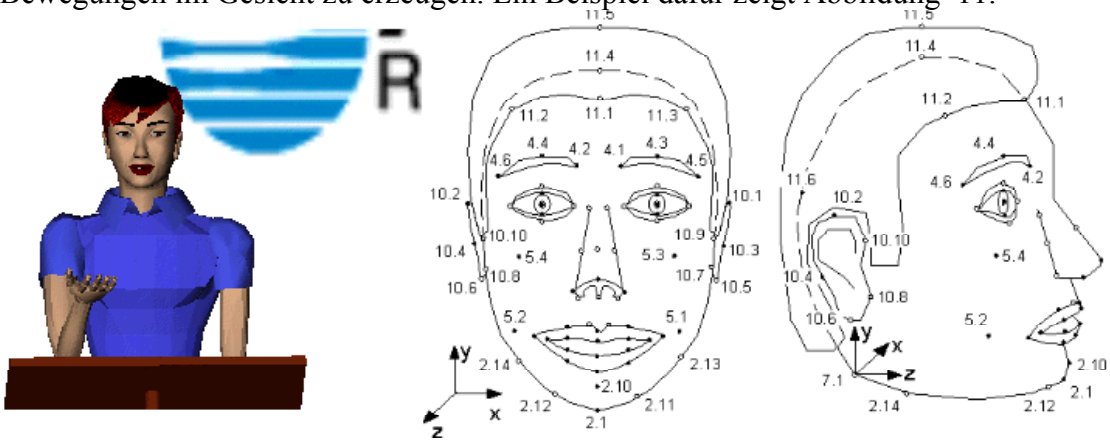


Abbildung 10: Facial Animation (Bild aus [6])

In Verbindung mit dem *Text To Speech System* kann dann mit wenig Bandbreite ein virtuelles Gegenüber erschaffen werden. Die Animation von Körpern ist analog zu verstehen.

2D-Meshes

Unter einem 2D-Mesh versteht man eine Triangulierung des Bildes, also eine Aufteilung in Dreiecke. Auf diese Dreiecke können nun wie auf Polygone aus der 3D Darstellung Texturen gelegt werden, anstatt das ganze Objekt mit einer Textur zu belegen. So gelingt es aber durch Manipulation an den Knotenpunkten das ganze Objekt scheinbar zu animieren. Abbildung 12 zeigt diese Aufteilung auf einem Fisch, der beispielsweise so animiert werden könnte, als ob er schwimmt.

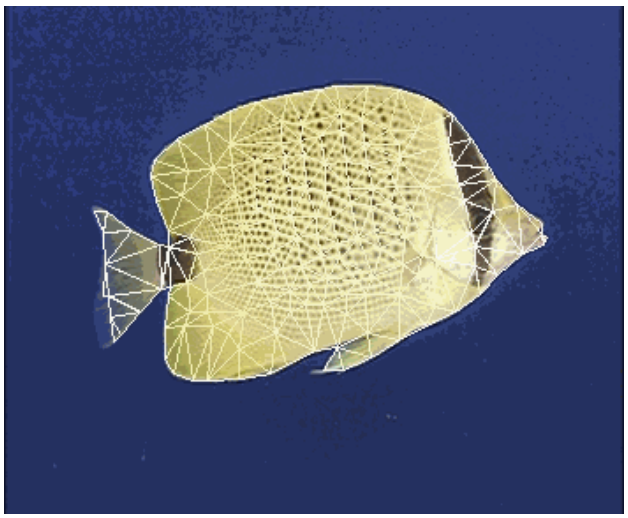


Abbildung 11: 2D-Meshes (Bild aus [1])

Extras

In diesem Kapitel nenne ich einige erwähnenswerte Teile von MPEG-4, die in keines der anderen Kapitel gepasst haben.

