ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Departement Elektrotechnik
Institut für Integrierte Systeme
Prof. Wolfgang Fichtner

JPEG-3D Codec Software

Axel Burri

Semesterarbeit SS 2001

Betreuung: Andreas Burg

19. August 2001

Inhaltsverzeichnis

1 Einleitung **6**

2 Video Codec Design **7**
2.1 Anforderungen . . . 7
2.1.1 Akzeptanz . . . 7
2.2 Programmiersprache . . . 7
2.3 Fehlerresistenz . . . 8
2.3.1 Fehlertypen . . . 8
2.3.2 Anforderungen . . . 8
2.3.3 Mechanismen zur Fehlerresistenz . . . 9
2.4 Kompression . . . 9

3 Dateiformat **11**
3.1 JPEG vs. J3D . . . 11
3.2 Marker . . . 11
3.2.1 Codes . . . 13
3.3 Struktur . . . 13
3.3.1 Sequence . . . 14
3.3.2 Frame . . . 14
3.3.3 Scan . . . 14
3.3.4 Group of Cubes . . . 15

4 J3D Programmbibliothek **16**
4.1 Installation . . . 16
4.2 Ausführen des Encoders/Decoders . . . 17
4.2.1 Encoder . . . 17
4.2.2 Decoder . . . 17
4.3 Benutzung der Programmbibliothek . . . 18
4.3.1 J3D - API . . . 18
4.3.2 Module . . . 19

5 Programmstruktur **23**
5.1 Module . . . 23
5.2 Encoder . . . 23

5.2.1 Main Controller (cmainct.c) 25
5.2.2 Preprocessing Controller (cprepct.c) 25
5.2.3 Input Manager (cinput.c) 25
5.2.4 Color Converter (ccolor.c) 25
5.2.5 Coefficient Controller (ccoefct.c) 25
5.2.6 Forward 3D-DCT (cf3ddct.c) 26
5.2.7 Variable Length Coder (cvlc.c) 26
5.2.8 Marker Writer (cmarker.c) 26
5.2.9 MCU Manager (cmcu.c) 26
5.2.10 Output Manager (coutput.c) 26
5.3 Decoder 26
5.3.1 Main Controller (dmainct.c) 28
5.3.2 Coefficient Controller (dcoefct.c) 28
5.3.3 Variable Length Decoder (dvlc.c) 28
5.3.4 Inverse 3D-DCT (di3ddct.c) 28
5.3.5 Marker Reader (dmarker.c) 28
5.3.6 Input Manager (dinput.c) 28
5.3.7 MCU Manager (dmcu.c) 28
5.3.8 Postprocessing Controller (dpostct.c) 29
5.3.9 Color Converter (dcolor.c) 29
5.3.10 Output Manager (doutput.c) 29

6 Schlussfolgerungen und Ausblick 30

Abbildungsverzeichnis

3.1 Vergleich JPEG - J3D 12
3.2 Dateiformat von J3D 13

5.1 Datenfluss Encoder 24
5.2 Baumstruktur Module (Encoder) 24
5.3 Datenfluss Decoder 27
5.4 Baumstruktur Module (Decoder) 27

Tabellenverzeichnis

2.1 Programmiersprachen im Vergleich 8

3.1 Marker Codes 13

4.1 API Kompression 19
4.2 API Dekompression 20

Kapitel 1

Einleitung

Digitale Videoübertragung ist nicht mehr wegzudenken. Videokonferenzen, welche im Business Sektor bereits weit verbreitet sind, dringen nun auch allmählich in den Consumer Sektor vor; Bildtelefonie wird durch in den letzten Jahren massiv steigende Bandbreite und Verfügbarkeit im Internet wie auch im Mobilen Sektor ermöglicht.

Zur digitalen Übertragung von Videodaten ist es unumgänglich, die gigantische Datenflut zu komprimieren. Dazu gibt es schon einige weit verbreitete Algorithmen wie z.B. MPEG-2, welches nicht zuletzt durch den Aufschwung der DVD grosse Akzeptanz gefunden hat. Obschon diese Videoformate beträchtliche Kompressionsraten aufweisen, haben sie doch einen grossen Nachteil: sie sind sehr Rechenintensiv und benötigen folglich teure Hardware um sie überhaupt zu betreiben. Ausserdem sind sie Asymmetrisch, d.h. der Aufwand auf der Kompressionsseite ist weitaus grösser als jener auf der Dekompressionsseite.

JPEG-3D (kurz J3D), der Inhalt dieser Semesterarbeit geht einen anderen Weg. Ressourcenhungrige Algorithmen wie Motion Compensation werden hier nicht benötigt, vielmehr basiert die Kompression von J3D auf die Transformation mittels 3D-DCT, einem einfachen und schnellen Algorithmus. Andreas Burg und Roni Keller haben in Ihrer Diplomarbeit [2] bereits einen Integrierten Schaltkreis gebaut, welcher eindrücklich zeigt, wie schlank und effizient (im Vergleich zu MPEG-2 Encoder) sich ein Video-Codec mittels 3D-DCT implementieren lässt. Ihrem Codec fehlt jedoch ein geeignetes Dateiformat, welches Fehlerresistenz gewährleisten und zudem eine solide Basis für Erweiterungen bieten soll.

Aus diesem Grund wurde das J3D Softwarepaket implementiert, welches mittels 3D-DCT Videodaten komprimiert, ein bekanntes Dateiformat erweitert (JPEG) und zugleich eine Plattform für Erweiterungen und Experimente bietet [1].

Kapitel 2

Video Codec Design

2.1 Anforderungen

Der Video-Codec hat folgende Anforderungen zu erfüllen:

- Fehlerresistenz: Möglichkeit der Übertragung von Videodaten über fehlerbahaftete Übertragungsmedien
- Modularität: Möglichkeit der einfachen Erweiterung und Plattform für Experimente mit dem Algorithmus
- Gute Kompressionsrate
- Akzeptanz: Wahl eines Formates welcher auf möglichst breiter Basis akzeptiert und auch benutzt wird
- Bestmögliche Performance

2.1.1 Akzeptanz

Um eine möglichst breite Akzeptanz des Video-Codecs zu erreichen, wurde mit JPEG ein Format als Basis gewählt, welches sehr ähnliche Algorithmen benutzt und heutzutage sehr weit verbreitet ist. Für weitergehende Informationen betreffend Dateiformat siehe Kapitel 3.

