Fakultät für Informatik

Professur Rechnernetze
und verteilte Systeme

Diplomarbeit

Videoaufbereitung für Peer-To-Peer-Videostreaming

Autor: Martin Fiedler

Betreuer: Prof. Dr. Uwe Hübner
Dr. Jörg Anders
Dr. Robert Baumgartl

Datum: 31. Mai 2005

Inhaltsverzeichnis

1. Aufgabenstellung

Entwicklung einer Peer-To-Peer-Lösung für Videostreaming (Videoteil)

Für die Übertragung von Live-Videodaten an mehrere Empfänger gibt es als etablierte Technologien die Übertragung per Multicast an mehrere Empfänger sowie die Unicast-Übertragung an jeden Empfänger von einem Verteilpunkt aus. Nachteilig ist im ersten Fall, dass viele ISPs kein Multicast unterstützen, im zweiten Fall der hohe Bandbreitenbedarf am Verteilpunkt.

Die Unterschiede in der verfügbaren Bandbreite von ISP-Zugängen (DSL, ISDN) und Campusnetzwerken (z.B. Studentennetzen) erfordern es außerdem, das Videomaterial in mehreren Qualitätsstufen anzubieten.

Im Team soll ein System entworfen und realisiert werden, das aus einer Quelle effizient verschiedene Qualitätsstufen des Videomaterials erzeugt (»Videoteil«) und dieses mittels Peer-To-Peer-Technologie verteilt (»Netzwerkteil«).

Aufgabenstellung »Videoteil«

- Evaluierung von Open-Source-Implementierungen von Videocodecs nach den Kriterien Geschwindigkeit, Bildqualität und Codequalität
- Entwicklung eines Video-Encoders, der in einem Encodingvorgang zwei verschiedene Qualitätsstufen an Videomaterial erzeugen kann, z.B. eine Qualitätsstufe für DSL-Anschlüsse und eine für breitbandige Netze (z.B. Studentenwohnheime)
- Untersuchung der Faktoren, welche die Latenzzeiten im Encoding- und Decodingprozess bestimmen
- Untersuchung der notwendigen Hardwareressourcen beim Encoder

Die entstandenen Programme sind unter eine Open-Source-Lizenz zu stellen.

2. Einsatzzweck

Beim Verteilen von Videodaten gibt es zwei verschiedene Einsatzszenarien: Liveübertragung und On-Demand-Übertragung. Bei der Liveübertragung werden die Videodaten mit möglichst geringer Verzögerung zum Empfänger übertragen. Alle Empfänger einer Liveübertragung bekommen nahezu gleichzeitig dieselben Daten. Bei On-Demand-Übertragung fordern die Teilnehmer verschiedene Teile eines aufgezeichneten Medienstroms an. Das heißt, daß zwei Teilnehmer der Übertragung nicht zwangsläufig zur gleichen Zeit denselben Inhalt zu sehen bekommen. Bei einer On-Demand-Übertragung besteht auch die Möglichkeit im Medienstrom zu springen („vor- und zurückspulen“).

Der Hintergrund der Aufgabenstellung ist die Verteilung von Vorlesungen. Dabei ist Liveübertragung nützlich, um Interaktion zwischen Vorlesendem und Zuschauern möglich zu machen, zum Beispiel eine Konferenz auf einem Jabber- oder IRC-Server. Denkbar wäre aber auch eine Telefonkonferenz über Voice-over-IP oder ein herkömmliches Telefonnetz. Neben Liveübertraungn ist auch On-Demand Zugriff sinnvoll, zum Beispiel um zur Prüfungsvorbereitung einzelne Vorlesungen noch einmal anzusehen. Dazu müssen aufgenommene Vorlesungen gespeichert werden. Um Interaktivität zu ermöglichen, soll die Zeit zwischen Aufnahme und Wiedergabe (Latenzzeit) bei Liveübertragungen so gering wie möglich sein.

Bei Vorlesungen ist es nützlich, mehrere Videoquellen verwalten zu können, zum Beispiel um mit einer Kamera den Vorlesenden abzubilden, mit einer anderen Kamera Tafelbild oder Präsentationsfolien zu übertragen und bei Diskussionen das Publikum bei Interaktion mit dem Vortragenden zu zeigen.

Neben Vorlesungen sind auch andere Anwendungen möglich, zum Beispiel die Übertragung von Studentenfernsehen, was sich damit preiswerter realisieren läßt als über Funk oder Übertragung in Kabelnetzen.

Die verfügbare Bandbreite im Internet variiert sehr stark. Universitäten sind üblicherweise mit hoher Bandbreite in Forschungsnetzen mit dem Internet verbunden. An die Netze der Universitäten sind oftmals auch Studentennetze angeschlossen, die Studentenwohnheime mit breitbandigem Internetzugang (zur Zeit meist 10 oder 100 MBit/s Ethernet) versorgen. Die breite Masse der Bevölkerung verfügt dagegen üblicherweise über nicht mehr als einen DSL-Anschluß oder sogar nur einen für Videoübertragung ungeeigneten ISDN-Zugang. Das System soll sowohl für Anwender in Studentenwohnheimen als auch Nutzer mit DSL-Anschlüssen geeignet sein. Weiterhin soll das System zumindest auf Endanwenderseite leicht auf verschiedene Plattformen portierbar sein.

3. Überblick

In dieser und der begleitenden Arbeit [1] soll ein konkretes System beschrieben werden, welches die angesprochenen Anforderungen weitestgehend abdeckt. Diese Arbeit beschäftigt sich dabei konkret mit der Aufbereitung der Videodaten und der dazugehörigen Tonspur von der Aufnahme mittels Videokamera bis zur Übergabe an das Streaming-Netzwerk. [1] beschreibt dann die konkrete Auslieferung der Streaming-Daten an die Clients.

3.1. Gesamtsystem

Bei dem in [1] implementierten Netzwerk handelt es sich um ein semidezentrales Peer-To-Peer-Netzwerk. Dabei ist jeder Client (Peer) ein cachender Proxy, der bereits empfangene Teile von Datenströmen speichert und auf Wunsch an andere Peers weitergeben kann. Ein **Tracker**-Server verwaltet Informationen darüber, welche Clients welche Datenstromsegmente vorhalten. Die Clients stehen dabei ständig in Verbindung mit dem Tracker.

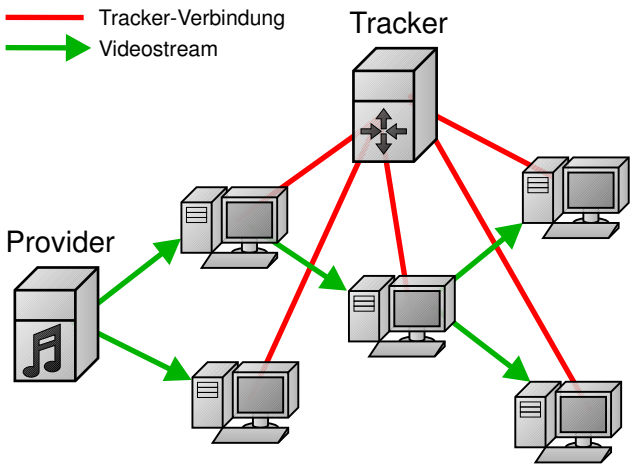


Abbildung 3.1.: Struktur des Peer-To-Peer-Streamingnetzwerks

Will ein Peer einen (Teil-)Strom empfangen, so fragt er zunächst den Tracker, von welchen anderen Peers er die gewünschten Daten beziehen kann. Dieser gibt eine Liste möglicher Kandidaten zurück, anhand welcher der Client entscheiden kann, welchen Peer er nach den konkreten Daten fragt. Empfängt er schließlich Nutzdaten, so informiert er den Tracker wiederum darüber, welche neuen Datenstromteile nun bei ihm im Cache liegen. Außerdem können währenddessen auch andere Peers von dem betrachteten Peer Daten empfangen.

Dieser Ansatz sorgt dafür, daß nur wenige Peers ihre Daten direkt vom **Stream-Provider** beziehen müssen, sondern die Ströme weitestgehend untereinander ausgetauscht werden können. Der Vorteil dieser

Methode liegt darin, daß auch Ströme mit hoher Bitrate zuverlässig an eine große Anzahl von Peers ausgeliefert werden können.

Als Stream-Provider kommen mehrere Möglichkeiten in Betracht: Ein Stream-Provider kann beispielsweise ein Archiv-Server sein, der aufgenommenes Material dauerhaft speichert. Im Fall von Live-Streaming ist der Stream-Provider hingegen ein System, welches in Echtzeit eingespeiste Videoströme dem Netzwerk zur Verfügung stellt. Um diesen Fall soll es in diesem Dokument gehen.

3.2. Struktur des Videoteils

Der Live-Stream-Provider, der Gegenstand dieser Arbeit ist, ist als Software realisiert und setzt sich aus mehreren Komponenten zusammen. Jede Komponente übernimmt dabei einen bestimmten Teil der Datenverarbeitung.

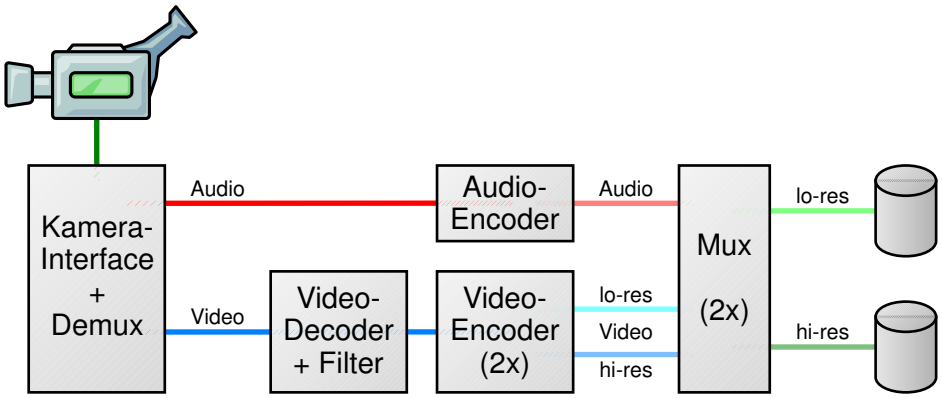


Abbildung 3.2.: Struktur des Streamproviders

- **Kamera-Interface / Demultiplexer**
  Die von einer Videokamera gelieferten Daten werden über eine geeignete Schnittstelle in das Programm übergeben. Dabei werden Bild- und Tondaten voneinander getrennt, um sie unabhängig weiterverarbeiten zu können.

- **Video-Decoder / Video-Filter**
  Die Videodaten werden, sofern notwendig, dekomprimiert und vorgefiltert, um die Qualität zu verbessern und optional ein »Senderlogo« einzublenden.

- **Video-Encoder**
  Die Videodaten werden in ein für Internet-Übertragung geeignetes Format codiert. Dabei werden zwei Codiervorgänge durchgeführt, um gleichzeitig ein hochauflösendes Videosignal für Intranet- oder LAN-Clients und ein niedriger aufgelöstes Signal für Breitband-Internetzugänge zu erzeugen. Dabei wird besonderes Augenmerk darauf gelegt, daß diese doppelte Codierung möglichst effizient erfolgt.

- **Audio-Encoder**
  Die Audiodaten werden parallel zu den Videodaten verarbeitet und ebenfalls in ein kompaktes Format komprimiert.

- **Multiplexer**
  Die entstandenen komprimierten Audio- und Videoströme werden zu kombinierten Datenströmen aus jeweils einem Audio- und einem Videosignal zusammengefaßt und über eine entsprechende Schnittstelle dem Peer-To-Peer-Netz zur Verfügung gestellt.

Die genannten Komponenten werden jeweils in eigenen Kapiteln dieser Arbeit genau beschrieben.

3.3. Implementation des Videoteils

Wie bereits erwähnt, handelt es sich beim hier vorgestellten Streamprovider um eine reine Softwarelösung, die als Userspace-Applikation namens `lvstream` auf einem herkömmlichen Arbeitsplatzrechner läuft.

3.3.1. Plattform

Als grundlegende Hardwareplattform dient dabei ein handelsüblicher PC, der weitestgehend ohne kostspielige Zusatzhardware auskommt. Als Prozessorarchitekturen sind in erster Linie die verbreiteten x86 oder x86_64 angedacht, jedoch ist die Software so gestaltet, daß sie auf andere Architekturen, gegebenenfalls mit leichten Geschwindigkeitseinbußen durch fehlende Optimierungen, portierbar ist.

Als zugrundeliegendes Betriebssystem für das System wird Linux verwendet, da es recht verbreitet ist und vergleichsweise guten Treibersupport für die bei Videoanwendungen benötigte Hardware bietet. Die Treiber verfügen darüber hinaus über leicht beherrschbare Programmierschnittstellen, welche die Entwicklung vereinfachen. Portabilität ist jedoch auch hier in eingeschränkter Form gewährleistet; zumindest auf andere Unix-artige Betriebssysteme ist das Programm recht einfach zu portieren. Gegen Microsoft Windows als Entwicklungsplattform sprechen die wesentlich komplexeren Programmierschnittstellen für Multimedia-Hardware, welche die Entwicklung des in dieser Arbeit beschriebenen Prototyps deutlich erschwert hätten.

3.3.2. Daten- und Kontrollfluß

Die Applikation ist konsequent multithreaded ausgelegt, um die Komplexität zu senken. Dabei laufen Videoaufnahme (Capture), Videoverarbeitung, Audioverarbeitung und ein Kontrollthread parallel zueinander und tauschen sich Daten jeweils über per Mutex synchronisierte Ringpuffer aus.

Im Capture-Thread wird eine Schleife ausgeführt, die sowohl Audio- als auch Videodaten für einen kompletten Frame sammelt. Da der Thread sonst keine Aufgaben erfüllen muß, wartet er dabei die längste Zeit auf eintreffende Daten. Sobald ein Frame komplett ist, werden Audio- und Videodaten voneinander getrennt und in die jeweiligen Ringpuffer eingetragen. Den Audio- und Video-Threads wird daraufhin ein Wakeup-Event geschickt, das sie darüber informiert, daß neue Daten eingetroffen sind.

Jeder Video-Ringpuffereintrag verfügt neben den reinen Frame-Daten über eine laufende Framenummer. Anhand dieser und der Framerate (bei PAL 25 Hz, bei NTSC 29,97 Hz) wird später ein Zeitstempel (**Timestamp**) berechnet. Dieser Zeitstempel ist eine wichtige Größe für den Decoder im Streaming-Client, der dadurch den Anzeigezeitpunkt jedes Frames kennt. Framedrops, die beim Überlauf eines Ringpuffers entstehen, äußern sich dabei durch einen Sprung in der Framenummer und somit auch in den Timestamps; die zeitliche Zuordnung bleibt bei diesem Vorgang intakt.

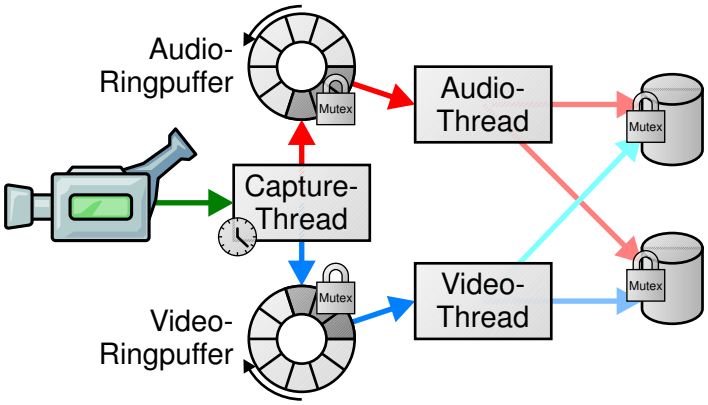


Abbildung 3.3.: Daten- und Kontrollfluß im Streamprovider

Auch die Audio-Daten werden bei der Ausgabe mit Timestamps versehen. Allerdings wird für sie im Ringpuffer keine Framenummer mitgespeichert, da in der aktuellen Implementation noch vorausgesetzt wird, daß keine Audio-Framedrops auftreten. In der Praxis wirkt sich diese Annahme auch in Extremsituationen mit vielen Video-Framedrops nicht nachteilig aus. Der Audio-Thread erhielt in Versuchen genügend Rechenzeit, um seine Aufgaben zu erfüllen.

Die Ringpuffer umfassen jeweils 10 Einträge. Dieser Wert ist so gewählt, daß kurzfristige Verzögerungen, wie sie in einem Multitasking-System ständig entstehen können, recht gut abgefangen werden können. Ein größerer Ringpuffer hingegen würde das Signal in ungünstigen Situationen mehr als nötig verzögern. Weiterhin gilt, daß der Audio-Ringpuffer, wie im vorangegangenen Absatz beschrieben, in kritischen Situationen deutlich weniger ausgelastet ist als der Video-Ringpuffer. Das bedeutet auch, daß die bearbeiteten Audio-Daten im von `lvstream` ausgegebenen Datenstrom zeitlich eher erscheinen als die zugehörigen, um die Ringpufferlatenz verzögerten Videodaten. Bis zu einem gewissen Grad können Zwischenpuffer im Decoder der Stream-Clients solche Diskrepanzen ausgleichen, jedoch sollten die zeitlichen Unterschiede nicht unnötig groß werden.

Die Video- und Audio-Threads gleichen sich im Aufbau: Sie warten zunächst, bis sie durch einen Event vom Capture-Thread auf neu angekommene Daten aufmerksam gemacht werden. Sobald das geschehen ist, entnehmen sie in einer Schleife jeweils einen Frame aus dem Ringpuffer, verarbeiten (also komprimieren) ihn, und geben ihn aus, bis keine Einträge mehr im Ringpuffer vorliegen. Zur Ausgabe verwenden beide Threads dabei gemeinsame, wie die Ringpuffer durch Mutex-Objekte synchronisierte Ausgabekontexte, jeweils einen Kontext für den hochauflösenden und den niedrig aufgelösten Datenstrom.

Die gesamte Architektur ist datenflußgesteuert: Sobald Daten vorliegen, werden die entsprechenden Aktivitäten zur Bearbeitung ausgelöst. Werden keine Daten eingegeben, laufen die Ringpuffer leer und es findet keine Bearbeitung mehr statt. Diese Tatsache, in Verbindung damit, daß die eingehenden Frames einfach durchnumeriert werden, läßt sich damit zusammenfassen, daß der Capture-Thread beziehungsweise die Videoquelle als **Timing-Quelle** für das gesamte System dient. Setzt der Eingabedatenstrom aus, wird dies vom System in seiner derzeitigen Implementation nicht bemerkt – es findet währenddessen einfach keine weitere Bearbeitung statt. Dieses Verhalten hat vor allem für Tests den Vorteil, daß durch die Eingabeframerate der gesamte Konvertierungsprozeß verlangsamt werden kann, um beispielsweise auf langsameren Rechnern Tests durchzuführen.

4. Eingabe

4.1. Möglichkeiten der Videoeingabe

Als Video- und Audioquelle empfehlen sich allein schon aus Kostengründen übliche Videokameras (»Camcorder«). Diese Geräte liefern Videobilder in Fernsehqualität, im in Europa üblichen PAL-Standard also 50 Halbbilder pro Sekunde mit je 288 sichtbaren Zeilen Bildinhalt. Die resultierende Vollbildauflösung von 576 Zeilen ist auch zur Aufnahme von Texten in für Vortragsfolien üblichen Größen ausreichend.

Begleitend zum Bild liefern Camcorder auch ein Tonsignal in einer Qualität, die praktisch nur durch die Güte der eingebauten Mikrofone begrenzt wird. Außerdem verfügen die Geräte über einen Anschluß für ein externes Mikrofon, was beispielsweise beim Übertragen von Vorlesungen nützlich ist, wenn die Kamera weit vom Vorlesenden aufgestellt ist und das integrierte Mikrofon über die große Distanz zu viele Nebengeräusche einfängt.

Bei nahezu allen derzeit erhältlichen Modellen handelt es sich um digitale Camcorder nach der DV- oder MiniDV-Norm, die mit zwei Arten von Schnittstellen ausgeliefert werden: Zum einen verfügen sie über analoge Ausgänge, die ein Stereo-Audio- und Composite-Videosignal über Cinch-Anschlüsse sowie ein S-Video-Signal über einen Mini-DIN-Anschluß ausgeben. Außerdem besitzen alle DV-Digitalcamcorder einen digitalen Ausgang nach dem IEEE-1394-Standard.

Den beiden verfügbaren Schnittstellentypen entsprechend gibt es auch zwei verschiedene Möglichkeiten, die Geräte an den für die Codierung zuständigen PC anzuschließen:

4.1.1. Analoger Anschluß

Es existieren verschiedene Hardwarelösungen, um analoge Audio- und Videosignale auf einem handelsüblichen PC einzulesen. Die einfachste Möglichkeit ist dabei die Verwendung einer normalen TV-Karte oder Grafikkarte mit TV-Eingang für den Videoteil; den Audioteil übernimmt die heutzutage üblicherweise bereits auf dem Mainboard vorhandene Audio-Hardware.

Ein derart realisierter Anschluß der Kamera ist zwar aus Hardwaresicht einfach zu bewerkstelligen, stellt allerdings größere Anforderungen an die Software: Da es sich bei der TV-Karte und dem Soundchip um zwei völlig unabhängige Geräte handelt, die jeweils von eigenen Treibern mit eigener Verwaltung von DMA-Puffern angesprochen werden, ist es keine triviale Aufgabe, Bild und Ton zu synchronisieren. Hinzu kommt, daß Video und Audio auch über voneinander unabhängige Zeitgeber getriggert werden: Die Videokarte liefert Frames in dem Rhythmus, der vom eingehenden Videosignal der Kamera vorgegeben wird, der Audiochip hingegen tastet das anliegende Signal mit einer festen Rate ab. Durch unvermeidliche Gangungenauigkeiten laufen der Video- und Audio-Zeitgeber dabei immer weiter auseinander. Obwohl die Abweichungen deutlich unter 1% liegen, machen sie dennoch eine aufwendige Synchronisation notwendig[1].

[1] Ein Zehntelpromille sind auf 1 Stunde Laufzeit immerhin schon über eine Drittelsekunde Unterschied.

Diese Probleme können umgangen werden, indem höherwertige Hardware zum Einsatz kommt, die Video und Audio gleichzeitig mit präzisem Timing aufnimmt (»Field-Locked Audio«). Solche Capture-Karten sind jedoch im preisgünstigen Consumer-Markt praktisch nicht erhältlich; Ausnahmen bilden dabei einige Karten oder externe USB-Boxen mit integriertem MPEG-2-Encoder. Einige Vertreter dieser Produktgattung können synchrone Audio- und Videodaten liefern, jedoch sind sie verglichen mit einfachen TV-Karten recht teuer. Außerdem liefern sie ein bereits mit einem recht aufwendigen Verfahren vorkomprimiertes Signal, das im Streaming-Rechner zunächst dekomprimiert werden muß, was sehr viel Rechenzeit kostet.

Als weiterer Nachteil gilt die niedrigere Bild- und Tonqualität, die durch die analoge Übertragung erzielt wird. Insbesondere mit Composite-Bildsignalen sind keine befriedigenden Ergebnisse zu erzielen.

4.1.2. Digitaler Anschluß

Als Alternative zu den problembehafteten analogen Übertragungs- und Anschlußvarianten bietet sich die Nutzung des Digitalausgangs moderner Videokameras an. Solche verfügen, wie bereits erwähnt, fast ausschließlich über eine standardisierte IEEE-1394- / FireWire- / i.Link / S400-Schnittstelle[2], die Audio-/Videoströme im ebenfalls standardisierten DV-Format ausgeben kann.

Bei IEEE 1394 handelt es sich um eine schnelle serielle Schnittstelle mit einer Bruttodatenrate von typischerweise 400 MBit/s.
DV, ausgeschrieben »Digital Video«, ist ein Industriestandard für die Aufzeichnung und Übertragung von Video- und zugehörigen Audiodaten. Es existieren mehrere Varianten, die sich hauptsächlich in der verwendeten Datenrate und dem Format der Bandaufzeichnung unterscheiden. Im Consumer-Markt vorherrschend ist dabei das MiniDV-Format mit 25 MBit/s. Die Videodaten liegen dabei in einem schwach verlustbehaftet komprimierten Format vor, das dem JPEG-Verfahren ähnelt.

