Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Fabian Jäger

Eine Videokonferenz-Software für das iPhone

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Fabian Jäger

Eine Videokonferenz-Software für das iPhone

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Schmidt
Zweitgutachter : Prof. Dr. Meisel

Abgegeben am April 29, 2010

Fabian Jäger

Thema der Bachelorarbeit

Eine Videokonferenz-Software für das iPhone

Stichworte

iPhone, Videokonferenz, Moviecast, Daviko, Codec, Objective-C, XCode, iPhone OS, iPhone SDK, OpenGL ES, AudioToolbox, Videokodierung, Portierung

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Realisierung einer Videokonferenz-Software auf dem iPhone. Die Software basiert auf der Daviko-Software und soll in das Moviecast-Projekt der HAW Hamburg integriert werden. Es werden die Eigenschaften des iPhones und dessen Betriebssystem untersucht und bei der Entwicklung berücksichtigt. Abschließend wird das Zeitverhalten gemessen, analysiert und mit Referenzgeräten verglichen.

Fabian Jäger

Title of the paper

A video conference software for the iPhone

Keywords

iPhone, video conference, Moviecast, Daviko, codec, Objective-C, XCode, iPhone OS, iPhone SDK, OpenGL ES, AudioToolbox, video coding, porting

Abstract

This bachelor-thesis focuses on the realization of a video conference software for the iPhone. The software is based on a software named Daviko and intends to integrate into the Moviecast project developed by the HAW Hamburg. The characteristics of the iPhone and its operationsystem are analized and considered in the developement process. Finally the performance is measured, discussed and compared to other devices.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Codeverzeichnis

1 Einleitung

Die rasante Weiterentwicklung der Mobilfunkgeräte zu vollständigen mobilen Computern macht auch neue Formen der Kommunikation möglich. Handys sind schon lange mehr als nur Telefone. Sie werden als Terminplaner, als Abspielgerät für Multimedia oder als Internetzugang genutzt. Diese Vielseitigkeit ist erst durch die ständige Weiterentwicklung der Prozessoren, Netzwerkschnittstellen und Grafikprozessoren möglich geworden, die für immer mehr Rechenleistung auf den Handys sorgen. Diese ist inzwischen groß genug, um auch über komplexe Programme, wie z.B. eine Videokonferenz, nachdenken zu können. Da hierzu Video- und Audiosignale in Echtzeit verarbeitet und transportiert werden müssen, sind die Anforderungen an die Hardware sehr hoch. Zusätzlich hat die Entwicklung der Betriebssysteme einen großen Einfluss, da diese einen einheitlichen und effizienten Zugriff auf die Hardware ermöglichen. Sie stellen grundlegende Bibliotheken zur Verfügung, mit denen der Implementierungsaufwand gering gehalten werden kann. Eine Videokonferenzsoftware für Handys ist deshalb von großem Interesse, weil diese im Gegensatz zu Computern als Kommunikationsmedium entwickelt wurden und dafür auch hauptsächlich genutzt werden.

1.1 Problemstellung

Es soll eine bereits existierende Videokonferenzsoftware auf das iPhone portiert werden. Die Software, die als Grundlage dient, wurde basierend auf der PC-Version für Windows Mobile Geräte entwickelt. Bedingt durch die Einschränkungen der mobilen Geräte hat sie im Vergleich zur PC Version einen geringeren Funktionsumfang. Die Version, die zur Portierung genutzt werden soll, ist ein stabiler Zwischenstand, bei dem noch nicht alle geplanten Funktionen implementiert sind.

Die Software muss an die besonderen Eigenschaften des iPhones angepasst werden. Das Betriebssystem und die verfügbaren Bibliotheken sind dabei die ausschlaggebenden Komponenten. Die Software soll sich in das bestehende Netzwerk integrieren und vollständig kompatibel zu allen Teilnehmern sein.

Anschließend sollen die Ergebnisse gemessen und mit bestehenden Systemen verglichen werden.

1.2 Allgemeine Grundlagen

Um ein Verständnis für die Funktionsweise der Software zu entwickeln, sind einige Grundlagen notwendig. In diesem Kapitel werden daher Begriffe und Protokolle erläutert, die häufig im Zusammenhang mit Videokonferenzen verwendet werden.

1.2.1 Portierung

“Portierung“ ist kein klar definierter Begriff und bezeichnet in der Praxis häufig eine Auswechslung von Komponenten innerhalb bestehender Systeme.

Dazu zwei Beispiele:

Bei eingebetteten Systemen besteht häufig das Problem, dass die Software speziell an die verwendete Hardware angepasst ist. Möchte man auf eine andere Hardware umsteigen, muss die Software angepasst und somit portiert werden.

Bei einer Software, die auf einem Betriebssystem abläuft, ist die Hardwareabhängigkeit meist nicht entscheidend, da diese durch das Betriebssystem gekapselt wird. Jedoch besteht eine Abhängigkeit zum Betriebssystem, die bei dessen Austausch eine Portierung der Software notwendig macht.

Dieses sind nur zwei Beispiele dafür, was mit “Portierung“ gemeint sein kann. Um den Begriff klarer zu definieren, verwende ich ihn in dieser Arbeit wie folgt:

> Unter einer Portierung versteht man bei der Softwareentwicklung den Vorgang, ein Computerprogramm, das unter einem bestimmten Betriebssystem, einer Betriebssystemversion oder Hardware abläuft, auch auf anderen Betriebssystemen, anderen Versionen oder anderer Hardware lauffähig zu machen (babylon, 2010).

1.2.2 Videokonferenz

Bei einer Videokonferenz handelt es sich um eine Technik, die Konferenzen über ein Netzwerk mit Ton- und Videodaten in Echtzeit ermöglicht.

Die Idee einer Videokonferenz ist nicht neu, konnte sich aber lange Zeit nicht durchsetzen. Ihre Entwicklung scheiterte oft an der Hardware, die entweder zu langsam oder zu teuer war und dem Fehlen einer Infrastruktur, über die die Daten versendet werden konnten. Erst die Flexibilität des Internets sowie leistungsfähigere Prozessoren und Netzwerke für mobile

Geräte, wie z.B. UMTS, machen eine günstige und einfache Videokonferenz möglich (vgl. Cycon u. a., 2008, S. 1).

1.2.3 Streaming und Datenübertragung

Beim Streaming handelt es sich um eine Datenübertragung, bei der die Datengröße zum Beginn der Übertragung nicht bekannt ist. Die Daten werden in einem ständigen Strom übertragen, ohne dass für den Empfänger das Ende abzusehen ist. Bei einer Videokonferenz werden die Ton- und Videodaten als Strom verteilt, da die Daten kontinuierlich entstehen und übertragen werden müssen. Für die Übertragung der Daten können, je nach den Anforderungen, verschiedene Protokolle benutzt werden. Für Multimediadaten wird häufig das Real-Time Protocol (vgl. Schulzrinne u. a., 2003) benutzt, das hauptsächlich mit UDP verwendet wird.

Bei einer audiovisuellen Echtzeitübertragung ist die schnelle Übertragung wesentlich wichtiger als die Vollständigkeit der Daten. Das liegt an der Wahrnehmung des Menschen. Ein verlorenes Datenpaket fällt weniger auf als ein vollständiger Datenstrom, der aber stockend ankommt.

Üblicherweise wird UDP als Transportprotokoll verwendet, da TCP für diese Aufgabe ungeeignet ist. Bei TCP kann es zu Unterbrechungen des Stroms kommen, z.B. durch Head-of-line-blocking, bei dem nach einem Paketverlust der gesamte Strom so lange angehalten wird, bis das Paket erneut gesendet und empfangen wurde.

Da UDP keine Datenflusskontrolle besitzt, ist es möglich, dass ein schneller Sender einen langsamen Empfänger überlastet. Dieses Verhalten kann durch das Real-Time Control Protocol (RTCP), welches zu RTP gehört, verhindert werden. RTCP kann an die Anforderungen angepasst werden und so z.B. die Einhaltung einer angemessenen Dienstgüte (QoS[1]) regeln, indem es die Übertragungsrate der Teilnehmer anpasst.

1.2.4 Session Initiation Protocol

Um eine Videokonferenz aufzubauen genügt es nicht, nur Daten verschicken und empfangen zu können, sondern es müssen auch die Teilnehmer eindeutig identifizierbar sein. Außerdem kann es sein, dass einzelne Teilnehmer hinzukommen oder auflegen. Für diese Sitzungsverwaltung wird das zustandsbasierte Session Initiation Protocol (vgl. Rosenberg u. a., 2002) verwendet.

[1] Quality of Service

SIP ist ausschließlich für die Verwaltung einer Sitzung zuständig. Im Gegensatz zu anderen Protokollen, wie z.B. H.323, ist dabei zunächst nicht festgelegt, welche Art von Daten ausgetauscht werden sollen. Ein häufiges Anwendungsgebiet ist die Verwaltung von Audio- und Videoübertragungen.

Bevor eine Multimediaübertragung gestartet werden kann, müssen die Teilnehmer festlegen, welche Codecs sie beherrschen, mit welchem Protokoll die Übertragung stattfinden soll und welche Netzwerkadressen zum Senden und Empfangen benutzt werden sollen. Für diese Aufgaben ist das Session Description Protocol (vgl. Handley u. a., 2006) zuständig. Jeder Teilnehmer kann ein Angebot im Netzwerk verteilen, das eine Liste von Codecs enthält, die er verarbeiten kann. Es wird dabei zwischen Codecs unterschieden, die der Teilnehmer senden kann und denen, die er empfangen kann. Bei einer Videokonferenz kann ein Teilnehmer üblicherweise dieselben Codecs senden und empfangen. Auf diese Angebote können alle weiteren Teilnehmer reagieren, indem sie eine Antwort zurücksenden, die auch eine Liste von möglichen Codecs enthält. Mit diesem Offer/Answer Prinzip können die verwendeten Codecs für eine Verbindung ausgehandelt werden.

