Prüfer: Prof. Dr. Paul Levi

Betreuer: Dipl. Inf. Niels Mache

begonnen am: 4.5.97

beendet am: 4.11.97

CR-Klassifikation: I.4.2

Diplomarbeit Nr. 1678

Szenenadaptive, parallele Codierung von hochaufgelösten Videosequenzen unter Berücksichtigung der Wahrnehmungsphysiologie

Dirk Farin

Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D–70565 Stuttgart

Inhaltsverzeichnis

Kapitel 1

Einleitung

Unter der Motivation der besseren Qualität und des höheren Komforts bei der digitalen Übertragung von Video- und Audiodaten findet eine zunehmende Digitalisierung dieser Übertragungstechniken statt. Nach der Digitalisierung der Audioaufzeichnung durch die CD folgt nun die Digitalisierung der Videoaufzeichnung und -übertragung. Da für eine Videoübertragung viel höhere Datenraten als für Audioübertragungen erforderlich sind, ist es erforderlich, Kompressionsverfahren zur Reduktion der Datenrate ohne allzu große qualitative Verluste zu benutzen. Übertragungen mit HDTV-Auflösungen bis zu 1920×1152 Bildpunkte sind aufgrund der zu hohen benötigten Übertragungsbandbreite ökonomisch überhaupt nicht mit konventioneller Technik ohne Kompression machbar.

Während für die Echtzeitcodierung Hardwarelösungen entwickelt werden, gewinnen aber auch die reinen Softwareimplementierungen an Bedeutung, da mit heute verfügbaren CPUs eine Echtzeitcodierung in kleinen Mehrprozessorrechnern (ca. 4 CPUs) möglich erscheint. Die Codierung in Software hat gegenüber der Hardwareimplementierung den Vorteil, daß sie flexibler in der Auswahl der möglichen Auflösungen und Algorithmen sind. Besonders, wenn keine Echtzeitcodierung erforderlich ist, wie bei der Produktion von Videos zur Archivierung auf Speichermedien, können Softwareencoder ausgefeiltere Algorithmen benutzen, die eine höhere Bildqualität bei erhöhter Rechenzeit erlauben. Bei einem Anstieg der Prozessorleistungen können dann diese besseren Algorithmen auch für die Echtzeitcodierung eingesetzt werden.

Ziel dieser Arbeit war es, einen Software-Encoder für MPEG-2 Video zu schreiben, der für Mehrprozessorrechnern parallelisiert und szenenadaptiv ist. Letzteres heißt, daß er plötzliche Bildwechsel erkennt und durch die angepaßte Positionierung der Referenzbilder effizient codiert. Entstanden ist daraus der Encoder „SAMPEG-2“ (für **S**cene-**A**daptive-**MPEG-2**).

Kapitel 2

Wahrnehmungsphysiologie

2.1 Das menschliche Sehsystem

2.1.1 Rezeptoren

In der Netzhaut befinden sich zwei Typen von lichtempfindlichen Rezeptorzellen: einerseits die *Zapfen*, die wieder in 3 Typen unterteilt werden können, die alle eine andere Absorptionskurve aufweisen und bei unterschiedlichen Wellenlängen ihr Absorptionsmaximum erreichen, andererseits die *Stäbchen*, die zwar keine unterschiedlichen Farben unterscheiden können, dafür aber deutlich lichtempfindlicher sind, als die Zapfen. Für den bei der Betrachtung von Videosequenzen relevanten Helligkeitsbereich sind allerdings nur die Zapfen von Bedeutung.

Die drei Absorptionsmaxima der Zapfen liegen im roten (580nm), grünen (545nm) bzw. blauen (440nm) Spektralbereich. Das Empfindlichkeitsmaximum der grünen Zapfen liegt etwa um 5% höher, als das der roten und dieses wiederum um ein Vielfaches höher als das der blauen.

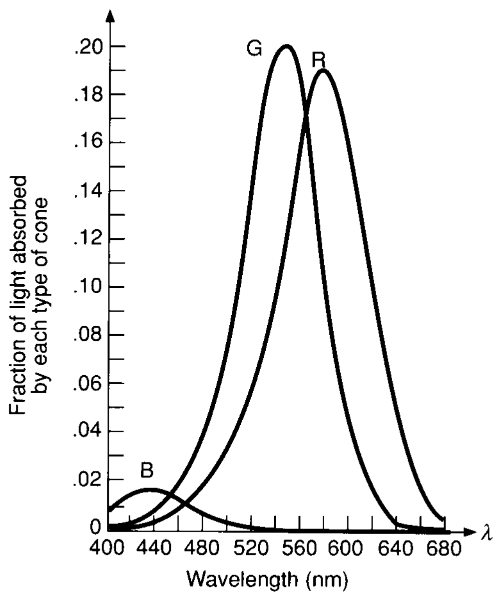


Abbildung 2.1: Spektrale Empfindlichkeit der 3 Zapfentypen (entnommen aus [FvDFH90])

Auf der menschlichen Netzhaut befinden sich etwa 1-10 Millionen Zapfen und 100 Millionen Stäbchen. Die Dichte ist aber über die Netzhaut ungleichmäßig verteilt. Sie erreicht ihr Maximum im *gelben Fleck* (*Fovea centralis*) mit etwa 160000 Zapfen pro mm^2. Auch der relative Anteil der Stäbchen und Zapfen ist ungleichmäßig verteilt. So befinden sich im gelben Fleck sehr viele kleine Zapfen, um ihn herum größere Zapfen und am Rand des Sehbereiches nur noch Stäbchen.

2.1.2 Netzhaut

Eine erste Vorverarbeitung der visuellen Information geschieht schon in der Netzhaut (siehe Abbildung 2.2). Ausgehend von den Stäbchen und Zapfen werden die Reize von Horizontalzellen,

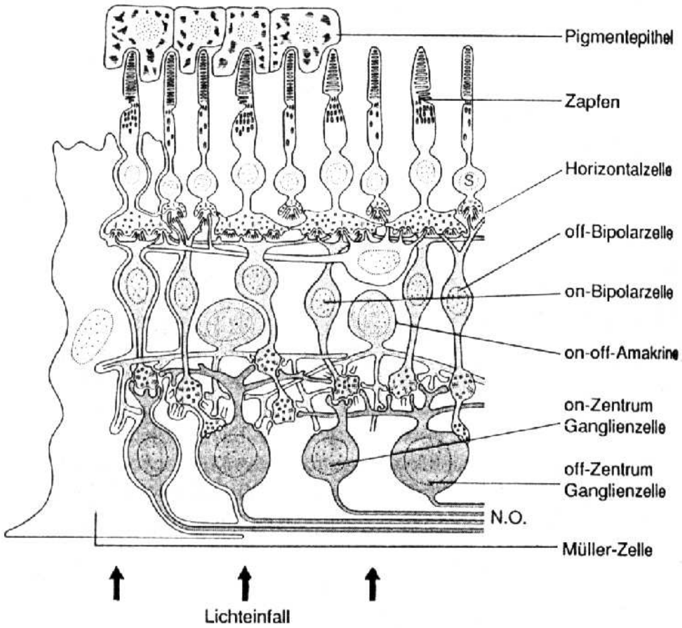


Abbildung 2.2: Signalverarbeitung in der Netzhaut (entnommen aus [HHK94])

Bipolarzellen und Amakrinen vorverarbeitet und schließlich über die Ganglienzellen an den Sehnerv weitergeleitet. Der Bereich des Sehfeldes, der sich auf die Reaktion einer Zelle auswirkt, wird als ihr *Rezeptives Feld* (RF) bezeichnet. Das rezeptive Feld einer (On-Zentrum)-Ganglienzelle ist schematisch in Abbildung 2.3 dargestellt. Fällt ein Lichtreiz auf den Zentrumsbereich des rezeptiven

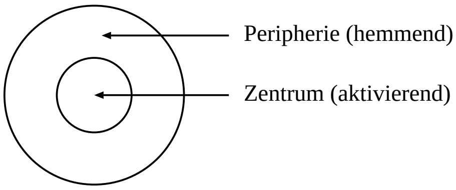


Abbildung 2.3: Rezeptives Feld einer On-Zentrum-Ganglienzelle

Feldes, so wirkt sich dies in einer Aktivierung der Ganglienzelle aus. Ein Lichtreiz im peripheren Bereich wirkt sich hingegen hemmend auf die Ganglienzelle aus. Fällt auf beide Bereiche gleich viel Licht, so gleichen sich diese Reaktionen in etwa aus. Eine nennenswerte Aktivierung tritt somit nur bei den Ganglienzellen auf, in deren rezeptive Felder eine Kante zwischen zwei unterschiedlich hellen Bereichen fällt. Die rezeptiven Felder nebeneinanderliegenden Ganglienzellen sind nun keineswegs disjunkt, sondern überschneiden sich zu einem großen Teil. Weiterhin sei noch erwähnt, daß es neben den beschriebenen On-Zentrum-Ganglienzellen auch den komplementären Typ, nämlich die Off-Zentrum-Ganglienzellen, gibt. Bei diesen ist die Funktion von Zentrum und Peripherie gerade vertauscht. Die genaue Funktion der Ganglienzellen unterscheiden sich und man kann diese in 3 Hauptklassen (die α-, β- und γ-Ganglienzellen) einteilen. Dabei besitzen die α-Zellen große rezeptive Felder und dienen vorwiegend dem Erkennen von Bewegung, wohingegen die β-Zellen viel kleinere rezeptive Felder besitzen und somit eine hohe räumliche Auflösung erreichen, was sie zur Erkennung feiner Details befähigt. Die β-Zellen nehmen einen Anteil von 80% der Ganglienzellen ein.

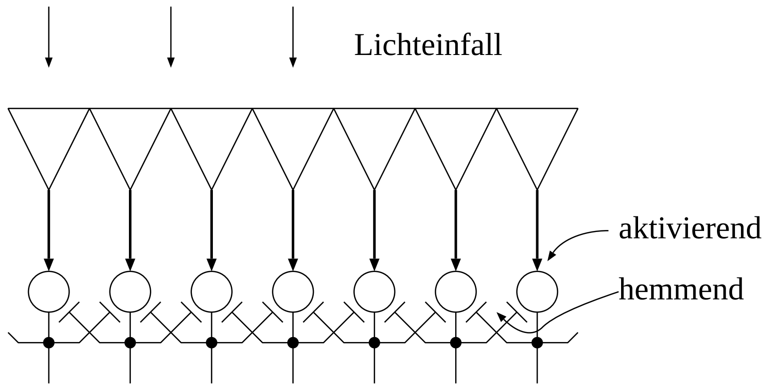


Abbildung 2.4: Prinzipdarstellung der Entstehung des Signals an den On-Zentrum-Ganglienzellen

2.1.3 Sehrinde

Die Sehrinde befindet sich am hinteren Teil des Gehirns. Auf ihr wird ein Abbild der von den Ganglienzellen erzeugten Signale abgebildet. Dabei bleiben zwar die räumlichen Nachbarschaften erhalten, aber aufgrund der unterschiedlich hohen Dichte der Nervenzellen in der Netzhaut ist das Abbild so verzerrt, daß die hochauflösenderen Gebiete des Sehfeldes auch einen größeren Bereich auf der Sehrinde einnehmen. Abbildung 2.5 zeigt sehr stark vereinfacht den Aufbau eines Elements der Sehrinde. Nebeneinanderliegende Säulen sind jeweils auf Muster in ganz bestimmten Richtungen

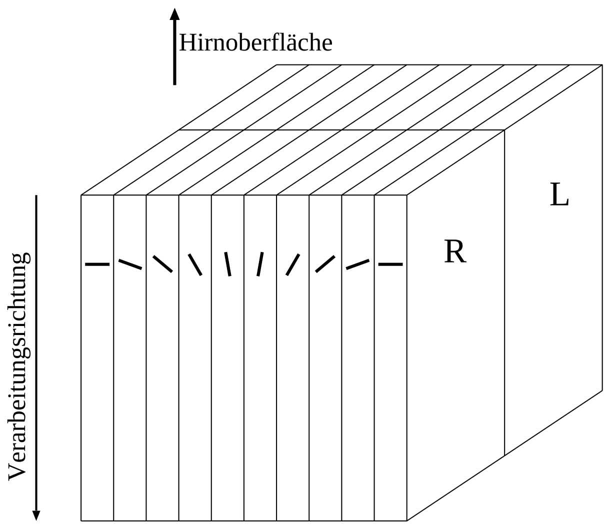


Abbildung 2.5: Stark vereinfachter Aufbau einer Einheit der Sehrinde

abgestimmt. Mit zunehmender Tiefe und fortschreitender Verarbeitung wird die Information immer allgemeiner. Während zu Anfang die Neuronen noch spezifisch auf Orientierung und Plazierung im Sehfeld reagieren, wird in tieferen Schichten von der örtlichen Lage abstrahiert.

2.2 Systemtheoretische Beschreibung

2.2.1 Ortsfrequenzauflösung

Abbildung 2.6 zeigt die Kontrastempfindlichkeit bei Präsentation von Cosinusbalken mit dem Kontrast m, der Ortsfrequenz v und der Zeitfrequenz f, gegeben durch: $L = L_0(1 + m \cos 2\pi v x \cdot \cos 2\pi f t)$. Auffällig ist die starke Abnahme der Kontrastempfindlichkeit bei höheren Ortsfrequenzen.

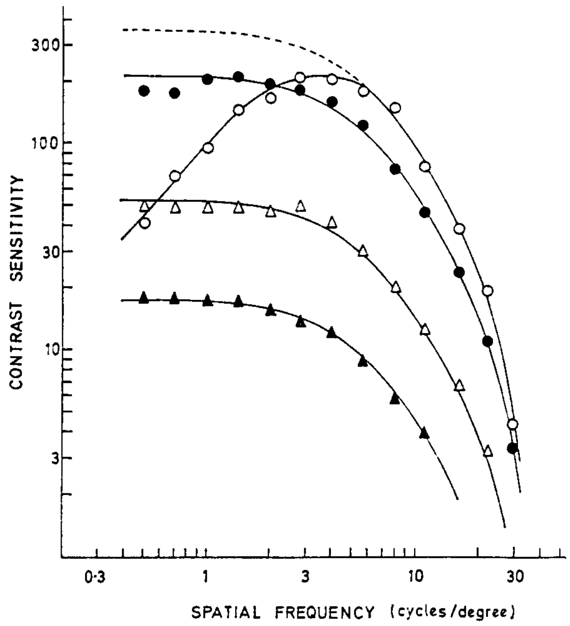


Abbildung 2.6: Kontrastempfindlichkeit in Abhängigkeit der Ortsfrequenz. Die Kurven entsprechen unterschiedlichen Zeitfrequenzen: ∘ 1 Hz, • 6 Hz, △ 16 Hz und ▲ 22 Hz (aus [Rob66]).

2.2.2 Ortsverhalten

Betrachtet man das Bild rechts oben in Abbildung 2.7, dessen Helligkeitsverlauf in dem Diagramm darunter dargestellt ist, so empfindet man an den Knickpunkten des Helligkeitsverlaufes links einen dunklen und rechts einen hellen Streifen, die als *Machbänder* bezeichnet werden. Es zeigt sich, daß dies eine Folge des Bandpaßverhaltens im Sehsystem ist (Abbildung 2.7 links oben mit Impulsantwort, Sprungantwort und Rampenantwort darunter in dieser Reihenfolge). Eine zweidimensionale Variante zur Veranschaulichung dieses Verhaltens ist das in Abbildung 2.8 dargestellte Hermanngitter. Hier tritt jeweils an den Kreuzungspunkten der hellen Linien der Effekt einer Verdunklung ein. Wir können diesen Effekt in Einklang mit unserer Theorie der rezeptiven Felder bringen, wenn wir die Funktion eines rezeptiven Feldes sehr stark vereinfacht durch den in Abbildung 2.9 links oben dargestellten linearen Faltungskern beschreiben. Das zentrale Feld entspricht dabei dem Zentrum des rezeptiven Feldes einer Ganglienzelle. Wendet man diesen Operator auf das Bild eines Hermanngitters an, so erhält man für den Kreuzungspunkt eine im Vergleich zur übrigen Kante dunkler empfundene Helligkeit.

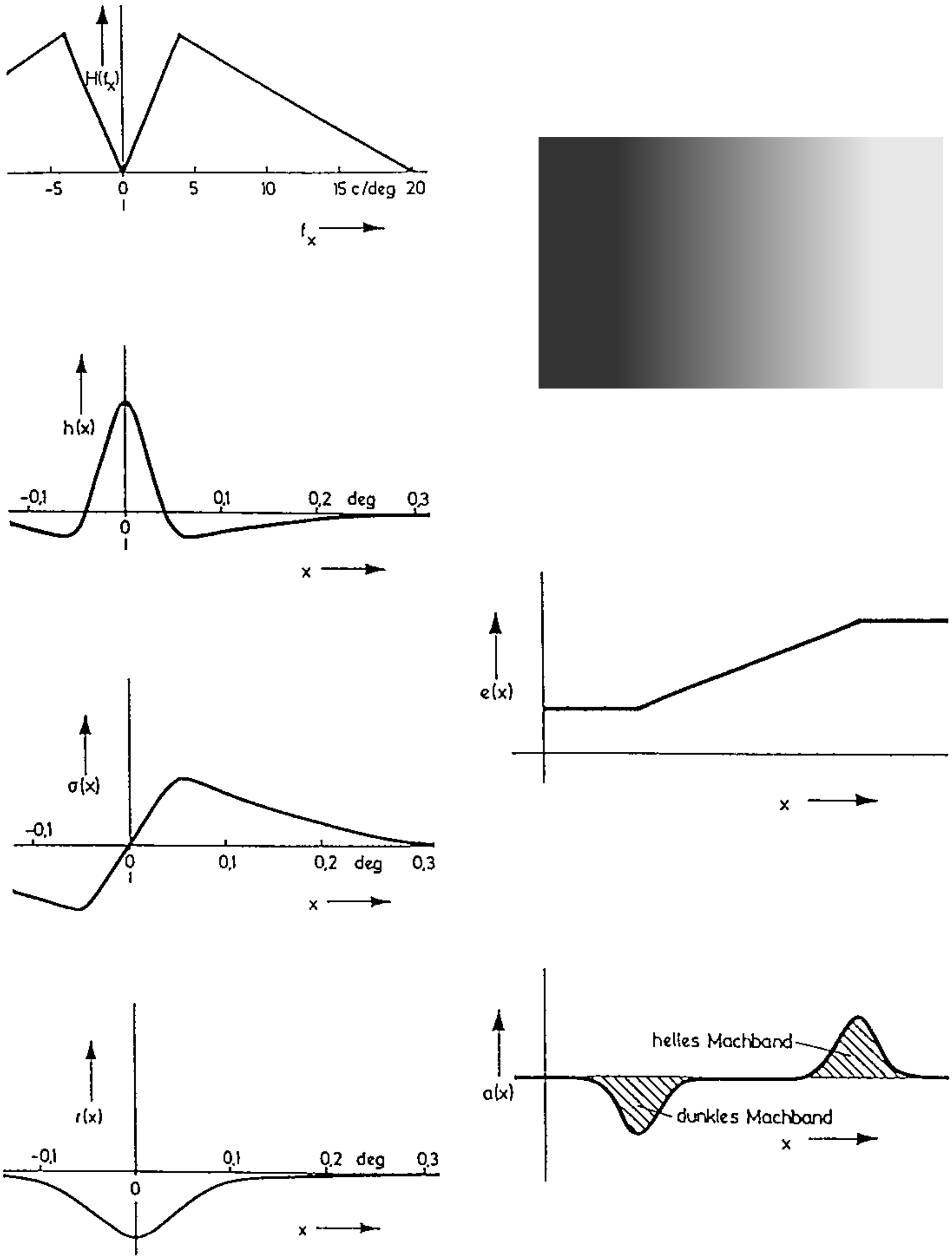


Abbildung 2.7: Machbandeffekt (entnommen aus [Hau94])

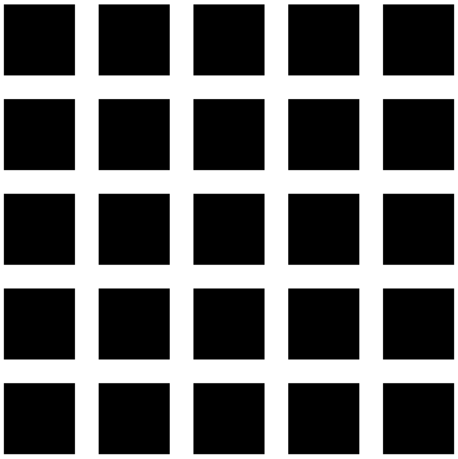


Abbildung 2.8: Hermanngitter

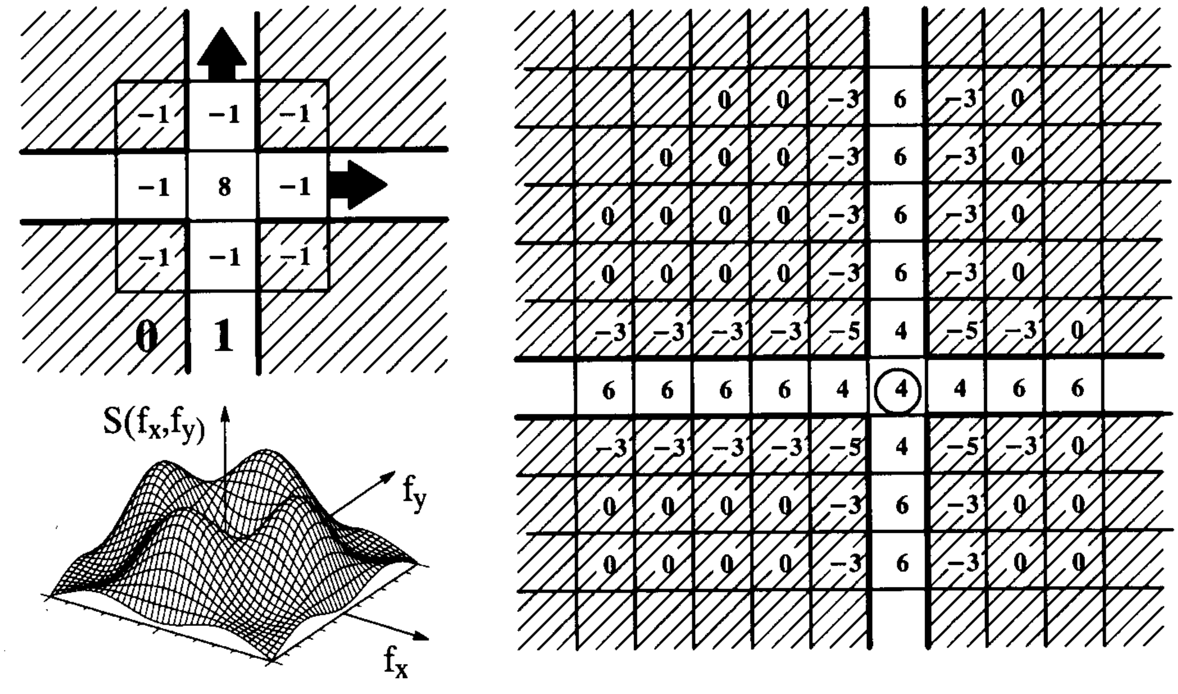


Abbildung 2.9: Erklärung zur Entstehung des Hermanngitter (aus [Hau94]).

2.3 Helligkeits- und Farbwahrnehmung

2.3.1 Weber-Fechner-Gesetz

Mißt man die minimale wahrnehmbare Leuchtdichteänderung in Abhängigkeit der Umgebungshelligkeit, so erhält man den in Abbildung 2.10 dargestellten Verlauf. Die wesentliche Erkenntnis ist, daß sich der minimal wahrnehmbare Leuchtdichteunterschied ΔL in dem Bereich der „normalen" Umgebungshelligkeiten in guter Näherung durch

$$\frac{\Delta L}{L} \approx 2\%$$

angeben läßt (*Weber-Fechner-Gesetz*). Für große Helligkeiten gilt diese Gesetzmäßigkeit jedoch aufgrund des Auftretens von Blendungen nicht mehr. Auch bei niedrigen Helligkeiten verringert sich die Empfindlichkeit. In Abbildung 2.10 deutlich erkennbar ist auch der Knickpunkt, der dadurch entsteht, daß die Stäbchen mit ihrer hohen Empfindlichkeit in diesem Helligkeitsbereich die Verringerung der Kontrastempfindlichkeit noch hinauszögern können. Die physiologische Er-

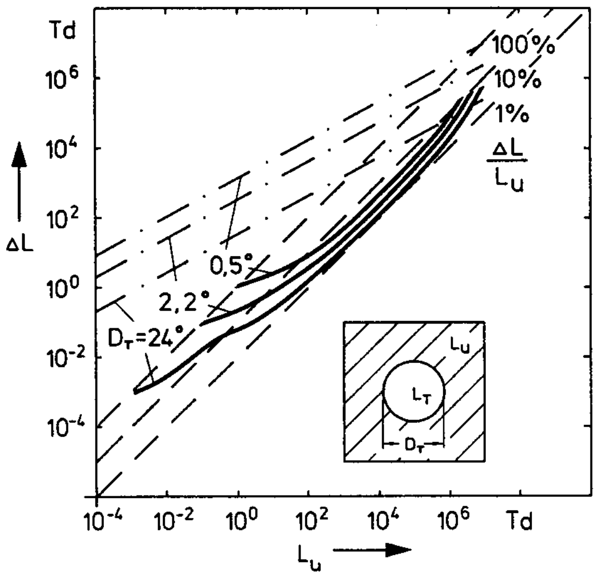


Abbildung 2.10: Wahrnehmbarkeitsschwelle von Leuchtdichteänderungen (aus [Hau94]).

klärung für dieses Verhalten liegt darin, daß bei der Verarbeitung der Helligkeitsinformation eine Logarithmierung der Helligkeit vorgenommen wird.

2.3.2 Gegenfarbenmodell

Während die Farbdarstellung zu Beginn der Verarbeitung durch die unterschiedliche spektrale Empfindlichkeit der Zapfen durch Rot-, Grün- und Blau-Kanäle geschieht, wird in späteren Verarbeitungsschritten mit einem Helligkeitskanal, einem Rot-Grün- und einem Blau-Gelb-Differenzkanal gearbeitet. Abbildung 2.11 zeigt vereinfacht, wie diese Signale aus den Rot-, Grün-, Blau-Eingabesignalen erzeugt werden.

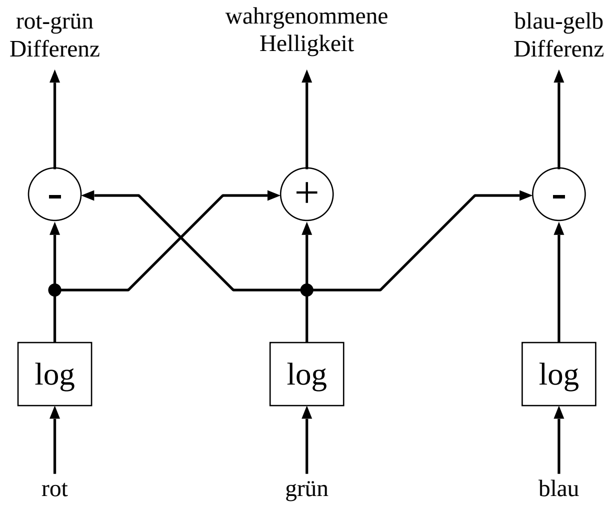


Abbildung 2.11: Gegenfarbenmodell

2.4 Maskierungseffekte

2.4.1 Temoral Masking

Nach einem vollständigen Bildwechsel vergeht eine kurze Zeit, bis das Sehsystem die neuen Details wahrnehmen kann. Dies kann man bei der Komprimierung ausnutzen, indem man in dieser Zeitspanne größere Fehler in Bild zuläßt. Umgekehrt gibt es auch einen vergleichbaren Effekt, der dazu führt, daß man auch die Detail kurz *vor* einem Wechsel des Bildinhalts nicht in vollem Umfang wahrnehmen kann.

Es existieren zwei Ausprägungen dieses Effekts. Für sehr kurze Zeitspannen bleibt die Helligkeitsempfindung konstant, wenn das Produkt aus Darstellungsdauer und Helligkeit gleich bleibt (Gesetz von Bloch). Dies wird z.B. beim zeilenweisen Aufbau eines Bildschirmbildes ausgenutzt, das mit einer bestimmten Mindestfrequenz dargestellt werden muß, um flimmerfrei zu wirken.

Auf höheren Verarbeitungsebenen, bei denen der Bildinhalt eine wesentliche Rolle spielt, ist die Verarbeitungsgeschwindigkeit allerdings geringer, so daß die kritische Zeitspanne durchaus einige hundert Millisekunden dauern kann. Dieser Effekt wird in SAMPEG durch die automatische Szenenwechselerkennung (Kapitel 6) ausgenutzt.

2.4.2 Motion Masking

Beginnt ein Objekt sich zu bewegen, dauert es eine gewisse Zeit, bis das Auge der Bewegung folgen kann. In dieser Zeit können feine Details nicht wahrgenommen werden. Sobald das Auge dieser Bewegung jedoch folgt, kann es diese Details wieder erkennen, es sei denn, das Objekt bewegt sich so schnell, daß das Auge der Bewegung nicht gut genug folgen kann. Dieser Effekt wird schon durch die Aufnahmetechnik ausgenutzt. Durch die nicht beliebig kurze Verschlußzeit der Kamera wird der Bildinhalt immer über eine gewisse Zeitperiode integriert. Bewegt sich ein Objekt sehr schnell, so gehen dadurch seine Details im Bild verloren. Dies ist auch vorteilhaft für die Codierung, da die hohen Frequenzen nicht mehr codiert werden müssen.

2.4.3 Activity Masking

In Bildteilen, die von starken Kanten durchzogen werden, sind kleine Helligkeitsschwankungen nicht so sichtbar, wie in gleichmäßigen Flächen. Die genaue quantitative Beschreibung dieses Effekts ist allerdings noch immer nicht vollständig geklärt, da es sich gezeigt hat, daß der Effekt sehr komplex ist. Es existiert sogar der komplementäre Effekt, daß kleine Helligkeitsschwankungen in der Umgebung von bestimmten Mustern besser sichtbar werden. Während einiges über die absolute Schwelle bekannt ist, unter der Helligkeitsschwankungen nicht mehr sichtbar sind, arbeitet man bei der Videocodierung teilweise in Bitratenbereichen, in der sichtbare Codierungsfehler über dieser absoluten Schwelle nicht zu vermeiden sind. Sämtliche Verfahren, die sich bemühen, diesen Maskierungseffekt auszunutzen, sind deshalb heuristischer Natur.

2.4.4 Luminance Masking

Aus dem Weber-Fechner-Gesetz folgt direkt, daß Fehler in hellen Bildregionen nicht so sehr wahrgenommen werden, wie in dunkleren. Interessanterweise scheint aber auch der gegenteilige Effekt zu existieren, daß auch Fehler in dunklen Bereichen eines insgesamt mittelhellen Bildes nicht so sehr wahrgenommen werden. Mir ist keine theoretische Erklärung für diesen Effekt bekannt. Meine Vermutung ist, daß es sich um einen rein psychologischen Effekt handelt. In Filmszenen wird nämlich schon durch die Regie eine Vorauswahl getroffen, welche Bildbereiche wichtig sind. Diese sind entsprechend gut ausgeleuchtet. Dunkle Bereiche wie Schatten genießen dagegen beim Betrachter wenig Aufmerksamkeit.

Kapitel 3

Videocodierung nach MPEG-2

3.1 Entstehungsgeschichte

1991 wurde MPEG-1 als Internationaler Standard verabschiedet. Das Ziel war die Codierung von Audio- und Videodaten auf Speichermedien wie CDROM mit Bitraten von ca. 1,5 Mbits/s. Bei dieser Datenrate, welche eine gute Bildqualität bei einer Auflösung von 352×240 (VideoCD) ermöglicht, lassen sich etwa eine Stunde Video mit Ton auf eine CDROM mit 650 MB Speicherkapazität unterbringen. MPEG-1 unterstützt zwar prinzipiell schon höhere Bitraten und Auflösungen, aber man trifft auf das Problem, daß die heute übliche Videoübertragung im Interlace-Format geschieht, was von MPEG-1 nicht effizient codiert werden kann.

Das war der Grund für die Entwicklung von MPEG-2, der dann 1995 als Internationaler Standard vorlag. Das ursprünglich geplante MPEG-3 zur Codierung von Video in den zukünftigen HDTV-Auflösungen von bis zu 1920×1152 wurde gestoppt und in MPEG-2 integriert, so daß sich nun auch mit MPEG-2 HDTV-Auflösungen effizient codieren lassen. Auch die Teile *Audio* und *System* (zuständig für das Multiplexen mehrerer Audio- und Videostreams) wurden überarbeitet. MPEG-2 Audio unterstützt nun die effiziente Codierung von 5-Kanal-Ton (Dolby-Surround). MPEG-2 Systems verbesserte die Übertragung über verlustbehaftete Kanäle und enthält weitere Features, wie die Verschlüsselung (Scrambling). Trotz allem ist MPEG-2 sehr ähnlich zu MPEG-1 und bildet eine Übermenge über die MPEG-1 Sytax, so daß jeder MPEG-2 Decoder auch MPEG-1-Streams decodieren kann.

Die Anwendungsgebiete von MPEG-2 sind sehr vielfältig und unter sie fallen z.B. digitale Fernsehübertragungen (auch HDTV) über terrestrische Sender (DVB), Kabel oder Satellit, digitaler Rundfunk (DAB), interaktive Speichermedien (DVD), digitale Videorecoder, Videokonferenzen, Multimedia-Mailing, Videoübertragung über ATM-Netzwerke und Videofernüberwachung.

3.2 Überblick

3.2.1 Prinzipien

Der MPEG-Video-Kompressions-Algorithmus basiert auf zwei grundlegenden Techniken: Blockbasierter Bewegungskompensation zur Ausnutzung der zeitlichen Redundanz und DCT basierter Transformationscodierung zur Ausnutzung der räumlichen Redundanzen. Sowohl die Übertragung der Motionvektoren der Bewegungskompensation, wie auch der DCT-Koeffizienten erfolgt Huffmancodiert.

Die Grundeinheit der Bewegungskompensation wird als Makroblock bezeichnet und ist 16×16-Luminanz-Pels groß. Die DCT wird dagegen auf 8×8-Pels großen Blöcken durchgeführt.

3.2.2 Interlaced-Video

Verarbeitet man Interlaced-Video als Vollbilder, so tritt an bewegten Objektkanten der sogenannte *Mäusezähncheneffekt* auf, da die beiden Halbbilder einen zeitlichen Abstand voneinander haben, in dem sich das Objekt schon ein Stückchen weiterbewegt haben kann (Abbildung 3.1). Dies hat

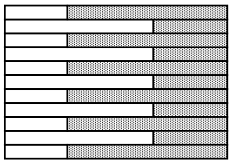


Abbildung 3.1: Mäusezähncheneffekt an bewegten Objektkanten bei Interlaced-Video

einige negative Auswirkungen auf die Codierung:

- Bei der DCT-Transformation treten sehr hohe vertikale Frequenzen auf. Da die Codierung der Koeffizienten gerade unter der Annahme optimiert wurde, daß hohe Frequenzen seltener auftreten als niedrige, hat das gravierende Auswirkungen auf die erzeugte Bitanzahl.
- Die Bewegungskompensation liefert an diesen Kanten keine gute Prädiktion, wenn sich die Geschwindigkeit der Objektbewegung ändert (Länge der Zähne verändert sich mit der Objektgeschwindigkeit).
- Redundanzen zwischen den beiden Halbbildern können nicht ausgenutzt werden, da beide immer gemeinsam codiert werden.

Ein Hauptziel von MPEG-2 war es deshalb, Methoden zur effizienteren Codierung von Interlaced-Video bereitzustellen. Das wurde insbesondere dadurch erreicht, daß nun einerseits Halbbilder getrennt codiert werden können und es andererseits neue Bewegungskompensationsmethoden gibt, die gezielt auf einzelne Halbbilder zugreifen können.

3.2.3 Stream-Hierarchie

Abbildung 3.2 zeigt den Aufbau eines MPEG-2-Video-Streams in einer Top-Down Aufspaltung. Aus der ursprünglichen Videosequenz werden jeweils einige aufeinanderfolgende Bilder zu einer *Group-Of-Pictures (GOP)* zusammengefaßt. Da MPEG-2 sowohl Progressive-Video (nur Vollbilder) wie auch Interlaced-Video unterstützt, kann jedes dieser Bilder ein *Frame* (ein Vollbild) oder auch nur ein *Field* sein (einzelnes Halbbild). Ebenso wie in MPEG-1 werden den Bildern die Typen *I, P* und *B* zugewiesen. Jedes dieser Bilder wird in *Slices* unterteilt. Jeder Slice ist 16 (Luminanz)-Pels hoch und auch ein Vielfaches von 16 Pels breit. Jeweils 16×16 Pels bilden einen Makroblock. Jeder Makroblock kann auf mehrere Weisen codiert werden. Man unterscheidet zwischen *Intra-codierten* Makroblöcken ohne Bewegungskompensation und *Inter-codierten* Makroblöcken, die (i.A.) einen oder zwei *Motionvektoren* zur Bewegungskompensation und die Differenzdaten enthalten. Die eigentlichen Bilddaten werden durch 4 Luminanzblöcke und 2,4 oder 8 Chrominanzblöcke codiert, die jeweils 8×8-Koeffizientenmatrizen sind, die durch DCT-Transformation aus den (ggf. bewegungskompensierten) Bilddaten gewonnen wurden.

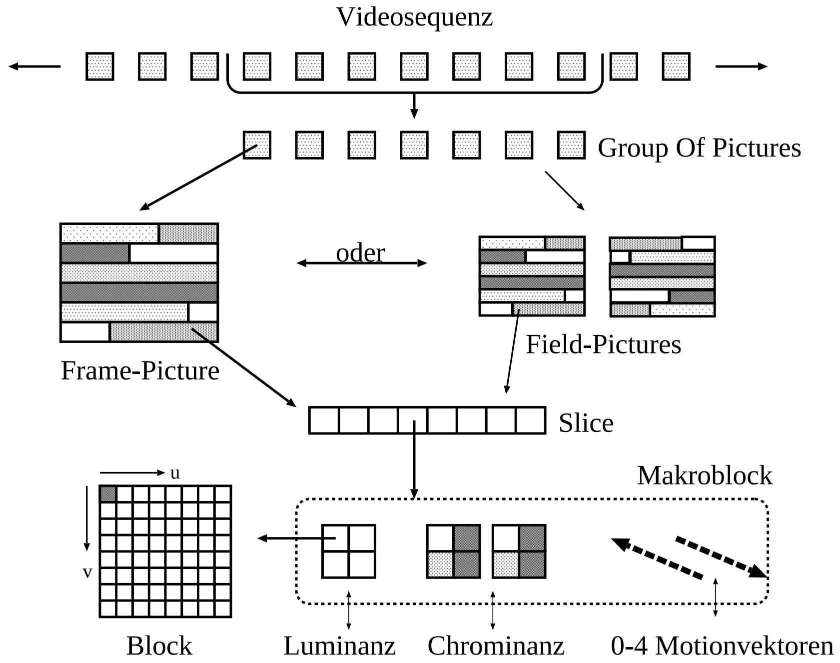


Abbildung 3.2: Hierarchie eines MPEG-2-Video-Streams

3.2.4 YCbCr-Farbraum

MPEG-2 benutzt für die Bildcodierung den YCbCr-Farbraum. Je nach Wahl der Primärfarben r,g,b ergibt sich eine leicht unterschiedliche Transformationsmatrix M von R,G,B nach Y,Cb,Cr. Welche Primärfarben benutzt werden, wird im Bitstream in der `sequence_display_extension` angegeben. Am häufigsten werden dabei Matrixkoeffizienten nach MPEG-1 benutzt. Die Farbkonversion lautet damit:

$$\begin{pmatrix} Y' \\ Cb' \\ Cr' \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.421 & -0.079 \end{pmatrix} \begin{pmatrix} R' \\ G' \\ B' \end{pmatrix}$$

Eine etwas schnellere Berechnung ist möglich, indem man die Eigenschaften

$$Cb' = \frac{0.5}{1 - M_{13}}(B' - Y') \qquad \text{und} \qquad Cr' = \frac{0.5}{1 - M_{11}}(R' - Y')$$

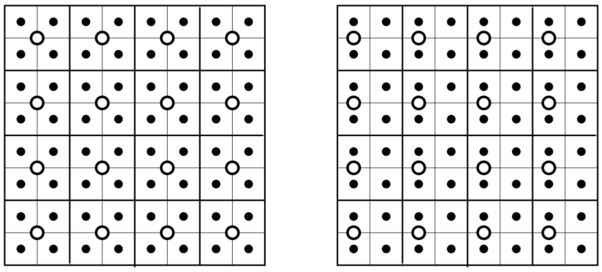
ausnutzt. Um den Wertebereich dem Intervall $[0; 255]$ anzupassen, wird noch folgende Skalierung durchgeführt:

$$Y = \frac{219}{255}Y' + 16 \qquad Cb = \frac{224}{255}Cb' + 128 \qquad Cr = \frac{224}{255}Cr' + 128$$

Der YCbCr-Farbraum ist an das Modell zur Farbwahrnehmung nach Hochberg angelehnt, das in Abschnitt 2.3.2 beschrieben ist. Dabei ist Cb der *Blau-Gelb*-Kanal und Cr der *Rot-Grün*-Kanal.

3.2.5 Farb-Samplepositionen

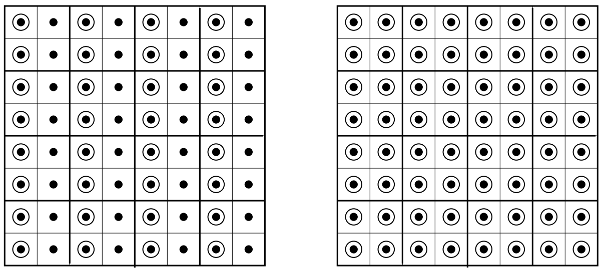
Während MPEG-1 die Auflösung der Farbkanäle stets horizontal und vertikal um den Faktor 2 dezimierte (4:2:0 Format[1]), sind bei MPEG-2 durch höhere Qualitätsforderungen und Übertragung von Interlaced-Bildern Erweiterungen notwendig. Zuerst einmal ist zu beachten, daß bei MPEG-2 die Position, auf das sich das Chroma-Sample bezieht, von der Position bei MPEG-1 unterscheidet (Abbildung 3.3). Da üblicherweise die Chromasamples an denselben Positionen, wie die Luminanzsamples entnommen werden, wurde bei MPEG-1 das Chroma-Sample durch Mittelwertbildung aller 4 umgebenden Positionen erzeugt. Bei MPEG-2 geschieht dies dagegen nur durch den Mittelwert der Positionen direkt über und unter der Chroma-Sampleposition.



(a) MPEG-1 (b) MPEG-2 (4:2:0)

Abbildung 3.3: Chroma-Sampleposition bei MPEG-1 und MPEG-2 4:2:0 Format. Die ausgefüllten Punkte stellen die Helligkeitssamples (Y) und die Kreise die Farbsamples (Cb und Cr) dar.

MPEG-2 unterstützt auch die Codierung mit einer höheren Farbauflösung als MPEG-1. Dazu stehen das 4:2:2-Format (mit lediglich horizontaler Reduktion um den Faktor 2) und das 4:4:4-Format (keine Reduktion der Farbauflösung) zur Verfügung (siehe Abbildung 3.4).

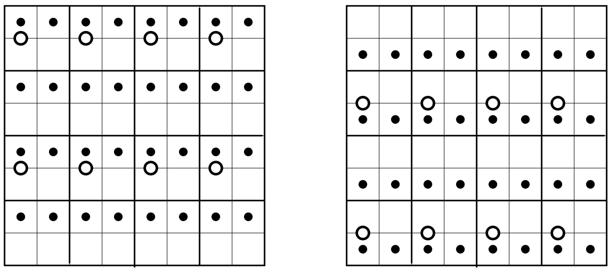


(a) MPEG-2 (4:2:2) (b) MPEG-2 (4:4:4)

Abbildung 3.4: Chroma-Sampleposition im 4:2:2 und 4:4:4 Format

Bei der Codierung von Interlaced-Bildern werden die Farbsamples nicht in die Mitte zwischen die Helligkeits-Samples eines Halbbildes gelegt (siehe Abbildung 3.5). Dadurch ergeben sich im entstehenden Vollbild die gleichen Positionen unabhängig davon, ob das Bild *progressive* oder *interlaced* codiert wird.

[1](4:2:0) wird manchmal auch als (4:1:1) bezeichnet, was aber zu Verwechslungen mit einem älteren 4:1:1-Format führen kann, daß eine horizontale 4:1 Farbreduktion aber keine vertikale Farbreduktion beinhaltete.



(a) erstes Halbbild (b) zweites Halbbild

Abbildung 3.5: Chroma-Sampleposition bei Interlaced-Material

Je nachdem, welches Chroma-Format benutzt wird, werden in einem Makroblock 6 (4:2:0), 8 (4:2:2) oder 12 (4:4:4) 8×8-Blöcke codiert (Abbildung 3.6). Vier davon sind immer Luminanz-Blöcke, die restlichen in gleichen Teilen Cb- und Cr-Chrominanzkanäle.

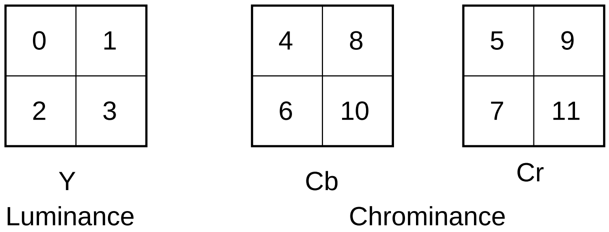


Abbildung 3.6: 8×8 Blöcke innerhalb eines Makroblocks.

3.2.6 Bewegungskompensation

MPEG benutzt zwei Typen der Bewegungskompensation. Zum einen die *Prediction*, bei der für einen zu codierenden Block aus einem zeitlich vorhergehenden Bild ein ähnlicher Bereich gleicher Größe aber ggf. abweichender Position ausgewählt wird. Codiert wird dann nur noch die Differenzinformation zwischen diesem Block und dem zu codierenden. Der andere Modus ist *Interpolation.* Hierbei wird sowohl aus einem vergangenen, wie auch aus einem zukünftigen Bild ein Bereich ausgewählt und dann nur die Differenz aus dem Mittel dieser beiden Blöcke und dem zu codierenden Block übertragen. Es wird damit eine zeitliche Interpolation gebildet (siehe Abbildung 3.7). Dieser Modus wird häufiger auch als *bidirectional prediction* bezeichnet. MPEG-1 bot die Auswahl zwischen einer Full-Pel genauen und einer Half-Pel genauen Angabe der Motionvektoren. MPEG-2 arbeitet grundsätzlich mit Half-Pel Auflösung. Befindet sich der Motionvektor auf einer Half-Pel Position, so werden die umgebenden Pixel gemittelt, um die Prädiktion zu erhalten. Da dadurch auch eine Rauschunterdrückung erzielt wird, sind die Half-Pel Position leicht bevorzugt.