Um DV-Videodaten in einen PC einzugeben, benötigt dieser ebenfalls ein IEEE-1394-Interface. PCI-Steckkarten mit derartigen Anschlüssen sind mittlerweile preisgünstig zu erhalten, außerdem sind auf Mainboards integrierte IEEE-1394-Interfaces inzwischen recht häufig anzutreffen.

Dadurch, daß die Video- und Audiodaten von der Kamera in einem gemeinsamen, isochronen Datenstrom zum Host-PC übertragen werden, sind Timing-Probleme wie bei der analogen Übertragung ausgeschlossen. Jeder übertragene Videoframe wird von einer nahezu konstanten Anzahl Audio-Samples begleitet. Daher ist eine aufwendige Synchronisation nicht notwendig.

Somit vereint diese Methode der Datenzuführung alle Vorteile in sich: Sie ist kostengünstig, liefert hohe Qualität, da alle Daten ausschließlich digital übertragen und verarbeitet werden, und sie ist auf Softwareseite leicht zu implementieren. Daher wird auch dieser Ansatz in der vorliegenden Implementation verfolgt.

Von Nachteil ist lediglich, daß auch bei dieser Variante ein bereits vorkomprimierter Videostrom zunächst decodiert werden muß, um die eigentlichen Bilddaten zu erhalten und weiterverarbeiten zu können. Doch ist der dafür notwendige Rechenaufwand im Vergleich zu einer MPEG-Decodierung, die bei der Verwendung von analogen Hardware-Encodern als Datenquelle notwendig wäre, deutlich geringer und somit noch akzeptabel. Dies liegt darin begründet, daß MPEG ein vollständiges, aufwendiges Interframe-Videokompressionsverfahren darstellt, wohingegen DV lediglich eine einfache *Intra*frame-Kompression durchführt.

[2] Die vier genannten Begriffe sind in der Tat Synonyme, geprägt von verschiedenen Herstellern, die IEEE-1394 unter jeweils eigenem Namen implementieren.

4.2. Software-Umsetzung

Wie im vorangegangenen Abschnitt erläutert, wird als Datenquelle eine IEEE-1394-Schnittstelle verwendet. Nahezu alle verfügbaren Host-Controller für diese Schnittstelle folgen auf der PC-Plattform einem einheitlichen Programmierinterface, dem Open Host Controller Interface (OHCI). Daher arbeiten die Geräte in der Regel problemlos mit einem gemeinsamen Treiber, unter Linux `ohci1394.ko`, zusammen. Deutlich seltener und meist nur auf Standalone-PCI-Karten sind Hostadapter mit dem PCILynx-Chip von Texas Instruments anzutreffen, für die Linux jedoch ebenfalls einen Treiber anbietet.

Zum eigentlichen Ansprechen der Schnittstelle existieren unter Linux gegenwärtig zwei verschiedene APIs: raw1394 und dv1394 [2]. Beide bestehen jeweils aus einem Kernelmodul und einer Userspace-Library. Die ältere Library video1394 gilt als überholt und wurde daher nicht berücksichtigt.

4.2.1. Capture via libraw1394

Die libraw1394 stellt ihrem Namen entsprechend ein Low-Level-API zur Verfügung, mit dem der Programmierer Pakete senden und empfangen kann. Eine Interpretation der Daten nimmt die Bibliothek dabei nicht vor. Das Empfangen eines DV-Datenstroms geht daher mit recht einfachen Operationen vonstatten: Es wird ein sogenannter isochroner Übertragungskanal zur Kamera geöffnet, von dem daraufhin die einteffenden Datenpakete rechtzeitig abgeholt werden müssen. Dabei stellt die raw1394-Library einen Event-Loop-Mechanismus mit Callbacks bereit.

Mit der raw1394-Bibliothek selbst ist lediglich der Empfang der jeweils 480 Byte großen **DIF**-Blöcke (Digital Interface Format) möglich. Um einen kompletten DV-Frame zu erhalten, der im Fall des europäischen DV-PAL 144 Kilobyte groß ist, müssen jeweils 300 DIF-Blöcke nach einer festen Vorschrift im Frame angeordnet werden[3].

`lvstream` verwendet libraw1394 für die direkte DV-Eingabe. Die kritischen Codeteile zur Frame-Reassemblierung wurden dabei auf Grundlage des Programm `dvgrab` [3] implementiert, einer unter GPL stehenden IEEE-1394-Capture-Anwendung.

Problematisch am libraw1394-Ansatz ist jedoch, daß die zugrundeliegende Bibliothek auf x86_64-Systemen noch nicht vollständig 64-Bit-sicher zu sein scheint: Der Versuch, eine 32-bittige libraw1394 auf einem 64-Bit-Kernel einzusetzen, scheiterte.

4.2.2. Capture via libdv1394

Neben libraw1394 existiert mit libdv1394 eine weitere Bibliothek, die speziell auf DV-Capture und -Ausgabe ausgerichtet ist. So führt sie DV-Frame-Reassembly direkt im Treiber aus, und muß auch vor der Verwendung schon auf NTSC oder PAL eingestellt werden, da sonst das Kernelmodul abstürzt.

dv1394 führt eine Vielzahl neuer Devices ein, von denen direkt DV-Daten gelesen werden können: `/dev/ieee1394/dv/host0/PAL/in` führt z.B. das DV-PAL-Videosignal der Kamera, die am ersten 1394-Port angeschlossen ist. Die Daten können dabei direkt wie aus einer Pipe ausgelesen werden; zudem werden eine Anzahl `ioctl()`-Aufrufe, Polling und memory-mapped Buffer unterstützt.

[3] Beim amerikanischen und japanischen DV-NTSC handelt es sich um 250 Blöcke, die einen Frame von 120 Kilobyte Größe ergeben. Da bei PAL jedoch 25 Frames pro Sekunde und bei NTSC 29,97 Frames pro Sekunde übertragen werden, ist die Datenrate mit 3,6 MByte pro Sekunde bei beiden Standards nahezu gleich.

Praktische Tests mit libdv1394 im Laufe der Entwicklung von `lvstream` schlugen allerdings fehl, und da raw1394-Capture hingegen in Versuchen problemlos funktionierte, wird dv1394-Capture nicht verwendet.

4.2.3. Capture via Pipe

Alternativ zu direktem 1394-Capture kann zur Compilierzeit auch »indirekte« DV-Eingabe über die Standardeingabe aktiviert werden. Diese Lösung ist zwar weniger elegant und effizient als direkte Aufnahme, allerdings ist sie ungleich flexibler:

- libdv1394-Capture ist zumindest in der einfachsten Variante über die Device Files möglich.
- Es können auch völlig andere Capture-Applikationen verwendet werden, z.B. um die 64-Bit-Probleme von libraw1394 zu umgehen. So läuft ein 64-Bit-`dvgrab` hervorragend als Frontend für ein 32-Bit-`lvstream`.
- Es ist möglich, im Vorfeld aufgezeichnete DV-Ströme in das Programm einzuspeisen. Da die Eingabe gleichzeitig als Systemzeitgeber fungiert (vgl. 3.3.2), kann der Datenstrom zudem in beliebiger Geschwindigkeit eingespielt werden, um z.B. auch auf langsameren Systemen Tests in verlangsamter Echtzeit zu fahren, oder das Verhalten bei Framedrops zu simulieren. Für diesen Zweck wurde ein kleines Tool namens `dvcat` entwickelt, welches einen DV-Strom aus einer Datei liest und mit beliebiger Framerate an `lvstream` weitergibt.

5. Video-Preprocessing

Die vom Capture-Teil eingelesenen DV-Videoframes passieren vor der endgültigen Komprimierung einige Filter, die zur weiteren Verarbeitung nötig sind oder die Bildqualität verbessern.

5.1. Dekomprimierung der DV-Frames

Der erste der Preprocessing-Filter ist kein Filter im eigentlichen Sinne, sondern er entpackt die komprimiert vorliegenden DV-Frames in ihr Rohformat, eine in Anhang B beschriebene Variante von ITU-R BT.601 mit 4:2:0-Subsampling. Danach liegen die Daten als unkomprimierte Bilder im Speicher und können problemlos weiterverarbeitet werden.

Im Laufe der Entwicklung wurden zwei verschiedene DV-Decoderbibliotheken auf ihre Verwendbarkeit getestet. Bei den beiden Kandidaten libavcodec [4] und libdv [5] handelt es sich um Open-Source-Bibliotheken unter LGPL-Lizenz. Da beide Decoder in den verbreiteten Media-Player MPlayer [6] integriert sind, konnten stichprobenartige Tests recht einfach durchgeführt werden, ohne eigene Testapplikationen entwickeln zu müssen.

Als Testplattform dienten dabei zwei PCs, auf denen je ein PAL-Clip und ein NTSC-Clip von jeweils ca. 30 Sekunden Länge decodiert wurde. Als Bewertungskriterium wird die relative Decodiergeschwindigkeit, also das Verhältnis der Spielzeit zur Decodierzeit, verwendet. Die MPlayer-Komandozeile lautete:

```
mplayer -benchmark -nosound -vo null -vc <codec> <file>
```

Außer Konkurrenz wurden dabei auch zwei kommerzielle Closed-Source-Decoder von Sony und MainConcept getestet. Alle Tests wurden dreifach wiederholt und der jeweils beste Wert ausgewählt.

Decoder	Parameter	Celeron 2,4 GHz		Celeron 450 MHz	
		PAL-Clip	NTSC-Clip	PAL-Clip	NTSC-Clip
libavcodec	`-vc ffdv`	478%	489%	111%	111%
libdv	`-vc libdv`	219%	231%	63%	67%
MainConcept	`-vc mcdv`	465%	—	122%	—
Sony	`-vc qdv`	397%	404%	126%	121%

Tabelle 5.1.: Ergebnisse des DV-Decoder-Benchmarks

In den Testergebnissen wird deutlich, daß libavcodec, MainConcept und der Sony DV-Decoder in etwa die gleiche Performance liefern. Der MainConcept-Decoder hat jedoch Probleme mit dem NTSC-Testclip; er stürzt mit einer Zugriffsverletzung ab. Weit abgeschlagen landet libdv mit etwa nur der halben Performance bei jedem Test hinten. Daher wird für `lvstream` der ffmpeg/libavcodec-Decoder verwendet, da er eine ordentliche Performance erreicht und die kommerziellen Decoder aus Lizenzgründen ausscheiden.

5.2. Deinterlacing

Ein allgemeines, schwer zu lösendes Problem bei der digitalen Videoverarbeitung stellt das Interlacing dar. Es entsteht dadurch, daß aus hauptsächlich historischen Gründen bei PAL-Video nicht 25 Vollbilder je Sekunde übertragen werden, sondern 50 Halbbilder. Dabei sind die Halbbilder jeweils um eine Bildzeile zueinander versetzt. Betrachtet man ein Vollbild, so bestehen die ungeraden Bildzeilen aus dem ersten Halbbild, die geraden Bildzeilen hingegen stammen aus dem zweiten Halbbild. Da die Halbbilder jedoch zu verschiedenen Zeiten aufgenommen wurden, sind im Vollbild somit zwei zeitlich unterschiedliche Bilder »verwoben«.

Abbildung 5.1.: Interlacing am Beispiel einer sich schnell bewegenden Hand

Aus der Verwebung der beiden Halbbilder ergeben sich einige praktische Probleme. Das bekannteste resultiert daraus, daß Computermonitore, moderne Fernsehgeräte und Flachbildschirme immer Vollbilder ausgeben. Eine direkte Anzeige des im Bildspeicher interlaced vorliegenden Bildes würde die Vermischung der Bilder sichtbar machen und wäre sehr unangenehm (»Kammartefakte«). Daher müssen spätestens bei der Anzeige die verwobenen Halbbilder wieder in Vollbilder überführt werden; wie das genau geschieht, ist eine Angelegenheit der verwendeten Soft- oder Hardware. Es existieren etliche verschiedene Algorithmen, die diesen **Deinterlacing** genannten Prozeß realisieren.

Es ist jedoch wünschenswert, das Deinterlacing schon an einem früheren Punkt in der Verarbeitungskette durchzuführen. Der Hauptgrund dafür liegt darin, daß auch bei der Kompression der Videobilder der Rechenaufwand für interlaced Material deutlich höher liegt und zudem eine geringere Effizienz erreicht wird, da Abhängigkeiten zwischen den Halbbildern bei der Codierung berücksichtigt werden müssen. Wird hingegen der Fakt ignoriert, daß das Ausgangsmaterial interlaced vorliegt und einfach non-interlaced codiert, so äußern sich die Kammartefakte als hochfrequente vertikale Störungen, die sich schlecht komprimieren lassen.

Im konkreten Fall von `lvstream` kommt noch hinzu, daß das Mehrfach-Kompressionsverfahren voraussetzt, daß sich die Bilder in der hohen und niedrigen Auflösung möglichst gleichen. Nun verschwinden jedoch spätestens beim Reduzieren der Auflösung die Interlacing-Artefakte (vgl. 5.4), und es würde ein hochauflösendes interlaced Bild und ein niedrig aufgelöstes interlace-freies Bild zum Encoder geschickt. Beide Bilder weisen jedoch z.T. erhebliche inhaltliche Unterschiede auf.

All diese Faktoren sprechen dafür, Deinterlacing bereits direkt nach dem Decodieren des Videobildes durchzuführen. Problematisch ist jedoch, daß für diesen Schritt nur möglichst wenig Rechenzeit verwendet werden soll, gute Deinterlacing-Algorithmen jedoch inakzeptabel komplex sind. Daher kommt ein Verfahren zum Einsatz, das einen sinnvollen Kompromiß aus Komplexität und Qualität darstellt.

Bei dem verwendeten Verfahren handelt es sich um einen adaptiven Deinterlacer in einfachster Form: Ein Halbbild wird prinzipiell unverändert übernommen und das andere Halbbild je nach dem Grad der Sichtbarkeit von Kammartefakten entweder übernommen oder aus dem ersten Halbbild interpoliert.

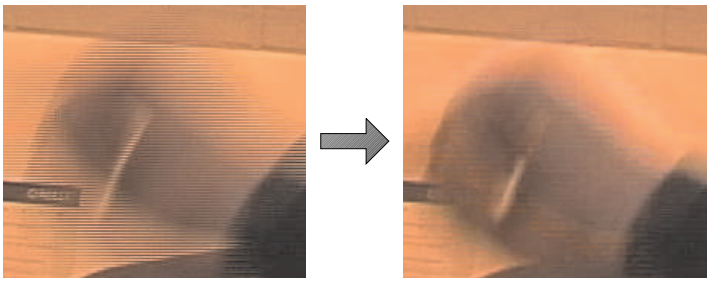


Abbildung 5.2.: Beispiel des Deinterlacers

Der konkrete Algorithmus lautet dabei wie folgt: Sei i_0 der Intensitätswert (Graustufen- oder Farbartwert) in der aktuellen, i_{-1} in der darüberliegenden und i_1 in der darunterliegenden Bildzeile, aber gleichen Bildspalte. Es wird nun ein Wert $a = \frac{1}{2}(i_{-1} + i_1)$ aus den umliegenden Zeilen interpoliert. Ist der Unterschied dieses interpolierten Wertes zum aktuellen Pixelwert, $|i_0 - a|$, größer als ein bestimmter Schwellwert t, so wird angenommen, daß an der betrachteten Bildposition Kammartefakte sichtbar wären, und es wird $i_0 := a$ gesetzt. Dieser Prozeß wird für alle Bildspalten jeder zweiten Bildzeile durchgeführt. Der Wert für t liegt dabei typischerweise bei nur ca. 5% des maximalen Intensitätswertes; seine Auswahl ist ein kritischer Faktor für die Qualität der Resultate.

Diese sehr einfache Heuristik zur Ermittlung der Sichtbarkeit von Kammartefakten liefert eine akzeptable Bildqualität, ist jedoch nicht perfekt: An Stellen mit nur leichter Bewegung werden oft noch leicht sichtbare Artefakte übriggelassen, während harte vertikale Kontraste, die nicht durch Interlacing entstehen, weichgezeichnet werden. Solche Kontraste können beispielsweise bei scharf abgebildeten Schriften entstehen.

Da sowohl die Luminanz- als auch die Chrominanzkomponenten des Bildes interlaced sind, müssen alle drei Komponenten deinterlaced werden. Dies geschieht in der vorliegenden Implementation unabhängig voneinander.

Der Algorithmus verändert die Bilddaten »in-place«. Dabei wird im Programm der Puffer, in dem libavcodec den dekomprimierten Frame ausgegeben hat, direkt verändert. Dies ist ohne Folgen möglich, da es sich bei DV um ein reines Intraframe-Kompressionsverfahren handelt, und der Puffer bei jedem Decodiervorgang von libavcodec lediglich geschrieben, aber nicht gelesen wird. Für Interframe-Verfahren wie MPEG wäre diese Vorgehensweise nicht möglich, da libavcodec die Daten im Puffer noch (unverändert) benötigen würde, um nachfolgende Frames zu decodieren.

5.3. Logo-Einblendung

Keine unbedingt notwendige, aber dennoch nützliche Funktion ist die Möglichkeit, beliebige Logos in das Videobild einzumischen. Auch für dieses Filter ist das Rechenzeit-Budget sehr begrenzt, daher sind auch die Fähigkeiten recht beschränkt. Das verwendete Verfahren hellt einfach durch Addition von Intensitätswerten den Luminanzkanal gezielt auf, die Chrominanzkanäle bleiben unverändert. Dieser sehr einfache Ansatz erzeugt dennoch Ergebnisse, die den einfarbigen Logo-Einblendungen von vielen Fern-

sehanstalten gleichen – die Senderlogos von ARD, ZDF, RTL, Sat.1, ProSieben und etlichen weiteren Programmen werden auf ähnliche Weise erzeugt.

Abbildung 5.3.: Beispiel für eine Logo-Einblendung

Die praktische Handhabung des Logo-Filters sieht folgendermaßen aus: Der Betreiber des Streaming-Netzes erstellt ein Graustufenbild, welches das einzublendende Logo enthält. Schwarze Bereiche in diesem Bild werden dabei »transparent« dargestellt, und je höher die Intensität, desto stärker wird das spätere Videobild aufgehellt. Weiße Bildpunkte im Logo werden auch im ausgegebenen Videobild immer weiß.

Der Name der Logo-Datei wird, gemeinsam mit einem Koordinatenpaar zur Positionierung innerhalb des Videobilds, als Parameter an `lvstream` übergeben (siehe A.2). Auch dynamische Logos sind eingeschränkt möglich, da das Programm auf das USR1-Signal reagiert und daraufhin die Logo-Datei neu lädt.

5.4. Skalierung

Eines der Ziele des in dieser Arbeit vorgestellten Systems besteht in der Erzeugung zweier Ströme verschiedener Auflösung aus einer gemeinsamen Videoquelle. Der hochauflösendere der beiden Videoströme wird dabei nahezu direkt in der Quellauflösung codiert, besitzt also volle PAL-Auflösung. Lediglich die bei der digitalen Abtastung nach ITU-R BT.601 entstehenden Overscan-Pixel (siehe Anhang B) werden verworfen. Aus dem Eingabebild mit 720x576 Pixeln Größe (NTSC: 720x480) entsteht folglich ein komprimiertes Ausgabebild mit 704x576 Pixeln (704x480 für NTSC).

Das Videobild in der geringen Auflösung wird aus diesem hochauflösenden Bild erzeugt. Dabei wird die Auflösung in beiden Koordinatenrichtungen auf genau die Hälfte reduziert, es entsteht also ein Bild mit 352x288 Pixeln Größe (352x240 bei NTSC). Der Faktor 2 ist notwendig, damit die im folgenden Kapitel beschriebene Mehrfach-Codierung funktioniert.

Die konkrete Implementation besteht aus einem einfachen nicht-überlappenden 2x2-Box-Filter mit Rundungsausgleich. Anders ausgedrückt, werden jeweils 4 Pixel (2 nebeneinander, 2 übereinander) zusammengefaßt und der durchschnittliche Intensitätswert berechnet. Da somit auch jeweils zwei benachbarte Zeilen zu einer gemeinsamen vereinigt werden, werden bei diesem Vorgang eventuelle übrig gebliebene Interlacing-Kammartefakte entfernt. Das skalierte Bild ist somit eine gleichmäßige Mischung der beiden originalen Halbbilder.

6. Videocodierung

Momentan sind zwei Lösungsansätze verbreitet, um Videostreaming in mehreren Auflösungsstufen zu ermöglichen: Der naive, aber verbreitetste Ansatz besteht darin, das Videomaterial mehrfach komplett zu komprimieren und anschließend als getrennte Ströme zu den Clients zu übertragen. Dieses Verfahren ist sehr einfach, jedoch benötigt es übermäßig viele Ressourcen: Die rechenaufwendige Videokompression wird mehrfach vollständig durchgeführt, obwohl gleiche Bildinhalte, gegebenenfalls in verschiedenen Auflösungen, am Eingang vorliegen.

Ein anderer Ansatz beruht auf einer Änderung der Kompressionsverfahren an sich, so daß ein Datenstrom entsteht, der in zwei Teile geteilt werden kann: Ein **Base Layer** enthält das Videomaterial in »schlechter« Qualität (also geringer Auflösung), ein **Extension Layer** enthält Informationen, um aus den Base-Layer-Daten ein Video in »besserer« Qualität (mithin höherer Auflösung) zu rekonstruieren. Man spricht in diesem Zusammenhang von »echtem« Scalable Video. An Clients, die lediglich das gering aufgelöste Video sehen wollen, wird dabei nur der Base Layer verteilt. Clients, die das hochauflösende Video sehen wollen, erhalten Base Layer *und* Extension Layer, da beide zur Decodierung notwendig sind.

Diese Methode ist wesentlich ökonomischer, da der Video-Encoder mit einem Durchlauf beide Versionen des Stromes erzeugt, mit einem Rechenzeitaufwand, der üblicherweise nur wenig über dem liegt, der für das hochaufgelöste Video allein anfallen würde.

Die Scalable-Video-Verfahren sind standardisiert: So gibt es in MPEG-2 das **Spatial Scalable Profile** und in MPEG-4 das **Simple Scalable Profile**, die beide die besprochene Art von Scalability realisieren. Das Problem bei diesen Verfahren ist jedoch, daß verbreitete Decoder keine Scalability unterstützen. Lediglich die Referenzimplementationen für die jeweiligen Standards beherrschen sämtliche Scalability-Modi, nur sind diese wiederum nicht für den praktischen Einsatz zur Echtzeit-Decodierung, sondern für Evaluierungs- und Testzwecke gedacht.

Daher wird für die in dieser Arbeit besprochene Implementation ein dritter Weg gewählt: Ein Standard-Encoder wird so erweitert, daß er mehrere vollständige Ströme gleichzeitig erstellen kann, die dann mit gewöhnlichen, verbreiteten Decodern entpackt und dargestellt werden können. Insofern entspricht dieser Ansatz dem zuerst genannten, mit dem Unterschied, daß der Encoder hier gezielt die Tatsache ausnutzt, daß das gleiche Ausgangsmaterial mehrfach ausgegeben werden soll.

6.1. Arbeitsprinzip typischer Videocodecs

Typische Videocodecs wie die der MPEG-Familie arbeiten mit bewegungskompensierter Interframe-Codierung. Dabei wird nur aller 12 bis 300 Videoframes ein komplettes Bild übertragen (**I-Frame, Intra-Frame**). Die dazwischen liegenden Bilder werden aus den jeweils letzten Bildern »vorgergesagt« (**P-Frames, Predicted Frames**) und lediglich der Unterschied zum vorhergehenden Bild codiert. Zusätzlich existieren **B-Frames**, bei denen der jeweils letzte *und folgende* I- oder P-Frame als Referenz verwendet wird. Von diesen **Bidirectional Predicted Frames** werden typischerweise zwischen 0 und 3 Stück jeweils zwischen zwei benachtbaren I- oder P-Frames verwendet.