Profile

Da MPEG-4 so viele Möglichkeiten bietet Videos zu kodieren und es des weiteren viele Möglichkeiten gibt, wie man dieses Video konsumiert (viele verschieden leistungsfähige Endgeräte), gibt es in MPEG-4 wie schon in MPEG-2 die Profile. Profile sind Stufen, die den Encoder in der Anzahl der Tools, der Bitrate, Bildgröße oder beispielsweise der Anzahl der darstellbaren Objekte beschränken, um das Endgerät nicht zu überlasten. Weiterhin haben die Profile (Simple, Advanced, etc.) noch Level (0, 1, 2, etc.), die andere Fähigkeiten beschränken. So kann der Encoder auf Level 0 nur 99 Makroblöcke (ein Makroblock entspricht 16x16 Pixel) darstellen. So kann dann beim Abspielen auf einem bestimmten Gerät gewährleistet werden, dass Videos mit bestimmten Profilen dargestellt werden können.

Sicherheit

Um die Urheberrechte zu wahren und allzu freizügigem Kopieren Einhalt zu gebieten, wurde das digitale Wasserzeichen eingeführt. Damit kann zwar noch kopiert werden, allerdings erst, wenn dieses herausgerechnet wurde, was immer einen Qualitätsverlust zur Folge hat.

Zusammenfassung

MPEG-4 ist der erste Multimediastandard, der natürliche und synthetische Ursprungsdaten in gleichem Maße gut komprimiert, der inhaltsorientiert arbeitet und nicht nur blockbasiert. Durch die gute Skalierbarkeit ist er für viele Nutzer mit verschiedenen Endgeräten interessant und auch für den Home-Entertainment Bereich verwendbar, da er ein ISO-Standard ist. Dies ist schon MPEG-1 mit CD-i, MPEG-2 mit DVD und MP3 gelungen. Mit DIVX gibt es schon einen MPEG-4 ähnlichen Codec, der unter Computernutzern sehr verbreitet ist. Wenn große Hersteller auf den MPEG-4 Zug aufspringen, ist er praktisch nicht aufzuhalten.
Zur Zeit wird noch an MPEG-7 und MPEG-21 gearbeitet, Standards die noch mehr auf Multimedia setzten, aber MPEG-4 nur ergänzen sollen, da es nicht mehr um Kompression, sondern um Beschreiben, Wiederfinden, Zugreifen und Vertreiben geht. Das liegt jedoch noch in ferner Zukunft und wird sicher noch überarbeitet werden.

Abkürzungen

AAC	Advanced Audio Coding
BIFS	Binary Format for Scene Description
CELP	Code Excitat Linear Prediction
CSS	Content Scrambling System
DCT	Diskrete Cosinus Transformation
ES	Elementary Stream
FDP	Facial Definition Parameter
HVXC	Harmonic Vector eXcitation Code
ISO	International Standardisation Organisation
MIDI	Musical Instrument Digital Interface
MPEG 1	Moving Picture Expert Group Version 1
MPEG 2	Moving Picture Expert Group Version 2
MPEG 4	Moving Picture Expert Group Version 4
SAOL	Structured Audio Orchestra Language
TTS	Text To Speech
Twin VQ	Transform-domain Weighted Interleave Vector Quantisation
VRML	Virtual Reality Modeling Language

Literatur

[1] R. Koenen, International Organisation for Standardisation: Overview of the MPEG-4 Standard (Mar.2001) (http://www.m4if.org/resources/Overview.pdf?PHPSESSID=c0a331e42a760490049a429a4e14b53d)

[2] Jürgen Herre, Fraunhofer Institute for Integrated Circuits (IIS): MPEG- 4 General Audio Coding (http://www.tnt.uni-hannover.de/project/mpeg/audio/general/vancouver-general-audio.pdf)

[3] Julien Signès, Yuval Fisher, Alexandros Eleftheriadis: MPEG-4's Binary Format for Scene Description (http://leonardo.telecomitalialab.com/icjfiles/mpeg-4_si/5-BIFS_paper/5-BIFS_paper.htm)

[4] R. Koenen, International Organisation for Standardisation: From MPEG-1 to MPEG-21: Creating an Interoperable Multimedia Infrastructure (Dec. 2001) (http://mpeg.telecomitalialab.com/documents/from_mpeg-1_to_mpeg-21.htm)

[5] Peter Haighton: MPEG-4 Users Frequently Asked Questions (http://www.m4if.org/resources)