2.2 Programmiersprache

Die heutzutage wohl am weitesten verbereitete Sprache, um schnelle und modular aufgebaute Programme zu schreiben ist C++. Die Wahl der Programmierspache fiel jedoch auf C, da einerseits die Performance etwas besser ist und andererseits, was noch wichtiger ist, der frei im Quellcode verfügbare IJG JPEG Codec [3] in dieser Sprache geschrieben wurde (s. Tab. 2.1). So können ähnliche Teile sehr einfach übernommen, angepasst und erweitert werden.

	C	C++	Java
Performance	+	+	-
Plattformunabh.	-	-	+
Enwicklungszeit	+	o	-
Erweiterbarkeit	o	+	+

Tabelle 2.1: Programmiersprachen im Vergleich

2.3 Fehlerresistenz

2.3.1 Fehlertypen

Es gibt verschiedene Typen von Fehlern, welche einen Videostrom während des Transports beeinträchtigen können. Sie sind abhängig vom Transportmedium und -protokoll.

Paketverluste sind bei Paketorientierten Transportprotokollen die wohl wahrscheinlichsten Fehlerursachen. Protokolle wie TCP, welche mit Paketverlusten umgehen können (indem sie die Pakete nochmal anfordern) sind für die Übertragung von (Live-) Video-Datentrömen aufgrund hoher Latenzzeiten weniger geeignet. Ein Codec muss mit solchen Verlusten umgehen können.

Bitfehler kommen vor allem bei Drahtlosen Übertragungsmedien vor. Ein Codec sollte also eine möglichst hohe Bitfehlerrate akzeptieren können.

Bursts sind ebenfalls bei Drahtlosen Übertragungsmedien ziemlich häufig, auch diese Fehlerquellen sollte ein Codec akzeptieren.

Es besteht für alle Arten von Fehlern die Möglichkeit, durch Einfügen von Redundanz oder mittels geeigneten Protokollen einen Datenstrom über Fehlerbehaftete Übertragungsmedien sicher zu übertragen. Diese haben aber den Nachteil, dass sie zum Teil einen gewaltigen Daten-Overhead mit sich tragen und somit den Videostrom entweder stark verzögern oder sich nachteilig auf die Bandbreite auswirken.

2.3.2 Anforderungen

Der Betracher einer Videsequenz kann einige Fehler verschmerzen, bestenfalls bemerkt er diese gar nicht. Es ist deshalb nicht zwingend nötig, einen Video-Datenstrom fehlerfrei zu übertragen, umso wichtiger wird aber der schonende Umgang mit Fehlern. So kann zum Beispiel ein durch Paketverluste verlorengegener Bildteil durch eine Interpolation des vorhergehenden und nachfolgenden Bildes ersetzt werden.

2.3.3 Mechanismen zur Fehlerresistenz

Resynchronisation

Resynchronisationsfähigkeit ist die wohl wichtigste Eigenschaft eines Fehlerresistenten Codecs. Sie kann durch Einfügen von Markern erreicht werden, wie dies auch im J3D Codec gemacht wird (s. Kap. 3.2).

Der J3D Codec ist mit Schwerpunkt auf eine Übertragung mittels Paketbasierten Protokollen konzipiert worden, kann aber ohne weiteres auch mit anderen Protokollen umgehen. Es gibt jedoch einige Punkte welche für eine optimale Fehlerbehandlung beachtet werden sollten:

- Bei Paketbasierten Protokollen ist es von Vorteil, dass jeweils am Anfang jedes Paketes ein Marker steht. Ist dies nicht der Fall, werden nach einem Paketverlust im nachfolgenden (korrekt übertragenen) Paket alle Daten bis zum ersten Marker verworfen, da bis zu diesem hin keinerlei Synchronisation möglich ist.
- Bei häufigen Bitfehlern/Bursts sollte das Restart Marker Intervall vernünftig klein gewählt werden. So wird verhindert dass grosse Teile der Bilder verworfen werden.

Interleaving

Unter Interleaving versteht man die nicht lineare Anordnung der Bildblöcke im Datenstrom. Sie werden also nach einem definierten Muster verstreut kodiert. Wird das Interleaving intelligent gewählt, verstreuen sich fehlerhaft übertragene Daten über das gesamte Bild anstatt einen zusammenhängenden Bereich zu zerstören. Dies bringt einen deutlich besseren Gesamteindruck des Bildes.

Der J3D-Codec ist für Interleaving vorbereitet, hierzu müssen aber im MCU Manager (s. Kap. 5) einige kleinere Änderungen vorgenommen werden.

2.4 Kompression

Die Kompression in J3D erfolgt analog zu JPEG und wird hier deshalb nur grob beschrieben. Für weiterführende Informationen siehe [4].
Die folgenden Schritte werden für jeden einzelnen Cube (8x8x8 Pixel) abgehandelt:

1. Shiften
2. 3D-DCT anwenden
3. Quantisieren
4. Letzten DC Koeffizienten vom aktuellen subtrahieren

5. Cube neu anordnen (Zigzag Modell[1])
6. Zero run length encoding
7. Non-zero Koeffizienten in binäre Zahlen variabler Länge und dessen Länge aufspalten
8. Entropie Encoder (Huffmann) auf Lauflänge anwenden
9. Entopie Kodierte Informationen ausgeben

[1] im Gegensatz zu JPEG ein 3D-Zigzag Modell [2]

Kapitel 3

Dateiformat

3.1 JPEG vs. J3D

Das Dateiformat von J3D ist sehr stark an demjenigen von JPEG angelehnt. Der einzige Unterschied (nebst grösseren Blöcken) ist, dass wir es hier nicht mit einem einzigen Bild mit wohldefinierter Geometrie (und somit endlicher Dateigrösse) zu tun haben, sondern mit einem nichtdeterministischen Datenstrom (*bitstream*). Dieses Problem ist jedoch sehr einfach zu umgehen, indem ein EOI Marker (End Of Image), welcher in JPEG das Ende der Datei anzeigt, in J3D als Ende einer Sequenz angesehen wird, also nicht zwingend das Ende des gesamten Datenstromes anzeigt. Nebst den unterschiedlichen Blockgrössen (JPEG 8x8, J3D 8x8x8) und daraus resultierenden grösseren Quantisierungstabellen gibt es deshalb keine weiteren Unterschiede zu JPEG. (siehe auch Abb. 3.1)

3.2 Marker

Die Marker dienen einerseits dazu, allerlei Informationen über den Datenstrom zu definieren (wie z.B. die Bildgeometrie oder die Anzahl Farbkomponenten). Andererseits sind Marker ein wesentlicher Bestandteil des Resynchronisationsmodells, da sie ein wohldefiniertes Format besitzen und somit im Datenstrom einfach erkannt werden können. So ist der Decoder nicht gezwungen, den Datenstrom von Anfang an gelesen zu haben, sondern kann sich mittendrin einhängen, auf den nächsten Marker warten und kennt danach alle zur Synchronisation nötigen Informationen.

