Informatik-Projekt 8: Progressive Movie File

Stefan Weber

Frühlingssemester 2011

Ein Projekt im Studiengang Master Of Science in Engineering

Fachhochschule Nordwestschweiz, Hochschule für Technik
Institut für Mobile und Verteilte Systeme
Steinackerstrasse 5, CH-5210 Windisch

Projektbetreuer: Prof. Dr. Christoph Stamm

Abstract

Dies ist die Dokumentation des Informatik-Projekts 8 von Stefan Weber, am Master-Studiengang für Informatik (MSE) der Fachhochschule Nordwestschweiz.

Das Progressive Graphics File (PGF) Bildformat stellt das Fundament dieses Projekts dar. Mit seiner progressiven Kompressionstechnik ist es das ideale Format für eine adaptive Video Streaming Applikation. Im vergangenen Semester hat der Studierende im Rahmen des MSE-Projekts IP7 die Basis für dieses Projekt gelegt. Die Basiskomponenten Encoder und Decoder sowie der Streaming-Kontext sind für das auf Einzelbild-Kompression basierende bisherige *MPGF*-Codec (Motion Progressive Graphics File) implementiert und sollen als Basis für das aktuelle Projekt genutzt werden.

In der vorliegenden Arbeit wird die bisherige Version des Video-Codec (MPGF) optimiert für die Applikation in Live-Streaming. Es werden erweiterte Video-Kompressionstechniken vorgestellt und eine Architektur für das *Progressive Movie File* (PMF) vorgeschlagen.

Der Autor

Stefan Weber hat nach seiner Lehre als Mediamatiker von 2007-2010 den Bachelor-Studiengang für Informatik mit Spezialisierung in *Information Processing and Visualization* an der Fachhochschule Nordwestschweiz absolviert. Im Anschluss ist er im Herbst 2010 in den Vollzeit MSE-Studiengang für Informatik eingetreten, in dessen Rahmen dieses Projekt stattfindet.

Inhaltsverzeichnis

1 Einführung **3**

1.1 Aufgabenstellung 3

2 Motion-PGF **4**

2.1 Live-MPGF 4

2.1.1 Wavelet-Pyramide in PGF 4

2.1.2 Anpassung von LPGF 5

2.1.3 Performance-Gewinn 5

2.1.4 Messdaten 7

2.2 PMF1: LPGF mit Tiles anstatt Levels 8

2.2.1 Untersuchung PMF1 Performance 9

3 Multithreading **13**

3.1 Media Foundation asynchron 13

3.2 Auslagern der Verarbeitung 14

3.3 Multi-Threaded Decoder Performance 15

4 Bewegungsschätzung **17**

5 PMF Video Format **18**

5.1 Differenzbild Architektur 18

5.2 PMF Architektur Vorschlag 19

5.3 Live-Streaming 21

6 Reflexion **23**

7 Ehrlichkeitserklärung **24**

Literaturverzeichnis **25**

1 Einführung

Abbildung 1 illustriert den Kontext, in welchem die vorliegende Arbeit stattfindet.

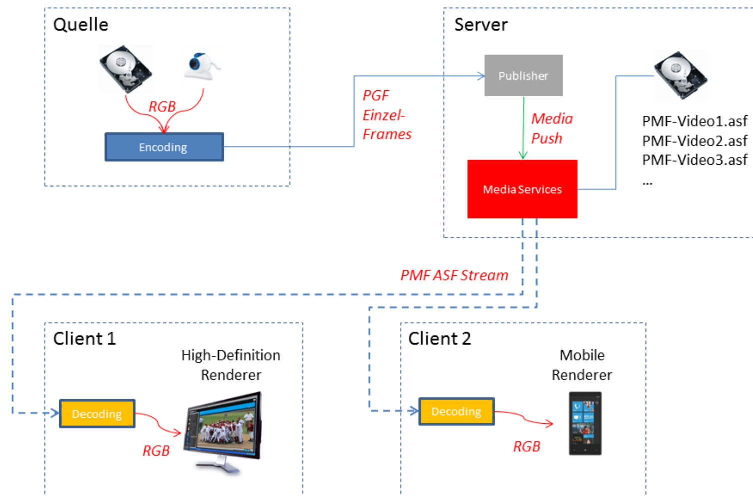

Abbildung 1: Eine Übersicht über den Kontext der vorliegenden Arbeit

In diesem Semester konzentriere ich mich auf die Arbeit des Encodings (Absender) und Decodings (Empfänger). Einerseits werden weitere Verbesserungen am bisherigen Format diskutiert und implementiert (Kapitel 2 bis 3). Dabei liegt ein Schwerpunkt immer darauf, die erhaltenen Resultate durch empirische Messungen belegen zu können und daraus Schlussfolgerungen zu erhalten. Andererseits werden erweiterte Techniken der Videokompression eingeführt und eine theoretische Architektur für das Codec PMF aufgebaut (Kapitel 4 bis 5).

1.1 Aufgabenstellung

Um für die weiteren Arbeiten gut vorbereitet zu sein, sind die aktuellen Motion-PGF Codecs zu überarbeiten (Performance, Flexibilität und Robustheit erhöhen). Studieren Sie bereits realisierte Multiscale-Video Architekturen und entwickeln Sie ein Konzept für eine Multiscale-Video-Architektur basierend auf PGF. Realisieren Sie erste Teile dieses Konzepts in Absprache mit Ihrem Betreuer und dokumentieren Sie Ihr Vorgehen in einem Projektbericht und einem Paper-Draft für eine wissenschaftliche Publikation.

2 Motion-PGF

Im Informatik-Projekt 7 realisierte ich unter Verwendung von Einzelbild-Kompression des *Progressive Graphics File (PGF)* [1, 2] das Video-Codec *Motion PGF (MPGF)*. Dabei wird jedes Video-Frame komplett codiert als Einzelbild in den Videostream eingespiesen. In einem ersten Teil dieses Projektes werden weitere Verbesserungen an MPGF vorgenommen. Insbesondere die Performance und die Code-Struktur sollen optimiert werden. Dieses Kapitel beschreibt einige der Massnahmen, welche die genannten Eigenschaften verbessern sollen.

2.1 Live-MPGF

Um das Szenario des Live-Streaming besser angehen zu können, wird der bestehende MPGF Codec angepasst. Das neue Akronym LPGF (*Live-MPGF*) steht für das veränderte Format. In LPGF soll das progressiv aufgebaute PGF Bildformat ausgenutzt werden, um ein Videoframe in mehreren Schritten verarbeiten zu können:

2.1.1 Wavelet-Pyramide in PGF

PGF nutzt zur Bildkompression die Wavelet-Theorie. Dabei werden die einzelnen rekursiven Wavelet-Skalen bzw. Detailstufen in PGF als *Levels* bezeichnet. Abbildung 2 illustriert die Wavelet-Pyramide mit 3 Levels.

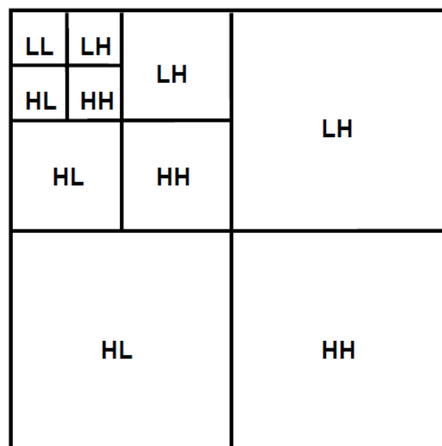

Abbildung 2: Die DWT-Pyramide eines PGF Bildes (Quelle: [1]).

Wie in der Wavelet-basierten Bildkompression üblich, werden die Pyramiden-Teile mit den Kürzeln H (*High Pass* oder Details) und L (*Low Pass* oder Approximation) identifiziert. So werden auf jedem Level (oder *Skala*) LL, LH, HL und HH rekursiv aus der nächst höheren Approximation (LL) errechnet. Dabei gilt im Zweidimensionalen der erste Buchstabe für die Zerlegung der Zeilen und der zweite für die Spalten. So ist LL deshalb das Vorschaubild, weil in Zeilen-

und Spaltenrichtung die Approximation berechnet wird. LH enthält horizontale, HL vertikale und HH diagonale Strukturen.