Diese Trennung von Sitzungsverwaltung und Medienaushandlung macht SIP sehr flexibel, da auch für unbekannte Protokolle mit geringem Aufwand eine Medienaushandlung entworfen werden kann.

SIP ist unabhängig vom Transportprotokoll, weshalb es möglich ist, SIP mit UDP, TCP oder SCTP zu verwenden. Da SIP selbst schon ein Handshake-, Wiederholungs- und Timeout-Verfahren besitzt, wird TCP oder SCTP eher selten eingesetzt. Es bietet sich daher an, UDP zu verwenden, um einen zusätzlichen Verbindungsaufbau und unnötigen Overhead zu vermeiden (vgl. Trick, 2007, S. 124).

2 Grundlagen der Videokodierung

Da Videodaten sehr groß sind, ist es häufig notwendig, diese zu komprimieren. Kompressionsverfahren mit den dazugehörigen Dekompressionsverfahren nennt man Codec[1]. Diese Verfahren können entweder als Hardware, als Software oder gemischt realisiert werden, wobei die reine Hardwarelösung in der Regel am schnellsten ist.

Viele gängige Codecs sind asymetrisch in ihrer Ausführgeschwindigkeit. Das bedeutet, dass das Komprimieren deutlich rechenintensiver ist als das Dekomprimieren, was sich auch auf die Geschwindigkeit auswirkt. Das liegt an der verwendeten Technik der Codecs.

Predictive Interframe Codecs, wie z.B. H.261, H.263, H.264, MPEG-1 oder MPEG-2 greifen auf mehrere Verfahren zurück, um eine möglichst hohe Kompression zu erreichen. Einige Verfahren wie z.B. die statistischen Kompressionsverfahren werden auch in der Bildverarbeitung eingesetzt um einzelne Bilder zu komprimieren, während andere versuchen, die räumlichen und zeitlichen Redundanzen zu erkennen und diese für die Komprimierung zu nutzen.

Zunächst ist es aber wichtig, sich mit der Wahrnehmung des Menschen auseinanderzusetzen, um die Kompressionsverfahren zu verstehen. In natürlichen Bildern sind die meisten Informationen in der Helligkeit gespeichert und nicht in der Farbe. Der Mensch hat sich im Laufe der Evolution angepasst und sieht kleine Helligkeitsunterschiede besser als kleine Farbunterschiede. In der Abbildung 2.1 lässt sich dieses gut beobachten. Ganz oben befindet sich das Originalbild gefolgt von der Helligkeit, dem Farbton und der Sättigung. Man kann sehen, dass die Helligkeit für den Betrachter die meisten Informationen enthält. Daher sollte darauf geachtet werden, dass die Kompression bei diesen Informationen möglichst verlustarm vorgenommen wird.

2.1 Farbmodell und Farbunterabtastung

Ein übliches Format für Bilder ist das RGB Format, das jedoch eher ungeeignet zur Bildkompression ist. Weitaus besser eignet sich das YCbCr, bei dem ein Bild in Helligkeit, Farbton und Sättigung dargestellt wird.

[1] **Co**mpression/**Dec**ompression

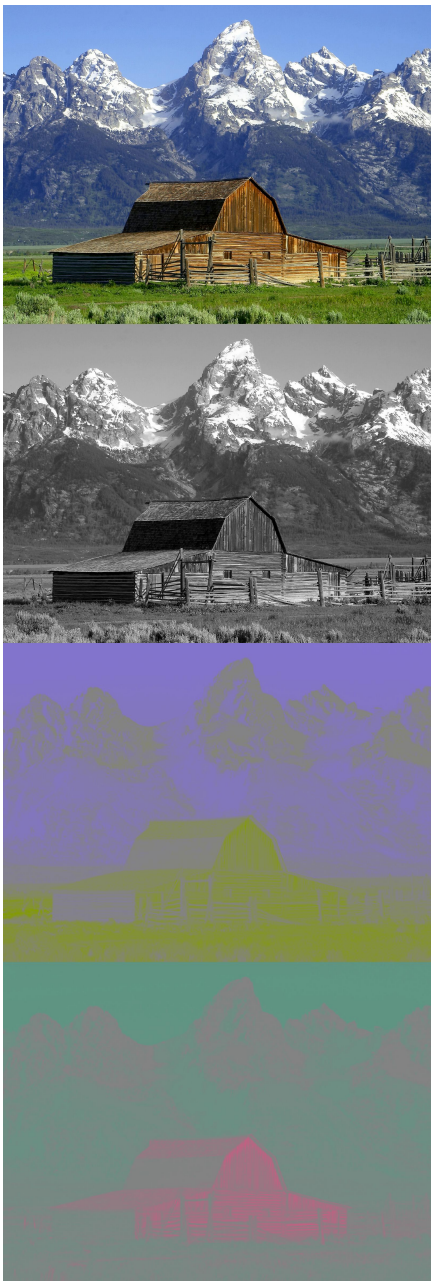

Abbildung 2.1: *Beispiel YCbCr (Quelle: de.wikipedia.org)*

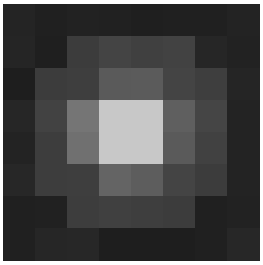

Abbildung 2.2: *8x8 Grauwertblock vor der Kompression*

Die für den Menschen wichtigen Informationen eines Bildes befinden sich in der Helligkeit, während kleine Änderungen im Farbton oder in der Sättigung vom Menschen nicht wahrgenommen werden. Diese Eigenschaft kann zur Kompression genutzt werden, indem hohe Frequenzen aus dem Farbton und der Sättigung herausgefiltert werden. In der Praxis wird dies mit einer 2x2 Matrix berechnet, die für jeweils vier Pixel einen Mittelwert berechnet. Auf diese Weise wird nur noch ein Viertel der Pixel für Helligkeit und Sättigung benötigt. Dieses Format wird YCbCr 4:2:0[2] genannt und wird häufig in der Nachrichtentechnik verwendet.

2.2 Diskrete Cosinustransformation

Bei der diskreten Cosinustransformation (DCT) werden die Helligkeitssignale in einen Frequenzbereich transformiert, um eine räumliche Dekorrelation zu erreichen. Dazu wird das Bild in gleich große Blöcke eingeteilt, von denen jeder Block dann der diskreten Cosinustransformation unterzogen wird.

Ein Block kann z.B. eine Größe von 8x8 Pixel besitzen und zur Vereinfachung pro Pixel einen 8 Bit Grauwert enthalten. Die Matrix zu einem Block, der in Abbildung 2.2 dargestellt ist, sieht wie folgt aus:

$$\begin{pmatrix} 36 & 32 & 34 & 33 & 31 & 32 & 32 & 35 \\ 37 & 31 & 60 & 68 & 64 & 66 & 38 & 34 \\ 30 & 60 & 62 & 90 & 91 & 68 & 62 & 37 \\ 38 & 66 & 117 & 200 & 200 & 95 & 69 & 35 \\ 35 & 61 & 112 & 200 & 200 & 90 & 65 & 35 \\ 39 & 61 & 62 & 100 & 93 & 68 & 60 & 35 \\ 32 & 33 & 61 & 64 & 62 & 60 & 32 & 34 \\ 32 & 38 & 39 & 30 & 30 & 30 & 32 & 39 \end{pmatrix}$$

[2] Die Bezeichnung kommt noch aus der analogen Nachrichtentechnik und hat daher keine Relevanz für die Berechnung

Die DCT kann mit folgender Formel berechnet werden:

$$F(u,v) = \frac{Cu}{2}\frac{Cv}{2}\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos\left(\frac{(2x+1)*u\pi}{16}\right)\cos\left(\frac{(2y+1)*v\pi}{16}\right)$$

$$mit$$

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

Nach der DCT erhält man als Ergebnis eine Matrix, die aber nun die Koeffizienten für die einzelnen Frequenzanteile enthält und wie folgt aussieht:

$$\begin{pmatrix} 493 & 2 & -181 & -8 & 36 & 4 & -21 & 4 \\ 1 & -3 & -1 & 5 & 0 & 4 & 1 & 0 \\ -193 & -3 & 164 & 5 & -44 & -4 & 40 & -8 \\ 0 & 3 & -3 & 3 & 7 & 4 & 0 & -2 \\ 45 & 7 & -48 & -12 & 30 & 0 & -15 & 12 \\ -1 & 1 & 0 & -1 & -2 & -2 & 1 & 1 \\ -34 & -2 & 50 & 3 & -19 & -4 & -19 & -6 \\ -9 & -3 & 0 & -2 & 0 & -3 & 0 & 2 \end{pmatrix}$$

An der Position 0,0 befindet sich der Koeffizient, der den Grundfarbton bestimmt. Dieser wird auch als DC[3]-Koeffizient bezeichnet. Alle anderen Werte werden als AC[4]-Koeffizienten bezeichnet und werden zu dem Grundton addiert.

Große gleichmäßige Flächen schlagen sich in niedrigeren Frequenzen nieder, während kleine Details und kontrastreiche Kanten sich in hohen Frequenzen niederschlagen.

Anhand des Index eines Koeffizienten lässt sich erkennen, für welche Frequenz er verantwortlich ist. Je höher der Index, desto höher die Frequenz. Durch Umwandlung der Helligkeit in ihre Frequenzanteile erhält man noch keine Kompression, sondern nur eine Zerlegung, die unabhängig von der Raumauflösung ist. Die diskrete Cosinustransformation ist nicht verlustbehaftet und lässt sich durch die inverse DCT wieder umkehren.