Dieselbe Strategie kann der Decoder auch bei Bitfehlern verfolgen: er wartet (bzw. verwirft alle Informationen) einfach auf den nächsten Marker (mehr dazu siehe Kap. 2.3).

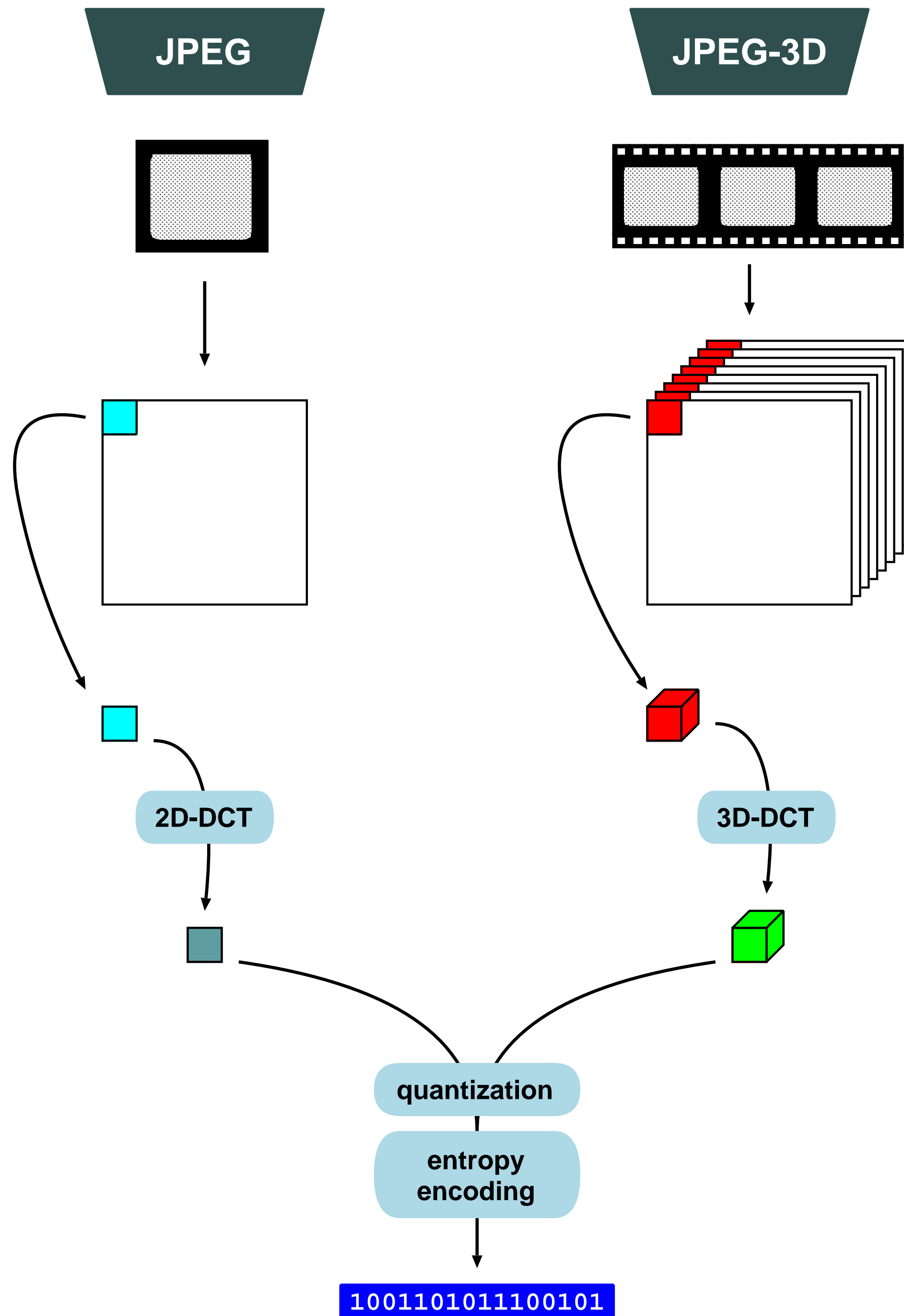

Abbildung 3.1: Vergleich JPEG - J3D

3.2.1 Codes

Ein Marker ist mindestens 2 Bytes lang und beginnt immer auf Bytegrenzen mit einem `0xff` Datenwort, gefolgt von einem 1 Byte langem Marker-Code, welcher die Art des Markers definiert. Bei Marker variabler Länge (z.B. DHT) definieren die nachfolgenden 2 Bytes dessen Gesamtlänge.

Tabelle 3.1 zeigt eine Übersicht der verwendeten Marker, für deren genaue Definition siehe [4].

Label	Code	Beschreibung	Kapitel
`SOI`	`0xd8`	Start Of Sequence	3.3.1
`EOI`	`0xd9`	End Of Sequence	3.3.1
`SOF`	`0xc0`	Start Of Frame (Header)	3.3.2
`SOS`	`0xda`	Start Of Scan (Header)	3.3.3
`DQT`	`0xdb`	Quantisation Table Definition	3.3.1
`DRI`	`0xdd`	Restart Interval Definition	3.3.1
`DHT`	`0xc4`	Huffman Table Definition	3.3.2
`RSTn`	`0xd0+n`	Restart Marker n=[0..7]	3.3.3

Tabelle 3.1: Marker Codes

3.3 Struktur

Abbildung 3.3 zeigt die Aufteilung einer J3D-Videosequenz in *sequence*, *frame* und *scan*. Diese Aufteilung erlaubt grosse Flexibilität in Hisicht auf Kompressionsrate und Transport über fehlerbehafteten Übertragungskanälen.