3.2.7 Frametypen

Die Bilder einer Sequenz werden in verschiedene Bild-Typen eingeteilt:

I-Bilder Hier wird keine Bewegungskompensation benutzt. Damit bieten diese Frames einen Aufsetzpunkt zum Beginn der Decodierung. Außerdem wird durch gelegentliches Einstreuen von

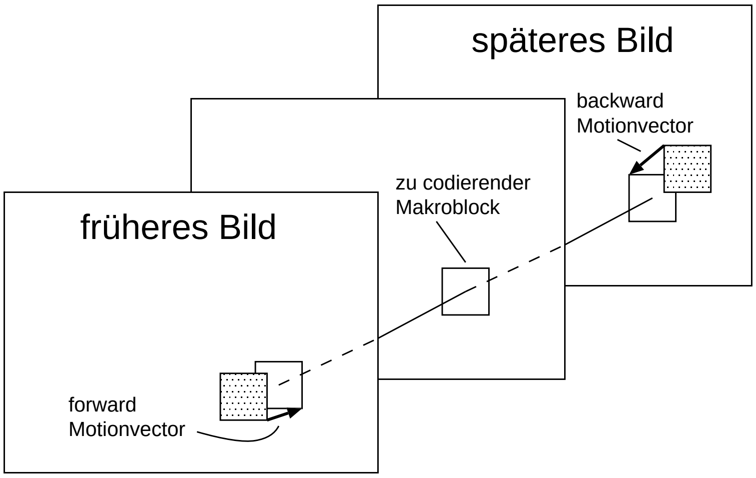


Abbildung 3.7: Bidirektionale Prädiktion mit Forward- und Backward-Motionvektor

I-Bildern die Fehlerfortpflanzung, die sich durch mehrfache aufeinanderfolgende Prädiktionen ergibt, abgebrochen.

P-Bilder Jeder Makroblock innerhalb eines P-Bilds kann sowohl ohne, wie auch mit Bewegungskompensation (Prediction) codiert werden. Referenzbild zur Bewegungskompensation ist dabei das jeweils letzte zurückliegende I- oder P-Bild. Es ist darauf zu achten, nicht allzu lange Ketten von P-Bilder zu erzeugen, da ein Fehler, der durch eine geringfügig abweichende Berechnung der IDCT oder einen Übertragungsfehler verursacht wurde, mit ins nächste Bild übernommen wird.

B-Bilder erlauben alle Modi der Bewegungskompensation, also insbesondere die bidirectional prediction, was den Frames auch ihren Namen gegeben hat. Als Referenzbilder fungieren dabei jeweils das letzte und das nächste I- oder P-Bild. Außerdem ist es möglich, nur eines der beiden Referenzbilder zur Prädiktion zu nutzen.

Die Frametypen werden normalerweise in einem „I B B P B B P B B P"-Muster verteilt. In Abbildung 3.14 sind zusätzlich noch die Abhängigkeiten durch die Prädiktion eingezeichnet. Die Benutzung von B-Bildern bietet eine Reihe von Vorteilen:

- Durch die zeitliche Interpolation wird der Rauschanteil im Bild reduziert, was sich auch auf die Codierungseffizienz auswirkt.
- Verdeckungen im Bild können besser codiert werden, da für den neuen Bildinhalt entweder eine Prädiktion aus dem Referenzbild der Vergangenheit oder dem der Zukunft gewonnen werden kann.
- Dadurch, daß B-Bilder nicht als Referenzbilder benutzt werden, wird die Fehlerfortpflanzung durch die P-Bild-Ketten verlangsamt.

aber auch einige Nachteile:

- Da B-Bilder erst codiert werden können, wenn das zukünftige Referenzbild codiert wurde, entsteht eine Verzögerung in der Codierung.

- Da zwei Referenzbilder gespeichert werden müssen, erhöht sich der benötigte Bildspeicher von 2 (Original und Referenz) auf 3 komplette Bilder.
- Die ohnehin aufwendige Motionvektorsuche erschwert sich in B-Bildern nochmals, da für jeden Block zwei Motionvektoren berechnet werden müssen.

3.2.8 Prädiktionstypen

Je nachdem, ob die Bilder als Frames oder Field codiert werden, ergeben sich unterschiedliche Prädiktionstypen. Betrachten wir zuerst die Möglichkeiten bei Frame-Bildern: dafür steht einerseits die von MPEG-1 übernommene und jetzt als *Frame Prediction for Frame Pictures* (Abbildung 3.8(a)) bezeichneten Methode zur Verfügung. Die Grundeinheit der Bewegungskompensation ist dabei ein 16×16 Block, dessen Zeilen, wenn das Videomaterial aus zwei Halbbildern besteht, immer abwechselnd von den Halbbildern stammen. Bei Interlaced-Video ist dieser Modus nur für einen stillstehenden oder langsam, gleichmäßig bewegten Hintergrund effizient. Für unregelmäßige Bewegungen besser geeignet ist bei Frame-Bildern hingegen die *Field Prediction for Frame Pictures* (Abbildung 3.9). Hierbei werden die Zeilen eines Makroblocks in die beiden Halbbilder getrennt, was dann zwei 16×8-Blöcke ergibt, für die jeweils getrennt ein Motionvektor übertragen wird. Ist das Referenzbild ein Frame-Picture, so werden die Daten für den 16×8-Block nur aus jeder 2. Zeile übernommen. Bei diesem Modus werden für jeden Makroblock in P-Bildern 2 und in B-Bildern 4 Motionvektoren übertragen.

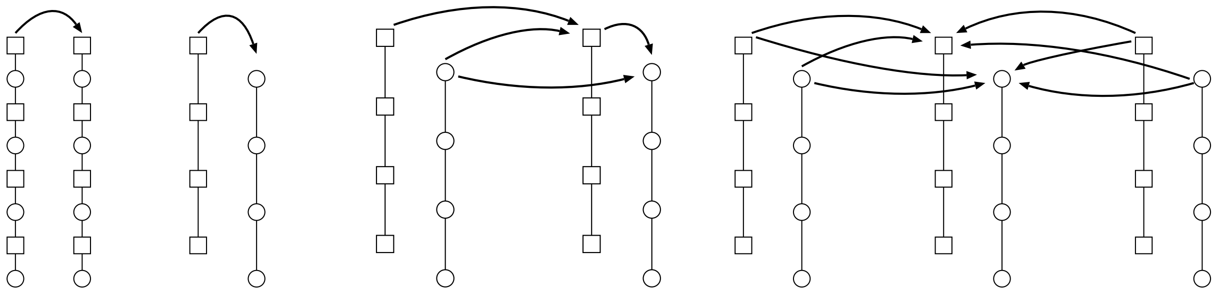


Abbildung 3.8: Motionvektor-Prädiktionstypen. Dargestellt sind die Zeilen des ersten Halbbildes (Quadrate) und die Zeilen des zweiten Halbbildes (Kreise). (a): Frame Prediction for Frame Pictures (entsprechend MPEG-1), (b)-(d): Field Prediction for Field Pictures

Für Field-Bilder gibt es den Modus *Field Prediction for Field Pictures* (Abbildung 3.8 (b)-(d)). Dieser Modus funktioniert prinzipiell wie Frame Prediction for Frame Pictures, nur daß die Bewegungskompensation hier für die Halbbilder getrennt vorgenommen wird. Jeder Motionvektor kann sich auf eines der beiden zuletzt codierten Halbbilder beziehen. Zu beachten ist hier, daß für das erste Halbbild von P-Bildern die beiden Halbbilder des vergangenen Vollbildes als Referenz dienen, für das zweite Halbbild allerdings einerseits das entsprechende Halbbild aus dem vergangenen Vollbild und das andere Halbbild, das davor für das jetzige Vollbild codiert wurde.

Der Vollständigkeit halber seien hier noch die zwei übrigen Modi genannt. 16×8 *MC for Field Pictures* entspricht Field Prediction for Field Pictures, jedoch wird auch hier, wie bei Field Prediction for Frame Pictures der Makroblock in zwei 16×8-Blöcke unterteilt, was für sehr unregelmäßige

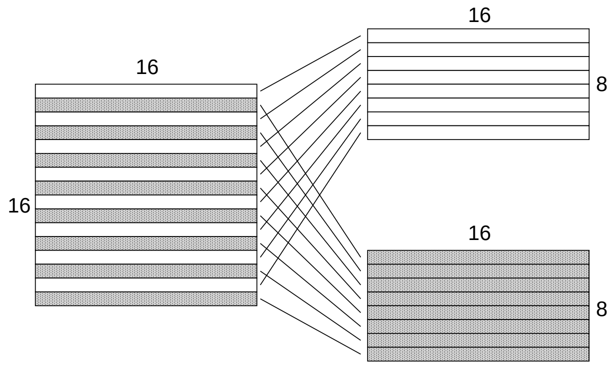


Abbildung 3.9: Field Prediction for Frame Pictures

Bewegungen von Vorteil sein kann. Denn man beachte, daß ein MB bei Field-Bildern doppelt so hoch wie breit ist. *Dual-Prime for P-Pictures* ist ein Modus, der nur benutzt werden kann, wenn zwischen dem P-Bild und der Referenz keine B-Bilder liegen. Das besondere an diesem Modus ist, daß, obwohl die Bewegungskompensation für beide Halbbilder getrennt durchgeführt wird, nur ein Motionvektor (und eine kleiner Differenzvektor) übertragen wird.

3.2.9 DCT-Transformation

Bei Bildkompressionsverfahren ist die 2D-DCT-Transformation sehr beliebt, da sie einerseits fast so gute Ergebnisse liefern kann, wie die Karhunen-Loeve-Transformation, andererseits aber sehr schnell zu berechnen ist. Im uns interessierenden 2 dimensionalen Fall lautet die FDCT:

$$F(u,v) = \frac{C(u)}{2}\frac{C(v)}{2}\sum_{y=0}^{7}\sum_{x=0}^{7} f(x,y)\cos((2x+1)u\pi/16)\cos((2y+1)v\pi/16)$$

$$C(u) = \begin{cases} 1/\sqrt{2} & \text{für } u = 0 \\ 1 & \text{für } u > 0 \end{cases}$$

und IDCT entsprechend:

$$f(x,y) = \sum_{u=0}^{7}\sum_{v=0}^{7} \frac{C(u)}{2}\frac{C(v)}{2} F(u,v)\cos((2x+1)u\pi/16)\cos((2y+1)v\pi/16)$$

Schnelle Verfahren zur Berechnung dieser Transformationen sind in Kapitel 4 erläutert. Warum sich diese Transformation so gut für die Bildkomprimierung eignet, ist intuitiv auch aus Abbildung 3.10 ersichtlich, in der die Basisfunktionen dargestellt sind. Man erkennt, daß die Muster mit $u + v \leq 2$ häufig in Bildern vorkommenden Mustern entsprechen.

Die Transformationskoeffizienten werden vor der Übertragung quantisiert. Dabei wird unterschieden, ob der Makroblock *Intra-* (ohne Bewegungskompensation) oder *Inter*-codiert wurde. Bei Intra-Codierung wird ein Uniform-Quantisierer benutzt, bei Inter-Codierung dagegen ein Dead-Zone-Quantisierer (Abbildung 3.11). Die Schrittweite T ist:

$$T = \frac{qscale \cdot w_{u,v}}{16}$$

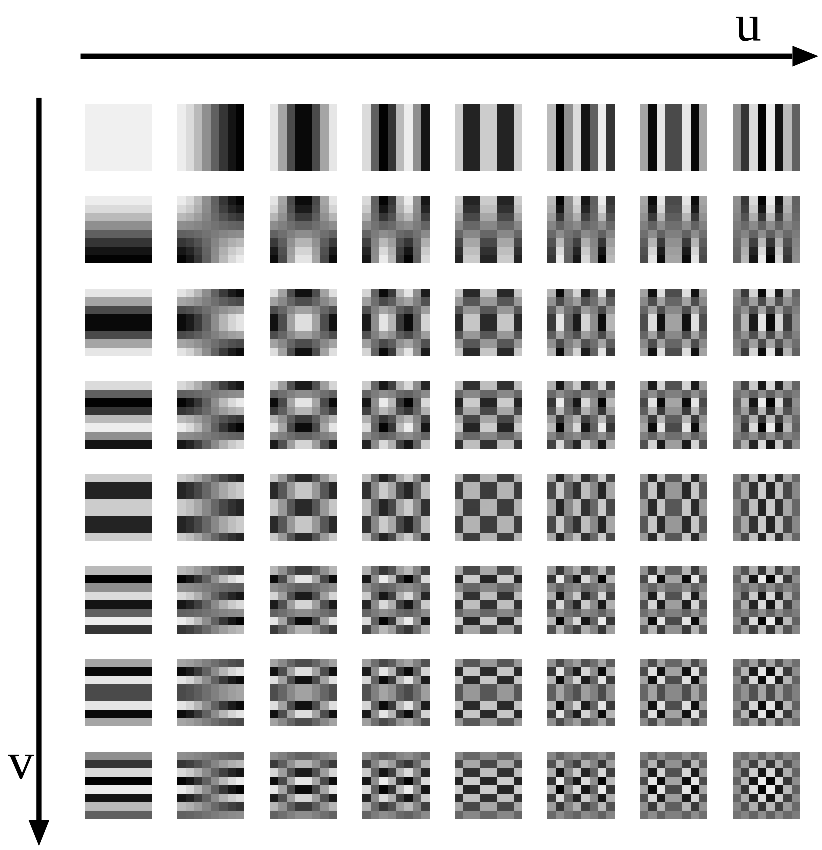


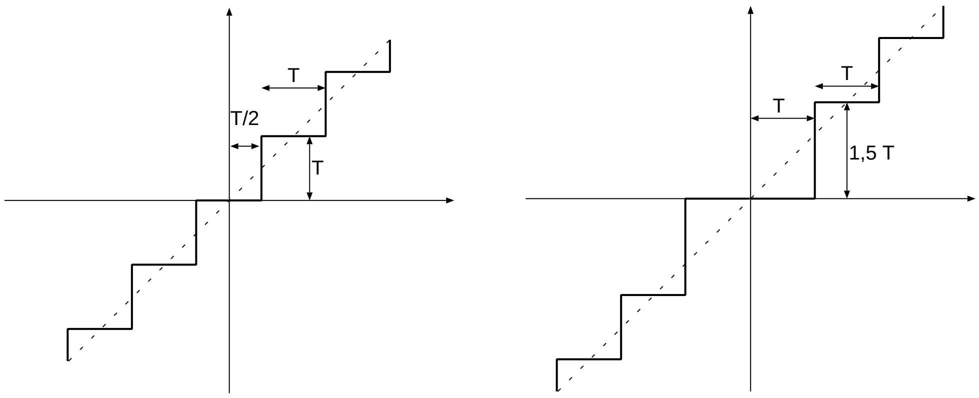
Abbildung 3.10: 2D-DCT Basisfunktionen

Dabei ist *qscale* ein Parameter, der entweder auf Slice-Ebene (ohne Codierungsoverhead) oder auf Makroblockebene (mit kleinem Overhead) geändert werden kann. $w_{u,v}$ ist eine Quantisierungsmatrix, welche bewirkt, daß die Basisfunktionen der DCT je nach Sichtbarkeit von Quantisierungsfehlern unterschiedlich stark quantisiert werden. Die Standard-Matrizen sind in Abbildung 3.12 angegeben. Diese Quantisierungsmatrizen können in einem `sequence_header` geändert werden. Also maximal nach jeder GOP.

3.3 Streamsyntax

Vorweg etwas zur Notation der Header-Felder. Die MPEG-2 Stream-Syntax bildet eine Übermenge der MPEG-1 Syntax. Neue Felder wurden an die Header als *Extension*-Headers angefügt. Teilweise wurde die Bitbreite von alten Feldern dadurch erhöht, daß die zusätzlichen Bits in einen Extension-Header übernommen wurden. In den in diesem Abschnitt enthaltenen Tabellen ist die Bitbreite deshalb mit der Bezeichnung $a + b$ angegeben. Dabei sind a die Anzahl der Bits im alten Header-Feld und b die in der Extension hinzukommenden Bits. Werden bestimme Bitbreiten nur unter besonderen Codierungsbedingungen erreicht, so sind diese zusätzlichen Bits in Klammern angegeben.

Die Syntaxbeschreibung mußte natürlich teilweise stark vereinfacht werden. Headerstukturen und Felder, die hier nicht von wesentlicher Bedeutung sind, wurden weggelassen.



(a) Intra-Blocks (b) Inter-Blocks

Abbildung 3.11: Quantisierungsfunktion für Intra- und Inter-Makroblock-Modes

$$\begin{pmatrix} 8 & 16 & 19 & 22 & 26 & 27 & 29 & 34 \\ 16 & 16 & 22 & 24 & 27 & 29 & 34 & 37 \\ 19 & 22 & 26 & 27 & 29 & 34 & 34 & 38 \\ 22 & 22 & 26 & 27 & 29 & 34 & 37 & 40 \\ 22 & 26 & 27 & 29 & 32 & 35 & 40 & 48 \\ 26 & 27 & 29 & 32 & 35 & 40 & 48 & 58 \\ 26 & 27 & 29 & 34 & 38 & 46 & 56 & 69 \\ 27 & 29 & 35 & 38 & 46 & 56 & 69 & 83 \end{pmatrix} \qquad \begin{pmatrix} 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \\ 16 & 16 & 16 & 16 & 16 & 16 & 16 & 16 \end{pmatrix}$$

(a) Intra-Matrix (b) Inter-Matrix

Abbildung 3.12: Standard Quantisierungsmatrizen

3.3.1 Sequence

In der obersten Ebene wird der Videostream prinzipiell beliebig in Teilen zerlegt, die jeweils mit einem `sequence_header` eingeleitet werden. Hier werden grundlegende Eigenschaften des Videostreams angegeben, wie Bildgröße und Bildrate, aber ggf. auch die Quantisierungsmatrizen, wenn eigene Matrizen benutzt werden sollen. Soll die Decodierung mitten in einem Videostream beginnen können (wie z.B. bei Fernsehübertragungen unbedingt gefordert), so muß der Sequence-Header in Abständen wiederholt werden. Alle Felder bis auf die Quantisierungsmatrizen müssen dabei aber konstant gehalten werden.
Die Bedeutung der nicht offensichtlichen Felder ist:

vbv buffer size definiert die minimale Größe des benötigten VBV-Buffers zum Empfang des Bitstreams in Einheiten von 16384 bits.

profile and level id Profile und Level des Bitstreams. Siehe Abschnitt 3.5.

progressive sequence ist dann auf „1“ gesetzt, wenn die Videosequenz nur aus Progressive-codierten Bildern besteht. Damit spart man sich die Bits, mit denen man bei jedem Bild den Bildtyp angeben müßte.

sequence_header	
horizontal size	12+2
vertical size	12+2
aspect ratio	4
frame rate	4+7
bit rate	18+12
vbv buffer size	10+8
optional:	
intra quantiser matrix	64*8
optional:	
inter quantiser matrix	64*8
sequence_extension	
profile and level id	8
progressive sequence	1
chroma format	2
low delay	1

Abbildung 3.13: Aufbau eines `sequence_headers`.

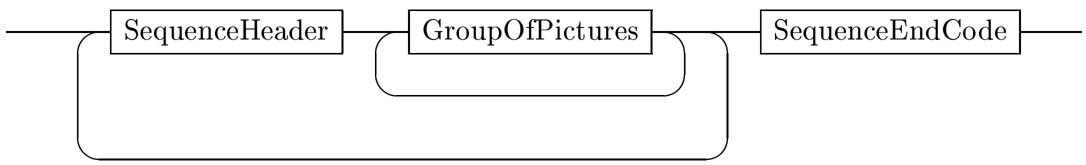
chroma format Art des Farbsamplings (4:2:0, 4:2:2 oder 4:4:4).

low delay Wenn dieses Flag auf „1“ gesetzt ist, dann enthält der Stream keine B-Bilder und der VBV-Buffer-Unterlauf ist erlaubt.

Es existieren weiter Header für besondere Zwecke. So können z.B. genauere Displayangabe (r,g,b Primärfarben) und Copyrightangaben gemacht werden, sowie alle 4 Quantisierungsmatrizen ausgetauscht werden (Inter und Intra, jeweils für Luminanz und Chrominanz getrennt).

Nach dem `sequence_header` folgen beliebig viele *Group of Pictures* (GOP). Am Ende des Streams folgt dann noch ein `sequence_end_code`, nach dem alle Parameter des `sequence_headers` wieder geändert werden dürfen.

VideoSequence



3.3.2 Group Of Pictures

Eine Group of Pictures besteht aus einem `group_of_pictures_header` und beliebig vielen Bildern. Die Bilder sind in der GOP jedoch nicht in der Reihenfolge gespeichert, in der sie angezeigt werden sollen. Das würde nämlich erfordern, daß die komprimierten Daten der B-Bilder so lange zwischengespeichert würden, bis das nächste I- oder P-Bild decodiert wurde, und erst danach diese B-Bilder zu decodieren. Dadurch könnte man den Abstand zwischen der Decodierung der I/P-Bildern nicht konstant halten, da eine längere Folge von B-Bildern die Ankunft der Daten für das P-Bild zu sehr verzögern würde. Man bräuchte mehrere Bildspeicher, um das auszugleichen und hätte ohne zusätzliche Beschränkung noch immer keine Garantie für den Erfolg.

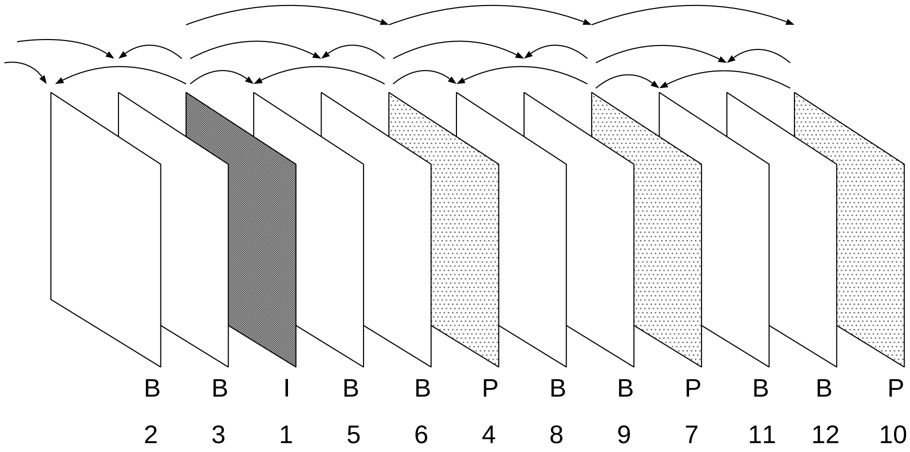


Abbildung 3.14: Abhängigkeiten innerhalb einer GOP und Übertragungsreihenfolge (Nummerierung unter den Frames)

Um dem Problem zu begegnen, wird die Übertragung der P-Bilder vor die B-Bilder gelegt, die sich vor diesem P-Bild befinden. In Abbildung 3.14 ist eine Bildsequenz in *Darstellungsreihenfolge* zu sehen. Die davon abweichende *Übertragungsreihenfolge* ist unter den Bildern angegeben.

Die Grundidee einer *Group of Pictures* war ursprünglich, eine Sequenz fortlaufender Bilder zusammenzufassen, die unabhängig von Bildern aus anderen GOPs decodiert werden können. Diese Eigenschaft ist so streng aber nicht vorhanden. Es gelten hingegen die folgenden abgeschwächten Eigenschaften:

1. Eine GOP beginnt in Übertragungsreihenfolge immer mit einem I-Bild.
2. Eine GOP beginnt in Darstellungsreihenfolge immer mit einem I- oder B-Bild und endet mit einem I- oder P-Bild.

Da sich vor dem ersten I-Bild in Darstellungsreihenfolge einige B-Bilder befinden können, die Informationen aus der letzten GOP benötigen, ist eine solche GOP nicht mehr unabhängig decodierbar, es sei denn, in den B-Bildern am Beginn werden nur backward-Motionvektoren (vom ersten I-Bild der GOP) benutzt.

group_of_pictures_header	
time code	25
closed gop	1
broken link	1

Abbildung 3.15: Aufbau eines `group_of_pictures_header`.

Im Header finden sich neben dem absoluten Timecode noch zwei für den Videoschnitt (der sehr einfach an GOP-Grenzen durchgeführt werden kann) wichtige Flags im Header: An dem Flag *closed gop* kann man erkennen, ob alle Bilder der GOP unabhängig von anderen GOPs decodiert werden können (siehe Tabelle 3.3.2 auf der vorherigen Seite). Das Flag *broken link* wird hingegen gesetzt, wenn beim Schnitt die vorhergehende GOP entfernt wurde. Ein Decoder darf dann die ersten B-Bilder nicht decodieren, da die Motion-Vektoren sich auf falsches Bildmaterial beziehen würden.

		I	B	B	P	B	B	P	B	B	P	Bsp. für eine *closed GOP*
B	B	I	B	B	P	B	B	P	B	B	P	Bsp. für eine *open GOP*

Tabelle 3.1: closed und open GOPs

3.3.3 Picture

Ein Bild (*Picture*) wird in mehrere Slices zerlegt. Jeder Slice besteht aus einer Folge von Makroblöcken. War es bei MPEG-1 noch möglich, daß das ganze Bild von einem einzigen Slice abgedeckt wurde, so erfordert MPEG-2, daß mindestens jede Makroblockreihe in einer eigenen Slice codiert wird. Dafür ist es nicht mehr notwendig, das ganze Bild mit Slices abzudecken. Was der Decoder mit den unbedeckten Bereichen zu machen hat, ist jedoch nicht definiert.

picture_header	
temporal reference	10
picture coding type	3
vbv delay	16
picture_coding_extension	
fcodes	4*4
intra dc precision	2
picture structure	2
qscale type	1
intra vlc format	1
alternate scan	1
progressive frame	1

Abbildung 3.16: Aufbau eines `picture_headers`.

Beschreibung der `picture_header`-Felder:

temporal reference gibt die zeitliche Position des Bildes innerhalb seiner GOP in Darstellungsreihenfolge an.

picture coding type Bildtyp (I,P oder B. D-Bilder sind in MPEG-2 nicht erlaubt).

vbv delay definiert, wie lange die codierten Bilddaten im Decoderbuffer bleiben. Gemessen wird vom Empfang des ersten Bytes bis zu der Entnahme aller Bilddaten.

fcodes definieren den maximalen horizontal und vertikalen Bereich der Motionvektoren. Zusätzlich auch noch nach forward- und backward-Motionvektoren getrennt. Näheres dazu unter Abschnitt 3.3.5 auf Seite 29.

intra dc precision Genauigkeit mit der der DC-Koeffizient quantisiert wird. Siehe Abschnitt 3.3.5 auf Seite 30.

picture structure Ob es ein Field- oder Frame-Bild ist.

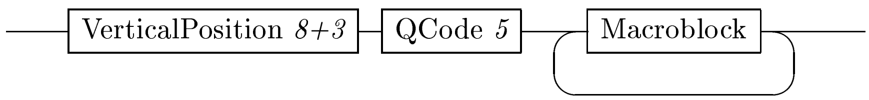
qscale type, intra vlc format, alternate scan Siehe auch Abschnitt 3.3.5 auf Seite 30.

progressive frame wird auf „0“ gesetzt, wenn das Bild aus zwei Halbbildern besteht, die einen zeitlichen Abstand voneinander haben.

3.3.4 Slice

Ein Slice bietet die kleinste Einheit, auf die nach einem Übertragungsfehler synchronisiert werden kann, denn alle kleineren Syntaxelemente bestehen nur noch aus Codes variabler Länge ohne festes Bitmuster, an dem man den Anfang der Struktur erkennen könnte. Der einzige für uns wesentliche Eintrag in einer Slice ist der notwendige QCode, über den die *qscale* gesetzt wird, was wiederum die Quantisierer-Schrittweite regelt.

Slice



3.3.5 Makroblock

Der Makroblock (MB) ist die letzte Klammer um die kleinsten Codierungseinheiten, d.h. die enthaltenen 8×8-DCT-Koeffiziente-Blöcke und die Motionvektoren für diesen MB.

Header

macroblock	
address increment	1–11
macroblock type	1–9
bei P- und B-Bilder:	
motion type	2
falls qscale codiert:	
quantiser scale code	5
bei P- und B-Bildern:	
→ (forward) **motionvector**	
bei B-Bildern:	
→ (backward) **motionvector**	
falls CBP codiert:	
coded block pattern	3–9 (+2+6)
für alle codierten Blöcke:	
→ **block**	

Abbildung 3.17: Aufbau des Makroblock-„Headers“.

Die Codierung beginnt mit einem *address increment*. Normalerweise ist dieser auf „1“ gesetzt. Ist der Wert größer, so werden einige Makroblöcke nicht codiert. Diese übersprungenen MBs werden folglich als *Skipped Macroblocks* bezeichnet. Was mit diesen nicht codierten Makroblöcken passiert, ist vom Bildtyp abhängig. In I-Bilder sind Skipped MBs nicht erlaubt, in P-Bildern wird der Codierungstyp auf *Inter* gestellt, der Motionvektor $(0;0)$ benutzt und keine Differenzbildinformationen benutzt. In B-Bildern wird auch der Codierungstyp *Inter* gewählt. Jedoch bleibt der Motionvektor der gleiche, wie der vorhergehende und auch die Referenzbezüge der Motionvektoren (forward, backward oder bidirektional) bleiben gleich. Der *macroblock address increment* wird als VLC codiert. So benötigt der Increment-Wert „1“ nur ein einziges Bit.

Daran schließt sich der Makroblock-Typ an. In einem VLC werden dabei gemeinsam diese verschiedenen Codierungsoptionen codiert:

macroblock_quant Ob der QCode in dem Makroblock geändert wird. Wenn ja, gilt er von diesem MB an bis zum nächsten Wechsel; längstens also bis zum Ende des Slices.

macroblock_motion_forward Ob ein forward-Motionvektor vorhanden ist.

macroblock_motion_backward entsprechendes für den backward-Motionvektor.

macroblock_pattern Ist dieses Flag gesetzt, so werden nicht alle (8×8)-Blöcke des Makroblocks DCT-Koeffizienten übertragen. Es wird dann im folgenden ein *coded block pattern* eingefügt, aus dem abzulesen ist, welche Blöcke codiert sind. Uncodierte Blöcke enthalten in allen Koeffizienten „0“.

macroblock_intra Wenn gesetzt, dann wird der MB ohne Bewegungskompensation codiert und sämtliche Blöcke werden übertragen. Werden keine Scalability-Modes für die Codierung benutzt (siehe Abschnitt 3.6), so ergibt sich dieses Flag zwingend aus den drei vorhergehenden.

spatial_temporal_weight_code, permitted spatial_temporal_weight_classes sind auch nur für Scalability-Modes von Bedeutung.

Einige dieser Flags schließen sich von vornherein gegenseitig aus (etwa *macroblock_intra* und *macroblock_motion_forward*), außerdem ist die Menge der Kombinationen der obigen Flags für jeden Frametyp begrenzt. Für I-Bilder ist z.B. nur das Flag `macroblock_quant` variierbar, alle anderen sind fest. Es ergeben sich also nur zwei Einträge. Für P-Bilder ergeben sich aber schon 7 und für B-Bilder 11 Einträge. Es existieren weitere VLC-Tabellen für die Codierung mit den Scalability-Modes mit bis zu 20 Einträgen. In Tabelle 3.18 sind die VLC-Tabellen für I- und P-Bilder beispielhaft angegeben.

Das Headerfeld *motion type* wählt den Typ der Bewegungskompensation und muß nicht immer übertragen werden. Auch der anschließende *quantizer scale code* ($QCode$) wird nur übertragen, wenn das entsprechende Flag im *macroblock type* gesetzt wurde. Auch das optionale Feld *coded block pattern* (CBP), das durch das Flag `macroblock_pattern` aktiviert wird, ist als VLC codiert. Der Wert des CBP ist ein Bitmuster, das für die 6 ersten Blöcke (4 Luminanz und 2 Chrominanz) bestimmt, ob sie jeweils zu codieren sind. Da bei einem Chromaformat von 4:2:2 oder 4:4:4 mehr Blöcke zu codieren sind, werden die Bitmuster zur Aktivierung an den CBR angehängt.

Motionvektoren

Die Motionvektoren werden immer als Differenz zum letzten codiert. Enthielt der letzte MB keinen entsprechenden Motionvektor, so wird die Prädiktion des Motionvektors wieder auf $(0;0)$ gesetzt.

I-Bild	
1	Intra
01	Intra, Quant
P-Bild	
1	MC, Coded
01	No MC, Coded
001	MC, Not Coded
00011	Intra
00010	MC, Coded, Quant
00001	No MC, Coded, Quant
000001	Intra, Quant

Abbildung 3.18: Beispiele von VLC-Tabellen für den `macroblock_type`.

Die Codierung der Motionvektoren erfolgt in horizontalen und vertikalen Teil getrennt und ist etwas trickreich. Eine Komponente dx des Vektores wird folgendermaßen aufgeteilt:

$$dx = |motion\_code| \cdot 2^{f\_code-1} - residual \cdot sign(motion\_code)$$

$motion\_code$ hat dabei den Wertebereich $[-16; 16]$; f_code wird im `picture_header` angegeben und bestimmt den Suchbereich. $residual$ besteht aus $f\_code - 1$ Bits, es sei denn, $motion\_code$ ist 0, dann wird $residual$ nicht codiert und auch auf 0 gesetzt. Die Codierung im Bitstream erfolgt durch einen VLC für $motion\_code$, gefolgt von den Bits für $residual$, wenn $motion\_code \neq 0$ ist.

motionvector	
bei Field-Prediction:	
field select	1
(horizontal) motion code	1–11
residual	0–8
(vertical) motion code	1–11
residual	0–8

Der Grund für die etwas unkonventionelle Codierung ist, daß damit eine nicht-Veränderung einer Motionvektorkomponente unabhängig vom Suchbereich durch ein einziges Bit codierbar ist (VLC für „$motion\_code = 0$“). Außerdem wird durch die VLCs berücksichtigt, daß kleine Motionvektoränderungen häufiger sind, als große. Dieses Prinzip wurde aber nicht vollständig verfolgt, da die letzten Stellen der Motionvektoren kaum miteinander korreliert sind und man sich so sehr große VLC-Tabellen spart.

DCT-Koeffizienten

Eine bisher noch nicht erwähnte Beziehung ist die zwischen $qscale$ und $QCode$. In der Gleichung zur Bestimmung der Quantisierer-Schrittweite tritt der Parameter $qscale$ auf, im Bitstream wurde aber immer nur ein $QCode$-Parameter übertragen. Die Funktion, die $QCode$ auf $qscale$ abbildet, wird durch den Eintrag `qscale_type` vorgenommen. Bei einem `qscale_type` gleich 0 ergibt sich $qscale$ einfach durch $qscale = QCode \cdot 2$. Bei `qscale_type` gleich 1 besteht diese Beziehung:

Es bleiben nun noch die (quantisierten) DCT-Koeffizienten zu codieren. Hierbei ist zunächst einmal zu beachten, daß bei Intra-Blöcken die DC-Komponente $u = v = 0$ unabhängig von den AC-Komponenten ($u + v \neq 0$) durch ein anderes Verfahren codiert wird.

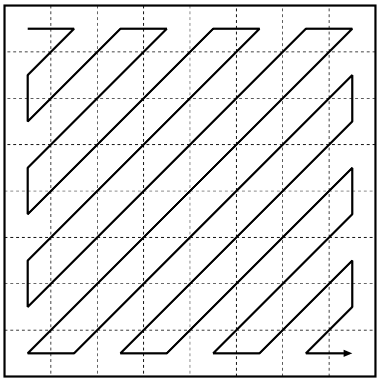
$QCode$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$qscale$	1	2	3	4	5	6	7	8	10	12	14	16	18	20	22	24
$QCode$	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
$qscale$	28	32	36	40	44	48	52	56	64	72	80	88	96	104	112	

block	
Bei Intra-Blocks:	
Bei Luminanz-Blocks:	
dct dc size luminance	2–9
Falls 'size' ungleich 0:	
differential	1–11
Bei Chrominanz-Blocks:	
dct dc size chrominance	2–9
Falls 'size' ungleich 0:	
differential	1–11
sonst:	
erster DCT Koeffizient ($u = v = 0$)	3–24
restliche DCT Koeffizienten	—
End Of Block	2

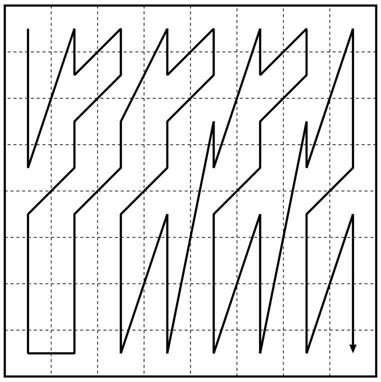
Bei den DC-Koeffizienten wird nur die Differenz zu dem letzten codierten DC-Koeffizienten des gleichen Kanals (Y,Cb,Cr) codiert. Dazu wird erst einmal festgestellt, wieviele Bits lang der Absolutwert der Differenz ist. Diese Länge wird über einen VLC codiert (und zwar nach getrennten Tabellen für Luminanz und Chrominanz). Diese Bits werden nun (leicht verändert, was hier vernachlässigt werden soll) gespeichert. Es ist zu beachten, daß keine Information darüber übertragen werden muß, ob die Differenz negativ oder positiv ist. Dies läßt sich daran erkennen, daß das erste Bit nach dem Längen-VLC gleich 0 ist. Dann kann die Differenz nämlich nicht positiv sein, da sich sonst eine kleinere Länge ergeben hätte.

Um die AC-Koeffizienten zu codieren, werden diese linear angeordnet. Die Reihenfolge wird das Flag `alternate_scan` in der `picture_header_extension` angegeben. Ist es auf 0 gesetzt, so wird die schon von MPEG-1 bekannte Zig-Zag-Ordnung benutzt (Abbildung 3.19(a)), sonst die Alternate Scan Ordnung (Abbildung 3.19(b)). Die Zig-Zag-Ordnung ordnet die Koeffizienten etwa nach aufsteigender Frequenz. Da die Koeffizienten mit höheren Frequenzen stärker quantisiert werden, erhöht sich die Wahrscheinlichkeit, daß diese ganz zu 0 quantisiert werden. Aufeinanderfolgende „0"-Koeffizienten können besonders effizient codiert werden, wie wir gleich sehen werden. Die Alternate Scan-Ordnung ist für die Codierung von Interlaced Video im Frame-Format gedacht. Da hier häufig sehr große horizontale Frequenzen auftreten, werden diese relativ weit vorne eingeordnet.

Die Codierung der Koeffizienten geschieht nun durch VLCs, die jeweils die Anzahl der 0-Koeffizienten vor dem nächsten Koeffizienten $\neq 0$ und den Wert des Koeffizienten zusammen codieren. Dies bewirkt eine effizientere Codierung, da es wahrscheinlicher ist, daß nach einer langen 0-Reihe ein kleiner Koeffizient folgt, als ein großer und umgekehrt. Nach dem letzten von 0 verschiedenen Koeffizienten kann die Codierung des Blockes vollständig beendet werden, indem ein *End Of Block* (EOB) codiert wird. Nun ist nicht für jede Kombination aus Länge der 0-Reihe und Koeffizient ein Eintrag in der VLC-Tabelle enthalten. Für die selteneren Fälle existiert dabei auch noch die Möglichkeit, Länge der 0-Reihe und Koeffizient als *fixed length code* zu speichern, was



(a) Zigzag Scan



(b) Alternate Scan

Abbildung 3.19: Zigzag Scan (MPEG-1 und MPEG-2) und Alternate Scan (nur MPEG-2)

aber verhältnismäßig viele Bits benötigt (24). Es bleibt noch anzumerken, daß es bei MPEG-2 zwei dieser Tabellen gibt. Wird im `picture_header` das Flag `intra_vlc_format` auf 1 gesetzt, so wird für die Codierung von Intra-Blöcken eine andere VLC-Tabelle benutzt, die etwas effizienter ist.

3.4 Video Buffering Verifier

Durch die Komprimierung wird ein Bitstream mit stark variierender Bitrate erzeugt, der so nicht direkt über einen Kanal mit konstanter Bitrate übertragen werden kann. Um das zu ermöglichen wird auf Encoder- und Decoder-Seite ein Buffer eingeführt, der mit konstanter Bitrate geleert bzw. gefüllt wird und aus dem die Daten ungleichmäßig entnommen werden können. Der Encoder hat die Codierungsparameter (in erster Linie die Quantisierung) so zu wählen, daß dieser Buffer nicht über- und auch nicht unterläuft. Abbildung 3.20 zeigt den Bufferfüllstand im Laufe der Zeit bei der Übertragung der Bildsequenz „I P B B B P B B B P B B B ... ". Der Video Buffering Verifier ist ein idealisiertes Modell eines Buffers, bei dem die Daten eines Bildes augenblicklich entfernt und eingefügt werden können. Eine davon abweichende Realisierung muß diese Abweichung ggf. durch die Größe des Buffers ausgleichen. Die im Stream angegebenen Buffergrößen, die für die Decodierung mindestens erforderlich sind, beziehen sich immer auf dieses idealisierte Modell.

3.5 Profiles und Levels

MPEG-2 definiert ein so großes Gebiet von Möglichkeiten und einen so weiten Einstellbereich der Codierungsparameter, daß es vor allem für eine Hardwareimplementierung unmöglich ist, alles zu unterstützen. Deshalb gibt es Einschränkungen in der Syntax und der Semantik, um eine Teilmenge zu definieren, die für bestimmte Anwendungen ausreichend ist. Ein *Profile* definiert dabei eine Teilmenge der Syntax und ein *Level* eine Teilmenge der erlaubten Parameterwerte. Beide sind unabhängig voneinander definiert und hierarchisch aufgebaut. D.h. ein höheres Profile oder ein höherer Level umfaßt stets die niedrigeren. Die Profiles und ihre wesentlichen Einschränkungen sind:

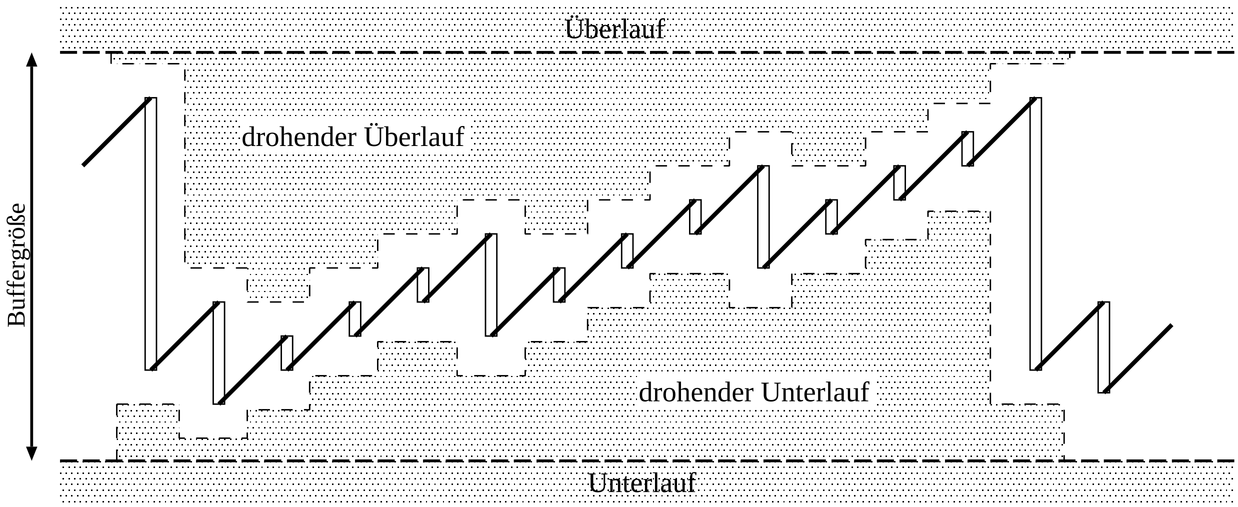


Abbildung 3.20: Video Buffer Verifier

Syntaxelement	**Simple**	**Main**	**High**
chroma format	4:2:0	4:2:0	4:2:2 oder 4:2:0
Bildtypen	I,P	I,P,B	I,P,B
intra dc precision	8–10	8–10	8–11

In dieser Tabelle wurden die beiden Profiles *SNR* und *Spatial* unterschlagen, die zwischen *Main* und *High* angesiedelt sind und speziell für die Scalability-Modes zuständig sind.
Die Levels sind im wesentlichen definiert durch:

Syntaxelement	**Low**	**Main**	**High-1440**	**High**
Auflösung	352×288	720×576	1440×1152	1920×1152
(Voll)Bilder pro Sekunde	30	30	60	60
Bitrate in Mb/s	4	15	60	80
vertikaler MV-Bereich	$[-64; 63,5]$	$[-128; 127,5]$	$[-128; 127,5]$	$[-128; 127,5]$
VBV Buffergröße in Bits	475 136	1 835 008	7 340 032	9 781 248

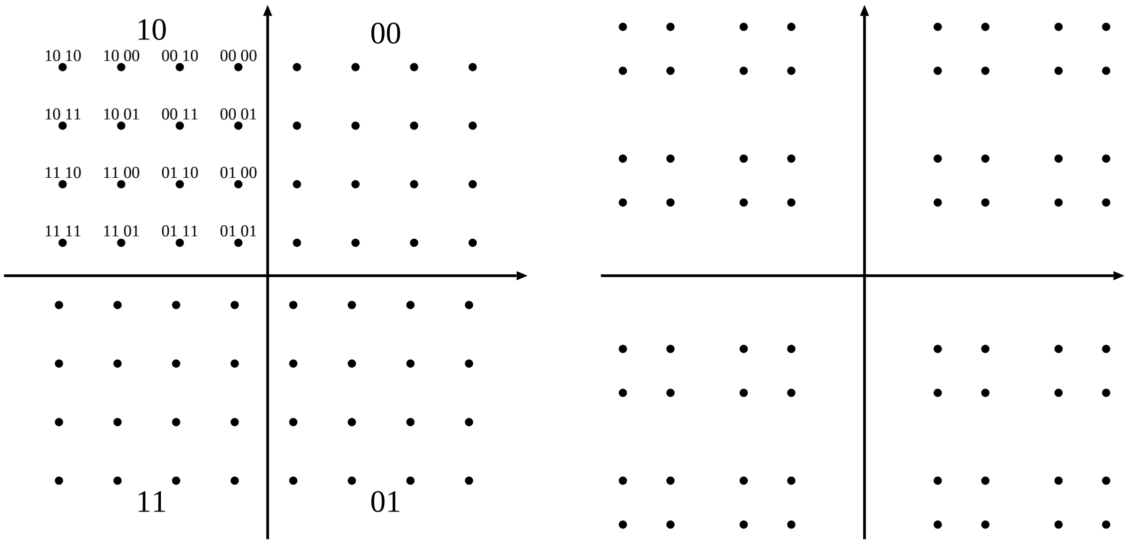
Kombinationen aus Profile und Level sind üblicherweise in einer Kurzschreibweise wie **MP@ML** oder **HP@LL** angegeben. Das derzeit üblichste Format ist **MP@ML**, das für die Übertragung von Video entsprechend dem derzeitigen analogen Format geeignet ist. **High-1440** und **High** sind mit ihren hohen Auflösungen für HDTV-Anwendungen gedacht.

3.6 Scalability Modes

3.6.1 Anwendungsgebiete

MPEG-2 unterstützt eine hierarchische Codierung der Videodaten. Diese hat zwei wesentliche Anwendungsgebiete: will man im Sinne des technologischen Fortschritts z.B. von den bisher benutzten Auflösungen zu HDTV-Auflösungen migrieren, so können die alten Geräte diese höheren Anforderungen nicht erreichen. Um aber höhere Auflösungen übertragen zu können, ohne daß sämtliche Empfangsgeräte ausgetauscht werden müssen, codiert man in mehreren Ebenen. Eine untere Ebene enthält den Datenstrom in dem bislang gewohnten Format und eine höhere Ebene codiert *aufbauend* auf den unteren Ebenen die Mehrinformation dazu. MPEG-2 erlaubt bis zu drei Ebenen, die als *Base*, *Enhancement 1* und *Enhancement 2* bezeichnet werden.