Die differenzielle Übertragung der Bilddaten ist jedoch allein nicht ausreichend, um bewegte Objekte effizient zu codieren. Für diesen Zweck wird zusätzlich eine **Bewegungskompensation** (**Motion Compensation**) durchgeführt. Dabei wird für jeden **Makroblock** des Bildes (typischerweise ein Bereich von 16x16 Pixeln) bei der Codierung ein **Bewegungsvektor** (**Motion Vector**, **MV**) ermittelt und übertragen (**Motion Estimation**, **Bewegungsabschätzung**). Ein solcher Bewegungsvektor gibt an, welche Stelle im Referenzbild am besten mit dem Inhalt des fraglichen Makroblocks korrespondiert. Es wird dabei keine absolute Bildposition übertragen, sondern nur die Abweichung der ermittelten Stelle von der Position des betrachteten Makroblocks. Der Decoder verwendet dann den Block an der angegebenen Bildposition als Referenz.

Der übrig bleibende Unterschied zwischen dem bewegungskompensierten Referenzbild und dem tatsächlichen Bild wird einer Transformations- und Entropiecodierung unterzogen. Dabei handelt es sich bei nahezu allen bekannten Verfahren um eine **Diskrete Cosinustransformation** (DCT) auf 8x8 Pixel großen Blöcken. Die transformierten Werte werden im Anschluß **quantisiert**. Dieser Schritt ist der einzige tatsächlich verlustbehaftete Vorgang während der ganzen Codierung. Die quantisierten Werte werden mittels Huffman-Codes oder arithmetischer Codierung in den Datenstrom geschrieben. Der Decoder kehrt diesen Prozeß um und erhält das bis auf die Quantisierungsfehler identische Originalbild.

6.1.1. Komplexität des Codierungsvorgangs

Analysen am Laufzeitverhalten üblicher Video-Encoder ergeben, daß im wesentlichen zwei Arbeitsschritte den Laufzeitbedarf für den Codiervorgang ausmachen: Die Bewegungsabschätzung und die DCT-Transformation.

Auf ein drittes sehr laufzeitintensives Element, die Farbraumkonvertierung, soll an dieser Stelle nicht eingegangen werden. Die Konvertierung benötigt zwar ebenfalls viel Rechenzeit, wenn sie in Software durchgeführt wird, jedoch ist sie kein eigentlicher Bestandteil des Codierungsprozesses und wird üblicherweise nur in Hardware bei der Aufnahme und bei der Erzeugung eines Anzeigebildes durchgeführt.

6.1.1.1. Bewegungsabschätzung

Die Bewegungsabschätzung wird im Encoder realisiert, indem der zu codierende (Makro-)Block mit Blöcken des Referenzbildes verglichen wird. Dabei wird ein gewisser, meist rautenförmiger Bereich um den fraglichen Block »abgesucht«, indem für jeden potentiellen Bewegungsvektor die Differenz aus dem bewegungskompensierten Block des Referenzbildes und dem zu codierenden Block gebildet wird. Die Abweichung wird anschließend bewertet und es wird letztlich der Bewegungsvektor für die weitere Codierung verwendet, der am besten bewertet wurde.

Für die Bewertung der Differenzblöcke können verschiedene Funktionen verwendet werden. Die gebräuchlichste, weil schnellste, ist SAD (Sum of Absolute Differences), also die Summe der absoluten Beträge der Differenzwerte im Block. Alternativ kann die Summe der Quadrate der Differenzwerte verwendet werden (SSE, Sum of Squared Errors).

Die Komplexität der Bewegungsabschätzung wird also durch folgende Faktoren bestimmt:

- sie muß für jeden (Makro-)Block des Bildes durchgeführt werden;
- bei B-Frames wird nicht nur gegen ein Referenzbild verglichen, sondern gegen zwei;
- für jeden (Makro-)Block wird eine gewisse Menge von Bewegungsvektoren untersucht;

- für jeden untersuchten Bewegungsvektor wird der zugehörige Block aus dem Referenzbild bestimmt und mit dem gesuchten Block verglichen;
- das Bestimmen des untersuchten Blocks (Motion Compensation) kann seinerseits Interpolation oder Extrapolation aus dem Referenzbildmaterial erfordern, da Bewegungsvektoren auf Bruchteile von Pixeln genau sind und bei einigen Verfahren auch teilweise außerhalb des Bildes zeigen dürfen.

6.1.1.2. Diskrete Cosinustransformation

Bei der 2D-DCT, wie sie in der Bild- und Videokompression verwendet wird, handelt es sich um eine **separierbare** lineare zweidimensionale Transformation auf Blöcken von 8x8 Elementen. »Separierbar« bedeutet dabei, daß die zweidimensionale Transformation als Folge von eindimensionalen Transformationen dargestellt werden, die nacheinander auf die Zeilen und Spalten des Blocks angewendet werden. Jede dieser 1D-DCTs kann mittels einer Transformationsmatrix berechnet werden; es existieren jedoch auch andere Methoden, welche die mathematischen Eigenschaften der DCT zu ihrer schnelleren Berechnung ausnutzen.

Die DCT ist ein Prozeß, der auf reellen Zahlen arbeitet und daher aufwendige Fließkommaarithmetik erfordern würde. Allerdings gibt es inzwischen eine große Zahl hochoptimierter Implementationen, die von den Fixed-Point-Integer- oder Floating-Point-SIMD-Funktionseinheiten moderner Prozessoren Gebrauch machen.

6.2. Ansatz für Mehrfach-Encoding

Die Tatsache, daß die Bewegungsabschätzung den größten Teil der Encoding-Zeit benötigt, bildet den Ansatz für das gleichzeitige Erzeugen mehrerer Auflösungsstufen: Nur für die geringere Auflösung wird die Suche vollständig betrieben. Die derart gewonnenen Bewegungsvektoren stellen, entsprechend skaliert, eine gute Approximation der Bewegungsvektoren in der höheren Auflösung dar. Somit ist lediglich eine Verfeinerung der bereits ermittelten Vektoren notwendig. Diese Methode spart einen erheblichen Teil der Rechenzeit für die hohe Auflösung ein. Die Codierung von I-Frames kann damit zwar nicht beschleunigt werden, die deutlich häufigere Berechnung von P- und B-Frames kann jedoch deutlich profitieren.

Entscheidender Punkt in diesem Verfahren ist dabei die geeignete Skalierung der Bewegungsvektoren: Die Ermittlung der Bewegungsvektoren im Encoder für die geringe Auflösung liefert als Resultat eine Art »Karte« von Bewegungsvektoren (**Motion Map**), in der für jeden Bildblock die geschätzte Bewegungsrichtung verzeichnet ist. Die Größe der Motion-Map-Blöcke muß dabei nicht zwingend mit der Größe der Makroblöcke (16x16 Pixel) identisch sein: MPEG-4 ASP zum Beispiel kennt einen Modus namens Inter4MV, der den vier Luminanzblöcken (jeweils 8x8 Pixel) innerhalb eines Makroblocks eigene Bewegungsvektoren zuordnen kann, und bei H.264 können die Bewegungsvektoren sogar für nur 4x4 Pixel große Bereiche angegeben werden.

Grundlage der Bewegungsvektorskalierung ist nun die Annahme, daß sich jeder Bildbereich in der höheren Auflösung in die selbe Richtung bewegt wie in der geringeren Auflösung. Folglich kann für jeden Block der hohen Auflösung der Bewegungsvektor desjenigen Blocks der niedrigen Auflösung, in dem der betrachtete »hochauflösende« Block liegt, als Approximation verwendet werden. Dabei muß der Betrag des Bewegungsvektors ebenfalls entsprechend vergrößert werden, da die Bewegung im hochaufgelösten Bild entsprechend »schneller« stattfindet: Eine Bewegung von d Pixel pro Frame in der geringeren Auf-

lösung würde beispielsweise bei einem Größenfaktor von 2 in der höheren Auflösung mit einem Betrag von $2d$ erscheinen.

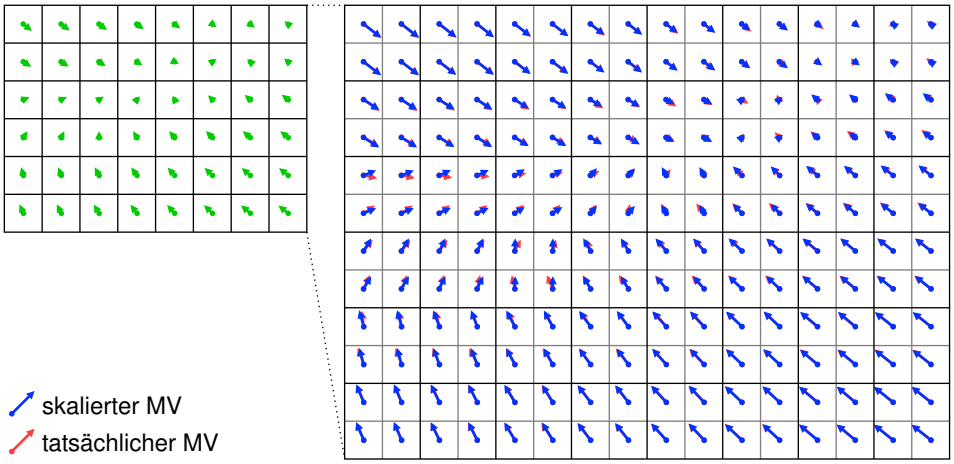


Abbildung 6.1.: Bewegungsvektor-Skalierung

Um deckungsgleiche Vektoren erzeugen zu können, ist es allerdings notwendig, daß die Abmessungen des hochaufgelösten Bildes sowohl in der Breite als auch in der Höhe ein ganzzahliges Vielfaches des niedrig aufgelösten Bildes betragen. Im konkreten Fall von `lvstream` handelt es sich dabei um den Faktor 2, das Videobild ist also in der höheren Auflösung doppelt so breit und hoch wie in der niedrigen Auflösung. Bei nicht-ganzzahligen Vielfachen ließe sich den Motion-Map-Blöcken in der hohen Auflösung nicht mehr eindeutig jeweils ein Block der niedrigen Auflösung zuordnen, und das Kriterium der räumlichen Korrelation würde nicht mehr erfüllt werden. Unter der Annahme, daß sich das Bild einigermaßen gleichmäßig bewegt, die 1. Ortsableitung der Bewegungsvektoren also in jedem Block recht gering ist, ist eine Interpolation zwar sinnvoll, jedoch tritt diese Annahme in der Praxis kaum ein.

Die durch die Skalierung erzeugten Vektoren sind, wie bereits erwähnt, nur Approximationen. Dies hat zwei Gründe: Zum einen wird die Genauigkeit der Vektoren bei der Anpassung der Beträge praktisch reduziert, aus Halbpixel-genauen Vektoren in der geringen Auflösung entstehen also mindestens Vollpixelgenaue Vektoren in der hohen Auflösung. Zum anderen ist die entstehende große Motion Map durch die Ortsauflösung der kleineren Motion Map limitiert: Lokal abgegrenzte Bewegungen, die für das Blockraster der niedrigen Auflösung nicht erfaßbar, in der hohen Auflösung jedoch signifikant sind, werden nicht korrekt wiedergegeben.

Daher muß sich an den Skalierungsprozeß zumindest eine Verfeinerung der skalierten Vektoren anschließen, um die genannten Ungenauigkeiten wieder auszugleichen. Dabei ist hilfreich, daß sich die tätsächliche Bewegungsrichtung der einzelnen Blöcke in fast allen Fällen nur um den Betrag des Genauigkeitsverlustes vom skalierten Wert unterscheidet – eine einfache Verfeinerungsoperation ist ausreichend. Größere Abweichungen sind meist lokal begrenzt und erfordern daher vergleichsweise wenig zusätzlichen Suchaufwand.

6.3. Implementationsgrundlage

Als Grundlage für die in dieser Arbeit vorgestellten Implementation dient der Open-Source-MPEG-4-Encoder XviD [7]. Für diese Entscheidung waren folgende Überlegungen ausschlaggebend:

MPEG-4 ASP

MPEG-4 ist als Basisformat hervorragend geeignet, da das MPEG-4 (Advanced) Simple Profile für PC-Anwendungen der verbreitetste Videokompressionsstandard ist. So existieren unzählige kommerzielle und freie Decoderimplementationen für nahezu alle erdenklichen Plattformen.

Die Effizienz von MPEG-4 (Advanced) Simple Profile ist für heutige Verhältnisse ausreichend. MPEG-2 wurde aufgrund der niedrigeren Effizienz nicht gewählt. Für das deutlich effizientere H.264 existieren bisher leider keine ausgereiften, effizienten Encoder, und Decoder sind noch nicht besonders verbreitet.

Open Source

Der Encoder muß als Open Source vorliegen, da er modifiziert und anschließend wieder als Open Source freigegeben werden muß. Es existieren zwei Kandidaten, die diese Anforderungen erfüllen: Die libavcodec aus dem ffmpeg-Projekt [4], und XviD [7], welches 2001 aus dem OpenDivX-Projekt hervorgegangen ist. ffmpeg steht unter der GNU Lesser General Public License (LGPL), XviD unter der GNU General Public License (GPL).

Geschwindigkeit

Die Geschwindigkeit von XviD und ffmpeg wurde anhand der Codierung einer 33-sekündigen Testsequenz verglichen. Dabei kam MEncoder 1.0-pre7 zum Einsatz. Es wurden Codierungsparameter gewählt, die dem Streaming-Einsatz ähneln. Dies bedeutet vor allem Verzicht auf B-Frames und besonders rechenzeitlastige Operationen wie Rate-Distortion-Optimierung. Es wurde eine konstante Bitrate von 1500 kbps verwendet.

Die MEncoder-Parameter für XviD lauteten:

```
-ovc xvid -xvidencopts bitrate=1500:me_quality=6:chroma_me:vhq=0:
                       max_bframes=0:max_key_interval=50
```

Für ffmpeg/libavcodec kamen folgende Parameter zum Einsatz:

```
-ovc lavc -lavcopts vcodec=mpeg4:vbitrate=1500:v4mv:mbd=1:
                    vmax_b_frames=0:keyint=50
```

Die Messung wurde dreifach wiederholt und der beste (unverfälschteste) Wert ausgewählt. Dabei erreichte libavcodec auf dem Testsystem (AMD Athlon XP 2200+) mit 33,1 fps eine etwa doppelt so hohe Geschwindigkeit wie XviD (16,3 fps). Dies ist umfassenden Optimierungen bei ffmpeg zu verdanken.

Bildqualität

Gleichzeitig mit den Geschwindigkeitsmessungen wurden Qualitätsmessungen durchgeführt. Dabei wurde die in MEncoder eingebaute Möglichkeit verwendet, während des Encodierens PSNR-Messungen (Peak Signal-To-Noise Ratio) durchzuführen.

Die PSNR-Werte berechnen sich wie folgt: Seien $a_{x,y}$ und $b_{x,y}$ die zu vergleichenden Bildkanäle. Alle Intensitätswerte aus diesen Bildern liegen im Bereich von 0 bis $I_{max} = 255$. Es gilt nun

$$\mathrm{MSE}(a,b) = \sum_{y=0}^{y_{max}} \sum_{x=0}^{x_{max}} (a_{x,y} - b_{x,y})^2$$

$$\mathrm{PSNR}(a,b) = 10 \cdot \log_{10} \left(\frac{{I_{max}}^2}{\mathrm{MSE}(a,b)} \right)$$

Dabei ist bezeichnet MSE eine Hilfsfunktion, die mittlere quadratische Abweichung (Mean Squared Error). Bei PSNR handelt es sich streng genommen lediglich eine Umrechnung dieses Wertes in die logarithmische dB-Notation. Der PSNR-Wert kann als Störabstand interpretiert werden: Je höher der Wert, desto größer ist der Abstand des Fehlers zum Signal selbst, desto besser die Qualität.

Da PSNR nur auf jeweils einer Graustufenebene arbeitet, wurden die MSE-Werte für die drei beteiligten Kanäle im Verhältnis Y:Cr:Cb = 70:15:15 gewichtet, um einen gemeinsamen PSNR-Wert zu erhalten. Diese Festlegung ist zwar willkürlich, lehnt sich aber an die tatsächlichen Flächenverhältnisse beim 4:2:0-Subsampling an.

Die Testsequenz, die für die Beurteilung verwendet wurde, besteht in der ersten Hälfte aus heftigen Bewegungen und Schwenks; in der zweiten Hälfte zeigt sie ein im Wesentlichen statisches, aber detailliertes Bild (abgefilmter Inhalt eines TFT-Monitors, auf dem ein Konsolenfenster zu sehen ist).

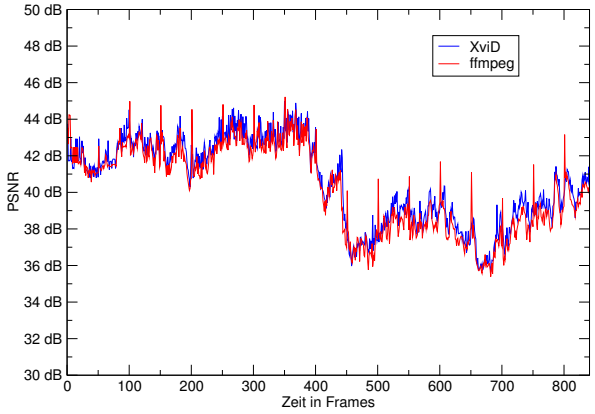


Abbildung 6.2.: PSNR-Verlauf einer Testsequenz für XviD und ffmpeg

In den Meßwerten ist erkennbar, daß XviD generell eine etwas bessere Qualität, also ein höheres Signal-zu-Fehler-Verhältnis liefert. Bei der Codierung von I-Frames bietet ffmpeg/libavcodec die bessere Qualität, was dazu führt, daß der Graph in regelmäßigen Abständen kurze Peaks aufweist. Vergleicht man die aus den Median-Werten der MSE erzeugten PSNR-Werte, so liegt XviD mit 40,87 dB gegenüber libavcodec mit 40,71 dB knapp vorn.

Codequalität / Erweiterbarkeit

Um die notwendigen Veränderungen am Code durchführen zu können, muß dieser gut les- und wartbar sein. In dieser Disziplin fiel die Wahl trotz der erheblich größeren Geschwindigkeit von libavcodec schließlich doch auf XviD, da dieses aus vielen klar strukturierten, gut kommentierten Funktionen besteht. Dies ist vor allem der Tatsache zu verdanken, daß für das gesamte Projekt festgelegte, sinnvolle Coding-Style-Regeln definiert sind, die sehr gut eingehalten wurden.

ffmpeg hingegen ist recht kompromißlos auf Ausführungsgeschwindigkeit optimiert. Daher ist es nicht sehr leicht, Einstieg in den Code zu finden. Insbesondere das für diese Arbeit kritische Modul zur Bewegungsabschätzung hätte erheblich mehr Einarbeitungszeit erfordert, als das bei XviD der Fall war.

6.4. Umsetzung

Um die in 6.2 beschriebene mehrfache Codierung zu realisieren, mußten einige Änderungen an XviD vorgenommen werden. Dabei wurde an mehreren Stellen eingegriffen: Zum einen mußten Möglichkeiten geschaffen werden, um auf die Encoder-eigenen Motion Maps zuzugreifen, und zum anderen mußte dafür gesorgt werden, daß der Encoder für die hohe Auflösung die eingespielte Motion Map auch tatsächlich verwendet. Ein weiteres Problem ergab sich im Laufe der Entwicklung in Hinblick auf die Echtzeittauglichkeit; auch dieses konnte nur durch einen Eingriff in XviD behoben werden.

Bei der Umsetzung wurde auf eine Unterstützung von B-Frames verzichtet, da diese deutlich komplizierter zu handhaben und für Echtzeitanwendungen aufgrund der höheren Encoderverzögerung ohnehin weniger gut geeignet sind.

6.4.1. Ein- und Ausgabe von Motion Maps

Das XviD-API läßt gewöhnlich keinen Zugriff auf die beim Encoding bestimmten Bewegungsvektoren zu. Um diese der Anwendung direkt verfügbar zu machen, wären Änderungen in wichtigen internen Datenstrukturen mit weitreichenden Konsequenzen notwendig gewesen. Doch solche Einschnitte waren gar nicht notwendig, da XviD über eine art Plugin-System verfügt. Solche Plugins werden über einen Callback-Mechanismus an mehreren Stellen während der Codierung jedes Frames aufgerufen und erhalten Zugriff auf einige interne Datenstrukturen des Encoders. Das gesamte Rate-Control-System von XviD ist beispielsweise über Plugins realisiert.

Jedes Plugin verfügt dabei über einige Flags, die Auskunft darüber geben, welche Daten es benötigt oder verändern will. Nur die angegebenen Daten werden dem Plugin vor dem Aufruf zur Verfügung gestellt oder nach dem Aufruf wieder in interne Encoder-Strukturen eingetragen.

Das Plugin-System wurde nun so erweitert, daß als zusätzliches Datenobjekt auch Bewegungsvektorkarten ausgetauscht werden können. Dabei überträgt das modifizierte XviD die Bewegungsvektoren, die es in einer internen Struktur makroblockweise organisiert hat, pluginseitig in ein einfaches zweidimensionales Array aus Bewegungsvektorkoordinaten, beziehungsweise kopiert die Vektoren aus dem Array wieder zurück in die eigenen Datenstrukturen. Dem Inter4MV-Modus von MPEG-4 ASP Rechnung tragend, entsprechen die Blöcke der Motion Map für das Plugin jeweils 8x8 Bildpunkten. Im konkreten Fall von `lvstream` mit dem festen Vergrößerungsfaktor 2 entsteht also nach der Skalierung ein Bewegungsvektor je Makroblock (16x16 Pixel).

6.4.2. Verfeinerung der Bewegungsvektoren

XviD selbst sieht in der Grundversion nicht vor, daß Bewegungsvektor-Approximationen extern vorgegeben werden können, sondern berechnet für jeden Frame komplett neue Vektoren. Um die gewünschte Funktionalität anzubieten, wurden neue sogenannte »Motion Flags« zur Steuerung des Bewegungsabschätzungsalgorithmus definiert. Diese Flags regeln üblicherweise Parameter wie Form des Suchmusters, Genauigkeit der Suche, Verwendung von Inter4MV usw.

Eines der neuen Flags, `XVID_ME_HINTED`, sorgt dafür, daß die Suche nach Bewegungsvektoren nicht beim Nullvektor beginnt, sondern an der vorher mittels Plugin zugeführten Position. Das Flag `XVID_ME_REFINE_ONLY` sorgt in Verbindung mit dem Hinted-Flag dafür, daß keine vollständige Bewegungsvektorsuche durchgeführt wird: Es werden alle Schritte der eigentlichen Suche übersprungen, lediglich die Verfeinerungsoperationen am Ende werden noch durchgeführt.

6.4.3. Ablauf des Encodingprozesses

Insgesamt läuft der Encodingprozeß schrittweise wie folgt ab:

- Das gefilterte, hochauflösende Videobild dient als Eingabe.
- Das letzte Filter, der Scaler, rechnet das Videobild auf die Hälfte seiner ursprünglichen Auflösung herunter (siehe 5.4).
- Dieses reduzierte Bild wird mit einem gewöhnlichen Encoder zu einem MPEG-4-Elementardatenstrom codiert.
- Ein in diesen Encoder eingebrachtes Plugin liest dabei die ermittelte Motion Map aus.
- Die Motion Map wird anschließend nach dem Verfahren aus 6.2 skaliert.
- Ein zweiter Encoder komprimiert das Bild in der hohen Auflösung. Dabei sorgt ein Plugin dafür, daß die zuvor ermittelten und skalierten Bewegungsvektoren als Referenz in den Encoder eingespeist werden. Entsprechende Motion Flags sorgen dafür, daß der Encoder die vorgegebenen Vektoren lediglich verfeinert.