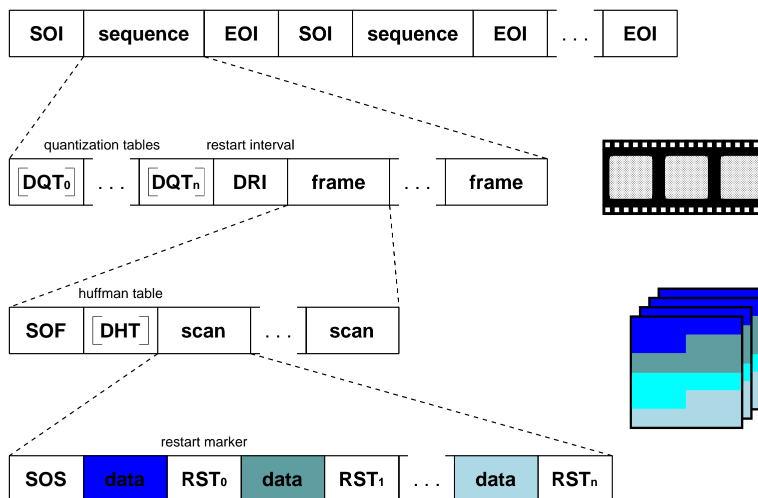

Abbildung 3.2: Dateiformat von J3D

3.3.1 Sequence

Eine *sequence* repräsentiert eine gesamte Videosequenz[1]. Sie wird durch SOI und EOI Marker eingeschlossen. In einem J3D-Datenstrom können beliebig viele *sequences* auftreten. Eine *sequence* ist folgendermassen unterteilt:

Quantization Tables (DQTn, optional) Definiert Quantisierungstabellen. Es können bis zu vier gleichzeitig verwendet werden. Die Zuordnung einer Quantisierungstabelle zu einer Farbkomponente wird im SOF Marker (s. 3.3.2 und Abb. 3.3) definiert.

Restart Interval (DRI) Definiert die Anzahl MCU's pro *group of cube* und somit die Anzahl benötigter Restart Markers (RSTn) pro *scan*.

Frames Mindestens ein *frame* (s. 3.3.2).

3.3.2 Frame

Ein *frame* definiert die Attribute der einzelnen Bilder. Durch Verwendung mehrerer *frames* ist es somit möglich, u.a. die Grösse oder Anzahl Farbkomponenten des Bildes mitten im Datenstrom zu ändern. Dies kann äusserst nützlich sein falls die verfügbare Bandbreite des Übertragungskanals sich ändert.

Start Of Frame (SOF) Definiert die Bildgeometrie (Breite, Höhe) und die Anzahl Farbkomponenten inklusive Downsampling-Factor und Index der Quantisierungstabellen (welche im Header einer *sequence* (s. 3.3.1) definiert wurden).

Huffman Table (DHT, optional) Definiert Huffmann-Tabellen. Es können maximal zwei Huffmann-Tabellenpaare (DC und AC) gleichzeitig verwendet werden. Die Zuordnung einer Huffmann-Tabelle zu einer Farbkomponente wird im SOS Marker (s. 3.3.3 und Abb. 3.3) definiert.

Scans Mindestens ein *scan* (s. 3.3.3).

3.3.3 Scan

Ein *scan* beinhaltet die eigentlichen Bilddaten und Informationen zu deren Dekodierung. Diese Daten werden in *group of cubes* unterteilt (s. 3.3.4) und mit Restart Marker (zur Resynchronisation) versehen.

Start Of Scan (SOS) Definiert die Anzahl Farbkomponenten und deren Huffmann-Tabellen-Index (welche im Header eines *scans* (s. 3.3.3) definiert wurden).

[1] Analog zu einem Einzelbild in JPEG

Data Jeweils eine *group of cubes* Bilddaten (s. 3.3.4) als Huffmann-Kodierter Datenstrom, gefolgt von einem Restart Marker.

Restart Marker (RSTn) Dient zur Resynchronisation bei Datenverlust.

3.3.4 Group of Cubes

Eine *group of cubes* stellt eine Menge von DCT-Koeffizienten-Würfel dar, dessen Anzahl im DRI Marker (s. 3.3.1) bestimmt wurde. Im Datenstrom ist sie Huffmann-Kodiert und bildet darin die kleinste Einheit.

Kapitel 4

J3D Programmbibliothek

In diesem Kapitel wird die Programmbibliothek [1], insbesondere der Encoder und Decoder, näher erläutert. Die Programmbibliothek wurde für UNIX geschrieben und unter Solaris und Linux getestet.

4.1 Installation

Mit dem Befehl:

```
make [debug=high|normal|none] [optimize=full|none]
```

wird das Programmpaket compiliert. Anschliessend finden sich im Verzeichnis zwei ausführbare Programme (`cj3d` und `dj3d`) sowie die Programmbibliothek `libj3d.a`.

- Die optionalen `debug=` Flags steuern die debug-Ausgaben der Bibliothek[1]. Mit `debug=none` werden keinerlei Laufzeitinformationen ausgegeben, mit `debug=normal` die wichtigsten, mit `debug=high` zusätzlich auch allerlei Informationen über interne Tabellen.

- Die optionalen `optimize=` Flags steuern den Optimierer des Compilers, welcher mit `optimize=full` auf maximaler Stufe läuft.

Anschliessend müssen die Programme `cj3d` und `dj3d` manuell in ausführbare Ordner und die Programmbibliothek `libj3d.a` in ein Bibliotheksverzeichnis kopiert werden:

```
cp cj3d dj3d /usr/local/bin/
cp libj3d.a /usr/local/lib/
```

[1]Soll die Ausgabe des Encoders nach `stdout` geleitet werden muss unbedingt mit `debug=none` kompiliert werden!

4.2 Ausführen des Encoders/Decoders

4.2.1 Encoder

Der Encoder wird folgendermassen aufgerufen:

```
cj3d [-o output_file] [-q quality] input_files...
```

Dieser Aufruf startet die Kompression der Dateien `input_files`. Diese müssen im BMP-Format vorliegen.

Flags

-o output_file	Output Dateiname. Wird dieser nicht angegeben erfolgt die Ausgabe nach `stdout`.
-q quality	Qualität der Kompression. Analog zu JPEG im Bereich [0..100] (0 = schlechteste Qualität und beste Kompressionsrate). Standardwert: 50.

Beispiele

Hier einige Beispiele zur Benutzung des Encoders:

```
(1) cj3d -q 30 -o test.j3d inimg/*.bmp
(2) cj3d inimg/*.bmp | send_video
```

Im Beispiel (1) werden alle BMP-Dateien aus dem Verzeichnis `inimg` mit 30% Qualität in eine Datei `test.j3d` geschrieben. Beispiel (2) zeigt die Benutzung des Encoders zum Versenden des Datenstromes mittels eines zusätzlichen (fiktiven) Programmes `send_video`. Mittels einer Pipe[2] werden die Ausgangsdaten des Encoders direkt nach `stdin` des Send-Prozesses geleitet.