Eine zweite Anwendung ist die *graceful degradation*. Vom analogen Fernsehen ist man es gewöhnt, daß bei abnehmender Empfangssignalqualität die Bildqualität langsam abnimmt. Bei digitalen Verfahren ist dahingegen ein ziemlich abrupter Abbruch der Übertragung normal. Um auch hier ein verlangsamtes Abbrechen zu ermöglichen, wird ein Modulationsverfahren gewählt, das eine hierarchische Decodierung erlaubt. Eine Möglichkeit ist z.B. von einer einfachen 64-QAM Modulation (Abbildung 3.21(a)) auszugehen. Jeder Phasen/Amplitude-Kombination entspricht dabei ein 6-stelliges Bitmuster. Der Trick ist nun, die Vektorzustände im I-Q-Diagramm ungleichmäßig zu verteilen (Abbildung 3.21(b)). Verteilt man die Bits von 3 Codierungsebenen so, daß die Hauptquadranteninformation je 2 Bits des *Base*-Layers entsprechen (usw.), so erhält man für die einzelnen Ebenen verschiedene Rauschabstände. Anstatt komplett abzubrechen, werden nur noch die einwandfrei empfangbaren Anteile decodiert. Es ergibt sich ein langsamerer Abfall der Qualität (siehe Abbildung 3.22). In MPEG-2 werden 4 Arten von Scalability-Modes definiert, die auch miteinander kombiniert werden können.



(a) Uniform 64-QAM

(b) Non-Uniform 64-QAM

Abbildung 3.21: Vektorendpunkte bei gleichmäßiger und ungleichmäßiger Verteilung bei der 64-QAM

3.6.2 Data-Partitioning

Data-Partitioning ist die einfachste Scalability-Variante. Bei ihr wird ein Bitstream so in zwei getrennte aufgeteilt, daß der erste die gesamten Kontrollinformationen und die niederfrequenten DCT-Koeffizienten enthält. Der zweite enthält zur Synchronisation mit der Base-Layer ebenfalls die Kontrollinformationen, aber nur die höherfrequenten DCT-Koeffizienten. Der genaue Punkt, an dem die Koeffizienten aufgeteilt werden, ist variabel.

Data-Partitioning ist hauptsächlich für die Behandlung von Zellenverlust in ATM-Netzwerken

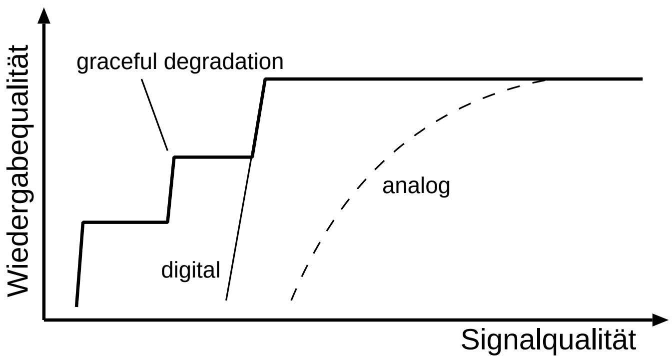


Abbildung 3.22: Abfall der Wiedergabequalität bei sinkender Signalqualität

gedacht. Eine sehr schnelle Umcodierung des Videos auf andere Bitraten ist ohne großen Aufwand möglich.

3.6.3 SNR-Scalability

SNR-Scalability hat eine gewisse Ähnlichkeit mit Data-Partitioning. Auch hier werden nur die DCT-Koeffizienten verändert. Anstatt höherfrequente Koeffizienten einfach aus dem Base-Layer zu entfernen, werden hier jedoch alle Koeffizienten lediglich stärker quantisiert und die Differenz dann in einem höheren Layer feiner quantisiert übertragen. Die Bildqualität des Base-Layers ist besser als bei Data-Partitioning, da die Schärfe nicht so stark abnimmt, sondern nur das Rauschen entlang von Kanten verstärkt wird.

Auch diese Umcodierung ist noch mit sehr geringem Aufwand möglich. Nach der Huffmandecodierung müssen die Koeffizienten lediglich neu quantisiert werden, eine DCT ist nicht erforderlich. Allerdings muß beachtet werden, daß durch die höherquantisierten Koeffizienten ein gewisser Fehlerdrift erzeugt werden kann, wenn das höhere Layer nicht decodiert wird. Ein Encoder, der diesen Drift verhindert, ist aber nur unwesentlich umfangreicher und das Problem kann durch eine Verkürzung des Abstandes von I-Bildern auch verringert werden.

SNR-Scalability ist hauptsächlich für die Anwendung des „graceful degradation“ geeignet.

3.6.4 Spatial-Scalability

Bei *Spatial-Scalability* wird im Base-Layer das Video mit einer verringerten Auflösung übertragen. Das höhere Layer enthält eine Mischung aus neuen Motion-Vektoren, DCT-Koeffizienten und Gewichtungsfaktoren. Zur Decodierung muß zunächst der Base-Layer komplett decodiert werden. Dieses Bild wird dann hochskaliert und ggf. als Ausschnitt eines größeren betrachtet. Bei der Decodierung des höheren Layers wird das decodierte Bild des Base-Layers als Prädiktion benutzt. Das endgültige Bild entsteht dann durch Benutzung der neuen Motionvektoren und dem Differenzsignal, das aus den DCT-Koeffizienten gewonnen wird. Das ganze Verfahren ist sehr kompliziert und aufwendig. Die Bildqualität ist hierbei jedoch auch bei alleiniger Decodierung des Base-Layers sehr gut.

Gedacht ist dieser Scalability-Mode für den einfachen Umstieg von Videosystemen im PAL-Format zu Systemen mit HDTV-Auflösungen.

3.6.5 Temporal-Scalability

Hierbei wird die Anzahl der Frames pro Sekunde verringert. Hat das Ausgangsvideo z.B. 50 Bilder/s, so kann es so codiert werden, daß der Base-Layer beispielsweise nur 10 Bilder/s enthält und der Enhancement-Layer die übrigen. Die Codierung ist wesentlich komplexer, als man vielleicht im ersten Moment vermuten würde, da es z.B. bei Temporal-Scalability abweichend von der Regel ausnahmsweise erlaubt ist, B-Bilder als Referenzframes zu benutzen.

Auch dieser Modus eignet sich gut für die Anwendung der „graceful degradation“.

3.6.6 Hybrid Scalability

Hybrid Scalability bezeichnet keinen eigenen Scalability-Mode, sondern man versteht darunter die Verknüpfung mehrerer Scalability-Modes in einer Übertragung. So kann z.B. zwischen dem Base- und Enhancement-Layer SNR-scalability benutzt werden und daraufhin nochmals Spatial-Scalability zwischen dem Enhancement-Layer und dem Higher-Layer. Allerdings sind nicht sämtliche Kombinationen erlaubt. Bei einer Übertragung mit 64-QAM bietet sich die Aufteilung in 3 Ebenen an, die alle die gleiche Bitrate zugewiesen bekommen.

3.7 Prinzipieller Aufbau eines Encoders und eines Decoders

Hier soll kurz der grundlegende Aufbau eines MPEG-Encoders und -Decoders dargestellt werden, wobei auf eine mögliche Parallelisierung hier nicht eingegangen wird. Zum besseren Verständnis empfiehlt es sich, zuerst den Decoder zu betrachten, denn auch jeder Encoder benötigt intern einen vollständigen Decoder zur Codierung.

3.7.1 Decoder

Der grobe Aufbau eines einfachen Decoder ist in Abbildung 3.23 zu sehen. Ausgehend vom Eingabebuffer werden die Daten Huffman-dekomprimiert und im wesentlichen die DCT-Koeffizienten von den Motionvektordaten getrennt. Nach der Dequantisierung der Koeffizienten und der inversen DCT-Transformation werden die erhaltenen Bilddaten gegebenfalls zu einer Prädiktion addiert. Dies geschieht allerdings nur in P- oder B-Bildern und auch hier nur dann, wenn ein Makroblock Inter-codiert ist. Im I-Bild und in Inter-codierten Makroblöcken ist die Prädiktion 0 und das Ergebnis der DCT-Transformation ist direkt das Ausgabebild (im YCbCr-Farbraum).

Ist das eben decodierte Bild ein I- oder B-Bild, so wandert es für die spätere Benutzung als Prädiktion in einen Bildspeicher. Das vorhin dort gespeicherte Bild wandert in einen zweiten Bildspeicher. Man denke daran, daß die Bilder nicht in der richtigen zeitlichen Reihenfolge übertragen werden. Vielmehr werden sie so übertragen, daß sich in einem Bildspeicher immer die Daten für die forward-Prädiktion und in dem anderen die Daten für die backward-Prädiktion befinden. Zusammen mit den im Stream codierten Motionvektoren erstellt die Motion-Compensation-Einheit nun mit den beiden Bildern in den Bildspeichern eine Prädiktion für die P- und B-Bilder.

Nicht dargestellt ist im Bild, daß die Darstellung eines fertig dekomprimierten I- oder P-Bildes ggf. verzögert werden muß, da ja die B-Bilder zeitlich zu spät übertragen werden.

3.7.2 Encoder

Betrachten wir den Encoderaufbau in Bild 3.24, so können wir darin alle wesentlichen Teile des Decoders (Motion-Compensation, 2 Bildspeicher, inverse Quantisierung und IDCT) in gleicher

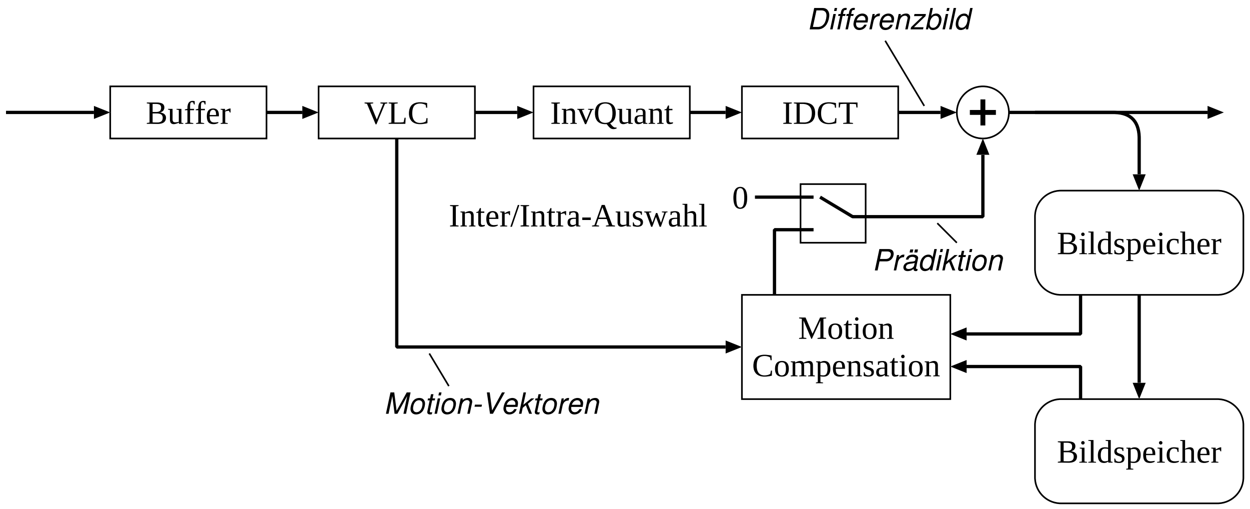


Abbildung 3.23: MPEG Decoder

Zusammenschaltung wiederfinden. Da die genaue Funktionsweise des Encoders nicht festgelegt wurde, kann ein Encoder zwar auch anders aufgebaut werden, aber die prinzipielle Funktionsweise wird immer so wie hier geschildert, aussehen. Betrachten wir zu Beginn die Codierung eines I-Bildes. Dabei wird die Prädiktion abgeschaltet, indem sie auf 0 gesetzt wird. In die Forward-DCT kommt also direkt das Eingabebild. Die erhaltenen Koeffizienten werden nun quantisiert, huffmancodiert und in den Ausgabebuffer geschrieben. Gleichzeitig werden die soeben codierten Daten aber wieder decodiert (inverse Quantisierung, IDCT, Abspeichern im Bildspeicher). Wird nun das nächste Bild codiert, das nun ein P- oder B-Bild sein soll, dann tritt die Motion-Estimation in Aktion, die zu codierendes Bild und die Bilder in den Bildspeichern vergleicht und passende Motionvektoren sucht. Diese werden einerseits huffmancodiert in den Ausgabestream eingefügt und gleichzeitig der Motion-Compensation übergeben, die wie vorher der Decoder die Prädiktion berechnet. Diese wird nun vom Eingabebild subtrahiert. Der weitere Vorgang ist wie gehabt, nur daß nun nicht mehr direkt das Eingabebild DCT-transformiert wird, sondern das Differenzbild (es sei denn, die Motion-Estimation hat sich entschlossen, einen Makroblock *Intra* zu codieren).

Wird verlangt, daß ein erzeugter Bitstream eine vorgegebene Bitrate einhält, so ist noch eine Rate-Control-Einheit erforderlich. Meist wird diese realisiert, indem der Zustand des Ausgabebuffers überwacht wird und in Abhängigkeit davon die Quantisierungsschrittweite geregelt wird. Ebenfalls nicht vorgeschrieben ist ein Modul, das eine Adaptive Quantisierung vornimmt, indem es das Eingabebild auf mögliche Maskierungseffekte untersucht und damit auch auf die Quantisierungsschrittweite einwirkt. Mit diesem nicht trivial zu realisierendem Modul kann eine Verbesserung der subjektiven Bildqualität erreicht werden.

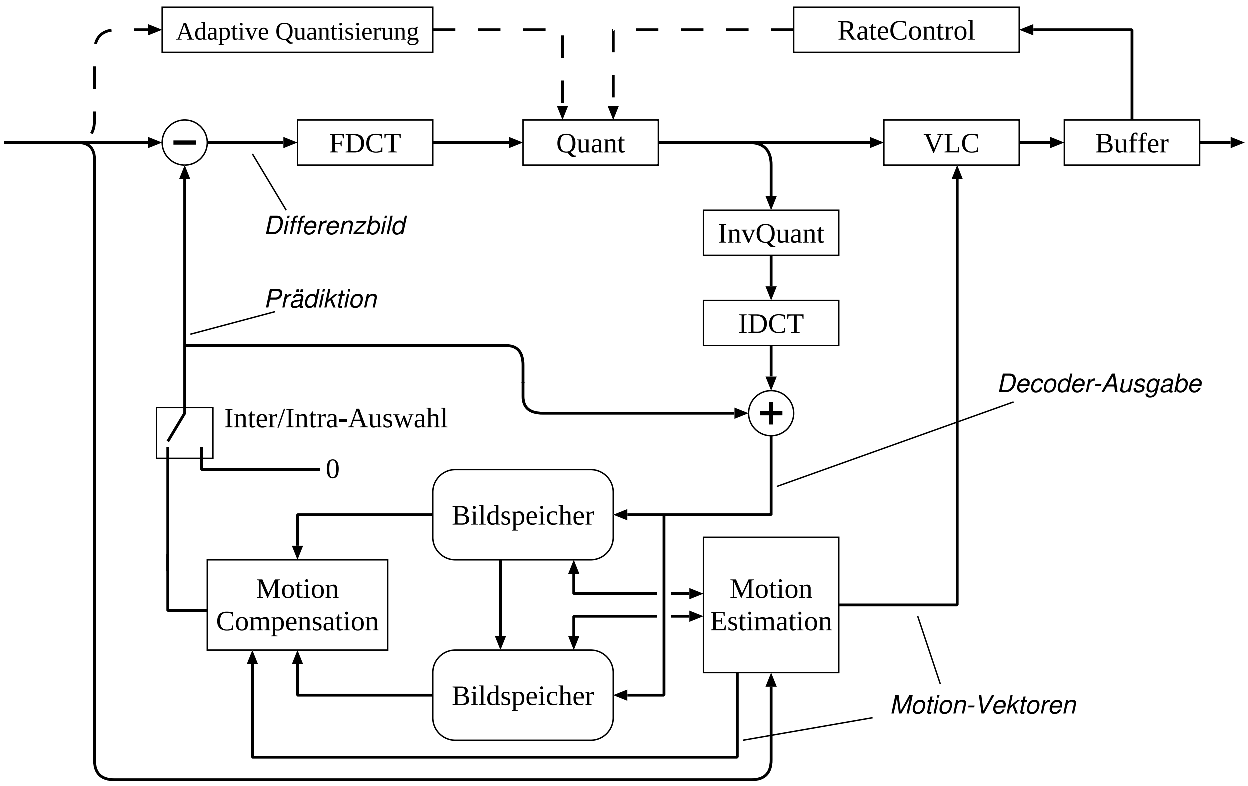


Abbildung 3.24: MPEG Encoder

Kapitel 4

Schnelle DCT

Ein wesentlicher Punkt bei der Codierung nach MPEG ist die DCT-Transformation. Sie nimmt einen so großen Teil der benötigten Rechenzeit ein, daß es sich lohnt, hier schnelle Algorithmen zu benutzen. Aufgrund einer Symmetrieeigenschaft der DCT lassen sich sämtliche Algorithmen zur DCT (auch *Forward DCT*, FDCT, genannt) auch direkt auf die inverse DCT (IDCT) anwenden.

4.1 Matrixdarstellung der DCT-Transformation

4.1.1 Der eindimensionale Fall

Betrachten wir zunächst den eindimensionalen Fall der DCT:

$$S(u) = \frac{C(u)}{2} \sum_{x=0}^{7} s(x) \cos((2x+1)u\pi/16)$$

mit

$$C(u) = \begin{cases} 1/\sqrt{2} & \text{für } u = 0 \\ 1 & \text{für } u > 0 \end{cases}$$

Die eindimensionale DCT-Transformation kann auch durch eine Matrixmultiplikation dargestellt werden. Dabei sei $\mathbf{s} = (s_0; \dots; s_7)^{\mathrm{T}}$ der (Spalten-)Vektor der Eingabedaten und $\mathbf{D} = (d_{u;x})$ die Transformationsmatrix mit $d_{u;x} = \frac{C(u)}{2} \cos((2x+1)u\pi/16)$. Die Transformation läßt sich nun einfach mittels

$$\mathbf{S} = \mathbf{D} \cdot \mathbf{s} = \begin{pmatrix} d_{0;0} & d_{0;1} & \dots & d_{0;7} \\ d_{1;0} & d_{1;1} & \dots & d_{1;7} \\ \vdots & \vdots & \ddots & \vdots \\ d_{7;0} & d_{7;1} & \dots & d_{7;7} \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ \vdots \\ s_7 \end{pmatrix}$$

darstellen. Eine Zeile der Matrix entspricht dabei einer Basisfunktion der Transformation.

Eine Methode, die Transformation zu beschleunigen, wird im folgenden sein, eine Faktorisierung der Matrix $\mathbf{D} = \mathbf{D_n} \cdot \mathbf{D_{n-1}} \cdot \ldots \cdot \mathbf{D_1}$ zu finden, so daß zur Berechnung von $\mathbf{S} = \mathbf{D_n} \cdot \mathbf{D_{n-1}} \cdot \ldots \cdot \mathbf{D_1} \cdot \mathbf{s}$ möglichst wenige Multiplikationen (mit Multiplikanten $\notin \{0; 1; -1\}$) benötigt werden.

Da die DCT eine orthonormale Transformation ist, gilt ($\mathbf{E}$ sei die 8×8-Einheitsmatrix):

$$\mathbf{D} \cdot \mathbf{D}^{\mathrm{T}} = \mathbf{E}$$

$\mathbf{D}$ ist damit eine orthogonale Matrix. Damit ergibt sich aber auch:

$$\mathbf{D}^{\mathrm{T}} = \mathbf{D}^{-1} \tag{4.1}$$

Diese Eigenschaft ist sehr hilfreich, denn haben wir einen schnellen Algorithmus für die FDCT gemäß einer Faktorisierung $\mathbf{D} = \mathbf{D_n} \cdot \mathbf{D_{n-1}} \cdot \ldots \cdot \mathbf{D_1}$, so haben wir damit auch gleich einen schnellen Algorithmus für die IDCT mit der Faktorisierung $\mathbf{D^{-1}} = \mathbf{D}^{\mathrm{T}} = \mathbf{D_1^{\mathrm{T}}} \cdot \mathbf{D_2^{\mathrm{T}}} \cdot \ldots \cdot \mathbf{D_n^{\mathrm{T}}}$. Wir werden uns deshalb im folgenden auf die FDCT beschränken.

4.1.2 Der zweidimensionale Fall

Sei $\mathbf{D^{(1)}}$ die Transformationsmatrix für eine eindimensionale Transformation, dann läßt sich die zugehörige zweidimensionale Transformationsmatrix $\mathbf{D^{(2)}}$ durch das Tensorprodukt

$$\mathbf{D^{(2)}} = \mathbf{D^{(1)}} \otimes \mathbf{D^{(1)}} = \begin{pmatrix} d_{0;0}\mathbf{D^{(1)}} & d_{0;1}\mathbf{D^{(1)}} & \ldots & d_{0;7}\mathbf{D^{(1)}} \\ d_{1;0}\mathbf{D^{(1)}} & d_{1;1}\mathbf{D^{(1)}} & \ldots & d_{1;7}\mathbf{D^{(1)}} \\ \vdots & \vdots & \ddots & \vdots \\ d_{7;0}\mathbf{D^{(1)}} & d_{7;1}\mathbf{D^{(1)}} & \ldots & d_{7;7}\mathbf{D^{(1)}} \end{pmatrix}$$

darstellen. Als Eingabevektor dient dann der Vektor $(s_0; \ldots; s_{63})^{\mathrm{T}}$ mit $s_{8u+v} = s(u,v)$. Anschaulich bedeutet dies, daß wir die Eingabewerte in Spalten aufteilen und diese zu einer langen Spalte zusammensetzen (siehe Abbildung 4.1).

Abbildung 4.1: Zusammensetzen des 64-Elemente Eingabevektors (gedreht)

4.2 Row-Column Verfahren

Betrachten wir die Definition der 8×8-Punkt FDCT

$$\mathcal{S}(u,v) = \frac{C(u)}{2}\frac{C(v)}{2}\sum_{y=0}^{7}\sum_{x=0}^{7} s(x,y)\cos((2x+1)u\pi/16)\cos((2y+1)v\pi/16)$$

mit

$$C(u) = \begin{cases} 1/\sqrt{2} & \text{für } u = 0 \\ 1 & \text{für } u > 0 \end{cases}$$

und der 8×8-Punkt IDCT

$$s(x,y) = \sum_{u=0}^{7}\sum_{v=0}^{7} \frac{C(u)}{2}\frac{C(v)}{2}\mathcal{S}(u,v)\cos((2x+1)u\pi/16)\cos((2y+1)v\pi/16)$$

und zählen, wieviele Operationen zur Berechnung notwendig sind. Dabei berechnen wir natürlich die Cosinus-Werte nicht wirklich, sondern speichern die auftretenden Werte in einer Tabelle. Auch die Konstanten $\frac{C(\cdot)}{2}$ können wir im voraus mit den Werten in der Cosinus-Tabelle zusammenfassen, so daß wir in Wirklichkeit $\frac{C(u)}{2}cos((2x+1)u\pi/16)$ in die Tabelle eintragen. Für die naive Berechnung eines Koeffizienten benötigen wir jeweils 64 Multiplikationen und 63 Additionen. Insgesamt für alle 64 Koeffizienten kommen wir somit auf 4096 Multiplikationen und 4032 Additionen. Das wäre für die Praxis viel zu viel.

Glücklicherweise sieht man, daß man schon durch geschickte Klammerung die Anzahl erheblich verringern kann. Dazu schreiben wir die beiden Gleichungen in leicht anderer Form:

$$\mathcal{S}(u,v) = \frac{C(v)}{2}\sum_{y=0}^{7}\left(\frac{C(u)}{2}\sum_{x=0}^{7} s(x,y)\cos((2x+1)u\pi/16)\right)\cos((2y+1)v\pi/16)$$

$$s(x,y) = \sum_{u=0}^{7}\frac{C(u)}{2}\left(\sum_{v=0}^{7}\frac{C(v)}{2}\mathcal{S}(u,v)\cos((2y+1)v\pi/16)\right)\cos((2x+1)u\pi/16)$$

Nun lassen sich jeweils beide Summen getrennt nacheinander berechnen, was jedesmal einer eindimensionalen 8-Punkt DCT entspricht. Wir kommen nun auf 1024 Multiplikationen und 896 Additionen. Das ist schon eine recht starke Reduktion der Anzahl an Operationen (aber es geht noch bei weitem besser). Dieser Ansatz ist als *Row-Column*-Ansatz bekannt. Würden wir einen schnellen Algorithmus zur Berechnung der eindimensionalen 8-Punkt DCT haben, so könnten wir diesen Ansatz benutzen, um daraus direkt einen Algorithmus für eine schnelle zweidimensionale 8×8-DCT zu erhalten.

4.3 Scaled DCT

Da nach der FDCT zumeist direkt eine Quantisierung der Koeffizienten erfolgt, bzw. entsprechend eine Dequantisierung vor der IDCT, könnten wir viele Multiplikationen bei der DCT effektiv einsparen, wenn diese am Schluß berechnet würden, da diese dann mit der Quantisierung zusammengefaßt werden können. Diese Algorithmen bezeichnet man als *scaled DCTs*. Die in Abschnitt 4.5 und 4.6 vorgestellten Algorithmen fallen in diese Klasse.

Eine 8-Punkt DCT kann nicht mit weniger als 11 Multiplikationen durchgeführt werden. Der Algorithmus von [AAN88] (Abschnitt 4.5) benötigt dafür zwar 13 Multiplikationen, hat aber 8 am Ende, so daß effektiv nur noch 5 für die DCT übrig bleiben.

4.4 IDCT Genauigkeitsanforderungen nach IEEE 1180

Im MPEG Standard wird darauf verwiesen, daß eine IDCT Implementierung die im IEEE Std 1180-1990 Standard [IEE91] beschriebenen Genauigkeitsanforderungen zu erfüllen hat, um zu verhindern, daß durch abweichende Implementierungen ein sichtbarer Drift im decodierten Bild entsteht. Dazu werden jeweils 10000 8×8-Blöcke mit einem genau definierten Zufallszahlengenerator für die Wertebereiche $[-256; 255]$, $[-5; 5]$ und $[-300; 300]$ erzeugt. Diese werden mit einer mindestens 64 bit Floating-Point Zahlen benutzende FDCT transformiert und die Koeffizienten auf den Bereich $[-2048; 2047]$ beschränkt. Diese Koeffizienten werden dann einerseits wieder mit einer mindestens 64 bit Floating-Point IDCT-Implementierung zurücktransformiert und andererseits auch mit der zu prüfenden IDCT. Wenn $e_k(x,y)$ mit $0 \leq x, y \leq 7$ die Differenzen zwischen den beiden IDCT Ergebnissen von Block k angibt, so sind folgende Beschränkungen einzuhalten:

$$|e_k(x,y)| \leq 1 \qquad \forall k, x, y$$

$$\frac{1}{10000}\sum_{k=1}^{10000} e_k^2(x,y) \leq 0,06 \qquad \forall x,y$$

$$\frac{1}{64000}\sum_{x=0}^{7}\sum_{y=0}^{7}\sum_{k=1}^{10000} e_k^2(x,y) \leq 0.02$$

$$\frac{1}{10000}\sum_{k=1}^{10000} e_k(x,y) \leq 0.015 \qquad \forall x,y$$

$$\frac{1}{64000}\sum_{x=0}^{7}\sum_{y=0}^{7}\sum_{k=1}^{10000} e_k(x,y) \leq 0.0015$$

Zusätzlich ist gefordert, daß die IDCT von einem Koeffizientenblock, der nur aus Nullen besteht, auch nur Nullen enthält.

4.5 Algorithmus nach Arai, Agui, Nakajima

Der nun vorzustellende Algorithmus nach [AAN88] ist zwar nicht der schnellste derzeit bekannteste in Bezug auf die Anzahl der Multiplikationen, läßt sich aber sehr einfach sowohl mit Floating-Point wie auch mit Integerzahlen mit guter Genauigkeit implementieren. Auch läßt er sich sehr gut auf SIMD-Architekturen parallelisieren, da jeweils 8 Datenpfade unabhängig voneinander berechnet werden können. Zu guter letzt ist es auch noch ein Scaled-DCT-Algorithmus, was die effektive Anzahl an Multiplikationen auf 5 reduziert. Da dieser Algorihmus in der Implementierung benutzt wurde, soll er etwas detailierter beschrieben werden.

4.5.1 Rückführung der 8-Punkt DCT auf die 16-Punkt DFT

Ahmed [NAR74] hat schon angedeutet und Haralick [Har76] hat es weiter ausgeführt, daß es möglich ist, die Koeffizienten einer N-Punkt DCT aus den ersten N Koeffizienten einer diskreten $2N$-Punkt Fouriertransformation (DFT) zu erhalten. Dazu ergänzte er die 8 Eingabewerte $s(x)$ mit $x \in \{0\ldots 7\}$ durch weitere acht Werte, die den ersten acht Eingabewerten in umgekehrter Reihenfolge entsprechen (siehe auch Abbildung 4.2 auf der nächsten Seite). Er erhält damit den Eingabevektor

$$f(x) = \begin{cases} s(x) & x \in \{0\ldots 7\} \\ s(15-x) & x \in \{8\ldots 15\} \end{cases}$$

Mit der Definition der 16. komplexen Einheitswurzel durch $\omega_{16} = e^{2\pi j/16}$ ist die 16-Punkt DFT definiert durch:

$$\mathcal{F}(u) = \sum_{x=0}^{15} f(x)\omega_{16}^{ux} \tag{4.2}$$

Nun wenden wir die DFT auf die Eingabedaten $f(x)$ an:

$$\mathcal{F}(u) = \sum_{x=0}^{7} f(x)\omega_{16}^{ux} + \sum_{x=8}^{15} f(15-x)\omega_{16}^{ux} \tag{4.3}$$

$$= \sum_{x=0}^{7} s(x)\omega_{16}^{ux} + \sum_{x=0}^{7} s(x)\omega_{16}^{u(15-x)} \tag{4.4}$$

$$= \sum_{x=0}^{7} s(x)(\omega_{16}^{ux} + \omega_{16}^{16u-u(x+1)}) \tag{4.5}$$

Da $\omega_{16}^{16} = 1$ und damit auch $\omega_{16}^{16n} = 1$ für alle $n \in \mathbb{N}$, gilt weiter:

$$\mathcal{F}(u) = \sum_{x=0}^{7} s(x)(\omega_{16}^{ux} + \omega_{16}^{-u(x+1)}) \tag{4.6}$$

Multiplizieren wir das ganze mit $\frac{1}{2}\omega_{16}^{u/2}$, so erhalten wir

$$\omega_{16}^{u/2}\mathcal{F}(u) = \sum_{x=0}^{7} s(x)(\omega_{16}^{u(x+\frac{1}{2})} + \omega_{16}^{-u(x+\frac{1}{2})}) \tag{4.7}$$

$$= \sum_{x=0}^{7} s(x)(e^{\frac{2\pi j}{16}u(x+\frac{1}{2})} + e^{-\frac{2\pi j}{16}u(x+\frac{1}{2})}) \tag{4.8}$$

$$= \sum_{x=0}^{7} s(x)(e^{\frac{\pi j}{16}u(2x+1)} + e^{-\frac{\pi j}{16}u(2x+1)}) \tag{4.9}$$

$$= \sum_{x=0}^{7} s(x)\cos((2x+1)u\pi/16) \tag{4.10}$$

Womit wir die DCT-Koeffizienten (bis auf die konstanten Faktoren $\frac{C(\cdot)}{2}$) durch die Koeffizienten der DFT-Transformation bestimmt haben. Tseng und Miller [TM78] haben nun weiter festgestellt, daß es reicht, den Realteil der DFT zu berechnen, was im folgenden gezeigt werden soll.

Da die Koeffizienten der DCT-Transformation reell sind und damit die rechte Seite der Gleichung (4.10) rein reell ist, muß auch das Produkt auf der linken Seite rein reell sein. Zerlegen wir $\mathcal{F}(u)$ in Real- und Imaginärteil $\mathcal{F}(u) = A_u + jB_u$, so erhalten wir:

$$\frac{1}{2}\mathcal{F}(u)\omega_{16}^{u/2} = (A_u + jB_u)(\cos(\pi u/16) - j\sin(\pi u/16))$$

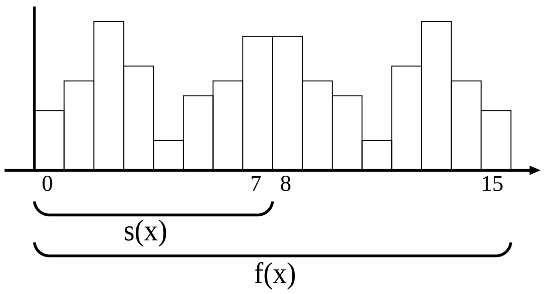


Abbildung 4.2: Eingabe für die DFT zur Berechnung der DCT

Da beide Seiten reell sein müssen, erhalten wir weiter:

$$jB_u \cos(\pi u/16) - A_u j \sin(\pi u/16) = 0$$

$$B_u = A_u \frac{\sin(\pi u/16)}{\cos(\pi u/16)}$$

Dies in (4.5.1) eingesetzt ergibt nun:

$$\begin{aligned}
\mathcal{F}(u)\omega_{16}^{u/2} &= A_u \cos(\pi u/16) + B_u \sin(\pi u/16) && (4.11)\\
&= A_u\big(\cos(\pi u/16) + \frac{\sin^2(\pi u/16)}{\cos(\pi u/16)}\big) && (4.12)\\
&= A_u \frac{1}{\cos(\pi u/16)} && (4.13)\\
&= \mathrm{Re}\{\mathcal{F}(u)\}\sec(\pi u/16) && (4.14)
\end{aligned}$$

Insgesamt erhalten wir also:

$$\sum_{x=0}^{7} f(x)\cos((2x+1)u\pi/16) = \frac{1}{2}\mathrm{Re}\{\mathcal{F}(u)\}\sec(\pi u/16)$$

4.5.2 Schnelle 16-Punkt DFT nach Winograd

Winograd zeigt in [Win78] eine neue schnelle Methode der Fouriertransformation mit einer minimalen Anzahl an Multiplikationen. Im Anhang gibt er die Algorithmen für einige Eingabelängen (darunter auch 16) explizit an. Der Algorithmus für die 16-Punkt DFT mit den Eingabedaten $f_0 \dots f_{15}$ und der Ausgabe $A_0 \dots A_{15}$ lautet ($u = \frac{2\pi}{16}$):

$$\begin{array}{llll}
s_1 = f_0 + f_8 & s_2 = f_0 - f_8 & s_3 = f_4 + f_{12} & s_4 = f_4 - f_{12}\\
s_5 = f_2 + f_{10} & s_6 = f_2 - f_{10} & s_7 = f_6 + f_{14} & s_8 = f_6 - f_{14}\\
s_9 = f_1 + f_9 & s_{10} = f_1 - f_9 & s_{11} = f_5 + f_{14} & s_{12} = f_5 - f_{13}\\
s_{13} = f_3 + f_{11} & s_{14} = f_3 - f_{11} & s_{15} = f_7 + f_{15} & s_{16} = f_7 - f_{15}\\
s_{17} = s_1 + s_3 & s_{18} = s_1 - s_3 & s_{19} = s_5 + s_7 & s_{20} = s_5 - s_7\\
s_{21} = s_9 + s_{11} & s_{22} = s_9 - s_{11} & s_{23} = s_{13} + s_{15} & s_{24} = s_{13} - s_{15}\\
s_{25} = s_{17} + s_{19} & s_{26} = s_{17} - s_{19} & s_{27} = s_{21} + s_{23} & s_{28} = s_{21} - s_{23}\\
s_{29} = s_{25} + s_{27} & s_{30} = s_{25} - s_{27} & s_{31} = s_{22} + s_{24} & s_{32} = s_{22} - s_{24}\\
s_{33} = s_6 + s_8 & s_{34} = s_6 - s_8 & s_{35} = s_{10} + s_{16} & s_{36} = s_{10} - s_{16}\\
s_{37} = s_{12} + s_{14} & s_{38} = s_{12} - s_{14} & s_{39} = s_{35} + s_{37} & s_{40} = s_{36} + s_{38}
\end{array}$$

$$\begin{array}{llll}
m_1 = s_{29} & m_2 = s_{30} & m_3 = s_{26} & m_4 = s_{38} j \sin 4u\\
m_5 = s_{18} & m_6 = s_{20} j \sin 4u & m_7 = s_{31} j \sin 2u & m_8 = s_{32}\cos 2u\\
m_9 = s_2 & m_{10} = s_4 j \sin 4u & m_{11} = s_{33} j \sin 2u & m_{12} = s_{34}\cos 2u\\
m_{13} = s_{39} j \sin 3u & m_{14} = j(\sin u - \sin 3u)s_{35} & m_{15} = j(\sin u + \sin 3u)s_{37} & \\
m_{16} = s_{40}\cos 3u & m_{17} = (\cos u + \cos 3u)s_{36} & m_{18} = (\cos 3u - \cos u)s_{38} &
\end{array}$$

$s_{41} = m_3 + m_4 \quad s_{42} = m_3 - m_4 \quad s_{43} = m_5 + m_7 \quad s_{44} = m_5 - m_7$
$s_{45} = m_6 + m_8 \quad s_{46} = m_6 - m_8 \quad s_{47} = m_{43} + m_{45} \quad s_{48} = m_{43} - m_{45}$
$s_{49} = s_{44} + s_{46} \quad s_{50} = s_{44} - s_{46} \quad s_{51} = m_9 + m_{12} \quad s_{52} = m_9 - m_{12}$
$s_{53} = m_{10} + m_{11} \quad s_{54} = m_{10} - m_{11} \quad s_{55} = m_{13} + m_{14} \quad s_{56} = m_{13} - m_{15}$
$s_{57} = m_{17} - m_{16} \quad s_{58} = m_{18} - m_{16} \quad s_{59} = s_{51} + s_{55} \quad s_{60} = s_{51} - s_{55}$
$s_{61} = s_{52} + s_{56} \quad s_{62} = s_{52} - s_{56} \quad s_{63} = s_{53} + s_{57} \quad s_{64} = s_{53} - s_{57}$
$s_{65} = s_{54} + s_{58} \quad s_{66} = s_{54} - s_{58} \quad s_{67} = s_{59} + s_{63} \quad s_{68} = s_{59} - s_{63}$
$s_{69} = s_{60} + s_{64} \quad s_{70} = s_{60} - s_{64} \quad s_{71} = s_{61} + s_{65} \quad s_{72} = s_{61} - s_{65}$
$s_{73} = s_{62} + s_{66} \quad s_{74} = s_{62} - s_{66}$
$A_0 = m_1 \quad A_1 = s_{67} \quad A_2 = s_{47} \quad A_3 = s_{72} \quad A_4 = s_{41} \quad A_5 = s_{71}$
$A_6 = s_{48} \quad A_7 = s_{68} \quad A_8 = m_2 \quad A_9 = s_{69} \quad A_{10} = s_{49} \quad A_{11} = s_{74}$
$A_{12} = s_{42} \quad A_{13} = s_{73} \quad A_{14} = s_{50} \quad A_{15} = s_{70}$

4.5.3 Vervollständigung nach Arai, Agui, Nakajima

Nutzt man die Symmetrie der Eingabewerte aus und streicht alle Gleichungen, die nur zur Berechnung des Imaginärteils und der 8 unbenutzten Ausgabewerte A_8 bis A_{15} dienen, so können die benötigten Gleichungen nochmals erheblich reduziert werden. Hier sei gleich das Ergebnis anhand des Flußgraphen in Abbildung 4.3 gegeben. Dabei sind Additionspunkte dunkel gezeichnet, eine gestrichelte Linie entspricht der Negation des Wertes und die Kästen stehen für Multiplikation mit einem konstanten Faktor.

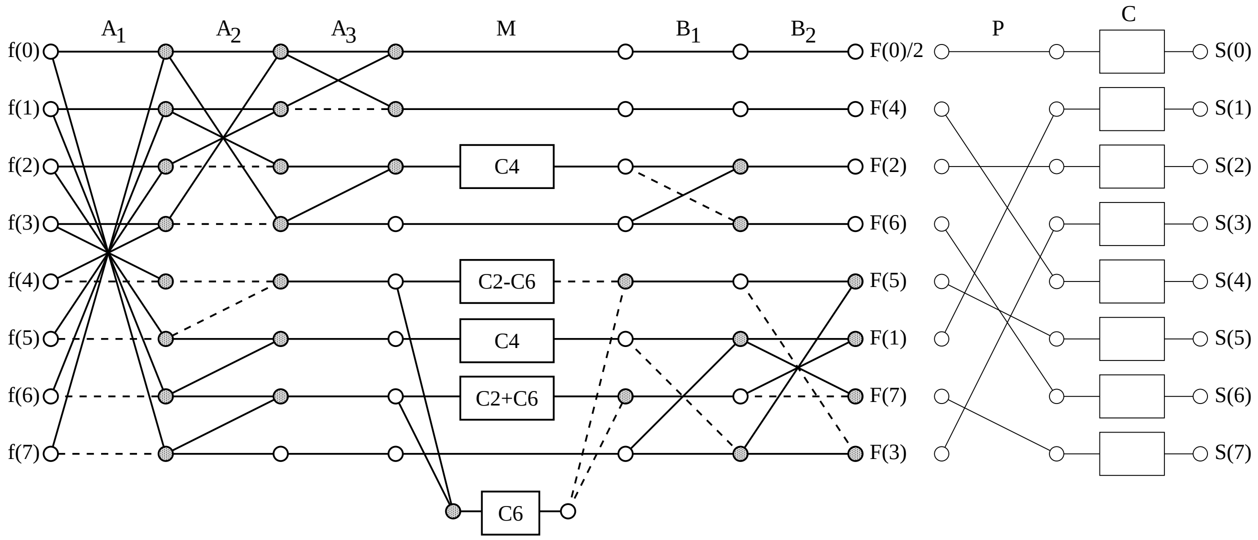


Abbildung 4.3: Flußgraph für 8-Punkt DCT nach Arai, Agui, Nakajima

Die ersten 6 Stufen berechnen die DFT Koeffizienten. Danach folgt eine Permutationsstufe, die ohne Rechenaufwand durch Umbenennen der Variablen erfolgen kann. Schließlich folgt noch die Skalierung zu den DCT-Koeffizienten, ggf. mit gleichzeitiger Quantisierung.

Nach diesem Algorithmus sind noch 13 Multiplikationen pro 8-Punkt DCT nötig. Würde man dies nun mittels des Row-Column-Ansatzes zur 8×8-Punkt DCT erweitern, so wären für diese nur noch $(5 + 8) \cdot 8 \cdot 2 - 64 = 144$ Multiplikationen nötig, da die letzten Multiplikationen in der Quantisierung aufgehen.

An dieser Stelle sei darauf hingewiesen, daß dieser Flußgraph als Faktorisierung der Transformationsmatrix aufgefaßt werden kann. Jeder Berechnungsstufe entspricht dabei ein Faktor, so daß sich folgende Faktorisierung ergibt:

$$\mathbf{D} \cdot \mathbf{s} = \mathbf{CPB_2B_1MA_3A_2A_1} \cdot \mathbf{s}$$

Als Beispiel sei hier der Inhalt der Matrizen $\mathbf{A_1}$ und $\mathbf{M}$ angegeben:

$$\mathbf{A_1} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \quad \mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & C_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -C_2 & 0 & -C_6 & 0 \\ 0 & 0 & 0 & 0 & 0 & C_4 & 0 & 0 \\ 0 & 0 & 0 & 0 & -C_6 & 0 & C_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Man sieht, daß obiger Algorithmus die Matrixmultiplikation von $\mathbf{M}$ nicht naiv ausführt, sondern durch eine abweichende Klammerung der Terme mit 5 statt 6 Multiplikationen auskommt, da

$$s_4' = -C_2 s_4 - C_6 s_6 = -C_2 s_4 + C_6 s_4 - C_6 s_4 - C_6 s_6 = -(C_2 - C_6) s_4 - C_6(s_4 + s_6)$$

und

$$s_6' = -C_6 s_4 + C_2 s_6 = -C_6 s_4 + C_6 s_6 - C_6 s_6 + C_2 s_6 = (C_2 + C_6) s_6 - C_6(s_4 + s_6)$$

beide den gleichen Term $C_6(s_4 + s_6)$ enthalten.

Beim Übergang zur zweidimensionalen DCT können wir nochmals einige Operationen einsparen. Sei dazu $\mathbf{D'} = \mathbf{PB_2B_1MA_3A_2A_1}$, also $\mathbf{D}$ ohne die Skalierungsstufe, so können wir das Distributivgesetz der Tensormultiplikation anwenden und erhalten

$$\mathbf{D} \otimes \mathbf{D} = (\mathbf{C} \cdot \mathbf{D'}) \otimes (\mathbf{C} \cdot \mathbf{D'}) = (\mathbf{C} \otimes \mathbf{C}) \cdot (\mathbf{D'} \otimes \mathbf{D'})$$

Da $\mathbf{C}$ eine Diagonalmatrix ist, ist auch $\mathbf{C} \otimes \mathbf{C}$ eine Diagonalmatrix. Unser Vorgehen zur Berechnung der zweidimensionalen DCT ist nun, das Produkt $\mathbf{D'} \otimes \mathbf{D'}$ durch den Row-Column-Ansatz zu berechnen (wozu $2 \cdot 8 \cdot 5 = 80$ Multiplikationen benötigt werden) und danach alle 64 Ausgangswerte über die Diagonalmatrix $\mathbf{C} \otimes \mathbf{C}$ zu skalieren, was wieder mit der Quantisierung verknüpft werden kann. Insgesamt werden für die DCT dann nur noch 80 Multiplikationen benötigt.

Es sei angemerkt, daß sich aus obigem Flußgraph auch direkt ein Algorithmus für die IDCT ablesen läßt. Wie wir in Gleichung (4.1) festgestellt haben, genügt es, die Transformationsmatrizen zu transponieren. Dies entspricht aber genau dem Spiegeln des Flußgraphen an einer Vertikalen. Man muß also den Flußgraphen nur umdrehen und hat damit den Algorithmus für die IDCT.

4.5.4 Integer-Implementierung

Da Integerberechnungen i.A. schneller auszuführen sind, als Floating-Point-Berechnungen, ist man bestrebt, auch bei der DCT-Berechnung Integer- bzw. Fixed-Point-Arithmetik zu benutzen. Da in der Berechnung irrationale Zahlen eingehen, entsteht die Gefahr einer zu geringen Genauigkeit im Vergleich zu der Floating-Point-Implementierung. Deshalb ist es wichtig, soviele Nachkommastellen wie möglich zu beücksichtigen.

Betrachten wir zuerst die FDCT mit ihrem Flußgraphen in Abbildung 4.4. An jedem Knoten ist nun noch die maximal benötigte Vorkommabitbreite angegeben. Die Eingabewerte benötigen

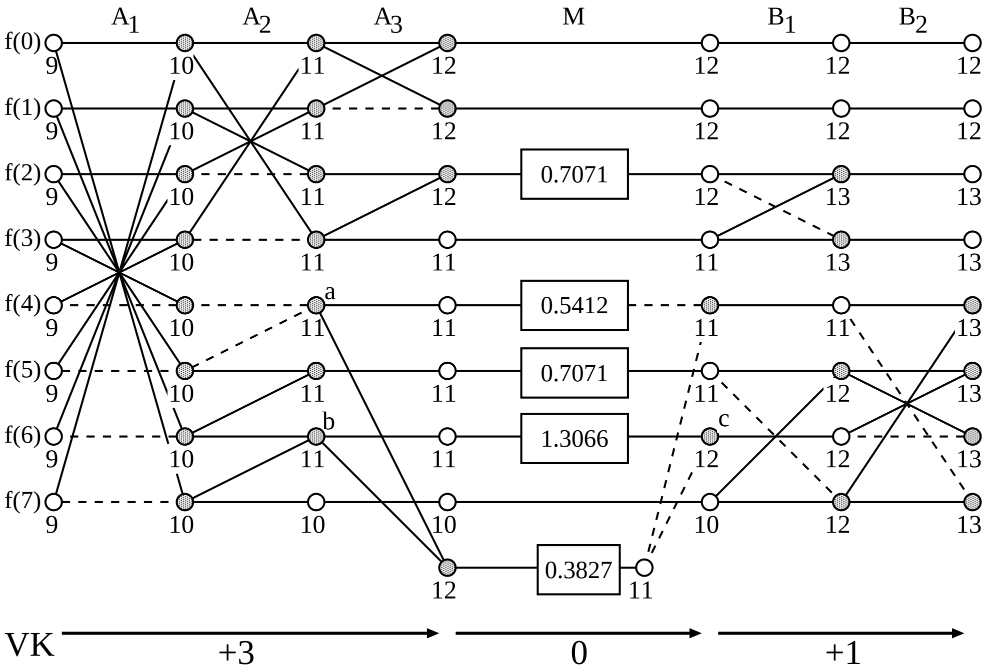


Abbildung 4.4: Flußgraph für 1D-FDCT mit benötigten Vorkommastellen

9 Bits (8 Bit + Vorzeichen) und bei jeder Addition bzw. Subtraktion kann sich die benötigte Breite um 1 Bit erhöhen. Die Addition des Eingabewertes für die "Umleitung" wurde in diesem Flußgraphen eine Stufe nach vorne geschoben, um zu verdeutlichen, daß dadurch nicht mehr Bits eingeführt werden, als in den anderen Reihen. Bis auf einen sind alle Multiplikationsfaktoren der M-Stufe mit Faktoren < 1, so daß sich die Anzahl der Vorkommabitbreite nicht vergrößert. Nicht ganz so offensichtlich ist lediglich, daß auch an Punkt c sich die Bitbreite nur um eins erhöht, obwohl an diesem Punkt sowohl eine Addition als auch direkt davor eine Multiplikation mit einem Faktor > 1 stattfindet. Der Wert an dieser Stelle berechnet sich zu:

$$c \approx 1.3066 \cdot b - (a + b) \cdot 0.3827 = 0.9239 \cdot b - 0.3827 \cdot a$$

Dieser Wert ist maximal für $a = -b$ und beträgt dann

$$c \approx 1.3066 \cdot b$$

und benötigt somit höchstens ein Vorkommabit mehr als b.