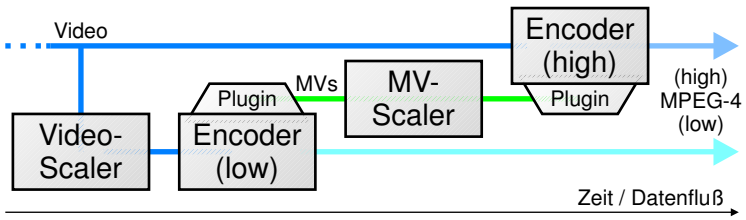


Abbildung 6.3.: Arbeitsweise des Mehrfach-Encoders

Beim vorliegenden Verfahren sinkt der notwendige Zeitverbrauch für die Motion Estimation in der höheren Auflösung um den Faktor 12 bis 20. Dies bedeutet, daß die Motion Estimation für die hohe Auflösung sogar 3- bis 5-mal *weniger* Operationen benötigt als für die niedrige Auflösung. Dabei sind praktisch keine Einbußen in der Bildqualität zu erkennen (siehe auch 9.2).

6.4.4. Behandlung von Framedrops

Als weiche Echtzeitanwendung unter einem Multitasking-System muß `lvstream` tolerant gegenüber kurzzeitigen Störungen sein. Wie in 3.3.2 bereits erwähnt, wird dies unter anderem durch Ringpuffer erreicht. Sollten dennoch Frames verlorengehen, so ist ferner sichergestellt, daß die Zeitstempel der nachfolgenden Frames korrekt sind. Diese Maßnahme allein ist jedoch nicht ausreichend. Tests mit dem verbreiteten Media-Player MPlayer [6] ergaben, daß eine bloße Anpassung der Timestamps zwar dafür sorgt, daß sich Video und Audio nach einigen Sekunden wieder synchronisieren – dennoch bleibt das Bild nicht wie erwartet während des Framedrops stehen, sondern es wird einfach der folgende Frame abgespielt. Die Timestamps dienen MPlayer also im Wesentlichen zur Synchronisation; die absolute Wiedergabezeit wird hingegen maßgeblich durch den Videostrom definiert[1]. Das unvermittelte Fehlen eines oder mehrerer Video-Frames erzeugt also praktisch einen ungültigen Datenstrom.

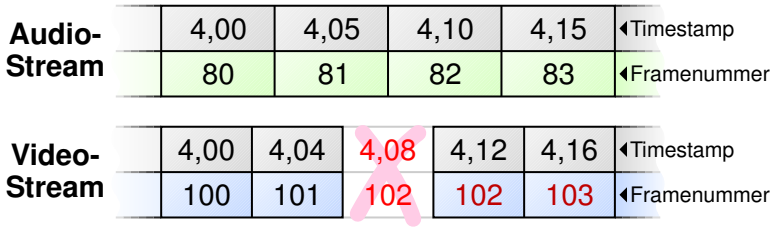


Abbildung 6.4.: Auswirkungen eines Framedrops

Um dieses Problem zu lösen, müssen die fehlenden Frames auch im reinen Videostrom erkennbar sein. Dies wird durch das Einfügen von »leeren« Frames in den Datenstrom realisiert. MPEG-4 unterstützt solche Null-Frames in der Form von P-Frames mit einem Flag namens »not coded« im Header. Da im Datenstrom selbst jedoch Framenummern enthalten sind, können solche Leerframes nicht einfach aus einem statischen Puffer in den Datenstrom eingefügt werden, sondern müssen direkt von XviD erzeugt werden. XviD unterstützt dies jedoch nicht explizit. Also wurde die Möglichkeit, »not coded«-Frames in den Datenstrom einzufügen, nachgerüstet: Erhält der Encoder den Auftrag, einen P-Frame zu codieren, ohne daß dabei Bilddaten angeliefert werden, so erzeugt er automatisch einen leeren Frame. Da bei diesem Prozeß kein Aufwand für Bildverarbeitung getrieben werden muß, ist der Zeitbedarf für die Herstellung des Frames praktisch gleich Null.

Trifft der Decoder auf einen solchen Frame, so wiederholt er einfach das letzte Bild, was exakt dem gewünschten Effekt entspricht.

[1] MPlayer kennt auch einen Modus, in dem das Timing komplett von der Tonspur gesteuert wird; dieser Modus würde das vorliegende Problem jedoch nur umgehen, aber nicht lösen.

7. Audioverarbeitung

Die Verarbeitung der Audiodaten läuft parallel zur Videoverarbeitung in einem eigenen Thread ab. Auch die Audiodaten werden aus einem Ringpuffer gelesen, in dem jeder Eintrag die PCM-Audiosamples für jeweils einen Videoframe enthält. Im Laufe der weiteren Verarbeitung wird zunächst die Anzahl der Samples an den erwarteten Wert angepaßt, um eventuelle Abweichungen zu korrigieren. Danach findet (optional) ein einfaches Downsampling statt, bevor die Audiodaten schließlich verlustbehaftet komprimiert und nach einer Zwischenpufferung dem Multiplexer zugeführt werden.

7.1. Anpassung der Audiosamples

Da die gesamte Anwendung Videoframe-gesteuert arbeitet, ordnen sich auch die Audiodaten diesem Schema unter. Dies bedeutet, daß alle Soundsamples in Blöcken von jeweils einer Videoframelänge bearbeitet werden. Bei dem Demultiplexen der Audiodaten aus den DV-Frames entstehen automatisch Blöcke der gewünschten Größe – für PAL enthält demnach jeder Frame 1920 Samples:

$$\frac{48000\ \frac{Samples}{Sekunde}}{25\ \frac{Frames}{Sekunde}} = 1920\ \frac{Samples}{Frame}$$

Dieser Wert ist jedoch theoretisch; tatsächlich finden sich gelegentlich in den eintreffenden Audio-Paketen mehr oder weniger Samples. Diese Abweichung beträgt oft nur wenige Samples: Die zum Test verwendete Kamera, eine Panasonic NV-DA 1, liefert beispielsweise ca. aller 20 Frames 4 Audio-Samples zuviel. Dennoch sind solche Abweichungen inakzeptabel, da sie über längere Zeit zu einer Audio-/Video-Desynchronisierung führen können: Bei der Testkamera läuft die Audioabtastung ca. 0,00083% zu schnell, was schon nach einer Stunde eine Abweichung von 30 ms, also fast einer Videoframelänge, ausmachen würde.

Daher ist eine Anpassung notwendig, welche die eintreffenden Audio-Pakete stets auf die gewünschte Länge bringt. Dazu kommt ein einfacher Point-Sampling-Algorithmus zum Einsatz, der überschüssige Samples gleichmäßig über den Frame verteilt entfernt, beziehungsweise gleichmäßig verteilt fehlende Samples wiederholt. Dieses Verfahren ist sehr einfach und schnell zu implementieren, liefert jedoch im Regelfall eine äußerst schlechte Klangqualität. Da an dieser Stelle jedoch nur eine minimale Veränderung von ca. 1:500 durchgeführt wird, sind die entstehenden Artefakte unhörbar.

Deutlichere Artefakte entstehen dennoch, wenn `lvstream` mit falschen Audiodaten gespeist wird: DV-Kameras können Ton entweder in 48 kHz Samplerate bei 16 Bit Auflösung oder in 32 kHz Samplerate bei 12 Bit Auflösung liefern. Ist an der Kamera versehentlich der ungebräuchliche 32-kHz/12-Bit-Modus eingestellt, so treffen statt den erwarteten 1920 Samples lediglich 1280 Samples pro Frame ein. Die Audioframe-Anpassung verarbeitet solche Signale klaglos; jedoch wird dann eine Veränderung der Sampleanzahl im Verhältnis 3:2 durchgeführt, was sich in deutlich hörbaren Artefakten niederschlägt. Um solche Signale in akzeptabler Qualität verarbeiten zu können, wären bei der vorliegenden Implementa-

tion weitere Anpassungen notwendig. Da allerdings jede DV-Kamera per Standard den 48-kHz-Modus anbieten muß, ist eine solche Erweiterung nicht unbedingt notwendig.

7.2. Resampling

Bevor die eigentliche Codierung stattfindet, werden die Sampledaten für jeden Frame noch einem Resampling-Schritt unterzogen. Resampling ist sinnvoll, da die Tondaten in einer Qualität vorliegen, die deutlich höher ist als die tatsächlich benötigte: Es werden 48 kHz, Stereo geliefert, obwohl für den Anwendungsfall des Streamings von Vorträgen oder Vorlesungen 24 kHz und Mono völlig ausreichend sind. Diese Beschränkung aufs Wesentliche schlägt sich auch positiv in der benötigten Audio-Bitrate nieder.

Obwohl der nachfolgende Audio-Encoder die Konvertierung selbst übernehmen könnte, wurde in `lvstream` eine eigene Resampling-Routine implementiert. Der Grund dafür ist in erster Linie die erzielte Geschwindigkeit: Der Encoder verwendet eine aufwendige Polyphase-Filterung, für den vorliegenden Anwendungszweck ist jedoch ein einfaches lineares Filter ausreichend. Daher werden die Samples vor der Eingabe in den Encoder einfach zu Gruppen zusammengefaßt: Für den oben beschriebenen Anwendungsfall »48 kHz Stereo auf 24 kHz Mono« wird aus jeweils 4 Samplewerten (2 Samples à 2 Kanäle) der Mittelwert gebildet und als Samplewert abgespeichert. Somit findet gleichzeitig eine Reduktion der Samplerate um einen ganzzahligen Faktor und ein Downmix der beiden Stereo-Kanäle auf Mono statt.

7.3. Audiocodierung

Selbst nach der im vorigen Abschnitt beschriebenen Reduktion der Samplingrate und Kanalanzahl liegen die Audiodaten immer noch mit einer Datenrate von 48 Kilobyte pro Sekunde vor, was für Streaming noch nicht geeignet ist. Eine weitere, verlustbehaftete Reduktion im Verhältnis von etwa 10:1 ist daher notwendig.

Das verwendete Kompressionsverfahren für die Audiodaten muß dabei folgenden Anforderungen genügen:

- Es muß die gewünschte Kompressionsrate von ca. 10:1 bei akzeptabler Klangqualität erreichen.
- Es sollte eine hinreichende Verbreitung aufweisen, um nicht auf eine oder wenige Client-Applikationen eingeschränkt zu sein.
- Es muß eine Open-Source-Encoderimplementation vorliegen, die in diesem Projekt verwendet werden kann.
- Es muß ein gut programmierbares API für den Encoder existieren.
- Der Encoder darf nur wenig Rechenzeit verwenden.

Folgende Kandidaten wurden dabei im Vorfeld der Implementation untersucht:

- **Ogg Vorbis**
  Vorbis ist ein hochqualitativer Audiocodec, der sowohl im Design als auch in der Implementation konsequent auf Open Source setzt. Jedoch ist der Ressourcenverbrauch im Encoder sehr hoch; zudem ist Ogg Vorbis noch nicht auf allen Plattformen weit verbreitet.

- **Windows Media Audio (WMA)**
  WMA ist lediglich auf der Windows-Plattform verbreitet; als Open Source existieren lediglich Implementationen von Decodern.

- **MPEG Audio Layer 3 (MP3)**
  MP3 ist mit deutlichem Abstand das verbreitetste Audio-Kompressionsformat im PC-Umfeld. Auf praktisch jeder Plattform kann MP3 abgespielt werden. Zudem existiert mit LAME [8] ein ausgereifter und schneller Encoder, der in Form der Bibliothek libmp3lame einfach in eigene Programme zu integrieren ist. MP3 ist allerdings das älteste und ineffizienteste Kompressionsformat; eine höhere Kompressionsrate als 10:1 ist nur mit deutlichen klanglichen Einbußen möglich. Positiv hingegen ist, daß MP3 dadurch auch die wenigsten Ressourcen verbraucht.

- **MPEG Advanced Audio Coding (AAC)**
  AAC als Nachfolger der MPEG-Audio-Familie ist momentan noch nicht sehr weit verbreitet. Mit FAAC existiert zwar eine Open-Source-Encoderimplementation, jedoch hat diese noch nicht den selben Reifegrad wie LAME erreicht. Zudem ist die Codierung ebenfalls deutlich langsamer als bei MP3.

In Anbetracht der genannten Faktoren fiel die Wahl schließlich auf MP3. `lvstream` verwendet typischerweise eine Codierung mit 40 Kilobit pro Sekunde, was bei den genannten Standardwerten für die Eingangsdaten einer Kompressionsrate von 9,6:1 entspricht. Diese Datenrate ist ein guter Kompromiß aus Datenmenge und Tonqualität. Da der so erzeugte Audio-Bitstrom eine hinreichend geringe Datenrate für das Streaming zu Breitbandanschlüssen aufweist, andererseits aber die Tonqualität noch so gut ist, daß auch in Hochgeschwindigkeitsnetzen eine höhere Datenrate nicht zwingend erforderlich ist, wird der so erzeugte Datenstrom für beide Video-Bitströme gleichzeitig verwendet.

7.4. Audio-Pufferung und Frame-Zusammensetzung

Ein spezifisches Problem entsteht bei der Codierung der Audiodaten nach MP3 und einigen anderen Formaten in der Praxis: MP3 teilt zwar seinen Bitstrom in diskrete Frames auf, jedoch können die tatsächlichen Daten für einen Frame auch in benachbarte Frames hineinreichen. Folgerichtig liefert der Encoder auf der Ausgabeseite nicht immer vollständige Frames aus, sondern nur so viele Daten, wie tatsächlich im letzen codierten Audio-Frame vorliegen. Für die Speicherung und Übertragung in einem Containerformat müssen die Frames jedoch vollständig vorliegen. Dies macht eine kurze Zwischenpufferung der Audio-Daten notwendig.

7.4.1. Problemstellung

MP3 verfügt, wie viele andere Audio-Kompressionsverfahren auch, über eine Technik namens **Bit-Reservoir**: Die Daten eines Frames können noch vor dem Ende des Payload-Bereiches enden; somit können die Daten des folgenden Frames noch *vor* dessen Header beginnen. Diese Technik macht es möglich, daß der Encoder bei einfach zu codierenden Frames Bits aufsparen kann, um diese dann in komplexere Frames zu investieren.

Gibt man Sampledaten in die LAME-Library ein, so werden diese zunächst in Frames zu jeweils 1152 oder 576 PCM-Samples, je nach verwendeter Samplerate, unterteilt. Samples am Ende des Eingabepuffers, die keinen vollständigen Frame mehr ergeben, werden zwischengespeichert, bis der nächste LAME-Aufruf stattfindet. Jeder Frame wird dann individuell codiert und in den Ausgabepuffer geschrieben.

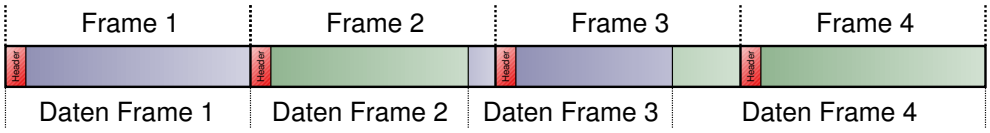


Abbildung 7.1.: MP3-Framing mit Bit Reservoir

Dabei endet die Ausgabe mit dem letzten Byte, das vom letzten vollständig eingegebenen Frame tatsächlich *verwendet* wird – der Frame muß dabei nicht zwingend vollständig ausgegeben werden. Vielmehr können noch Daten im Frame folgen, die jedoch erst mit der Ausgabe des nächsten Frames tatsächlich ausgegeben werden können.

7.4.2. Umsetzung der Lösung

Um das Framing-Problem zu lösen, verfügt `lvstream` über einen weiteren kleinen Zwischenpuffer, der die von LAME ausgegebenen, potentiell unvollständigen Frames aufnimmt und komplette Frames zur Weiterverarbeitung ausgibt. Treffen neue Daten ein, werden anschließend so lange Daten wieder ausgegeben, bis entweder gar keiner oder ein unvollständiger Frame am Ende übrig ist. Dazu inspiziert der Algorithmus die Header der MP3-Frames, um deren Länge zu berechnen. So lange noch mindestens so viele Daten im Puffer sind, wie ein Frame lang ist, kann dieser aus dem Puffer ausgegeben werden.

Die Größe des Puffers ist dabei so bemessen, daß mehr Frames hineinpassen, als bei der Codierung der Audiodaten für einen Videoframe entstehen. Da die Codierung immer in Blöcken stattfindet, die einem Videoframe entsprechen, und nach jedem Codiervorgang alle Frames bis auf den unvollständigen letzten wieder entfernt werden, ist das ausreichend.

7.4.3. Generierung von Timestamps

Wie in 3.3.2 beschrieben, werden den Audio- und Videodaten Timestamps zugeordnet, damit der Decoder am anderen Ende des Systems Bild und Ton zeitlich korrelieren und synchron abspielen kann. Während sich für die Videoframes der Timestamp einfach aus der Formel $Framenummer \div Framerate$ ergibt, ist bei den Audioframes zu beachten, daß diese eine völlig andere Länge haben als die Videoframes: In der typischen `lvstream`-Konfiguration (PAL, 24 kHz, 40 kpbs) dauert ein Videoframe 40 ms, ein Audioframe jedoch 24 ms. Folglich entstehen für jeden Videoframe abwechselnd 1, 2 und 2 Audioframes.

Der naive Ansatz zum Timestamping besteht darin, den Audioframes, die parallel zu jedem Videoframe entstehen, einfach den Video-Timestamp zuzuordnen. Dadurch weichen die Audio-Timestamps zwar fast immer ein wenig von dem tatsächlichen Abspielzeitpunkt ab, typische Mediaplayer tolerieren diese Abweichungen (**Jitter**) jedoch bis zu einem gewissen Maß.

Dennoch ist es wünschenswert, den Audio-Frames korrekte Timestamps zuzuweisen. Dies stellt in der Praxis kein Problem dar, da die Framedauer aus Samplerate und der Anzahl der Samples pro Frame im Vorfeld schon bekannt ist. Somit kann ähnlich wie bei bei den Videoströmen eine »Framerate« berechnet werden, und eine korrekte zeitliche Markierung aller Frames ist möglich. In `lvstream` wird daher diese Methode zur Generierung der Audio-Timestamps eingesetzt.

Video	Timestamp	**Audio**	Timestamp	**Jitter**
Frame 0	0 ms	Frame 0	0 ms	$\pm$0 ms
Frame 1	40 ms	Frame 1	24 ms	$-$16 ms
		Frame 2	48 ms	$+$8 ms
Frame 2	80 ms	Frame 3	72 ms	$-$8 ms
		Frame 4	96 ms	$+$16 ms
Frame 3	120 ms	Frame 5	120 ms	$\pm$0 ms
Frame 4	160 ms	Frame 6	144 ms	$-$16 ms
		Frame 7	168 ms	$+$8 ms
Frame 5	200 ms	Frame 8	192 ms	$-$8 ms
		Frame 9	216 ms	$+$16 ms
Frame 6	240 ms	Frame 10	240 ms	$\pm$0 ms

$\vdots$

Tabelle 7.1.: Jitter bei naivem Audio-Timestamping

8. Multiplex und Ausgabe

Nach der Bearbeitung der Audio- und Videodaten liegen im System drei Datenströme vor:

- Ein MPEG-4-Videostrom mit hoher Bildauflösung.
- Ein MPEG-4-Videostrom mit niedriger Bildauflösung.
- Ein gemeinsamer MP3-Audiostrom.

Um jeweils ein Paar aus Bild- und Toninformationen problemlos gleichzeitig und zueinander zeitsynchron ausliefern und transportieren zu können, werden sie in einem gemeinsamen Datenstrom zusammengeführt (**gemultiplext**). Somit entstehen zwei Multiplex-Datenströme aus jeweils einem Video- und einem Audiostrom. Diese werden anschließend in Zugriffseinheiten (**Access Units**) unterteilt und dem Peer-To-Peer-Netzwerk zur Verfügung gestellt.

8.1. Auswahl eines Containerformats

Die Notwendigkeit des Multiplexens ergibt sich zum einen dadurch, daß ein einzelner Datenstrom leichter (im Sinne von »weniger Protokollaufwand«) zu transportieren ist als getrennte Datenströme. Zum anderen gewährleistet erst der gemeinsame Transport, daß Video- und Audiodaten zueinander synchron abgespielt werden können. Dies wäre bis zu einem gewissen Grad zwar auch mit zwei getrennten Strömen (**Elementarströmen**) möglich, jedoch spätestens nach einer Suchoperation (**Seek**) im Datenstrom können die Daten nicht mehr zeitlich korreliert werden, da Video- und Audio-Frames eine unterschiedliche Rate aufweisen und nach einem Sprung aus den Elementarstromdaten normalerweise nicht erkannt werden kann, wie viele Frames übersprungen wurden.

Für die Verwendung im Streamingsystem spielt außerdem eine weitere Eigenschaft eine wichtige Rolle: Will ein Client den Datenstrom ab einer bestimmten Stelle empfangen, so muß es möglich sein, ihm einen Teildatenstrom zu liefern, der zwar alle für eine Decodierung notwenigen Teile enthält, jedoch nicht zwingend am Anfang, sondern an einem beliebigen Zeitpunkt beginnt. Da das verwendete Containerformat für den Netzwerkteil transparent sein soll, muß dies durch einfaches Auslassen von Datenblöcken möglich sein, ohne daß dabei Header- oder sonstige Datenstrukturen verändert werden müssen.

Zusammengefaßt sind an das verwendete Multiplexformat (**Containerformat**) folgende Anforderungen zu stellen:

- Die verwendeten Video- und Audioformate müssen unterstützt werden.
- Es muß mindestens ein Video- und ein Audiostrom gemeinsam transportiert werden können.
- Den einzelnen Paketen bzw. Frames der Elementarströme müssen **Zeitstempel** (Timestamps) zugeordnet werden können, um die Ströme zeitlich synchron abspielen zu können.

- Suchen (»Spulen«, Seeking) im Datenstrom muß möglich sein, ohne auf zusätzliche Datenstrukturen mit fester Position im Datenstrom (zum Beispiel Header oder Indexblöcke) angewiesen zu sein.
- Es muß mit geringem Aufwand möglich sein, aus einem vollständigen, korrekt abspielbaren Datenstrom einen ebenfalls korrekt abspielbaren Teildatenstrom zu erzeugen, der einen zeitlichen Ausschnitt des Gesamtstroms enthält.
- Es sollten gegenüber den Nutzdaten möglichst wenig zusätzliche Daten (**Overhead**) für die Verwaltung hinzugefügt werden.
- Die Erstellung von Datenströmen sollte nicht übermäßig komplex sein, oder es sollte eine Programmierschnittstelle dafür bereitstehen.
- Das Format sollte von etablierten Softwareplayern problemlos abgespielt werden können, so keine Neuimplementation oder weitreichende Modifikation der Client-Software notwendig wird.

Anhand dieser Anforderungen wurden die verbreitetsten Containerformate auf ihre Tauglichkeit für das Streamingsystem untersucht.

8.1.1. Microsoft AVI (Audio/Video Interleave)

Microsofts AVI ist nach MPEG das verbreitetste Containerformat. Es ist recht komplex, schlecht dokumentiert und läßt viele moderne Features vermissen. Insbesondere fehlen Timestamps, wodurch das Spulen ohne Indexdaten und das einfache Erstellen von Teildatenströmen unmöglich werden.

8.1.2. Microsoft ASF (Advanced Streaming Format)

Das ebenfalls von Microsoft stammende Advanced Streaming Format wurde mit dem Windows Media Player 6 eingeführt. Es wurde als Ersatz für das ältere AVI konzipiert, hat sich aber bisher noch nicht in selbem Maße durchgesetzt. Es behebt dessen Probleme und erweist sich von den Features her als durchaus brauchbar für `lvstream`. Jedoch ist das Format relativ komplex und es ist eine andere Version dokumentiert als tatsächlich verwendet wird [9], so daß das Erstellen von Datenströmen vergleichsweise schwierig ist, zumal unter Linux keine funktionierende Bibliothek oder dergleichen existiert, die dies erledigen könnte. Das VideoLAN-Projekt bietet zwar einen ASF-Multiplexer an, dieser arbeitet jedoch leider nicht korrekt: Die damit erzeugten Ströme können mit MPlayer, dem VideoLAN-Client und dem Windows Media Player mitunter nicht fehlerfrei wiedergegeben werden.