[3] Direct Current
[4] Alternating Current

2.3 Quantisierung

Bei der Quantisierung wird die Matrix, die von der DCT erstellt wurde, weiter bearbeitet. Die analogen Frequenzen werden auf einen Wertebereich abgebildet, was als Diskretisierung bezeichnet wird. Ziel ist es, die hohen Frequenzen herauszufiltern, da diese vom Auge kaum war genommen werden. Dieses wird erreicht, indem man jeden einzelnen Koeffizienten durch einen Quantisierungswert teilt. Dieser wird in der Regel für jeden einzelnen Index aus einer festgelegten Tabelle herausgelesen. Diese Tabelle enthält empirisch ermittelte Divisoren, die auf die Wahrnehmung des Menschen abgestimmt sind, um den wahrgenommenen Qualitätsverlust möglichst gering zu halten.

Der Grundton wird in der Regel nicht verändert, während der Divisor umso größer wird, je höher die Frequenz ist. Als Ergebnis erhält man eine Matrix, in der die Koeffizienten für die hohen Frequenzen 0 sind. Durch die Rundung auf ganzzahlige Werte, die für die in Abschnitt 2.4 beschriebene Lauflängenkodierung notwendig sind, verliert man Informationen, wodurch ein Qualitätsverlust stattfindet.

Wird der Quantisierungsfaktor zu hoch gewählt, werden alle AC-Koeffizienten auf 0 gerundet und es bleibt nur der DC-Koeffizient übrig. Der komplette Block nimmt dann den Grundton an, wodurch das Bild pixelig wirkt.

Zur Vereinfachung werden in diesem Beispiel alle Frequenzanteile auf 0 gesetzt, bei denen einer der beiden Indizes größer als vier ist. Folgende Matrix ergibt sich:

$$\begin{pmatrix} 493 & 2 & -181 & -8 & 0 & 0 & 0 & 0 \\ 1 & -3 & -1 & 5 & 0 & 0 & 0 & 0 \\ -193 & -3 & 164 & 5 & 0 & 0 & 0 & 0 \\ 0 & 3 & -3 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Wandelt man diese Matrix mit der inversen DCT um, erhält man folgende Matrix als Ergebnis, die in Abbildung 2.3 als Bild dargestellt ist.

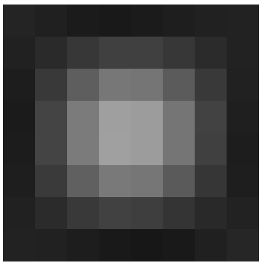

Abbildung 2.3: *8x8 Grauwertblock nach der Kompression*

$$\begin{pmatrix} 36 & 32 & 27 & 25 & 27 & 30 & 33 & 34 \\ 33 & 42 & 55 & 64 & 64 & 55 & 43 & 34 \\ 29 & 57 & 94 & 119 & 117 & 91 & 57 & 33 \\ 27 & 68 & 123 & 159 & 156 & 117 & 66 & 31 \\ 28 & 68 & 123 & 160 & 156 & 117 & 64 & 29 \\ 30 & 58 & 96 & 121 & 118 & 90 & 54 & 30 \\ 33 & 43 & 57 & 65 & 62 & 52 & 41 & 34 \\ 34 & 33 & 29 & 25 & 23 & 25 & 32 & 38 \end{pmatrix}$$

Trotz der Reduzierung der Daten um drei Viertel kann das Bild rekonstruiert werden, ohne dass es für den Menschen unkenntlich wird.