Betrachten wir nun den gesamten 2D-Algorithmus. Der Flußgraph des vollständigen 2D-Algorithmus ist in Abbildung 4.5 schematisch dargestellt. Es sind die zwei hintereinandergeschalteten 1D-Flußgraphen mit den Multiplikationsstufen M_1 und M_2 und die hinten kombinierte Skalierungsstufe C hervorgehoben. Direkt darunter sind in der VK-Zeile die benötigten Vorkommabitbreiten angegeben, die sich aus obiger Überlegung ergeben. Direkt vor der abschließenden Skalierungsstufe C beträgt diese 17 Bit. In der Skalierungsstufe treten wieder bis auf eine Ausnahme nur Faktoren < 1 auf, so daß sich die Bitbreite nicht erhöht. Die Ausnahme ist der Faktor für den Koeffizienten mit $u = v = 7$, der etwa 1.64 beträgt. Diesen Wert ignorieren wir aber mit der Rechtfertigung, daß die Abschätzung der Bitbreite für den untersten Datenpfad im Flußgraphen, der für diesen Wert ja zweimal durchlaufen wird, noch etwas Spielraum offen läßt (z.B. an der Addition in Stufe B_1 wird

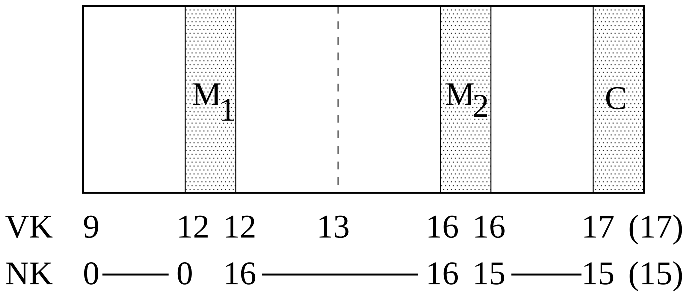


Abbildung 4.5: Schematischer Flußgraph für 2D-FDCT mit Angabe der Vorkomma- und Nachkommabitbreite

lediglich eine 10 mit einer 11 Bit breiten Zahl addiert, die zuvor auch noch mit $\sqrt{\frac{1}{2}}$ multipliziert wurde). Andererseits ist es extrem unwahrscheinlich, daß die maximale Frequenz mit fast maximaler Intensität im Bild auftritt. Somit erhalten wir auch eine Vorkommabitbreite von 17 Bits nach der Skalierungsstufe. Damit scheidet eine Implementierung mit 16 Bit-Integers von vorneherein aus. Somit wählen wir eine Implementierung mit 32 Bit Breite.

Damit wir nicht zu viele Shift-Operationen zur Reduzierung der Nachkommabitbreite benötigen, halten wir die Zahl der Nachkommabits während der Additionsstufen konstant. Da die maximale Vorkommabitbreite am Ende der Additionsstufen auftritt, ist diese für die Anzahl der Nachkommabits maßgeblich. In Abbildung 4.5 sind in der Zeile „NK“ die sich ergebenden Nachkommabitbreiten bei 32 Bit-Integers eingetragen. Die Breite 0 zu Beginn ergibt sich einfach daraus, daß die Eingabe aus ganzen Zahlen besteht und die ersten Nachkommastellen erst bei der ersten Multiplikation eingeführt werden.

Auch die Multiplikationsfaktoren werden auf 32 Bit Breite gebracht, indem sie mit 2^{30} multipliziert werden, bevor sie nach Integer umgewandelt werden. Das entspricht dann einer Vorkomma- und 30 Nachkommastellen, was nötig ist, um auch die größten Faktoren (< 2) darstellen zu können.

IDCT

Das Vorgehen für die IDCT entspricht im wesentlichen dem bei der FDCT und soll hier nicht wiederholt werden. Die entsprechenden Diagramme seien aber in Abbildung 4.6 und 4.7 angegeben.

Genauigkeit der Integer-Implementierungen

Die Genauigkeit der IDCT Implementierung wurde mit dem Testprogramm von Tom Lane überprüft und erfüllt die Vorgaben aus IEEE 1180. Die Ergebnisse sind in Anhang A aufgeführt.

4.6 Algorithmus nach Feig

Der Algorithmus nach Feig, Winograd [FW92] basiert auf der Faktorisierung der Transformationsmatrix nach Arai, Agui, Nakajima. Er erreicht eine weitere Reduktion der Multiplikationen durch einen direkten zweidimensionalen Ansatz in der Multiplikationsstufe. Betrachten wir dazu den Übergang von der 1D- zur 2D-Transformation über das Tensorprodukt:

$$\mathbf{D} \otimes \mathbf{D} = \mathbf{C P B_2 B_1 M A_3 A_2 A_1} \otimes \mathbf{C P B_2 B_1 M A_3 A_2 A_1}$$

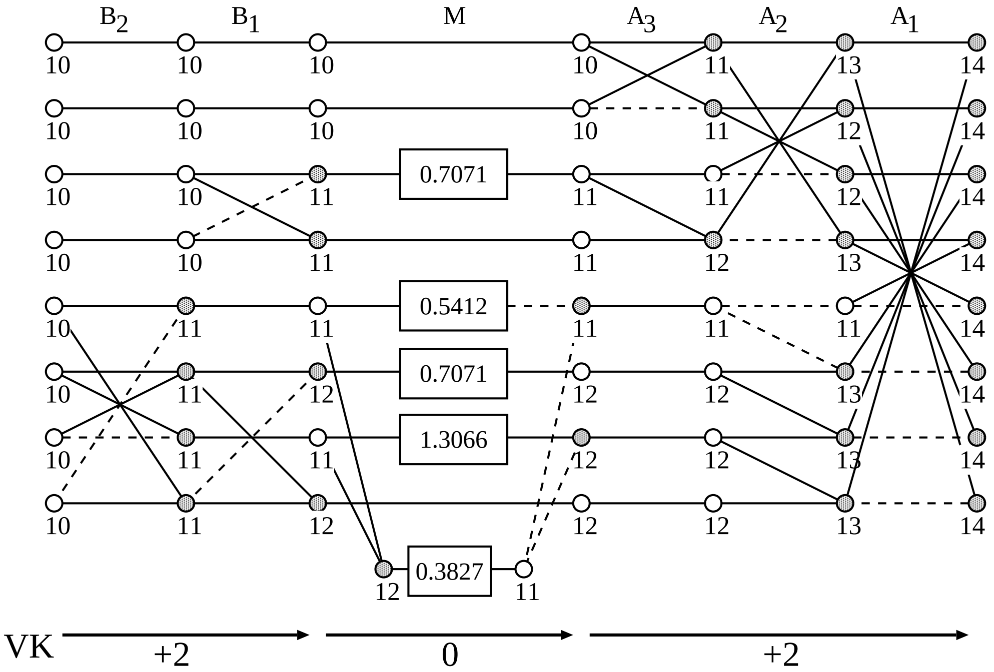


Abbildung 4.6: Flußgraph für 1D-IDCT mit benötigten Vorkommastellen

Für jede Stufe haben wir zwei Möglichkeiten. Wir können die Stufe mittels dem Row-Column-Ansatz berechnen, oder zuvor das Distributivgesetz der Tensormultiplikation anwenden, so wie wir es oben schon für die Skalierungsstufen gemacht haben. Da in den Additionsstufen keine Multiplikationen auftreten, ist hier eine Optimierung nicht unbedingt erforderlich und wir werten sie deshalb im Row-Column-Verfahren aus. Nur die Multiplikationsstufe multiplizieren wir aus:

$$\mathbf{D} \otimes \mathbf{D} = \underbrace{(\mathbf{C} \otimes \mathbf{C})}_{ausmult.} \underbrace{(\mathbf{PB_2B_1} \otimes \mathbf{PB_2B_1})}_{Row-Column} \underbrace{(\mathbf{M} \otimes \mathbf{M})}_{ausmult.} \underbrace{(\mathbf{A_3A_2A_1} \otimes \mathbf{A_3A_2A_1})}_{Row-Column}$$

Es zeigt sich, daß dadurch in der 2D-Multiplikationsstufe statt vorher $8 \cdot 2 \cdot 5 = 80$ Multiplikationen nur noch 54 Multiplikationen und 6 Multiplikationen mit $\frac{1}{2}$ (was bei Integer-Implementierung durch ein billiges Bitshiften nach rechts zu erreichen ist) benötigt werden. Das Ergebnis ist in [FW92] oder in [PM93] nachzulesen.

4.7 Algorithmus nach Cho, Lee

Der Ansatz dieses Algorithmus ist ein völlig anderer als bei den bisher besprochenen. Cho und Lee zeigen in [CL91], daß es möglich ist, ein 2D-DCT aus N 1D-DCTs zu berechnen anstatt aus $2N$, wie bei dem Row-Column-Ansatz. Die Grundidee ist, die Beziehung

$$\cos\frac{(2i+1)u\pi}{2N}\cos\frac{(2j+1)v\pi}{2N} = \frac{1}{2}\Big(\cos\frac{(2i+1)u+(2j+1)v}{2N}\pi + \cos\frac{(2i+1)u-(2j+1)v}{2N}\pi\Big)$$

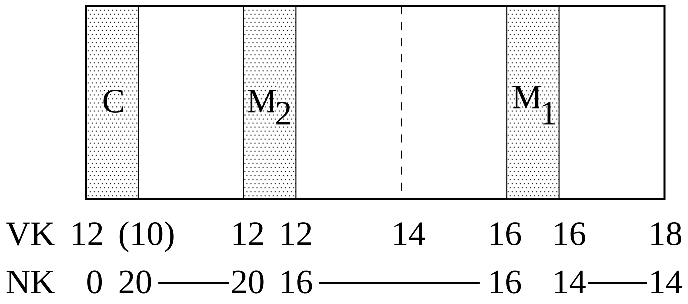


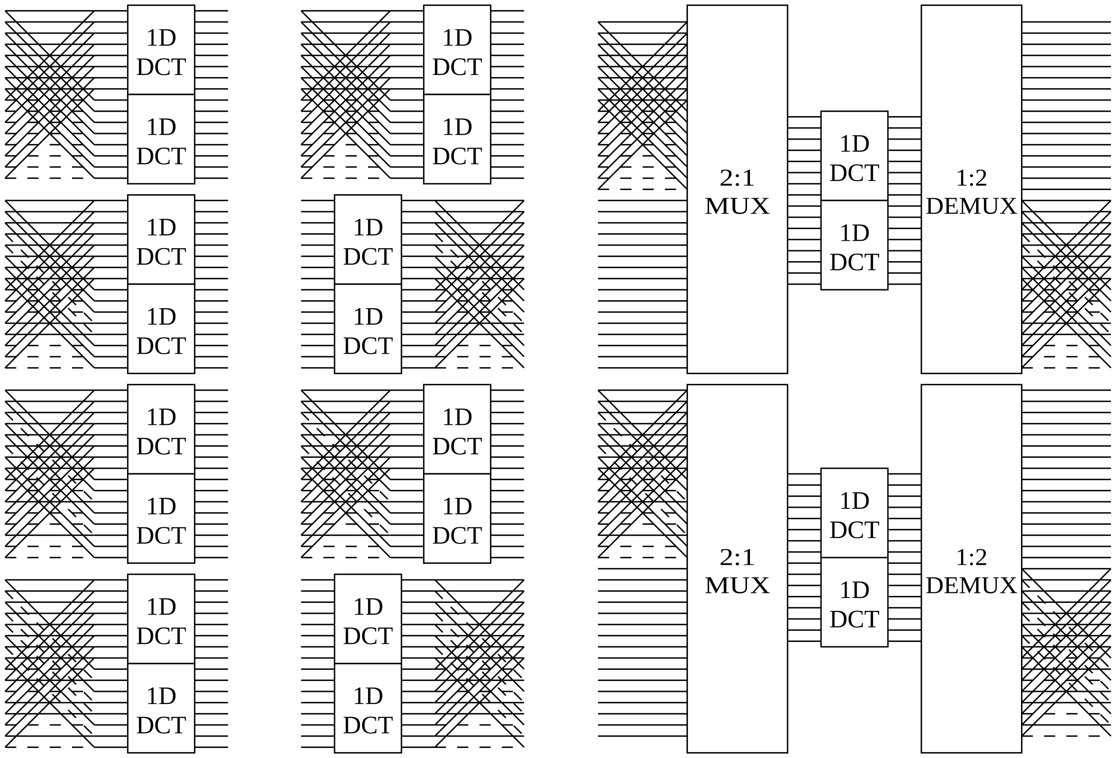
Abbildung 4.7: Schematischer Flußgraph für 2D-IDCT mit Angabe der Vorkomma- und Nachkommabitbreite

zu benutzen und damit die 2D-DCT so darzustellen:

$$S(u,v) = \frac{1}{2}\left(\underbrace{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} s(u,v)\cos\frac{(2i+1)u+(2j+1)v}{2N}\pi}_{A(u,v)} + \underbrace{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} s(u,v)\cos\frac{(2i+1)u-(2j+1)v}{2N}\pi}_{B(u,v)}\right)$$

Sie zeigen nun, daß $A(u,v)$ und $B(u,v)$ beide mit insgesamt nur N 1D-DCTs berechnet werden können. Diese Herleitung ist sehr aufwendig und soll hier nicht dargestellt werden. Ein Teil des Ergebnisses ist in Abbildung 4.8(a) als Flußgraph zu sehen. Die Eingabewerte auf der linken Seite werden jedoch nicht in der Reihenfolge angelegt, wie man vielleicht annehmen könnte, sondern ziemlich gut vermischt. Nach der gezeigten DCT-Stufe wird die Ausgabe noch durch 3 (auch ziemlich wilde) Additionsstufen geschickt und fast alle Werte müssen daraufhin noch eine Position nach rechts geshifted werden.

Das schöne an diesem Algorithmus ist, daß er sich in Hardware mit sehr wenig Aufwand realisieren läßt, wenn man ihn noch etwas umformt. Wegen der Linearität der DCT-Transformation läßt sich die DCT-Transformation und die Additionen davor vertauschen (siehe Abbildung 4.8(b)). Da die DCT-Stufen auch als Pipeline implementiert werden können, kann die nächste Eingabe schon angelegt werden, wenn die letzte Eingabe noch nicht berechnet worden ist. Deswegen kann man je zwei DCT-Blöcke durch einen einzigen ersetzen. Die Daten werden dann über Multiplexer, Demultiplexer Paar durch diese geführt. Dies ist dann nur unwesentlich langsamer (nämlich genau eine Additionsstufe) als ursprünglich, benötigt dafür aber nur noch 4 statt 8 DCT-Blöcke.



(a) (b) (c)

Abbildung 4.8: Algorithmus nach Cho, Lee (Ausschnitt)

Kapitel 5

Motionvektor-Suche

Algorithmen zur Motionvektor-Suche können anhand dreier Merkmale klassifiziert werden: das verwendete *Abstandsmaß*, der *Suchalgorithmus* und der *Suchbereich*. Gelegentlich sind Suchalgorithmus und Suchbereich auch aneinander gebunden, wie z.B. bei der Grundversion von TSS und NTSS (siehe Abschnitt 5.3.5 und 5.3.6).

5.1 Abstandsmaße

Um überhaupt nach einem „besten" Motionvektor suchen zu können, müssen wir erst einmal definieren, was wir als den Abstand zwischen zwei Blöcken verstehen. Das Idealziel wäre, direkt die Anzahl der zur Codierung der Differenzinformationen, plus den für den Motionvektor benötigten Bits, als Abstand zu benutzen. Das scheidet jedoch aufgrund des extremen Rechenbedarfs für praktische Zwecke aus.

Wir benötigen deshalb eine relativ einfach zu berechnende Funktion $\mathbb{N}^{256} \times \mathbb{N}^{256} \to \mathbb{R}$, die einem Referenzblock und einem Bildblock einen „Abstand" zuweist, der eine möglichst monoton ansteigende Folge bildet, ordnet man die Eingabe nach aufsteigender Zahl der benötigten Bits zur Codierung. Wir betrachten lediglich den Helligkeitskanal, da die Farbkanäle aufgrund der anderen Samplingfrequenz nicht ganz einfach zu berücksichtigen sind. Eine Verbesserung würde sich nur an Stellen ergeben, die die gleiche Helligkeitsstruktur in unterschiedlichen Farben haben. Dies ist aber auf den recht kleinen Suchbereichen ziemlich unwahrscheinlich.

Ein häufig benutzter Abstand ist der *mean square error* (MSE), der (wenn $f(x, y)$ das Originalbild, $\hat{f}(x, y)$ das Referenzbild, $(dx; dy)$ der Motionvektor und $(x_0; y_0)$ die linke obere Ecke des Blocks im Originalbild ist) folgendermaßen definiert ist:

$$MSE(x_0, y_0, dx, dy) = \frac{1}{256} \sum_{x=0}^{15} \sum_{y=0}^{15} (f(x + x_0, y + y_0) - \hat{f}(x + x_0 + dx, y + y_0 + dy))^2$$

Dieser Abstand hat jedoch aufgrund der auftretenden Multiplikationen immer noch einen für praktische Zwecke, insbesondere für die Hardwarerealisierung, recht hohen Rechenaufwand. Deswegen wird in der Praxis meistens die Quadrierung durch die einfachere Betragswertfunktion ersetzt und dieser Abstand dann als *mean absolute difference* (MAD) bezeichnet[1]:

[1] Teilweise liest man auch die Bezeichnung *Mean Absolute Error* (MAE).

$$MAD(x_0, y_0, dx, dy) = \frac{1}{256} \sum_{x=0}^{15} \sum_{y=0}^{15} |f(x + x_0, y + y_0) - \hat{f}(x + x_0 + dx, y + y_0 + dy)|$$

Der konstante Normierungsfaktor 1/256 ist für die Optimierung unwesentlich und kann einfach weggelassen werden. Manchmal spricht man dann auch von der *sum of absolute differences* (SAD). Gelegentlich ist es im Sinne einer einheitlichen Darstellung vorteilhaft, daß man MSE und MAD verallgemeinert durch $\sum |\cdots|^p$ schreiben kann. MAD ergibt sich dabei für $p = 1$ und MSE für $p = 2$.

Ein sehr selten aufgeführtes Maß ist die *Cross-Correlation Function* (CCF), die aufgrund des hohen Berechnungsaufwands unattraktiv für die praktische Anwendung ist:

$$CCF(x_0, y_0, dx, dy) = \frac{\sum_{x=0}^{15} \sum_{y=0}^{15} f(x + x_0, y + y_0) \cdot \hat{f}(x + x_0 + dx, y + y_0 + dy)}{\left[\sum_{x=0}^{15} \sum_{y=0}^{15} f^2(x + x_0, y + y_0)\right]^{1/2} \left[\sum_{x=0}^{15} \sum_{y=0}^{15} \hat{f}^2(x + x_0 + dx, y + y_0 + dy)\right]^{1/2}}$$

Da es vermehrt Hardwareimplementationen für Blockvergleiche gibt, wurden auch spezielle Abstandsmaße entwickelt, die sich besonders gut in Hardware realisieren lassen. Dazu zählen die *Pel Difference Classification* (PDC):

$$PDC(x_0, y_0, dx, dy) = \sum_{x=0}^{15} \sum_{y=0}^{15} T(x_0 + x, y_0 + y, dx, dy)$$

mit

$$T(x', y', dx, dy) = \begin{cases} 1 & |f(x', y') - \hat{f}(x' + dx, y' + dy)| < Threshold \\ 0 & sonst \end{cases}$$

oder auch der *Minimized Maximum Error* (MiniMax):

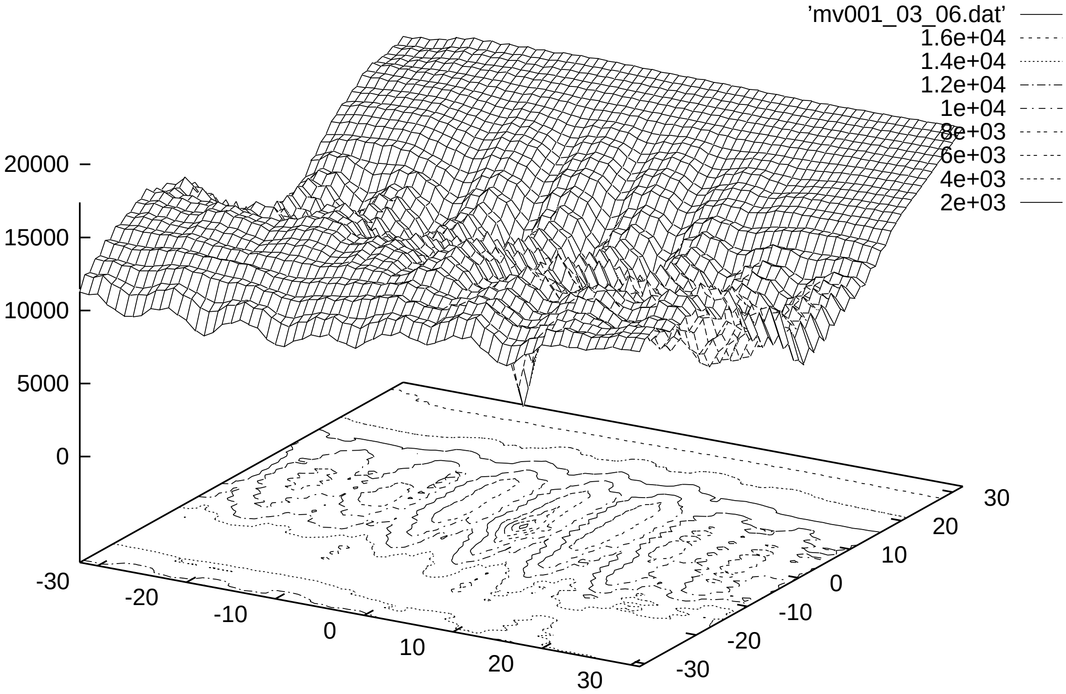
$$MiniMax(x_0, y_0, dx, dy) = \max_{0 \leq x, y \leq 15} |f(x + x_0, y + y_0) - \hat{f}(x + x_0 + dx, y + y_0 + dy)|$$

Das MiniMax-Maß hat gegenüber dem PDC den Vorteil, daß das Ergebnis nicht von der optimalen Wahl eines Thresholds, der sich bei verschiedenen Bildern auch unterscheiden kann, abhängt. Das hardwareorientierte MiniMax-Maß erreicht eine ähnlich hohe Qualität wie die MAD (PSNR des Differenzbildes ist ca. um 0.2-1 dB niedriger), die Hardwarerealisierung benötigt jedoch 16% weniger Chipfläche, was den geringeren PSNR durch mehr Suchpositionen mehr als ausgleichen kann.

5.2 Beispiele von Suchräumen

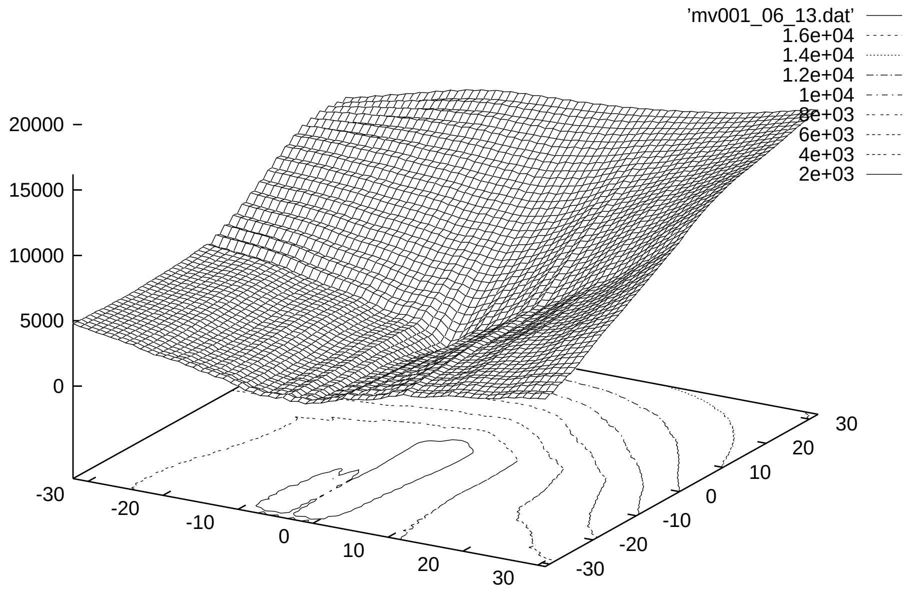
Um uns eine genauere Vorstellung von dem Suchraum zu machen, betrachten wir die in Abbildung 5.1(a) dargestellte Szene. Für die drei in ihr markierten Makroblöcke wurde eine volle Motionvektorsuche mit SAD-Maß in Half-Pel-Genauigkeit mit dem Suchbereich von horizontal und vertikal jeweils $[-32; 31]$ Half-Pels vorgenommen. Das entspricht somit der Verschiebung entsprechend 16 Pixel, also der Größe eines Makroblocks. Der für jeden möglichen Motionvektor ermittelte

(a) Originalbild

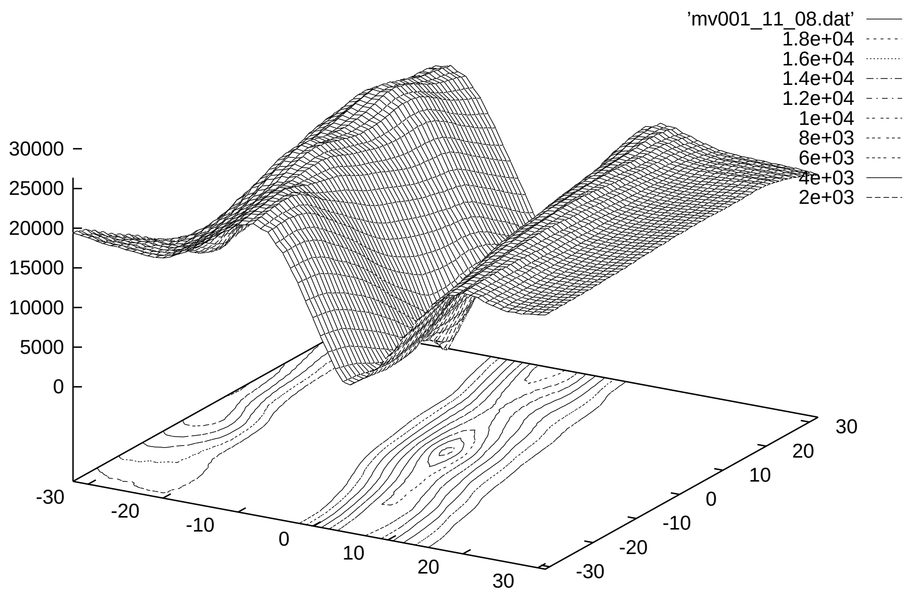


(b) MV-Fehler für Block 03/06

Abbildung 5.1: Originalbild und Fehlerverteilung im Suchraum von Block 03/06



(a) MV-Fehler für Block 06/13



(b) MV-Fehler für Block 11/08

Abbildung 5.2: Fehlerverteilung im Suchraum von Block 06/13 und Block 11/08

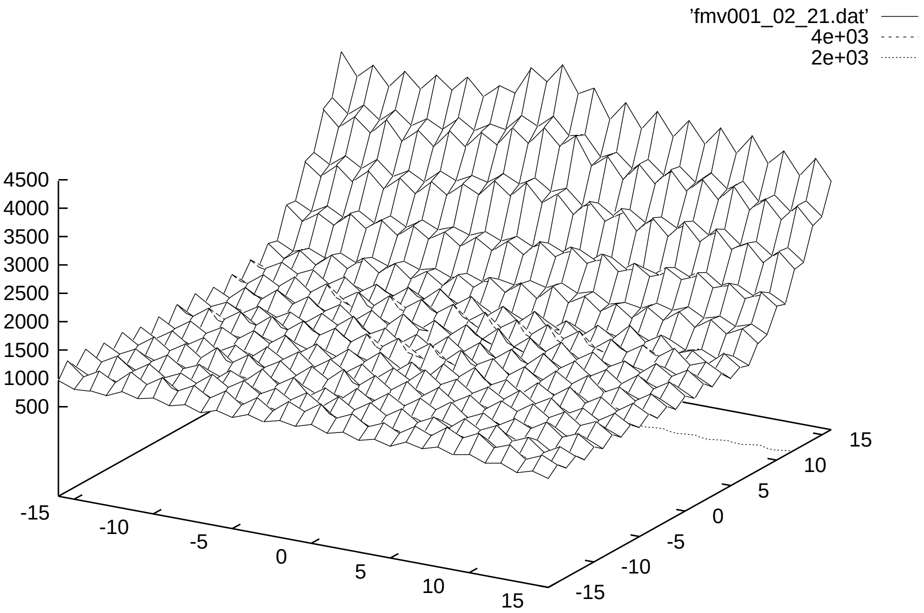


Abbildung 5.3: Beispiel mit deutlich niedrigeren Half-Pel-Fehlern

Fehler ist in den Abbildungen 5.1(b) und 5.2 dargestellt. Der jeweils optimale Motionvektor ist in allen drei Fällen der Nullvektor.

Betrachten wir zunächst Block 06/13. Dieser enthält zufälligerweise ziemlich genau die rechte obere Ecke einer kaum strukturierten Fläche. In der Darstellung des Fehlers in Abhängigkeit des Motionvektors in Abbildung 5.2(a) sieht man, wie der Fehler immer mehr anwächst, je weiter der Block über die Grenze der Flächen bewegt wird. Eine Bewegung innerhalb der eigenen Fläche (nach links unten) bewirkt dagegen nur ein viel langsameres Anwachsen des Fehlers.

Betrachten wir nun Block 11/08, eine scharfe Kante zwischen Schwarz und Weiß. In der Fehlerverteilung in Abbildung 5.2(b) sieht man dann auch, daß der Fehler bei einer Bewegung senkrecht zur Kante extrem stark ansteigt, entlang der Kante aber nur weit weniger schwankt. Zu erkennen ist auch, daß sich die beiden schwarzen Ecken links unten und oben (beides Ausschnitte der Krawatte) sich auch etwas weiter oben befinden (Kinn und Krawatte). Wegen dieser Ähnlichkeit befindet sich am Rand des Suchbereichs (bei maximaler vertikaler Verschiebung) nochmals ein Minimum.

Ein ähnlicher Effekt, nur diesmal viel gehäufter ist auch in Abbildung 5.1(b) zu sehen. Hier reihen sich viele Minima in horizontaler Richtung, da sich die Buchstabenmuster sehr ähnlich sehen (das „i“ kommt gut mit dem „l“ zur Deckung und auch die „o“s sind diesen nicht unähnlich).

5.2.1 Suche mit Half-Pel-Auflösung

Bei einer Suche in Half-Pel-Auflösung werden, wenn der Motionvektor nicht ganzzahlig ist, die Pixelwerte von zwei oder vier benachbarten Pixel gemittelt. Dadurch wird das Rauschen reduziert, was wiederum bewirkt, daß dies eine bessere Prädiktion liefert. In Abbildung 5.3 ist das sehr deutlich zu sehen. Eine Konsequenz daraus ist, daß man ein Gradientenabstiegsverfahren zur Suche nicht direkt mit der Half-Pel-Auflösung durchführen kann, da fast regelmäßig alle 2×2-Pixel ein lokales Minimum auftritt. Eine Standardmethode ist deshalb, die Motionvektorsuche nur in Full-Pel-Auflösung durchzuführen und die 8 Half-Pel-Vektoren um den gefundenen Motionvektor danach auf eine bessere Lösung zu untersuchen (Abbildung 5.4).

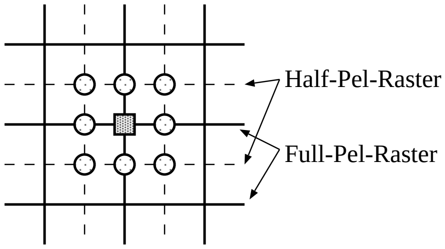


Abbildung 5.4: Pixel an Full-Pel Position und seine 8 umgebenden Pixel an Half-Pel-Positionen

5.3 Blockbasierte Suchverfahren

Wir betrachten im folgenden immer die Suche für Blöcke der Größe 16×16 Pels und den Suchbereich $[-H; H], [-V; V]$. H und V sind meist Potenzen von 2. Der Einfachheit halber bezeichnen wir H und V in HalfPel-Einheiten und vernachlässigen, daß der Suchbereich in positive Richtung aufgrund der Codierung um eine Einheit kleiner ist. D.h. ein Suchbereich von $H = 16$ entspricht einem realen Pixelbereich von $dx \in [-8; 7, 5]$.

Die in SAMPEG implementierten Suchverfahren sind im Titel mit einem „■“ markiert, im Gegensatz zu den mit „□“ markierten.

5.3.1 Full Search (2DFS) ■

Das offensichtlichste Suchverfahren, das auch die besten Motionvektoren findet, ist die *volle Suche*. Der Fehler für jeden möglichen Vektor wird berechnet und der minimale gewählt. Dieses sehr aufwendige Verfahren läßt sich etwas beschleunigen, wenn die Berechnung des Fehlers schon dann abgebrochen wird, wenn die Zwischensumme den bisher minimalen Fehler überschritten hat. Allerdings funktioniert das nur, wenn sämtliche Summanden nichtnegativ sind und die Fehlersumme somit monoton steigt. Dies ist z.B. bei MSE und MAD der Fall. Unter der Annahme, daß der beste Motionvektor oft in der Nähe des 0-Vektors liegt, ist es dann vorteilhaft, den Suchbereich ausgehend vom Zentrum z.B. in einer Spirale abzusuchen. SAMPEG sucht zwar streng von links oben nach rechts unten, da aber vor jeder eigentlichen Motionvektorsuche der Vektor getestet wird, der auch im letzten Makroblock benutzt wurde (und der auch im aktuellen meist in der Nähe der Optimimums liegt) erhält man dadurch schon eine sehr gute Fehlerabschätzung.

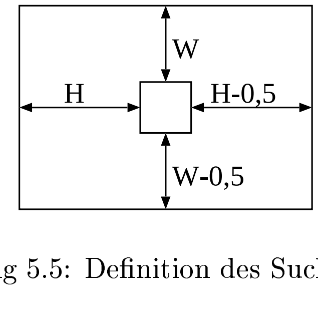


Abbildung 5.5: Definition des Suchbereiches

Oft beschränkt man die *Full-Search* Suche nur auf die ganzzahligen Motionvektoren und berechnet danach nur die 8 Halfpel-Vektoren um den besten ganzzahligen Motionvektor.

5.3.2 One At A Time Search (OTS) ■

Diese Methode sucht zuerst ausgehend vom Ursprung das Minimum in horizontaler Richtung. Um festzustellen, in welcher Richtung das Minimum liegt, werden dazu zuerst die beiden Positionen links und rechts neben dem Ursprung untersucht (Positionen „1“ in Abbildung 5.6). In dieser Richtung wird daraufhin weitergegangen, bis ein (vielleicht auch nur lokales) Minimum erreicht ist (Position „2“). Nun wird genau der gleiche Vorgang noch einmal in der Vertikalen wiederholt. Das diesmal gefundene Minimum ist dann das Ergebnis der Suche.

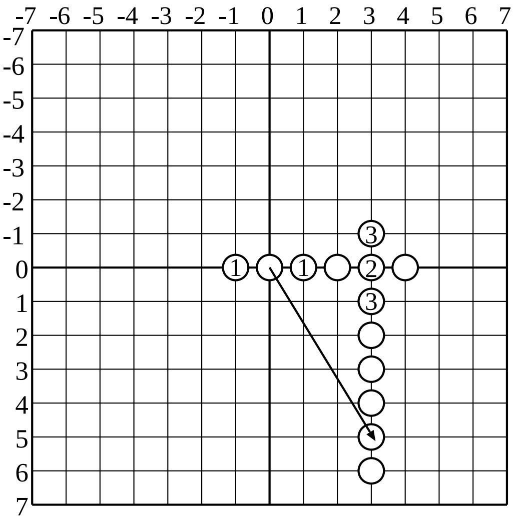


Abbildung 5.6: „One at a time search

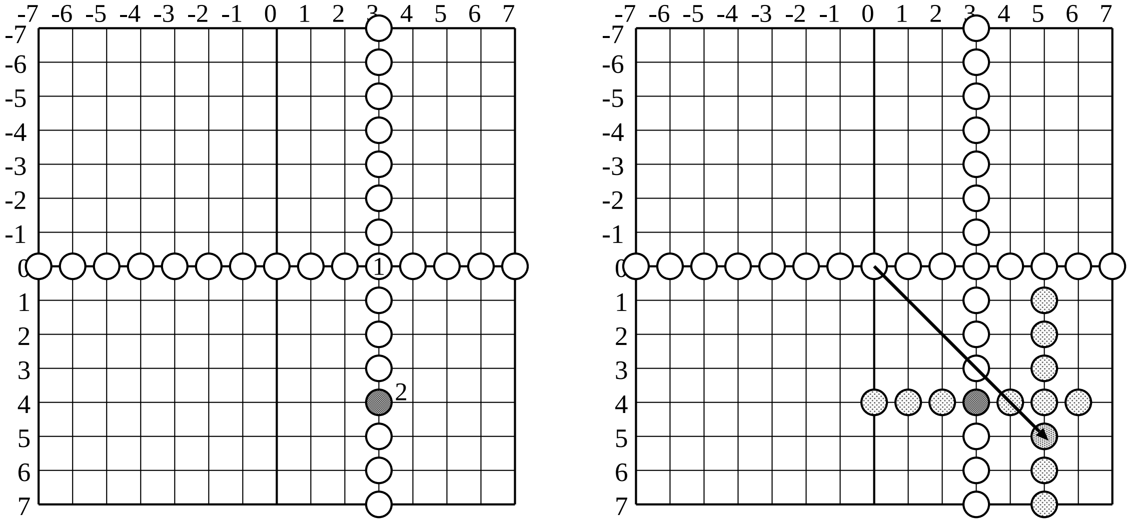
Der Nachteil dieses Verfahrens ist, daß es sehr anfällig ist, sich in lokalen Minima festzusetzen, aus denen es nicht mehr herausfinden kann. Aus diesem Grund kann die Suche nicht direkt in Half-Pel Genauigkeit durchgeführt werden, denn durch die leicht geringeren Fehler an den Half-Pel Positionen würde OTS gleich darin anhalten. Es muß also in Full-Pel Auflösung gesucht werden und erst in einem zweiten Schritt können dann die 8 Half-Pel-Positionen daneben auf geringere Fehler untersucht werden.

5.3.3 One Dimensional Full Search (1DFS) ■

Aufgrund der Probleme der OTS mit lokalen Minima wurde in [CCC94] ein etwas erweiterter Algorithmus vorgeschlagen. Auch hier wird zuerst das Minimum in horizontaler Richtung gesucht. Dabei wird aber der ganze horizontale Bereich berechnet. An der Position des Minimums wird das in der auch im vollen Suchbereich Vertikalen wiederholt (Abbildung 5.7(a)). In einem zweiten Schritt wird nun ausgehend von dem bisher gefundenen Minimum das Ganze wiederholt, nur diesmal mit halbiertem Suchbereich (Abbildung 5.7(b)). Dadurch, daß immer der ganze Bereich abgesucht wird, sinkt die Gefahr, in ein lokales Minimum zu geraten.

5.3.4 Block Based Gradient Descent Search (BBGDS) □

Ein Verfahren, das die Suche als Gradientenabstiegsverfahren durchführt, wird in [LF96] beschrieben. Ausgehend vom Zentrum werden die Fehler des Zentrums und der 8 direkten Nachbarn berechnet. Befindet sich das Minimum in der Mitte, so wird gestoppt. Befindet sich hingegen das Minimum an einem Randpunkt, so wird wiederum das Minimum der 8-Nachbarschaft bestimmt. Das wird so lange wiederholt, bis das Minimum in der Mitte liegt. Man beachte, daß bei der Verschiebung des betrachteten Bereiches nicht alle Punkte neu berechnet werden müssen. Sie müssen im nächsten Schritt auch nicht in die Suche nach dem Minimum einbezogen werden, da sie ja nicht kleiner sein können, als die aktuelle Position, denn sonst wären sie in einem früheren Schritt schon ausgewählt worden.



(a) Erster Schritt

(b) Zweiter Schritt

Abbildung 5.7: One Dimensional Full Search

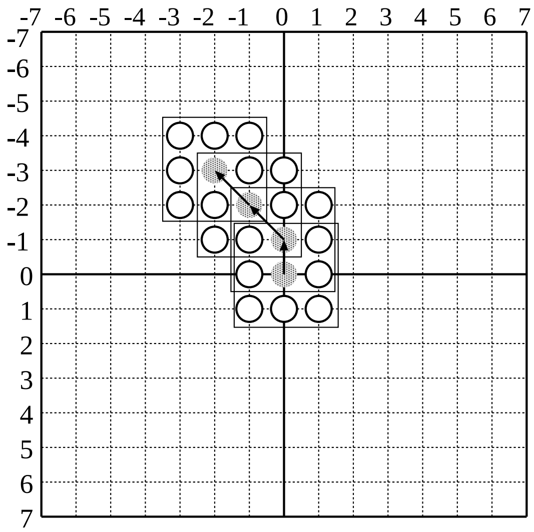
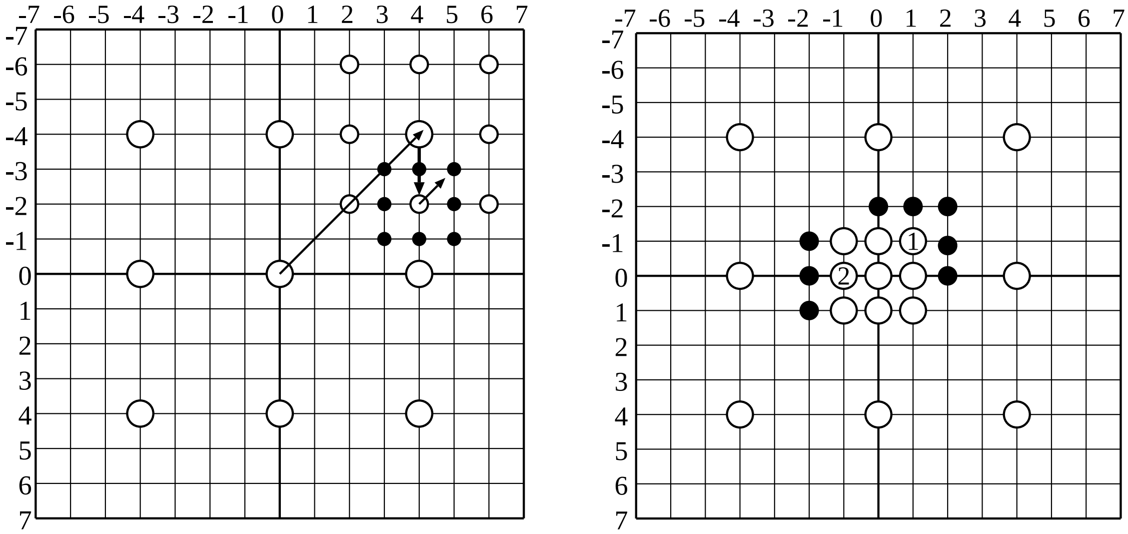


Abbildung 5.8: Block Based Gradient Descent Search



(a) Three Step Search

(b) New Three Step Search

Abbildung 5.9: „Three Step Search“ und „New Three Step Search“

Die Gefahr, daß der Algorithmus in ein lokales Minimum gerät, kann verringert werden, indem man nicht mehr in jedem Schritt nur einen 3×3-Bereich betrachtet, sondern einen größeren, was natürlich den Rechenbedarf erhöht.

5.3.5 Three Step Search (TSS) ■

Die *Three Step Search* ist ein sehr effizientes Suchverfahren für kleine maximale Bewegungsvektoren ($dx, dy \in [-7; 7]$). Das Verfahren ist in Abbildung 5.9(a) schematisch dargestellt. Wie der Name schon sagt, arbeitet es in drei Stufen. In der ersten Stufe werden 9 Positionen mit $dx, dy \in \{-4; 0; 4\}$ untersucht und davon die Position mit dem minimalen Fehler bestimmt. Im zweiten Schritt wird nun der Fehler der 8 Positionen mit dem Abstand 2 von der bisher besten Position bestimmt und wieder die Position mit dem minimalen Fehler gewählt. Man beachte, daß die zuvor gewählte Position (die sich jetzt in der Mitte der neuen Positionen befindet) auch nun wieder gewählt werden kann, wenn sie den minimalen Fehler besitzt. Das Ganze wiederholt sich ein drittes Mal mit dem Abstand 1. Der in diesem letzten Schritt bestimmte Vektor wird als bester Vektor der TSS-Suche gewählt.

In der SAMPEG Implementierung wurde die Suche etwas verallgemeinert. Die Anzahl der Schritte ist nicht mehr auf 3 begrenzt, was den Suchbereich mit jedem zusätzlichen Schritt verdoppelt. Allerdings sinkt der Qualität der gefundenen Vektoren recht schnell mit der Anzahl der Schritte. Mit mehr als 5 Schritten (entspricht einem Suchbereich von $\pm$ 32 Pixel in Halfpel-Auflösung) sollte diese Suchmethode nicht benutzt werden.

5.3.6 New Three Step Search (NTSS) ■

Untersucht man die Verteilung der Motionvektoren in realem Videomaterial, so fällt auf, daß der überwiegende Teil der Vektoren sehr klein ist. Die Idee ist nun, anstatt der gleichmäßigen Verteilung von Suchpunkten in TSS eine Verteilung zu benutzen, welche das Zentrum bevorzugt[2]. Das wird in der *New Three Step Search* (NTSS) [LZL94] realisiert, indem im ersten Schritt der Suche zu den aus der TSS bekannten Positionen noch die 8 Positionen um das Zentrum berechnet werden. Befindet sich das Minimum im ersten Schritt an einer der 8 äußeren Positionen, so wird fortgefahren, wie bisher. Befindet es sich im Zentrum, so wird sofort abgebrochen. Befindet es sich hingegen an einem der 8 Nachbarpunkte des Zentrums (Positionen „1“ und „2 “ in Abbildung 5.9(b)), so werden in einem zweiten Schritt dessen 8 Nachbarpunkte berechnet. Das Minimum dieser Punkte incl. des mittleren gilt dann als Ergebnis der Suche.