Weitere Informationen über Wavelet-basierte Bildkompression sind in [3, 4] zu finden. Die Anwendung in PGF ist in [1, 2] erläutert.

2.1.2 Anpassung von LPGF

Abbildung 3 zeigt die erste Veränderung schematisch auf: Bei LPGF werden die Videoframes nicht mehr als komplette Einzelbilder übertragen, sondern die Levels der Bilder werden genutzt. Sobald der Absender eine Skala fertig kodiert hat, kann er diesen Level abschicken. Mit jedem empfangenen Level kann der Empfänger dann bereits eine Skala dekodieren, während der Absender noch am nächsten Level des gleichen Bildes arbeitet.

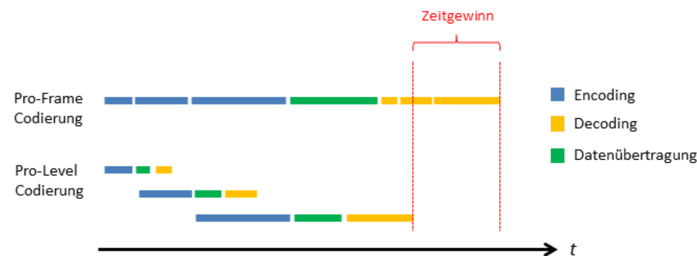

Abbildung 3: Skizze der Version LPGF

Abbildung 3 macht deutlich, warum die Aufteilung in einzelne *Levels* die Performance des Codecs je nach Anwendungsfall erheblich verbessert. Sobald Encoding und Decoding der Daten gleichzeitig auf entfernten Rechnern (z.B. Live-Streaming) oder separaten CPU-Kernen stattfindet, kann diese Version des Codec seine Stärken ausspielen.

2.1.3 Performance-Gewinn

Dass die beschriebenen Änderungen auch praktisch einen Vorteil für die Performance mit sich bringen wird in diesem Abschnitt erläutert.

Die im folgenden erklärten Zeitgewinne, insbesondere der Einbezug des Netzwerks (Abbildung 4 und 5) sind theoretisch berechnet. Die Werte der Prozessierungsdauer resultieren demgegenüber tatsächlich aus empirischen Messungen. Als Video wurde eine Sequenz der Grösse 640x360 verwendet, der Rechner ist ein Lenovo Z61m Notebook (Intel Core 2 CPU, 2 Ghz, 3 GB RAM, Windows 7 32-Bit). Die Prozessierungsdauer ist immer zu relativieren, insbesondere mit Netzwerkübertragung können andere Videogrössen oder Rechner zu unterschiedlichen Verhältnissen führen.

Abbildung 4 illustriert die gemessenen Zeiten für jeweils 1 durchschnittliches Video-Frame. Die Messungen in LPGF sind jeweils in Encoding und Decoding

der Levels sowie in zwei Preprocessing und einem Postprocessing Schritt aufgeteilt. Der Preprocessing Schritt enthält das Instanzieren des PGF-Bildes für das aktuelle Frame (blau) und den pro Bild nur einmal notwendige Initialisierungsschritt, der unter anderem das Schreiben des Header-Objekts von PGF beinhält (rot). Der Postprocessing Schritt läuft im Decoder ab und enthält das Abrufen des fertigen Bitmaps aus dem decodierten PGF Objekt.

Die erste Grafik der Abbildung 4 macht ersichtlich, dass beide Versionen, LPGF und MPGF, in einer Single-Threaded Umgebung, wo Encoder und Decoder strikt sequentiell arbeiten müssen, ähnlich gut performen. Bei MPGF fällt ein kleiner Overhead weg, der durch die Aufteilung in mehrere Pakete anfällt. Zudem kann das PGF-Bildformat den Speicher etwas besser ausnutzen, wenn das ganze Bild auf einmal codiert wird, wie das bei MPGF der Fall ist. Dies hat zur Folge, dass bei LPGF im Schnitt rund 0.1% mehr Daten verarbeitet werden müssen.

Die zweite Illustration der Abbildung 4 zeigt den Ablauf bei Encoding und Decoding auf zwei Rechnern (oder CPU-Kernen) ohne Berücksichtigung einer Netzwerkübertragung. Damit kann durch Verwendung von LPGF im Durchschnitt bereits 15% der bei MPGF nötigen Zeit pro Frame eingespart werden, da der Encoding-Thread nicht auf die Fertigstellung des Decodings warten muss.

Die beiden Grafiken der Abbildung 5 berücksichtigen zusätzlich noch die Übertragungszeit über ein Netzwerk mit 100 bzw. 20 Mbit/s Übertragungsrate. Die Übertragungszeit wurde dabei mit den durchschnittlichen Level-Grössen berechnet. Am Beispiel ist bei 100 Mbit/s eine Steigerung der Performance um 19%, bei noch 20 Mbit/s um 22%, zu erkennen.

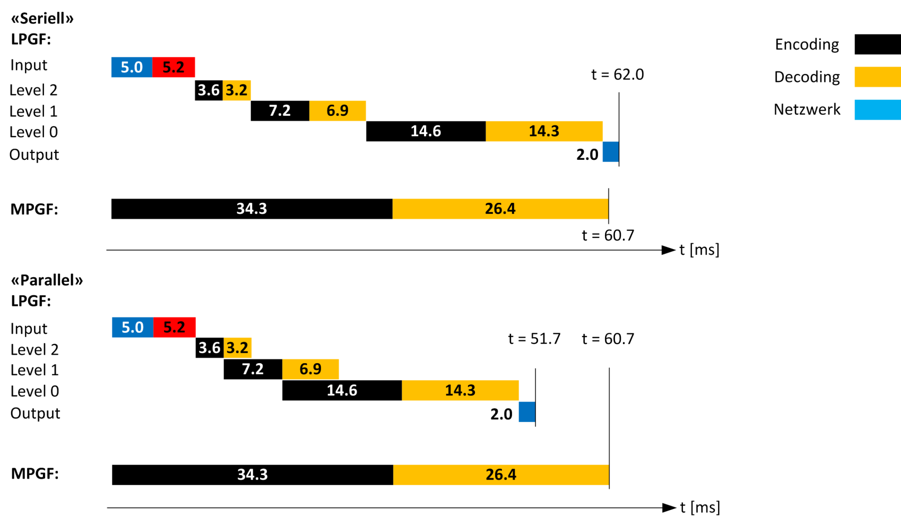

Abbildung 4: Vergleich zwischen MPGF und LPGF

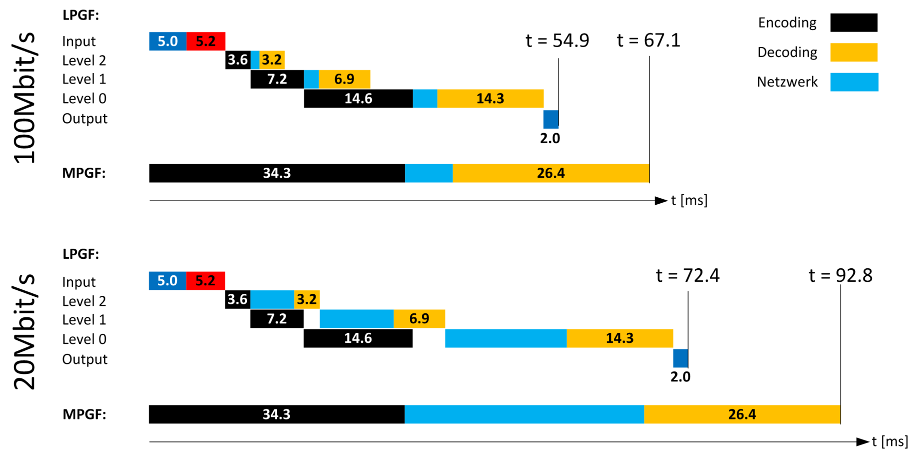

Abbildung 5: Vergleich zwischen parallelen MPGF und LPGF Versionen mit theoretischer Netzwerk-Übertragung über 100 bzw. 20Mbit/s

2.1.4 Messdaten

Die folgende Tabellen listen die Mittelwerte der erhobenen Messdaten auf, bei LPGF für die einzelnen Levels dargestellt. Die Werte sind in Millisekunden angegeben. In den Illustrationen der Abbildung 6 sind zudem die Werte für minimale, mittlere und maximale encodierte Frame-Grösse eingezeichnet.