4.2.2 Decoder

Der Decoder wird folgendermassen aufgerufen:

```
dj3d [-i input_file] output_mask
```

Dieser Aufruf startet die Dekompression. Die Ausgabe erfolgt in die Dateien `output_maskXXXXX.rgb` wobei `XXXXX` durch eine aufsteigende Zahlenfolge ersetzt wird. Das Format dieser Dateien ist RGB, das heisst es werden nur die Rot/Grün/Blau Rohdaten ausgegeben.

[2] Eine Pipe (das Symbol '|') leitet `stdout` des ersten Progzesses nach `stdin` des zweiten Prozesses

Flags

-i input_file	Input Dateiname. Wird dieser nicht angegeben wird der Input von `stdin` gelesen.

Beispiele

Hier einige Beispiele zur Benutzung des Decoders:

```
(1) dj3d -i test.j3d outimg/frame
(2) receive_video | dj3d outimg/frame
```

Im Beispiel (1) wird die Datei `test.j3d` dekodiert, bei (2) wird die Ausgabe eines zusätzlichen (fiktiven) Programmes `receive_video` als Input-Datenstrom verwendet. Die Videosequenz wird im Ordner `outimg` als einzelne Bilddateien (`frame00000.rgb frame00001.rgb ...`) ausgegeben. Diese Sequenz kann z.B. mit dem Progamm `animate` angezeigt werden:

```
animate -delay 10 -depth 8 -size 176x144 outimg/frame*.rgb
```

Die Angabe der Geometrie der Bilddaten (`-size 176x144`) ist unerlässlich da RGB-Dateien in dieser Hinsicht keinerlei Informationen enthalten.

4.3 Benutzung der Programmbibliothek

Die Ausführbaren Programme zur Komression bzw. Dekompression (`cj3d, dj3d`) sind Beispiele wie die Programmbibliothek `libj3d` benutzt werden kann. Dieser Abschnitt soll erläutern, wie `libj3d` in eigenen Programmen eingebunden wird.

4.3.1 J3D - API

Das API der Programmbibliothek besteht aus einigen einfachen Befehlen zum Starten der Kompression bzw. Dekompression, welche hier kurz erklärt werden. Um auf die im folgenden aufgelisteten Befehle zugreifen zu können, muss zunächst die Datei `j3dlib.h` eingebunden werden:

```
#include "j3dlib.h"
```

Ausserdem muss dem Compiler mitgeteilt werden, dass die Library `libj3d.a` benötigt wird:

```
gcc -c testapp.c -o testapp.o
gcc -o testapp testapp.o libj3d.a
```

Dieses Beispiel Kompiliert ein Programm `testapp` welches die J3D-Programmbibliothek verwendet.

Kompression

Tabelle 4.1 zeigt eine Auflistung der Befehle, welche für die Kompression benötigt werden. Die Reihenfolge sollte möglichst beibehalten werden.

j3d_compress_ptr j3d_create_compress() Generiert eine Struktur vom Typ `j3d_compress_struct` welche alle Informationen bezüglich Format, Tabellen und Modulen beinhaltet. Diese Struktur wird in allen folgenden Funktionsaufrufen benötigt.

void j3d_c_set_std_params(j3d_compress_ptr) Setzt Standardwerte in alle Parameter der `j3d_compress_struct`. Es ist empfehlenswert, diese Prozedur aufzurufen, auch wenn die Parameter nachträglich geändert werden.

void j3d_init_compress(j3d_compress_ptr) Initialisiert den Encoder und alle benötigten Module. Die benötigten Parameter sollten an dieser Stelle alle bereits definiert worden sein.

int j3d_c_add_input_file(j3d_compress_ptr, char*) Fügt der Input-Dateiliste eine Datei hinzu. Die Dateien werden vom Input-Modul nach dem FIFO-Prinzip abgearbeitet.

void j3d_start_compress(j3d_compress_ptr) Startet den Kodierungsprozess. Dieser Terminiert erst wenn die gesamte Dateiliste abgearbeitet worden ist.

float j3d_get_compress_ratio(j3d_compress_ptr) Gibt die erreichte Kompressionsrate (in Prozent, relativ zur Anfangsgrösse) an.

Tabelle 4.1: API Kompression

Dekompression

Tabelle 4.2 zeigt eine Auflistung der Befehle, welche für die Dekompression benötigt werden. Die Reihenfolge sollte möglichst beibehalten werden.

4.3.2 Module

Für den Gebrauch der Programmbibliothek als Standard-Codec ist die Kenntnis des Aufbaus der einzelnen Module irrelevant. Sobald aber mit einem Teil

j3d_decompress_ptr j3d_create_decompress() Generiert eine Struktur vom Typ `j3d_decompress_struct` welche alle Informationen bezüglich Format, Tabellen und Modulen beinhaltet. Diese Struktur wird in allen folgenden Funktionsaufrufen benötigt.

void j3d_d_set_std_params(j3d_decompress_ptr) Setzt Standardwerte in alle Parameter der `j3d_decompress_struct`. Es ist empfehlenswert, diese Prozedur aufzurufen, auch wenn die Parameter nachträglich geändert werden.

void j3d_init_decompress(j3d_decompress_ptr) Initialisiert den Decoder und alle benötigten Module. Die benötigten Parameter sollten an dieser Stelle alle bereits definiert worden sein.

void j3d_start_decompress(j3d_decompress_ptr) Startet den Kodierungsprozess. Dieser Terminiert erst wenn der Input-Datenstrom abgearbeitet wurde.

Tabelle 4.2: API Dekompression

der Spezifikation experimentiert werden will (z.B. anderer Kompressionsalgorithmus oder Entropie-Coder), ist es sehr nützlich, etwas mehr über die einzelnen Module zu wissen. In diesem Abschnitt wird gezeigt, wie ein Standardmodul durch ein eigenes ersetzt werden kann.

Um einen näheren Einblick in Aufbau und Struktur der Module zu erhalten siehe Kapitel 5.