5.3.7 Dynamic Search Window Adjustment (DSWA) □

Die Idee dieser Methode [LWLS93] ist, die schrittweise Verkleinerung des Suchbereiches bei der TSS davon abhängig zu machen, wie sehr sich die Fehler der Prüfpunkte unterscheiden. Liegt der Fehler eines Prüfpunktes stark unterhalb der anderen, so wird der Suchbereich stärker um diesen Punkt eingeschränkt. Sind die Fehler etwa gleich verteilt, wird der Suchbereich langsamer verkleinert. Sei dazu P die Menge der Prüfpunkte in einem Schritt der TSS und $p_1 \in P$ der Punkt mit $MAD(p_1) = \min_{p \in P}(MAD(p))$ (Prüfpunkt mit kleinstem Fehler), sowie $p_2 \in P$ mit $MAD(p_2) = \min_{p \in P \setminus \{p_1\}}(MAD(p))$ der Prüfpunkt mit zweitkleinstem Fehler. Dann berechnen wir

$$G = \frac{MAD(p_2) - MAD(p_1)}{MAD(p_2)}$$

und entscheiden uns für den Verkleinerungsfaktor gemäß:

$$\text{Verkleinerung des Suchbereiches} = \begin{cases} \text{Faktor } 4 & \text{falls } T_H < G \\ \text{Faktor } 2 & \text{falls } T_L < G \leq T_H \\ \text{Faktor } \frac{4}{3} & \text{falls } G \leq T_L \end{cases}$$

5.4 Optimierung des Prüfpunktmusters

Es existieren Untersuchungen darüber, ob es durch eine optimierte Verteilung der Prüfpunkte im Suchbereich möglich ist, die Wahrscheinlichkeit für eine gute Lösung zu erhöhen [ZLL97]. Der oben vorgestellte NTSS Algorithmus versuchte, ausgehend von dem gleichmäßigen Prüfpunktmuster der TSS, eine Verbesserung zu erzielen, indem er Prüfpunkte um das Zentrum hinzufügte. Dies basiert auf der Beobachtung, daß besonders in Anwendungen wie der Bildtelefonie die Anzahl der unbewegten Bildteile überwiegt.

Das Problem läßt sich etwas allgemeiner definieren, indem wir mit $C = \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_M\}$ die Menge der Prüfpunkte bezeichnen und mit $\mathbf{x}_g$ die Position des globalen Minimums. Wir bezeichnen weiterhin den minimalen Abstand eines Prüfpunktes vom globalen Minimum durch

$$D_{CPP}(\mathbf{x}_g, C) = \min_i ||\mathbf{x}_g - \mathbf{c}_i||$$

wobei $|| \cdot ||$ für die Euklidische Distanz steht (CPP steht für *checking point pattern*). Ist uns eine Verteilung der korrekten Motionvektoren durch $P(\mathbf{x}_i)$ gegeben, so können wir die optimale Positionierung der Prüfpunkte erhalten, indem wir versuchen, den mittleren minimalen Abstand zwischen Prüfpunkt und globalem Minimum zu minimieren:

$$\overline{D}_{CPP} = \sum_{\mathbf{x}_i} P(\mathbf{x}_i) ||\mathbf{c}_{\mathbf{x}_i} - \mathbf{x}_i||$$

[2]Siehe Abschnitt 5.4.

$\mathbf{c}_{\mathbf{x}_i}$ ist dabei der dem Punkt $\mathbf{x}_i$ am nächsten gelegene Prüfpunkt aus C. Ein Prüfpunktmuster, das $\overline{D}_{CPP}$ minimiert, ist wichtig, denn je weiter ein Prüfpunkt vom globalen Minimum entfernt ist, desto größer ist die Gefahr, bei einem Gradientenabstiegsverfahren in einem zweiten Schritt in ein lokales Minimum zu geraten. Die Verteilung der optimalen Motionvektoren läßt sich mit verschiedenen Ansätzen ermitteln. Ist der Inhalt der Sequenzen immer recht ähnlich, so kann diese durch eine Reihe von Trainingssequenzen ermittelt werden. Ebenso ist es aber denkbar, das Prüfpunktmuster während der Videosequenz immer dem aktuellen Szenengeschehen anzupassen. Je länger die Sequenz ist, aus der die Motionvektorverteilung ermittelt wurde, desto regelmäßiger ist das entstehende Prüfpunktmuster.

5.5 Motionvektor-Verteilungen

5.5.1 Zeitliche Korrelation

Gegeben sei ein Suchbereich S und darauf eine Motionvektorverteilung $S \to \mathbb{R}$ für das Bild a und den Abstand d zum Referenzbild durch $P_a^d(\mathbf{x})$. Um zu untersuchen, wie groß die Korrelation zwischen zeitlich aufeinanderfolgenden Bildern a und b ist, berechnen wir:

$$C(a,b,d) = \frac{\sum\limits_{\mathbf{x}\in S} P_a^d(\mathbf{x}) P_b^d(\mathbf{x})}{\left\{\sum\limits_{\mathbf{x}\in S} P_a^d(\mathbf{x})^2\right\}^{1/2} \left\{\sum\limits_{\mathbf{x}\in S} P_b^d(\mathbf{x})^2\right\}^{1/2}}$$

Meßdaten für die Standardsequenzen *Football* und *Table-Tennis* sind in Tabelle 5.1 zusammengefaßt. Es bestätigt die intuitive Vermutung, daß die Korrelation mit zunehmendem Abstand der Bilder abnimmt und zwar je schneller, desto ungleichmäßiger die Bewegung in der Sequenz ist.

	Football			Table-Tennis		
$\lvert b-a \rvert$	Min.	Avg.	Max.	Min.	Avg.	Max.
1	0.004	0.806	0.999	0.019	0.949	0.999
2	0.001	0.651	0.999	0.032	0.895	0.999
3	0.001	0.546	0.999	0.034	0.832	0.999

Tabelle 5.1: Korrelation der Motionvektorverteilung zwischen zeitlich entfernten Bildern und $d = 1$ (entnommen aus [ZLL97]).

5.5.2 Interblock-Korrelation

Unter der *Interblock-Korrelation* versteht man die Korrelation der Motionvektoren innerhalb eines Bildes. Um diese zu bestimmen, wurde (ebenfalls in [ZLL97]) für jeden Makroblock das Prüfpunktmuster anhand der Motionvektoren der acht umgebenden Makroblöcke bestimmt (ohne zu beachten, daß das natürlich in Wahrheit ein nichtkausaler Vorgang ist). Gemessen wurde dann die über alle Makroblöcke gemittelte D_{CPP}. Diese Ergebnisse sind in Tabelle 5.2 aufgeführt. Zu erkennen ist, daß dabei D_{CPP} ziemlich linear mit dem Abstand zum Referenzbild steigt.

5.5.3 Interframe-Korrelation

Hierbei messen wir, wie groß der mittlere D_{CPP} (siehe vorheriger Abschnitt) ist, wobei wir diesmal das Prüfpunktmuster ausgehend von den Motionvektoren des Makroblocks und seiner 8 Nach-

	Football			Table-Tennis		
Distanz d	Min.	Avg.	Max.	Min.	Avg.	Max.
1	0.003	1.07	3.10	0.012	0.233	2.80
2	0.021	2.62	6.31	0.022	0.718	5.27
3	0.043	3.95	8.20	0.083	1.34	6.77

Tabelle 5.2: Interblock-Korrelation (gemitteltes D_{CPP}) der Motionvektorverteilung (entnommen aus [ZLL97]).

barblöcke im *vorhergehenden* Bild bestimmen (dies stellt nun ein kausales Verfahren dar). Die Ergebnisse sind in Tabelle 5.3 aufgeführt. Es zeigt sich, daß sie erwartungsgemäß nicht so gut sind, wie bei der Interblock-Korrelation; insbesondere, was den maximalen Abstand betrifft.

	Football			Table-Tennis		
Distanz d	Min.	Avg.	Max.	Min.	Avg.	Max.
1	0.000	1.24	4.95	0.000	0.289	7.41
2	0.009	2.78	9.11	0.016	0.697	13.52
3	0.021	4.13	10.49	0.06	1.15	19.49

Tabelle 5.3: Interframe-Korrelation (gemitteltes D_{CPP}) der Motionvektorverteilung (entnommen aus [ZLL97]).

5.6 Weitere Suchverfahren

5.6.1 Fast Full Search

In [LT97] wird vorgeschlagen, das Abstandsmaß durch ein zusätzliches Abstandsmaß zu ergänzen, das schneller berechnet werden kann und eine untere Abschätzung für das eigentlich benutzte Abstandsmaß liefert. Dadurch ist es möglich, Suchpositionen auszuschließen, ohne vorher den eigentlichen Abstand zu berechnen. Dazu werden drei Maße angegeben, die zur Bestimmung einer Abschätzung für MSE ($p = 2$) und MAD ($p = 1$) benutzt werden können:

massive projection

$$M_p(x_0, y_0, dx, dy) = |m_f(x_0, y_0) - m_{\hat{f}}(x_0 + dx, y_0 + dy)|^p$$

mit

$$m_f(x_0, y_0) = \sum_{x=0}^{15} \sum_{y=0}^{15} f(x_0 + x, y_0 + y)$$

vertical projection

$$V_p(x_0, y_0, dx, dy) = \sum_{y=0}^{15} |v_f(x_0, y_0, y) - v_{\hat{f}}(x_0 + dx, y_0 + dy, y)|^p$$

mit

$$v_f(x_0, y_0, y) = \sum_{x=0}^{15} f(x_0 + x, y_0 + y)$$

horizontal projection

$$H_p(x_0, y_0, dx, dy) = \sum_{x=0}^{15} |h_f(x_0, y_0, x) - h_{\hat{f}}(x_0 + dx, y_0 + dy, x)|^p$$

mit

$$h_f(x_0, y_0, x) = \sum_{y=0}^{15} f(x_0 + x, y_0 + y)$$

Untere Schranken erhält man nun sehr schnell aufgrund der folgenden Eigenschaften, die sich durch Anwenden der Schwarzschen Ungleichungen leicht beweisen lassen:

- $M_1(x_0, y_0, dx, dy) \leq MAD(x_0, y_0, dx, dy)$
- $V_1(x_0, y_0, dx, dy) \leq MAD(x_0, y_0, dx, dy)$
- $H_1(x_0, y_0, dx, dy) \leq MAD(x_0, y_0, dx, dy)$
- $M_2(x_0, y_0, dx, dy) \leq 16^2 \cdot MSE(x_0, y_0, dx, dy)$
- $V_2(x_0, y_0, dx, dy) \leq 16 \cdot MSE(x_0, y_0, dx, dy)$
- $H_2(x_0, y_0, dx, dy) \leq 16 \cdot MSE(x_0, y_0, dx, dy)$

5.6.2 Integral Projection Search

Ein ähnlicher Ansatz ist das *Integral Projection*-Verfahren [SS96]. Anstatt den Blockfehler anhand eines Abstandsmaßes zu bestimmen, wird wie in Fast Full Search die vertical und horizontal projection für alle 16×16-Blöcke im Referenzbild und des zu suchenden Blockes berechnet und dann das eigentliche Matching mit einem normalen Abstandsmaß nur auf diesen Projektionen gemacht. Das Interessante an diesem Verfahren ist, daß der horizontale und der vertikale Anteil des Motionvektors unabhängig voneinander berechnet werden können (Abbildung 5.10).

5.6.3 Eigenes Fast Full Search

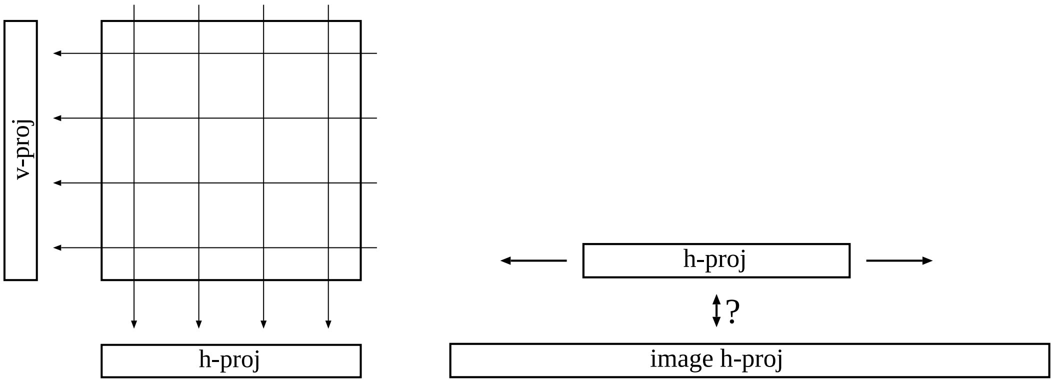
In diesem von mir entwickelten Fast Full Search-Verfahren wird auch eine volle Suche durchgeführt, wobei durch ein Ausschlußkriterium der Suchraum sehr schnell verkleinert werden kann, ohne dabei Gefahr zu laufen, optimale Lösungen zu entfernen. Das Verfahren konnte leider nicht mehr in SAMPEG implementiert werden, so daß keine Erfahrung existiert, wie stark der Suchraum in der Praxis durch das Kriterium tatsächlich eingeschränkt werden kann und welcher Geschwindigkeitsgewinn sich dadurch ergibt.

Lemma 1 *Für $A, B, C \in \mathbb{R}$ gilt: $|C - A| \geq |B - A| - |C - B|$.*

Beweis: Aus der Dreiecksungleichung $|a + b| \leq |a| + |b|$ erhalten wir mit $a = C - A$, $b = B - C$, $a + b = C - A + B - C = B - A$: $|B - A| \leq |C - A| + |B - C|$. □

Sei wieder $\hat{f}(x, y)$ mit $0 \leq x < W$, $0 \leq y < H$ das Referenzbild und $b(x, y)$ der Makroblock, für den ein Motionvektor bestimmt werden soll, sowie $d(x_0, y_0)$ mit $0 \leq x_0 < W - 17$, $0 \leq y_0 < H - 16$ eine für das Referenzbild vorausberechnete Funktion mit

$$d(x_0, y_0) = \sum_{0 \leq x, y \leq 15} |\hat{f}(x_0 + x + 1, y_0 + y) - \hat{f}(x_0 + x, y_0 + y)|$$



(a) Erzeugen der horizontalen und vertikalen Projektionen durch Aufsummieren der Pixelwerte.

(b) Vergleich der Block-Projektion mit der Projektion des Referenzbildes

Abbildung 5.10: Integral Projection Search

Mit der linken oberen Ecke des Vergleichsblockes im Referenzbild bei $(x_0; y_0)$ sei die SAD gegeben durch

$$SAD(x_0, y_0) = \sum_{0 \leq x,y \leq 15} |\hat{f}(x + x_0, y + y_0) - b(x, y)|$$

Wollen wir die $SAD(x_0 + 1, y_0)$ berechnen, so können wir mittels $SAD(x_0, y_0)$ und $d(x_0, y_0)$ eine untere Schranke für den zu erwartenden Fehler angeben:

$$SAD(x_0 + 1, y_0) \geq SAD(x_0, y_0) - d(x_0, y_0)$$

Beweis: Aus dem Lemma folgt mit $A = b(x, y)$, $B = \hat{f}(x + x_0, y + y_0)$ und $C = \hat{f}(x + x_0 + 1, y + y_0)$:

$$\begin{aligned} \sum_{x,y} |\hat{f}(x + x_0 + 1, y + y_0) - b(x, y)| \geq & \sum_{x,y} |\hat{f}(x + x_0, y + y_0) - b(x, y)| - \\ & \sum_{x,y} |\hat{f}(x + x_0 + 1, y + y_0) - \hat{f}(x + x_0, y + y_0)| \end{aligned}$$

$$SAD(x_0 + 1, y_0) \geq SAD(x_0, y_0) - d(x_0, y_0)$$

□

Ganz entsprechend kann man auch eine untere Schranke für aufeinanderfolgende vertikale Positionen angeben: $SAD(x_0, y_0 + 1) \geq SAD(x_0, y_0) - d'(x_0, y_0)$, wobei nun d' definiert ist durch:

$$d'(x_0, y_0) = \sum_{0 \leq x,y \leq 15} |\hat{f}(x_0 + x, y_0 + y + 1) - \hat{f}(x_0 + x, y_0 + y)|$$

Der Algorithmus

Der Algorithmus bestimmt zuerst einen möglichst guten Motionvektor, also z.B. den 0-Vektor oder den Vektor des letzten Blockes. Danach beginnt eine volle Suche. Jedesmal, wenn wir eine Position testen, die einen größeren Fehler als den bisher minimalen Fehler hat, können wir ggf.

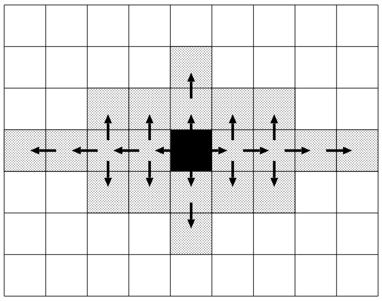


Abbildung 5.11: Elimination von Suchpositionen bei *Eigenes Fast Full Search*

mehrere Positionen von der Suche ausschließen. Dazu subtrahieren wir jedesmal, wenn wir von der berechneten Position einen Schritt nach rechts gehen $d(\cdot,\cdot)$ von dem berechneten Fehler. Solange diese Abschätzung größer ist, als der bisher minimale gefundene Fehler, kann diese Position nicht optimal sein. Dasselbe machen wir in die anderen Richtungen (siehe Abbildung 5.11). Da in flachen Bildregionen sowohl $d()$ wie auch $d'()$ relativ klein sind, werden diese durch diesen Algorithmus ziemlich schnell durchsucht, wenn der zu suchende Block eine Textur enthält. Texturblöcke werden entsprechend langsamer durchsucht. Die Vorberechnung von $d(x_0, y_0)$ und $d'(x_0, y_0)$ ist nicht so aufwendig, wie es auf den ersten Blick erscheinen mag, denn diese muß ja für jedes Referenzbild nur einmal gemacht werden. Da bei n B-Bilder zwischen den P-Bildern in jedem Referenzbild $2n$ oder $2n+1$ mal gesucht wird, fällt die Vorverarbeitung nicht mehr so ins Gewicht.

5.6.4 Weitere Suchverfahren

Motion-Field Subsampling

Beim *Motion-Field Subsampling* wird nicht für jeden Makroblock ein Motionvektor gesucht, da angenommen wird, daß die Verteilung von Motionvektoren sehr gleichmäßig über das Bild ist und man deshalb einen Motionvektor schon aus den umgebenden hinreichend gut abschätzen kann.

Hierarchische Suche

Hierbei wird das Bild in mehreren Stufen tiefpaßgefiltert und in der Auflösung reduziert. Man nimmt dann eine Motionvektorsuche in einer niedrigeren Auflösung vor und verfeinert diesen Vektor, indem man auf immer höheren Auflösungen in einem kleinen Umkreis sucht.

5.7 Zur Implementation

Die Zusammenfassung der Performance der oben beschriebenen und implementierten Algorithmen ist in Abschnitt 9.1.5 auf Seite 104 zu finden. Hier folgen noch einige kleinere Implementierungsdetails.

5.7.1 Auswahl eines Makroblocktyps

Nachdem man einen passenden Motionvektor bestimmt hat, bleibt immer noch die Frage, ob der betreffende Makroblock nun mit dieser Prädiktion codiert wird oder lieber als Intra-Block ohne Prädiktion. Dies geschieht, indem für ihn jeweils mit und ohne Prädiktion die Anzahl der Bits abgeschätzt wird, die zur Codierung benötigt werden (siehe dazu Abschnitt 7.5.5). Der Codierungstyp, für den die wenigsten Bits erwartet werden, wird ausgewählt.

In SAMPEG wurde auch ein exaktes Verfahren implementiert, das alle Möglichkeiten vollständig codiert und die tatsächlichen Bits zählt. Dieses hat einen vergleichsweise hohen Rechenaufwand und lohnt sich für die Praxis nicht.

Es zeigt sich, daß fast alle Makroblöcke mit Prädiktion codiert werden. Ist die Szenenerkennung aktiviert und ist die Prädiktion deshalb niemals *ganz* falsch, so wäre es sogar möglich, auf die Makroblocktypauswahl ganz zu verzichten und alle Blöcke mit Prädiktion zu codieren.

5.7.2 Automatische Reduktion des Suchbereiches

Macht man den Suchbereich größer, so steigt zwar die Chance, eine gute Prädiktion zu finden, allerdings benötigt man dann auch mehr Bits zur Codierung der Motionvektoren. Diese Mehrkosten für die Codierung können sehr schnell den Gewinn durch die bessere Prädiktion übersteigen. Deshalb wird in der Implementation von SAMPEG der Suchbereich nach der Motionvektorsuche so weit eingeschränkt, daß fast alle Motionvektoren in ihm liegen. Die wenigen Makroblöcke, deren Motionvektoren den Suchbereich überschreiten, werden Intra-codiert. Aber selbst das garantiert noch nicht, daß die Größe eines codierten Bildes mit dem Suchbereich (bei Full-Search) sinkt, da vielleicht in glatten Bereichen viele sehr große Motionvektoren gefunden werden, obwohl kleine Vektoren fast genauso gut wären. Dann wird auf einen hohen Suchbereich geschaltet, der dann ineffizienter sein kann.

Kapitel 6

Szenenwechselerkennung

6.1 Problem

Codiert man eine Videosequenz mit einer immer gleichbleibenden GOP-Struktur, so kann durch eine ungünstige Lage des Szenenwechsels[1] innerhalb der GOP ein Qualitätsverlust die Folge sein. Betrachten wir als Beispiel die Anordnung in Abbildung 6.1. Da der Bildinhalt der neuen Szene beträchtlich von der letzten abweichen kann, wird das dem Szenenwechsel folgende P-Bild kaum Motionvektoren enthalten können, die eine brauchbare Prädiktion liefert. Folglich wird die Codierung bedeutend mehr Bits benötigen, will man eine konstante Bildqualität beibehalten. Aber nicht nur für die Codierung dieses P-Bildes entstehen Nachteile, auch die beiden vorhergehenden B-Bilder können nur noch das vorausgehende P-Bild als gute Prädiktion benutzen. Entsprechendes gilt für die beiden folgenden B-Bilder.

Ein weiterer Nachteil dieser Anordnung ist, daß zukünftige Videoschnittsysteme, die direkt auf den komprimierten Daten arbeiten werden, bevorzugt an GOP-Grenzen schneiden sollten, da dies hier ohne Umcodierung und somit verlustlos möglich ist[2]. Befindet sich nun ein Szenenwechsel innerhalb dieser GOP, so muß diese bei einem Schnitt an dieser Stelle immer ganz entfernt werden, da sich ja in der GOP Bilder aus den beiden zu trennenden Szenen befinden.

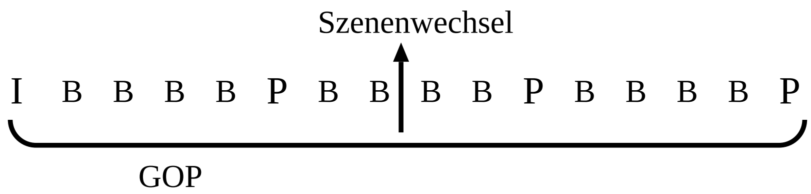


Abbildung 6.1: Ungünstige Lage eines Szenenwechsels

6.2 Ziel

Betrachtet man das Problem allein aus der Sicht der Codierungseffizienz, so ist die Positionierung nach Abbildung 6.2 sehr effizient. Man beachte, daß die dem Wechsel folgende GOP ihrer ersten

[1]Soweit nicht anders angegeben, betrachten wir hier nur einen plötzlichen Wechsel zwischen zwei Szenen und keine Überblendungen oder sonstige Blenden.
[2]siehe Abschnitt 3.3.2 auf Seite 25

B-Bilder beraubt wurde. Die Positionierung des Szenenwechsels am Anfang der GOP ermöglicht den verlustlosen Videoschnitt und alle P- und B-Bilder erhalten optimale Referenzbilder. Für diese

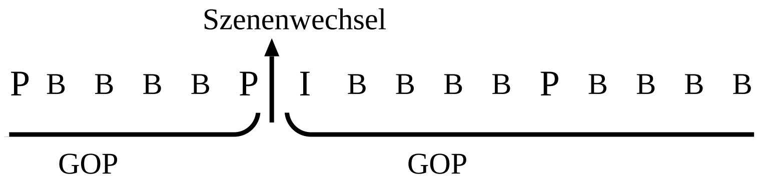


Abbildung 6.2: Günstige Lage eines Szenenwechsels

Anordnung gibt es jedoch zwei Kritikpunkte:

1. Durch den Wegfall der B-Bilder am Beginn der GOP erleiden wir einen kleinen Effizienzverlust in der Codierung. Dieser fällt jedoch nicht besonders stark aus, da die geringere Zahl der B-Bilder durch eine Verlängerung der GOP ausgeglichen werden kann. Die relative Anzahl der B-Bilder zu den P-Bildern sinkt dadurch jedoch minimal und durch eine längere Kette von P-Bildern erhöht sich die Fehlerfortpflanzung geringfügig.

2. Der Verlauf der erzielten Bildqualität widerspricht genau dem visuellen Empfinden eines Betrachters. Während das codierte Video die maximale Bildqualität im ersten Bild (dem I-Bild) nach dem Wechsel erzielt, kann der Mensch diese Qualität aufgrund der Bewegungsmaskierung[3] erst einige Bilder später wahrnehmen.

Eine dementsprechend verbesserte Anordnung ist in Abbildung 6.3 zu sehen. Die im vorigen Vorschlag entfernten B-Bilder sind nun wieder enthalten und der Szenenwechsel wurde auf den ersten B-Bild vor dem (ersten) I-Bild der GOP gesetzt. Diese B-Bilder lassen sich zwar schlechter codieren als sonst, da ein Referenzbild praktisch für die Prädiktion ausfällt, aber wegen der Bewegungsmaskierung darf die Qualität dieses Bild verhältnismäßig niedrig sein. Das I-Bild folgt mit einigem Abstand und wird wieder mit der gewohnt hohen Qualität codiert.

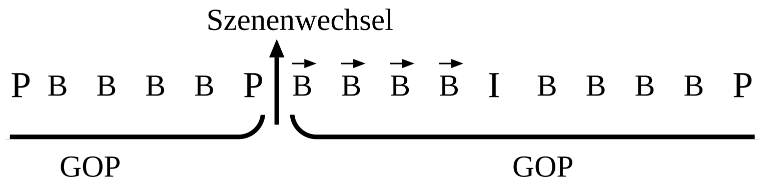


Abbildung 6.3: Beste Lage eines Szenenwechsels

Man beachte, daß diese Anordnung immer noch einen verlustlosen Videoschnitt erlaubt. Denn da die ersten B-Bilder das vorhergehende P-Bildes sowieso nicht sinnvoll nutzen können, kann man die Motionvektorsuche gleich so einschränken, daß nur Backward-Prädiktion von dem folgenden I-Bild benutzt wird (was durch die Pfeile über den betreffenden B-Bildern in Abbildung 6.3 angedeutet werden soll). Durch diese Einschränkung darf das `closed_gop`-Flag im GOP-Header gesetzt werden, wodurch die GOP unabhängig von der vorhergehenden decodierbar wird und ein Schnitt darf erfolgen, ohne daß das `broken_link`-Flag im GOP-Header gesetzt werden muß [4].

[3]siehe Abschnitt 2.4.2 auf Seite 13
[4]siehe Abschnitt 3.3.2 auf Seite 25

6.3 Erkennung des Szenenwechsels

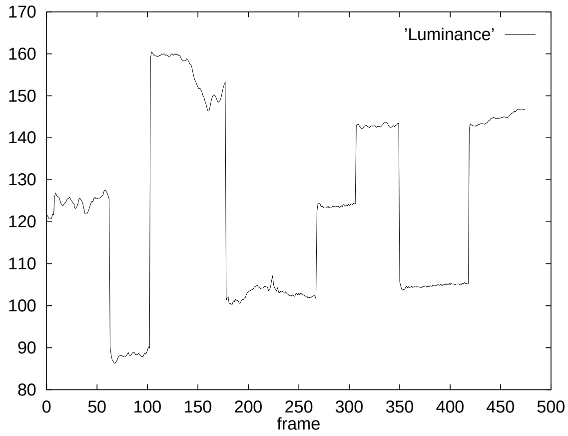
6.3.1 Erkennung durch ungenügende Prädiktion

Es ist theoretisch möglich, Szenenwechsel sehr zuverlässig zu erkennen, indem man eine Motionvektorsuche über das vorhergehende Bild macht. Nach der Makroblock-Typbestimmung, die wie gewohnt abläuft, kann anhand der Anzahl der Blöcke, die Intra-codiert werden, entschieden werden, ob zwischen den beiden Bildern ein wesentlicher Wechsel stattfindet. Die Schwelle läßt sich relativ unkritisch wählen.

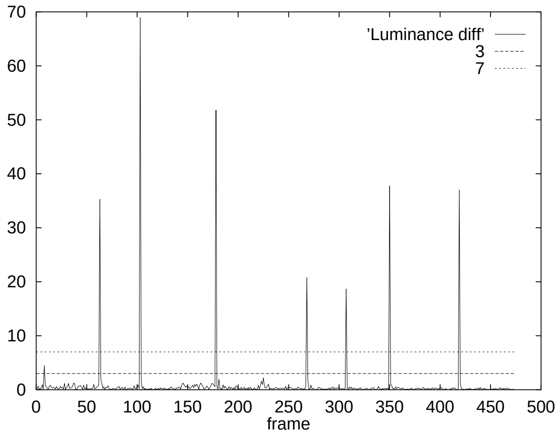
Der ganz große Nachteil dieser Methode ist aber der für die praktische Verwendung viel zu große Rechenaufwand, der in keinem Verhältnis zu der gewonnenen Information steht. Leider können die gefundenen Motionvektoren auch nicht für die Codierung der B- und P-Bilder herangezogen werden, da sie sich im allgemeinen auf die falschen Referenzbilder beziehen.

6.3.2 Erkennung anhand der Bildhelligkeit

Eine demgegenüber sehr effiziente Methode, die fast genauso gute Ergebnisse liefert, ist, die absolute Differenz der mittleren Helligkeiten benachbarter Bilder zu betrachten. In Abbildung 6.4(a) ist die mittlere Helligkeit der Bilder einer Testsequenz über die Zeit aufgetragen. Die Differenzen der benachbarten Bilder sind in Abbildung 6.4(b) dargestellt. Deutlich sind die Sprünge an den Szenenwechseln[5] zu erkennen. Lediglich der Sprung an Bild 8 ist etwas klein. Es wurde auch untersucht, andere Parameter wie die Varianz oder die mittlere Amplitude eines Farbartkanals zu benutzen. Das brachte i.A. jedoch schlechtere Ergebnisse. Möglich wäre es hingegen, mehrere Parameter gemeinsam zu betrachten. Einen wesentlichen Gewinn wird man damit aber nicht mehr erzielen, da die Helligkeitsunterschiede schon sehr zutreffende Aussagen erlauben.



(a) Luminanz



(b) Luminanz-Differenzen

Abbildung 6.4: Szenenwechselerkennung anhand der mittleren Bildhelligkeit

[5]Szenenwechsel finden an den Bildern 8, 63, 103, 176, 266, 305, 348 und 417 statt.

6.3.3 Ablauf

Zu Beginn wird die gewünschte GOP-Länge m und der Lookahead l angegeben, wobei $l-1$ der größten erwünschten GOP-Länge entspricht. Der Ablauf der Szenenerkennung ist nun, daß in jedem Schritt der Anfang der nächsten GOP gesucht wird, und nachdem dieser feststeht, die Bildtypen der vorhergehenden GOP festgelegt werden. Dazu müssen vor jedem Schritt immer l Bilder vorausgeladen werden. Dies ist in Abbildung 6.5 dargestellt. Die Rechtecke stellen den Lookahead-Bereich und die dicken Balken die in diesem Schritt erstellte GOP dar. Bereits codierte Bereiche sind gestrichelt eingezeichnet.

Abbildung 6.5: Ablauf der Szenenerkennung

Ein Szenenwechsel kann nun einfach erkannt werden, indem eine Schwelle festgelegt wird, über der der Helligkeitsunterschied zweier benachbarter Bilder liegen muß. Dazu muß die mittlere Helligkeit nicht einmal explizit berechnet werden, da man diese als Nebenprodukt bei der Berechnung der Varianz erhält, die für andere Codierungsstufen benötigt wird.

Dieses einfache Prinzip der Szenenerkennung wurde bei der Implementierung jedoch noch etwas verfeinert.

6.3.4 Verfeinerung

Es werden zwei Schwellen T_1, T_2 ($T_1 < T_2$) festgelegt und zwei Arten von Szenenwechseln unterschieden. Ein *potentieller Szenenwechsel*, der vorliegt, wenn für die Helligkeitsdifferenz ΔL gilt: $T_1 < \Delta L < T_2$ und ein *sicherer Szenenwechsel*, für den $\Delta L \geq T_2$ gelten muß[6]. Der Unterschied in der Behandlung liegt darin, daß zwar bei beiden der GOP-Anfang auf das betreffende Bild verschoben wird, aber die Forward-Prädiktion der ersten B-Bilder der GOP nur deaktiviert wird, wenn ein *sicherer Szenenwechsel* vorliegt. Damit ist eine konstante Bildqualität gesichert, auch wenn fälschlicherweise ein Szenenwechsel erkannt wurde. Befindet sich hingegen an einer Stelle ein Szenenwechsel, der nur als *potentiell* eingestuft wurde, so wird dieser auch optimal codiert. Lediglich ein Videoschnitt ist an dieser Stelle nicht ohne Verlust der B-Bilder möglich.

Befindet sich in dem untersuchten Bereich kein Szenenwechsel, so wird der nächste GOP-Anfang an die Stelle mit dem größten Helligkeitswechsel gelegt. Unter Umständen könnten damit aber sehr kurze oder sehr lange GOPs gebildet werden, was einerseits aus Gründen der Codierungseffizienz und andererseits aus Gründen der Fehlerfortpflanzung nicht erwünscht ist. Um diesem entgegenzuwirken, werden die Helligkeitsdifferenzen vor dem Vergleich noch mit einem Gewichtungsfaktor $w(n)$ multipliziert.

$$w(n) = \begin{cases} \frac{1}{2}(1 + (\frac{n}{m})^\alpha) & n \leq m \\ \frac{1}{2}(1 + (\frac{l-1-n}{l-1-m})^\alpha) & n > m \end{cases}$$

[6] Als günstige Werte haben sich $T_1 = 3$ und $T_2 = 7$ erwiesen, siehe Abbildung 6.4(b).

Dabei ist m die gewünschte GOP-Länge, l die Länge des Lookahead-Bereiches (siehe auch Abbildung 6.6) und α ein Parameter ≥ 1, der dazu führt, daß die angestrebte GOP-Größe um so genauer eingehalten wird, je höher α gewählt wird.

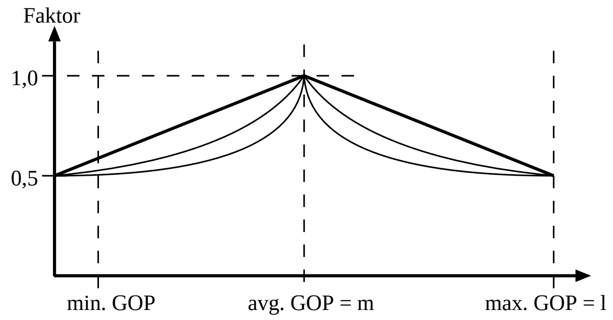


Abbildung 6.6: Gewichtungsfaktor

Zusätzlich wird noch eine harte Grenze für die minimale GOP-Größe eingeführt, die auf keinen Fall unterschritten werden darf. Da die GOP-Länge nun variabel ist, kann es passieren, daß sie mit einem `PP`- oder `PBP`-Muster aufhören würde. Da dies einen Effizienzverlust in der Codierung bewirken würde, wird das vorletzte P-Bild in ein B-Bild umgewandelt, wenn zwischen ihm und dem letzten P-Bild nicht mehr als 2 B-Bilder liegen.

Weitere Ergänzungen sind denkbar; z.B. könnte man die Anzahl der B-Bilder am Beginn einer GOP von der „Stärke“ des Szenenwechsels abhängig machen. Sinnvollerweise wären dazu Testreihen notwendig, wie die Maskierung von der Helligkeitsdifferenz abhängt. In der Implementierung wurde die Anzahl der B-Bilder per Default auf 4 eingestellt, was ein sehr vorsichtiger Wert ist (ausgehend von 25 Bilder/s). Auch ist nicht offensichtlich, was die „Stärke“ eines Szenenwechsels ausmacht. Es scheint so, daß alleine die Differenz der Helligkeitsunterschiede nicht für eine Charakterisierung ausreicht, sondern daß auch Parameter wie die Komplexität der Folgeszene eine Rolle spielen.

6.4 Ergebnisse

Zur Beurteilung wurde untersucht, wie sich die erzeugte Bitrate ändert, wenn die Szenenerkennung hinzugeschaltet wird. Die Quantisierung der einzelnen Bildtypen wurde dabei konstant (auf QScale=16) gelassen; der Effekt der Bewegungsmaskierung wurde also abgeschaltet. Bei einer Szenenwechselhäufigkeit von etwa 1 pro 50 Bilder erhält man eine Reduktion der Bitrate um etwa 0,3-0,4%. Wurde umgekehrt die Bitrate festgelegt und der RMS-Fehler der Ausgabe verglichen, so konnte keine eindeutige Verbesserung ausgemacht werden.

Ganz anders sehen die Ergebnisse allerdings aus, wenn die Bewegungsmaskierung aktiviert wird. Gemessen wurde die Ausgabebitrate, wenn *qscale* bei den 4 B-Bildern nach dem Wechsel und den (4-6) B-Bildern und dem P-Bild vor dem Wechsel um den Faktor 4 bzw. 3 erhöht wurde. Eine Verschlechterung der Bildqualität ist dabei nicht wahrnehmbar, vorausgesetzt, das Video wird mit 25 Bilder/s abgespielt. Wieder bei einer Szenenwechselhäufigkeit von etwa 1 pro 50 Bilder erhält man eine Reduktion der Bitrate von nun etwa 12-14%.

Wenn wir davon ausgehen, daß bei einem Szenenwechsel etwa 10 Bilder (5 davor und 5 danach) mit niedriger Qualität übertragen werden, so erhalten wir als obere Grenze für den erzielbaren Gewinn 20%. Denn gehen wir von dem Extremfall aus, daß diese 10 Bilder überhaupt nicht übertragen werden, so müssen statt der 50 Bilder zwischen zwei Szenenwechseln nur noch 40 übertragen werden. Man spart also 10 von 50 Bildern = 20%.

Angesichts des vernachlässigbaren Rechenaufwandes zur Erkennung der Szenenwechsel ist die Aktivierung der Szenenwechselerkennung unbedingt anzuraten. Die bessere Editiermöglichkeit ist dabei ein schöner Nebeneffekt.

6.5 Behandlung von Überblendungen und Blenden

Das oben beschriebene Verfahren geht von abrupten Szenenwechseln aus. Teilweise werden jedoch auch Übergänge benutzt, die sich über mehrere Bilder erstrecken. Diese reichen von einfachen Überblendungen innerhalb ca. 4 Bilder bis hin zu längeren Wischblenden, während denen Ausschnitte der vorhergehenden und der folgenden Szenen zu sehen sind. Die automatische Erkennung dieser Übergänge ist weit schwieriger und wurde nicht implementiert. Eine praktikablere Lösung als die automatische Erkennung dieser vielfältigen Übergänge scheint, daß schon bei der Erstellung des Videos vom Schnittprogramm, das die Blenden erzeugt, Informationen an den Encoder gegeben werden, wo sich welcher Übergang befindet.

Optimale Arrangements wären z.B. folgende: Bei kurzen Überblendungen innerhalb sehr weniger Bilder die Anordnung nach Abbildung 6.7. Das entspricht in etwa unserem Ansatz für den

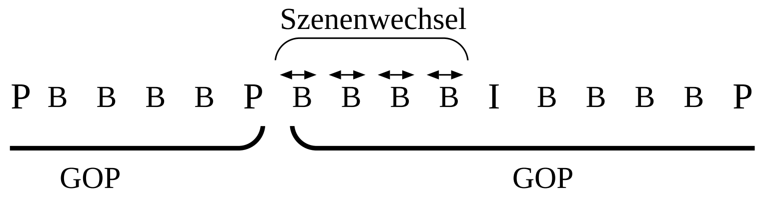


Abbildung 6.7: Behandlung einer weichen Überblendung über wenige Bilder

abrupten Szenenwechsel mit dem Unterschied, daß hierbei für die B-Bilder sowohl Forward- wie auch Backward-Prädiktion erlaubt sind. Folglich darf hier nicht das `closed_gop`-Flag gesetzt werden, was aber auch genau den erwünschten Effekt hat. Denn würde später an dieser Stelle geschnitten, so müssten die B-Bilder entfernt werden, was die Blende gezielt beseitigt.

Bei noch längeren Blenden, die sich in der Größenordnung von einer halben bis einer Sekunde befinden, bietet sich an, die Blende komplett in einer eigenen GOP zu codieren, um sie wieder leicht von den umgrenzenden Szenen trennen zu können.

Kapitel 7

Adaptive Quantisierung und Rate Control

Adaptive Quantisierung und Rate Control sind hier in einem Kapitel zusammengefaßt, da sie trotz ihrer abweichenden Aufgabe nicht unabhängig voneinander implementiert werden können, denn beide benutzen zur Regelung die Quantisierungsschrittweise (die indirekt durch *qscale* gesetzt wird), um ihr Ziel zu erreichen. Zwar ist es möglich, die Adaptive Quantisierung als Modulation des durch die Rate Control vorgegebenen *qscale*s zu betrachten, aber da die Bitrate nicht nur von dem *qscale*, sondern auch vom Bildinhalt abhängt, muß die Adaptive Quantisierung mit in die Regelschleife der Rate Control einbezogen werden.

Die Verfahren lassen sich in *kausale* und *nichtkausale* Verfahren einteilen. Unter einem kausalen Verfahren versteht man dabei ein Verfahren, daß seine Entscheidung nur aufgrund von Daten trifft, die zeitlich zurückliegen. Es kann zurückliegende Codierungsschritte nicht widerrufen und neu codieren. Ein nichtkausales Verfahren kann dagegen z.B. in der Rate Control die Codierung eines ganzen Bildes so lange optimieren, bis die Parameter optimal eingestellt sind.

Die Adaptive Quantisierung ist wie auch die Szenenerkennung keine notwendige Komponente eines Encoders. Sie dient ausschließlich der Verbesserung der erzielten Bildqualität bei einer festen Bitrate. In der Literatur sind Verbesserungen von bis zu einer Reduktion der Bitrate um 30% bei Codierung mit Adaptiver Quantisierung im Gegensatz zu einer Codierung ohne Adaptive Quantisierung bei gleicher Bildqualität genannt. Dieser Wert mag meiner Einschätzung nach etwas hoch gegriffen sein.

Die Rate Control ist für jeden Encoder Pflicht, der Bitstreams mit konstanter Bitrate (CBR = constant bit rate) erzeugen will. Das Verfahren, das dazu benutzt wird, ist jedoch frei wählbar.

Der Berkeley-MPEG1-Encoder besitzt weder Adaptive Quantisierung noch Rate Control, sondern benutzt jeweils konstante *qscale*s für I,P und B-Bilder. Diese liegen im Standardfall bei 8,10,25, können aber geändert werden.

7.1 Prinzipien der Adaptiven Quantisierung

Das Ziel der Adaptiven Quantisierung ist es, durch eine über das Bild variierende Quantisierungsschrittweite an den Stellen zusätzliches Rauschen in Kauf zu nehmen, an denen es nicht besonders sichtbar ist, um an anderen Stellen, an denen Codierungsfehler auffälliger sind, das Rauschen zu reduzieren. Da es sich dabei nur um eine Umverteilung des Fehlers handelt, wird sich dies kaum auf einfache Fehlermaße wie RMS oder SNR auswirken. Eine Bewertung ist nur subjektiv oder mit

Modellen möglich, die das menschliche Sehsystem modellieren.

Wichtig ist auch eine Abschätzung der Bildkomplexität. Bei Sequenzen mit geringer Komplexität (Tagesschau) kann einem I-Bild im Vergleich zu den P- und B-Bildern ein größerer Anteil der Bits zugewiesen werden, als in Sequenzen, in denen große zeitliche Veränderungen enthalten sind.

7.2 Prinzipien der Rate Control

Die Rate Control besitzt streng genommen zwei Aufgaben. Einerseits sorgt sie dafür, daß die Bitrate des erzeugten Bitstreams in einem eingeschränkten Bereich bleibt oder im Extremfall sogar konstant ist. Die zweite Aufgabe ist, abzusichern, daß der Eingangsbuffer im Decoder nicht über- oder unterläuft[1]. Erfüllt ein Encoder die zweite Aufgabe, so wird er als *VBV konform* betrachtet. Die VBV Konformität kann im übrigen auch durch ein an einen nicht VBV konformen Encoder angehängtes Modul erreicht werden, das bei (Empfangs-)Bufferüberlauf die Anzahl der Bits pro Bild künstlich durch Einfügen eigentlich nutzloser Füllbytes vergrößert und bei Unterlauf durch einfaches Entfernen von Koeffizienten die Bildgröße verkleinert. Dies sollte am zweckmäßigsten nur an B-Frames durchgeführt werden, da sich dann die zusätzlich eingefügten Fehler nicht längere Zeit durch die Prädiktionen im Bild halten.

Der übliche Ansatz zur Realisierung einer Rate Control ist, den Ausgabebuffer im Encoder zu überwachen und die Abweichung von seinem aktuellen Füllstand zum Sollfüllstand zur Steuerung der Quantisierungsschrittweite zu benutzen. In Extremfällen kann es trotz einer Regelung der Quantisierungsschrittweite bis an die erlaubten Grenzen möglich sein, daß zu viele oder zu wenige Bits codiert werden. Für diesen Fall muß der Encoder weitere Notmaßnahmen ergreifen, wie eben das Einfügen von Füllbytes oder das Streichen von Koeffizienten.

7.3 Heuristische Verfahren

7.3.1 Prinzip

Das Ziel aller Verfahren ist, eine über das ganze Bild gleichmäßig empfundene Qualität zu erreichen. Da es mangels eines ausreichenden Modells für das menschliche Sehsystem kein zuverlässiges Maß für die wahrgenommene Qualität eines Makroblocks gibt, schätzen einige Verfahren die Empfindlichkeit des Auges auf Codierungsfehler durch andere Maße, wie z.B. die Varianz, ab. Dabei wird angenommen, daß in stark texturierten Regionen (also Regionen mit einer hohen Varianz) Fehler nicht so sehr erkannt werden, wie in flachen Regionen (mit geringerer Varianz).

Ein anderer Ansatz ist, die Makroblöcke in mehrere Klassen einzuteilen, wobei meist die Klassen *Textur*, *Ebene* und *Kante* benutzt werden. Hier weiß man jetzt ungefähr, daß die Qualität bei Blöcken der Klasse „Ebene" schneller mit steigendem *qscale* abnimmt, als bei Blöcken der Klasse „Textur" (siehe Abbildung 7.1). Gibt man einen Qualitätslevel vor, so ergeben sich daraus die zugehörigen *qscale*s für die Makroblockklassen. Durch eine Abschätzung der benötigten Bits zur Codierung und eine schrittweise Korrektur des Qualitätslevels, bis die gewünschte Ausgabebitzahl erreicht ist, erhält man eine gleichmäßige Bildqualität.

[1]Bei der Codierung von Low-Delay-Streams darf der Buffer unterlaufen. Dieser Modus wird von den meisten Encodern aber gar nicht unterstützt, da dies ein spezieller Modus mit einigen zusätzlichen Einschränkungen ist.