Encoding:

Modus	Pre-P.	Levels	Total
640x360 LPGF	5.0	*(5.2)* + 3.6 + 7.2 + 14.6	**35.5**
640x360 MPGF			**34.3**
1280x720 LPGF	20.6	*(22.9)* + 4.1 + 9.4 + 20.9 + 29.0	**106.8**
1280x720 MPGF			**108.3**

Die hohen Werte beim ersten Level des LPGF-Encodings (in Klammern) rühren von dem in Abbildung 4 rot eingefärbten Teil her, welcher eine Vorverarbeitung durch PGF repräsentiert, unter Anderem das Schreiben des File-Headers und die Durchführung der Diskreten Wavelet-Transformation. Die Spalte *Pre-Processing* gibt an, wie viel Zeit der Encoder für das einmalige Empfangen der zu codierenden Rohdaten benötigt.

Decoding:

Modus	Levels	Post-P.	Total
640x360 LPGF	3.2 + 6.9 + 14.3	2.0	**26.4**
640x360 MPGF			**26.4**
1280x720 LPGF	4.1 + 9.1 + 20.7 + 35.9	9.2	**79.0**
1280x720 MPGF			**76.0**

Die Spalte *Post-Processing* in der Decoder-Tabelle gibt die gemessene Zeit an, welche für das Versenden der unkomprimierten Ausgangsdaten benötigt wurden.

Aus diesen Messdaten wird ersichtlich, dass die neue Version LPGF – wenn die reine Rechenzeit betrachtet wird – aus den genannten Gründen ähnliche oder oft sogar höhere Rechenleistung erfordert als die MPGF Version. Dieser kleine Overhead wird jedoch kompensiert durch die Verwendung einer parallelisierbaren Architektur, wie in Abbildung 4 gezeigt.

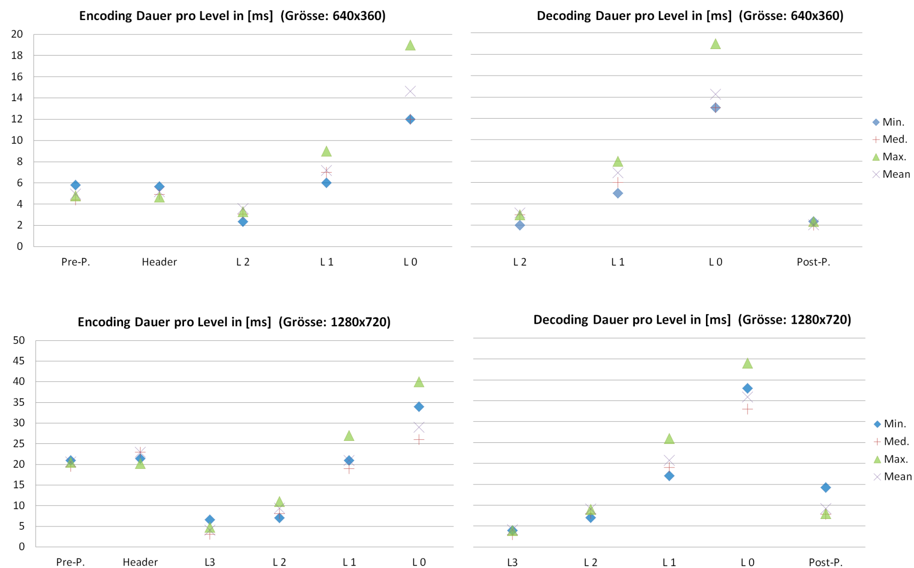

Abbildung 6: Messwerte pro Bild-Level für Encoding (links) und Decoding.

2.2 PMF1: LPGF mit Tiles anstatt Levels

Dank der Architektur von PGF ist der in den vorherigen Abschnitten diskutierte Effekt noch weiter denkbar: Die bisher übermittelten Pakete mit PGF-Levels können durch die Aufteilung in kleine Bildausschnitte (im folgenden *Tiles* genannt) noch weiter verkleinert werden – der Zeitgewinn, wie in Abbildung 3 illustriert, sollte noch weiter optimiert werden können. Wir wollen diese Version *PMF1* (Progressive Movie File, Version 1) nennen.

Es wird also kein komplettes Level mehr versandt, sondern nur ein Ausschnitt: Tiles sind in PGF speziell definierte Bildausschnitte (*Region Of Interest*) der Wavelet-*Bildpyramide* (siehe Abschnitt 2.1.1). Per Definition ist ein Tile in PGF immer so gross wie die kleinste Approximation (*Vorschaubild*) der Pyramide.

In Abbildung 2 wurde der Aufbau der Bildpyramide dargestellt. Dabei stellt das linke obere LL-Rechteck jeweils das Vorschaubild (die *Approximation*) dar, welches gerade auch die Grösse der Tiles in PGF definiert. Die Grafik macht ersichtlich, dass die Anzahl Tiles (t) folgendermassen im Verhältnis zu den Anzahl Levels n steht: $t = 4^n$. Im gezeigten Beispiel sind 3 Levels vorhanden, es sind demnach $4^3 = 64$ Tiles in der Grösse des LL-Blocks verfügbar.

Nun wollen wir die Performance von PMF1 untersuchen. Abbildung 7 zeigt, dass die Version PMF1 (als Quadrate dargestellt) in der Messung deutlich schlechter performt als die Version LPGF – die rot gestrichelte Linie im Diagramm zeigt

die Abspieldauer des Videos an. PMF1 lässt demnach keine ruckelfreie Wiedergabe des Videos zu. Die Gründe für dieses schlechte Abschneiden wollen wir im nächsten Abschnitt untersuchen.

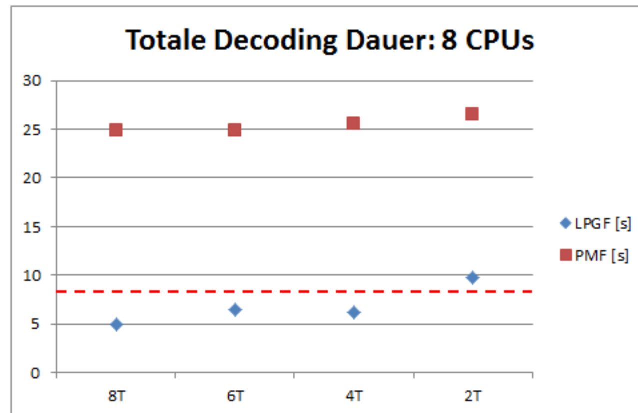

Abbildung 7: Vergleich von LPGF und PMF1.

2.2.1 Untersuchung PMF1 Performance

Um das schlechte Abschneiden von PMF1 erklären zu können, wird eine Applikation entwickelt, welche die Situation des Decodierens simuliert. Es existieren die Aufgaben "Lesen", "Schreiben" und "Decodieren".

Die Resultate der Simulation sind als Untergrenze (*Lower Bound*) zu verstehen, da nur die effektive Prozessierungsdauer eingerechnet wird – entstehender Overhead in Media Foundation durch das Handling der Media Session wird nicht berücksichtigt.

Setup

Erste Variable ist die Grösse des Samples. Messungen des Videos "Wildlife" zeigen eine durchschnittliche Grösse von 3060 Bytes, mit einer Standardabweichung von 6057. Diese Samplegrösse ist exponentiell verteilt, wie Abbildung 8 zeigt.