[6] Michael Repplinger: Überblick über den neuen MPEG-4 Standard (Jan 2000) (http://graphics.cs.uni-sb.de/Courses/ws9900/cg-seminar/Ausarbeitung/Michael.Repplinger/)

[7] John Watkinson: MPEG-4 secrets (May 2002) (http://broadcastengineering.com/ar/broadcasting_mpeg_secrets/)

Fraktale Kompression

von

Alexandar Rakovski

Einleitung

Natürliche Strukturen wie die Wolken am Himmel, Gebirgsketten, Küsten, Felsen, organische Gewebe, Pflanzen oder mineralische Oberflächen sind scheinbar unüberschaubar komplex, besitzen tatsächlich aber eine geometrische Regelmässigkeit. Auch Prozesse wie der Wachstum von Zellen oder Kristalle können als Ergebniss eines Iterationsfortschrittes dargestellt werden. Aufgrund dieser Erkentniss sind in den letzten zwanzig Jahren verschiedene Anwendungen der fraktalen Geometrie entstanden. Auf der Abbildung ist eine völlig künstliche Welt zu sehen. Der Autor Przemyslaw Prusinkiewicz (Computergrafiker

an der Universität Calgary) musste seine Kollegen überzeugen, dass er nicht geschummelt hat und er kein echtes Foto präsentiert. Heutzutage finden regelmässig Wettbewerbe statt, in dennen Bilder gezeigt werden, die durch Fraktale generiert wurden. Die wichtigste Aufgabe dabei ist es die passende Funktion zu finden, die die eine oder andere Pflanze, Felsen, Küste u.a. generieren. Die berechtigte Frage, die sich dabei stellt ist, ob die Funktionen automatisch bestimmt werden können, z.B. aus einem fertigen Bild. Diese Frage wird das inverse Problem der fraktalen Kompression genannt und ist Hauptthema unseres Artikels. Wir fangen mit den mathematischen Grundlagen an.

Mathematische Grundlagen

Unter einem Bild werden wir die Abbildung einer Trägermenge auf die Menge der Farben verstehen:

Definition 1 *Ein Bild ist ein Tripel* $(T; (F; d); b)$*, wobei*

- T *eine beliebige Menge (die Trägermenge),*
- $(F; d)$ *ein kompakter metrischer Raum (die Menge der Farben) und*
- b *eine Funktion von* T *nach* F *ist.*

Falls T *und* F *aus dem Zusammenhang klar sind, sprechen wir oft auch nur von einem Bild* b*.*

Definition 2 *Sei* T *eine Tragermenge und* $(F; d)$ *ein Farbraum. Dann ist*

$$\mathcal{B}(T, F) := \{b | b : T \to F\}$$

die Menge der Bilder über T *(mit Farbraum* F*). Falls* T *und* F *klar (oder unwichtig) sind, schreiben wir auch kurz* $\mathcal{B}$*.*

Beispiele für Träger:

- Diskreter Träger: $T = \{1, \ldots, m\} \times \{1, \ldots, n\}$ - ein Bild $m \times n$-gross $(n, m \in \mathbb{N})$,
- Diskreter Träger, 1-dimensional: $T = \{1, \ldots, n\}$, $(n \in \mathbb{N})$,
- Kontinuierlicher Träger: $T = [0, x] \times [0, y] \subset \mathbb{R}^2$

Beispiele für Farbmengen (F, d)**:**

- Schwarz-Weiss-Bilder: $F = \{0, 1\}$,
- Graustufenbilder, kontinuierlich: $F = [0, 1]$, 0 entspricht schwarz, 1 bedeutet weiss,
- Graustufenbilder, diskret: $F = \{0, 1, \ldots, 255\}$, 0 entspricht schwarz, 255 - weiss,
- Farbbilder, diskret: $F = \{0, 1, \ldots, 255\}^3$ - RGB-Modell.

Man kann in den ersten drei Fällen als Metrik d die absolute Differenz der Werte wählen. Bei den Farbbildern kann man z.B. den euklidischen Abstand der Punkte im $\mathbb{R}^3$ nehmen.