Module erstellen

Jedes Modul muss eine gewisse Anzahl von Funktionen bereitstellen, welche in der Datei `j3dint.h` definiert sind. Bereits vorhandene Funktionsdefinitionen sollten in dieser Datei nicht verändert werden, um die korrekte Ausführung der Standardmodule zu gewährleisten. Es ist jedoch durchaus möglich, die Moduldefinitionen um eigene Funktionen zu erweitern.

Anhand eines Beispieles wird nun erläutert, wie ein eigenes Modul erstellt werden kann. Es soll im Folgenden das Modul `cinput.c` (encoder input manager) durch ein eigenes ersetzt werden. Ein Modul hat immer folgenden Aufbau:

```
#define J3D_INTERNALS
#include "j3dlib.h"

...

/* Private subobject */
typedef struct {
    struct j3d_c_input pub; /* public fields */

    /* private variables */

    int my_private_variable;
} my_input;
typedef my_input * my_input_ptr;


/* exported function (see j3dint.h for definition) */
static int my_exported_function
           (j3d_compress_ptr cinfo, j3d_rgb_image out_img) {
    /* enable private variables */
    my_input_ptr input = (my_input_ptr) cinfo->input;

    ...
}

...

void my_input_init_function(j3d_compress_ptr cinfo) {
    my_input_ptr input = (my_input_ptr) malloc(sizeof(my_input));

    ...

    /* assign methods */
    input->pub.read_rgb_image = my_exported_function;

    /* assign module */
    cinfo->input = (struct j3d_c_input *) input;
}
```

Bemerkungen:

- Die exportierte Funktion `my_exported_function` muss genau dieselbe Struktur haben wie jene in `j3dint.h` definierte.
- Es muss dafür gesorgt werden, dass die Funktion `my_input_init_function` global sichtbar ist.

Module verwenden

Ein neu erstelltes Modul ist nun ganz einfach in die bestehende Applikation einzubauen. Im folgenden Beispiel soll das Input-Standardmodul mit dem

vorher erstellten eigenen Modul ersetzt werden:

```
int main(int argc, char **argv)
{
    j3d_compress_ptr cinfo = j3d_create_compress();

    ...

    j3d_init_compress(cinfo);

    my_input_init_function(cinfo);

    ...
}
```

Nach dem Aufruf von `my_input_init_function` zeigt der Pointer `cinfo->input` zum eigenen Modul anstelle des Standardmodules (wie es nach `j3d_init_compress` der Fall war). Insbesondere zeigt der Pointer `cinfo->input->read_rgb_image` nun zur eigenen funktion `my_exported_function`.

Kapitel 5

Programmstruktur

Dieses Kapitel soll einen einführenden Einblick in die Programmstruktur der J3D-Bibliothek gewähren. Weiterführende Informationen können dem gut dokumentierten Quellcode [1] entnommen werden. Das Programm ist sehr stark an die Struktur des IJG JPEG codecs [3] angelehnt, da wie schon im Kapitel 3 erwähnt zahlreiche Teile von J3D und JPEG gleich sind und sich daher auch viele Routinen und insbesondere die Programmstruktur sehr ähnlich sind.

5.1 Module

Der Quellcode ist in sogenannte Module aufgeteilt, welche jeweils für einen spezifischen Teil des Programmablaufes verantwortlich sind. So enthält z.B. das Modul `ccolor` Routinen zur Farbkonversion und das Modul `cinput` die Routinen zum Lesen der Input Dateien.

Die Module können sich gegenseitig aufrufen, d.h. es ergibt sich eine Baumstruktur (siehe Abb. 5.2, 5.4) von aufeinanderfolgenden Modulen. Zeiger auf die einzelnen Module sind in den Structs `j3d_compress_struct` bzw. `j3d_decompress_struct` gespeichert. So kann jede Routine, die eine solche Struct sieht, eine Funktion eines beliebigen Modules ausführen.

Die Funktions- und Variablendefinitionen der einzelnen Module sind in `j3dint.h` zu finden und werden in den nächsten Abschnitten näher erläutert.

5.2 Encoder

Abbildung 5.1 zeigt den Datenfluss des Kompressionsalgorithmus, Abbildung 5.2 zeigt die Baumstruktur (Abhängigkeit) der benutzten Module. Dies sind beides nur vereinfachte Darstellungen, welche dem grundlegenden Verständnis dienen sollen.

Anschliessend folgt eine detaillierte Erklärung der einzelnen Module welche vonm Encoder benötigt werden.

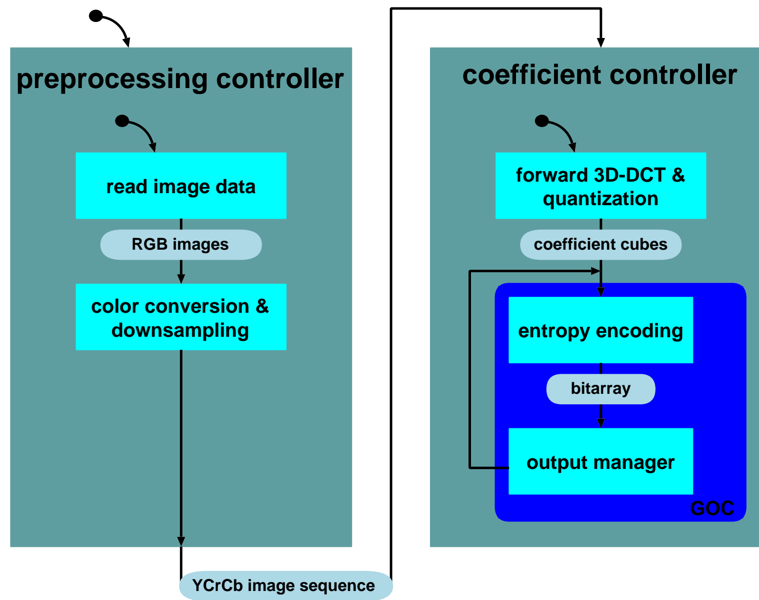

Abbildung 5.1: Datenfluss Encoder

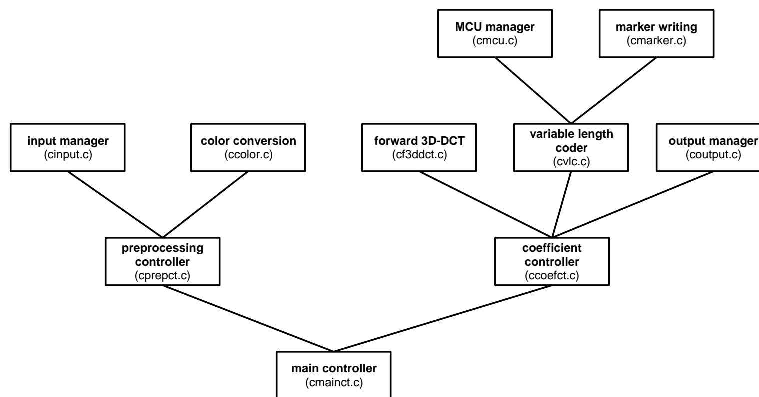

Abbildung 5.2: Baumstruktur Module (Encoder)