8.1.3. Apple QuickTime / MPEG-4 Systems

Das QuickTime-Dateiformat ist von Apple 1991 eingeführt und seitdem stetig weiterentwickelt worden. Es ist in seiner Struktur dem AVI-Format ähnlich, jedoch nochmals komplexer, und teilt auch einige seiner Probleme: So sind keine Timestamps oder sonstigen Möglichkeiten vorgesehen, um ohne einen Index Synchronisation und Seeking zu ermöglichen.

Das Dateiformat von MPEG-4 Systems ist dem QuickTime-Format sehr ähnlich (Zitat aus [10]: »QuickTime was taken as a starting point«), versteht sich aber wie dieses nur als Austauschformat, nicht als Streamingformat.

8.1.4. Matroska

Das Matroska-Format ist eine Open-Source-Entwicklung, die versucht, alle für ein Containerformat denkbaren Features wie mehrere Ton- und Bildspuren, Untertitel, Kapitelmarken und sogar DVD-ähnliche Menüs zu implementieren. Dieser Tatsache, in Verbindung mit der Entscheidung für eine binäre XML-Representation als Basis-Datenformat, ist die enorm hohe Komplexität geschuldet, die zum Erstellen und Abspielen von Matroska-Datenströmen notwendig ist. Obwohl Matroska vom Design her streambar ist, bleibt es in der praktischen Anwendung vornehmlich auf die Verwendung als Dateiformat mit der typischen Dateiendung .mkv beschränkt.

8.1.5. RealMedia

RealMedia ist ein etabliertes Streaming-Format, das nahezu alle wichtigen Features wie beispielsweise Timestamps unterstützt. Allerdings ist das Datenformat stark vom Hersteller RealNetworks geprägt und unterstützt daher lediglich die von diesem Unternehmen angebotenen Audio- und Videocodecs. Daher kommt RealMedia als Containerformat für dieses Streaming-Projekt nicht in Betracht.

8.1.6. Nullsoft Streaming Video (NSV)

Die Firma Nullsoft ist durch ihren Medienplayer »Winamp« bekannt geworden und bietet seit einigen Jahren auch ein eigenes HTTP-basiertes Streamingsystem an. Das zugehörige Transportformat ist äußerst einfach aufgebaut und auf die allernötigsten Features reduziert. Dieser einfache Aufbau ermöglicht es, auch ohne daß eine fertige Bibliothek existiert, in wenigen Zeilen Code NSV-Streams zu erzeugen. Die Tatsache, daß im Datenstrom periodisch der komplette (wenngleich sehr kleine) Header wiederholt wird, ermöglicht die leichte Erstellung von Teildatenströmen, was mangels Timestamps jedoch stets Audio/Video-Asynchronität zur Folge hat. Außerdem sind dadurch Suchoperationen im Datenstrom ausgeschlossen.

8.1.7. MPEG PS (Program Stream)

MPEG-Programmströme kommen immer dann zum Einsatz, wenn MPEG-2-Videostreams mit Tonspuren auf Datenträgern abgespeichert werden; so sind auf DVDs und Video-CDs Programmströme enthalten, und wenn in der Umgangssprache von »MPEG-Dateien« gesprochen wird, so ist im Falle von MPEG-2 auch von Programmströmen die Rede. Aus technischer Sicht handelt es sich bei MPEG-2 PS um eine einfache Konkatenation von sogenannten **PES**-Paketen (**Packetized Elementary Stream**) mit einigen zusätzlichen Header-Paketen. Die PES-Pakete ihrerseits beinhalten jeweils eine Anzahl von Video- oder Audioframes.

Das PS-Format kann die meisten der gestellten Anforderungen erfüllen: Es besitzt Timestamps, wodurch Seeking möglich ist, und durch Neuzusammensetzen bestehender Ströme können Teilströme erzeugt werden. Leider sind Programmströme jedoch im Videoteil fest auf MPEG-2 orientiert, so daß sich keine mit anderen Verfahren codierten Bilddaten in standardkonformen PS übertragen lassen.

8.1.8. MPEG TS (Transport Stream)

Der Aufbau von MPEG-Transportströmen ist stark von dem der Programmströme verschieden. Der Grund dafür liegt in der verschiedenen Ausrichtung: Während Programmströme hauptsächlich zur festen Speicherung von Videos auf vergleichsweise sicheren Datenträgern konzipiert sind, sind Transportströme ganz auf Streaming-Anwendungen ausgelegt. Dabei handelt es sich jedoch nicht um Internet-Streaming, sondern um die breitbandige Verteilung mehrerer unabhängiger **Programme** aus jeweils eigenen Video- und Audiospuren über Broadcast-Einrichtungen. So bilden MPEG-Transportströme die Grundlage der digitalen Fernsehstandards DVB (Europa), ATSC (Amerika) und ISDB (Japan).

Auch TS übertragen PES-Pakete, jedoch sind diese nochmals in Pakete mit einer festen Länge von 184 Byte zuzüglich 4 Byte Header verpackt. Diese feste Paketlänge ermöglicht eine einfachere Resynchronisierung nach Empfangsfehlern. Als Broadcast-Format verwenden Transportströme keine Stream-Header im herkömmlichen Sinne, sondern sogenannte **PSI-Tabellen** (Program Structure Information) in speziell formatierten TS-Paketen. Diese Tabellen werden typischerweise in festgelegten Intervallen wiederholt, so daß ein Empfänger jederzeit in den Datenstrom einsteigen kann und beispielsweise bei DVB nach maximal 100 ms die Struktur des Datenstroms bekannt ist.

Im Gegensatz zu Programmströmen sind Transportströme durch die wesentlich komplexeren PSI-Strukturen in der Lage, neben MPEG-2 zumindest auch die Nachfolgestandards aus der MPEG-Familie, namentlich MPEG-4 ASP und MPEG-4 AVC (H.264), zu transportieren. Leider sind diese Teile der Spezifikation noch nicht vollständig mit den derzeit verfügbaren Software-Playern interoperabel.

Ebenfalls ungünstig für den Endnutzer-orientierten Internet-Streaming-Einsatz wirkt sich der vergleichsweise hohe Overhead aus, der nur zum kleinen Teil durch die häufig wiederholten Paketheader- und PSI-Strukturen entsteht. Viel gravierender wirkt sich aus, daß ein korrekter Transportstrom stets mit einer festen Datenrate gesendet werden muß. Es müßten dabei selbst beim Einsatz von konstanten Elementarstrom-Bitraten sowohl für Video als auch für Audio noch etliche leere Pakete (**Null-Pakete**) eingefügt werden, um Toleranzen in der Bitratenverteilung abzudecken. Dies ist bei der Übertragung von Internet-Streamingvideo natürlich sehr unökonomisch.

8.1.9. Ogg (RFC 3533) / Ogg Media

Ogg ist wie Matroska eine Open-Source-Entwicklung, für die keinerlei Lizenzgebühren anfallen. Es wurde von der Xiph.org Foundation in Verbindung mit dem freien Audiocodec Vorbis entwickelt. Ogg selbst ist ein Framing-Format, das lediglich beschreibt, wie Pakete von einem oder mehreren Elementarströmen zu einem Gesamtdatenstrom zusammengesetzt werden können [11] [12]. Dabei werden Timestamps unterstützt. Das Framing ist zwar auf die Verwendung mit Vorbis optimiert, funktioniert jedoch auch mit anderen Formaten.

Ogg bietet selbst keinerlei Signalisierung, welche Datenformate transportiert werden. Diese Aufgabe muß das transportierte Medium übernehmen, also beispielsweise Vorbis oder die ebenfalls von Xiph.org angebotenen Videocodecs Theora und Tarkin. Der Transport von Medienströmen, die mit anderen Codecs erstellt wurden, wird nur durch eine Erweiterung namens »Ogg Media« (OGM), die nicht von Xiph.org selbst stammt, ermöglicht. OGM ist dennoch vergleichsweise weit verbreitet und wird von den gängigen Playern MPlayer, VideoLAN, Xine und (mit entsprechenden Plugins) sogar Windows Media Player unterstützt.

Es existieren keine fertigen Programmierbibliotheken zur Erstellung von OGM-Inhalten, jedoch sind Ogg und OGM strukturell recht einfach, was eine »from scratch«-Implementierung als Alternative zuläßt.

8.1.10. Zusammenfassung

Die Eigenschaften der betrachteten Containerformate in Bezug auf die in 8.1 genanntem Anforderungen sind in der folgenden Tabelle zusammengefaßt. Die Werte für Overhead und Komplexität sind dabei grob qualitativ gegenüber dem Durchschnitt des Feldes abgeschätzt:

	AVI	**ASF**	**QT**	**MKV**	**Real**	**NSV**	**PS**	**TS**	**OGM**
konzipiert für	File	Stream	File	File	Stream	Stream	File	Stream	File
Codec-Support	ja	ja	ja	ja	nein	ja	nein	ja	ja
Timestamps	nein	ja	nein	ja	ja	nein	ja	ja	ja
Seek ohne Index	nein	ja	nein	ja	ja	nein	ja	ja	ja
Teildatenströme	nein	ja	nein	nein	ja	nein	ja	ja	ja
Overhead	mittel	mittel	mittel	gering	gering	mittel	mittel	hoch	gering
Komplexität	mittel	mittel	mitel	hoch	mittel	gering	mittel	hoch	gering
erzeugbar	ja	nein	ja	ja	nein	nein	ja	ja	ja
abspielbar	ja	ja	ja	ja	ja	ja	ja	nein	ja

Tabelle 8.1.: Übersicht über die Features der untersuchten Containerformate

Insgesamt wird ersichtlich, daß OGM die Anforderungen am besten erfüllt. Daher wird in der der Implementation von `lvstream` auch dieses Format eingesetzt. Prinzipiell ist es jedoch mit vertretbarem Aufwand möglich, das Containerformat nachträglich zu wechseln, solange zumindest die Anforderung der Erzeugbarkeit von Teildatenströmen erfüllt ist.

8.2. Unterteilung in Access Units

Wie in 8.1 bereits angedeutet, soll der Datenstrom durchsuchbar sein, ohne daß irgend welche Teile der Netzwerkinfrastruktur Kenntnis vom Aufbau des Containerformats haben müssen. Aus Netzwerksicht soll es sich um anonyme Daten handeln, die lediglich intakt zum Empfänger transportiert werden müssen. Um dennoch das Suchen und Springen im Datenstrom zu ermöglichen, müssen schon im Streamprovider entsprechende Vorkehrungen getroffen werden.

Die konkrete Implementation teilt den Videodatenstrom in Zugriffseinheiten (**Access Units**, **AU**) ein[1]. Eine solche AU enthält die Video- und Audiodaten für einen zeitlich abgegrenzten Bereich. In der Praxis werden hier Werte von 2 bis 20 Sekunden verwendet; die Werte sind ein Kompromiß aus Netzwerkoverhead und Komfort: Jeder AU-Wechsel stellt ein Handover dar und erzeugt Netzwerkoverhead, andererseits sollten AUs möglichst klein sein, damit genaue Sprünge möglich sind.

[1] Der Begriff »Access Unit« ist dem MPEG-Standard entliehen, wo derselbe Begriff mit einer sehr ähnlichen Bedeutung verwendet wird.

8.2.1. Eigenschaften von Access Units

Die AUs werden in einer zeitlichen Reihenfolge erzeugt: AU #1 liegt am Anfang des Datenstroms, AU #2 folgt unmittelbar nach Ablauf von AU #1, AU #3 nach AU #2, usw. Eine besondere Rolle kommt der Access Unit mit der festen Nummer 0 zu: Sie enthält keine tatsächlichen Mediendaten, sondern lediglich den Datenstromheader mit den zur Decodierung zwingend nötigen Parametern. Davon abgesehen sind alle anderen AUs »self-contained« – sie können abgespielt werden, ohne daß Daten aus anderen AUs, ausgenommen der Header-AU, bekannt sein müssen.

Somit kann aus beliebigen Kombinationen von AUs ein abspielbarer Datenstrom zusammengesetzt werden, solange gewährleistet ist, daß AU #0 am Anfang des Stroms gesendet wird. Bei angenommenen 10 Sekunden AU-Länge würde so die Konkatenation der AUs 0, 1 und 2 die ersten 20 Sekunden des Datenstroms repräsentieren; AU 0, 7, 8, 9, ... würden einen Strom liefern, der 1 Minute nach Beginn anfängt. Insbesondere sind dabei auch unstetige Kombinationen möglich: AU 0, 1, 2, 3, 13, 14, 4, 5 entspräche beispielsweise einer Sequenz, die zunächst 30 Sekunden am Anfang spielt, dann plötzlich bei der 2. Spielminute fortsetzt und nach 20 weiteren Sekunden wieder an der alten Position fortfährt.

Auf diese Art und Weise können den Clients im Streaming-Netzwerk beliebige Teile des Datenstroms nach Bedarf zugesendet werden. Auch das Anspringen von beliebigen AUs ist somit möglich: empfängt ein Client gerade AU #3 und möchte eine Minute nach vorne springen, so wird als nächste AU nicht #4 angefordert, sondern #10. Einzige Einschränkung ist dabei, daß der Sprung nicht sofort ausgeführt wird, sondern erst nach Ablauf der aktuellen AU. Deswegen ist es wünschenswert, die Access Units möglichst kurz zu halten.

Bei nicht-konsekutivem Zusammensetzen von AUs muß zudem in Kauf genommen werden, daß der Medienplayer auf Clientseite zumindest kurzzeitig keine korrekte Synchronisation herstellen kann. Dieses Problem rührt daher, daß bei einem Sprung in der AU-Sequenz sich auch sprunghaft die Zeitmarken im Datenstrom ändern – eine scheinbare Störung, auf die sich der Player erst einrichten muß. In praktischen Tests mit dem verbreiteten Programm MPlayer zeigte sich dabei, daß bei größeren Sprüngen einige Sekunden vergehen können, bis Audio und Video wieder perfekt synchronisiert sind.

8.2.2. Implementation

Um den geschilderten Eigenschaften gerecht zu werden, müssen zwei Vorkehrungen in `lvstream` getroffen werden:

- Jede AU muß mit einer vollständigen Ogg-Seite, der atomaren Dateneinheit des Ogg-Formats, beginnen und enden, so daß bei einem Sprung keine fehlerhaften Seiten im Decoder ankommen.
- Jede AU muß mit einem I-Frame beginnen, damit die Decodierung in der neuen AU auf jedem Fall mit einem korrekten Referenzbild startet.

Der zweite genannte Punkt hat umgekehrt zur Folge, daß neue Access Units immer nur dann beginnen können, wenn ein I-Frame im Datenstrom vorliegt. Daher ist es von Bedeutung, daß das AU-Intervall ein ganzzahliges Vielfaches vom I-Frame-Intervall sein sollte. Ist das nicht gegeben, so wird der Beginn der nächsten AU bis zum nächsten I-Frame hinausgezögert.

Trifft `lvstream` im Videodatenstrom auf einen I-Frame, so wird geprüft, ob die nächste Access Unit fällig ist. Wenn ja, wird noch vor dem Einfügen des I-Frames in den Multiplexstrom die gerade aktive

Ogg-Seite geschlossen (»geflusht«) und der Datenstromausgabe mitgeteilt, daß danach eine neue AU beginnt. Erst dann wird der I-Frame tatsächlich in den Ogg-Seitenpuffer geschrieben.

8.3. Ausgabe der Daten

Nach dem Erstellen der Container-Datenströme und deren Unterteilung in Access Units werden diese dem Peer-To-Peer-Streamingnetzwerk übergeben. Da es sich bei dem zum Einsatz kommenden, in [1] beschriebenen Netzwerk um cachendes Netzwerk handelt, unterscheidet sich der Rechner, auf dem `lvstream` als Stream-Provider läuft, nicht von einem beliebigen anderen Peer, der die gestreamten Daten ebenfalls im Cache vorhält. Weiterhin werden die Daten über das HTTP-Protokoll mit gewöhnlichen GET-Requests transportiert; bei den Streamcaches handelt es sich folglich um nahezu gewöhnliche HTTP-Proxies.

Für den Stream-Provider bedeutet dies, daß seine Aufgaben darin liegen, die gestreamten Daten per HTTP freizugeben, und den Tracker darüber zu unterrichten, daß ein neuer Stream verfügbar ist. Dieser Ansatz läßt weitgehende Vereinfachungen zu – in letzter Konsequenz hat das zur Folge, daß `lvstream` keinerlei Netzwerkfunktionen selbst anbieten muß: Die Stream-Daten werden von einem gewöhnlichen HTTP-Server wie Apache ausgeliefert, und ein Perl-Script namens `provider.pl` aus [1] kündigt den neuen Strom dem Tracker an.

8.3.1. Übergabeformat

Auf einem Webserver oder -Proxy, der einen Stream nach [1] bereitstellt, muß eine festgelegte Verzeichnisstruktur existieren. Grundlage ist ein Basisverzeichnis für den Stream, das per HTTP-URL direkt erreichbar ist, beispielsweise

http://example.com/streams/MyStream/

Darunter finden sich zwei Verzeichnisse hi und lo, die jeweils den hochauflösenden und niedrig aufgelösten Stream enthalten. In den Verzeichnissen befinden sich jeweils von 0 an durchnumerierte Dateien, von denen jede eine Access Unit an Mediendaten beinhaltet. Die AU, deren Erstellung gerade im Ablauf ist, wird in einer Datei current_au hinterlegt. Zusätzlich gibt es eine Datei index, in der die Zuordnung der Abspielzeiten zu den AU-Nummern abgelegt wird.

8.3.2. Übergabeprotokoll

In [1] ist gefordert, daß Access Units erst verteilt werden können, wenn sie vollständig sind. Aus diesem Grunde werden AUs während der Erstellung erst in eine Datei mit dem festen Namen current_au geschrieben, die nicht per HTTP erreichbar ist und erst nach der Fertigstellung ihren endgültigen Namen erhält.

Außerdem erhält die index-Datei eine zweite Bedeutung: In ihr kann indirekt mitverfolgt werden, welche AU zuletzt fertiggestellt wurde. Das Script zur Tracker-Kommunikation, `provider.pl`, macht davon Gebrauch, indem es den Inhalt der Datei mitverfolgt. Sobald in der Datei das Vorhandensein einer neuen AU erkenntlich wird, teilt das Script dies umgehend dem Tracker mit, der diese neue AU daraufhin dem gesamten Netzwerk verfügbar macht.

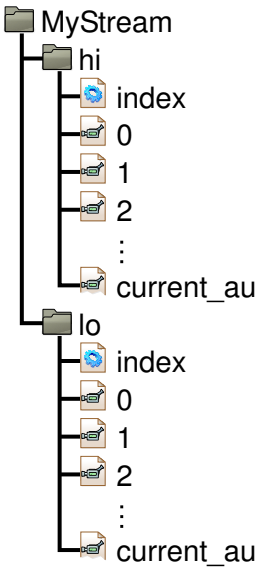


Abbildung 8.1.: Verzeichnisstruktur eines Streams

9. Performance

9.1. Rechenzeitbedarf

Wie eingangs erwähnt, soll es sich bei der in dieser Arbeit vorgestellen Software um eine Lösung handeln, welche die Codierung von Audio und Video in Echtzeit erledigt. Dies impliziert natürlich die Voraussetzung, daß der Code performant genug sein muß, um auf üblicher, bezahlbarer Hardware eine Echtzeitcodierung zu bewältigen. Um diesen Faktor zu untersuchen, wurden anhand einer Testsequenz Benchmarks auf mehreren x86-Computersystemen durchgeführt.

Es wurden jeweils Testläufe für vier verschiedene Qualitätseinstellungen gestartet:

- Eine Codierung in der maximalen Qualitätsstufe (`–q 9`, Motion Estimation für hohe und niedrige Auflösung unabhängig voneinander in bester Qualität).
- Eine Codierung in der höchsten Standard-Qualitätsstufe (`–q 8`, sehr genaue Motion Estimation in der kleineren Auflösung, MV-Skalierung und -Verfeinerung für die größere Auflösung).
- Eine Codierung in der niedrigsten Standard-Qualitätsstufe (`–q 1`, grobe Motion Estimation in der kleineren Auflösung, MV-Skalierung und -Verfeinerung für die größere Auflösung).
- Eine Codierung in der minimalen Qualitätsstufe (`–q 0`, grobe Motion Estimation in der kleineren Auflösung und *zusätzlich* deaktivierte Verfeinerung der skalierten Bewegungsvektoren in der hohen Auflösung).

Jeder Testlauf wurde dabei dreifach wiederholt und anschließend das beste, von den Einflüssen der Testumgebung am wenigsten verfälschte Ergebnis ausgewählt.

Prozessor				**Meßergebnisse** (fps)			
Modell	Kern	Taktung	L2-Cache	–q 9	–q 8	–q 1	–q 0
AMD Athlon XP	Thoroughbred	1800 MHz	256 KiB	13,56	19,58	22,25	28,49
AMD Athlon XP	Barton	2083 MHz	512 KiB	15,86	18,49	20,91	29,00
Intel Pentium M	Banias	1700 MHz	1024 KiB	15,70	22,94	26,81	33,48
Intel Celeron D	Prescott	2400 MHz	256 KiB	15,30	24,69	29,09	34,00
AMD Athlon 64	Newcastle	2000 MHz	512 KiB	19,96	30,10	35,17	43,31
AMD Athlon 64	Winchester	1800 MHz	512 KiB	19,43	31,10	36,28	45,52

Tabelle 9.1.: Codiergeschwindigkeit auf verschiedenen Prozessoren

Die Ergebnisse zeigen, daß es zumindest auf aktuellen Mainstream-Rechnern mit Athlon-64-CPU problemlos möglich ist, `lvstream` bei höchster Qualität in Echtzeit zu betreiben; das Celeron-D-System

verfehlt nur sehr knapp die 25-fps-Marke. In der minimalen Qualitätsstufe sind sogar alle getesteten Systeme in der Region um 2 GHz Taktfrequenz in der Lage, ein Video ohne Framedrops zu erstellen.

Besonderes Augenmerk ist auf den Vergleich zwischen den Qualitätsstufen –q 8 und –q 9 zu legen: Hier zeigt sich, daß die Methode der intelligenten Gewinnung der Bewegungsvektoren tatsächlich etwa 50% schneller ist als eine konventionelle, vollständige Bewegungsabschätzung.

In einem weiteren Versuch wurden die »Hot Spots« ermittelt, in denen die meiste Rechenzeit aufgewendet wird. Dazu wurde eigens eine Spezialversion von `lvstream` erstellt, die single-threaded arbeitet. Außerdem wurden darin die wichtigsten Komponenten auf CPU-Takte genau vermessen. Dies wäre mit der multi-threaded-Standardversion nicht sicher möglich gewesen, da es in dieser leicht passieren könnte, daß ein Thread während der Messung durch einen anderen unterbrochen wird. Die single-threaded-Version stellt sicher, daß das Programm sich zumindest nicht selbst beeinflussen kann – dennoch sind Unterbrechungen durch das Betriebssystem oder andere Prozesse in den Meßwerten enthalten. Um die so entstandenen Ungenauigkeiten zu minimieren, wurden abermals alle Messungen dreifach wiederholt und anschließend Mittelwerte berechnet.

	–q 9	–q 8	–q 1	–q 0
Eingabe	2 135	1 329	2 088	1 815
Audio-Demuxer	109	107	107	109
Audio-Anpassung	13	12	12	12
Audio-Resampling	6	6	6	6
Audio-Encoder	949	917	941	945
DV-Decoder	12 163	11 688	12 055	12 041
Deinterlacer	4 056	3 886	4 013	4 019
Logo-Einblendung	14	13	14	13
Bild-Skalierung	2 210	2 123	2 184	2 179
MV-Skalierung	69	65	71	72
Encoder (lo-res)	15 875	15 492	5 180	5 140
Encoder (hi-res)	64 970	27 436	27 505	17 471
Ausgabe	191	184	189	187
Sonstiges	6	7	6	5
Gesamt	102 766	63 265	54 371	44 014

(Millionen Taktzyklen für die Codierung eines 33,6 Sekunden langen Testclips auf einem Intel Celeron/Coppermine-2)

Tabelle 9.2.: Rechenzeitanteile der einzelnen Komponenten

Betrachtet man die Verteilung der Rechenleistung auf die verschiedenen Komponenten des Systems, fallen sofort die zwei Encoder und der DV-Decoder als die rechenintensivsten Bestandteile auf. Insbesondere der MPEG-4-Encoder für die höhere Auflösung benötigt einen Großteil der Leistung; jedoch ist auch erkenntbar, wie die intelligente Bewegungsabschätzung bei etwa gleicher Qualität den Rechenzeitanteil um fast 60% reduzieren kann.