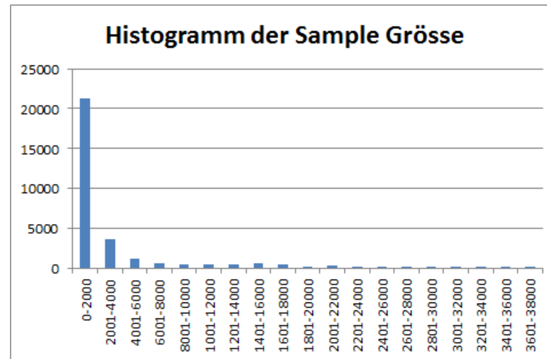

Abbildung 8: Histogramm der Samplegrössen des PMF1-codierten Videos "Wildlife".

Die Lese- und Schreibdauer wird mit Hilfe eines Festplattenbenchmarks approximiert, wobei die Lesedauer von der Grösse des zu lesenden Samples abhängt. Geschrieben wird ein ganzes, unkomprimiertes, Videoframe (konstante Grösse).

Für die Dauer des Decodierens werden Messdaten verwendet, in Abhängigkeit der Sample-Grösse. Abbildung 9 zeigt ein Histogramm der gemessenen Decoding-Dauern pro Byte. Auf dieser Grundlage wird eine exponentiell verteilte Zufallszahl um den (gemessenen) Mittelwert von 0.001ms mit Standardabweichung von 0.02 generiert. Diese Decoding-Dauer pro Byte wird dann für die Simulation der Dauer eines Samples mit der Sample-Grösse multipliziert.

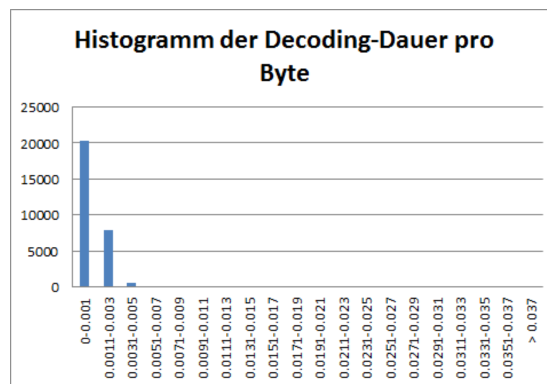

Abbildung 9: Histogramm der Decoding-Dauer pro Byte des Videos "Wildlife".

Vorgehen

In der Applikation wird ein primitives Scheduling simuliert. Die Applikation teilt die Decoding-Arbeit so gut wie möglich auf die zur Verfügung stehenden Threads auf, mit folgenden Einschränkungen:

- Ein einzelnes Frame muss sequentiell verarbeitet werden. Alle Samples dieses Frames werden deshalb auf dem gleichen Thread decodiert (ein Frame ist somit an einen Thread gebunden). Dies weil PGF eine sequentielle

Verarbeitung der Bildsamples vorschreibt.

- Die Lese- und Schreiboperationen werden sequentiell ausgeführt, da die Festplatte im einfachsten Fall nur an einer Position lesen oder schreiben kann. Dazu wird für IO-Operationen ein Thread reserviert.

Beim Start der Applikation wird ein Sweep-Line ähnliches Vorgehen gestartet: Für jeden Event-Punkt wird eine Callback-Funktion aufgerufen, dort wird das weitere Vorgehen entschieden. So wird beispielsweise beim Ende des Lesens von Sample 1 der Lese-Start von Sample 2 und das Decodieren von Sample 1 auf einem anderen Thread in die Zeitachse eingefügt.

Die Simulation kann mit der Geschwindigkeit des zu simulierenden Rechners konfiguriert werden. In dieser Dokumentation betrachten wir zwei Konfigurationen, "Laptop" und "Workstation", welche sich stark unterscheiden. So hat der simulierte Laptop nur 2 CPU-Kerne und eine geringere Leistung von CPU und Harddisk. Die Workstation hingegen bietet 8 virtuelle CPU-Kerne und ein RAID-0.

Die erhaltenen Resultate betrachten wir als eine untere Grenze, ein *Lower Bound* für den Decoding-Vorgang des Gesamtvideos. Weiter wird ersichtlich, welche Threads wie lange warten müssen (Idle-Zeit). Daraus lassen sich Schlüsse auf das Verhalten des Decoders ziehen.

Resultate

Die Simulation zeigt nun die bremsenden Bottlenecks der Verarbeitung auf. Beispielsweise wird ersichtlich, dass bei der Workstation alle Threads gleichmässig ausgelastet sind (fast keine Idle-Zeiten). Mit den Einstellungen für den Laptop hingegen ist das Decoding im Verhältnis zum Input/Output viel langsamer – daraus resultiert Idle-Zeit auf dem IO-Thread. Der folgende Output der Simulation für die Performance von Laptop und Workstation illustriert dies:

Laptop		**Workstation**	
Thread	Idle-Zeit	Thread	Idle-Zeit
IO	*87.1s*	*IO*	*0.5s*
1	0.0s	1	0.1s
2	0.0s	2	0.1s
		3	0.1s
		4	0.1s
		5	0.1s
		6	0.0s
		7	0.1s
		8	0.0s
Totalzeit:	**124.5s**	Totalzeit:	**12.2s**

Der Grund für die Wartezeit des IO-Threads bei der Laptop-Simulation sind die geringere Decoding-Performance, verstärkt durch die kleine Anzahl Threads.

Eine weitere Differenz entsteht aus dem Vergleich zwischen den Performances von Laptop und Workstation: Ist die Workstation bei Festplattenzugriffen bereits um Faktor 3 schneller, ist der Unterschied bei der Decoding-Dauer nochmals höher (Faktor 5) – und zwar pro Thread.

In der Simulation lassen sich die Anzahl Samples pro Frame (und proportional

deren Grösse) zwar variieren. Allerdings verändert sich dadurch die Performance nicht, da in der Simulation kein Verwaltungs-Overhead pro Sample berücksichtigt wird: Die nötige Rechenzeit besteht lediglich aus der Multiplikation von benötigter Zeit pro Byte, der Grösse der Samples und der Anzahl Samples im Video.

Die Gesamtzeit von 125 Sekunden ist nun ein erster Ansatz für die Untergrenze des Video-Decodings mit PMF1 für die Laptop-Konfiguration. Eine Messung der effektiven Decoding-Dauer ergab 132 Sekunden, also lediglich 5% höher – die real erreichte Geschwindigkeit liegt bereits nahe an der möglichen Untergrenze.

Nicht so bei der Workstation-Konfiguration: die theoretische Untergrenze von 12 Sekunden wird in der realen Messung beinahe verdoppelt – 23 Sekunden dauert das Decoding des Gesamtvideos. Ein möglicher Grund für diese Differenz ist der bereits erwähnte Input/Output-Overhead für das Handling der Daten, welcher in die Simulation nicht adäquat einfliesst. Dort hat der IO-Thread bei der Laptop-Konfiguration eine hohe Idle-Zeit, welche in Realität noch ausgenutzt werden kann – im Gegensatz zu der Workstation-Konfiguration.

3 Multithreading

Moderne Rechner haben mehrere CPU-Kerne. Bessere Leistung wird heutzutage nicht mehr durch höhere Taktraten erreicht, sondern durch die Verwendung mehrerer CPU-Kerne. Um eine optimale Performance zu erreichen, muss diesem Umstand Rechnung getragen werden.

In diesem Kapitel wollen wir das Codec durch die Einführung von Asynchronität und multi-threading für mehrere CPU-Kerne optimieren. Als Erstes wollen wir einige Gedanken zur Implementation betrachten. Danach werden konkrete Messdaten analysiert.

Die grundsätzliche Idee wird in der Skizze der Abbildung 10 illustriert. Die frei gewordenen Threads 1 und 3 nehmen die Arbeit sofort wieder auf (Frames #5 bzw. #6). Es wird sofort klar, dass die aufgezeigte Codec-Version auf einem Rechner mit mehreren Prozessorkernen besser performen dürfte, als eine synchrone Version ohne Ausnutzung mehrerer Threads.