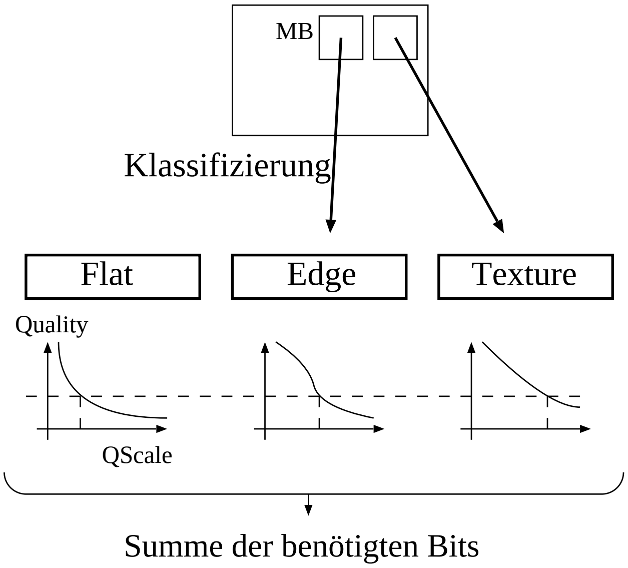


Abbildung 7.1: Prinzip des Adaptiven Quantisierung mit Makroblockklassifikation

7.4 Verfahren aus der Literatur

7.4.1 Verfahren von ***Pickering, Arnold***

Obwohl sich der Artikel von Pickering und Arnold [PA94] nicht direkt auf die Codierung mit dem MPEG-Verfahren bezieht, läßt es sich ohne großen Aufwand dafür anpassen. Der wesentliche Unterschied ist, daß der Parameter q zur Einstellung der Quantisiererschrittweite hier im Bereich $q \in [0; 175]$ variiert werden kann. Dies ist bei der Anwendung auf das MPEG-Verfahren zu beachten, dessen *qscale*-Parameter mit einem anderen Wertebereich arbeitet. Die Konstanten sind dabei entsprechend anzupassen oder neu zu bestimmen. Dieses Verfahren vereint Adaptive Quantisierung und Rate Control in einem einzigen Prozess, indem es den Parameter q in mehrere Summanden aufteilt:

$$q = \underbrace{q_{act} + q_{mtn}}_{AdaptiveQuantisierung} + \underbrace{q_{fld}}_{RateControl}$$

Dabei bezieht sich q_{act} auf die Kontrastmaskierung, q_{mtn} auf die Bewegungsmaskierung und mit q_{fld} wird die Rate-Control durchgeführt.

Kontrastmaskierung

Die Aktivität in einem Makroblock wird für 8×8-Blöcke getrennt bestimmt. Dazu wird jeder Block nochmals in mehrere 4×4-Subblöcke I_i unterteilt (siehe Abbildung 7.2). Die Helligkeitswerte

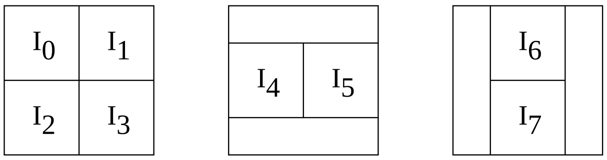


Abbildung 7.2: Aufteilung eines 8×8-Blocks in 4×4-Blöcke

in Subblock I_i seien durch $I_i(x,y)$ mit $0 \leq x,y \leq 3$ gegeben. Für jeden Subblock werden die Differenzen zweier horizontal bzw. vertikal benachbarter Pixel bestimmt und aufsummiert:

$$\delta_h I_i = \frac{1}{3 \cdot 4} \sum_{x=0}^{3} \sum_{y=0}^{4} |I_i(x,y) - I_i(x+1,y)|$$

$$\delta_v I_i = \frac{1}{3 \cdot 4} \sum_{x=0}^{4} \sum_{y=0}^{3} |I_i(x,y) - I_i(x,y+1)|$$

Daraus werden nun die Kantenaktivitäten bestimmt, die durch λ_i mit $0 \leq i \leq 7$ bezeichnet werden. Diese werden berechnet durch:

$$\lambda_i = \begin{cases} \frac{1}{2}(\ln(\delta_h I_i) + \ln(\delta_v I_i)) & \text{für } 0 \leq i \leq 3 \\ ln(\delta_v I_i) & \text{für } i \in \{4,5\} \\ ln(\delta_h I_i) & \text{für } i \in \{6,7\} \end{cases}$$

Die Subblöcke I_4 bis I_7 dienen dabei zur Detektion von Kanten, die genau durch die Mitte eines Blockes laufen. Das Minimale λ_i wird nun mit λ_{min} bezeichnet und q_{act} wird berechnet durch:

$$q_{act} = C_1 \lambda_{min} - C_2 \left(\frac{1}{8} \left(\sum_{i=0}^{7} \lambda_i \right) - \lambda_{min} \right)$$

Für die Konstanten C_1 und C_2 werden die Werte $C_1 = 6$ und $C_2 = 0,7$ empfohlen, was durch subjektive Tests ermittelt wurde. Die Wirkungsweise dieses Ausdrucks ist, daß durch den ersten Summanden ($C_1 \lambda_{min}$) die Quantisierung erhöht wird, wenn die „Rauhigkeit" eines Blockes zunimmt. Um dabei aber zu verhindern, daß an Kanten zwischen Textur und glatter Fläche relativ grob quantisiert wird, was zu dem Ringing-Effekt führen würde, regelt der zweite Summand gegen, je stärker sich die Subblockinhalte innerhalb eines Blockes unterscheiden.

Dieses Verfahren zur Behandlung der Kontrastmaskierung wurde testweise in SAMPEG implementiert, brachte jedoch nicht die erhoffte Verbesserung. Zwar ist die Unterscheidung zwischen Textur und Fläche sehr zuverlässig, aber die Verbesserung der Kanten war nicht zuverlässig genug, um den Ringing-Effekt zu verhindern.

Bewegungsmaskierung

Zur Bestimmung der Bewegungsmaskierung wird für jeden Block b eines Bildes n (mit den Helligkeitswerten $f_{n,b}(x,y), \quad 0 \leq x,y \leq 8$) die mittlere Helligkeit durch

$$\sigma_{n,b} = \frac{1}{64} \left(\sum_{x=0}^{7} \sum_{y=0}^{7} f_{n,b}(x,y) \right)$$

und die Kantenaktivität anhand der oben bestimmten λ_i durch

$$\eta_{n,b} = \frac{1}{8} \left(\sum_{i=0}^{7} \lambda_i \right)$$

berechnet und daraus durch

$$\rho_{n,b} = |\eta_{n,b} - \eta_{n-1,b}| + |\sigma_{n,b} - \sigma_{n-1,b}|$$

ein Maß für die Größe der Veränderung zwischen zwei Bildern ermittelt. Durch ρ wird zwar nicht direkt ein Maß für die Geschwindigkeit einer Bewegung berechnet, aber unter der Annahme, daß die Veränderung in einem Block steigt, je höher die Geschwindigkeit ist, sollte ρ von der Bewegungsgeschwindigkeit abhängen. Da langsame Bewegungen noch gut mit dem Auge verfolgt werden können, wird noch ein Threshold T_ρ eingeführt, den ρ überschreiten muß, bevor es sich auf die Quantisierung auswirkt:

$$q_{mtn} = \begin{cases} 0 & \text{falls } \rho \le T_\rho \\ \frac{\rho - T_\rho}{C_3} & \text{falls } \rho > T_\rho \end{cases}$$

C_3 ist mit 3 und T_ρ mit 25 angegeben.

Rate Control

Um die erzeugte Bitrate in einem vorgegebenen Bereich zu halten, wird für jedes Bild ein Wert q_{fld} so bestimmt, daß die Anzahl der Bits für das codierte Bild eingehalten wird. Da es, wie oben erwähnt, nicht praktikabel ist, die genaue Beziehung zwischen einem Block und der Anzahl der erzeugten Bits in Abhängigkeit von f anzugeben, wird ein *Ortsfrequenzvektor* mit niedrigerer Dimension bestimmt, der dennoch eine genaue Schätzung der Anzahl der erzeugten Bits erlaubt.

Grundlage dafür ist eine vereinfachte Variante der 2-dimensionalen Haar-Transformation. Ein Block D mit Helligkeitswerten $D(x,y)$, $0 \le x, y \le 7$ wird dabei in die Blöcke $D_0, \ldots, D_3$ zerlegt. D_0 enthält einfach den DC-Anteil des Blockes:

$$D_0(x,y) = \frac{1}{64} \sum_{i=0}^{7} \sum_{j=0}^{7} D(i,j)$$

Der Block D_1 stellt die Anteile der ersten harmonischen Frequenzkomponenten des Blocks D dar (mit $x', y' \in \{0,4\}$ und $x, y \in \{0,1,2,3\}$):

$$D_1(x+x', y+y') = \frac{1}{16} \sum_{i=x'}^{x'+3} \sum_{j=y'}^{y'+3} (D(x+x', y+y') - D_0(x+x', y+y'))$$

Entsprechendes wird für die zweiten harmonischen Frequenzkomponenten wiederholt (mit $x', y' \in \{0,2,4,6\}$ und $x, y \in \{0,1\}$):

$$D_2(x+x', y+y') = \frac{1}{4} \sum_{i=x'}^{x'+1} \sum_{j=y'}^{y'+1} (D(x+x', y+y') - D_1(x+x', y+y') - D_0(x+x', y+y'))$$

und schließlich noch einmal für die dritte:

$$D_3(x,y) = D(x,y) - D_2(x,y) - D_1(x,y) - D_0(x,y)$$

Man beachte, daß der Block D_0 nur einen konstanten Wert besitzt, Block D_1 nur aus 4 konstanten Blöcken der Größe 4×4 besteht und D_2 aus 4×4 Blöcken der Größe 2×2 besteht. Aus den Richtungsableitungen dieser Blöcke wird nun ein Vektor V erzeugt:

$$V = \begin{pmatrix} D_0(0,0) \\ \frac{1}{2}(\delta_v D_1 + \delta_h D_1) \\ \delta_v D_2 \\ \delta_h D_2 \\ \delta_v D_3 \\ \delta_h D_3 \end{pmatrix}$$

Dabei ist δ_h und δ_v ähnlich wie oben, nur diesmal auf 8×8-Blöcke definiert. Anhand von Trainingssequenzen wird nun ein Codebook erstellt, über das später durch Angabe von V und q die erwartete Anzahl von Bits zur Codierung dieses Blockes nachgesehen werden kann. Summiert man dies über alle Blöcke im Bild auf, erhält man die geschätzte Anzahl der Bits für das codierte Bild. Um nun ein Bild mit einer vorgegebenen Anzahl Bits zu codieren, wird q_{fld} solange verändert, bis die Schätzung der gewünschten Anzahl Bits entspricht.

Eine Schwäche dieses Ansatzes ist, daß davon ausgegangen wird, daß eine lineare Verschiebung aller im Bild vorhandenen q-Werte die Qualität gleichmäßig verändert. In Wirklichkeit ist jedoch die $q \leftrightarrow$ *Qualität*-Beziehung abhängig vom Blockinhalt. Das heißt also, daß selbst, wenn die q-Werte nach der Adaptiven Quantisierung so verteilt wurden, daß die Qualität über das ganze Bild gleichmäßig verteilt war, durch die Addition eines konstanten Wertes zu allen q-Werten, sich Unterschiede in der Qualität innerhalb eines Bildes ergeben können.

7.4.2 Klassifikationsverfahren von *Puri, Aravind*

Dieses Verfahren [PA91] bezieht sich direkt auf die Codierung nach dem MPEG-Standard und kombiniert Adaptive Quantisierung und Rate Control. Da sehr viele Tabellen mit statistischen Daten benutzt werden, muß für jede zu unterstützende Bitrate eine Reihe von Testsequenzen codiert werden, um die benötigten Daten zu erhalten. Weil diese Werte teilweise nur durch subjektive Tests ermittelt werden können, ist dies sehr aufwendig. Außerdem sind sämtliche Werte abhängig von der GOP-Struktur. Da für jede unterstützte GOP-Struktur eigene Datensätze notwendig wären, wird man sich sauf eine feste GOP-Struktur beschränken.

Abschätzung der Szenenkomplexität

Der erste Schritt des Verfahrend schätzt die Szenenkomplexität ab und verteilt davon ausgehend die Anzahl der Bits, die zur Codierung der verschiedenen Bildtypen benutzt werden dürfen. Die Szenenkomplexität wird jeweils während der Codierung der vorhergehenden GOP ermittelt, da angenommen wird, daß sich die Komplexität im Laufe der Zeit nur langsam ändert, was natürlich bei Szenenwechseln nicht der Fall sein muß. Je nach der Varianz der (Differenz-)bilder der I-, P- und B-Bilder werden diesen nach I, P und B getrennt die Bildkomplexitäten *Low*, *Medium*, *High* und *Very high* zugeteilt. Alle davon möglichen Kombinationen werden darauf wiederum in 8 Klassen eingeteilt, die nun die Szenenkomplexität angeben. Jede Klasse steht dabei für typische Vorgänge im Bild wie „Langsame Hintergrundverschiebung und einfache Bewegung in detailarmer Szene“ oder „Schneller Zoom mit komplexen neuen Bereichen in einer mittel- bis hochdetailierten Szene“. Ausgehend von dieser Szenenkomplexitäts-Klasse und der Zielbitrate werden den I-, P- und B-Bildern getrennte Bitraten zugeteilt.

Einteilung der Makroblöcke in *Statistical Activity Classes*

Die Makroblöcke werden im folgenden anhand ihrer Varianz im Helligkeitskanal in die drei Klassen *Ebene*, *Textur* und *Kanten* (oder allgemeiner: *Inhomogenität*) eingeteilt. Danach wird in einem zweiten Schritt jedem Makroblock des Typs *Ebene* oder *Textur* je nach der Größe der Varianz eine von 16 *Variance model activity class*es (VMAC, $V_0, \dots, V_{15}$) zugeteilt. In einem *Kanten*-Makroblock wird dagegen jedem in ihm enthaltenen Block je nach Stärke der Kante eine von 4 *Edge model activity class*es (EMAC, $E_0, \dots, E_3$) zugeteilt.

Zur Bestimmung der Makroblockklasse seien $v_1, \dots, v_4$ die Varianzen der 4 im Makroblock enthaltenen 8×8-Blöcke und sämtliche auftretenden T_i geeignet zu wählende Thresholds. Weiterhin sei $v_{avg} = \frac{1}{4}(v_1 + v_2 + v_3 + v_4)$, $v_{max} = \max(v_1, v_2, v_3, v_4)$ und $v_{min} = \min(v_1, v_2, v_3, v_4)$. Ein Makroblock wird als *Ebene* klassifiziert, wenn gilt:

$$v_{avg} < T_1 \wedge v_{max} < T_2$$

Erfüllt er diesen Test, wird ihm abhängig von v_{avg} eine VMAC-Klasse zugeordnet. Ansonsten wird er dem Texturtest unterzogen:

$$v_{min} > T_1 \wedge \left(\Big(\underset{i \neq j \neq k \neq i}{\exists} : (\frac{v_{max}}{v_i} < T_3) \wedge (\frac{v_{max}}{v_j} < T_3) \wedge (\frac{v_{max}}{v_k} < T_4) \Big) \vee \underset{0 \leq i \leq 3}{\forall} : \frac{v_{max}}{v_i} < T_5 \right)$$

Wird dieser Test erfüllt, so handelt es sich um einen Textur-Block und auch ihm wird eine VMAC-Klasse zugeordnet. War auch der Texturtest nicht erfolgreich, werden den in ihm enthaltenen Blöcken EMAC Klassen zugeordnet. Das Vorgehen ist dabei so aufwendig, daß ich hier nur auf den Originalartikel verweisen will.

Diese Klassifizierung der Makroblöcke wurde testweise in SAMPEG implementiert, aber auch hier konnte die erwartete Trennschärfe zwischen den Klassen nicht erreicht werden.

Wahrgenommene Bildqualität und Bitverteilung

Um eine erreichte Bildqualität zu beurteilen, wurden sämtliche Artefakttypen getrennt mit einer Bewertung versehen, wie sichtbar dieser Artefakt subjektiv je nach objektiver Stärke ist. Durch Aufsummieren dieser Bewertungen wurde dann die ganze Bildqualität bewertet und durch Testreihen wurde ermittelt, unter welcher Bitrate und Szenenkomplexität man mit welcher Bildqualität rechnen muß.

Die Aufgabe ist nun, für jeden geeigneten Makroblock abhängig von der erwarteten Bildqualität einen Wert für *qscale* zu vergeben, so daß die Gesamtsumme der vergebenen Bits pro Makroblock der Zielvorgabe für das Gesamtbild entspricht. Dazu wurden zuvor für sämtliche möglichen *qscale*s in Abhängigkeit der VMAC- und EMAC-Klassen die zu erwartenden Bits gemessen und daraus sogenannte Quantisierungs-Strings ermittelt. Für jede Ziel-Qualität existiert so ein Quantisierungs-String, der jeder VMAC- und EMAC-Klasse einen *qscale*-Wert zuordnet. Es existieren dabei noch Feinheiten, wie unterschiedliche Strings für verschiedene durchschnittliche Makroblockhelligkeiten, die hier nicht beschrieben werden sollen.

Rate Control

Da mit den bisherigen Maßnahmen immer noch nicht garantiert ist, daß eine vorgegebene Bitrate eingehalten wird, wird in regelmäßiger Abständen der Bufferfüllstand gemessen. Je nach Bufferfüllstand werden Korrektur-Strings dazu benutzt, die *qscale*-Werte ggf. zu erhöhen, um einen Bufferüberlauf zu verhindern.

7.4.3 Klassifikationsverfahren von *Chun, Lim, Cho, Ra*

Auch dieses Verfahren aus [CLCR93] nimmt eine Klassifizierung der Blöcke in die Typen *Ebene*, *Kante* und *Textur* vor. Es bezieht sich nicht direkt auf MPEG-2 und arbeitet mit einer Makroblockgröße von 4×2 Blöcken zu je 8×8 Pixel. Die Grundidee ist bei diesem Verfahren, die Klassizierung an den (noch unquantisierten) DCT-Transformationskoeffizienten durchzuführen. Dazu werden in jedem DCT-Block drei Bereiche untersucht (siehe Abbildung 7.3). Für jeden Bereich wird der Energiegehalt berechnet. Der Anteil der horizontalen Energie wird durch E_h, der vertikale durch

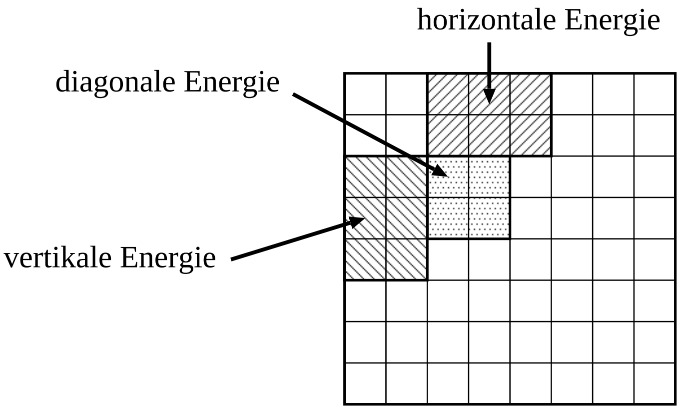


Abbildung 7.3: Energieverteilung im Frequenzraum

E_v und der diagonale durch E_d bezeichnet. Weiterhin sei $E_a = \frac{1}{3}(E_h + E_v + E_d)$, E_m das Minimum und E_M das Maximum von E_h, E_v und E_d. Anhand dreier Thresholds T_1, T_2, T_3 wird nun eine Klassifizierung durchgeführt:

$$\text{Blocktyp} = \begin{cases} \text{Ebene} & \text{für } E_a \leq T_1 \\ \text{Textur} & \text{für } E_a > T_1 \\ & \text{oder } E_m > T_2 \wedge \frac{E_m}{E_M} > T_3 \\ \text{Kante} & \text{sonst} \end{cases}$$

Die Anzahl der Blöcke in einem Makroblock, die in eine dieser Klassen fallen, werden gezählt und in einem Vektor (#Ebene, #Kante, #Textur) zusammengefaßt. Anhand dieses Vektors wird nun der Gesamttyp des Makroblocks bestimmt, wie in Abbildung 7.4(a) zu sehen ist. Ähnlich wie bei der Zig-Zag-Anordnung der DCT-Koeffizienten für die Lauflängencodierung werden auch hier die Makroblöcke so angeordnet, daß Makroblöcke, bei denen Codierungsfehler auffälliger sind, sich weiter vorne befinden. Die Adaptive Quantisierung wird nun durchgeführt, indem *qscale* für einen Makroblock umso größer gewählt wird, je weiter hinten er sich in der linearen Anordnung befindet. Die Rate Control kann mit einem traditionellen Verfahren durchgeführt werden, das anhand der aktuellen Bufferfüllung eine Verschiebung des Quantisierungsfaktors durchführt.

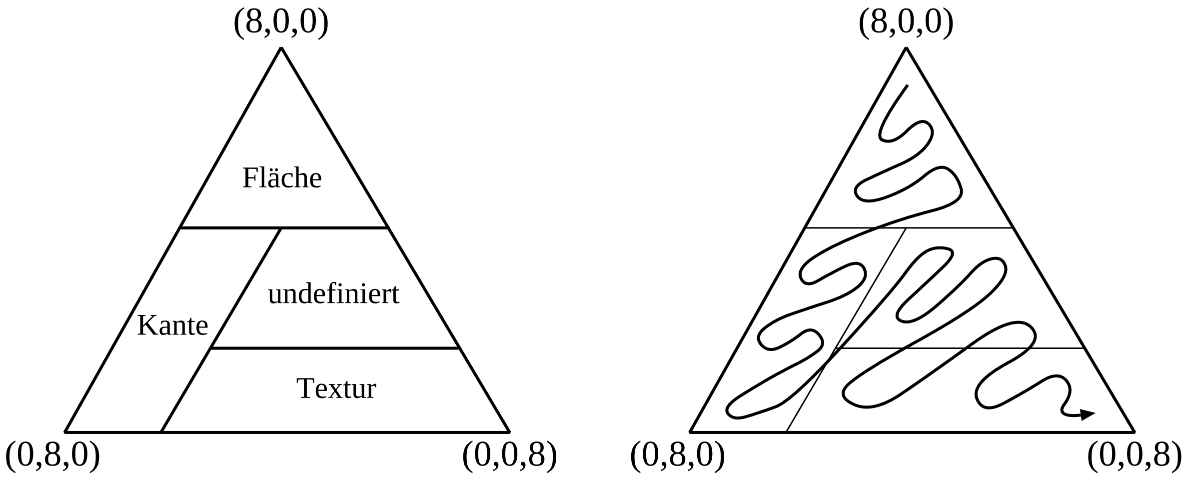
Da dieses Verfahren mit Makroblöcken arbeitet, die sich aus 4×2-Blöcken zusammensetzen, läßt es sich nicht direkt für MPEG-2 einsetzen, es sei denn, man beschränkt sich darauf, maximal für jeden zweiten MPEG-2-Makroblock den *qscale*-Faktor zu verändern.

7.4.4 Verfahren von *TestModel 5 (MSSG)*

Dieses Verfahren arbeitet in drei Stufen. In der ersten werden ausgehend von der Komplexität der verschiedenen Bildtypen dem jeweils nächsten zu codierenden Bild die Anzahl der zur Verfügung stehenden Bits zugewiesen. Die zweite Stufe überwacht den Füllstand des Ausgabebuffers und regelt damit den mittleren *qscale*-Wert der Makroblöcke. Dieser Wert wird im dritten Schritt durch eine Adaptive Quantisierung moduliert.

Allokation von Bits für das nächste Bild

Für jeden Bildtyp wird anhand der Anzahl der Bits, die zur Codierung des letzten Bildes dieses Typs gebraucht wurden (S_i, S_p, S_b) und dem mittleren *qscale*-Wert dieses Bildes (Q_i, Q_p, Q_b) die



(a) Bestimmung des Makroblocktyps anhand der Typverteilung der in ihm enthaltenen Blocktypen

(b) Lineare Anordnung der Makroblocktypen

Abbildung 7.4: Bestimmung des Makroblocktyps und lineare Anordnung der Makroblocktypen. Jedem Punkt im Dreieck entspricht einem möglichen Vektor aus den Anzahlen der Blöcke eines Typs.

Bildkomplexität abgeschätzt:

$$X_i = S_i \cdot Q_i \qquad X_p = S_p \cdot Q_p \qquad X_b = S_b \cdot Q_b$$

Nun bezeichne R die Anzahl der Bits, die noch zur Codierung der aktuellen GOP zur Verfügung stehen. R wird nach der Codierung jedes Bildes um die entsprechende Anzahl Bits erniedrigt und zu Beginn jeder neuen GOP um den Wert $bitrate \cdot GOPlength/framerate$ erhöht, also der Anzahl der Bits, die während der Zeitdauer einer GOP übertragen werden. N_p sei die Anzahl der P-Bilder die noch in der aktuellen GOP zu codieren sind und N_b entsprechend die Anzahl der noch zu codieren B-Bilder in der aktuellen GOP. Weiterhin seien K_p und K_b zwei Konstanten, die im TM5-Encoder mit $K_p = 1.0$ und $K_b = 1.4$ gewählt wurden. Die Anzahl der Bits, die an ein Bild vergeben werden, berechnet sich nun je nach Bildtyp durch:

$$T_i = \frac{R}{1 + N_p \frac{X_p}{X_i K_p} + N_b \frac{X_b}{X_i K_b}} \qquad T_p = \frac{R}{N_p + N_b \frac{X_b K_p}{X_p K_b}} \qquad T_b = \frac{R}{N_b + N_p \frac{X_p K_b}{X_b K_p}}$$

Rate Control

Um die Differenz zwischen erzeugter und gewünschter Bitrate zu überwachen und gegenzuregeln, wird für jeden Bildtyp ein virtueller Buffer benutzt. Während der Codierung eines Bildes wird der entsprechende Buffer mit den bisher codierten Bits gefüllt und dieser Füllstand wird mit dem theoretisch erwarteten Füllstand verglichen. Die Differenz dient zur Regelung der Quantisierung. Betrachten wir die Codierung eines Bildes vom Typ x mit $MBcnt$ Makroblöcken. Den realen Bufferfüllstand nach der Codierung von i Makroblöcken bezeichnen wir durch B_i^x. Nach der Theorie

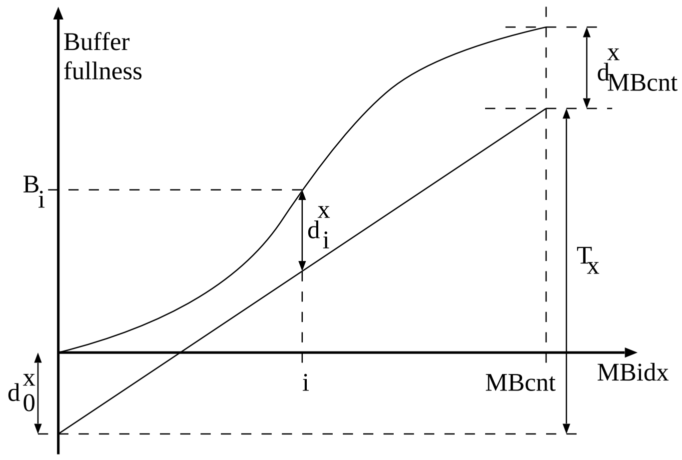


Abbildung 7.5: Rate Control anhand des virtuellen Bufferfüllstands bei TM5.

sollte sich der Buffer im Idealfall mit konstanter Geschwindigkeit über das Bild füllen. Nach der Codierung von i Makroblöcken sollten also

$$\frac{T_x \cdot i}{MBcnt}$$

Bits im Buffer enthalten sein. Dies wird aber nicht der Realität entsprechen und wir berechnen deshalb die Abweichung (Siehe Abbildung 7.5[2])

$$d_j^x = d_0^x + B_{j-1}^x - \frac{T_x \cdot (j-1)}{MBcnt}$$

Zur Regelung berechnen wir nun $qscale$ durch

$$qscale = \frac{d_j^x \cdot 31}{r}$$

mit dem Reaktionsparameter $r = 2 \cdot bitrate/framerate$.

Adaptive Quantisierung

Die Adaptive Quantisierung orientiert sich an der Varianz der Blöcke. Dazu werden die Varianzen $v_1, \ldots, v_4$ der 4 Blöcke des Makroblocks berechnet (ich vereinfache hier den Vorgang etwas, da in TM5 für die Codierung von Interlaced-Material die Zerlegung des Makroblocks etwas anders vorgenommen wird). Daraus ergibt sich ein Aktivitätswert für Makroblock j:

$$act_j = 1 + \min(v_1, v_2, v_3, v_4)$$

Anhand der im während der Codierung des letzten Bildes gewonnenen mittleren Aktivität act_{avg} wird diese noch normalisiert:

$$\widehat{act}_j = \frac{(2act_j) + act_{avg}}{act_j + 2act_{avg}}$$

und dient nun zur Modulation von $qscale$:

$$qscale_j := qscale_j \cdot \widehat{act}_j$$

[2]Diagramm wurde einer Zeichnung in der Dokumentation von TM5 nachempfunden.

Bewertung

Ziel von TM5 war es nicht, ein ausgefeiltes System zur Adaptiven Quantisierung zu implementieren, da das Programm nur als Referenzmodell für andere Implementierungen dienen sollte. So berücksichtigt die Stufe der Adaptiven Quantisierung nur ungenügend, daß Makroblöcke mit Kanten fein quantisiert werden müssen. Dies wird nur erreicht, wenn ein Block des Makroblocks komplett in einem ebenen Bereich liegt. Weitere Schwachpunkte sind, daß durch die Abschätzung von Parametern aus vergangenen Codierungsschritten bei einem Szenenwechsel diese Parameter gegebenenfalls unpassend gewählt werden. Dies betrifft sowohl die Bitallokation in der ersten Stufe und die Adaptive Quantisierung in der dritten Stufe. Zwar fällt eine schlechte Codierung an diesen Stellen wegen des Szenenwechsels nicht so sehr auf, aber durch die Codierungsineffizienz wird eine generell niedrigere Bildqualität erzeugt. Die Rate Control in der zweiten Stufe ist nicht VBV konform. Allerdings ist keines der oben betrachteten Verfahren VBV konform.

7.5 Implementierung bei SAMPEG

Das implementierte Verfahren basiert auf Ideen aus den oben beschriebenen Verfahren, benutzt aber davon abweichende Methoden zur Bestimmung der Maskierungen und der geschätzten Anzahl Bits pro Makroblock. Auch das SAMPEG-Verfahren arbeitet auf mehreren Ebenen. Zuerst wird für eine zu codierende GOP die Anzahl der Bits festgelegt, die ein Frame benutzen darf. Die zweite Ebene besteht aus mehreren individuell zuschaltbaren Modulen zur Adaptiven Quantisierung, die das bisher beliebig gesetzte, aber konstante *qscale* modulieren. Eine dritte Ebene versucht nun, die *qscale*-Werte in einem Bild so durch Addition eines Offsets zu verschieben, daß die erzeugte Anzahl von Bits dem in der ersten Ebene bestimmten Wert entspricht. Eine vierte Ebene ist die Szenenerkennung, die auch auf den *qscale*-Wert eines Bildes einwirken kann.

7.5.1 Bitallokation

Die Allokation von Bits für die Bilder wird immer für eine gesamte GOP durchgeführt, damit die Codierung der Bilder parallel laufen kann. Sei die zu erzeugende Bitrate gegeben durch r, die Bildrate durch f und die Anzahl der Bilder in der aktuellen GOP durch n_I, n_P und n_B, was sich aufgrund der Szenenerkennung von Fall zu Fall ändern kann. Zusätzlich benutzen wir noch einen *Buffer-Mismatch*-Wert m, welcher die aufsummierten Abweichungen der tatsächlich codierten Bits von der vorhergesagten Anzahl enthält. Zu Beginn der Codierung setzen wir m auf 0. Wir legen das Größenverhältnis zwischen den Bildtypen durch zwei Konstanten C_{PI} und C_{BI} fest. Dabei legt C_{PI} fest, um welchen Faktor ein P-Bild kleiner sein soll, als ein I-Bild. Entsprechend ist C_{BI} definiert.

Damit können wir nun die Größen S_I, S_P, S_B der Bilder in der GOP berechnen:

$$S_I = \frac{m + r \cdot (n_I + n_P + n_B)/f}{n_I + n_P/C_{PI} + n_B/C_{BI}}$$

$$S_P = \frac{S_I}{C_{PI}} \qquad S_B = \frac{S_I}{C_{BI}}$$

Nachdem ein Bild komplett codiert worden ist, wird die Differenz aus der realen Größe und der hier berechneten Größe zu dem Buffer-Mismatch-Wert m addiert. Je nach Vorzeichen von m stehen dann in der nächsten GOP mehr oder weniger Bits als im Durchschnitt zur Verfügung, womit erreicht wird, daß eine anvisierte Bitrate genau eingehalten wird.

7.5.2 Luminance Masking

Hierbei wird der in Abschnitt 2.4.4 auf Seite 14 beschriebene Effekt ausgenutzt, daß Codierungsfehler besonders in Regionen mit mittlerer Helligkeit erkannt werden. Durch die mittleren Helligkeit eines Makroblocks, die schon bei der Szenenerkennung berechnet wurde und deshalb ohne Mehraufwand zur Verfügung steht, wird anhand der in Abbildung 7.6 dargestellten Funktion der *qscale*-Wert des Makroblocks erhöht, falls der Makroblock sehr dunkel oder sehr hell ist. Dadurch,

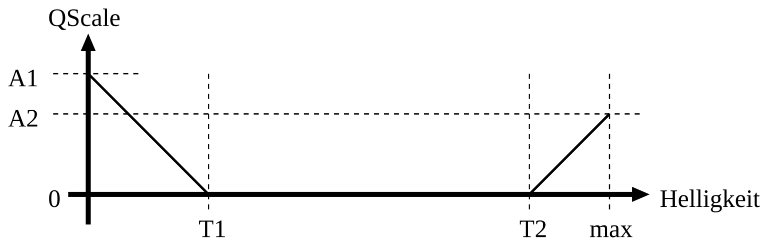


Abbildung 7.6: Luminance Masking

daß die Bitrate insgesamt konstant gehalten wird, sinken damit alle im Bild vorhandenen *qscales* etwas und die Gesamtqualität steigt. Dies geschieht zwar auf Kosten der dunklen und hellen Makroblöcke, aber dort fallen Fehler nicht so sehr auf. Die benutzten Konstanten T_1, T_2, A_1 und A_2 wurden experimentell ermittelt.

7.5.3 Activity Masking

Nachdem die Ergebnisse sämtlicher Versuche mit einem Klassifizierungsverfahren nicht robust genug erschienen, wird in SAMPEG jetzt ein Verfahren benutzt, das zwar das gleiche Verhalten wie ein Klassifizierungsverfahren zeigt, jedoch ohne den Klassifizierungsschritt auskommt. Der erste Schritt besteht darin, genau wie in Abschnitt 7.4.1 beschrieben, die Werte $\lambda_0, \ldots, \lambda_3$ zu berechnen. Der Grund dafür, daß anstatt der Varianz die Beträge der Richtungsableitungen benutzt werden ist, daß die Varianz keine Aussage darüber macht, wie die Helligkeitswerte innerhalb des Blockes verteilt sind. Wenn wir die beiden Blöcke in Abbildung 7.7 betrachten, so sieht man, daß beide zwar

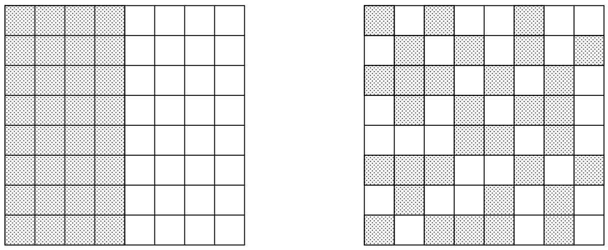
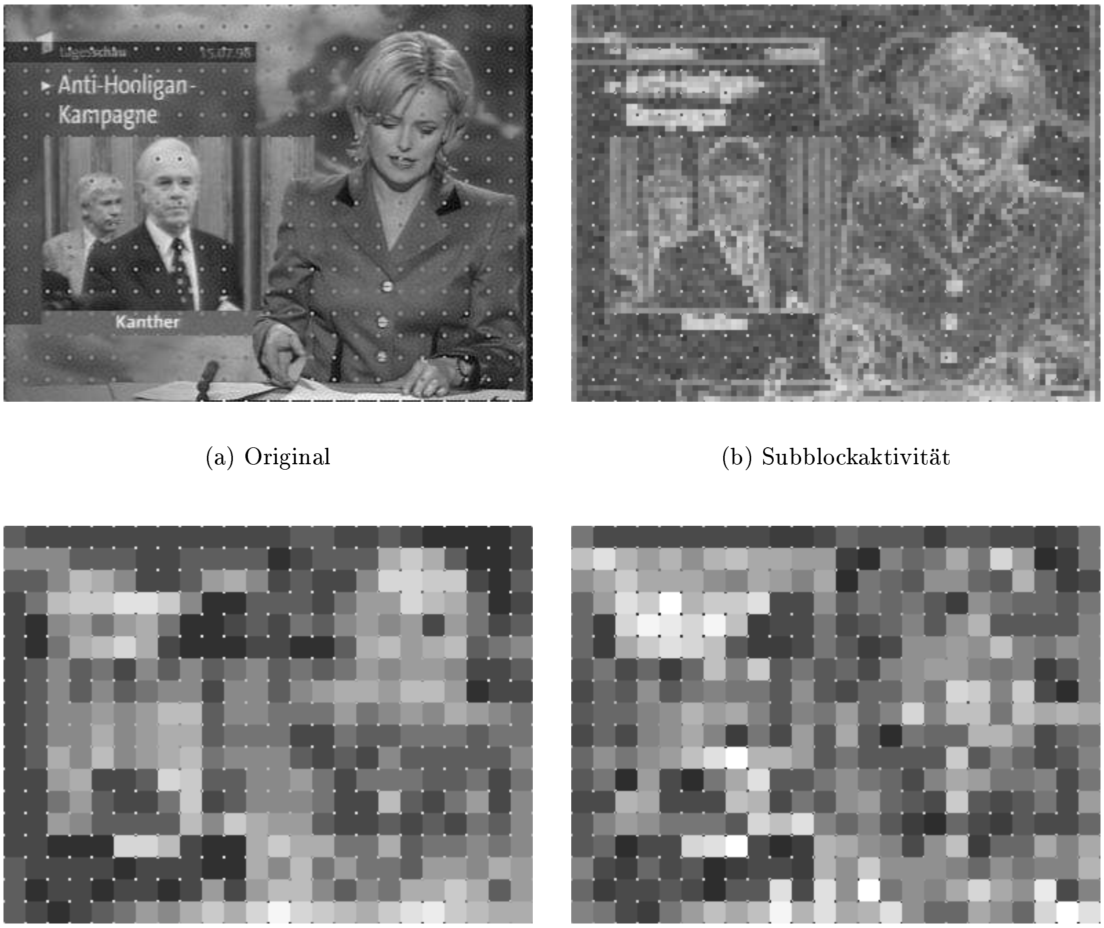


Abbildung 7.7: Beide Blöcke besitzen die gleiche Varianz, jedoch verschiedene Richtungsableitungen.

die gleiche Varianz haben, aber der linke wegen des drohenden Ringing-Effekts feiner quantisiert werden muß als der der rechte, der eine Textur enthält und sehr grob quantisiert werden kann. Das in SAMPEG benutzte Maß

$$\lambda_i = subact_i = \ln\left(\sum_{x=0}^{N-1}\sum_{y=0}^{M}|f(x+1,y)-f(x,y)| + \sum_{x=0}^{N}\sum_{y=0}^{M-1}|f(x,y+1)-f(x,y)|\right)$$

eignet sich besser, da es für Kantenblöcke niedrigere Werte als für Texturblocke und selbstverständlich auch für Flächenblöcke niedrigere Werte als für Kantenblöcke ergibt. Außerdem läßt es sich auch schneller berechnen als die Varianz. Man beachte, daß die Berechnung der λ_i in Anlehnung an das menschliche Sehsystem erfolgt. Die Differenzbildung entspricht in etwa dem Aufbau der rezeptiven Felder und die Logarithmierung dem Weber-Fechner-Gesetz. Die Subblockaktivität λ_i wird nun unabhängig für alle 4×4-Subblöcke eines Makroblocks berechnet. Abbildung 7.8(b) zeigt die Subblockaktivitäten für das Bild aus Abbildung 7.8(a). Für jeden Makroblock werden jetzt



(a) Original

(b) Subblockaktivität

(c) Busyness

(d) Ringing

Abbildung 7.8: Adaptive Quantisierung bei SAMPEG

zwei Werte berechnen: die *Busyness*, welche Beschreibt wie hoch die Texturstärke in einem Makroblock ist und das *Ringing*, das die Gefahr beschreibt, daß ein Ringing-Effekt auftritt. Die Busyness eines Makroblocks ist einfach der Mittelwert der im Makroblock enthaltenen Subblockaktivitäten. Für die Berechnung des Ringing seien $subact_{x,y}$ mit $0 \leq x \leq 3$ die Subblockaktivitäten in einem Makroblock. Das Ringing wird berechnet durch

$$ringing = \sum_{x=0}^{2} \sum_{y=0}^{3} |subact_{x+1,y} - subact_{x,y}| + \sum_{x=0}^{3} \sum_{y=0}^{2} |subact_{x,y+1} - subact_{x,y}|$$

Folglich wird durch *ringing* beschrieben, wie sehr sich die Texturstruktur innerhalb eines Blockes verändert. Enthält ein Makroblock zum einen ein stark strukturiertes Gebiet und eine gleichmäßige Fläche, so wird *subact* im ersten Gebiet groß und im zweiten klein sein; es ergibt sich ein hoher *ringing*-Wert. Die Adaptive Quantisierung wird nun dadurch erwirkt, daß der Wert

$$C_1 \cdot busyness^{C_2} - C_3 \cdot ringing^{C_4}$$

zu *qscale* addiert wird. C_1 bis C_4 sind wieder feste, durch subjektive Versuche ermittelte Konstanten. In den Abbildungen 7.8(c) und 7.8(d) sind die *busyness*-Werte und *ringing*-Werte für ein Bild dargestellt. Zu erkennen sind die hohen *busyness*-Werte in der Schrift und den Haaren der Sprecherin. Würde man diesen Wert alleine zur Steuerung der Adaptiven Quantisierung benutzen, erhielte man in der Schrift einen starken Ringing-Effekt, wogegen dieser in den Haaren nicht auffallen würde, da hier eine gleichmäßige Textur vorliegt. Betrachten wir nun die Verteilung des *ringing*-Wertes, so sehen wir, daß auch dieser an der Stelle, an der die Schrift im Bild ist, sehr hoch ist. Dadurch verliert der *busyness*-Wert an dieser Stelle seine Wirkung. Man sieht, daß der *ringing*-Wert besonders zwischen den Textzeilen sehr groß ist, also genau an der Stelle, an der der Ringing-Effekt besonders sichtbar ist. Umgekehrt sieht man, daß der *ringing*-Wert anders als der *busyness*-Wert in den Haaren der Sprecherin relativ klein ist. Hier wird also die Quantisierung vergröbert, was infolge fehlender Kanten nicht so sehr auffällt.

7.5.4 Activity Masking mit JND-Map

Ein zweiter Ansatz zum Activity Masking, der aber dem zuletzt beschriebenen sowohl in der Leistung, als auch in der Geschwindigkeit unterlegen ist, soll hier kurz erwähnt werden. Dieser Ansatz beruht darauf, daß versucht wird, für jedes Bild eine sogenannte JND-Map (JND = just noticeable difference) aufzubauen. Für jede Bildstelle wird dabei berechnet, wie wahrscheinlich es ist, daß ein menschlicher Betrachter hier einen Fehler einer bestimmten Stärke erkennt. Auf Grundlage der Erkenntnis, daß ein großer Helligkeitssprung im Bild dazu führt, daß man in seiner Umgebung Fehler schlechter wahrnimmt, wird zuerst für ein Bild f wieder die Richtungsableitungen gebildet, so daß ein neues Bild $\hat{f}$ entsteht, das diesen Inhalt hat:

$$\hat{f}(x,y) = \frac{1}{2}|f(x+1,y) - f(x,y)| + \frac{1}{2}|f(x,y+1) - f(x,y)|$$

Da sich dieser Maskierungseffekt auf ein kleines Gebiet um den Helligkeitssprung auswirkt, wird in etwa eine Tiefpassfilterung auf $\hat{f}$ vorgenommen. Die Berechnung ist aber etwas anders. In unserer JND-Map $j(x,y)$ soll jede Position die Wahrscheinlichkeit angeben, daß ein Fehler nicht gefunden wird (wir berücksichtigen hier nicht die Stärke des Fehlers, da dieses nur einen konstanten Faktor zur Folge hätte). An allen Stellen, an denen ein Helligkeitssprung der Stärke d (normalisiert auf $[0;1]$) auftritt, nehmen wir an, daß dadurch die Erkennungswahrscheinlichkeiten für den Fehler in der Umgebung dieses Sprunges

$$\underline{\mathrm{P}}(\text{Fehler erkannt}) = d \cdot \begin{pmatrix} 0.0 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.0 \\ 0.1 & 0.2 & 0.3 & 0.3 & 0.3 & 0.3 & 0.2 & 0.1 \\ 0.1 & 0.3 & 0.6 & 0.7 & 0.7 & 0.6 & 0.3 & 0.1 \\ 0.1 & 0.3 & 0.7 & 1.0 & 1.0 & 0.7 & 0.3 & 0.1 \\ 0.1 & 0.3 & 0.7 & 1.0 & 1.0 & 0.7 & 0.3 & 0.1 \\ 0.1 & 0.3 & 0.6 & 0.7 & 0.7 & 0.6 & 0.3 & 0.1 \\ 0.1 & 0.2 & 0.3 & 0.3 & 0.3 & 0.3 & 0.2 & 0.1 \\ 0.0 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.0 \end{pmatrix}$$

beträgt. Der Helligkeitssprung befindet sich hierbei zwischen Element $\underline{P}_{4;4}$ und $\underline{P}_{5;4}$ bzw. $\underline{P}_{4;5}$. Eine Stelle, die durch zwei Helligkeitssprünge in ihrer Umgebung und den dabei bei ihr auftretenden Erkennungswahrscheinlichkeiten p_1 und p_2 maskiert wird, erhält die Erkennungswahrscheinlichkeit $p = 1 - (1 - p_1) \cdot (1 - p_2)$. Eine JND-Map, die so erhalten wurde, ist in Abbildung 7.9 dargestellt. Zur Adaptiven Quantisierung wird nun jeweils für jeden Makroblock der Mittelwert der JND-Map

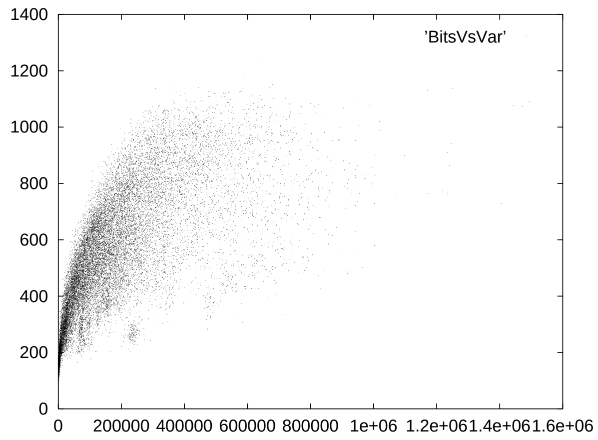
Abbildung 7.9: JND-Map für das Beispielbild

im Bereich des Makroblocks berechnet und *qscale* um so mehr erhöht, je größer dieser Wert ist. Wie bereits erwähnt, ist die Leistung dieser Methode nicht so gut, wie die des im letzten Abschnitts beschriebe. Es ist aber anzunehmen, daß durch ausgefeiltes Modell der menschlichen Wahrnehmung und einer entsprechenden besseren Generierung der JND-Map sich auch mit diesem Verfahren sehr gute Ergebnisse (wenn auch mit hohem Rechenaufwand) erzeugen lassen.