5.2.1 Main Controller (`cmainct.c`)

Der *Main Controller* arbeitet auf der Stufe einer *group of sequence* (s. Abb. 3.3, Seite 13), er ist zuständig für die Kompression eines gesamten Videos. Er fügt die SOI sowie EOI Marker in den Datenstrom ein und ruft für jede *sequence* (d.h. 8 Einzelbilder) den *Preprocessing Controller* gefolgt vom *Coefficient Controller* auf.

5.2.2 Preprocessing Controller (`cprepct.c`)

Der *Preprocessing Controller* arbeitet auf der Stufe eines *scans* (s. Abb. 3.3, Seite 13). Er ist zuständig für die Bereitstellung einer Sequenz im korrekten, für den *Coefficient Controller* lesbaren Formats. Er benötigt den *Input Manager* um die Einzelbilder einzulesen und konvertiert sie danach mit dem *Color Converter*. Als Rückgabewert liefert er eine `j3d_image_sequence`.

5.2.3 Input Manager (`cinput.c`)

Die einzige Aufgabe des *Input Managers* ist es ein Einzelbild einzulesen und dieses im RGB Format zurückzugeben. Dies ist normalerweise eine Bilddatei, es ist aber auch denkbar einen *Input Manager* zu schreiben welcher z.B. von einer Webkamera liest.

5.2.4 Color Converter (`ccolor.c`)

Der *Color Converter* erfüllt zwei Aufgaben:

1. Color Conversion: Das Eingangsbild im RGB Format wird ins YCrCb Format umgewandelt.

2. Color Downsampling: Die einzelnen Farbkomponenten werden abhängig vom `downsampling_factor[]` (`j3dlib.h`) komprimiert.

5.2.5 Coefficient Controller (`ccoefct.c`)

Der *Coefficient Controller* arbeitet auf der Stufe eines *scans* (s. Abb. 3.3, Seite 13). Er arbeitet in folgenden Schritten:

1. Transformieren der Eingangssequenz (mit Modul *Forward 3D-DCT*)

2. Scan Header (SOS Marker) schreiben (mit Modul *Marker Writer*)

3. Ganzen *scan* schreiben (mit Modul *Variable Length Coder*)

Das schreiben der Daten erfolgt zunächst in einen Buffer, welcher bei Bedarf vom *Output Manager* in eine Datei geleitet wird.

5.2.6 Forward 3D-DCT (`cf3ddct.c`)

Dieses Modul ist für die Transformation eines *scans* zuständig. Es teilt diesen in Blöcke auf (normalerweise 8x8x8 Pixel), wendet darauf den 3D-DCT Algorithmus [2] an, quantisiert und sortiert diese anschliessend.

Der Rückgabewert ist ein `j3d_dct_coef`, welches vom *Variable Length Coder* weiterverarbeitet werden kann.

5.2.7 Variable Length Coder (`cvlc.c`)

Der *Variable Length Coder* arbeitet auf der Stufe einer *group of cubes*. Er verwendet den *MCU Manager*, welcher ihm die erforderlichen Daten bereitstellt, und generiert mittels eines Huffmann-Encoders einen Datenstrom [4]. Dieser Datenstrom (`j3d_bitarray`) kann der *Output Manager* nun in eine Datei schreiben.

5.2.8 Marker Writer (`cmarker.c`)

Der *Marker Writer* stellt Funktionen für die Ausgabe korrekter Marker-Codes (s. Kap. 3.2) zur Verfügung.

5.2.9 MCU Manager (`cmcu.c`)

Der *MCU Manager* fasst einzelne *cubes* zu einer *MCU* (Minimum Coding Unit) zusammen. Er kennt die aktuelle Position des Encoders innerhalb eines *scans*, und liefert jeweils einzelne *cubes*.

5.2.10 Output Manager (`coutput.c`)

Der *Output Manager* stellt ein Bitarray (für den Datenstrom) zur Verfügung (`j3d_bitarray`) und ist zuständig für die Ausgabe des Datenstromes in eine Datei.

5.3 Decoder

Abbildung 5.3 zeigt den Datenfluss des Dekompressionsalgorithmus, Abbildung 5.4 zeigt die Baumstruktur (Abhängigkeit) der benutzten Module. Dies sind beides nur vereinfachte Darstellungen, welche dem groben Verständnis der Struktur dienen sollen.

Anschliessend folgt eine detaillierte Erklärung der einzelnen Module welche von Decoder benötigt werden.