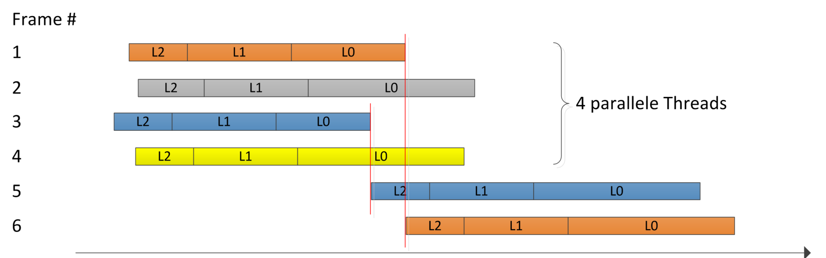

Abbildung 10: Eine Skizze zum Prinzip der asynchronen Frame-Verarbeitung.

Das parallele Verarbeiten von Frames hat jedoch auch einen Nachteil: Wenn mit Bewegungskompensation gearbeitet werden soll, ist unter Umständen das Referenzframe noch nicht fertig, da es in einem anderen Thread verarbeitet wird. Diesem Umstand ist bei einer Implementation der Bewegungskompensation Rechnung zu tragen, zum Beispiel in dem eine ganze *GOP* (Group Of Pictures, die Gruppe von Frames bis zum nächsten Key Frame) auf einem Thread sequentiell verarbeitet wird. Dies würde "inter-Thread" Referenzen verhindern.

3.1 Media Foundation asynchron

In diesem Abschnitt werden die grundlegenden Änderungen an einem Media Foundation *Transform* (z.B. Encoder, Decoder) diskutiert, um Asynchronität zu ermöglichen. Siehe dazu auch [1].

Das asynchrone Verarbeiten von Daten in einem Video Encoder/Decoder ist wichtig, damit Rechner mit mehreren Prozessoren oder Prozessorkernen optimal unterstützt werden können. Zwar ist die Unterstützung von mehreren Threads auch mit dem bisher verwendeten synchronen Modell denkbar. Durch die Einführung von Asynchronität mittels Events (Nachrichten) kann das Transform aber ideal reagieren: Sobald ein Thread ein Frame fertig verarbeitet hat, kann

[1] http://msdn.microsoft.com/en-us/library/dd317909.aspx. Stand: 14.07.2011

das Transform einen Event schicken, um den nächsten Input anzufordern. In einer synchronen Architektur hat das Transform keine Kontrolle und muss warten, bis der Aufrufer neuen Input sendet bzw. den fertigen Output konsumiert.

Grundsätzlich implementiert ein asynchrones Transform zusätzlich ein Interface (*IMFMediaEventGenerator*). Darüber wird der Aufrufer die Events asynchron aus der Queue des Transforms abrufen.

Sobald das Streaming beginnt (d.h. beim Erhalt der "START_OF_STREAM" Nachricht), kann das Transform die ersten "Need Input"-Events senden. Ab diesem Zeitpunkt korrespondiert jeder Aufruf von *ProcessInput* mit einem solchen Event, stellt also eine Reaktion darauf dar.

Sobald das Transform fertige Daten enthält, kann es "Has Output"-Events senden. Für jeden dieser Events wird *ProcessOutput* aufgerufen.

3.2 Auslagern der Verarbeitung

Um effektiv alle Prozessorkerne auszunutzen müssen wir noch einen Schritt weiter gehen. Die effektive Arbeit muss auf mehrere Threads verschoben werden.

Dazu implementieren wir zuerst das *IMFAsyncCallback* Interface (siehe [2]) auf dem Transform. Dieses schreibt im wesentlichen eine Callback-Methode (*Invoke*) vor. Diese Methode wird später aus dem fremden Thread aufgerufen.

Um dies zu bewerkstelligen verwenden wir sogenannte *Work Queues* aus der Media Foundation Plattform. Eine Applikation kann solche Queues instanzieren und Objekte, welche das IMFAsyncCallback-Interface implementieren, in diese Queues einfügen (*MFPutWorkItem*). Die Work Queue ruft nachher die Callback-Methode Invoke des übergebenen Objekts von ihrem eigenen Thread aus auf. Da jede Work Queue auf ihrem eigenen Thread läuft, wird somit die Callback-Methode *multithreaded* ausgeführt. Abbildung 11 (Quelle: [3]) illustriert diesen Vorgang.

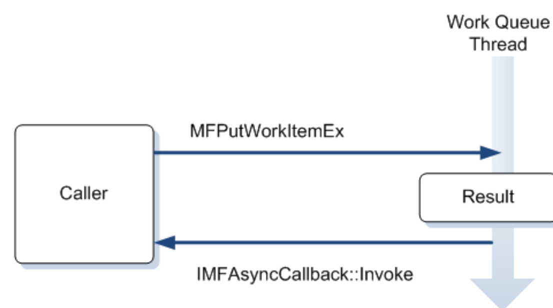

Abbildung 11: Aufruf der Callback-Methode aus der Work Queue.

[2]http://msdn.microsoft.com/en-us/library/ms699856.aspx. Stand: 14.07.2011
[3]http://msdn.microsoft.com/en-us/library/ms703187.aspx. Stand: 14.07.2011

3.3 Multi-Threaded Decoder Performance

Im folgenden wollen wir die effektiv praktisch gemessenen Ergebnisse analysieren. Dazu wurden auf dem Quadcore-Testrechner Messungen mit und ohne aktiviertem *Hyperthreading* (d.h. mit 4 bzw. 8 sichtbaren virtuellen Kernen) durchgeführt.

Abbildung 12 zeigt die gemittelten Messresultate für LPGF. Die Messwerte wurden mit unterschiedlicher Anzahl aktivierter Threads für den Videocodec ermittelt, von 1 bis 8. Die gestrichelte Linie stellt die Abspieldauer des Testvideos dar (9 Sekunden). Eine Decoding-Dauer darüber lässt somit keine Ruckel- oder Verlustfreie Wiedergabe mehr zu – die Ausnutzung der vorhandenen CPU-Kerne bringt den Codec für dieses Video auf dem Testrechner in den benutzbaren Bereich.

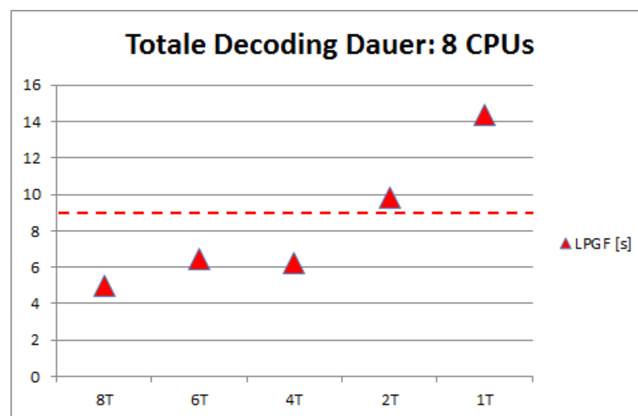

Abbildung 12: LPGF Decoding Dauer des Testvideos "Wildlife".

- Es sollten mindestens so viele Threads gestartet werden sollten, wie auch wirklich physische oder virtuelle CPU-Kerne vorhanden sind. Wenn jedoch zu viele Threads gestartet werden, müssen sie sich die CPU-Kerne teilen, was zu unnötig erhöhtem Verwaltungsaufwand führt – wenn hingegen zu wenige Threads aktiv sind, lassen sich die Anzahl Kerne nicht ausnutzen. Auf dieser Grundlage wird der Codec so programmiert, dass abhängig vom System die Anzahl Threads zur Laufzeit gleich der Anzahl verfügbarer CPU-Kerne gewählt wird.

Im Verlauf des Projekts wurde eine neue Version des PGF-Bildcodecs freigegeben. Dabei wird für bereits die Bildkompression auf mehrere Threads ausgelagert, unter Verwendung von OpenMP. In Abbildung 13 wird die Performance der neuen PGF-Version mit der bisherigen verglichen.