Das Deinterlace-Filter ist für etwa 10% der Rechenzeit verantwortlich. Dies läßt eine Optimierung in MMX sinnvoll erscheinen. Das bereits in MMX implementierte Logo-Filter trägt beispielsweise nicht nennenswert zur Gesamtlast bei. Auch die Skalierung des Bildes bietet noch Spielraum für derartige

Optimierungen. Erstaunlich hoch ist der Anteil der Dateneingabe (gemessen wurden Testläufe über eine mit `cat` befüllte Pipe). Die Audioverarbeitung hingegen nimmt einen überraschend geringen Teil ein.

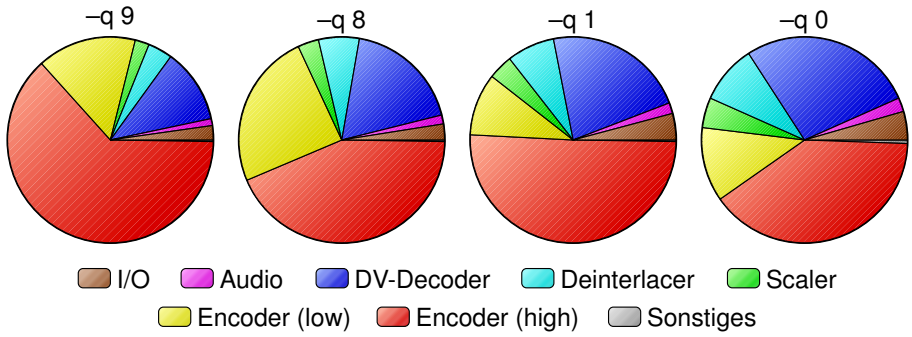


Abbildung 9.1.: Diagrammdarstellung der Rechenzeitanteile

9.2. Bildqualität

Ein entscheidender Faktor für den Einsatz als praxistaugliche Streaming-Lösung ist eine akzeptable Bildqualität. Um diese einschätzen zu können, wurden anhand der Testsequenz aus 6.3 die Abweichungen zum unkomprimierten Referenzbild exemplarisch untersucht. Dabei kam als Vergleichskriterium abermals PSNR zum Einsatz. Für diesen Versuch wurde jedoch das generierte Bildsignal nicht direkt im Encoder ausgewertet, sondern extern mittels libavcodec decodiert und anschließend verglichen.

Als Referenzbild diente jeweils das Bildsignal, das in `lvstream` direkt vor oder hinter dem Scaler anliegt – Deinterlacer und Logo-Einblendung wurden also bereits vorgenommen. Somit wird die in der Praxis erreichbare Bildqualität beurteilt.

Die Messungen erfolgten für die höhere Auflösung mit den selben Einstellungen wie in 9.1: Eine Codierung, bei der die Encoder unabhängig voneinander arbeiten (`–q 9`), zwei Läufe mit den Standard-Qualitätseinstellungen `–q 8` und `–q 1`, sowie ein Vergleich mit der minimalen Qualitätsstufe (`–q 0`). Für die Bewertung der geringen Auflösung wurden nur die Standard-Qualitätsstufen verglichen, da sich `–q 9` und `–q 0` lediglich auf die Generierung der Bewegungsvektoren für die höhere Auflösung auswirken (siehe auch A.2.3).

Auflösung	Messung	PSNR		
		Minimum	Maximum	Gesamt
352x288	–q 8	29,11 dB	42,88 dB	34,75 dB
	–q 1	26,76 dB	42,50 dB	32,44 dB
704x576	–q 9	34,13 dB	44,49 dB	38,74 dB
	–q 8	33,53 dB	44,26 dB	38,33 dB
	–q 1	32,88 dB	44,37 dB	38,06 dB
	–q 0	30,60 dB	42,56 dB	35,51 dB

Tabelle 9.3.: Ergebnisse der Qualitätsuntersuchungen

Die Ergebnisse entsprechen vollständig den Erwartungen: Die Qualitätsunterschiede zwischen der höchsten und niedrigsten Qualität in der niedrigen Auflösungsstufe sind recht hoch; in der höheren Auflösung wirken sie sich hingegen weniger stark aus, da sie durch die Bewegungsvektorverfeinerung größtenteils wieder ausgeglichen werden. Nur die minimale Qualitätsstufe ohne den Verfeinerungsschritt weist eine entsprechende Verschlechterung auf.

Von besonderem Interesse ist der Vergleich der Codierung basierend auf skalierten und verfeinerten Vektoren mit der Codierung, die ihre Bewegungsvektoren auch für die größere Auflösung mit hoher Präzision völlig neu erzeugt. Der Unterschied von nur etwa 0,4 dB zeigt, daß das vorliegende, bisher kaum optimierte Verfahren schon sehr dicht am theoretischen Optimum liegt. Ein Restfehler entsteht praktisch nur noch durch die fehlende weitergehende Suche, falls Details vorliegen, die feiner als das Bewegungsvektorraster der geringen Auflösung sind (siehe auch 6.2).

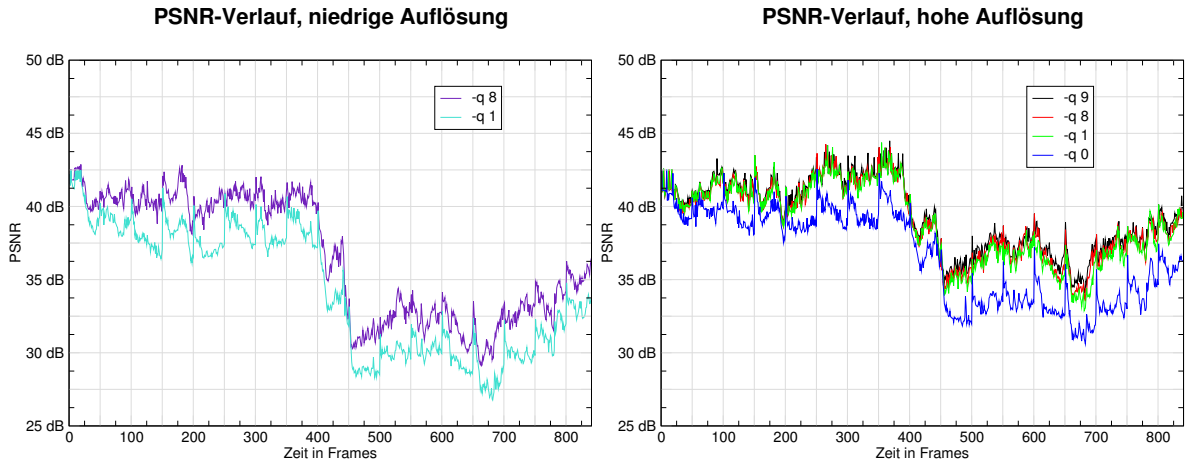


Abbildung 9.2.: PSNR-Verlauf einer Testsequenz in verschiedenen Qualitätsstufen

In den den PSNR-Verlaufsgraphen fällt auf, daß sie periodische Muster zeigen. Die Ursache dafür liegt in den Intraframe-Intervallen: Nach einem I-Frame ist die Bildqualität zunächst recht gut, wird aber durch die aufeinander basierenden P-Frames wieder langsam degradiert.

9.3. Containerformat-Overhead

Das Ogg-Containerformat ist von seinen Entwicklern explizit auf niedrigen Overhead ausgelegt. In den folgenden Berechnungen wird ermittelt, wie groß der Overhead für den Standard-Anwendungsfall ist.

9.3.1. Aufbau des Containerformats

Ogg-Ströme bestehen aus sogenannten **Seiten** mit einer empfohlenen Größe von 4 bis 8 Kilobyte. `lvstream` folgt dieser Empfehlung und beginnt nach 4 KiB bei nächster Gelegenheit eine neue Seite.

Die Seiten transportieren ein oder mehrere Elementarstrom-Pakete. Dabei können bis zu 255 Pakete in einer Seite zusammengefaßt sein; umgekehrt kann sich jedoch auch ein Paket über mehrere Seiten erstrecken, was bei Video-Frames oft vorkommt.

Ein **logischer Bitstrom** ist das Ogg-Äquivalent eines MPEG PES: Eine Folge von Seiten, deren Payload-Pakete zum selben Elementarstrom gehören. Durch Multiplexen erzeugt man einen **physischen Bitstrom**, in dem sich Seiten der beteiligten logischen Ströme abwechseln.

Die Seiten bestehen jeweils aus bis zu 255 **Segmenten**, die ihrerseits maximal 255 Byte fassen. Pakete mit einer Länge von weniger als 255 Byte werden 1:1 als Segmente übernommen. Längere Pakete werden in mehrere Segmente zu je 255 Byte aufgeteilt, nur das letzte Segment ist kürzer. Ein **Lacing** genanntes Schema codiert die Länge der Segmente in einem Ogg-Paket: Im Header gibt ein Byte an, wie viele Segmente in der Seite enthalten sind. Darauf folgt ein Byte Längenangabe pro Segment. Bei segmentierten Paketen, die Seitengrenzen überstreichen, wird in den nachfolgenden Seitenheadern durch ein Flag signalisiert, daß es sich um eine Fortsetzung handelt.

Der Seitenheader selbst ist 27 Byte lang, zuzüglich der Längenbytes. Er enthält ein festes Bitmuster zur Resynchronisation, eine Versionsnummer, Flags, einen 64-Bit-Timestamp für das letzte in der Seite endende Paket, eine Bitstromnummer (um die logischen Bitströme innerhalb eines physischen Bitstroms auseinander halten zu können), eine Seiten-Sequenznummer, eine CRC32-Prüfsumme und schließlich die Segmentanzahl.

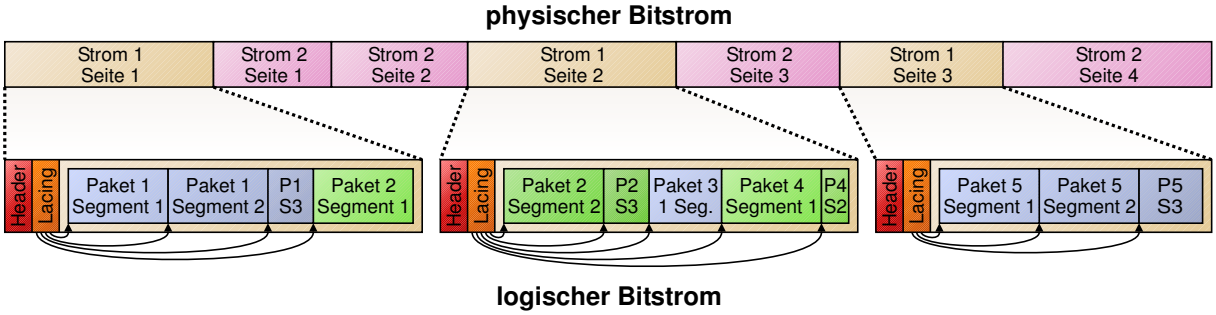


Abbildung 9.3.: Beispiel der Funktionsweise von Ogg-Framing

OGM fügt innerhalb der Pakete noch einen minimalen Header hinzu. Bei Videoframes ist dieser typischereweise nur 1 Byte lang und gibt an, ob es sich um einen Keyframe (I-Frame) handelt oder nicht. Bei Audioframes umfaßt er typischerweise 3 Byte: Ein Byte Header und zwei Byte für die Anzahl der Samples, die im Paket codiert sind.

9.3.2. Overhead für Audio-Frames

Bei den `lvstream`-typischen 40 kbps MP3-Audio entstehen Audioframes von jeweils 120 Byte pro Sekunde:

$$\frac{40000\ \frac{Bit}{Sekunde} \times 576\ \frac{Samples}{Frame}}{24000\ \frac{Samples}{Sekunde}} = 960\ \frac{Bit}{Frame} = 120\ \frac{Byte}{Frame}$$

Durch den zusätzlichen Ogg-Media-Header enstehen somit jeweils 123 Byte große Pakete. Da derart kleine Pakete in ein Segment passen, kommt lediglich ein Lacing-Byte hinzu, so daß praktisch 124 Byte pro Audio-Frame anfallen. Durch den Versuch des Ogg-Encapsulators, die Pakete möglichst dicht an 4 KiB Größe zu halten, passen $\lceil (4096-27) \div 124 \rceil = 33$ solche Pakete in eine Seite hinein. Der Gesamtoverhead für OGM, Lacing und den Header beträgt somit im Mittel $3+1+27 \div 33 \approx 4,8$ Byte. Dies bedeutet einen relativen Overhead von $4,8 \div 120 \approx 4,02\%$.

9.3.3. Overhead für Video-Frames (hohe Auflösung)

Die Videodaten für die hohe Auflösung werden in den Standardeinstellungen mit einer weitgehend konstanten Bitrate von 1,5 Mbps übertragen. Das bedeutet eine mittlere Framegröße von 7,5 Kilobyte:

$$\frac{1500000\ \frac{Bit}{Sekunde}}{25\ \frac{Frames}{Sekunde}} = 60000\ \frac{Bit}{Frame} = 7500\ \frac{Byte}{Frame}$$

Der Ogg-Media-Header umfaßt hier nur 1 Byte; außerdem fallen $\lceil 7500 \div 255 \rceil = 30$ Lacing-Bytes an. Die 7531 Byte überstreichen im Schnitt $7531 \div (4096 - 27) \approx 1,85$ Seiten. Der absolute mittlere Overhead pro Frame beträgt also $1 + 30 + 1,85 \times 27 \approx 81$ Byte. Der relative Overhead fällt mit $81 \div 7500 \approx 1,08\%$ noch deutlich geringer aus als bei den Audio-Frames.

9.3.4. Overhead für Video-Frames (niedrige Auflösung)

Als Standardwert für die Video-Bitrate werden 300 kbps verwendet, die Videoframes sind also im Mittel 1,5 Kilobyte groß:

$$\frac{300000\ \frac{Bit}{Sekunde}}{25\ \frac{Frames}{Sekunde}} = 12000\ \frac{Bit}{Frame} = 1500\ \frac{Byte}{Frame}$$

Hier kommen zu dem OGM-Zusatzheader (1 Byte) noch $\lceil 1500 \div 255 \rceil = 6$ Lacing-Bytes hinzu. Es werden durchschnittlich $1507 \div (4096 - 27) \approx 0,37$ Seiten. Der absolute mittlere Overhead beläuft sich damit auf $1 + 6 + 0,37 \times 27 \approx 17$ Byte, der relative auf $17 \div 1500 \approx 1,13\%$.

9.3.5. Gesamt-Overhead

Der Gesamt-Overhead des Systems kann nun bestimmt werden, indem die Overhead-Werte der logischen Ströme nach Bitraten gewichtet werden:

$$\frac{40 \times 4,02\% + 1500 \times 1,08\%}{40 + 1500} \approx 1,16\%$$

$$\frac{40 \times 4,02\% + 300 \times 1,13\%}{40 + 300} \approx 1,47\%$$

Diese Resultate liegen im Rahmen der in RFC 3533 [11] für Ogg angegebenen typischen Overhead-Werte von $1 - 2\%$.

9.4. Speicherverbrauch

Der Speicherverbrauch einer Encoding-Sitzung mit `lvstream` setzt sich aus zahlreichen Komponenten zusammen.

Verbrauch externer Bibliotheken

- **libavcodec (DV-Decoder)**
  Der Speicherverbrauch des DV-Decoders wurde durch Verfolgen der Aufrufe des libavcodec-eigenen `malloc`-Wrappers ermittelt. Unter den allokierten Speicherblöcken finden sich neben einigen Verwaltungsdaten ein 256 KiB großer Datenstrompuffer sowie ein Framebuffer. Dieser umfaßt bei PAL einen Bereich von 752x608 Bildpunkten, was der nominellen DV-Auflösung von 720x576 zuzüglich 16 Pixel Rand an jeder Bildkante entspricht.

- **XviD-Verwaltungsdaten**
  Die beiden XviD-Instanzen verwenden jeweils einige Kilobyte an Daten, die Steuerungszwecken dienen, also keine Bilddaten enthalten. In diesen Verwaltungsdaten sind auch die Makroblockstrukturen enthalten, die für jeden Makroblock Parameter wie Stärke der Quantisierung, Bewegungsvektoren usw. speichern. Daher ist die Größe dieser Daten von der Bildauflösung abhängig.

- **XviD-Bilddaten**
  Für jede XviD-Instanz werden 10 Framebuffer gleicher Größe verwendet. Jeder Framebuffer besitzt dabei Abmessungen, die noch über der nominellen Bildgröße liegen, da XviD an jeder Kante des Bildes 64 Pixel Rand hinzufügt. Dieser Rand dient hauptsächlich der Vereinfachung der Motion Compensation: In MPEG-4 dürfen Bewegungsvektoren auch außerhalb des Bildbereichs zeigen, wobei die jeweiligen Randpixel nach außen wiederholt werden. Eine Erweiterung des Framebuffers um eben diese Randpixel eliminiert somit einen Spezialfall der Motion Compensation und trägt zu dessen Beschleunigung bei.

  Ferner wird jeder XviD-Framebuffer um eine weitere Bildzeile ergänzt; außerdem werden stets 64 zusätzliche Bytes allokiert, um auch mit »groben« SIMD-optimierten Operationen nicht versehentlich außerhalb des reservierten Speicherbereichs zu lesen oder schreiben. Unter Berücksichtigung des verwendeten 4:2:0-Chroma-Subsamplings (siehe B.3) ergeben sich somit folgende Formeln zur Bestimmung der Größe eines XviD-Framebuffers:

$$\begin{aligned}
\text{xvid_framesize}_{\text{Y}}(x,y) &= (x+2\times 64)\times(y+2\times 64+1)+64\\
\text{xvid_framesize}_{\text{C}}(x,y) &= \frac{x+2\times 64}{2}\times\frac{y+2\times 64}{2}+64\\
\text{xvid_framesize}(x,y) &= \text{xvid_framesize}_{\text{Y}}(x,y)+2\times\text{xvid_framesize}_{\text{C}}(x,y)
\end{aligned}$$

  Im Normalfall sind Bewegungsvektoren bei MPEG-4 auf halbe Pixel genau. Um bei der Motion Compensation nicht für jeden Block eine zeitaufwendige Interpolation der Halbpixel durchführen zu müssen, wird dies bei XviD ebenfalls nur einmal pro Frame durchgeführt. Aus dem Ursprungsbild werden dabei 3 Kopien erzeugt, die gegenüber dem Original jeweils um ein halbes Pixel horizontal, vertikal oder diagonal verschoben sind. Die Motion Compensation-Routine selbst wählt dann je nach Halbpixelversatz des Bewegungsvektors eines der 4 Bilder aus und überträgt lediglich komplette Pixel, ohne dabei komplexe Berechnungen durchzuführen oder Spezialfälle berücksichtigen zu müssen.

Die Zahl von insgesamt 10 Bildpuffern ergibt sich durch folgende Elemente:

- 4 Framebuffer (je 1 pro Halbpixelversatz) für den *aktuellen* Frame.
- 4 Framebuffer (je 1 pro Halbpixelversatz) für den *Referenz*frame (den letzten I- oder P-Frame).
- 1 Framebuffer für GMC (Global Motion Compensation). XviD legt diesen Puffer auch dann an, wenn keine GMC verwendet wird.
- 1 Framebuffer als Queue. Dem Encoder zugeführte Frames werden zunächst in dieser Queue abgelegt, bei reiner I/P-Frame-Codierung jedoch sofort wieder daraus entfernt.

- **LAME (MP3-Encoder)**
  Auch in der LAME-Library wurde durch Verfolgen von `malloc`- und `calloc`-Aufrufen der benötigte Speicherverbrauch ermittelt. Dieser ist mit rund 300 KiB recht gering.

Audio-Pipeline

- **Ringpuffer**
  Jeder Videoframe wird von rund 1920 Audio-Samples begleitet; die Ringpuffereinträge fassen daher einen »sicheren« Wert von 2048 Samples zu je 4 Byte (16-Bit-Samples, Stereo). Jeder Eintrag enthält zudem einen Längenwert (nochmals 4 Byte). Der Ringpuffer umfaßt insgesamt 10 Einträge.

- **Sampledaten**
  Nach dem Anpassen der Audiodaten (siehe 7.1) entsteht ein Puffer mit der festen Größe von 1920 Samples.

- **Resample-Puffer**
  Das Resampling von `lvstream` ist als wiederverwendbare Bibliothek geschrieben, die ihrerseits eine feste Puffergröße von 2 Sekunden verwendet. Bei der Standard-Zielsamplerate von 24 kHz (Mono) sind dies 24000 16-Bit-Samples.

- **Ausgabepuffer**
  Der Bitstrom-Ausgabepuffer, in dem die MP3-Frames zusammengesetzt werden (siehe 7.4), hat eine feste Größe von 8 KiB.

Video-Pipeline

- **Ringpuffer**
  Der Video-Ringpuffer umfaßt 10 Einträge, von denen jeder einen DV-Frame (bei DV-PAL 144000 Byte) und eine Framenummer (32-Bit-Integerwert) enthält.

- **Sprite-Daten**
  Jedes Pixel des eingeblendeten Sprites wird als 8-Bit-Wert im Speicher abgelegt.

- **Skalierte Bilddaten**
  Für das Ergebnis der Skalierung wird ein kleiner Framebuffer von 352x288 Bildpunkten (352x240 bei NTSC) angelegt. Auch dieser Framebuffer wird mit 4:2:0-Chroma-Subsampling angelegt.

- **Bewegungsvektoren**
  Die Bewegungsvektordaten bestehen aus zwei 32-Bit-Integerwerten für jeden 8x8-Block des hochauflösenden Bildes.

- **Ausgabepuffer**
  XviD gibt den erzeugten Datenstrom in einen Puffer aus, der von der Client-Applikation vorgegeben wird. `lvstream` reserviert hierfür 256 KiB Speicher pro XviD-Instanz.

Ausgabe

- **Ogg-Ausgabepuffer**
  Um die insgesamt 4 logischen Ogg-Ströme (Audio und Video, jeweils für hohe und niedrige Auflösung) ausgeben zu können, werden jeweils Puffer benötigt, welche einen Ogg-Header (27 Byte), Lacing-Werte (255 Byte) und Payload-Daten ($255 \times 255 = 65025$ Byte) speichern.

Gesamtverbrauch

Insgesamt ergibt sich somit bei Verwendung der Standardparameter ein Verbrauch von etwa 17 MiB, wobei der größte Teil des Verbrauchs duch die XviD-Framebuffer (8,3 MiB und 2,9 MiB) und -Verwaltungsdaten (0,3 MiB und 1,3 MiB) sowie den Video-Ringpuffer (1,4 MiB) zustandekommt.

Komponente	**Formel**	**Bytes**
Verbrauch externer Bibliotheken		
libavcodec-Daten		978 332
XviD-Verwaltungsdaten (low)		354 156
XviD-Bilddaten (352x288)	$10 \times$ xvid_framesize$(352, 288)$	3 001 920
XviD-Verwaltungsdaten (high)		1 409 100
XviD-Bilddaten (704x576)	$10 \times$ xvid_framesize$(704, 576)$	8 796 160
LAME-Daten		339 132
Audio-Pipeline		
Audio-Ringpuffer	$10 \times (2048 \times (2 \times 2) + 4)$	81 960
Audio-Sampledaten	$1920 \times (2 \times 2)$	7 680
Audio-Resample-Puffer	2×24000	48 000
Audio-Ausgabepuffer		8 192
Video-Pipeline		
Video-Ringpuffer	$10 \times (144000 + 4)$	1 440 040
Sprite-Daten	Beispiel: 136×35	4 760
Skalierte Bilddaten	$352 \times 288 + 2 \times ((352/2) \times (288/2))$	152 064
Bewegungsvektoren	$(704/8) \times (576/8) \times (2 \times 4)$	50 688
Video-Ausgabepuffer	2×262144	524 288
Ausgabe		
Ogg-Ausgabepuffer	$(2 \times 2) \times (27 + 255 + 255 \times 255)$	261 228
Insgesamt		17 457 700

Tabelle 9.4.: Speicherverbrauch der Encoder-Komponenten

In der Untersuchung des Speicherverbrauchs wurden kleinere Posten wie `lvstream`-interne Verwaltungsdaten, die nur wenige Kilobyte betragen, aus Gründen der Übersichtlichkeit ausgelassen.