2.4 Lauflängen- und Entropiekodierung

Eine weitere Kompression kann durch statistische Reduktion der Quantisierung erreicht werden. Zunächst wird die Lauflängenkodierung angewendet. Dazu werden die einzelnen Pixel der 8x8 Blöcke hintereinander aufgereiht, so dass die tiefen Frequenzen vorn und die hohen Frequenzen hinten stehen.

```
493 2 1 -193 -3 -181 -8 -1 -3 0 0 3 164 5 0 0 0 5 -3 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Die langen Reihen von Nullen können durch die Lauflängenkodierung komprimiert werden, indem nur die Anzahl der Wiederholungen gespeichert werden.

Bei der Entropiekodierung findet eine weitere Kompression der einzelnen Bilder statt. Ein häufig verwendeter Algorithmus ist die Huffman-Kodierung, bei der versucht wird, für jeden einzelnen Wert eine Bitkodierung zu finden, die einmalig ist und sich auch nicht durch

Tabelle 2.1: *Schlechte Entropiekodierung*

Wert	Bitkombination
A	10
B	01
C	0

Tabelle 2.2: *Gute Entropiekodierung*

Wert	Bitkombination
A	10
B	01
C	11

Kombination von Bitkodierung anderer Werte erzeugen lässt. Ein Beispiel für eine schlecht gewählte Kombination ist in Tabelle 2.1 zu sehen.

Bei der Kodierung der Zeichenfolge ABC entsteht die Bitfolge “10010“, die nicht eindeutig zurückübersetzt werden könnte, da sie entweder ABC oder ACA bedeuten kann. Besser wäre eine Huffman-Kodierung wie in Tabelle 2.2, die mit Hilfe eines Spannbaumes gefunden wurde.

2.5 Differenzkodierung

Die vorgenannten Verfahren beschreiben nur das Komprimieren einzelner Bilder, berücksichtigen allerdings noch nicht die Eigenschaften eines Videos.

Bei der Differenzkodierung wird nach Differenzen zum vorherigen Bild gesucht und die Abweichungen werden gespeichert. Diese Frames werden P-Frames genannt und enthalten nur die Differenzinformationen zu ihrem direkten Vorgänger, der auch ein P-Frame sein kann. Da aber am Anfang immer ein komplettes Bild übertragen werden muss, damit der Dekoder seine Vorhersage machen kann, gibt es I-Frames. Im Verlauf eines Videos werden in regelmäßigen Abständen I-Frames gesendet, um evtl. kleinere Abweichungen auszugleichen, die z.B. durch Rundungsfehler oder durch Paketverlust einer Videokonferenz auftreten können.

Es gibt zusätzlich noch B-Frames, die sich nicht nur auf ihren Vorgänger, sondern auch auf ihren Nachfolger beziehen. Diese sind bei einer Videokonferenz aber ungeeignet, da der

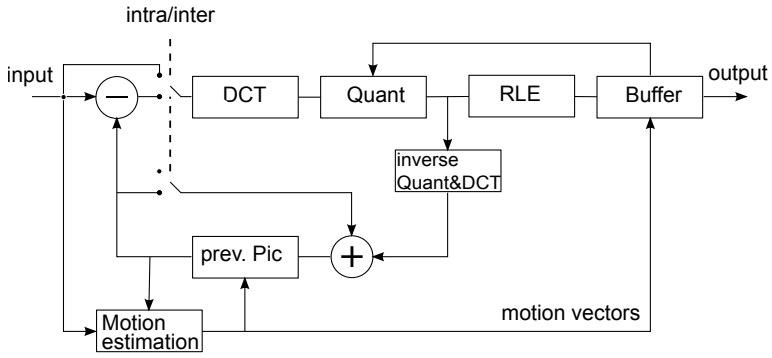

Abbildung 2.4: *Aufbau eines Enkoders (vgl. Ghanbari, M., 2003, S. 56)*

Nachfolger zur Laufzeit nicht bekannt ist und daher nur mit einer Verzögerung gesendet werden könnte.

Eine Folge von Frames, beginnend mit I-Frames und nachfolgenden P/B-Frames, wird Group of Pictures (GOP) genannt. Oft werden auch I-Frames als Intraframes und P/B-Frames als Interframes bezeichnet.

2.6 Motion Estimation

Bei der Motion Estimation wird versucht Bewegungen, die auf zwei Bildern stattgefunden haben, zu erkennen. Dazu wird das Bild in gleich große Blöcke eingeteilt, die als Macroblöcke bezeichnet werden. Um eine Bewegung zu erkennen, wird versucht einen Macroblock aus dem vorherigen Bild im aktuellen Bild wiederzufinden, um eine mögliche Verschiebung zu finden. Wenn das der Fall ist, wird eine Vorhersage aufgrund der Bewegung berechnet. Daher ist es nicht nötig den kompletten Block zu speichern bzw. übertragen, sondern es reicht, wenn man einen Bewegungsvektor ermittelt, der Informationen darüber enthält, wie der Block zu verschieben ist.

2.7 Aufbau eines Enkoders

In der Abbildung 2.4 ist der vereinfachte Aufbau eines Enkoders zu sehen. Ein ankommendes Videosignal läuft zuerst durch die diskrete Cosinustransformation und wird danach quantisiert. Die Daten die man erhält werden für zwei weitere Schritte benötigt. Sie werden einerseits mit der Lauflängenkodierung (in der Abbildung wird die Komponente als RLE[5]

[5] Run length encoding

bezeichnet) komprimiert und andererseits wird die inverse Quantisierung und inverse DCT berechnet und für die Differenzkodierung gespeichert. Man nimmt an dieser Stelle das rekonstruierte Bild, da es im Vergleich zum ursprünglichen Bild Rundungsfehler enthält, die auch beim Dekoder auftreten werden. Da der Dekoder die Differenzkodierung auf Bilder anwendet, die quantisiert, transformiert und rekonstruiert wurden, wird auf diese Weise sichergestellt, dass der En- und Dekoder zum gleichen Ergebnis kommen. Bei einem Intra-Frame wird die Differenzkodierung ausgeschaltet und ein vollständiges Bild wird übertragen. Die Motion Estimation ermittelt aus dem alten wiederhergestellten und dem aktuellen Bild die Bewegungsvektoren, die zusätzlich zum komprimierten Bild in den Buffer geschrieben werden, der daraufhin weiterverarbeitet werden kann.

3 Bewertung der Ausgangssituation

Ein Verständnis für die Software und deren Rahmenbedingungen ist eine wichtige Voraussetzung für eine erfolgreiche Portierung. Durch eine Analyse der Ausgangssituation verschafft man sich einen Überblick über die Gegebenheiten der unterschiedlichen Geräte, deren Betriebssysteme und die Funktionsweise der Software.

3.1 Das Quellsystem

Zunächst werden zwei Geräte vorgestellt auf denen die Software bereits funktionsfähig ist. Da für die Videokodierung viel Rechenleistung benötigt wird, wurden Geräte mit schnellen Prozessoren ausgewählt, deren für eine Videokonferenz interessante Hardwareeigenschaften wie folgt beschrieben werden:

Das Asus P565 nutzt Windows Mobile 6.1 als Betriebssystem und hat einen Marvell PXA930 800MHz ARM Prozessor. Als Arbeitsspeicher stehen 128MB DDR RAM zur Verfügung. Das 2,8” Display hat eine Auflösung von 480x640 Pixel und kann 65000 Farben darstellen. Die Kamera hat 3M Pixel beim Fotografieren und 300K Pixel bei einer Videokonferenz. Beim Aufnehmen von Videos sind 24fps möglich und beim Abspielen 30fps (vgl. Asus, 2009).

Das Samsung i8000 Omnia II benutzt Windows Mobile 6.5. Die Auflösung des 3,7” Displays beträgt 480x800 Pixel. Mit der 5M Pixel Kamera können Bilder mit einer Auflösung von 2560x1920 Pixel und Videos mit 720x480 Pixel aufgenommen werden (vgl. Samsung, 2009). Als Prozessor wird ein ARM SC36410 mit 800MHz verwendet.

Beide Modelle verfügen über eine zweite Kamera, um eine Videokonferenz zu ermöglichen. Die Frontkamera, die im Gegensatz zur Fotokamera auf den Benutzer gerichtet ist, hat in der Regel eine geringere Auflösung. Die hier aufgeführten Angaben beziehen sich nicht auf die Frontkamera, weshalb diese Auflösungen bei einer Videokonferenz nicht erreicht werden können. Zu den Frontkameras werden keine Angaben gemacht.

3.2 Das Zielsystem

Das Zielsystem ist ein iPhone 3Gs der Firma Apple.

Als Betriebssystem wird eine angepasste Version von MacOS X mit verringertem Funktionsumfang verwendet, iPhone OS genannt. MacOS X ist eine proprietäre Version des freien Betriebssystems Darwin, welches unter anderem auf BSD 5 basiert. Somit handelt es sich um ein POSIX-kompatibles Betriebssystem.

Der Prozessor wurde offiziell nie bekanntgegeben, es scheint sich aber um einen ARM Cortex A8, der mit 600MHz getaktet ist, zu handeln (vgl. www.anandtech.com, 2009).

Das iPhone besitzt nur eine Kamera zum Aufnehmen von Fotos und Videos[1]. Diese hat eine Auflösung von 3 Megapixel. Das 3,5” Display hat eine Auflösung 480x320 Pixel bei 163 ppi[2].

Im Gegensatz zu seinem Vorgängermodell, dem iPhone 3G, verfügt das iPhone 3Gs über OpenGL ES 2.0 als Schnittstelle für Multimedia und Spiele.

3.3 Daviko Videokonferenz

Daviko wird von der gleichnamigen Firma daviko GmbH für Windows entwickelt. Es handelt sich dabei um eine Mehrbenutzer-Videokonferenzsoftware, die ohne zentralen Server auskommt. Anstelle von Multipoint Control Units wird ein Peer-to-Peer Design für die Datenübertragung verwendet (vgl. Cycon u. a., 2008, S. 2). In Gruppenkonferenzen werden üblicherweise MCUs dafür eingesetzt die Verteilung der Daten vorzunehmen, wenn es mehr als zwei Teilnehmer gibt. Durch das Verwenden eines Peer-to-Peer Netzwerkes, bei dem alle Gesprächspartner direkt miteinander verbunden sind, werden dagegen keine MCUs zur Datenübertragung benötigt. Dahinter steht auf der mobilen Seite das Moviecast[3] Projekt der HAW Hamburg, das bereits auf die Verwendung von IPv6 ausgelegt ist.

Diese Software wurde bereits auf ein Windows Mobile Gerät portiert. Sie ist die Version, die als Grundlage zur Portierung auf das iPhone verwendet werden soll. Sie hat einen verringerten Funktionsumfang im Vergleich zur PC-Version und der Encoder/Decoder wurde auf kleine Geräte angepasst. Diese Version befindet sich in kontinuierlicher Entwicklung.