7.5.5 Rate Control

Haben wir die Adaptive Quantisierung durchgeführt, so ist jedem Makroblock ein *qscale*-Wert zugewiesen. Allerdings ist die dadurch erzeugte Bitzahl in diesem Bild nicht die, die wir im ersten Schritt dem Bild zugewiesen haben. Deshalb muß jetzt allen *qscale*-Werte gemeinsam ein konstanter Wert addiert werden, damit der erzeugten Bits der gewünschten Größe möglichst nahekommt. Wir schätzen dazu die Anzahl der Bits für jeden Makroblock getrennt und summieren diese Werte über das ganze Bild auf.

Zur Abschätzung benutzen wir, abweichend von den meisten Verfahren nicht die Varianz, sondern wieder die Summe der absoluten Richtungsableitungen $\delta_h M + \delta_v M$ des Makroblocks M. In Abbildung 7.10(a) wurde für eine Testsequenz die Anzahl der Bits in einem Makroblock in Abhängigkeit von seiner Varianz ermittelt. Man sieht, daß die Werte eine so große Streuung haben, daß die Abschätzung nicht sehr genau werden kann. Benutzen wir dagegen die Summe der absoluten Richtungsableitungen, wie in Abbildung 7.10(b) dargestellt, so erhalten wir eine vergleichsweise geringe Steuung. Die in den Abbildungen gezeigten Daten beziehen sich auf die Codierung von Intra-Makroblocks. Für Makroblocks, die mit forward, backward, oder bidirektionaler Prädiktion codiert wurden, werden die Varianz bzw. Summe der absoluten Richtungsableitungen der Differenzbilder benutzt. Es ergibt sich dabei in beiden Fällen ein prinzipiell gleicher Verlauf. Die Messdaten wurden stückweise durch Geraden approximiert und diese als Tabellen in die SAMPEG-Implementierung übernommen. Zur Approximation wird nun für jeden Makroblock je nach Makroblockcodierungstyp (Intra, forward-, backward-, bidirectional-prediction) in der zugehörigen Tabelle nachgesehen und die Anzahl der Bits aus den vorhandenen Tabelleneinträgen interpoliert. Da die Schätzung

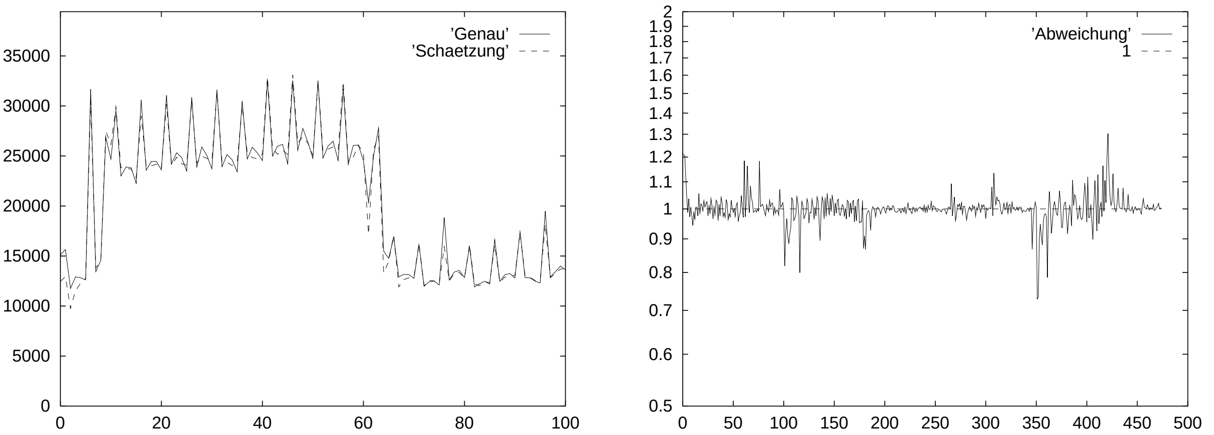


(a) Bits pro Makroblock in Abhängigkeit seiner Varianz

(b) Bits pro Makroblock M in Abhängigkeit von $\delta_h M + \delta_v M$

Abbildung 7.10: Adaptive Quantisierung bei SAMPEG

bei manchen Szenen generell über oder unter dem realen Wert liegt, wird noch für jeden Bildtyp getrennt ein Korrekturfaktor c_i, c_p, c_b eingeführt. Jedesmal, wenn ein Bild komplett codiert wurde und somit seine reale Größe bekannt ist, wird das Verhältnis c' von geschätzer zu realer Bildgröße gebildet. Der jeweilige Korrekturfaktor wird dann durch $c' := \frac{1}{2}(c' + c_x)$ angepaßt und auf den Wertebereich $[0,5;2]$ beschränkt. Dadurch ergeben sich gute Schätzungen, wie in Abbildung 7.11 zu sehen ist.



(a) Absolute geschätzte und reale Bildgrößen

(b) Quotient aus realer und geschätzter Bildgröße

Abbildung 7.11: Zutreffsicherheit der Schätzung einer Bildgröße

Es bleibt noch die Frage, wie ein geeigneter Offset q_{offs} für $qscale$ gefunden wird, damit das Bild in der richtigen Größe codiert wird. Betrachten wir dazu Abbildung 7.12, das den ungefähren Verlauf der Bildgröße in Abhängigkeit des $qscale$-Offsets angibt. Die Zielgröße ist durch die Horizontale vorgegeben. Ausgehend von zwei sicheren Werten für q_{offs} wird nun der ideale Wert

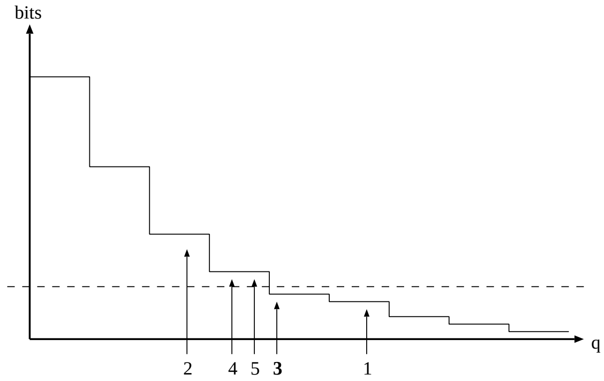


Abbildung 7.12: Binäre Suche

gesucht, indem auf dieser nicht explizit gegebenen Funktion eine binäre Suche durchgeführt wird, wobei aber zwei wesentliche Unterschiede zu einer gewöhnlichen binären Suche bestehen. Einmal wird die Suche nicht erst dann abgebrochen, wenn die Bildgröße genau mit dem Soll-Wert übereinstimmt (was aufgrund der Unstetigkeit der Funktion auch nicht geht), sondern schon nach einer festen Anzahl Interationen. Andererseits ist es hier so (auch da die Funktion unstetig ist), daß sich die Schätzung nicht unbedingt in jedem Schritt verbessern muß. In der Abbildung 7.12 ist z.B. zu sehen, daß das beste Ergebnis schon in Schritt 3 erreicht wird und die folgenden Schritte eine Verschlechterung darstellen. Aus diesem Grund wird nicht der Wert des letzten Schrittes als Ergebnis der Suche benutzt, sondern der beste bis dahin aufgetretene Wert für q_{offs}.

7.5.6 Bewertung

Da sich die subjektive Qualität nicht in objektive Zahlen fassen läßt, ist es schwer, eine Aussage über die durch die Adaptive Quantisierung erreichte Bildqualität zu machen. Es ist aber deutlich sichtbar, daß dadurch eine Verbesserung erzielt wird. Dabei scheint das Luminance-Masking die besten Ergebnisse zu liefern, gefolgt vom Activity-Masking. Das Activity-Masking mit JND-Ansatz ist dem anderen Ansatz aber im jetzigen Stadium sowohl in erreichter Qualitätsverbesserung, wie auch in der Rechnenzeit unterlegen. Da Luminance-Masking und Activity-Masking sich nicht ausschließen, sollten beide zur Codierung immer aktiviert werden.

Der derzeit größte Schwachpunkt scheint die von der Szenenkomplexität unabhängige Vergabe der Bits an die Bildtypen zu sein. Bei Szenen, in denen die meisten Stellen des Bildes unverändert bleiben (z.B. Nachrichtensendungen) kann man viel mehr Bits an die I-Bilder zuweisen, wovon man auch bessere Prädiktionen bekommt. Gute Werte sind in diesem Fall $C_{PI} = 4$ und $C_{BI} = 10$. Bei Szenen, in denen viel Bewegung herrscht, ist jedoch eine gleichmäßigere Verteilung günstiger. Gute Werte sind hier $C_{PI} = 2$ und $C_{BI} = 6$ bis hin zu $C_{PI} = 1,5$ und $C_{BI} = 3,0$ für sehr komplexe Szenen. Eine automatische Erkennung der Szenenkomplexität kann, wie unter [PA91] (Abschnitt 7.4.2) beschrieben, nachträglich noch hinzugefügt werden. Eine andere Möglichkeit der Implementierung wäre es, aus den Differenzbildern zweier aufeinanderfolgender Bilder abzuschätzen, wieviel Veränderung existiert und diesen Wert zur Berechnung geeigneter C_{PI} und C_{BI} zu benutzen.

Kapitel 8

Parallelisierung

8.1 Methoden den Parallelisierung

Bei der Parallelisierung sind 3 Ebenen unterscheidbar, auf denen parallelisiert werden kann. Die gröbste davon ist die Verteilung der Arbeit auf mehrere Rechner, eine weitere die Verteilung auf mehrere Threads innerhalb einer Mehrprozessormaschine und schließlich durch die Ausnutzung von SIMD-Befehlen, wie sie heute übliche Prozessoren meist speziell für die Unterstützung von Multimedia-Anwendungen enthalten.

8.1.1 Verteilung auf mehrere Rechner

Da unkomprimierte Videodaten eine sehr hohe Datenrate benötigen, sollten die Daten möglichst nicht zentral an einem Rechner eingespeißt und über ein Bus-Netzwerk verteilt werden, da man damit schnell an die Grenzen des Netzwerks stoßen dürfte. Abhilfe könnte ein sternförmiges Netzwerk bilden, wobei aber der zentrale Rechner die Rohdaten mit voller Bandbreite übertragen können muß. Diese Lösung dürfte nicht praktikabel sein.

Ein weiterer Nachteil der auf mehrere Rechner verteilten Echtzeit-Codierung ist, daß die Codierungsverzögerung mit der Anzahl der Rechner steigt, denn da die Bilder nicht unabhängig voneinander codiert werden können, muß die Aufgabe zumindest GOP-weise unterteilt werden. Bei einer GOP-Länge von 20 Bildern und 5 Rechner hätte man damit schon eine Mindestverzögerung von 4 Sekunden.

Sinnvoll ist dieser Ansatz höchstens für die Offline-Codierung, wenn eine hohe Qualität erwünscht wird und damit relativ viel Rechenzeit benötigt wird. Dieser Fall läßt sich jedoch sehr einfach realisieren, indem der zu codierende Stream einfach in so viele Teile aufgeteilt wird, wie Rechner vorhanden sind und man diese Teile unabhängig voneinander codieren läßt. Die erzeugten Streams kann man MPEG-konform einfach aneinanderhängen. Es entsteht eine winzige Ineffizienz, da an den Trennstellen keine B-Frames erzeugt werden können. Das ist aber bei längeren Streams zu vernachlässigen.

In dieser Arbeit soll auf diesen Ansatz nicht weiter eingegangen werden, da solch eine Verteilung manuell oder skriptgesteuert sehr gut machbar ist.

8.2 Multithreading

8.2.1 Worker Threads

Um die vom Encoder zu leistende Arbeit auf mehrere Prozessoren aufteilen zu können, muß sie in kleine Einheiten unterteilt werden, die möglichst wenig voneinander abhängig sind. Die obere Grenze für eine sinnvolle Unterteilung stellt wahrscheinlich eine GOP dar. Aber dieser Ansatz hat einen recht hohen Speicherbedarf, da für jeden Thread alle Bilder einer GOP im Speicher gehalten werden müssen. Dies ist nötig, da die Eingabe i.A. nur sequentiell effizient gelesen werden kann oder sogar von einer Echtzeitquelle stammt, die keinen wahlfreien Zugriff erlaubt. Eine andere offensichtliche Unterteilung ist, Slices getrennt zu codieren. In [BFS] werden diese verschiedenen Einteilungen, dort allerdings für den Fall eines MPEG-2 Decoders, untersucht.

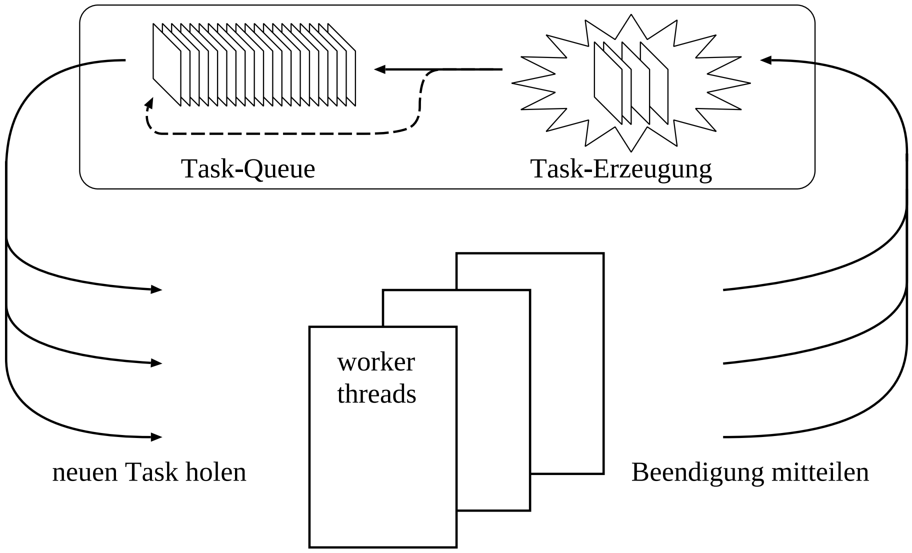


Abbildung 8.1: Aufgabenverteilung auf mehrere worker threads

Das Problem mit einem Ansatz, der sich direkt an der MPEG-Stream-Syntax ausrichtet, ist, daß man an verschiedenen Punkten Informationen aus mehreren Tasks zusammenführen muß, bevor weitercodiert werden kann. So müssen z.B., bevor die Adaptive Quantisierung ausgeführt werden kann, sämtliche Motion-Vektoren und die zugehörigen Differenzbilder berechnet worden sein, damit die Anzahl der Bits, die bei einem gegebenen Quantization-Scale erzeugt wird, abgeschätzt werden kann. Zwangsläufig wird ein Thread so lange blockieren müssen, bis z.B. alle Slices eines Bildes berechnet worden sind, es sei denn, man kapselt die abhängige Aufgabe in einen eigenen Task, der erst nach der Beendigung aller Slice-Tasks verteilt wird.

8.2.2 Abhängigkeitsgraphen

Dies führt uns zu dem Ansatz, der bei dieser Implementation gewählt wurde. Dabei wurde der Codierungsprozeß an sich in mehrere Teile zerteilt. Man kann sich das so vorstellen, daß jeder größere Funktionsaufruf in eine eigene Tasks eingebettet wird. Die Tasks entsprechen damit etwa Befehlen einer Meta-Programmiersprache, die der Encoder abarbeitet. Der große Unterschied ist hier aber, daß die Tasks nicht linear angeordnet sind, sondern daß die Ausführungsreihenfolge durch einen Graphen festgelegt wird, dessen Knoten die Tasks sind und dessen (gerichtete) Kanten anzeigen, welche Tasks vor der Ausführung dieses Tasks erledigt sein müssen. Alle Tasks die gemäß dieser Teilordnung parallel ausgeführt werden dürfen, können im Encoder auch tatsächlich auf mehrere Threads verteilt werden. Abbildung 8.2 zeigt einen fiktiven Taskgraphen. Tasks, die parallel zueinander ausgeführt werden können, sind hier übereinander angeordnet.

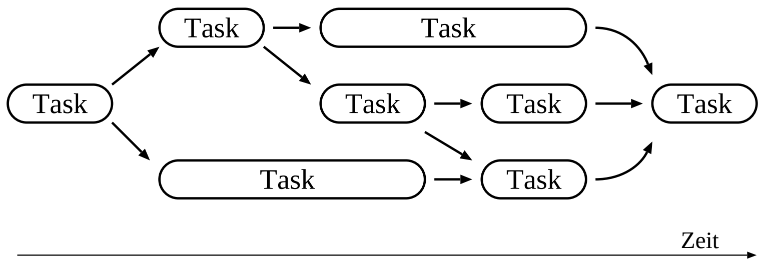


Abbildung 8.2: Beispiel für einen Task-Graphen

Alle Tasks und deren Abhängigkeiten sind in einem `Task-Pool`-Objekt gespeichert. Hat ein Thread die Arbeit an einem Task beendet, so teilt dieser das dem `Task-Pool` mit, der daraufhin den Task und alle Abhängigkeitskanten, die von diesem Task ausgehen, entfernt.

Der Vorgang in unserem Encoder ist noch etwas komplexer, denn selbstverständlich muß für jeden Frametyp ein anderer Graph aufgebaut werden, da ja ganz verschiedene Aktionen auszuführen sind (Motionvektor-Suche entfällt z.B. bei I-Frames; B-Frames brauchen nicht decodiert zu werden). Leider steht aufgrund der durch die Szenenwechselerkennung bedingten flexiblen GOP-Struktur nicht von vornherein fest, welcher Frame welchen Typ zugewiesen bekommt.

Das zweite Problem ist, daß der Graph für die gesamte Videosequenz sehr groß würde, ja bei einem Realtime-Stream sogar als unendlich betrachtet werden muß. Da der Graph aber zusammenhängend ist, muß der Graph während der Codierung „on-the-fly“ erzeugt werden. Für die Erzeugung und Verteilung der Tasks ist auch der `Task-Pool` zuständig. Aus ihm können Threads Tasks abholen, die sie zum jetzigen Zeitpunkt ausführen dürfen. Haben sie einen Task erledigt, so melden sie dies dem Task-Pool, der daraufhin andere Tasks (die von dem eben bearbeiteten abhängig sind) ggf. freischalten und in Zukunft verteilen kann. Zu geeignetem Zeitpunkt wird der Graph weiter vergrößert (siehe dazu auch Abbildung 8.1).

8.2.3 Der Abhängigkeitsgraph für die Codierung

Der Abhängigkeitsgraph wird in zwei Teile zerlegt. Ein Teil des Graphen ist für das Laden der Frames, die Einteilung der Frames in GOPs und die Verteilung von Frametypen zuständig. Dieser Teil ist in Abbildung 8.3 dargestellt.

Die Eingabe wird immer sequentiell gelesen und kann deshalb nicht parallelisiert werden. Es ist deshalb wichtig, daß so viel Rechenzeit wie möglich diesen `LoadFrame`-Tasks zugeordnet wird. Das wird erreicht, indem diese Tasks vorne in die Task-Queue einsortiert werden, was ihnen die maximale Ausführungspriorität verleiht. In Wahrheit muß noch beachtet werden, daß nur eine bestimmte Anzahl Bilder vorausgelesen wird, denn sonst würden wegen der hohen Ladepriorität erst *alle* Frames des Streams geladen und danach codiert. Immer, wenn genug Frames gelesen wurden, wird ein `Classify-Task` erzeugt, der die GOP-Einteilung vornimmt und die Frametypen setzt. Dies kann je nach Wunsch ein Task sein, der eine feste GOP-Struktur erzeugt oder auch ein Task, der aus den bereits berechneten statistischen Daten eine Szenenwechselerkennung durchführt.

Immer, wenn dieser `Classify-Task` beendet wurde, wird der restliche Graph für die klassifizierten Frames aufgebaut. Allerdings muß beachtet werden, daß der `Classify-Task` ggf. nicht

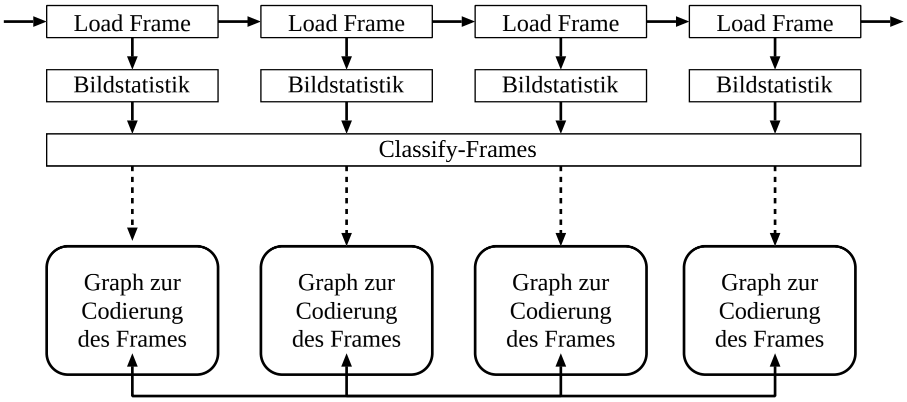


Abbildung 8.3: Task-Graph für den ersten Teil der Codierung (Frame-Klassifizierung)

allen Frames einen Typ gibt. Diese werden dann erst in einem späteren `Classify-Task` vergeben.

Die Abbildungen 8.4 und 8.5 zeigen diesen zweiten Graphen, beispielhaft für die Codierung einer Bildsequenz mit den Frametypen I,B,P. Man beachte, daß der Graph in Abbildung 8.4 in Darstellungsreihenfolge und in Abbildung 8.5 in Übertragungsreihenfolge der Frames dargestellt ist.

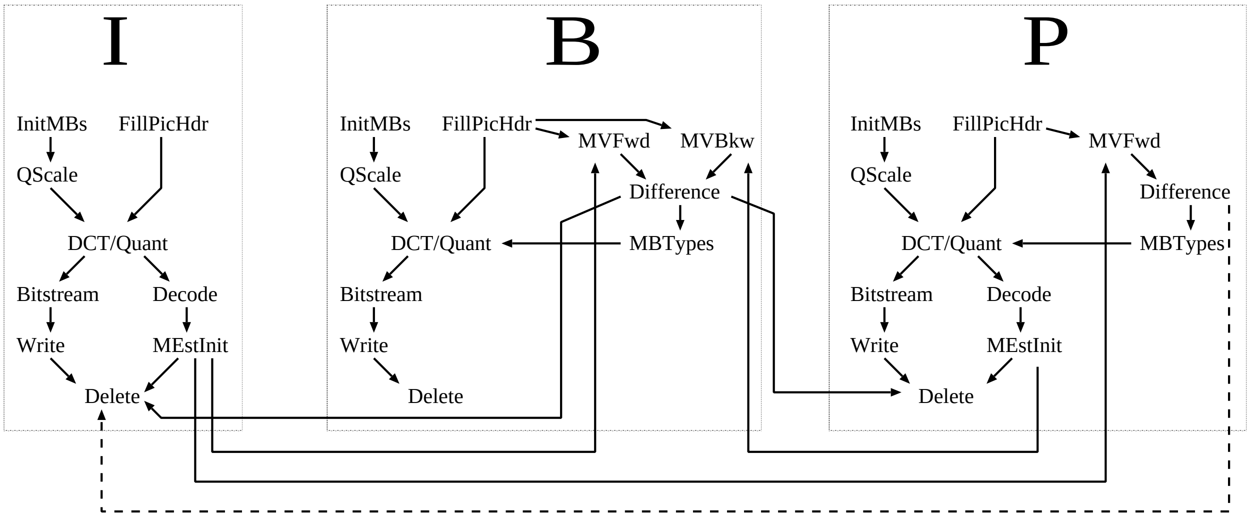


Abbildung 8.4: Task-Graph für den zweiten Teil der Codierung (Teil 1)

Die dargestellten Tasks haben die folgenden Aufgaben:

InitMBs Initialisierung der Makroblöcke, Unterteilen des Bildes in Slices.

FillPicHdr Ausfüllen des Picture-Headers.

QScale Zuweisen von geeigneten Quantization-Scales an die Makroblöcke. Hier wird ggf. die Rate-Control und die Adaptive Quantisierung durchgeführt.

DCT/Quant DCT-Transformation und Quantisierung der Koeffizienten.

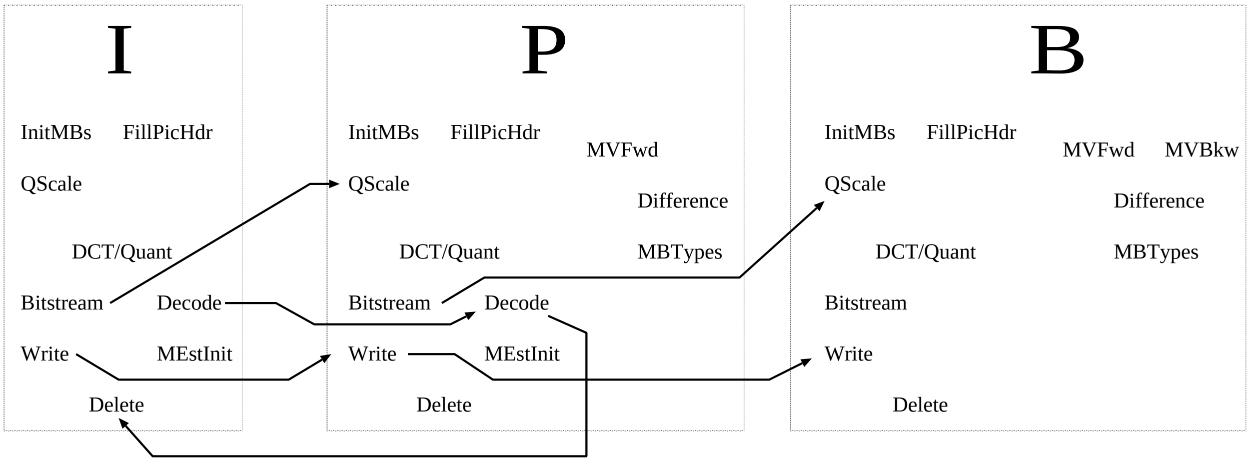


Abbildung 8.5: Task-Graph für den zweiten Teil der Codierung (Teil 2)

Bitstream Erzeugen des Huffman-codierten Bitstreams anhand der zwischengespeicherten DCT-Koeffizienten und sonstiger gesammelter Informationen. Die Daten werden aber nicht direkt auf die Ausgabe geschrieben, sondern in einem Buffer gesammelt, was es erlaubt, auch diese Aufgabe parallel zu bearbeiten.

Write Schreiben des Bitstreams für ein Bild (incl. eventueller vorausgehender Sequence- und GOP-Header) auf die Ausgabe.

Delete Löschen aller Informationen für ein Bild und Freigeben des dafür belegten Speichers.

Decode Decodieren des soeben codierten Bildes.

MEstInit Erzeugen von einer anderen Repräsentation des decodierten Bildes für die folgende Motionvektorsuche. Darunter fällt z.B. das Vorausberechnen von Bitmaps mit Half-Pel Verschiebungen.

MVFwd, MVBkw Motionvektorsuche. Für forward- und backward-Vektoren getrennt, damit dies parallel ablaufen kann.

Difference Differenzbild aus Eingabebild und der Prädiktion durch die Motionvektoren berechnen.

MBTypes Makroblock-weise Entscheidung, ob ein Makroblock Intra oder mit Forward, Backward oder Bidirektionaler Prädiktion codiert werden soll.

Beispielhaft sollen einige Abhängigkeiten erläutert werden: sehr offensichtlich sind die Abhängigkeiten innerhalb eines Frames. Die Abhängigkeiten von den `MEstInit`-Tasks zu den `MVFwd`-, `MVBkw`-Tasks sind notwendig, da die Motionvektorsuche die im `MEstInit`-Task vorberechnete Bitmaps benutzt, in der die Half-Pel-Interpolation schon durchgeführt wurde.

Man beachte auch, daß es relativ häufig Abhängigkeiten für den `Delete`-Task und Tasks anderer Frames gibt. Damit wird verhindert, daß Daten aus dem Speicher gelöscht werden, die später noch für die Codierung benötigt werden.

Die Verkettung der `Write`-Tasks in Bitstream-Reihenfolge (Abbildung 8.5) erzwingt, daß die codierten Frames in der richtigen Reihenfolge in die Ausgabe geschrieben werden.

Die Abhängigkeit vom `Bitstream`- zum `QScale`-Task (auch Abbildung 8.5) ist nur notwendig, wenn ein `QScale`-Task benutzt wird, der eine Bitratensteuerung benutzt, die den VBV-Bufferfüllstand nach der Codierung des letzten Frames benötigt. Ohne eine Bitratensteuerung oder einer Steuerung, welche die Bitrate ohne Bufferkontrolle konstant hält, kann diese Kante entfallen.

8.2.4 On-the-fly Generierung des Graphen

Dieser zweite Teil des Graphen wird immer stückweise aufgebaut. Und zwar wird immer das nächste Paar aus I- oder P-Frames (in Display-Ordnung mit beliebig vielen B-Frames dazwischen) gesucht und dann der Graph für diesen Abschnitt aufgebaut. Alle Tasks dieses Graphen (ohne die Tasks, die sich auf den hinteren I- oder P-Frame beziehen) werden in die Taskqueue übernommen. Danach wird der betrachtete Ausschnitt so weitergeschoben, daß er nun bei dem vormals hinteren I/P-Frame beginnt und bis zum nächsten I/P-Frame reicht. Die Einzelheiten der Grapherzeugung sind ziemlich komplex und sollen hier nicht beschrieben werden. Einige Eigenschaften der Abhängigkeiten sind bei der Implementierung von Vorteil:

- Es gibt (mit einer Ausnahme[1]) keine Abhängigkeiten, die sich nicht durch eine Verkettung von Abhängigkeiten darstellen läßt, die entweder in Display- oder Bitstream-Ordnung keine I- oder P-Frames überspringen.
- Jede Frameübergreifende Abhängigkeit impliziert auch eine Abhängigkeit in umgekehrter Richtung, wobei das Ziel der `Delete`-Task ist. Ausgenommen sind davon natürlich die `Delete`-Task-Abhängigkeiten selbst.

8.2.5 Vereinfachung

Da die Vielzahl der Tasks die Programmierung teilweise erheblich erschwert, wurde noch ein Mittelweg implementiert, bei dem viele Tasks zu einem einzigen `CodeFrame`-Task zusammengefaßt wurde (siehe gestrichelte Umrandung in Abbildung 8.6). Hier lassen sich neue Ideen relativ schnell umsetzen und testen. Die Parallelisierung geschieht hier aber fast ausschließlich auf Frameebene. Der neue (wesentlich kompaktere Graph) ist in Abbildung 8.7 dargestellt.

8.3 Benutzung von SIMD-Befehlen

8.3.1 MMX

Befehlssatz

MMX (MultiMedia eXtension) ist eine 1996 von Intel für ihre Pentium-Prozessoren eingeführte Befehlssatzerweiterung, die 57 neue Befehle zur Verfügung stellt, welche die 64bit breiten Fließkommaregister als SIMD-Register für 8× 8bit, 4× 16bit, 2× 32bit oder 1× 64bit Integerzahlenberechnungen mißbrauchen. Dies verhinderte, daß für sämtliche Betriebssysteme der Code für

[1] Die Ausnahme tritt nur bei der Codierung von Streams mit IPPPP-GOP-Muster auf. Das Problem ist, daß zu dem Zeitpunkt, da der Graph für die ersten beiden Frames I und P aufgebaut wird, noch nicht bekannt ist, daß der P-Frame decodiert werden muß. Im nächsten Schritt wird der Decodiertask zwar eingefügt, aber die Abhängigkeit zu dem `Delete`-Task des I-Frames kann nicht mehr erzeugt werden, da dieser in der Zwischenzeit vielleicht schon ausgeführt wurde. Dieses Problem kann aber sehr einfach umgangen werden, indem *jeder* P-Frame decodiert wird, was eine winzige Ineffizient bei der Codierung von IPPP-Sequenzen ergibt, da der letzte Frame vor dem I-Frame eigentlich nicht decodiert werden muß.

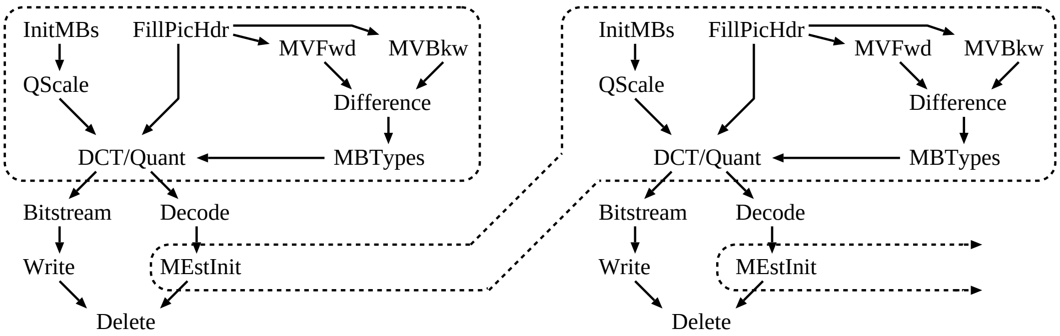


Abbildung 8.6: Zusammensetzung des `CodeFrame`-Tasks

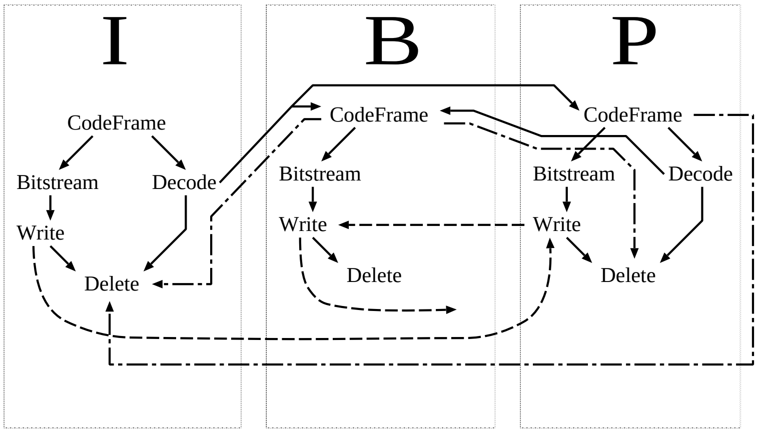


Abbildung 8.7: Verkürzter Abhängigkeitsgraph

die Registerrettung bei einem Taskwechsel umgeschrieben werden mußte. Die wesentlichen neuen Befehle lassen sich in diese Gruppen unterteilen:

- Addition/Subtraktion
- Pack/Unpack (z.B. Verteilen von 8× 8bit in einem Register auf 4× 16bit in zwei Registern und umgekehrt.)
- Vergleiche
- Logische Verknüpfungen, Shift-Operationen
- Multiplikation (Diese muß gegebenenfalls in zwei Befehlen ausgeführt werden, wobei beim ersten nur die höherwertigen Bits gespeichert werden und beim zweiten die niederwertigen. Dafür existieren zwei getrennte Befehle.)
- Eine Multiply-Add-Operation. Diese multipliziert die vorzeichenbehafteten 4× 16bit Zahlen aus 2 Registern miteinander, addiert das erste und zweite, sowie das dritte und vierte Produkt und speichert diese beiden Summen als 2× 32bit Integers in einem Register ab. Andere Datenformate sind hier leider nicht möglich.

Die Additions-/Subtraktions- und Pack-Operationen können auch *Saturated* ausgeführt werden; d.h. überschreitet ein Wert den möglichen Bereich im Zielregister, so wird er durch den maximal bzw. minimal möglichen Wert ersetzt.

Verwendung bei der Motionvektor-Suche

Der bei der Motionvektor-Suche rechenaufwendigste Teil ist die Berechnung der MAD-Differenz zweier Blöcke. Sind die beiden Blöcke durch $f_1(x,y)$ und $f_2(x,y)$ gegeben ($0 \leq x,y \leq 15$), so berechnet sich die MAD durch:

$$MAD \cdot 256 = \sum_{x=0}^{15} \sum_{y=0}^{15} |f_1(x,y) - f_2(x,y)|$$

Besonders, wenn der Befehlssatz eines Prozessors (wie auch der Pentium von Intel) keinen Befehl für den Absolutwert hat, benötigt die Berechnung relativ viel Zeit, da dann pro Differenz eine Verzweigung nötig wird, die sich auch durch eine eventuell vorhandene Branch-Prediction im Prozessor nicht vorhersagen läßt.

Bei der MMX-Implementation können immer 8 Pixel parallel bearbeitet werden. Eine Blockzeile muß also in zwei Schritten berechnet werden. Bezeichnen wir die Daten von $f_1(0,y)$ bis $f_2(7,y)$ (bzw. $f_1(8,y)$ bis $f_1(15,y)$ mit A und die Daten von f_2 entsprechend mit B, so wird die in Abbildung 8.8 dargestellte Berechnung ausgeführt. In einer saturated Subtraktion werden zuerst

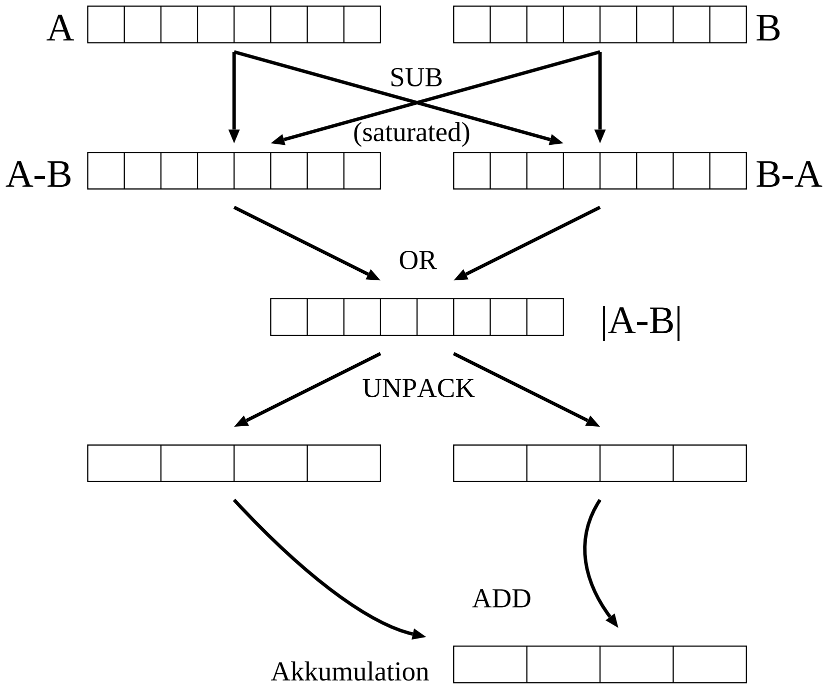


Abbildung 8.8: MAD Berechnung mit MMX-SIMD-Befehlen

die Differenzen $A - B$ und $B - A$ gebildet. Für den Fall, daß $A < B$ ist, so ist die entsprechende Position in $A - B$ gleich 0. Ist umgekehrt $A > B$, so ist die Position in $B - A$ gleich 0. Nun werden diese beiden Ergebnisse ODER-verknüpft. Da bei jeder Position mindestens in einem der Ergebnisse von $A - B$ und $B - A$ eine 0 stehen muß, erhalten wir damit die 8 absoluten Differenzen.

Diese addieren wir aber nicht sofort auf, sondern benutzen ein Register in 4× 16bit Anordnung für die Aufnahme der MAD-Zwischensumme. Die Summe der 4× 16bit-Zahlen in diesem Register stellt die MAD-Zwischensumme dar (16bit reichen für die Aufnahme der MAD aus, da diese maximal $256 \cdot 255$ betragen kann). Unsere 8 absoluten Differenzen teilen wir mit zwei `unpack`-Operationen in zweimal 4× 16bit auf und addieren diese zu diesem Akkumulationsregister. Erst wenn wir den ganzen 16×16 Block berechnet haben, addieren wir diese vier 16bit Werte auf, was durch die dafür erforderlichen Shift- und Maskierungs-(AND)-Befehle ziemlich aufwendig ist. Der Vorteil dieser Implementierung ist, daß sie völlig ohne Sprünge auskommt. Ein Nachteil ist, daß dadurch, daß die Fehlersumme erst am Ende aller Berechnungen eines Blockes berechnet wird, es nicht möglich ist, die Berechnung eines Blockes abzubrechen, sobald der Fehler einen maximal zulässigen Fehler überschreitet.

Weitere Anwendungsmöglichkeiten

Die MMX-Befehle sind auch für andere Zwecke im Encoder anwendbar. Ein einfaches Beispiel wäre die Berechnung der Differenzbilder, aber auch die FDCT und insbesondere die IDCT (durch ihre geringere Genauigkeitsanforderung) ließe sich durch Ausnutzen des MMX-Befehlssatzes beschleunigen.

3Dnow! und MMX2

Zusätzlich zu den von Intel eingeführten MMX-Befehlen hat AMD einige weitere Befehle eingeführt, die unter dem Namen 3Dnow! zusammengefaßt werden. Es handelt sich dabei fast ausschließlich um SIMD Befehle für Fließkommadaten, aber auch um zwei Befehle, die für die MPEG Komprimierung von Vorteil sind. `PAVGUSB` berechnet für zweimal 8× 8 bit Integerdaten einen gerundeten Mittelwert (also $(A+B+1)/2$). Diese Berechnung wird zur Berechnung der Half-Pel-Prädiktion und bei der bidirektionalen Prädiktion benötigt. Der zweite hier interessante Befehl ist `PREFETCH`, mit dem sich den Datencache mit Daten füllen läßt, die bald benötigt werden. Damit wird verhindert, daß der Prozessor auf das Eintreffen der Daten aus dem Speicher warten muß. Da bei der MPEG-Codierung sehr große Datenmengen in einer vorhersehbaren Reihenfolge bearbeitet werden müssen, dürfte sich damit ein Performancegewinn erzielen lassen.

Für 1999 hat Intel den MMX2-Befehlssatz angekündigt. Neu ist hier, daß nun 8 völlig neue Register mit jeweils 128 bit Breite mit 70 neuen SIMD-Befehlen zur Verfügung stehen. Für die MPEG-Codierung interessant sind hier vor allem das Äquivalent zu obigem `PAVGUSB` und ein Befehl, der die Summe der absoluten Differenzen von 8 Bytes bildet, was die oben ausführlich beschriebene Befehlsfolge durch einen einzigen Befehl ersetzt.

8.3.2 VIS

Auch SUN hat in ihren UltraSPACE-Prozessoren eine Befehlserweiterung mit SIMD-Befehlen unter dem Namen VIS (Visual Instruction Set) aufgenommen, der ähnliche Befehle zur Verfügung stellt, die der MMX-Befehlssatz. Ein Vorteil ist der hier vorhandene Befehl `vis_pdist` für die Summe der absoluten Differenzen.

Die VIS-Unterstützung für den Blockvergleich bei der Motionvektorsuche hat freundlicherweise Niels Mache zu der SAMPEG-Implementierung ergänzt.

8.4 Ergebnisse

8.4.1 Definition des Speedups

Der Speedup eines parallelen Programms ist definiert durch:

$$S_p(n) = \frac{T^*(n)}{T_p(n,m)}$$

wobei $T^*(n)$ die Ausführungszeit des schnellsten sequentiellen Programms bei Eingabegröße n und $T_p(n)$ die Ausführungszeit des parallelen Programms auf m Prozessoren ist. Da in unserem Fall der sequentielle Encoder nur ein Sonderfall des parallelen Encoders ist, ist $T^*(n) = T_p(n,1)$ und der Speedup ergibt sich gleich zu

$$S_p(n) = \frac{T_p(n,1)}{T_p(n,m)}$$

Die Eingabegröße n setzt sich hierbei aus vielen getrennten Faktoren zusammen, zu denen nicht nur die Länge der Sequenz zählt, sondern auch deren Auflösung und Codierungsparameter wie die Art der Motionvektor-Suche.

8.4.2 Meßergebnisse

SAMPEG konnte bisher nur auf einem Rechner mit maximal 2 Prozessoren getestet werden, der auch nicht ausschließlich für diesen Zweck zur Verfügung stand. Deshalb schwankten die erhaltenen Meßdaten auch bei identischen Eingabedaten in einem sehr großen Bereich. Mit 2 Threads auf 2 Prozessoren erhielt man einen Speedup von etwa 160-170%. Erhöht man die Anzahl der Threads auf 4 (aber immer noch auf 2 Prozessoren), so stieg der Speedup auf bis zu 190%. Das zeigt, daß bei einer exklusiven Nutzung des Rechners für den Encoder sehr hohe Speedupwerte zu erwarten sind.

Kapitel 9

Softwarebeschreibung

9.1 SAMPEG

9.1.1 Features

- Erzeugt MPEG-1 und MPEG-2 Streams mit beliebiger Auflösung und Bitrate.
- Liest PPM und Targa Bildsequenzen, sowie MJPEG-AVIs, wie sie von den meistbenutzten MJPEG-Karten für PCs (z.B. Miro DC10-DC30, FAST AV-Master) erzeugt werden.
- Ermöglicht die Erzeugung von MPEG-System-Streams mit Audio unter Zuhilfenahme eines getrennten MPEG-Audio-Encoders.
- Erzeugt VBR- und CBR-Streams.
- Unterstützt Multithreading auf Mehrprozessormaschinen mit beliebig vielen Prozessoren.
- SIMD Unterstützung durch Benutzen des MMX- und VIS-Befehlssatzes, wenn vorhanden.
- Automatische Szenenwechselerkennung. Alternativ ist beliebige GOP Struktur einstellbar (Erzeugung von I-only Streams oder I-P-only Streams).
- Adaptive Quantisierung durch Contrast Masking und Luminance Masking.
- Mehrere Motionvektorsuchalgorithmen mit beliebigem Suchbereich.
- Schnelle Algorithmen zur DCT-Transformation.

9.1.2 Bedienung

Eingabeformate

SAMPEG akzeptiert als Eingabe Einzelbilder, die im PPM oder Targa Format gespeichert sind (nur 24 Bit TrueColor) und AVI-Streams, die MJPEG-komprimierte Videosequenzen enthalten. Solche AVI-Streams werden von den bekannte MJPEG-Karten für PCs erzeugt und können direkt ohne Änderung verarbeitet werden. Da die MJPEG-Streams schon im YCbCr-Farbraum vorliegen, kann in SAMPEG auf die Konvertierung von RGB nach YCbCr verzichtet werden. Befindet sich in dem AVI auch eine Audio-Spur, so kann diese während der Codierung in eine eigene Datei extrahiert und von einem externen MPEG-Audio Encoder codiert werden.

Sollen PPM- oder Targa-Bildsequenzen benutzt werden, so ist als Dateiname ein `printf`-Ausdruck anzugeben, dem eine fortlaufende Bildnummer übergeben wird. Lauten die Dateinamen der Bilder z.B. `bild000.ppm`, `bild001.ppm`, ... , so muß als Dateiname „`bild%03d.ppm`“ angegeben werden. Bei Eingabe im PPM- oder Targa-Format muß dieses noch durch die Option `--ppm` bzw. `--targa` aktiviert werden.

Soll nicht die ganze Videosequenz codiert werden, so kann das Startbild mit `-s <nr>` und das Endbild mit `-e <nr>` angegeben werden. Zusätzlich existiert noch die Möglichkeit, mit der Option `--smallavi` MJPEG-AVI-Sequenzen auf ein Viertel des Auflösung zu reduzieren.

Optionen

Eine Übersicht der vorhandenen Optionen ist in Tabelle 9.1 aufgelistet. Diese Übersicht erhält man, wenn man SAMPEG mit der Option `-h` aufruft. Die meisten Optionen sind selbsterklärend; hier deshalb nur die wichtigsten:

`-1,-2` dienen zur Auswahl, ob ein MPEG-1 oder MPEG-2 Stream erzeugt werden soll. Defaultmäßig wird MPEG-2 erzeugt, das aber viele Decoder derzeit noch gar nicht anzeigen können. Einige der spezielleren Codierungsoptionen haben nur unter MPEG-2 eine Wirkung.

`-R` **num** Aktiviert die Rate Control-Einheit zur Codierung von CBR-Streams. Die Bitrate wird in Mb/s angegeben. `-R 1.5` erzeugt somit 1.5 Mb/s-Streams.