Um Bilder vergleichen zu können müssen wir eine Metrik auf $\mathcal{B}(T,F)$ definieren. Hier bieten sich verschiedene Möglichkeiten an, die von Fall zu Fall besser oder schlechter geignet sind:

Definition 3 ***Supremummetrik:*** *Seien b_1, b_2 zwei Bilder mit Träger T und Farbraum (F,d). Dann setzen wir:*

$$L_\infty(b_1,b_2) := \sup_{x\in T} d(b_1(x),b_2(x)).$$

$(\mathcal{B}(T,F),L_\infty)$ ist ein vollständiger metrischer Raum.

Definition 4 ***L_2-Metrik,*** *$\|\cdot\|$: Sei $T=\{t_1,\ldots,t_n\}$ und $b_1,\ b_2 \in \mathcal{B}(T,(F,d))$. Dann setzen wir:*

$$L_2(b_1,b_2) := \sqrt{\sum_{i=1}^{n} d(b_1(t_i),b_2(t_i))^2}.$$

Wir schreiben $\|b_1-b_2\|$ für $L_2(b_1,b_2)$.

Definition 5 ***Hausdorff Metrik für Schwarz-Weiss-Bilder:***

$$L_H(b_1,b_2) := \sup_{x,y}\{\|x-y\|_{\mathbb{R}_2} : x,y\in T; b_1(x)=1; b_2(y)=1\}.$$

Jedes Bild, das durch Fraktale generiert wird, ist Ergebnis von rekurisver Anwendung einer Funktion auf einem Startbild. Dabei ist das Startbild unwichtig. Das Ergebniss ist immer das Gleiche. Die letzte Tatsache ist als der Fixpunktsatz bekannt:

Definition 6 ***(Kontraktive Abbildung):*** *Eine Abbildung $f:\mathcal{B}\to\mathcal{B}$ heisst kontraktiv bzgl. einer Metrik L auf $\mathcal{B}$, wenn es ein $\delta<1$ gibt, so dass*

$$\forall b_1,b_2\in\mathcal{B}:\ L(f(b_1),f(b_2)) \leqslant \delta\cdot L(b_1,b_2)$$

gilt. δ heisst Kontraktionsfaktor von f.

Satz 1 ***(Banachsche Fixpunktsatz)*** *Sei $f:\mathcal{B}\to\mathcal{B}$ kontraktiv bzgl., L. Dann besitzt f einen eindeutig bestimmten Fixpunkt $\Omega_f\in\mathcal{B}$ mit $f(\Omega_f)=\Omega_f$. Ausserdem konvergiert für jedes $b\in\mathcal{B}$ die Folge $b,\ f(b),\ f(f(b)),\ f(f(f(b))),\ldots$ gegen Ω_f.*

Satz 2 *Wenn $\psi_1(b),\ldots,\psi_n(b)$ kontraktive Abbildungen bzgl. L_H sind, dann ist auch*

$$\Psi(b) := \bigcup_{i=1}^{n}\psi_i(b)$$

kontraktiv bzgl. L_H.

Beispiele für einen Fixpunkt:

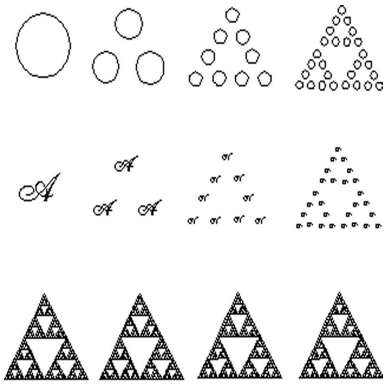
Sirpinski-Dreieck: Sei $A = 0.5 \cdot E \in \mathbb{R}^{2\times 2}$,

$$\psi_1(b) = Ax + o_1,$$

$$\psi_2(b) = Ax + o_2,$$

$$\psi_3(b) = Ax + o_3,$$

mit $o_1 = (0,0)^T,\ o_2 = (100,0)^T,\ o_3 = (30,30)^T$. Das Sirpinksi-Dreieck ist die Lösung der Gleichung: $\Psi(A) = A$:



Barnsley-Farn : $\psi_i(b) = A_i x + o_i,\ i = 1,\ldots,4,$

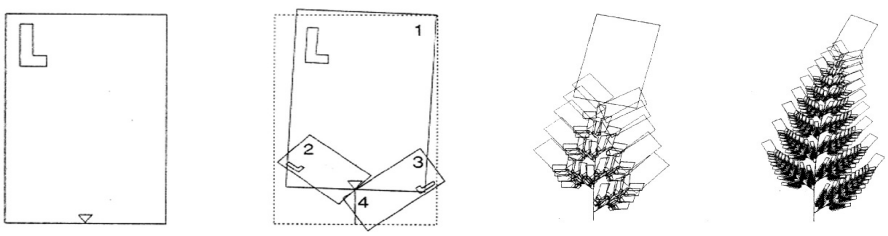
$$A_1 = \begin{pmatrix} 0.000 & 0.000 \\ 0.000 & 0.160 \end{pmatrix},\ o_1 = (0.000,\ 0.000)^T,$$

$$A_2 = \begin{pmatrix} 0.197 & -0.026 \\ 0.226 & 0.197 \end{pmatrix},\ o_2 = (0.000,\ 1.600)^T,$$

$$A_3 = \begin{pmatrix} -0.155 & 0.283 \\ 0.260 & 0.237 \end{pmatrix},\ o_3 = (0.000,\ 0.440)^T,$$

$$A_4 = \begin{pmatrix} 0.849 & 0.037 \\ -0.037 & 0.849 \end{pmatrix},\ o_4 = (0.000,\ 1.600)^T.$$

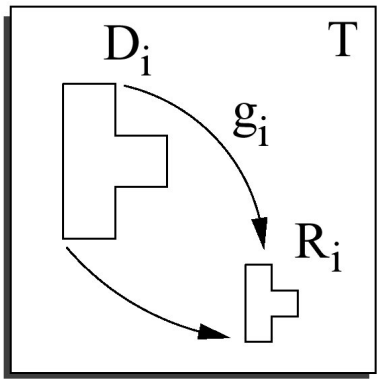
Um das Bild aufzubauen, wird die Abbildungsmaschine mit einem beliebigen Anfangsbild (hier ein Rechteck) iterriert:



Grundideen der fraktalen Kompression. Collage Coding

Die Grundidee der fraktalen Kompression besteht darin eine kontraktive Funktion f zu finden, für die $L(\Omega_f, b)$ möglichst klein ist. Diese Funktion f wird dann gespeichert. Beim Dekodieren wendet man f mehrfach auf ein beliebiges Ausgansbild an. Die so gewonnene Annährung von Ω_f wird dann als Ergebnis ausgegeben. Leider ist das Berechnen der optimalen kontraktiven Funktion f eines Bildes ein NP-vollständiges Problem. Man lässt deshalb eine gewisse Fehlertoleranz zu und gibt sich mit einer Aproximation zufrieden.

Zuerst wird der Träger T in disjunkte Teilmengen $T = R_1 \bigcup R_2 \bigcup \ldots \bigcup R_k$ zerlegt. Die R_i heissen Bereiche. Jeder Menge R_i wird eine Menge $D_i \subset T$, genannt Domain, zugeordnet. Zunächst wird die Domain mittels einer geometrischen Abbildung auf die



Grösse der Domain gebracht. Diese surjektive Abbildung von D_i nach R_i nennen wir g_i. Die Farbwerte der so auf Bereichsgrösse gebrachten Domain werden dann mittels einer kontraktiven affin linearen Abbildung $x \mapsto s_i \cdot x + o_i$ auf die Farbwerte der Bereiche abgebildet. Die Parameter hierbei sind der Skalierungsfaktor s_i, der in $[-1, 1]$ und der Offsetparameter o_i. Falls g_i injektiv sind, dann definieren wir:

$$f(b)(x) := s_i \cdot b(g_i^{-1}(x)) + o_i, \quad \text{wobei } i \text{ mit } x \in R_i \text{ ist.}$$

Sind die g_i nicht injektiv und T endlich:

$$f(b)(x) := s_i \cdot \left(\frac{1}{|g_i^{-1}(x)|} \sum_{y \in g_i^{-1}(x)} b(y) \right) + o_i, \quad \text{wobei } i \text{ mit } x \in R_i \text{ ist.}$$

Man nennt solche f gelegentlich **PIFS**: Partitioniertes Iterriertes Funktionen System.