[1] Videos sind nur mit dem 3Gs möglich
[2] pixel per inch
[3] moviecast.realmv6.org

3.3.1 Programmiersprache

Daviko wurde größtenteils in C geschrieben, was ein Vorteil für die Portierbarkeit ist. Dieser liegt in der sehr guten Unterstützung fast aller Betriebssysteme und in der Standardisierung der Sprache. Ein weiterer Vorteil gegenüber anderen Sprachen ist die Hardwarenähe und hohe Ausführungsgeschwindigkeit. Ohne diese Eigenschaften wäre die rechenintensive Anwendung auf dem iPhone kaum möglich.

Stellenweise wird aber auch C++ benutzt. Dabei kann es, trotz Standardisierung, zu Problemen auf unterschiedlichen Systemen kommen, wenn Klassen aus der C++ Standardbibliothek oder C++ spezifische Typen benutzt werden. Auf diese Elemente wird in Daviko aber verzichtet. Stattdessen wird nur zur Klassendeklaration auf C++ zurückgegriffen, um die Software klarer strukturieren zu können. Erweiterungen, die in C++ hinzugekommen sind, wie z.B. Namespaces, Referenzen oder Container aus der Standardbibliothek, werden nicht verwendet.

3.3.2 Aufbau der Software

Die Software besteht hauptsächlich aus den folgenden Komponenten:

- Kommunikation
- Encoder/Decoder
- Videoausgabe
- Videoeingabe
- GUI
- Audio

In der Abbildung 3.1 ist die Struktur der Software abgebildet. Zur Verdeutlichung der Funktionsweise sind die Klassen in drei Schichten angeordnet.

Die erste Schicht stellt jeweils die Basisklassen dar. Dabei ist zu beachten, dass “CodecSession“ und “BaseTCP“ abstrakte Klassen sind und dafür sorgen, dass die abgeleiteten Klassen in einem eigenen Thread laufen können. Cdaviko bearbeitet die Events, die entweder vom Benutzer oder von den BaseTCP-Threads kommen.

In der zweiten Schicht sind die Klassen erfasst, die die oben genannten Punkte implementieren. CMyDaviko verwaltet dabei die GUI, das Netzwerk wird von ListenTCP und CallTCP verwaltet und En- und DecoderSession sind für das En- und Decodieren und die Video Ein- und Ausgabe verantwortlich.

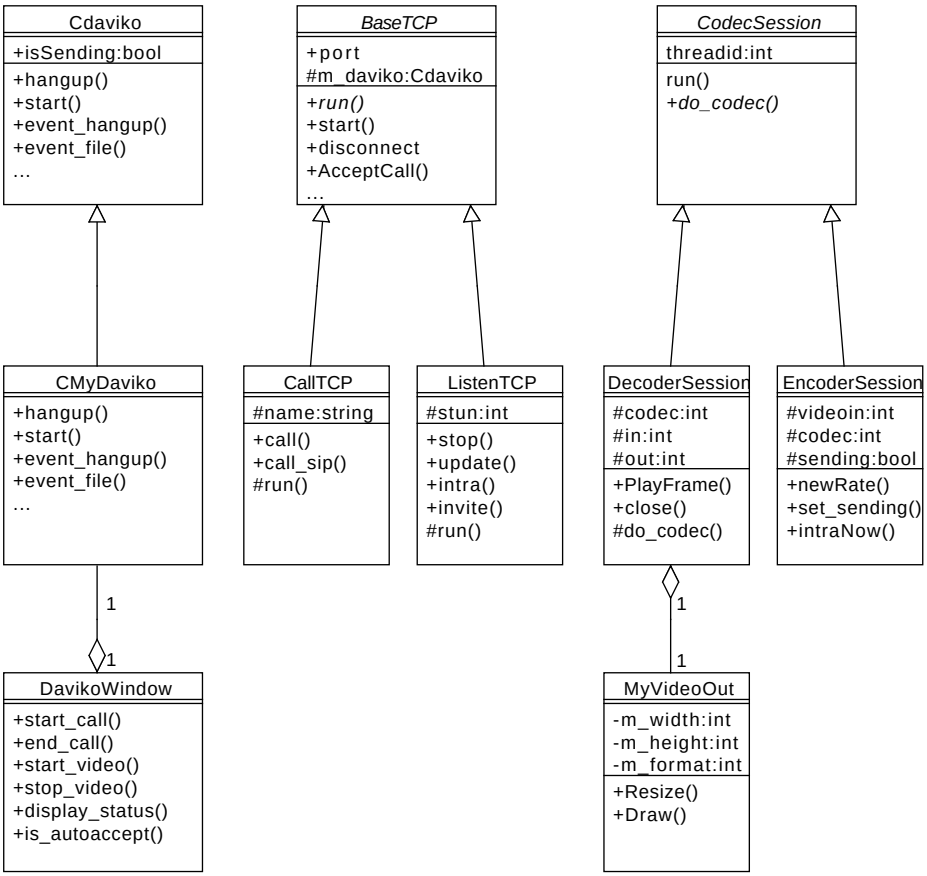

Abbildung 3.1: *Klassendiagramm der Ausgangssoftware*

In der dritten Schicht werden Spezialisierungen der Klassen gezeigt. Die Decoder-Klasse benutzt zum Ausgeben der Videodaten die MyVideoOut-Klasse und CMyDaviko wird von der DavikoWindow-Klasse, die die GUI bildet, instanziiert.

3.3.3 Funktionsweise

Die Funktionsweise ist in der Abbildung 3.2 dargestellt. Bis auf DavikoWindow und MyVideoOut werden diese Klassen als eigenständige Threads gestartet. Das Programm beginnt im DavikoWindow, indem CMyDaviko instanziiert wird.

DavikoWindow bildet die GUI des Programms und ist für die Interaktion mit dem Benutzer zuständig. Die jeweiligen Eingaben werden dann an CMyDaviko weitergereicht.

CMyDaviko übernimmt die Steuerung der Software. Sie kümmert sich um das Starten der einzelnen Threads und die Verarbeitung der Events.

CallTCP wird benötigt, um einen anderen Teilnehmer anzurufen. Dabei wird CallTCP als Thread gestartet und direkt nach dem Senden der Daten wieder beendet. Diese Klasse ist lediglich dazu da, den Anruf vorzubereiten. Sobald die Daten gesendet wurden, wird sie nicht mehr benutzt und alles Weitere (z.B. die Rückmeldung, ob das Gespräch angenommen wurde) wird über die ListenTCP Klasse geregelt.

ListenTCP ist die Netzwerkschnittstelle des Programmes und ist für das Senden und Empfangen der Daten zuständig. Bei empfangenen Daten wird entschieden, ob diese direkt weiterverarbeitet werden können (z.B. Videodaten) oder ob sie als Event an CMyDaviko delegiert werden.

DecoderSession bekommt eingehende Videodaten von der ListenTCP Klasse und dekodiert sie. Da eine Videokonferenz mit mehreren Teilnehmern möglich ist, können 32 Instanzen von DecoderSession an ListenTCP gebunden werden, die dann jeweils einen Teilnehmer verwalten. Die dekodierten Videodaten werden an MyVideoOut weitergereicht.

MyVideoOut stellt die von DecoderSession erhaltenen Videodaten auf dem Bildschirm dar, indem es auf DavikoWindow zugreift.

EncoderSession liest Videodaten aus der Kamera und enkodiert sie. Anschließend werden sie über die ListenTCP Klasse an alle Teilnehmer verschickt.

Ein Programmablauf könnte beispielsweise wie folgt aussehen:

Nach dem Starten des Programmes wird zunächst DavikoWindow und CMyDaviko instanziiert. CMyDaviko erstellt die Threads und Objekte, die für den Betrieb der Software notwendig sind. Nachdem eine Verbindung zum Server aufgebaut worden ist, können Anrufe entgegengenommen werden. Diese werden von ListenTCP als Event an CMyDaviko weitergereicht. In CMyDaviko wird entschieden, wie auf das Event reagiert werden soll. In diesem Fall wird es an DavikoWindow weitergereicht, damit der Benutzer entscheiden kann, ob das Gespräch angenommen werden soll oder nicht.

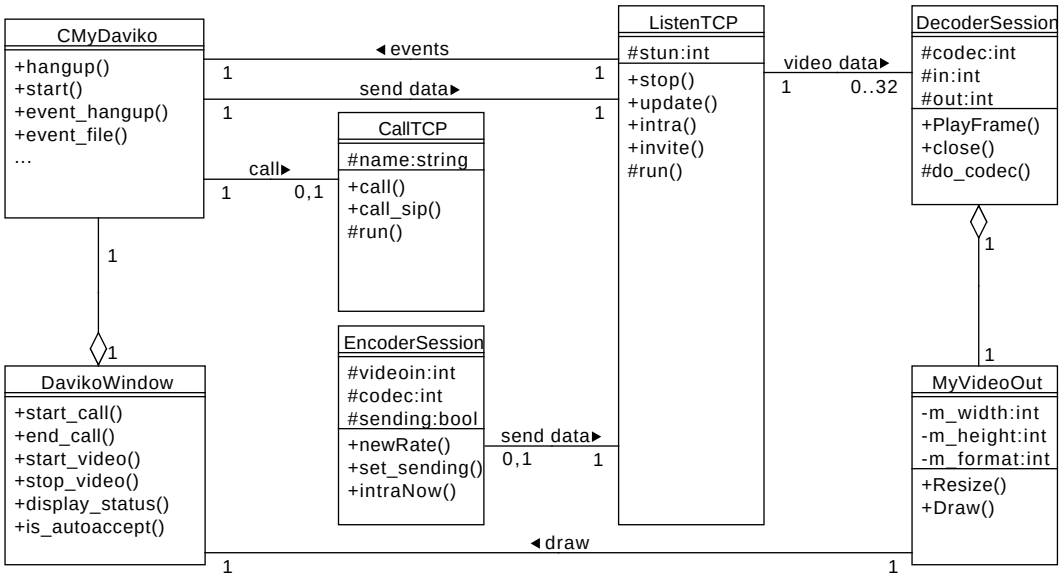

Abbildung 3.2: *Klassenbeziehungen der Ausgangssoftware*

Nimmt der Benutzer das Gespräch an, wird ein Event an CMyDaviko geschickt und das Annehmen des Gespräches wird einleitet. Sobald die Verbindung aufgebaut ist, werden laufend Videodaten empfangen, die von ListenTCP an eine DecoderSession weitergereicht werden. Diese läuft in einem eigenen Thread und dekodiert die Videodaten. Für die Darstellung auf dem Bildschirm ist die MyVideoOut Klasse zuständig, die die VideoDaten über DavikoWindow auf der GUI darstellt.

Parallel dazu werden die Kameradaten in der EncoderSession ausgelesen und enkodiert über ListenTCP verschickt. Sobald einer der Teilnehmer das Gespräch beendet, wird von CMyDaviko die Encoder- und DecoderSession gestoppt und somit das Senden und Empfangen von Videodaten eingestellt.

3.3.4 Netzwerk

Die Netzwerkverbindung wurde mit POSIX Sockets realisiert. Dies ist eine gute Voraussetzung für eine Portierung, da POSIX Sockets auf allen wichtigen Betriebssystemen vorhanden sind. Damit es nicht zu Problemen mit NAT[4]-Routern kommt, ist zusätzlich ein STUN[5]-Service (Rosenberg u. a., 2008) vorhanden, der von außen herauszufinden versucht, wie

[4]Network Address Translation
[5]Session Traversal Utilities for NAT

der Router zu passieren ist. Dazu wird ein STUN-Server benötigt, damit der Client zunächst seine öffentliche IP Adresse in Erfahrung bringen kann. Zusätzlich muss ermittelt werden, um welche Art von NAT-Gateway es sich handelt, um einen geeigneten Weg durch diesen hindurch zu finden.

Da die Software auch ohne den STUN-Service funktioniert, solange kein NAT-Router zwischen den beiden Teilnehmern verwendet wird, spielt er in dieser Arbeit jedoch nur eine untergeordnete Rolle.

3.3.5 Videoausgabe

Die Videoausgabe wurde mit der Simple-DirectMedia-Layer[6] Bibliothek realisiert. Sie steht unter der LGPL und bei ihrer Entwicklung wurde darauf geachtet, eine möglichst hohe Portabilität sicherzustellen.

Die SDL-Bibliothek stellt eine API für Multimedia- und Eingabegeräte zur Verfügung und wird von vielen Projekten hauptsächlich zur Entwicklung von Spielen und Multimediaanwendungen benutzt. SDL ist für alle großen Betriebssysteme verfügbar und erleichtert so das Portieren von Software. Der Vorteil von SDL ist die Hardwareabstraktion, die die Ausgabe realisiert und für jedes Betriebssystem die jeweils beste Ausgabeform wählt.

Da die SDL-Bibliothek schon viele Plattformen unterstützt, die dem iPhone OS ähnlich sind, und der Quellcode frei zugänglich ist und Änderungen zulässt, sollte die Verwendung der SDL-Bibliothek auch auf dem iPhone möglich sein.

3.3.6 Encoder und Decoder

Der En- und Decoder wurde von Daviko selbst entwickelt und stand mir nur als Closed-Source Bibliothek zur Verfügung. Er ist eine auf Echtzeitverarbeitung optimierte H264/MPEG-4 AVC Implementierung benannt DAVC (vgl. Cycon u. a., 2008, S. 2). Der Codec wurde für mobile Geräte optimiert, weshalb er bei der Motion Estimation auf das Integer-Pel Verfahren beschränkt ist.

Der Codec liefert auf einem Asus P735 mit 520 MHz eine Enkodierleistung von 10 fps, während parallel dazu 15 fps dekodiert werden (vgl. Cycon u. a., 2008, S. 1). Da der Codec in ANSI-C geschrieben ist, sollte er sich auch für das iPhone OS kompilieren lassen.

[6] www.libsdl.org

3.3.7 Videoeingabe

Um die Kamera unter Windows Mobile anzusprechen, wird die Hardware direkt verwendet. Zunächst muss die Kamera gefunden werden. Die entscheidende Stelle in der Init-Funktion der Kamera ist in der Datei VideoIn.cpp zu finden und sieht folgendermaßen aus:

Quellcode 3.1: *Kamera suchen unter Windows Mobile*