`-J -B -A -M` Diese aktivieren die verschiedenen Algorithmen zur Adaptiven Quantisierung. Dabei ist `-M` derzeit noch in Entwicklung und nicht funktionsfähig. Empfohlen werden die Optionen `-A` und `-B`. Diese Optionen sollten generell nur in Verbindung mit einer Codierung mit fester Bitrate benutzt werden.

`-q --qscaleI --qscaleP --qscaleB` Hiermit kann man bei deaktivierter Rate-Control für die Bildtypen feste *qscale*-Werte angeben. `-q` setzt dabei den gleichen *qscale*-Wert für alle Bildtypen.

`-P -g -p` Deaktiviert die Szenenerkennung. Es wird nun mit einer festen GOP-Struktur komprimiert. Dabei läßt sich die GOP-Länge durch `-g` einstellen und den Abstand der P-Bilder, bzw. den Abstand des ersten P-Bilds vom I-Bild durch `-p`.

`-x -d -a -f` Dies sind Codieroptionen, die nur unter MPEG-2 eine Auswirkung haben. `-a` aktiviert die Alternate-Scan-Ordnung anstatt der Zig-Zag-Anordnung, was bei der Codierung von Interlaced-Video Vorteile bringt. `-d` setzt die Quantisierschrittweite des DC-Koeffizienten. Ein höherer Wert bedeutet dabei eine *kleinere* Quantisiererschrittweite. Bei höheren Bitraten sollte dieser Wert auf 10 gesetzt werden, da damit das Hintergrundflimmern deutlich nachläßt. Ein Wert von 11 ist nur im High-Profile erlaubt. `-f` aktiviert verbesserte Intra-Bild Huffmantabellen. Es sollte bei MPEG-2 immer aktiviert sein. `-x` aktiviert die exponentielle *qcode* $\leftrightarrow$ *qscale*-Beziehung, was insbesondere bei hohen Bitraten von Vorteil ist. Es kann mit ruhigem Gewissen bei MPEG-2 immer aktiviert werden.

`-h -V` Wählt den horizontalen und vertikalen Suchbereich der Motionvektorsuche.

```
 SAMPEG-2   v0.5.0  (27.Okt.98)                     Dirk Farin
-------------------------------------------------------------------
 Syntax: enc [options] <input-filename-pattern>  <output-filename>

Optionen:
  -1, --mpeg1                 MPEG-1 Streams erzeugen
  -2, --mpeg2                 MPEG-2 Streams erzeugen (default)
  -s, --firstframe=NUM        Nummer des ersten einzulesenden Frames (Default: 1)
  -e, --lastframe=NUM         Nummer des letzten einzulesenden Frames
  -t, --targa                 Eingabe ist im Targa-Format
      --ppm                   Eingabe ist im PPM-Format
      --ppminterlaced         Eingabe ist im PPM-Format (beide Halbbilder getrennt)
  -m, --avi                   Eingabe ist ein MJPEG-AVI (Default)
  -S, --smallavi              Interlaced MJPEG-AVI auf ein viertel der Aufloesung reduzieren
  -W, --saveaudio=TXT         Audio aus AVIs als PCM-Datei abspeichern
  -T, --threads=NUM           Anzahl der Threads angeben (Default=1)
  -D, --verbosity=NUM         Menge der Ausgabemeldungen anzeigen (0-3). Default=2
  -h, --help                  Hilfe anzeigen

  -R, --bitrate=NUM           Setzt Bitrate (in Mb/s) und aktiviert CBR-Codierung.
  -J, --jnd                   Aktiviert JND-Map-Berechnung
  -B, --LumMasking            Aktiviert luminance-masking
  -A, --ActMasking            Aktiviert activity-masking
  -M, --MotionMasking         Aktiviert motion-masking
  -q, --qscale=NUM            Setzt quantisation-scale auf NUM (1-31) ; Default=8
      --qscaleI=NUM           Setzt I-Frame quantisation-scale
      --qscaleP=NUM           Setzt P-Frame quantisation-scale
      --qscaleB=NUM           Setzt B-Frame quantisation-scale
  -x, --dcttype1              Waehlt exponentielles qcode -> qscale   ; Default=linear (MPEG2)
  -d, --intraDCprec=NUM       Setzt die Intra-DC-Precision auf NUM (8-11) ; Default=8  (MPEG2)
  -a, --alternatescan         Benutze 'Alternate Scan'                               (MPEG2)
  -f, --intraVLCformat1       Benutze intra_vlc_format=1                             (MPEG2)
  -g, --GOPlength=NUM         Laenge einer GOP     ; Default=20  (fuer EncoderControl_Simple)
  -p, --PDistance=NUM         Abstand der P-Bilder ; Default= 5  (fuer EncoderControl_Simple)

  -F, --fullsearch            Motionvektorsuche mit Full-Search-Algorithmus
  -3, --TSS                   Motionvektorsuche mit Three Step Search (Default)
      --NTSS                  Motionvektorsuche mit New Three Step Search
  -O, --OTS                   Motionvektorsuche mit One at a time Search
      --1DFS                  Motionvektorsuche mit One dimensional full search
  -H, --hsearchrng=NUM        Horizontaler Suchbereich (in Pixel, moeglichst 2^n)
  -V, --vsearchrng=NUM        Vertikaler   Suchbereich (in Pixel, moeglichst 2^n)
      --accurateMBtype        MB-Typ durch Minimierung der tatsaechlichen Bitzahl bestimmen
  -P, --framepattern          Frametyp durch Muster festlegen (keine Szenenwechsel-Erkennung)

  -v, --visualizecoding=TXT   Bilder zur Visualisierung des Codierungsvorgangs ausgeben
      --showinfo=TXT          Spezielle Informationsausgaben anschalten
  -r, --showRMS               RMS-Fehler der Ausgabebilder ausgeben
      --nopredcorrect         Praediktion-Korrektur abschalten (nur fuer Debugging)
      --disablemmx            MMX-Unterstuetzung abschalten (nur fuer Debugging)
      --disablevis            VIS-Unterstuetzung abschalten (nur fuer Debugging)
```

Tabelle 9.1: Progammoptionen von SAMPEG.

9.1.3 Installation

SAMPEG läuft auf Unix-Systemen und benutzt für das Multithreading die Posix-Thread-Bibliothek. Soll der MJPEG-AVI-Loader benutzt werden, muß auch die `libjpeg` der *Independent JPEG Group* installiert sein.

Dem Programm liegt ein `configure`-Skript bei, das das verwendete System automatisch erkennt und schaut, ob die `libjpeg` im System installiert ist. Daraufhin werden die Optionen für die Compilierung automatisch gesetzt, so daß mit einem einfach `make`-Aufruf der Encoder erzeugt werden kann. Soll ein anderer Compiler als der Standardcompiler des Systems benutzt werden, so ist die Umgebungsvariable `CC` vor dem Aufruf des `configure`-Skripts auf den Namen des Compilers zu setzen.

9.1.4 Erzeugen von MPEG-System-Streams mit Audio

Enthält ein MJPEG-AVI-Stream auch einen Audio-Stream, so kann dieser mit der Option `-W` im PCM-Format in eine eigene Datei geschrieben werden. Diese kann dann mit einem MPEG-Audio-Encoder (z.B. der Referenzimplementation) codiert werden. Der Audiostream kann danach mit dem Videostream unter Zuhilfenahme eines MPEG-System-Multiplexers (z.B. `mplex` von Christoph Moar) in einen MPEG-System-Stream kombiniert werden, der nun auf jedem Audio-fähigen MPEG-Decoder abgespielt werden kann.

9.1.5 Performance

Motionvektorsuche

Ein Vergleich der implementierten Suchverfahren ist in den Tabellen 9.2 und 9.3 aufgeführt. Codiert wurde eine Tagesschauszene mit sehr wenig Bewegung im Bild und eine Motorsportszene mit sehr viel Bewegung im Bild codiert. Zur Messung wurde jeweils mit einer festen GOP-Struktur und festen *qscale*-Werten codiert, so daß anhand der entstehenden Dateigröße direkt die Qualität der Motionvektorsuche ersichtlich ist. Der optimale Motionvektor bei der Tagesschauszenen ist fast ausschließlich der Nullvektor, da der größte Teil des Bildes immer gleich bleibt. In der Motorsportszene ist jedoch ein Motorradrennen zu sehen, das sehr schnelle Kameraschwenks enthält. Ein Suchbereich von 32 Pixel ist hier notwendig, um die optimalen Vektoren finden zu können. In den Tabellen ist zu jedem Suchverfahren und Suchbereich jeweils die entstandene Filegröße, sowie die benötigte Rechenzeit notiert.

Bei der Analyse der Werte der Tagesschauszene sieht man, daß die 2DFS-Suche natürlich die optimalen Vektoren findet und damit den Maßstab für die anderen Verfahren bildet. Alle Verfahren liegen hier sehr eng zusammen. Nur die OTS-Suche liefert konstant schlechte Werte bei allen Suchbereichen. Da dabei auch der Rechenzeit konstant bleibt, ist anzunehmen, daß sie sich jeweils schnell in ein Minimum nahe dem Nullvektor verläuft. Man sieht auch den deutlichen Performanceabfall der TSS ab einem Suchbereich von 64 Pixel. Die NTSS scheint dagegen stabiler zu sein, liefert aber bei kleinen Suchbereichen schlechtere Ergebnisse.

Anders sehen die Ergebnisse bei der Motorradszene aus. Weit abgeschlagen ist auch hier die OTS-Suche. Erstaunlich schlechte Ergebnisse im Vergleich zur TSS liefert auch die NTSS. Das ist daraufhin zurückzuführen, daß das Suchmuster bei der NTSS für fast unbewegte Szenen optimiert wurde, was hier einfach nicht zutrifft. Wieder gute Ergebnisse liefert die TSS, etwas bessere bei allerdings höherer Rechenzeit die 1DFS und ab einem Suchbereich von 32 Pixel mit großem Abstand die besten Ergebnisse ergibt die 2DFS, deren Rechenzeit aber eigentlich schon außerhalb des praktisch anwendbaren Bereiches liegt.

H,V	**2DFS**		**1DFS**		**TSS**		**NTSS**		**OTS**	
8	221512	32.16s	221581	19.48s	221588	18.00s	222210	17.61s	240560	17.42s
16	221522	73.71s	221595	21.10s	221598	17.93s	222232	17.86s	240560	17.37s
32	221522	219.15s	221595	24.36s	221981	18.19s	222208	17.49s	240560	17.30s
64	221522	647.12s	221595	29.23s	222351	18.55s	222208	18.06s	240493	17.48s

Tabelle 9.2: Tagesschauszene, 40 Bilder, AMD k6 233MHz, MMX

H,V	**2DFS**		**1DFS**		**TSS**		**NTSS**		**OTS**	
8	679571	117.47s	693395	31.40s	692427	23.20s	729734	25.44s	841120	19.93s
16	635342	373.80s	662425	42.01s	673877	23.98s	731467	25.95s	836011	19.81s
32	590028	1225.61s	644145	62.34s	664541	24.50s	731999	26.79s	830020	20.14s
64	578024	3888.54s	645678	96.95s	695327	25.83s	757430	27.93s	822921	20.48s

Tabelle 9.3: Motorsportszene, 40 Bilder, AMD k6 233MHz, MMX

Ausführungsgeschwindigkeit

In den Tabellen 9.4 bis 9.7 wurde zusammengestellt, welche Programmteile welchen Anteil der Rechenzeit verbrauchen. Dazu wurde wieder die Tagesschauszene und die Motorsportszene codiert. Beide Sequenzen waren 50 Bilder lang und hatten eine Auflösung von 384×288. Codiert wurde auf einem AMD-k6 233MHz Rechner mit MMX und zwar jeweils mit TSS- und 2DFS-Motionvektorsuche und einem Suchbereich von horizontal und vertikal je ± 16 Halfpel. Die Gesamtlaufzeiten wurden ohne Profiler bestimmt, da dieser teilweise selbst einen erheblichen Rechenzeitbedarf hat.

Anteil	Laufzeit	Aufrufe	Programmteil
19.64%	3.74s	129600	FDCT
12.61%	2.40s	50	Bild laden, MJPEG dekomprimieren
12.08%	2.30s	50	Differenzbilder erzeugen
9.82%	1.87s	112752	Motion Compensation
5.78%	1.10s	186239	Blockvergleich der Motionvektorsuche
4.36%	0.83s	23829	IDCT
4.31%	0.82s	20600	Quantisierung
	22.31s		**Gesamtlaufzeit**

Tabelle 9.4: Tagesschauszene, 50 Bilder, TSS, AMD k6 233MHz, MMX

Anteil	Laufzeit	Aufrufe	Programmteil
44.83%	19.31s	4137849	Blockvergleich der Motionvektorsuche
8.87%	3.82s	129600	FDCT
8.43%	3.63s		(gprof-Profiler)
5.39%	2.32s	50	Bild laden, MJPEG dekomprimieren
5.13%	2.21s	50	Differenzbilder erzeugen
4.16%	1.79s	112752	Motion Compensation
	40.91s		**Gesamtlaufzeit**

Tabelle 9.5: Tagesschauszene, 50 Bilder, 2DFS, AMD k6 233MHz, MMX

Anteil	Laufzeit	Aufrufe	Programmteil
17.64%	4.59s	998877	Blockvergleich der Motionvektorsuche
13.76%	3.58s	129600	FDCT
10.22%	2.66s	112752	Motion Compensation
9.30%	2.42s	50	Bild laden, MJPEG dekomprimieren
8.46%	2.20s	50	Differenzbilder erzeugen
	28.74s		**Gesamtlaufzeit**

Tabelle 9.6: Motorsport, 50 Bilder, TSS, AMD k6 233MHz, MMX

Anteil	Laufzeit	Aufrufe	Programmteil
67.31%	114.44s	25683941	Blockvergleich der Motionvektorsuche
10.69%	17.17s		(gprof-Profiler)
2.88%	4.90s	37584	Motion-Estimation Hauptroutine
2.16%	3.68s	129600	FDCT
	144.82s		**Gesamtlaufzeit**

Tabelle 9.7: Motorsport, 50 Bilder, 2DFS, AMD k6 233MHz, MMX

9.2 Erweiterungen

9.2.1 Genaue Adaptive Quantisierung

Es existieren Verfahren, die über den Ansatz einer Lagrange-Multiplikator-Methode die optimale Quantisierung bestimmt. Dazu müssen die jeweiligen Rate-Distortion-Kurven der Quantisierer bekannt sein. Wählt man nun als Distortion ein physiologisch begründetes Maß, so sollte sich damit eine sehr gute Adaptive Quantisierung erreichen lassen. Der Rechenaufwand ist zwar immer noch sehr hoch, könnte sich aber für die Offline-Codierung durchaus lohnen.

9.2.2 Koeffizienten Thresholding

Bei niedrigen Bitraten ist es vorteilhaft, DCT-Koeffizienten-Blöcke, die nur sehr kleine Werte enthalten, völlig zu steichen, da dadurch besonders viele Bits gespart werden können. Diese gesparten Bits lassen dann eine insgesamt niedrigere Quantisierungsschrittweite zu. Auch dies läßt sich mit der Lagrange-Multiplikator-Methode optimal bestimmen.

Versuchsweise wurde in SAMPEG eine einfache Variante dieses Verfahrens implementiert, die einen DCT-Koeffizienten-Block entfernt, wenn die absolute Summe aller Koeffizienten einen (sehr kleinen) Wert nicht überschreitet. Das Ergebnis war eine deutliche Verbesserung der subjektiven Bildqualität. Zwar wurde das Bild insgesamt etwas unschärfer, was ein Beobachter allerdings als weicher empfinden mag. Ebene Hintergründe blieben sehr ruhig stehen und zeigten keine zeitliche Unruhe mehr.

9.2.3 Echtzeit MPEG Codierung von MJPEG-Streams

Es dürfte ohne allzu großen Aufwand möglich sein, den Encoder so zu modifizieren, daß er auf Standard-Workstations MJPEG-Streams in MPEG-Streams umcodiert, wenn man die Möglichkeiten von MPEG nicht vollständig ausnutzt. Wir beschränken uns auf die Codierung mit festem *qscale* und die Untersuchung von nur wenigen möglichen Motion-Vektoren.

Für die Codierung von I-Frames muß nur die Bitstream-Syntax veraendert werden. Die Suche der Motion-Vektoren beschränken wir auf die 9 Positionen mit den horizontal und vertikalen Offset von jeweils $\text{offs}_v, \text{offs}_h \in \{-16, 0, 16\}$ (Pixel), bzw. untersuchen nur den Vektor (0;0). Als Maß des Abstands zwischen den Makroblöcken benutzen wir direkt ein Maß auf den DCT-Koeffizienten. Wurde entschieden, daß der Makroblock mit Prädiktion codiert wird, so können direkt die DCT-Koeffizienten der Prädiktion von den Koeffizienten im Makroblock subtrahiert werden.

9.3 TrueColor-Patch für `mpeg2decode`

9.3.1 Beschreibung

Der MPEG-2-Decoder der *MPEG Software Simulation Group* (MSSG) erlaubt nur die Ausgabe von Videos auf 8 bit tiefen Displays und verwendet hierbei stets ein festes 4×4-Ditheringmuster. Dadurch können Feinheiten nicht mehr erkannt werden. Im Rahmen dieser Diplomarbeit wurde deshalb ein Patch entwickelt, der den Decoder befähigt, auch 16 und 24 bit tiefe TrueColor-Displays zu benutzen. Bei der Darstellung mit 16 bit wird dabei das 5:6:5-Verfahren benutzt, d.h. jeweils 5 bits für die Rot- und Blaukomponente und 6 bits für die Grünkomponente, da der Mensch im grünen Spektralbereich besonders empfindlich auf Intensitätsschwankungen ist.

Die Gesamtperformance des Decoders sinkt bei Benutzung der TrueColor-Modes um 25% (16 bit) bis 33% (24 bit), da die Farbtransformation nun mit höherer Genauigkeit berechnet werden muß.

9.3.2 Installation

Zur Installation muß die alte Datei `display.c` durch die neue ausgetauscht werden und das Programm neu compiliert werden. Weitere Änderungen sind nicht nötig.

Anhang A

Nachweis der IEEE 1180 Konformität der Integer IDCT

IEEE test conditions: -L = -256, +H = 255, sign = 1, #iters = 10000

```
Peak absolute values of errors:
   1    1    0    1    0    1    0    0
   0    1    1    0    0    0    0    0
   1    1    1    0    0    0    0    0
   0    0    0    0    1    0    0    1
   1    1    1    1    0    1    0    1
   0    1    1    0    0    0    0    1
   0    0    0    1    0    1    0    0
   0    1    0    0    1    0    1    0
Worst peak error = 1  (meets spec limit 1)

Mean square errors:
   0.0001   0.0003   0.0000   0.0002   0.0000   0.0001   0.0000   0.0000
   0.0000   0.0001   0.0001   0.0000   0.0000   0.0000   0.0000   0.0000
   0.0001   0.0001   0.0001   0.0000   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000   0.0001   0.0000   0.0000   0.0001
   0.0001   0.0001   0.0001   0.0001   0.0000   0.0002   0.0000   0.0001
   0.0000   0.0001   0.0002   0.0000   0.0000   0.0000   0.0000   0.0001
   0.0000   0.0000   0.0000   0.0001   0.0000   0.0001   0.0000   0.0000
   0.0000   0.0001   0.0000   0.0000   0.0001   0.0000   0.0001   0.0000
Worst pmse = 0.000300  (meets spec limit 0.06)
Overall mse = 0.000047  (meets spec limit 0.02)

Mean errors:
  -0.0001   0.0003   0.0000   0.0002   0.0000   0.0001   0.0000   0.0000
   0.0000   0.0001   0.0001   0.0000   0.0000   0.0000   0.0000   0.0000
  -0.0001   0.0001  -0.0001   0.0000   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000   0.0001   0.0000   0.0000  -0.0001
  -0.0001   0.0001  -0.0001   0.0001   0.0000   0.0002   0.0000   0.0001
   0.0000   0.0001   0.0002   0.0000   0.0000   0.0000   0.0000  -0.0001
   0.0000   0.0000   0.0000   0.0001   0.0000  -0.0001   0.0000   0.0000
   0.0000   0.0001   0.0000   0.0000  -0.0001   0.0000  -0.0001   0.0000
Worst mean error = 0.000300  (meets spec limit 0.015)
Overall mean error = 0.000016  (meets spec limit 0.0015)
```

```
0 elements of IDCT(0) were not zero
```

IEEE test conditions: -L = -5, +H = 5, sign = 1, #iters = 10000

```
Peak absolute values of errors:
   1   1   1   1   0   0   0   0
   0   0   1   0   0   0   0   0
   0   1   0   1   1   0   0   0
   1   0   1   1   1   0   0   0
   1   0   1   1   1   1   0   0
   0   1   1   0   1   1   1   0
   1   0   0   0   0   1   1   1
   1   1   1   0   0   1   1   0
Worst peak error = 1  (meets spec limit 1)

Mean square errors:
   0.0003   0.0004   0.0001   0.0003   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0002   0.0000   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0003   0.0000   0.0001   0.0001   0.0000   0.0000   0.0000
   0.0001   0.0000   0.0001   0.0001   0.0002   0.0000   0.0000   0.0000
   0.0003   0.0000   0.0001   0.0001   0.0002   0.0001   0.0000   0.0000
   0.0000   0.0001   0.0001   0.0000   0.0001   0.0002   0.0001   0.0000
   0.0001   0.0000   0.0000   0.0000   0.0000   0.0001   0.0001   0.0001
   0.0001   0.0001   0.0002   0.0000   0.0000   0.0001   0.0002   0.0000
Worst pmse = 0.000400  (meets spec limit 0.06)
Overall mse = 0.000075  (meets spec limit 0.02)

Mean errors:
  -0.0003   0.0004  -0.0001   0.0003   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0002   0.0000   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0003   0.0000   0.0001   0.0001   0.0000   0.0000   0.0000
   0.0001   0.0000   0.0001   0.0001   0.0000   0.0000   0.0000   0.0000
  -0.0003   0.0000  -0.0001   0.0001   0.0000   0.0001   0.0000   0.0000
   0.0000   0.0001  -0.0001   0.0000  -0.0001   0.0002   0.0001   0.0000
  -0.0001   0.0000   0.0000   0.0000   0.0000  -0.0001   0.0001  -0.0001
  -0.0001   0.0001  -0.0002   0.0000   0.0000  -0.0001   0.0000   0.0000
Worst mean error = 0.000400  (meets spec limit 0.015)
Overall mean error = 0.000013  (meets spec limit 0.0015)

0 elements of IDCT(0) were not zero
```

IEEE test conditions: -L = -300, +H = 300, sign = 1, #iters = 10000

```
Peak absolute values of errors:
   1   0   1   1   1   1   0   1
   1   0   0   1   1   0   0   0
   1   0   0   1   0   0   0   0
   0   0   0   0   1   1   1   0
   1   1   0   0   1   0   0   0
   0   0   1   1   0   0   0   0
   1   0   0   0   0   0   1   0
   0   1   0   0   1   1   0   0
Worst peak error = 1  (meets spec limit 1)
```

```
Mean square errors:
   0.0002   0.0000   0.0001   0.0002   0.0003   0.0001   0.0000   0.0001
   0.0001   0.0000   0.0000   0.0001   0.0001   0.0000   0.0000   0.0000
   0.0001   0.0000   0.0000   0.0001   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000   0.0001   0.0001   0.0001   0.0000
   0.0004   0.0003   0.0000   0.0000   0.0001   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0001   0.0001   0.0000   0.0000   0.0000   0.0000
   0.0001   0.0000   0.0000   0.0000   0.0000   0.0000   0.0001   0.0000
   0.0000   0.0002   0.0000   0.0000   0.0002   0.0002   0.0000   0.0000
Worst pmse = 0.000400  (meets spec limit 0.06)
Overall mse = 0.000056  (meets spec limit 0.02)

Mean errors:
  -0.0002   0.0000  -0.0001   0.0002  -0.0003   0.0001   0.0000  -0.0001
   0.0001   0.0000   0.0000   0.0001   0.0001   0.0000   0.0000   0.0000
  -0.0001   0.0000   0.0000   0.0001   0.0000   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0000   0.0000   0.0001  -0.0001   0.0001   0.0000
  -0.0004   0.0003   0.0000   0.0000  -0.0001   0.0000   0.0000   0.0000
   0.0000   0.0000  -0.0001   0.0001   0.0000   0.0000   0.0000   0.0000
   0.0001   0.0000   0.0000   0.0000   0.0000   0.0000   0.0001   0.0000
   0.0000   0.0002   0.0000   0.0000  -0.0002   0.0002   0.0000   0.0000
Worst mean error = 0.000400  (meets spec limit 0.015)
Overall mean error = 0.000003  (meets spec limit 0.0015)

0 elements of IDCT(0) were not zero
```

IEEE test conditions: -L = -256, +H = 255, sign = -1, #iters = 10000

```
Peak absolute values of errors:
   1   1   0   0   0   0   1   0
   0   1   1   1   1   0   0   1
   0   0   1   1   1   0   0   1
   0   1   0   0   1   0   1   1
   1   1   1   0   0   1   0   0
   0   1   1   1   1   1   1   0
   0   1   0   0   1   0   0   0
   1   0   1   1   0   1   1   0
Worst peak error = 1  (meets spec limit 1)

Mean square errors:
   0.0001   0.0002   0.0000   0.0000   0.0000   0.0000   0.0001   0.0000
   0.0000   0.0001   0.0001   0.0001   0.0001   0.0000   0.0000   0.0002
   0.0000   0.0000   0.0001   0.0001   0.0001   0.0000   0.0000   0.0001
   0.0000   0.0001   0.0000   0.0000   0.0001   0.0000   0.0001   0.0001
   0.0002   0.0002   0.0001   0.0000   0.0000   0.0003   0.0000   0.0000
   0.0000   0.0001   0.0001   0.0001   0.0001   0.0001   0.0002   0.0000
   0.0000   0.0002   0.0000   0.0000   0.0001   0.0000   0.0000   0.0000
   0.0002   0.0000   0.0001   0.0001   0.0000   0.0001   0.0001   0.0000
Worst pmse = 0.000300  (meets spec limit 0.06)
Overall mse = 0.000066  (meets spec limit 0.02)

Mean errors:
```

```
  -0.0001   0.0002   0.0000   0.0000   0.0000   0.0000  -0.0001   0.0000
   0.0000  -0.0001   0.0001  -0.0001   0.0001   0.0000   0.0000  -0.0002
   0.0000   0.0000  -0.0001   0.0001  -0.0001   0.0000   0.0000  -0.0001
   0.0000   0.0001   0.0000   0.0000  -0.0001   0.0000   0.0001   0.0001
  -0.0002   0.0002  -0.0001   0.0000   0.0000   0.0001   0.0000   0.0000
   0.0000   0.0001  -0.0001   0.0001  -0.0001   0.0001   0.0002   0.0000
   0.0000   0.0002   0.0000   0.0000  -0.0001   0.0000   0.0000   0.0000
  -0.0002   0.0000  -0.0001   0.0001   0.0000   0.0001   0.0001   0.0000
Worst mean error = 0.000200  (meets spec limit 0.015)
Overall mean error = 0.000003  (meets spec limit 0.0015)

0 elements of IDCT(0) were not zero
```

IEEE test conditions: -L = -5, +H = 5, sign = -1, #iters = 10000

```
Peak absolute values of errors:
   1   1   1   0   1   0   0   0
   0   0   1   0   0   0   0   0
   1   0   1   0   1   0   1   0
   0   0   0   1   1   0   0   0
   1   1   1   0   1   0   1   0
   0   0   1   0   1   0   0   0
   1   0   0   0   0   1   0   1
   0   0   0   1   0   1   1   0
Worst peak error = 1  (meets spec limit 1)

Mean square errors:
   0.0004   0.0001   0.0001   0.0000   0.0002   0.0000   0.0000   0.0000
   0.0000   0.0000   0.0001   0.0000   0.0000   0.0000   0.0000   0.0000
   0.0001   0.0000   0.0002   0.0000   0.0003   0.0000   0.0001   0.0000
   0.0000   0.0000   0.0000   0.0002   0.0002   0.0000   0.0000   0.0000
   0.0001   0.0003   0.0001   0.0000   0.0003   0.0000   0.0001   0.0000
   0.0000   0.0000   0.0001   0.0000   0.0001   0.0000   0.0000   0.0000
   0.0001   0.0000   0.0000   0.0000   0.0000   0.0002   0.0000   0.0001
   0.0000   0.0000   0.0000   0.0001   0.0000   0.0002   0.0001   0.0000
Worst pmse = 0.000400  (meets spec limit 0.06)
Overall mse = 0.000061  (meets spec limit 0.02)

Mean errors:
  -0.0004   0.0001  -0.0001   0.0000  -0.0002   0.0000   0.0000   0.0000
   0.0000   0.0000  -0.0001   0.0000   0.0000   0.0000   0.0000   0.0000
  -0.0001   0.0000  -0.0002   0.0000  -0.0003   0.0000   0.0001   0.0000
   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  -0.0001   0.0003  -0.0001   0.0000   0.0001   0.0000   0.0001   0.0000
   0.0000   0.0000  -0.0001   0.0000  -0.0001   0.0000   0.0000   0.0000
  -0.0001   0.0000   0.0000   0.0000   0.0000   0.0002   0.0000   0.0001
   0.0000   0.0000   0.0000   0.0001   0.0000   0.0002   0.0001   0.0000
Worst mean error = 0.000400  (meets spec limit 0.015)
Overall mean error = -0.000008  (meets spec limit 0.0015)

0 elements of IDCT(0) were not zero
```

IEEE test conditions: -L = -300, +H = 300, sign = -1, #iters = 10000

```
Peak absolute values of errors:
   0   1   1   0   0   0   0   1
   0   0   1   1   0   0   0   0
   1   0   0   0   0   0   0   1
   0   1   0   0   1   1   0   0
   1   0   0   0   0   0   0   0
   1   0   0   0   1   0   0   0
   1   1   0   0   0   0   0   0
   1   0   1   0   0   0   1   0
Worst peak error = 1  (meets spec limit 1)

Mean square errors:
   0.0000   0.0002   0.0001   0.0000   0.0000   0.0000   0.0000   0.0001
   0.0000   0.0000   0.0001   0.0001   0.0000   0.0000   0.0000   0.0000
   0.0002   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0001
   0.0000   0.0002   0.0000   0.0000   0.0002   0.0001   0.0000   0.0000
   0.0002   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
   0.0001   0.0000   0.0000   0.0000   0.0001   0.0000   0.0000   0.0000
   0.0001   0.0001   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
   0.0001   0.0000   0.0001   0.0000   0.0000   0.0000   0.0001   0.0000
Worst pmse = 0.000200  (meets spec limit 0.06)
Overall mse = 0.000036  (meets spec limit 0.02)

Mean errors:
   0.0000   0.0002  -0.0001   0.0000   0.0000   0.0000   0.0000   0.0001
   0.0000   0.0000   0.0001  -0.0001   0.0000   0.0000   0.0000   0.0000
  -0.0002   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000  -0.0001
   0.0000   0.0002   0.0000   0.0000  -0.0002   0.0001   0.0000   0.0000
  -0.0002   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  -0.0001   0.0000   0.0000   0.0000   0.0001   0.0000   0.0000   0.0000
  -0.0001   0.0001   0.0000   0.0000   0.0000   0.0000   0.0000   0.0000
  -0.0001   0.0000  -0.0001   0.0000   0.0000   0.0000  -0.0001   0.0000
Worst mean error = 0.000200  (meets spec limit 0.015)
Overall mean error = -0.000008  (meets spec limit 0.0015)

0 elements of IDCT(0) were not zero
```

Literaturverzeichnis

[AAN88] Y. Arai, T. Agui, and M. Nakajima. A fast DCT-SQ scheme for images. *Trans. of the IEICE*, page E 71(11):1095, November 1988.

[ADG+95] R. Aldridge, J. Davidoff, M. Ghanbari, O. Hands, and D. Pearson. Measurement of scene-dependent quality variations in digitally coded television pictures. *IEEE Proc.-Vis. Image Signal Processing*, 142(3):149–154, June 1995.

[Bar90] Peter G. J. Barten. Evaluation of subjective image quality with the square-root integral method. *Journal of the Optical Society of America A*, 7(10):2024–2031, October 1990.

[BBQ+97] Marina Bosi, Karlheinz Brandenburg, Schuyler Quackenbush, Louis Fielder, Kenzo Akagir, Hendrik Fuchs, Martin Dietz, Jürgen Herre, Grant Davidson, and Yoshiaki Oikawa. ISO/IEC MPEG-2 advanced audio coding. *Journal of the Audio Engineering Society*, 45(10):789–812, October 1997.

[BD91] Claude Bergeron and Eric Dubois. Gradient-based algorithms for block-oriented map estimation of motion and application to motion-compensated temporal interpolation. *IEEE Transactions on Circuits and Systems for Video Technology*, 1(1):72–85, March 1991.

[BFS] Angelos Bilas, Jason Fritts, and Jaswinder Pal Singh. Real-time parallel MPEG-2 decoding in software. Technical report, Department of Computer Science, Princeton University.

[Bra94] Karlheinz Brandenburg. ISO-MPEG-1 audio: A generic standard for coding of high-quality digital audio. *Journal of the Audio Engineering Society*, 42(10):780–791, October 1994.

[Bra97] Karlheinz Brandenburg. Overview of MPEG audio: Current and future standards for low-bit-rate audio coding. *Journal of the Audio Engineering Society*, 45(1/2):4–21, Jan/Feb 1997.

[CCC94] Mei-Juan Chen, Liang-Gee Chen, and Tzi-Dar Chiueh. One-dimensional full search motion estimation algorithm for video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(5):504–509, October 1994.

[CCCL95] Chen, Liang-Gee Chen, Tzi-Dar Chiueh, and Yung-Pin Lee. A new block-matching criterion for motion estimation and its implementation. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(3):231–236, June 1995.

[CCJC91] Liang-Gee Chen, Wai-Ting Chen, Yeu-Shen Jehng, and Tzi-Dar Chiueh. An efficient parallel motion estimation algorithm for digital image processing. *IEEE Transactions on Circuits and Systems for Video Technology*, 1(4):378–385, December 1991.

[CCL96] Chang Wen Chen, Lulin Chen, and Jiebo Luo. A cellular neuronal network for clustering-based adaptive quantization in subband video compression. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(6):688–692, December 1996.

[CJOS94] Tae-Yun Chung, Kyu-Hwan Jung, Young-Nam Oh, and Dong-Ho Shin. Quantization control for improvement of image quality compatible with MPEG2. *IEEE Transactions on Consumer Electronics*, 49(4):821–825, November 1994.

[CK97] Junavit Chalidabhongse and C.-C. Jay Kuo. Fast motion vector estimation using multiresolution-spatio-temporal correlations. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(3):477–488, June 1997.

[CL91] Nam Ik Cho and Sang Uk Lee. Fast algorithm and implementation of 2-D discrete fourier transform. *IEEE Transactions on Circuits and Systems*, 38(3):297–305, March 1991.

[CL93] Keith Hung-Kei Chow and Ming L. Liou. Genetic motion search algorithm for video compression. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(6):440–445, December 1993.

[CLCR93] K. W. Chun, K. W. Lim, H. D. Cho, and J. B. Ra. An adaptive perceptual quantization algorithm for video coding. *IEEE Transactions on Consumer Electronics*, 39(3):555–558, August 1993.

[CP94] Jinho Choi and Daechul Park. A stable feedback control of the buffer state using the controlled lagrange multiplier method. *IEEE Transactions on Image Processing*, 3(5):546–557, September 1994.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[CW97] Chih-Feng Chang and Jia-Shung Wang. A stable buffer control strategy for MPEG coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(6):920–924, December 1997.

[DL96] Wei Ding and Bede Liu. Rate control of MPEG video coding and recording by rate-quantization modeling. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(1):12–20, 1996.

[FvDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, principles and practice*. Addison-Wesley, 1990.

[FW92] E. Feig and S. Winograd. Fast algorithms for the discrete cosine transform. *IEEE Transactions on Signal Processing*, 40(9):2174–2193, September 1992.

[Gha90] M. Ghanbari. The cross-search algorithm for motion estimation. *IEEE Transactions on Communications*, 38(7):950–953, July 1990.

[GR] Kevin L. Gong and Lawrence A. Rowe. Parallel MPEG-1 video encoding. Technical report, Computer Science Division - EECS, University of California.

[GV91] C. A. Gonzales and E. Viscito. Motion video adaptive quantization in the transform domain. *IEEE Transactions on Circuits and Systems for Video Technology*, 1(4):374–378, December 1991.

[Har76] Robert M. Haralick. A storage efficient way to implement the discrete cosine transform. *IEEE Transactions on Computers*, C-25:764–765, July 1976.

[Hau94] Gert Hauske. *Systemtheorie der visuellen Wahrnehmung*. B.G.Teubner Stuttgart, 1994.

[HHK94] Hermann Haken and Maria Haken-Krell. *Erfolgsgeheimnisse der Wahrnehmung: Synergetik als Schlüssel zum Gehirn*. Ullstein, 1994.

[HLV96] Dzung T. Hoang, Elliot Linzer, and Jeffrey S. Vitter. Lexicographically optimal rate control for video coding with MPEG buffer constraints. Technical Report CS-1996-02, Department of Computer Science, Duke University, February 1996.

[HP92] C. Horne and A. Puri. Video coding with adaptive quantization and rate control. *SPIE Visual Communications and Image Processing*, 1818:798–806, 1992.

[HPN97] Barry G. Haskell, Atul Puri, and Arun N. Netravali. *Digital Video: an introduction to MPEG-2*. Digital Multimedia Standards. Chapman & Hall, 1997.

[HYL97] Shih-Chang Hsia, Jar-Ferr Yang, and Bin-Da Liu. Efficient psotprocessor for blocky effect removal based on transform characteristics. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(6):924–929, December 1997.

[HZY96] Yan Huang, Xinhua Zhuang, and Changsheng Yang. Two block-based motion compensation methods for video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(1):123–126, February 1996.

[IEE91] *IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform. IEEE Std 1180-1990*, March 1991.

[Jai89] Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall International, Inc., 1989.

[JCC94] Her-Ming Jong, Liang-Gee Chen, and Tzi-Dar Chiueh. Parallel architectures for 3-step hierarchical search block-matching algorithm. *IEEE Transactions on Circuits and Systems for Video Technology*, 4:407–415, August 1994.

[JGP92] Hartwig Jeschke, Klaus Gaedke, and Peter Pirsch. Multiprocessor performance for real-time processing of video coding applications. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(2):221–230, June 1992.

[Kel61] D. H. Kelly. Visual responses to time-dependent stimuli. ii. single-channel model of the photopic visual system. *Journal of the Optical Society of America*, 51(7):747–754, July 1961.

[Ken74] William B. Kendall. A new algorithm for computing correlations. *IEEE Transactions on Computers*, pages 88–90, January 1974.

[KHJ+96] Chung-Ming Kuo, Chaur-Heh Hsieh, Yue-Dar Jou, Hsieh-Cheng Lin, and Po-Chiang Lu. Motion estimation for video compression using kalman filtering. *IEEE Transactions on Broadcasting*, 42(2):110–116, June 1996.

[KK95] Shanika A. Karunasekera and Nick G. Kingsbury. A distortion measure for blocking artifacts in images based on human visual sensitivity. *IEEE Transactions on Image Processing*, 4(6):713–724, June 1995.

[KT] Marwan Krunz and Satish K. Tripathi. Modeling bit rate variations in MPEG sources. Technical report, University of Maryland Institute for Advanced Computer Studies.

[LA95] Gopal Lakhani and Vamsi Ayyagari. Improved huffman code tables for jpeg's encoder. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(6):562–564, December 1995.

[LC94] M.H. Lee and G. Crebbin. Image sequence coding using quadtree-based block-matching motion compensation and classified vector quantisation. *IEEE Proc.-Vis. Image Signal Processing*, 141(6):453–460, December 1994.

[LD97] Jungwoo Lee and BRadley W. Dickinson. Rate-distortion optimized frame type selection for MPEG encoding. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(3):501–509, June 1997.

[LF96] Lurng-Kuo Liu and Ephraim Feig. A block-based gradient descent search algorithm for block motion estimation in video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(4):419–423, August 1996.

[Lin95] Elliot Linzer. Maximum IDCT output of MPEG-2 encoders. Technical report, IBM Research, June 1995.

[LL97] Jianhua Lu and Ming L. Liou. A simple and efficient search algorithm for block-matching motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(2):429–433, April 1997.

[LT97] Yih-Chuan Lin and Shen-Chuan Tai. Fast full-search block-matching algorithm for motion-compensated video compression. *IEEE Transactions on Communications*, 45(5):527–531, May 1997.

[LVW96] Daniel Lauzon, André Vincent, and Limin Wang. Performance evaluation of MPEG-2 video coding for HDTV. *IEEE Transactions on Broadcasting*, 42(2):88–93, June 1996.

[LWLS93] Liang-Wei Lee, Jhing-Fa Wang, Jau-Yien Lee, and Jung-Dar Shie. Dynamic search-window adjustment and interlaced search for block-matching algorithm. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(1):85–87, February 1993.

[LZ93] Bede Liu and André Zaccarin. New fast algorithms for the estimation of block motion vectors. *IEEE Transactions on Circuits and Systems for Video Technology*, 3(2):148–157, April 1993.

[LZ96] Xiaobing Lee and Ya-Qin Zhang. A fast hierarchical motion-compensation scheme for video coding using block feature matching. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(6):627–635, December 1996.

[LZL94] Renxiang Li, Bing Zeng, and Ming L. Liou. A new three-step search algorithm for block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(4):438–442, August 1994.

[Mäu91] Rudolf Mäusl. *Fernsehtechnik*. Hüthig, 1991.

[MPFL96] Joan L. Mitchell, William B. Pennebaker, Chad E. Fogg, and Didier J. LeGall. *MPEG Video Compression Standard*. Digital Multimedia Standards. Chapman & Hall, 1996.

[NAR74] T. Natarajan N. Ahmed and K. R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, C-23:90–93, January 1974.

[NBK97] Balas Natarajan, Vasudev Bhaskaran, and Konstantinos Konstantinides. Low-complexity block-based motion estimation via on-bit transforms. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(4):702–706, August 1997.

[PA90] A. Puri and R. Aravind. On comparing motion-interpolation structures for video coding. *SPIE Visual Communications and Image Processing*, 1360:1560–1571, 1990.

[PA91] Atul Puri and R. Aravind. Motion-compensated video coding with adaptive perceptual quantization. *IEEE Transactions on Circuits and Systems for Video Technology*, 1(4):351–361, December 1991.

[PA92] A. Puri and R. Aravind. Adaptive perceptual quantization for video compression. *SPIE Visual Communications and Image Processing*, 1605:297–300, 1992.

[PA94] Mark R. Pickering and John F. Arnold. A perceptually efficient vbr rate control algorithm. *IEEE Transactions on Image Processing*, 3(5):527–531, September 1994.

[PAF97] Mark R. Pickering, John F. Arnold, and Michael R. Frater. An adaptive search length algorithm for block matching motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(6):906–912, December 1997.

[Pea90] William A. Pearlman. Adaptive cosine transform image coding with constant block distortion. *IEEE Transactions on Communications*, 38(5):698–703, May 1990.

[Pet95] M. Petrou. Separable 2-D filters for the detection of ramp edges. *IEEE Pro.-Vis. Image Signal Processing*, 142(2):228–231, August 1995.

[PM93] William B. Pennebaker and Joan L. Mitchell. *JPEG Still Image Data Compression Standard*. Van Nostrad Reinhold, 1993.

[PM96] Lai-Man Po and Wing-Chung Ma. A novel four-step search algorithm for fast block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3):313–317, June 1996.

[PSR] Ketan Patel, Brian C. Smith, and Lawrence A. Rowe. Performance of a software MPEG video decoder. Technical report, Computer Science Division - EECS, University of California.

[RE93] Standley J. Reeves and Steven L. Eddins. Comment on "iterative procedures for reduction of blocking effects in transform image coding". *IEEE Transactions on Circuits and Systems for Video Technology*, 3(6):439–440, December 1993.

[RKR96] P. Venkat Rangan, Srihari Sampath Kumar, and Sreerang Rajan. Continuity and synchronization in MPEG. *IEEE Journal on selected Areas in Communications*, 14(1):52–60, January 1996.

[RL] Viresh Ratnakar and Miron Livny. Rd-opt: An efficient algorithm for optimizing DCT quantization tables.

[Rob66] J. G. Robson. Spatial and temporal contrast-sensitivity functions of the visual system. *Journal of the Optical Society of America*, 56:1141–1142, August 1966.

[ROV94] Kannan Ramchandran, Antonio Ortega, and Martion Vetterli. Bit allocation for dependent quantization with applications to multiresolution and MPEG video coders. *IEEE Transactions on Image Processing*, 3(5):533–545, September 1994.

[RV94] Kannan Ramchandran and Martin Vetterli. Rate-distortion optimal fast thresholding with complete JPEG/MPEG decoder compatibility. *IEEE Transactions on Image Processing*, 3(5):700–704, September 1994.

[SG88] Yair Shoham and Allen Gersho. Efficient bit allocation for an arbitrary set of quantizers. *IEEE Transactions on Acoustics and Signal Processing*, 36(9):1445–1453, September 1988.

[SK91] S.C.Chan and K.L.Ho. A new two-dimensional fast cosine transform algorithm. *IEEE Transactions on Signal Processsing*, 39(2):481–485, February 1991.

[SKCJ97] Huifang Sun, Wilson Kwok, Max Chien, and C. H. John Ju. MPEG coding performance improvement by jointly optimizing coding mode decisions and rate control. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(3):449–457, June 1997.

[SS96] Ken Sauer and Brian Schwartz. Efficient block motion estimation using integral projections. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(5):513–518, October 1996.

[SX97] Y. Q. Shi and X. Xia. A thresholding multiresolution block matching algorithm. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(2):437–440, April 1997.

[TG96] H. Tan and Ghanbari. Block classification for an adaptive 1-D/2-D DCT video coding. *IEEE Proc.-Vis. Image Signal Processing*, 143(3):155–163, June 1996.

[TM78] B. D. Tseng and W. C. Miller. On computing the discrete cosine transform. *IEEE Transactions on Computers*, C-27(10):966–968, October 1978.

[TPN96] Soon Hie Tan, Khee K. Pang, and King N. Ngan. Classified perceptual coding with adaptive quantization. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(4):375–388, August 1996.

[WA85] Andrew B. Watson and Albert J. Ahumada, Jr. Model of human visual-motion sensing. *Journal of the Optical Society of America A*, 2(2):322–341, February 1985.

[Wan94] Limin Wang. Bit rate control for hybrid DPCM/DCT video codec. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(5):509–516, October 1994.

[Wat90] Andrew B. Watson. Perceptual-components architecture for digital video. *Journal of the Optical Society of America A*, 7(10):1943–1953, October 1990.

[Wen88] Broder Wendland. *Fernsehtechnik*. Hüthig, 1988.

[Win78] S. Winograd. On computing the discrete fourier transform. *Mathematics of Computation*, pages 175–199, January 1978.

[WP91] H.R. Wu and F.J. Paoloni. A two-dimensional fast cosine transform algorithm based on hou's approach. *IEEE Transactions on Signal Processing*, 39(2):544–546, February 1991.

[YK93] Robert W. Young and Nick G. Kingsbury. Frequency-domain motion estimation using a complex lapped transform. *IEEE Transactions on Image Processing*, 2(1):2–17, January 1993.

[Zak92] Avideh Zakhor. Iterative procedures for reduction of blocking effects in transform image coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(1):91–95, March 1992.

[ZLL97] Bing Zeng, Renxiang Li, and Ming L. Liou. Optimization of fast block motion estimation algorithms. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(6):833–843, December 1997.

Erklärung

Hiermit versichere ich, diese Arbeit
selbständig verfaßt und nur die
angegebenen Quellen benutzt zu haben.

(Dirk Farin)