Ein wichtiger Schritt ist es, zu einem Bild b eine Kontraktion f zu finden, die den Abstand $\|b - \Omega_f\|$ minimiert. Sehr hilfreich ist dabei der Collagen-Satz:

Satz 3. *Sei $f : \mathcal{B} \to \mathcal{B}$ kontraktiv bzgl. L_2 mit Kontraktionsfaktor δ und habe den Fixpunkt Ω_f. Dann gilt für alle $b \in \mathcal{B}$:*

$$\|b - \Omega_f\| \leqslant \frac{1}{1-\delta} \cdot \|b - f(b)\|.$$

Eine gute Idee ist also solche f zu suchen, die $\|b - f(b)\|$ minimieren. Es reicht den Ausdruck $\|(b|R_i) - f(b|D_i)\|$ zu minimieren. Man findet die besten Domains durch Durchprobieren aller (normalerweise polynomiell vieler) Möglichkeiten. $\|(b|R_i) - f(b|D_i)\|^2$ ist quadratische Form in s_i und o_i und $\min \|(b|R_i) - f(b|D_i)\|^2$ kann deshalb leicht berechnet werden.

Partitionierung

Wie soll das Bild in Bereiche R_i partitioniert werden? Hier bieten sich verschiedene Alternativen an (s. Abbidungen 1 bis 5).

Abbildung 1: **Uniform:** Übliche Bereichsgrössen 4×4 oder 8×8 Pixel

Abbildung 2: **Quadtree:** Man beginnt mit 32×32 und teilt in 4 gleich grosse Quadrate dort wo der Fehler grösser ist als ein Toleranzwert

Abbildung 3: **HV-Partitionierung (Horizontal Vertikal Partitionierung)**

Wir werden uns die adaptive Patitionierung genauer Anschauen. Man geht wie folgt vor:

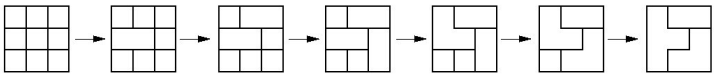
- Unterteile das Bild in atomare Blöcke (4×4) als Bereiche.
- Für jeden der Bereiche R_i wird eine Liste von guten Domains berechnet $D_{1,i}, \ldots, D_{d,i}$.

Abbildung 4: **Partitionierung durch Dreiecke**

Abbildung 5: **Adaptive Partitionierung:** Die Bereiche sind zusammenhängende Mengen von kleinen Bildblöcken 4×4 oder 8×8

- Solange die gewünschte Kompressionsrate noch nicht erreicht ist bzw. ein vorgegebener Collagefehler noch nicht überschritten wurde, wiederholen wir folgendes:

  (a) Wir wählen zwei benachbarte Bereiche der Partitionierung aus.

  (b) Dann erzeugen wir eine neue Konfiguration, in der diese beiden Bereiche zu einem gemeinsamen Bereich vereinigt wurden. Der Rest der Konfiguration sieht so aus wie bei der alten Konfiguration (s. Abbildung 6).

  (c) Die d Domains für die neuen Bereiche bestimmen wir dann so: Von den Vorgängern der Bereiche haben wir 2d Domains geerbt. Diese werden analog der zugehörigen Bereiche erweitert, so dass sie doppelt so groß wie die neuen Bereiche sind. Die so erhaltenen Domains werden dann mit der neuen Bereiche verglichen, und die besten d gelangen in die neue Konfiguration (s. Abbildung 7).