9.5. Latenzzeiten

Ein weiterer wichtiger Faktor bei Live-Streaming-Systemen ist die entstehende Latenzzeit, also die Verzögerung zwischen der Aufnahme von Bild und Ton und deren Wiedergabe. Es ist wünschenswert, diese Zeit so gering wie möglich zu halten, um beispielsweise Live-Konferenzen zu ermöglichen. Im folgenden sollen daher Faktoren untersucht werden, welche die Latenz beeinflussen.

Prinzipiell besteht die Gesamtlatenz (End-to-End-Latenz) aus drei Teilen: Zunächst tritt eine gewisse Verzögerung bereits beim Encoding der Daten ein. Danach müssen die Daten über das Netzwerk zu den Clients übertragen werden, und schließlich findet auch bei der Wiedergabe eine gewisse Verzögerung durch die notwendige Zwischenpufferung statt. Für die Netzwerklatenz sei auf [1] verwiesen, die beiden anderen Komponenten werden an dieser Stelle beschrieben.

Ein gewichtiger Anteil der Netzwerklatenz findet jedoch schon Beachtung in `lvstream`: Bei der Ausgabe der codierten Medienströme im »streambaren« Format werden immer nur komplette Access Units freigegeben. Dadurch ist in der vorliegenden Implementation die Länge einer Access Unit eine untere Schranke für die erreichbare Netzwerklatenz.

9.5.1. Encoder-Latenz

Im Encoder spielen die folgenden Faktoren eine Rolle für die Latenzzeit:

- Beim Capture werden so lange Daten gesammelt, bis ein Frame vollständig eingetroffen ist. Da eine DV-Kamera einen isochronen Datenstrom mit konstanter Datenrate überträgt, benötigt die Übertragung und Aufnahme eines Frames daher eine Framelänge an Zeit. Beim hierzulande gebräuchlichen PAL-System entspricht dies 40 ms.
- Die Bearbeitung (Decodierung, Filterung und Neucodierung) eines Bildet findet ebenfalls innerhalb eines Frame-Intervalls statt. Je nach Geschwindigkeit des Encoding-Rechners kann die effektiv benötigte Zeit beliebig kurz sein; durch das Echtzeitkriterium kann sie jedoch im Mittel 40 ms nicht übersteigen.
- Vor der Ausgabe einer Ogg-Seite werden erst so viele Daten akkumuliert, bis eine Seite voll belegt ist. Für Frames, die länger sind als die Seitengröße von 4 KiB (was für das hochauflösende Video der Normalfall ist), bleibt somit nahezu immer ein Überhang in der Folgeseite, der erst mit dem nächsten Frame, also 40 ms später ausgegeben wird.
  Sind die Frames kürzer als 4 KiB, werden sogar mehrere Frames in eine Seite verpackt. Daher kann es auch mehrere Framelängen dauern, bis die Daten tatsächlich ausgegeben werden. Der Videostrom der niedrigen Auflösung weist beispielsweise eine typische Framegröße von 1–2 Kilobyte auf, wodurch Wartezeiten von bis zu 4 Frames, also 160 ms auftreten können.

Die Encoder-Latenzzeit ist also abhängig von der auftretenden minimalen Framegröße und somit indirekt von der verwendeten Bitrate. Um den Zusammenhang zwischen Bitrate und Framegröße zu untersuchen, wurde die minimale Framegröße anhand einer Testsequenz empirisch über einen weiten Bitratenbereich bestimmt. Dabei ist zu erkennen, daß für den Videostrom mit der geringen Auflösung schon unter 1000 kbps eine »Sättigung« bei 700 Byte auftritt. Der Framegrößenverlauf für den hochauflösenden Videostrom zeigt hingegen einen groben Verlauf, aus dem sich durch Regressionsanalyse folgende Potenzfunktion als Approximation ableiten läßt:

$$\mathit{Framegröße}\ [\text{Byte}] = 24{,}573 + (\mathit{Bitrate}\ [\text{kbps}])^{0{,}65671}$$

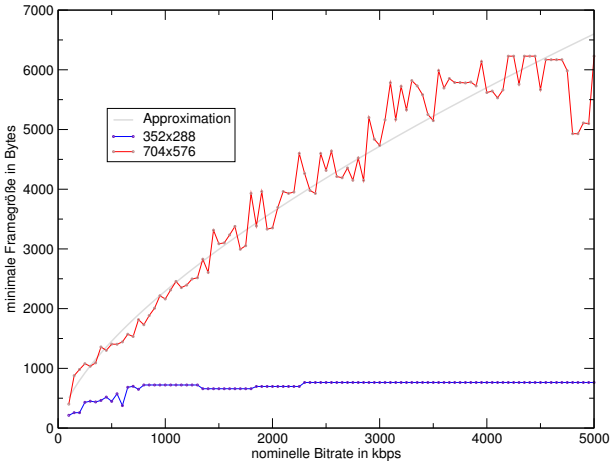


Abbildung 9.4.: Minimale Framegröße in Abhängigkeit von der Bitrate

Für einige exemplarische Bitraten sind gerundete Framegrößen und daraus resultierende Latenzzeiten in Tabelle 9.5 zusammengefaßt. Für die in `lvstream` in der Standardeinstellung verwendeten Datenraten von 300 und 1500 Kilobit pro Sekunde ergeben sich somit maximale Latenzzeiten von 440 ms beziehungsweise 120 ms.

Bitrate	min. Framegröße	Frames/Seite	max. Latenz
100 kbps (lo)	400 Byte	11	520 ms
300 kbps (lo)	500 Byte	9	440 ms
500 kbps (lo)	700 Byte	6	320 ms
1000 kbps (hi)	2000 Byte	3	200 ms
$\geq$1200 kbps (hi)	5000 Byte	1	120 ms

Tabelle 9.5.: Einige Bitraten und daraus folgende maximale Encoder-Latenzen

Die *minimale* Encoder-Latenz liegt bei knapp über 40 ms: Falls der codierte Frame auf einer Ogg-Seitengrenze endet, entsteht bei der Ausgabe keine zusätzliche Wartezeit, und lediglich Capture und Codierung bestimmen die Verzögerung.

Bei strenger Betrachtung ist dieser Wert nicht ausreichend, denn ein codierter Frame kann erst dann auch abgespielt werden, wenn die Audio-Daten für den Abspielzeitpunkt ebenfalls vorliegen. Da Audio-Seiten aufgrund der deutlich geringeren Datenrate wesentlich seltener geschrieben werden als Video-Seiten, ist dies ein nicht zu unterschätzender Beitrag zur Gesamt-Latenz. Ob die Audio-Latenz als Teil der Encoder- oder Decoder-Latenz angesehen wird, spielt für die Gesamt-Latenz keine Rolle. Allerdings darf sie nur auf einer der beiden Seiten in der Rechnung beachtet werden, da sie sonst doppelt in den Gesamtwert eingeht. Für die vorliegende theoretische Betrachtung soll sie nun als Teil der Decoder-Latenz aufgefaßt werden.

9.5.2. Decoder-Latenz

Auf Decoderseite ist der entscheidende Faktor die minimale Größe des Eingabepuffers, die notwendig ist, bis genügend Daten zum Abspielen vorhanden sind. Das Worst-Case-Szenario für die entstehende Latenzzeit besteht hierbei darin, daß die Datenrate der Netzanbindung exakt ausreicht, um den Strom in Echtzeit auszuliefern.

Die Größe des Eingabepuffers wird bestimmt durch vier Faktoren:

- Der größte im Stom auftretende Frame muß in den Puffer passen.
- Im Worst Case endet der Frame direkt nach Beginn der folgenden Ogg-Seite. Somit muß eine weitere Seite eingelagert werden können.
- Da die Video- und Audiodaten gemultiplext sind, muß der Decoder mindestens so viele Daten puffern, bis die zum Videoframe passenden Audioframes übertragen werden. Da auch die Audiodaten in ganzen Ogg-Seiten übertragen werden, ergibt sich die Wartezeit aus der Formel $Seitengröße \div Audiodatenrate$.
- Die Seite, welche die Audiodaten enthält, muß auch noch aufgenommen werden können.

Somit ergibt sich die gesamte Decoderlatenz als

$$t_L = \max\left(\frac{Framegröße}{Gesamtdatenrate}, \frac{Seitengröße}{Audiodatenrate}\right) + \frac{2 \times Seitengröße}{Gesamtdatenrate}$$

Als Referenz dient dabei nicht die minimale, sondern die maximale Framegröße, die praktisch immer durch einen I-Frame gestellt wird. Daher wurde die maximale I-Framegröße wie in 9.5.1 empirisch für eine große Menge von verschiedenen Bitraten berstimmt. Interessant ist dabei, daß im Bereich um 1900 kbps ein Übergang stattfindet: Bei niedrigeren Bitraten will XviD keine hochauflösenden I-Frames geringerer Größe mehr erzeugen, und umgekehrt hat XviD für die niedrigere Auflösung in diesem Bereich die höchste Qualitätsstufe erreicht. Einschränkender Faktor sind hierbei die in XviD festgelegten Limits für die Stärke der Quantisierung. Ansonsten ist, von heftigen Schwankungen abgesehen, ein grober linearer Verlauf zu erkennen, aus dem mittels linearer Regression folgender Zusammenhang ermittelt werden kann:

$$Framegröße\ [\text{Byte}] = 5758,1 + 11,419 \times Bitrate\ [\text{kbps}]$$

Ausgehend von diesen Formeln ergibt sich für den `lvstream`-Standardfall (1500 kbps Video + 40 kbps Audio = 1540 kbps gesamt; 4 KiB Seitengröße) ein Latenzwert von max(118,9 ms; 819,2 ms) + 42,6 ms = **861,7 ms**.

Dabei handelt es sich jedoch nur um einen Richtwert. Die Implementierung des Players kann auch andere Pufferstrategien anwenden, welche die Latenz vergrößern können. Eine größere Latenz ergibt sich dadurch, daß der Player einen noch größeren Puffer verwendet als unbedingt notwendig. Dies ist das Standardverhalten vieler Player. Die übermäßige Pufferung hat den Vorteil, daß sich kurzzeitige Schwankungen in der Netzwerkbandbreite nicht sofort als Aussetzer in der Wiedergabe äußern.

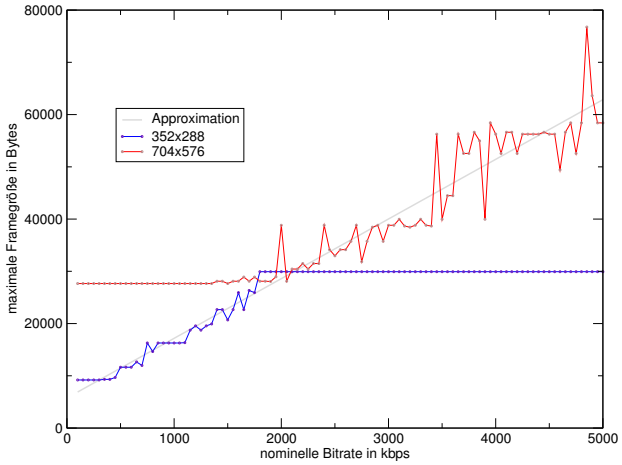


Abbildung 9.5.: Maximale Framegröße in Abhängigkeit von der Bitrate

9.5.3. Meßergebnisse

Zusätzlich zu den theoretischen Latenzzeit-Betrachtungen wurde eine praktische Messung auf einem System mit Athlon64-Prozessor (AMD Athlon64 3000+, Winchester-Kern) durchgeführt, um die Latenz in »Real-World«-Anwendungen anzugeben.

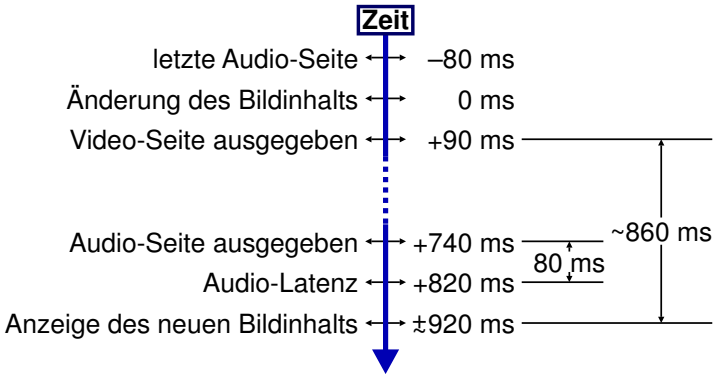
Grundlage war der folgende Versuchsaufbau: Mit einer Kamera wurde das Bild eines TFT-Monitors aufgenommen, mit `lvstream` verarbeitet, und per Named Pipe direkt an MPlayer [6] weitergereicht. Dieser zeigte dabei das Videobild so an, daß die Kamera das MPlayer-Fenster in ihrem Bildausschnitt wieder aufnimmt. Die verwendete `lvstream`-Version wurde so modifiziert, daß für ausgegebene Ogg-Seiten die aktuelle Systemzeit mitprotokolliert wurde.

Die eigentiche Messung fand statt, indem eine weitere kleine Hilfsapplikation, die ein einfarbiges Fenster anzeigt, ebenfalls in den aufgenommenen Bildausschnitt gebracht wurde. Per Mausklick konnte dabei die Farbe des Fensters umgeschaltet werden, was ebenfalls mitprotokolliert wurde.

Im aufgenommenen Video wurde nach dem Versuch durch die Betrachtung der einzelnen Frames festgestellt, in welchem Frame die Umschaltung der Farbe stattfand und in welchem Frame die Änderung auch im wieder aufgenommenen MPlayer-Fenster erschien. Durch die Differenz der Framenummern ergibt sich dadurch der Wert für die **End-to-End-Latenz** von **920 ms**.

Durch die Framenummer und die ausführlichen Protokolle wurde weiterhin festgestellt, zu welchem Zeitpunkt der Frame mit der Farbänderung komplett ausgegeben wurde. Durch Vergleich mit der Zeit der Farbänderung ergibt sich eine **Video-Latenz** von **89,4 ms**. Dies ist ein ausgesprochen guter Wert, der fast ausschließlich durch die Capture- und Ausgabe-Verzögerung zustandekommt.

Wie in 9.5.1 beschrieben, sind mit der Ausgabe des Videoframes jedoch noch nicht alle Daten vollständig, die zum Abspielen notwendig sind: Es fehlen die zugehörigen Audio-Informationen. Anhand des Zeitpunktes der Ausgabe der nächsten Audio-Seite wurde somit eine kombinierte **Audio-/Encoderlatenz** von **741,4 ms** ermittelt. Dieser Wert liegt im erwarteten Rahmen: Bei 40 kbps Audio-Datenrate und 4 KiB Seitengröße ergibt sich, daß im Mittel alle 819,2 ms eine Audio-Seite vollständig ist. Der gemessene Frame lag auch erwartungsgemäß etwa 77,8 ms nach dem letzten Audio-Seitenwechsel vor.



(Abbildung nicht maßstabsgetreu, Zeiten gerundet)

Abbildung 9.6.: Zeitlicher Verlauf der Latenzmessung

Vergleicht man den gemessenen Wert für die Audio-/Encoderlatenz wiederum mit dem zuvor ermittelten End-To-End-Latenzwert, stellt sich heraus, daß MPlayer für den betrachteten Frame eine Decoderlatenz von nur etwa 920 ms – 741,4 ms = 178,6 ms aufweist. Subtrahiert man von dieser Zahl die 77,8 ms, die seit dem Eintreffen der zu diesem Zeitpunkt abgespielten Audio-Seite vergangen sind, so erhält man ca. 100 ms restliche Zeit, die MPlayer für Pufferung, Decodierung und Anzeige des Videoframes benötigt.

Auch die berechnete Decoder-Latenz von 861,7 ms findet sich in den gemessenen Werten wieder: Als Summe mit der gemessenen Video-Latenz von 89,4 ms (die Audio-Latenz muß an dieser Stelle nicht betrachtet werden, denn sie ist bereits in der berechneten Decoder-Latenz enthalten) erhält man 951,1 ms. Die Differenz zum gemessenen Wert von 920 ms erklärt sich aus der Meßungenauigkeit: Die Methode der Latenzmessung durch Betrachten der aufgezeichneten Frames ist selbstverständlich auch nur auf eine Framelänge, also 40 ms, genau. Die 31,1 ms Abweichung liegen also innerhalb des akzeptierten Rahmens.

10. Zusammenfassung

In dieser Arbeit wurde ein Softwaresystem vorgestellt, das einem Peer-To-Peer-Videostreamingnetzwerk als Live-Stream-Quelle dient. Es erzeugt auf einem handelsüblichen PC aus mittels einer DV-Videokamera aufgenommenen Bild- und Tondaten in Echtzeit zwei Medienströme mit unterschiedlicher Qualität. Einer dieser Ströme ist für Breitband-Netzzugänge wie DSL konzipiert und liefert eine Bildauflösung von 352x288; der andere Strom dient der Versorgung von schnellen lokalen Netzen und bietet volle PAL-Auflösung (704x576).

Bei der Codierung der beiden Videoströme kommt dabei ein Verfahren zum Einsatz, welches die Generierung des höher aufgelösten Bildsignals um ca. 50% beschleunigt, dabei jedoch die Qualität nur unmerklich mindert. Um dies zu erreichen, werden die Bewegungsvektoren, die bei der Codierung des niedriger aufgelösten Bildes entstehen, bei der höheren Auflösung wiederverwendet, um den Suchraum der Bewegungsabschätzung deutlich zu verkleinern und somit die Performance zu verbessern.

Die vorliegende Implementation ist stabil und hinreichend schnell. Dennoch sind noch vielfältige Optimierungen möglich. Zur Steigerung der Geschwindigkeit bietet sich vor allem ein Umstieg auf libavcodec als zugrundeliegenden MPEG-4-Encoder an; weiterhin sind plattformspezifische Optimierungen der eingebauten Filter sinnvoll. Auch die erreichbare Qualität läßt sich noch verbessern, indem verfeinerte Algorithmen zur Anpassung der Bewegungsvektoren für das hochauflösende Videobild oder beispielsweise Rate/Distortion-Optimierungsmethoden verwendet werden. Nicht zuletzt kann auch die Usability durch ein grafisches Benutzerinterface mit Vorschaufunktion deutlich gesteigert werden.

Auf weite Sicht sind die in Kapitel 2 erwähnten erweiterten Einsatzszenarien anstrebenswert: Das gleichzeitige Übertragen von mehreren Bild- oder Tonsignalen sowie das Anlegen eines Rückkanals können das System nochmals stark aufwerten.

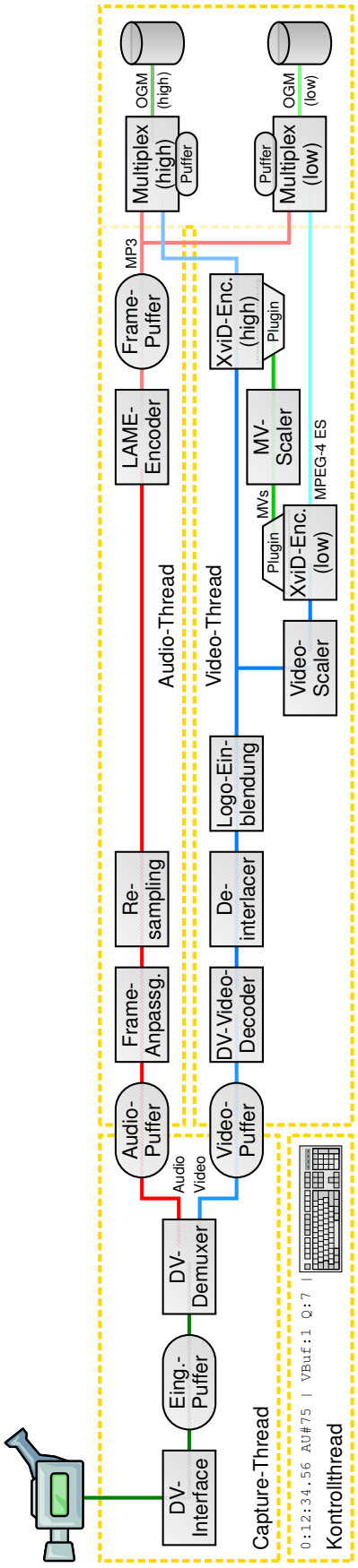


Abbildung 10.1.: Detaillierte Übersicht über alle Komponenten

A. Verwendung des Programms

A.1. Compilieren

Um `lvstream` zu übersetzen und zu verwenden, sind folgende Hardware- und Softwarekomponenten notwendig oder sinnvoll:

- Ein x86-kompatibler PC mit Linux als Betriebssystem. Auf anderen Betriebssystemen ist kein direktes IEEE-1394-Capture via libraw1394 möglich, es muß stattdessen die Pipe-Variante verwendet werden. Auf PCs mit x86_64-Architektur (AMD Athlon 64 oder Opteron, Intel Pentium 4 oder Pentium D mit EM64T) ist der Einsatz als 64-Bit-Programm zwar möglich, es wird jedoch empfohlen, `lvstream` auf einem 32-Bit-System zu compilieren, da die beteiligten Bibliotheken noch nicht vollständig auf die x86_64-Architektur optimiert sind.
- Ein intaktes Build-System aus GCC (getestet mit Version 3.3) und Entwicklerheadern für Kernel und einige Bibliotheken wie beispielsweise libpthread.
- *Empfohlen:* Das Zusatztool `dvgrab` [3] zum Aufnehmen von DV-Video in eine Pipe oder ein File.
- *Empfohlen:* Ein Mediaplayer wie MPlayer [6] zum Abspielen der generierten Medienströme.

Das `lvstream`-Paket bringt alle nötigen Nicht-Standardbibliotheken mit, jeweils in den Versionen, die zu Beginn dieser Arbeit aktuell waren:

- ffmpeg CVS vom 8.12.2004 [4]
- XviD (xvidcore) 1.1.0-beta1 vom 9.1.2005 [7] (modifiziert)
- LAME 3.96.1 vom 10.1.2005 [8]
- libraw1394 1.1.0 vom 26.11.2004 [2]

Um `lvstream` zu compilieren, genügt in der Regel

```
$ tar xjvf lvstream-distrib-1.0.tar.bz2
$ cd lvstream-distrib-1.0
$ make bootstrap
```

`make bootstrap` konfiguriert und compiliert dabei erst alle mitgelieferten Bibliotheken und schließlich `lvstream` selbst. Am Ende des Vorgangs liegt im Unterverzeichnis lvstream/ das fertige Executable vor. Die erwähnten Bibliotheken werden dabei statisch in das Binary gelinkt, so daß die entstandene Datei problemlos auf andere Rechner kopiert werden kann, ohne sie dort neu zu compilieren. Es ist jedoch darauf zu achten, daß der Zielrechner mindestens die selben CPU-Features unterstützt wie der Build-Rechner, da beim Konfigurieren der Bibliotheken Features wie MMX, SSE und SSE2 erkannt und für Optimierungen ausgenutzt werden.

In der Standardkonfiguration wird ein `lvstream`-Executable erzeugt, welches PAL-DV-Daten von der Standardeingabe entgegennimmt. Die verwendete Fernsehnorm und Capture-Methode können jeweils über Parameter von `make` (im Verzeichnis lvstream-distrib-1.0/ oder auch lvstream-distrib-1.0/lvstream/) eingestellt werden:

`make SYSTEM=PAL`	für PAL (704x576, 25 fps) *(Standardeinstellung)*
`make SYSTEM=NTSC`	für NTSC (704x480, 30 fps)
`make CAPTURE=PIPE`	für Capture aus der Standardeingabe *(Standardeinstellung)*
`make CAPTURE=RAW1394`	für direktes Capture von IEEE-1394

A.2. Einstellen der Parameter

Um die zahlreichen Laufzeitparameter einzustellen, bietet `lvstream` zwei Möglichkeiten an: Zum einen existieren getopt-ähnliche Kommandozeilenparameter, zum anderen können Standardwerte in einer Konfigurationsdatei hinterlegt werden.