```
1  DEVMGR_DEVICE_INFORMATION devInfo;
2  devInfo.dwSize = sizeof(devInfo);
3  HANDLE hnd =
4     FindFirstDevice(DeviceSearchByDeviceName, _T("CAM*"), &devInfo);
```

Es wird nach angeschlossenen Geräten gesucht, deren Name mit "CAM*"beginnt. Nachdem die Kamera gefunden wurde, kann sie initialisiert und auf sie zugegriffen werden.

Das Auslesen der Kamera geschieht auf folgende Weise:

Quellcode 3.2: *Kamera auslesen unter Windows Mobile*

```
1  VideoInDeviceHandler *m_handle = (VideoInDeviceHandler*) handle;
2  WaitForSingleObject(m_handle->m_pGrabber->nextFrame,INFINITE);
3  long bufferlen = m_handle->realW*m_handle->realH*4;
4  m_handle->m_pGrabber->GetCurrentBuffer(&bufferlen,(long*)m_handle->grabBuffer);
```

In der ersten Zeile wird ein Pointer auf die Adresse des Gerätes gesetzt. In der zweiten Zeile wird gewartet, bis die Kamera einen neuen Frame aufgenommen hat. Danach wird der Buffer kopiert und kann dann weiter verarbeitet werden. Der genaue Ablauf spielt in den beiden Codeauszügen keine große Rolle. Verdeutlicht werden soll nur, wie unter Windows Mobile die Kamera angesprochen werden kann. Es ist möglich, direkt auf die Hardware zuzugreifen, allerdings mit dem Nachteil, dass keine einheitliche API zum Auslesen der Kamera benutzt wird, die von der Hardware abstrahieren würde.

Die fehlende Hardwareabstraktion und Windows Mobile Systemaufrufe machen eine Portierung dieser Komponente nahezu unmöglich. An dieser Stelle ist es deshalb sinnvoller, das Auslesen der Kamera selbst neu zu programmieren.

3.3.8 GUI

Die GUI wurde mit der Windows Mobile API erstellt. Diese ist Windows Mobile spezifisch und nicht auf anderen Plattformen verfügbar, weshalb eine Neuentwicklung der GUI nötig ist. Da die Software ein Design hat, bei dem die GUI strikt vom restlichen Code getrennt ist, ist sie leicht austauschbar.

Tabelle 3.1: *Übersicht der Komponentenportierbarkeit*

KOMPONENTE	REALISIERUNG	PORTIERBARKEIT
Network	Sockets	sehr gut
Videoausgabe	SDL	gut
Videoeingabe	Hardware	sehr schlecht
Encoder/Decoder	Daviko	gut
GUI	Windows Mobile API	sehr schlecht
Audio	z.Z. nicht implementiert	-

3.3.9 Audio Ein- und Ausgabe

Da sich die Software selbst noch in der Entwicklung befindet, wurde die Audioübertragung noch nicht implementiert. In der ursprünglichen Software von Daviko wird zur Audioübertragung der Speexcodec[7] verwendet. Dieser Codec ist Open-Source und wurde speziell für Audiodaten entwickelt, die menschliche Sprache enthalten. Speex wurde für das Anwendungsgebiet der Internettelefonie optimiert und wird daher häufig in diesem Bereich eingesetzt.

3.4 Zusammenfassung der Portierbarkeit

Der Prozessor des iPhones ist im Vergleich zum Samsung i8000 Omnia II und zum Asus P565 ein wenig langsamer getaktet, sollte aber für die Software dennoch ausreichen. Problematischer ist das Fehlen einer Frontkamera beim iPhone. Daher wird zur Videoaufnahme die normale Kamera genutzt werden müssen. Ein sinnvoller Betrieb der Software ist damit allerdings nicht möglich.

Zur Übersicht, welche Teile der Software überhaupt vernünftig portierbar sind und bei welchen eine Neuentwicklung sinnvoller ist, sind in der Tabelle 3.1 die Portierungsaussichten der einzelnen Komponenten zusammengefasst dargestellt. Die größten Schwierigkeiten stellen die Kamera und die GUI dar.

[7] www.speex.org

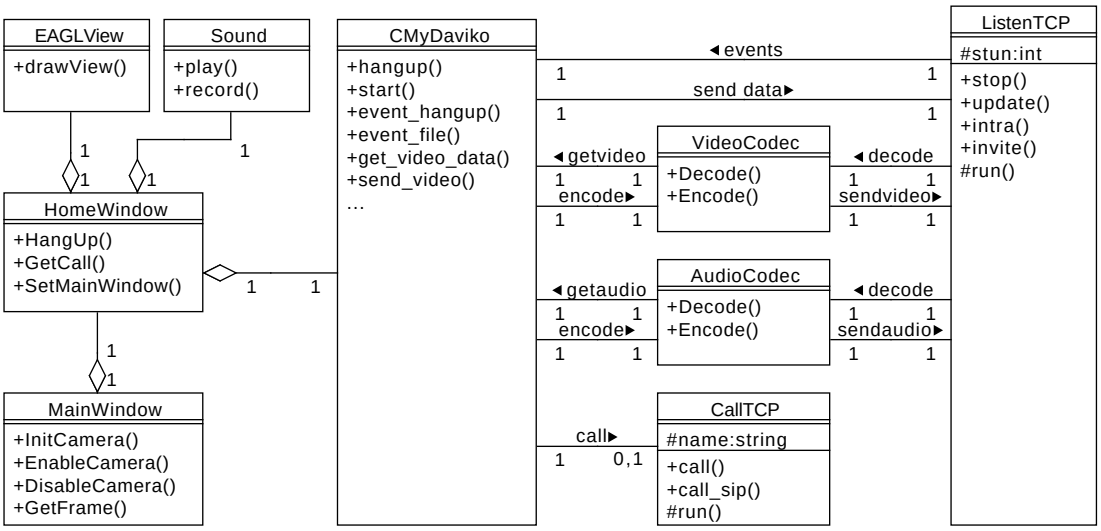

Abbildung 3.3: *Konzept*

3.5 Konzept

Aufbauend auf den vorgenannten Voraussetzungen wurde ein Konzept erstellt, wie die Software implementiert werden soll. Ein entsprechendes Klassendiagramm ist in der Abbildung 3.3 dargestellt.

Auf der linken Seite befinden sich die Klassen, die die GUI bilden. Diese sind für die Darstellung des Videostreams und das Auslesen der Kamera zuständig. Über CMyDaviko wird die Verwaltung der Videokonferenz geregelt und eine einfache Schnittstelle zum Senden oder Empfangen einzelner Frames bereitgestellt. Damit wird der Zugriff einheitlich und macht CMyDaviko vom Auslesen der Kamera und der Videodarstellung unabhängig.

Die Codecs werden in einer Klasse gekapselt, um auch an dieser Stelle den Zugriff einheitlich zu machen. Nach außen bietet diese nur wenige Funktionen. Hauptsächlich soll man mit ihnen Daten en- und dekodieren können.

4 Implementierung

Als Entwicklungsumgebung wird XCode von Apple verwendet. XCode bietet für die Entwicklung von iPhone Programmen einen Simulator an, der die meisten Eigenschaften des iPhones nachbilden kann. Einiges kann aber nicht simuliert werden wie z.B. die Kamera. Um den Code unabhängig von der Plattform entwickeln zu können, stellt XCode zwei Defines zur Verfügung:

Quellcode 4.1: *XCode Defines*

```
1 TARGET_OS_IPHONE
2 TARGET_IPHONE_SIMULATOR
```

Allerdings wird beim Ausführen des Simulators auch TARGET_OS_IPHONE gesetzt, da ein iPhone simuliert wird. Für einige Stellen im Code ist es aber erforderlich zu wissen, ob das Programm auf dem Simulator oder auf dem iPhone läuft, da der Simulator nicht alle Eigenschaften des iPhones simulieren kann. Insbesondere bei der Kamera kommt es dabei zu Problemen. Sobald versucht wird, im Simulator auf die Kamera zuzugreifen, stürzt die Anwendung ab, da der Simulator keine Kamera besitzt. Deshalb werden zusätzlich eigene Defines verwendet, um sicher zwischen Simulator und iPhone unterscheiden zu können und um von der Hardware unabhängig zu sein:

Quellcode 4.2: *Erstellen der eigenen Defines*