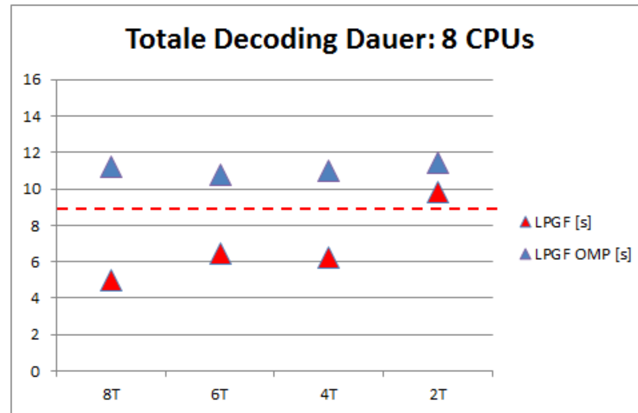

Abbildung 13: Vergleich der Decoding-Dauer mit und ohne OpenMP in PGF.

- Die Version mit aktiviertem OpenMP im Bildcodec (blau) schneidet deutlich schlechter ab als LPGF ohne OpenMP (rot). Dies ist auf die "doppelte Parallelität" zurück zu führen: Weil die Frames auf der Stufe des Videocodecs bereits parallel auf mehreren CPU-Kernen verarbeitet werden, stören die zustäzlichen Threads, welche bei jedem Aufruf des PGF Bilddecoders von OpenMP gestartet werden, mehr als dass sie nützen.

Abbildung 14 zeigt zwei der Resultate des Visual Studio 2010 Profilers. Die grünen Phasen zeigen effektive Verarbeitung/Auslastung an, die roten Abschnitte bedeuten Thread-Synchronisation. Wie erwartet lastet die Variante mit 4 Threads die in dieser Messung verfügbaren 4 CPU Kerne parallel aus, was zu einer niedrigeren Decoding-Dauer des Gesamtvideos führt.

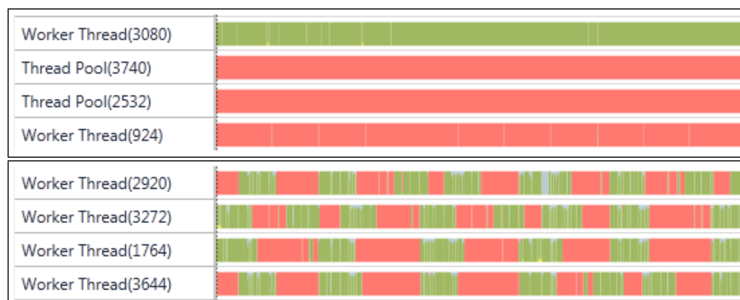

Abbildung 14: Vergleich zwischen 1 Thread (oben) und asynchron mit 4 Threads (unten) (Ausschnitt aus dem Visual Studio Thread-Profiling)

4 Bewegungsschätzung

In diesem Kapitel werden für den Leser die Grundlagen geschaffen, die nachfolgenden Erläuterungen vollständig zu verstehen. Insbesondere werden einige wichtige Begriffe eingeführt. Eine vertiefte theoretische Erarbeitung von Videokompressions-Techniken ist nicht Bestandteil dieser Arbeit. Ich verweise dafür auf Fachliteratur wie [3].

Unter dem Begriff *Bewegungsschätzung* (Englisch *Motion Estimation*) versteht man das Auffinden geeigneter Bewegungsvektoren, d.h. für jeden Bildblock aus Frame A einen möglichst ähnlichen Block in Frame B zu finden. Die örtliche Verschiebung zwischen dem Block aus Frame A und dem aus B nennt man Bewegungsvektor. Das Auffinden dieser Blöcke stellt auch heute noch einen Grossteil des Zeitaufwands für Videokompression dar. Bewegungsvektoren sind dennoch einer der wichtigsten Bestandteile eines modernen Video-Formates. Denn dadurch, dass nur eine Verschiebungsrichtung und ein (möglichst kleiner) Prädiktionsfehler abgespeichert bzw. übermittelt werden muss, wird eine sehr hohe Kompressionsrate möglich. Für weitere Informationen zu den theoretischen Hintergründen der Bewegungsvektoren verweise ich auf die Quellen [5] sowie [3].

In zahlreichen Forschungsprojekte wurden inzwischen ausgeklügelte Verfahren ermittelt, welche zum Ziel haben, rasch einen möglichst guten *Kandidatenblock* zu finden, und so den massiven Rechenaufwand reduzieren zu können. Ein Überblick ist unter anderem in [6] zu finden.

Alle Algorithmen benutzen Heuristiken, um aus einer Menge von Kandidatenblöcken möglichst rasch den besten zu ermitteln. Dabei werden grundsätzliche Charakteristiken von bewegten Bilddaten ausgenutzt, beispielsweise dass die Bewegung von Frame zu Frame meist nur gering ist. Grundsätzlich ist immer zwischen der Geschwindigkeit der Konvergenz und der erreichten Genauigkeit (Minimierung der Differenz) abzuwägen. Trivialerweise ergibt ein sogenannter *Full Search* die besten Ergebnisse mit der schlechtesten Performance: Dabei wird für jeden Bildblock das komplette Referenzbild nach einem möglichst ähnlichen Block durchsucht.

Interessante Verfahren sind unter anderem das PMVFAST (*Predictive Motion Vector Field Adaptive Search Technique*, erläutert in [7]) und das in x.264 verwendete UMHexagonS (unsymmetrical-cross multi-hexagon grid search, [8]). Diese Algorithmen erreichen eine hohe Geschwindigkeit mit wenig Einbussen bei der Qualität der Resultate. Für die Geschwindigkeit sind beispielsweise Abbruchkriterien wichtig, welche sicherstellen, dass der Algorithmus nicht vergebens weiter läuft. Ebenfalls kritisch ist die Wahl der Kandidaten-Vektoren, um die Anzahl Differenzbildungen minimieren zu können, indem rasch zu einem (möglichst globalen) Optimum gewandert wird. Es können auch Erfahrungswerte (vorherige Bewegungsvektoren an der gleichen Position) genutzt werden, um die Kandidatenliste während der Laufzeit dynamisch zu modifizieren und so rascher zu einem Optimum zu gelangen.

Spezifische Algorithmen sollen an dieser Stelle nicht genauer diskutiert werden. Der interessierte Leser sei auf die Fachliteratur über Bewegungsschätzung-Heuristiken [6] und [9], und insbesondere dem PMVFAST [7] oder dem UMHexagonS [8] Verfahren verwiesen.

5 PMF Video Format

In diesem Kapitel wollen wir das vorgestellte Video-Format zuerst theoretisch um fortgeschrittene Funktionalität erweitern, wie sie im Kapitel 4 beschrieben wurden.

5.1 Differenzbild Architektur

Abbildung 15 zeigt schematisch den Aufbau des Encoding-Moduls für PMF auf. Der Prozess t liefert dabei ein vorhergegangenes Frame aus dem Referenz-Stack.

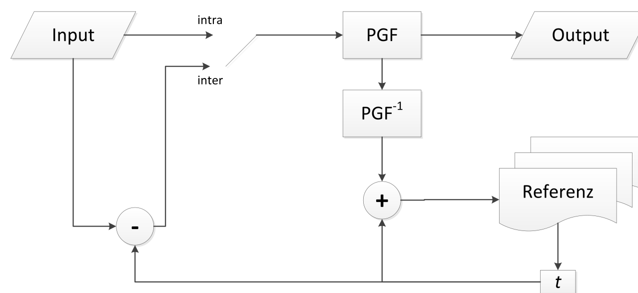

Abbildung 15: Design des PMF Codec mit einfacher Differenzbild-Übertragung

Im folgenden erläutern wir das oben stehende Schema an einer Beispielsequenz:

1. Input: Frame I_1. Beim ersten Frame ist der Modus *intra*, d.h. das Eingabe-Bild wird als Vollbild komprimiert und versandt. Output: $PGF(I_1) = I_1p$. Dieses Frame wird als Referenz gespeichert, nachdem es dekomprimiert wurde: $PGF^{-1}(I_1p) = I_1'$.
2. Zweites Frame: Input I_2. Der Modus wird auf *inter* gesetzt, d.h. es wird nur das Differenz-Bild zum vorherigen Frame übertragen: $PGF(I_1' - I_2) = I_2p$. Abgespeichert als Referenz wird wiederum dessen Rekonstruktion: $PGF^{-1}(I_2p) + I_1' = I_2'$.