Am Anfang wird das Bild in lauter atomare Blöcke unterteilt, und zu jeder dieser Bereiche muss eine Liste von d guten Domains gefunden werden. Wenn die atomaren Blöcke z.B. die Grösse 4×4 haben, so sind die möglichen Domains die nicht überlappenden Blöcke der Grösse. Eine Möglichkeit wäre der **Full search:** Wir vergleichen jeden Bereich mit jeder möglichen Domain, und nehmen dann für jeden Bereich die d besten Do-mains. Das liefert sicherlich die beste Bildqualität, dauert aber andererseits durch die vielen Vergleiche

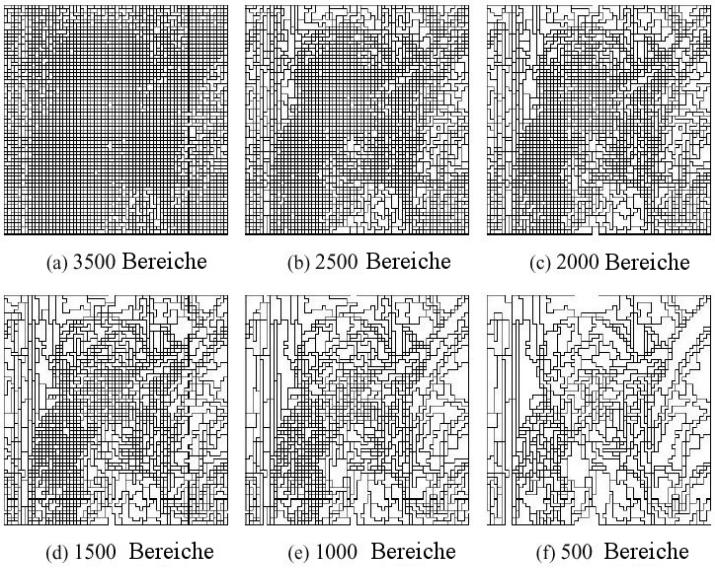


Abbildung 6: Vereinigung von Bereichen

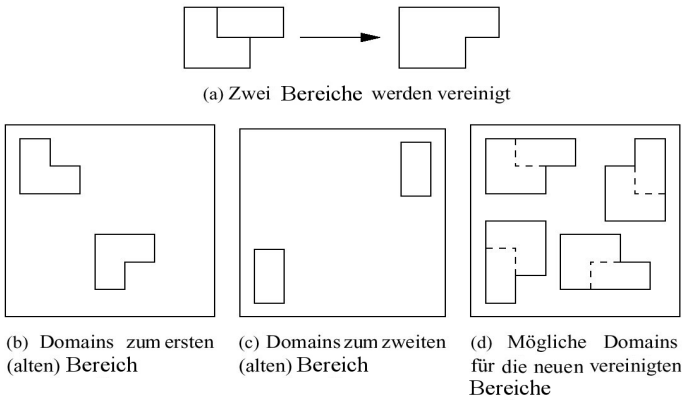


Abbildung 7: Vereinigung von Bereichen

auch sehr lange. Viel besser ist es die **nächster Nachbar Suche** zu verwenden. Zunächst normalisiert man Bereichs- und Domainvektoren x mittels:

$$x \mapsto \frac{x - \langle x, e\rangle e}{\|x - \langle x, e\rangle e\|}, \text{ wobei } e = \frac{1}{\sqrt{k}}(1, 1, \ldots, 1)^T \in \mathbb{R}^k.$$

Die beste Domain findet man, indem man zu einem Bereichsvektor den nächstliegenden Domainvektor sucht.

Transformationen. Kodierung der Farben und der Transformationen

Als Transformationen lässt man die üblichen affinen Abbildungen zu. Das heißt, zu jedem Bereich wird eine Domain gesucht, die unter einer kontraktiven linearen Abbildung die Bereiche möglichst gut approximiert. Zusätzlich erlaubt man dabei die acht Symmetrieabbildungen (s. Abbildung 8).