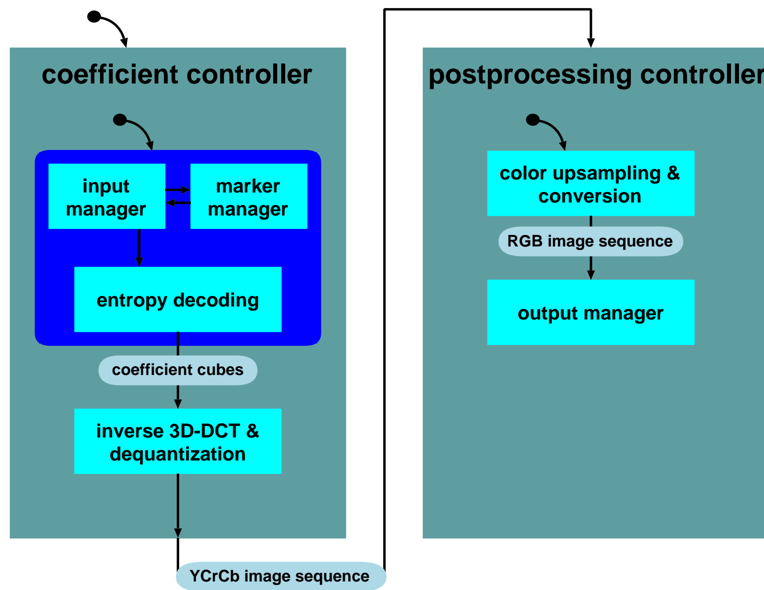

Abbildung 5.3: Datenfluss Decoder

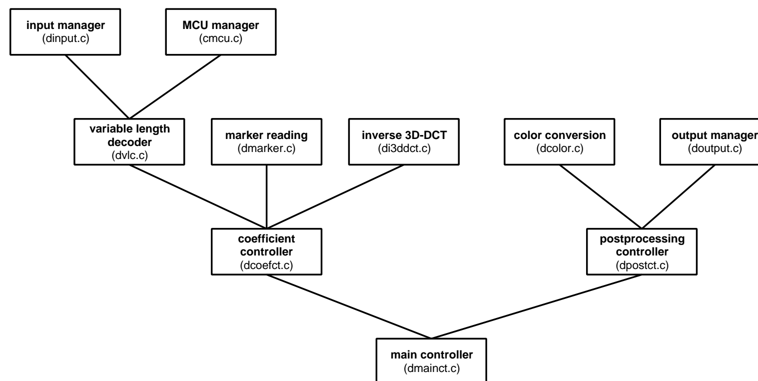

Abbildung 5.4: Baumstruktur Module (Decoder)

5.3.1 Main Controller (`dmainct.c`)

Der *Main Controller* arbeitet auf der Stufe einer *group of sequence* (s. Abb. 3.3, Seite 13), er ist zuständig für die Dekompression eines gesamten Videos. Er liest den SOI Marker und ruft für jede *sequence* (d.h. 8 Einzelbilder) den *Coefficient Controller* gefolgt vom *Postprocessing Controller* auf und wertet unbehandelte Marker aus.

5.3.2 Coefficient Controller (`dcoefct.c`)

Der *Coefficient Controller* arbeitet auf der Stufe eines *scans* (s. Abb. 3.3, Seite 13). Er ruft für jede *group of cubes* den *Variable Length Decoder* auf und behandelt gelesene Marker. Wurde ein *scan* erfolgreich gelesen startet er das Modul *Inverse 3D-DCT*.

5.3.3 Variable Length Decoder (`dvlc.c`)

Der *Variable Length Decoder* arbeitet auf der Stufe einer *group of cubes*. Er verwendet den *Input Manager* um Daten einzulesen, dekodiert sie mittels Huffmann-Tabellen und schreibt die Ausgabe in `coef_cubes`, welche ihm der *MCU Manager* bereitstellt.

5.3.4 Inverse 3D-DCT (`di3ddct.c`)

Dieses Modul ist für die Rücktransformation eines *scans* zuständig. Es teilt diesen in Blöcke auf (normalerweise 8x8x8 Pixel), dequantisiert sie und wendet darauf den i3D-DCT Algorithmus an [2].

Der Rückgabewert ist eine `j3d_image_sequence`, welche vom *Postprocessing Manager* weiterverarbeitet werden kann.

5.3.5 Marker Reader (`dmarker.c`)

Der *Marker Reader* erkennt und bearbeitet Marker (s. Kap. 3.2) aus dem Datenstrom. Er verwirft alle Daten welche sich zwischen der Aktuellen Position im Datenstrom und dem nächsten Marker befinden.

5.3.6 Input Manager (`dinput.c`)

Der *Input Manager* stellt einen Input-Buffer zur Verfügung und ist verantwortlich für das Einlesen der Daten aus der Datenquelle.

5.3.7 MCU Manager (`dmcu.c`)

Der *MCU Manager* kennt die Regeln, wie einzelne *cubes* zu einer *MCU* (Minimum Coding Unit) zusammengefügt sind und somit auch deren Rei-

henfolge im Datenstrom. Er liefert jeweils den nächsten *cube* zurück und meldet wann eine *MCU* oder ein *scan* fertig sind.

5.3.8 Postprocessing Controller (`dpostct.c`)

Der *Postprocessing Controller* arbeitet auf der Stufe eines *scans* (s. Abb. 3.3, Seite 13). Er ruft pro Einzelbild den *Color Converter* auf und schreibt es danach mittels *Output Manager* in eine Datei.

5.3.9 Color Converter (`dcolor.c`)

Der *Color Converter* erfüllt zwei Aufgaben:

1. Color Upsampling: Die einzelnen Farbkomponenten werden abhängig vom `downsampling_factor[]` (`j3dlib.h`) dekomprimiert.

2. Color Conversion: Das Eingangsbild im YCrCb Format wird ins RGB Format umgewandelt.

5.3.10 Output Manager (`doutput.c`)

Der *Output Manager* verwaltet die Output-Dateien und beschreibt diese mit Einzelbilder.

Kapitel 6

Schlussfolgerungen und Ausblick

Es wurde ein leistungsfähiges Programmpaket [1] imlementiert, welches den Anforderungen entsprechend modular aufgebaut und somit eine erweiterbare Plattform für Experimente mit dem 3D-DCT Algorithmus ist. Dieses Paket (Encoder, Decoder und Programmbibliothek) enthält alle nötigen Routinen zur fehlerresistenten Kompression und Dekompression eines Video-Datenstromes basierend auf dem J3D-Format (siehe Kap. 3).

J3D-Programmbibliothek

Die Bibliothek wurde auf Solaris 5.6 sowie Linux (Kernel 2.4) getestet, sollte aber auf jeder UNIX-Plattform ohne weiteres compiliert und ausgeführt werden können. Sie ist sehr effizient, es ist durchaus möglich in Echtzeit eine Sequenz von ca. 300x200 Bildpunkten bei 25Hz zu kodieren oder dekodieren.

Fehlerresistenz

Fehlerresistenz wurde nur auf Funktionalität getestet, d.h. es sind noch keine wirklich weitführende Analysen gemacht worden, etwa wieviel Paketverluste oder Bitfehler der Datenstrom erlaubt.

Literaturverzeichnis

[1] Axel Burri: Freie J3D Programmbibliothek. Internet-Adresse: `http://people.ee.ethz.ch/~aburri/download/j3d/`

[2] Andreas Burg, Roni Keller: A Real-Time Videocompression System Based on the 3D-DCT. Diplomarbeit, IIS-ETH Zürich, 2000.

[3] Independent JPEG Group: Freie JPEG Programmbibliothek. Internet-Adresse: `www.ijg.org`

[4] William B. Pennebaker & Joan L. Mitchell: *JPEG Still Image Data Compression Standard*. published by Van Nostrand Reinhold, 1993, ISBN 0-442-01272-1