Mit diesem Verfahren wird sichergestellt, dass die Qualität im Laufe des Encoding-Vorgangs, sprich mit der Abspielzeit des Videos, konstant gehalten werden kann. Dies soll im folgenden aufgezeigt werden: Durch die Verwendung des verlustbehafteten Modus von PGF wird bei jedem Durchlauf des PGF-Prozesses ein Fehler eingeführt.

- Bei einem *intra*-Frame ist der Fehler gleich dem PGF-Verlust durch Quantisierung.
- Im Falle von *inter*-Frames wird dieser Fehler entsprechend nur bei der Quantisierung der Differenz erzeugt: Da bei jedem neuen Frame die Differenz des Input-Bildes mit dem rekonstruierten (verlustbehafteten) Referenzbild erzeugt wird, kann sich der Fehler nicht von Frame zu Frame weiter vererben.

Bei falschem Design des Codec kann ansonsten der Fall auftreten, dass sich der entstandene Fehler fortpflanzt und somit eine stets abnehmende Video-Qualität verursacht.

5.2 PMF Architektur Vorschlag

In Abbildung 16 wird das vorherige Differenzbild-Schema erweitert um Bewegungsvektoren (siehe Kapitel 4). Zusätzlich wird die bisher betrachtete "Blackbox" des PGF-Prozesses aufgeteilt in die einzelnen Schritte der Bildkompression.

Beachten Sie auch die beiden unterschiedlichen Definitionen von Blöcken in diesem Schema:

- **PGF-Tile**: Ein relativ grosser "Block" (in PGF als *Region Of Interest* bezeichnet), siehe auch Kapitel 2.2. PGF ist auf Blockgrössen/Tiles von mindestens 90x90 Pixel ausgelegt, deshalb verwendet dieses Schema zur Kompression eine solche Blockgrösse.
- **Block**: Ein deutlich kleinerer Bildblock, welcher durch Matching für die Bewegungskompensation genutzt werden kann (siehe unten). Pro "Motion"-Block wird ein Bewegungsvektor (*Bewegungsschätzung*) und eine Differenz (*Bewegungskompensation*) erzeugt. Das Zusammensetzen der Differenzen ergibt das *Prädiktionsbild*.

Der Ablauf ist im Vergleich zur vorherigen Skizze etwas umfangreicher, im Grundsatz bleibt jedoch die gleiche Idee bestehen. Neu wird einzig nicht bloss eine Differenz gebildet, sondern **blockweise** die Differenz zwischen allen aktuellen Bildblöcken (aus Frame Nr. N) und dem jeweils besten Referenzblock (aus Frame Nr. N-1) errechnet und kodiert (es entsteht das *Prädiktionsbild*). Daraus entsteht für jeden Bildblock der Verschiebungsvektor (die Bewegung) gegenüber dem für die Differenz verwendeten Referenzblock. Diese beiden Werte werden *multiplexed* und an den Empfänger übermittelt. Für die lokale Referenz-Datenbank wird das Fehlerbild wiederhergestellt (dekodiert) und mit dem aus der *Bewegungskompensation* erhaltenen Prädiktionsbild rekonstruiert und abgespeichert:

1. **Segmentierung**: Das Input-Bild wird aufgeteilt in einzelne kleine Pixel-Blöcke (z.B. 16x16 oder 8x8 Pixel).
2. **Bewegungsschätzung**: Hier wird versucht, für die Bildblöcke möglichst gute Referenzblöcke zu finden (kleine Differenz). Siehe auch Kapitel 4. Output sind Bewegungsvektoren.
3. **Bewegungskompensation**: Aus den Bewegungsvektoren werden die Prädiktionsblöcke erzeugt. Diese enthalten die Bildinformation des Ziels der Vektoren.
4. **Differenzbildung**: Die Prädiktionsblöcke werden mit den Original-Blöcken verglichen, die Differenz daraus (das Fehlerbild) wird als 1. Komponente *Tile*-weise quantisiert, kodiert und multiplexed.

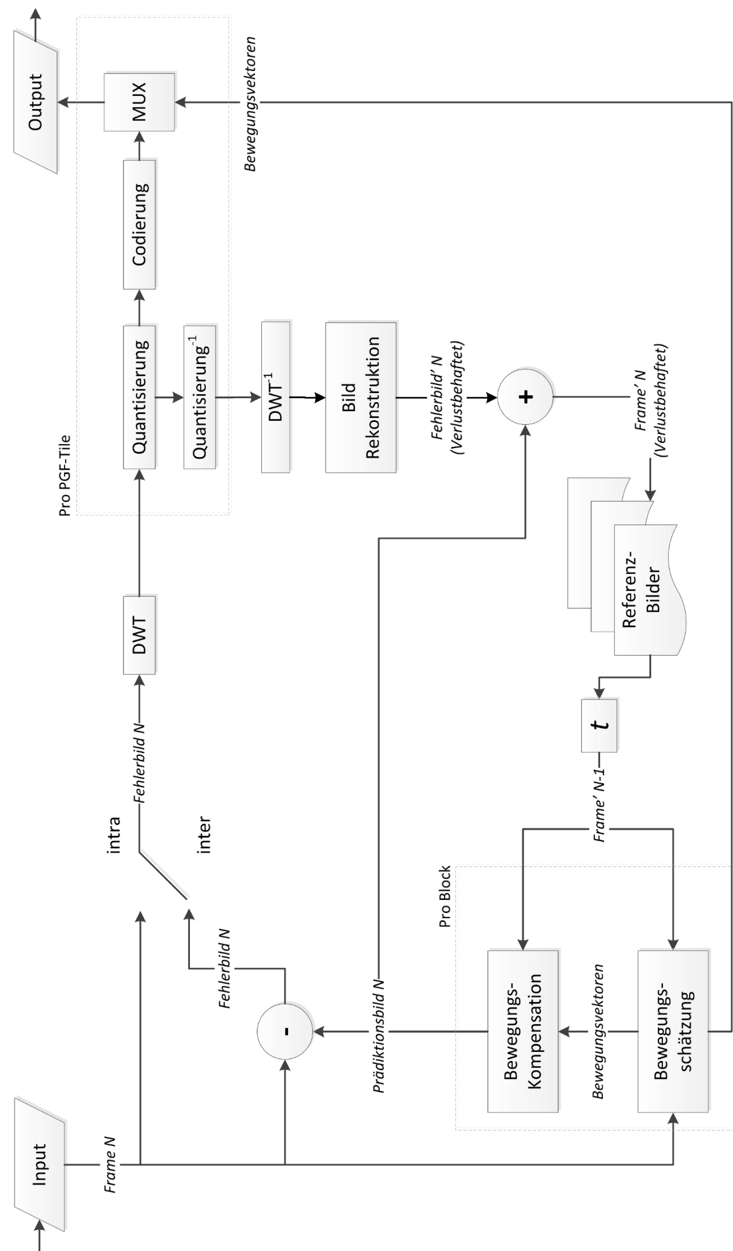

Abbildung 16: Architektur mit Bewegungsvektoren, Tiles- und Blockverarbeitung

5. **Muxing**: Als 2. Komponente werden die Bewegungsvektoren pro Block verwendet. Der Vorgang des Muxings kodiert die beiden Komponenten in einen Bitstream. Der Empfänger rekonstruiert dann das fertige Frame mit Hilfe der Bewegungsvektoren, seiner gespeicherten Referenzframes und dem empfangenen Prädiktionsfehler.

6. **Referenz**: Als Referenzframe auf der Absender-Seite wird die Rekonstruktion abgespeichert: Dazu werden die versandten Prädiktionsfehler mit den korrespondierenden Prädiktionsblöcken addiert.