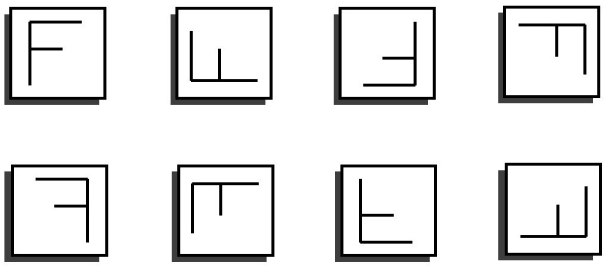


Abbildung 8: Die acht Symmetrieabbildungen

Für jeden Bereich R_i muss die Position der Domain D_i und die Parameter der Abbildung (s_i, o_i und die Art der Symmetrieabbildung) abgespeichert werden. Die Position wird einfach durch die x- und y-Koordinaten kodiert. Die Art der Symmetrieabbildung wird durch drei Bits kodiert. s wird linear quantisiert, der Wert also zur nächsten Zahl aus: $\{k/2^4 : k = -2^4, \ldots, 2^4\}$ gerundet:

$$\hat{s} = \frac{\lfloor 2^4 \cdot (s+1)\rfloor}{2^4} - 1.$$

Anschliessend wird o wie folgt quantisiert:

$$\hat{o} = \begin{cases} \frac{255(1+\hat{s})}{2^7-1}\lfloor \frac{2^7-1}{255(1+\hat{s})}(o + 255\hat{s})\rfloor - 255\hat{s} & : \hat{s} > 0 \\ \frac{255(1+\hat{s})}{2^7-1}\lfloor \frac{2^7-1}{255(1+\hat{s})}\rfloor & : \hat{s} \leqslant 0 \end{cases}$$

Man verwendet für die Kodierung der Farben das YUV-Modell:

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0,299 & 0,587 & 0,114 \\ -0,148 & -0,289 & 0,437 \\ 0,615 & -0,515 & -0,100 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Nach dieser Transformation enthält die Y-Komponente des Bildes die Helligkeitsinformation, die U- und V-Komponenten enthalten die Farbinformation. Da das Auge Unterschiede in der Helligkeit stärker wahrnimmt als Farbunterschiede, ist das Ziel, vor allem die Y-Komponente gut zu kodieren. Die Y-Komponente wird wie ein Graustufenbild fraktal kodiert. Für jeden Bereich wird der Durchschnitt der U- und V-Komponenten auf diesem Bereich gespeichert. Anschliessend wird die Folge dieser Werte gzip-komprimiert.

Anhang: Vergleich zwischen JPEG und fraktaler Kompression

Abbildung 9: Die fraktale Kompression ermöglicht eine bessere Skalierung des Bildes. Beim Dekodieren nimmt man dazu ein grösseres Startbild.

Literatur

[SR96] Dietmar Saupe, Matthias Ruhl, *Evolutionary fractal image compression*, in Proceedings IEEE International Conference on Image Processing (ICIP), Lausanne, September 1996. http://theory.lcs.mit.edu/ ruhl/papers/1996-icip.pdf

[RH97] Matthias Ruhl and Hannes Hartenstein *Optimal Fractal Coding is NP-Hard*, Data Compression Conference (DCC '97) Snowbird, March 1997. http://theory.lcs.mit.edu/ ruhl/papers/1997-dcc.pdf

[RHS97] Matthias Ruhl, Hannes Hartenstein, Dietmar Saupe *Adaptive partitionings in fractal image compression*, in: Proceedings IEEE International Conference on Image Processing (ICIP), Santa Barbara, October 1997. http://theory.lcs.mit.edu/ ruhl/papers/1997-icip.pdf

[G97] Söhnke Gorenflo, *Fraktale Kompression*, http://www-lehre.informatik.uni-osnabrueck.de/mm/mm05/meinMaster/Vortrag.pdf

[R97] Matthias Ruhl, *Fraktale Bildkompression - Adaptive Partitionierungen und Komplexität*, *Masters Thesis in Mathematics*, http://theory.lcs.mit.edu/ ruhl/papers/math-thesis-short.pdf

[K00] Wolfram Kühnert, *Von der Mehrfach-Verkleinerungs-Kopier-Maschine zum Collage Theorem*, http://www.kuehnert.de/mg/ifs/ifs005.htm

[FG00] *Fraktale Geometrie*, http://www-user.tu-chemnitz.de/ maku/Proseminar/salexa/Fraktale.htm

[SW01] *Computer-Pflanzen* von Oliver Deussen und Bernd Lintermann *Spektrum der Wissenschaft, 2/2001*