```
1 #if TARGET_IPHONE_SIMULATOR
2   #define TARGET_SIMULATOR 1
3 #else
4   #define TARGET_DEVICE 1
5 #endif
```

Im Code werden viele windowsspezifische Typen benutzt, die bei iPhone OS nicht vorhanden sind. Um nicht jede einzelne Variablendeklaration anpassen zu müssen, ist eine Defines.h Datei erstellt worden, in der per Typedef die unbekannten Datentypen angelegt sind. Dieses dient auch der Portierbarkeit, da bei weiteren Portierungen die Datentypen nur in dieser Datei angepasst werden müssen. Dieses Vorgehen ist einfach, wenn bekannt ist, wo sich die Definitionen der Datentypen in der Windows Dokumentation finden lassen. Jedoch gibt es auch undokumentierte Typen, deren Definitionen auf andere Weise beschafft werden müssen. Ein Weg dafür waren Open-Source-Projekte die versuchen, eine Windowsumgebung

zu simulieren, wie z.B. Wine[1] oder ReactOS[2], aus denen man ersehen kann, wie dort einige Datentypen implementiert worden sind.

Dieser vorbereitende Schritt beseitigte die Typenfehler und ergab zugleich einen Überblick darüber, an welchen Stellen noch Änderungen vorgenommen werden mussten.

4.1 Portierung des Netzwerkteils

Durch die Benutzung von POSIX Sockets lagen die Schwierigkeiten nicht so sehr bei der Netzwerkimplementierung, sondern bei den benutzten Threads. Die Threads wurden benutzt, um das Empfangen und Senden asynchron und damit unabhängig von der restlichen Software zu machen.

Die Threads wurden über die Windows API erzeugt und mussten deshalb ersetzt werden. MacOS X bietet dafür zwei Möglichkeiten. Zum einen angebotene NSThreads, die mit Objective-C erstellt werden können und zum anderen POSIX Threads.

Aus Gründen, die der Portierbarkeit der Software dienen, werden POSIX Threads verwendet. Da das Netzwerk mit POSIX Sockets realisiert wurde, bot sich auch der Gebrauch der POSIX Threads an. Dadurch enthielt die komplette Netzwerkkomponente nur noch plattformunabhängigen Code. Ein Nachteil der NSThreads ist zudem die Benutzung von Objective-C, wodurch C/C++ mit Objective-C gemischt werden müsste, was technisch möglich wäre, den Code aber unleserlich und nicht portierbar machen würde.

4.2 Portierung des Encoders/Decoders

Der En- und Decoder ist closed Source Software, was ihn für mich unportierbar macht. Da er jedoch schon erfolgreich unter Linux kompiliert wurde, gibt es eine Version, die auf einem Unix ähnlichen Betriebssystem läuft. Außerdem kann davon ausgegangen werden, dass diese keine Probleme mit dem ARM Prozessor hat, da sie auch schon auf dem Asus P565 und dem Samsung i8000 Omnia II zum Einsatz kam, die ebenfalls einen ARM benutzen.

Auf der Grundlage dieser Version ist ein En- und Decoder auch für das iPhone mit zwei möglichen Zielplattformen kompiliert worden, da das iPhone und der Simulator beide eigene Versionen benötigen. Beide Versionen wurden in das XCode Projekt eingefügt. Zusätzlich werden noch die Headerdateien im Projekt benötigt, damit auf die Bibliothek zugegriffen

[1] www.winehq.org
[2] www.reactos.org

werden kann. Dabei erkennt XCode beim Starten der Software anhand der Zielplattform, welche der beiden Bibliotheken benutzt werden soll.

4.3 Portierung der Videoausgabe

Bei der Videoausgabe geht es um die Darstellung der dekodierten Frames auf dem Display. Die Daten liegen nach der Dekodierung im YCbCr Farbmodell vor und müssen zunächst ins RGB Farbmodell konvertiert werden, da einige Bibliotheken nur dieses Format akzeptieren. Für die Darstellung auf dem Display gibt es mehrere Möglichkeiten, die in diesem Kapitel vorgestellt werden.

4.3.1 Simple DirectMedia Layer

Für die Videoausgabe wurde bisher SDL verwendet. Diese Bibliothek ist für viele Betriebssysteme verfügbar, daher besteht die Möglichkeit, dass sie auch schon für das iPhone existiert. Zur Zeit ist die Version 1.2.14 die stabile Version (vgl. SDL, 2009b), hat aber noch keine offizielle Unterstützung für das iPhone. Eine Portierung wäre zwar möglich gewesen aber in im Rahmen dieser Arbeit zu zeitaufwändig. Außerdem benutzt SDL OpenGL als Ausgabe (vgl. SDL, 2009a), die aber für das iPhone nicht vorhanden ist. Als Alternative zu OpenGL gibt es eine ähnliche Bibliothek, die sich OpenGL ES nennt.

Es gibt einen Versuch, die SDL auf das iPhone zu portieren, indem OpenGL ES verwendet wird. Das Projekt wird von Googles „Summer of Code“ unterstützt und ist nur in einer unstabilen Entwicklungsversion 1.3 vorhanden (vgl. SDL, 2009c). Das bedeutet, dass es eine auf dem iPhone lauffähige Version der SDL gibt, die sich allerdings noch in der Entwicklung befindet. Grundsätzlich sollte Software, die nicht als stabil gekennzeichnet ist, bei einer Portierung gemieden werden, um die Fehlersuche nicht unnötig zu erschweren, für die Portierung der Videoausgabe schien es aber die einfachste Lösung zu sein.

Die Entwicklungsversion ist entweder über das SVN zu bekommen oder man verwendet das Prerelease von der Version 1.3 (vgl. SDL, 2009d). Die von mir verwendete Version ist SDL 1.3 revision 5206. Diese Version ließ sich problemlos kompilieren und funktionierte, hatte aber das Problem, dass die Bibliothek für sich ein eigenes Fenster erwartet, wodurch die SDL den gesamten Bildschirm zur Darstellung nutzt und nicht dafür ausgelegt ist, in eine GUI integriert zu werden. Alle Versuche, die Bibliothek dazu zu bringen, sich in eine GUI zu integrieren, scheiterten, wodurch sich die Portierung schwieriger gestaltete als es zu erwarten war.

Es gibt zwei Möglichkeiten das Problem zu lösen. Man könnte die komplette GUI mit der SDL realisieren. Die Gestaltung wäre ziemlich aufwändig, weil die SDL nicht darauf ausgelegt ist GUIs zu designen. Daher würde auch das Look&Feel des iPhones verloren gehen. Ein weiterer Grund, der gegen die Verwendung der SDL spricht ist, dass diese Version als unstabil gekennzeichnet ist. Die gesamte GUI über eine unstabile Bibliothek zu realisieren, wäre fahrlässig.

Die andere Lösung ist, auf die SDL zu verzichten. Es gibt für das iPhone noch andere Alternativen ein Video darzustellen. Apple stellt eine offizielle API zum Abspielen von Videodaten zur Verfügung, auf die im nächsten Abschnitt 4.3.2 näher eingegangen wird.

4.3.2 MPMoviePlayerController

Mit dem MPMoviePlayerController kann man Video- und Audiodaten abspielen, die als Datei vorhanden sind oder über eine URL aufgerufen werden können.

In der iPhone OS Reference Library wird sie wie folgt beschrieben:

> „An MPMoviePlayerController object defines a full-screen movie player. You can use this class to play back movies stored in your application's bundle or support directories. You can also use it to play movies and audio files loaded from a network-based URL.“ (Apple, 2009a, S. 57)

Aus der Beschreibung wird bereits deutlich, dass es sich dabei um einen vollständigen Movieplayer handelt, der im Vollbildmodus läuft. Das ist für die Portierung der Videoausgabe unpraktisch, da die GUI noch weitere Elemente enthalten soll. Der entscheidende Nachteil ist aber die eingeschränkte Möglichkeit beim Abspielen von Videos. Diese müssen entweder als Datei vorliegen oder über ein URL erreichbar sein. Beides trifft auf einen Videostream, der über das Netzwerk kommt, nicht zu, wodurch die MPMoviePlayerklasse zum Wiedergeben von rohen Videodaten unbrauchbar ist. Ein Abspielen von Daten, die nicht als Datei vorliegen, ist damit nicht möglich.

Diese beiden schwerwiegenden Probleme führen dazu, dass diese Klasse als Videoausgabe bei einer Videokonferenz nicht verwendbar ist. Als Entwickler muss man daher die Videoausgabe selbst gestalten.

4.3.3 OpenGL ES

Eine Möglichkeit ist eine Videoausgabe mittels OpenGL ES, welche folgende Vorteile für das Projekt einer Videokonferenz auf dem iPhone besitzt:

1. Es wird von Apple für das iPhone angeboten.
2. Es ist für schnelle grafische Ausgaben ausgelegt.
3. Es lässt sich gut in GUIs integrieren.
4. Es ist durch das breite Anwendungsspektrum sehr flexibel.

Einen Nachteil stellt nur der höhere Arbeitsaufwand dar, der hauptsächlich dadurch entsteht, dass man OpenGL ES erst konfigurieren muss. Zudem ist es bei OpenGL ES nicht möglich, rohe Videodaten darzustellen. Darauf wird im Abschnitt 4.3.6 noch gesondert eingegangen.

4.3.4 Farbmodell

Nach dem Dekodieren liegen die Daten im YUV Farbmodell vor, nicht aber im benötigten RGB Farbmodell. Diese Umwandlung findet vorher in der SDL statt, die jedoch nicht mehr benutzt wird und sieht in der Theorie wie folgt aus:

$$\begin{aligned} R &= Y + V/0,877 \\ B &= Y + U/0,493 \\ G &= 1,7 \cdot Y - 0,509 \cdot R - 0,194 \cdot B \end{aligned}$$

Diese Art der Umrechnung ist aber ungünstig, da sie Gleitkommaoperationen enthält. Diese für Prozessoren sehr aufwändige Art der Berechnung müsste bei einem Bild für jeden einzelnen Pixel berechnet werden und ist daher ein sehr langsames Verfahren. Es gibt aber auch Berechnungsverfahren, die keine Gleitkommaoperationen benutzen sondern ausschließlich Festkommaoperationen verwenden, wodurch die Geschwindigkeit der Umwandlung erhöht wird. Nach der Umwandlung in das RGB Farbmodell liegen die Daten in einer Form vor, die für OpenGL ES benutzbar ist.

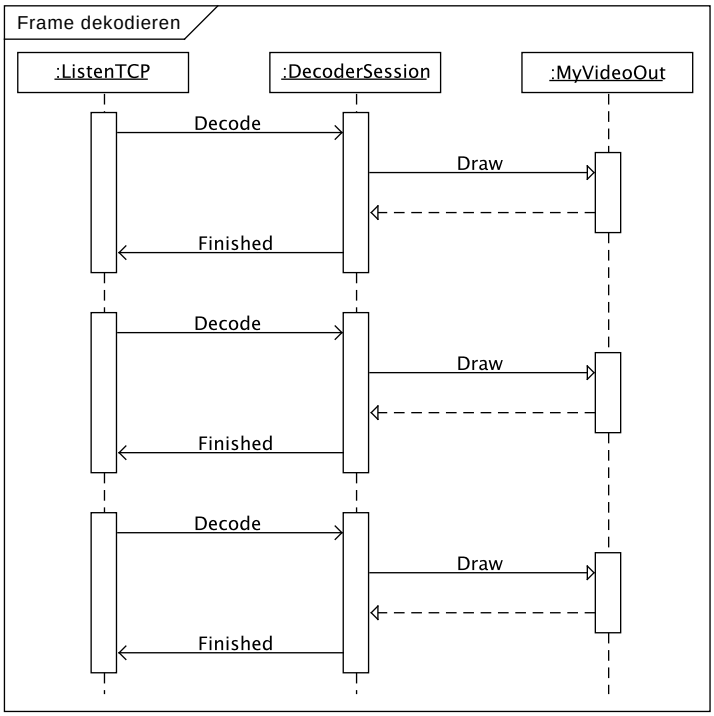

Abbildung 4.1: *Synchroner Dekodierablauf*

4.3.5 Implementierung der Videoausgabe

Bei der Implementierung werden einige Änderungen am Software-Design vorgenommen. Ursprünglich wurde jedes im Netzwerk-Thread empfangene Videopaket an den Decoder-Thread weitergereicht. Dort wurde es dekodiert und auf das Display ausgegeben, während der Netzwerk-Thread auf den Decoder-Thread wartete. Erst nachdem der Decoder seine Berechnungen abgeschlossen hatte, konnte der Netzwerk-Thread weiterlaufen. Grundsätzlich wäre der Ablauf auch über einen einfachen synchronen Methodenaufruf zu realisieren gewesen. Zur Verdeutlichung ist der Ablauf noch einmal in der Abbildung 4.1 dargestellt.

Das Designziel für die Unterteilung in unterschiedliche Threads lag darin, Prioritäten für die einzelnen Aufgaben verteilen zu können.

Dieses Design hat aber einige Nachteile. Zunächst ist es schwierig, die einzelnen Threads über Prioritäten zu koordinieren, insbesondere, wenn zusätzlich noch NSThreads benutzt werden. POSIX Threads kann man eine Priorität von 0 bis 31 zuweisen, während die NS-Threads von 0.0 bis 1.0 eingestellt werden können.

Außerdem muss der Netzwerk-Thread immer auf das komplette Decodieren und Darstellen

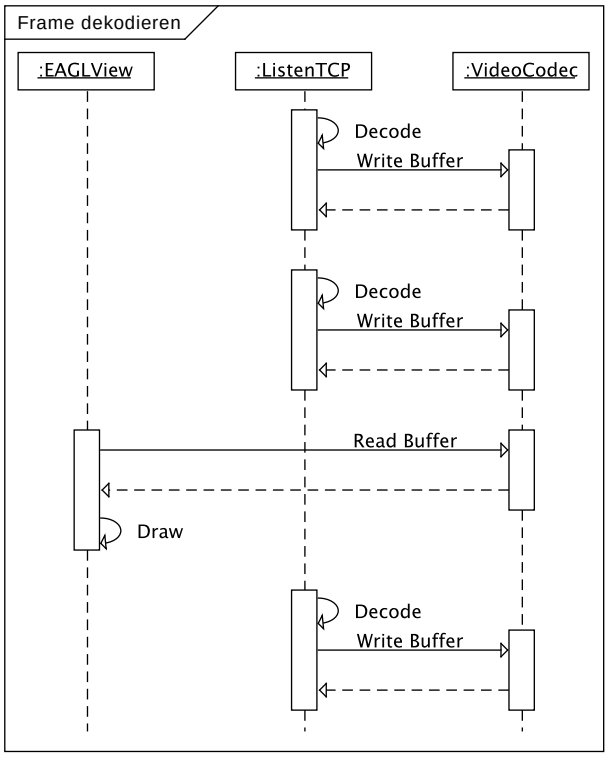

Abbildung 4.2: *Asynchroner Dekodierablauf*

warten, was, je nach der Größe der Frames, einige Zeit kosten kann. Da jeder Frame verarbeitet wird, ist es nicht möglich selbständig zu entscheiden, wie hoch die Framerate vom Display sein soll. Das kann bei zu vielen ankommenden Frames zu Performanceproblemen führen.

Die vorgenommenen Änderungen sind in dem Sequenzdiagramm der Abbildung 4.2 aufgeführt. Die Daten werden nicht mehr an einen anderen Thread weitergereicht, sondern direkt im Netzwerk-Thread dekodiert. Dabei werden die Frames nur in einem Buffer gespeichert und nicht direkt auf dem Display dargestellt. Das Dekodieren findet in einer neuen Klasse – VideoCodec – statt, in der sich auch der Buffer befindet und die die alte MyVideoOut Klasse ersetzt, wie in der Abbildung 4.3 dargestellt. Die Weiterverarbeitung der Daten führt die GUI selbstständig aus, indem sie die Daten in gewissen Abständen lädt und darstellt. Da nur noch die CMyDaviko-Klasse Zugriff auf VideoCodec hat, gibt es in CMyDaviko eine neue Methode, über die die GUI an die Daten kommen kann:

Quellcode 4.3: *Neuer Videodatenzugriff*