Beim Start des Programms mittels

```
lvstream -h
```

oder

```
lvstream --help
```

werden sämtliche Optionen mitsamt einer kurzen Erklärung angezeigt.

A.2.1. Verfügbare Optionen

–c, ––config_file — **name of config file to use**
(string, default »lvstream.conf«)
Der Name der Konfigurationsdatei, die vor dem Auswerten der restlichen Parameter gelesen werden soll.

–p, ––port — **number of IEEE1394 port to use**
(integer, range 0–15, default 0)
Falls mehrere Kameras an die IEEE-1394-Ports im System angeschlossen sind, kann mit dieser Option eine davon ausgewählt werden.
Diese Option ist nur verfügbar, wenn `lvstream` mit raw1394 als Capturequelle compiliert wurde.

–R, ––aud_samplerate — **resample audio to this rate (Hz)**
(integer, range 22050–48000, default 24000)
Die gewünschte Audio-Samplerate in Hertz.

–C, ––aud_channels — **use mono (1) or stereo (2) audio encoding**
(integer, range 1–2, default 1)
Dieser Wert gibt, an ob die Tonspur in Mono (Wert 1) oder Stereo (Wert 2) codiert werden soll.

–B, ––aud_bitrate
audio bitrate in kbps
(integer, range 8–320, default 40)
Die Audio-Datenrate in Kilobit pro Sekunde. Standardwert sind 40 kbps, da dies einen sinnvollen Kompromiß aus Bitrate und Qualität für die Sprachübertragung darstellt.

–Q, ––aud_quality
LAME algorithmic quality
(integer, range 0–9, default 7)
Der LAME-Qualitätsparameter kann Werte von 0 bis 9 annehmen. Niedrigere Werte bevorzugen die Codierqualität, höhere Werte die Geschwindigkeit.

–k, ––keyframe_interval
keyframe interval in frames
(integer, range 1–500, default 50)
Die maximale oder verbindliche Anzahl von Frames zwischen zwei I-Frames.

–f, ––force_keyframe_interval
strictly enforce static keyframe interval
(integer, range 0–1, default 1)
Steht dieser Wert auf 0, ist keyframe_interval als Maximalwert zu verstehen, XviD kann jedoch auch vorzeitig I-Frames einfügen, was bei der Anwendung für Vorlesungsstreaming jedoch eher unwahrscheinlich ist.
Steht dieser Wert auf 1, wird das Keyframe-Intervall garantiert eingehalten. Dies ist auch notwendig, um die Länge der Access Units konstant zu halten, da AUs immer an I-Frames beginnen.

–L, ––lo_bitrate
lo-res video bitrate in kbps
(integer, range 1–10000, default 300)
Die Datenrate des niedrig aufgelösten Videostreams in Kilobit pro Sekunde. Standard sind hier 300 kbps, so daß Video und Audio gemeinsam etwa die halbe Datenrate eines DSL-Anschlusses erreichen.

–H, ––hi_bitrate
hi-res video bitrate in kbps
(integer, range 1–10000, default 1500)
Die Datenrate des hoch aufgelösten Videostreams in Kilobit pro Sekunde. Hier sind 1500 kbps als Standardwert vorgegeben.

–a, ––au_length
length of an access unit in seconds
(integer, range 1–99999, default 10)
Die Länge einer Access Unit in Sekunden (*nicht* Frames wie beim Keyframe-Intervall).

–q, ––me_quality

motion estimation (ME) quality level

(integer, range 0–9, default 7)

Die Qualität der Bewegungsabschätzung für die geringere Auflösung kann in 8 Stufen von 1 (schlechteste Qualität, aber schnell) bis 8 (beste Qualität, aber langsam) eingestellt werden. Zusätzlich stehen mit den Stufen 0 und 9 zwei Optionen zur Verfügung, die vor allem für Vergleiche mit den Standard-Qualitätsstufen konzipiert sind.
Die Bedeutung der Qualitätsstufen ist in A.2.3 detailliert beschrieben.

–t, ––framedrop_tolerance

tolerated percentage of dropped frames before decreasing ME quality

(integer, range 0–100, default 10)

`lvstream` kann automatisch die Qualitätsstufe senken, wenn innerhalb einer Sekunde eine gewisse Prozentzahl an Framedrops vorliegt.

–s, ––sprite_file

uncompressed grayscale BMP file to overlay

(string, default: disabled)

Mit dieser Option kann der Name eines unkomprimierten Graustufenbildes im BMP-Format angegeben werden, das als »Senderlogo« ins Videobild eingeblendet werden soll. Das Bild sollte Abmessungen kleiner als 704x576 haben; zudem muß die Breite durch 8 teilbar sein.
Wird dem Programm während der Bearbeitung ein USR1-Signal geschickt, so lädt es die Bilddatei neu ein.

–S, ––sprite_origin

upper-left corner of sprite overlay

(string, default »0,0«)

Das Logo-Bild kann mit dieser Option an jede beliebige Stelle im sichtbaren Video-Bildausschnitt bewegt werden.

–d, ––deint_strength

deinterlacer strength

(integer, range 0–256, default 0)

Die Stärke des Deinterlacers entspricht 256 minus dem Schwellwert des Deinterlace-Filters aus Kapitel 5.2. Der Wert 0 (Standardeinstellung) bedeutet kein Deinterlacing. Empfohlen sind Werte von 240 bis 248.

–o, ––output_dir

directory to put segmented output in

(string, default: disabled)

In dem hier angegebenen Verzeichnis wird eine für das Peer-To-Peer-Netz geeignete Verzeichnisstruktur angelegt (siehe 8.3). Das Verzeichnis darf noch nicht existieren; `lvstream` weigert sich, bestehende Streams zu überschreiben.

–O, ––ogm_output **file name base to store unsegmented OGM output**
(string, default: disabled)
Ist zusätzlich oder zu Testzwecken die Ausgabe in normale OGM-Dateien erwünscht, die nicht in Access Units zerteilt sind, kann diese Option angegeben werden. Der Parameter für diese Option ist der Stammname der generierten Dateien; eine Eingabe wie `-O MyStream` erzeugt dabei zwei OGM-Dateien, MyStream_lo.ogm und MyStream_hi.ogm.

–b, ––benchmark **encode as fast as possible and print speed afterwards**
(integer, range 0–1, default 0)
Im Benchmark-Modus (–b 1) wird das Timing für die Echtzeitcodierung deaktiviert. Stattdessen werden die eingehenden Frames so schnell wie möglich aus der Capture-Pipe gelesen und encodiert. Am Ende wird eine kurze Statistik zur Verarbeitungsgeschwindigkeit ausgegeben.
Diese Option ist nur verfügbar, wenn `lvstream` für Pipe-Capture compiliert wurde.

A.2.2. Aufbau einer Optionsdatei

Sofern nicht auf der Kommandozeile anders angegeben, wertet das Programm noch vor dem Parsen der Parameter (außer –c) die Konfigurationsdatei lvstream.conf aus. Das Format dieser Datei ist sehr einfach gehalten: Jede Zeile repräsentiert einen Parameter. Dabei steht auf einer Zeile jeweils der lange Parametername (der Name der langen Kommandozeilenoption, jedoch ohne das führende ––), gefolgt von Whitespace und dem Parameterwert.

Als Beispiel folgt der Inhalt der lvstream.conf, die für die meisten Tests verwendet wurde:

```
au_length 10
framedrop_tolerance 10
sprite_file csntv.bmp
sprite_origin 560,517
deint_strength 244
```

A.2.3. Bedeutung der Qualitätsparameter

Die einzige Möglichkeit, den Kompromiß aus Rechenzeit und Bildqualität einzustellen, besteht in den Qualitätsparametern für die Bewegungsabschätzung. In `lvstream` kann damit die Genauigkeit der Bewegungsvektorsuche für die geringere Bildauflösung eingestellt werden; für die höhere Auflösung kommt (mit Ausnahme der Stufen 0 und 9) stets derselbe Verfeinerungsalgorithmus zum Einsatz.

Als Grundlage für die implementierten Qualitätsstufen dienten die Motion Estimation Presets der Video-for-Windows-Oberfläche für den XviD-Encoder.

Stufe 1 führt nur grundlegende Tests auf Makroblockebene durch und ist damit sehr schnell.

Ab **Stufe 2** findet die Bewegungskompensation auf halbe Pixel genau statt.

Stufe 3 führt eine leicht erweiterte Suche durch.

Stufe 4 ergänzt die Suche, indem es einen rautenförmigen Bereich systematisch absucht.

Ab **Stufe 5** werden Bewegungsvektoren für 8x8 Pixel große Blöcke bestimmt. In Stufe 5 selbst werden für diese feineren Blöcke jedoch noch keine komplexeren Suchmuster verwendet.

Stufe 6 führt die erweiterte und rautenförmige Suche auch für die 8x8-Blöcke ein.

Ab **Stufe 7** werden auch die Farbartinformationen zur Suche herangezogen.

Stufe 8 schließlich sucht nicht nur einen diamantenförmigen Bereich, sondern einen größeren quadratischen Bildbereich nach günstigen Bewegungsvektoren ab.

Zusätzlich existieren zwei Stufen, die das Konzept der Skalierung und Verfeinerung der Bewegungsvektoren auf die eine oder andere Weise umgehen. **Stufe 0** entspricht dabei der Stufe 1, nur daß für die hohe Codierungsauflösung die skalierten Bewegungsvektoren direkt verwendet und nicht mehr verfeinert werden. Das spart nochmals einen sehr großen Teil an Rechenzeit ein, jedoch leidet die Bildqualität merklich darunter (siehe auch 9.1 und 9.2).

Am anderen Ende der Skala befindet sich eine **Stufe 9**, die auf Stufe 8 basiert und zusätzlich die indirekte Bestimmung der Bewegungsvektoren komplett umgeht. Stattdessen wird auch für das hochauflösende Videobild eine vollständige Bewegungsabschätzung durchgeführt. Die beiden Encoder arbeiten somit völlig unabhängig voneinander.

A.3. Starten einer Encoding-Sitzung

Startet man `lvstream` ohne Parameter, so wird eine Codierung ohne Ausgabe durchgeführt. Erst durch die Optionen –o oder –O wird eingestellt, wohin die Ausgabe erfolgen soll.

Beispiel: Encoding via IEEE-1394

Wurde `lvstream` mit raw1394-Capture compiliert, so erfolgt der Start einer Sitzung einfach durch eine Kommandozeile wie beispielsweise

```
./lvstream -o MyStream
```

Dieser Befehl zeichnet mit den in der Konfigurationsdatei oder den Standardoptionen vorgegebenen Einstellungen in das Stream-Verzeichnis MyStream auf.

Beispiel: Encoding via Pipe

Um indirekt aus einer Pipe zu codieren, müssen einige weitere Vorkehrungen getroffen werden:

```
ttyp1$ mkfifo dv_in
ttyp1$ cat dv_in | ./lvstream -o MyStream
ttyp2$ dvgrab - >dv_in
```

Bei `dvgrab` handelt es sich um ein Tool, welches von einer DV-Kamera in Echtzeit Daten aufnimmt und in eine Pipe oder in ein File schreibt [3]. Der »Umweg« über die Named Pipe ist leider notwendig, da `lvstream` für die Initialisierung einige Zeit braucht, in der es noch keine Daten annimmt. Infolgedessen würden die Pipe-Puffer in der Zwischenzeit überlaufen, und das Programm würde defekte DV-Frames einlesen. Aus diesem Grund muß die eigentliche Datenquelle, sofern es sich nicht um eine einfache

synchrone Quelle wie `cat` beziehungsweise `dvcat` handelt, erst nach der Initialisierung von `lvstream` gestartet werden.

Mit dem mitgelieferten Tool `dvcat` können DV-Rohdaten in korrekter Geschwindigkeit aus DV-Filedumps »abgespielt« werden. Um aus einer im Vorfeld aufgenommenen Datei zu codieren, kommt eine Kommandozeile wie die folgende zum Einsatz:

```
./dvcat input.dv 25 | ./lvstream -o MyStream
```

Der zweite Parameter von `dvcat` ist dabei die gewünschte Framerate. Ein geringerer Wert verlangsamt den gesamten Codierprozeß. Dies ist sinnvoll, um `lvstream` auch auf langsameren Rechnern zu testen.

Programmablauf

Nach dem Start führt `lvstream` zunächst die genannten Initialisierungen durch. Danach findet der eigentliche Capturevorgang statt. Währenddessen wird auf der Konsole eine laufend aktualisierte Statuszeile angezeigt, die Auskunft über folgende Parameter gibt:

`0:00:33.68`	Die aktuelle Zeitmarke in Stunden/Minuten/Sekunden/Hundertstel.
`AU#4`	Die Nummer der aktuell codierten Access Unit.
`VBuf:1`	Die Anzahl momentan verwendeter Video-Ringpuffereinträge.
`Q:7`	Die Qualitätsstufe der Bewegungsabschätzung.
`drops: ...`	Die Anzahl der verlorenen Pakete oder Frames für Capture, Audio und Video. Das Auftreten von Video-Framedrops ist unkritisch, sobald jedoch DV- oder Audio-Framedrops gemeldet werden, ist der ausgegebene Strom mit hoher Wahrscheinlichkeit defekt.
`(0.0% = 25.0 fps)`	Der prozentuale Anteil gedroppter Videoframes in der letzten Sekunde und die daraus resultierende effektive Framerate.

Eine typische Codierungssitzung gibt im Ganzen etwa folgende Bildschirmmeldungen aus:

```
KeyJ's Live Stream Provider, version 1.0 (PAL) (pipe capture)
Initializing ...
audio_init(): using LAME 3.96.1
video_init(): using xvid-1.1.0-beta1
video_init(): creating lo-res XviD instance
video_init(): creating hi-res XviD instance
Starting Threads ...
Capturing, press ^C to quit ...
0:00:33.68 AU#4 | VBuf:0 Q:7 | drops: DV:0 A:0 V:0 (0.0% = 25.0 fps)
Capture done, waiting for all threads to finish ...
All threads finished, cleaning up ...
Done.
```

B. Digitale Repräsentation von Videobildern (ITU-R BT.601)

Obwohl im vorgestellten System die Videodaten von der Kamera bis zur Ausgabe komplett digital bearbeitet und übertragen werden, basiert der zugrundeliegende Standard [13] für die Bildabmessungen, Seitenverhältnisse, Wiederholraten und Farbformate auf dem analogen Übertragungssystem. Im folgenden Abschnitt soll daher am Beispiel des in Europa gebräuchlichen PAL-B/G-Systems kurz darauf eingegangen werden, die die Systemparameter zustandekommen.

B.1. Abtastung der Bildsignale

Im europäischen B/G-Fernsehsystem werden pro Sekunde 50 Halbbilder zu je $312\frac{1}{2}$ Zeilen übertragen. Zur Vereinfachung der Berechnungen kann man annehmen, daß 25 Vollbilder zu 625 Zeilen übertragen werden. Von diesen 625 Zeilen sind 576 »aktiv« und enthalten sichtbare Bilddaten. Die restlichen 49 Zeilen werden während des vertikalen Strahlrücklaufs der Anzeigeröhre übertragen, aber nicht dargestellt.

Jede Zeile hat eine Dauer von 64 μs:

$$\frac{1}{25\ \text{Hz} \times 625} = 64\ \mu\text{s}$$

Von diesen 64 μs enthalten wiederum nur ungefähr 52 μs aktive Bildinformationen; in den verbleibenden 12 μs findet der horizontale Strahlrücklauf statt.

Im Standard ITU-R BT.601 wird nun definiert, wie die Abtastung der Bildsignale erfolgen soll. Dabei ist für das Luminanzsignal (Helligkeitssignal) eine Samplerate von 13,5 MHz vorgesehen. Somit entstehen für jede Zeile 864 Abtastwerte:

$$13,5\ \text{MHz} \times 64\ \mu\text{s} = 864$$

Davon enthalten 702 Werte aktive Bildinformationen:

$$13,5\ \text{MHz} \times 52\ \mu\text{s} = 702$$

Da die Position der aktiven Bildinformationen innerhalb der Zeile je nach Quelle leicht variieren kann, definiert BT.601 einen »sicheren« Wert von 720 Samples pro Zeile. Systeme wie DV, welche die Samples der Austastlücken nicht digital mitübertragen, arbeiten daher meist mit einer Bildauflösung von 720x576.

Diese 720 Samples pro Zeile sind jedoch eine »pessimistische« Schätzung – effektiv sind schließlich nur 702 Pixel mit Bildinhalt gefüllt, es bleiben 18 Pixel **Overscan**. Daher ist ein weiteres Format gebräuchlich, nämlich 704x576. Es beinhaltet lediglich 2 Pixel Overscan, und hat zudem die für die Weiterverarbeitung in digitalen Systemen überaus günstige Eigenschaft, durch große Zweierpotenzen teilbar zu sein ($704 = 11 \times 64$).

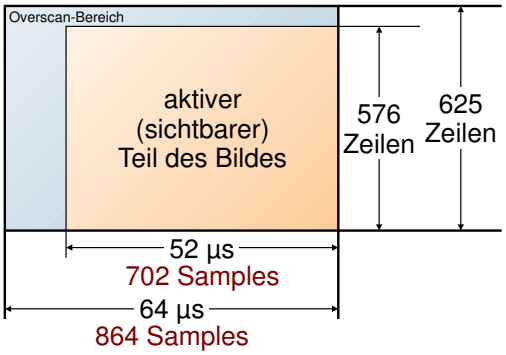


Abbildung B.1.: Struktur und Abtastung eines Videobilds

Diese beiden Formate können äquivalent verwendet werden, und sind einfach ineinander überführbar: Von 720x576 nach 704x576 werden an beiden Seiten des Bildes jeweils 8 Pixel verworfen, in der umgekehrten Richtung werden jeweils 8 schwarze Pixel angefügt.

`lvstream` macht von dieser Eigenschaft Gebrauch, denn die DV-Kamera liefert ein Bild mit 720x576 Bildpunkten, eine Auflösung von 704x576 ist jedoch günstiger: Da das angelieferte Bild für die niedrige Auflösungsstufe auf halbe Größe skaliert wird (352x288), sollten auch in dieser geringen Auflösung dennoch Höhe und Breite ganzzahlige Vielfache von 16 sein, so daß anschließend nur vollständige Makroblöcke codiert werden müssen. 720 ist jedoch nur durch 16, nicht aber durch 32 teilbar, weswegen dem 704er Format der Vorzug gegeben wurde.

B.2. Seitenverhältnis

Wie allgemein bekannt ist, werden die Bildsignale typischerweise für ein Anzeige-Seitenverhältnis von 4:3 übertragen. Die gebräuchlichen digitalen Bildgrößen 704x576 und 720x576 weisen jedoch offensichtlich nicht dieses Seitenverhältnis auf. Es handelt sich daher um sogenannte **anamorphe** Formate, bei denen die Bildpunkte keine quadratische Form haben.

Die korrekte Größe des Bildes bei quadratischen Pixeln beträgt 768x576 (von $768 = 576 \times \frac{4}{3}$). Das Seitenverhältnis jedes einzelnen Pixels in den anamorphen Formaten ergibt sich durch das Verhältnis aus diesem »gewünschten« Wert und der Anzahl der aktiven Samples in jeder Zeile (702). Es beträgt folglich 768:702 = 384:351. Dieses Verhältnis ist recht ungünstig, daher wird im Allgemeinen ein »runderer« Wert von 12:11 angenommen.

Als Konsequenz dessen müssen Videobilder in 704x576 bei der korrekten Anzeige auf einem Display mit quadratischen Pixeln auf 768x576 entzerrt werden ($704 \times \frac{12}{11} = 768$), und Bilder in 720x576 gar auf etwa 786x576 ($720 \times \frac{12}{11} = 785,45$).

Um dem Wiedergabegerät oder der Wiedergabesoftware zu signalisieren, daß anamorphes Material übertragen wird, finden sich in vielen Video-Kompressionsformaten und teilweise auch in Containerformaten Flags in den Headern, in denen meist ein Pixel-Seitenverhältnis (Pixel Aspect Ratio, PAR) oder ein Bild-Seitenverhältnis (Display Aspect Ratio, DAR) gespeichert wird.

Auch `lvstream` verwendet eine solche Signalisierung in den erzeugten MPEG-4-Datenströmen, um eine korrekte Wiedergabe zu gewährleisten. Leider wertet noch längst nicht jede Player-Software diese Informationen auch korrekt aus.

B.3. Darstellung von Farbinformationen

In B.1 wurde zunächst nur von einer Abtastung der Helligkeitsinformationen (Luminanz) ausgegangen. Zusätzlich existieren bei Farbfernsehsystemen zwei Kanäle mit Farbartsignalen (Chrominanz), die ebenfalls bei BT.601 abgetastet werden. Dies erfolgt jedoch gegenüber dem Luminanzsignal mit der halben Abtastrate (6,25 MHz). Auf digitaler Seite kommt dies einer Reduktion der Auflösung um 50% gleich: Die beiden Farbartbilder sind jeweils nicht 704x576, sondern nur 352x576 Bildpunkte groß. Diese Art der Darstellung nennt man **4:2:2**-Unterabtastung oder -**Subsampling**.

Consumer-DV und die meisten Videokompressionsverfahren gehen noch einen Schritt weiter und führen zusätzlich eine Reduktion in der vertikalen Richtung durch: Beim **4:2:0**-Subsampling werden die Chrominanzkanäle gegenüber dem Luminanzkanal sowohl in der Breite als auch in der Höhe auf die Hälfte reduziert, aus 704x576 wird somit 352x288. Dem Betrachter bleibt dieser Eingriff im Allgemeinen verborgen, da sich die resultierende minimale Verschlechterung der Bildqualität nur an scharf abgegrenzten farbigen Kanten äußert.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[1] Daniel Schreiber: Peer-To-Peer-Videostreaming. Diplomarbeit, Technische Universtität Chemnitz, 2005.

[2] libraw1394, libdv1394: IEEE 1394 for Linux.
http://www.linux1394.org/

[3] Arne Schirmacher, Dan Dennedy: Digital Video for Linux.
http://kino.schirmacher.de/

[4] FFmpeg Multimedia System. (libavcodec und libavformat)
http://ffmpeg.sourceforge.net/

[5] libdv: The Quasar DV Codec.
http://libdv.sourceforge.net/

[6] MPlayer - The Movie Player.
http://www.mplayerhq.hu/

[7] The XviD codec.
http://www.xvid.org/

[8] The LAME Project (LAME Ain't an MP3 Encoder).
http://lame.sourceforge.net/

[9] Advanced Systems Format (ASF) Specification.
http://www.microsoft.com/windows/windowsmedia/format/asfspec.aspx

[10] Olivier Avaro, Alexandros Eleftheriadis, Carsten Herpel, Niels Rump, Vishy Swaminathan, Javier Zamora, Michelle Kim: MPEG-4 Systems Frequently Asked Questions. ISO/IEC JTC1/SC29/WG11 N4291, Juli 2001.
http://www.chiariglione.org/mpeg/faq/mp4-sys/mp4-sys.htm

[11] Silvia Pfeiffer: The Ogg Encapsulation Format Version 0. RFC 3533, Mai 2003.
http://www.ietf.org/rfc/rfc3533.txt

[12] Ogg Framing Format documentation. Juli 2002.
http://www.xiph.org/ogg/doc/

[13] Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios. ITU-R Recommendation BT.601-5, Oktober 1995.

Selbständigkeitserklärung

Ich erkläre hiermit, daß ich die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Einzige Ausnahme ist Kapitel 2, welches gemeinsam mit Daniel Schreiber, dem Autor der begleitenden Arbeit [1], verfaßt wurde.

Chemnitz, den 31. Mai 2005