5.3 Live-Streaming

Bei Live-Streaming wird als zusätzliche Anforderung die mögliche Dauer der Verarbeitung eingeschränkt. So muss einerseits die Encoding-Komponente in der Lage sein, die ausgehandelte Anzahl Frames pro Sekunde liefern zu können. Auf der Empfänger-Seite muss das Gegenstück, der Decoder, andererseits eine ebenso schnelle Verarbeitung sicher stellen.

Diese Situation kann jedoch nicht vorausgesetzt werden, insbesondere nicht bei grossen Bildern oder hohen Frameraten – kurz: Bei mangelnder Leistung der entsprechenden Rechner oder Übertragungsmedien.

Um im Szenario des Live-Streamings trotz dieser Faktoren ein möglichst gutes Nutzerlebnis zu erreichen, wird im folgenden eine erste einfache Erweiterung, wie in Abbildung 17 illustriert, eingeführt.

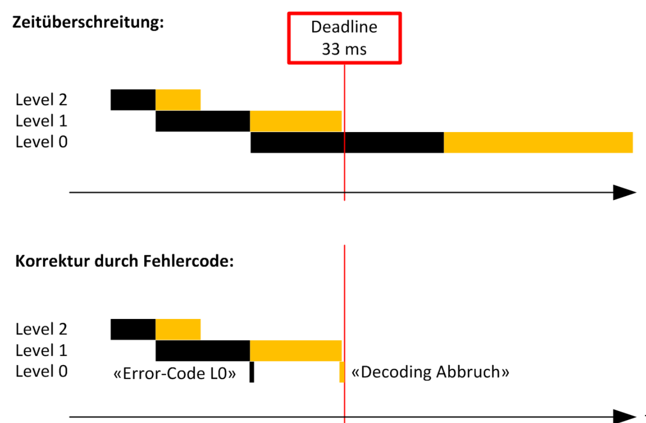

Abbildung 17: Sicherstellen der Framerate durch Einhalten der zur Verfügung stehenden Zeit.

Pro Frame steht eine gegebene Zeit zur Verfügung, welche nicht überschritten werden darf, um eine unterbruchsfreie Wiedergabe garantieren zu können. Diese Zeit gilt bei Live-Streaming in der Encoder- sowie in der Decoder-Komponente. Aus dieser Grund überwachen beide Komponenten die Zeit, welche sie für die Verarbeitung des aktuellen Frames bereits aufgewendet haben. Sobald die Zeit den gegebenen Rahmen sprengt, wird die Kompression bzw. Dekompression für das aktuelle Frame abgebrochen.

Encoder: Hier hat das Abbrechen zur Folge, dass keine weiteren Levels des aktuellen Frames codiert werden können. Im aktuellen Level wird ein Abbruch-Code in den Header geschrieben.

Decoder: Die Decoder-Komponente muss auf zwei unterschiedliche Ereignisse reagieren können:

1. Das Encoding-Modul konnte nicht alle Levels codieren. Ein Fehlercode wird empfangen.
2. Die verfügbare Zeit für das Decoding des aktuellen Frames ist abgelaufen.

Beide Fälle führen dazu, dass keine weiteren Levels mehr zur Verfügung stehen. Das aktuelle Frame muss mit den bis jetzt verfügbaren decodierten Daten dargestellt werden. Dazu muss das empfangene Bild auf die originale Bildgrösse hoch skaliert werden – der Nutzer wird dies als Qualitätsreduktion wahrnehmen.

Aus einem jeweiligen Event kann gelernt werden: So sollte die Encoding-Komponente auf die fehlende Zeit reagieren, indem die Qualität der produzierten Daten für die folgenden Frames reduziert wird und so die Verarbeitung rascher erfolgen kann.

Implementation: Die Grundlage für die Implementation wird während dieses Semesters bereits geschaffen. In PMF1 wird ein 4-Byte Header eingeführt, welcher vor jedem Sample abgespeichert wird. Das erste Byte dieses Headers gibt an, ob dieses Sample das letzte des aktuellen Frames ist oder nicht (0 = ”PMF_FRAME_CONTINUE”, 1 = ”PMF_FRAME_STOP”). Die PMF1 Decoding-Komponente verwendet dieses Byte bereits, um festzustellen, wann alle Samples für das aktuelle Frame erhalten wurden.

Die übrigen Bytes des Headers sind in der aktuellen Version nicht verwendet und leer zu lassen.

6 Reflexion

Ich konnte in diesem Semester einige unterschiedliche Aufgaben in Angriff nehmen, welche in dieser Dokumentation beschrieben sind. Insbesondere durfte ich meine Kenntnisse der Videokompression vertiefen, um die theoretischen Grundlagen für das neue PMF-Format zu schaffen.

Der grössere praktische Teil bestand jedoch darin, die Version PMF1 zu entwickeln, wobei das Resultat davon dann nicht unseren ersten Erwartungen entsprach: Die Version performt schlechter als LPGF. Die erhaltenen Resultate konnte ich mit dem Simulationsprogramm erklären – allerdings ist noch immer unklar, wie viele Samples pro Frame nun der Idealwert darstellt. Grund dafür ist, dass die Simulation nicht alle Faktoren mit einbeziehen kann, insbesondere die Daten-Prozessierung durch Media Foundation.

Eine Möglichkeit, die optimale Anzahl Samples pro Frame zu ermitteln, ist die Grösse der verwendeten PGF-Tiles zu verändern und die Performance empirisch zu messen. Dies durchzuführen fand leider in diesem Semester kein Platz mehr.

Im weiteren Verlauf dieses Master-Projekts wird die Einzelbild-Kompression durch Bewegungsschätzung ersetzt werden. Dies beinhaltet zahlreiche neue Herausforderungen, nebst der Implementation der Bewegungsschätzung wird auch die Zusammenfassung der Blöcke in optimal dimensionierte Tiles (als Einheiten für die PGF-Kompression) eine anspruchsvolle Aufgabe.

7 Ehrlichkeitserklärung

Der Autor des vorliegenden Dokuments, Stefan Weber, bestätigt hiermit, das Informatik-Projekt 8 am Institut für Mobile und Verteilte Systeme gewissentlich unter Einhaltung aller gebotenen Regeln durchgeführt zu haben.

Der Autor bestätigt weiter, dass keine Teile der Dokumentation oder des geschriebenen Quellcodes unrechtmässig kopiert wurden oder gegen schweizerisches Recht verstossen.

2. September 2011,

Stefan Weber

Literatur

[1] Christoph Stamm. *PGF: Progressive Graphics File.* University of Applied Sciences, Northwestern Switzerland.

[2] Christoph Stamm. *PGF: A new progressive file format for lossy and lossless image compression.* ETH Zurich, Institute of Theoretical Computer Science.

[3] Tilo Strutz. *Bilddatenkompression: Grundlagen, Codierung, Wavelets, JPEG, MPEG, H.264, Band 3.* Vieweg Praxiswissen, 2005.

[4] Werner Bäni. *Wavelets: Eine Einführung für Ingenieure, Band 2.* Oldenbourg Wissenschaftsverlag, 2005.

[5] Daniel Imhof und Renato Comelli. *Bewegungskompensation in Videodatenkompression: Informatik Bachelor Thesis*, 2008.

[6] Li Zi yin und Zou Xi-yong. Fast block-based motion estimation techniques in video compression. In *Congress on Image and Signal Processing*, 2008.

[7] Alexis M. Tourapis und Oscar C. Au und Ming L. Liou. *Predictive Motion Vector Field Adaptive Search Technique (PMVFAST) - Enhancing Block Based Motion Estimation.* Department of Electrical and Electronic Engineering, The Hong Kong University of Science and Technology, 2002.

[8] Xie Lifen und Huang Chunqing und Chen Bihui. Umhexagons search algorithm for fast motion estimation. In *Computer Research and Development (ICCRD), 2011 3rd International Conference, Band 1, Seiten 483-487*, März 2011.

[9] Hui Gu und Yan-Shan Li. *The Research Of Block Motion Estimation Algorithm In Video Compression.* College of Information Engineering, Zhejiang University Of Technology, 2004.