```
1  virtual unsigned char* get_video_data(int*w,int*h)
```

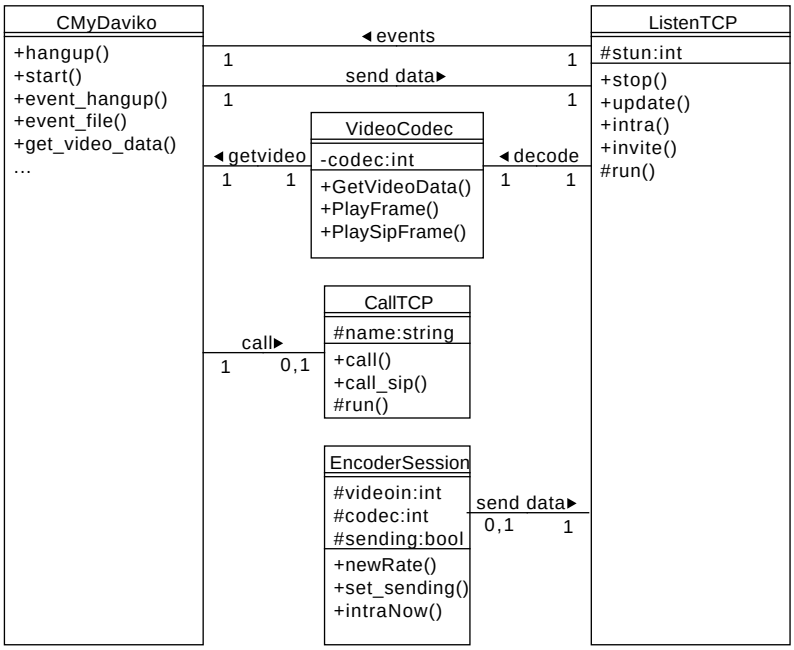

Abbildung 4.3: *Dekodieren mit VideoCodec*

Indem diese Daten mit einem Timer regelmäßig aus dem Buffer ausgelesen werden, kann eine Einstellung der Framerate unabhängig von der Menge der ankommenden Pakete erfolgen.

4.3.6 OpenGL ES Initialisierung

Quellcode 4.4: *OpenGL ES initialisieren*

```
1 - (void) InitOpenGLES{
2   glViewport(0, 0, backingWidth, backingHeight);
3   glMatrixMode(GL_PROJECTION);
4   glLoadIdentity();
5   glMatrixMode(GL_MODELVIEW);
6
7   glVertexPointer(2, GL_FLOAT, 0, spriteVertices);
8   glEnableClientState(GL_VERTEX_ARRAY);
9   glTexCoordPointer(2, GL_SHORT, 0, spriteTexcoords);
10  glEnableClientState(GL_TEXTURE_COORD_ARRAY);
11
12  glGenTextures(1, &texture);
13  glBindTexture(GL_TEXTURE_2D, texture);
14  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

```
15    glRotatef(180.0f, 0.0f, 0.0f, 1.0f);
16 }
```

In OpenGL ES gibt es keine Möglichkeit, die einzelnen Pixel einfach auf dem Bildschirm darzustellen. Stattdessen müssen die Daten zunächst in eine Textur umgewandelt werden. Üblicherweise wird eine solche Textur einmal angelegt und danach nicht weiter verändert. In der von mir gewählten Anwendung wird die Textur jedoch dafür verwendet, kontinuierlich neue Daten hineinzuladen und so eine Videoausgabe zu erzeugen. Die Einstellungen dafür sind ab der Zeile 12 zu finden. Dort wird eine neue Textur angelegt, in die später die Daten eingefügt werden können.

Um eine, je nach Anforderung, optimale Darstellung von Texturen zu erreichen, können zudem die Texturfilter konfiguriert werden. Dieses geschieht in der Zeile 14 über die Funktion glTexParameteri. In diesem Fall wird auf hohe Geschwindigkeiten optimiert.

Abschließend muss die gesamte Ausgabe um 180 Grad gedreht werden, da OpenGL ES das kartesische Koordinatensystem benutzt, während die dekodierten Bilder ihren Ursprung aber in der oberen linken Ecke haben. Nach dem Kopieren der Daten in die Textur würde diese deshalb ohne die Drehung verkehrt herum auf dem Display dargestellt werden. Dieser Effekt hätte auch schon beim Kopieren der Daten verhindert werden können, dafür aber eine umständliche und langsamere Kopierfunktion erfordert.

4.3.7 OpenGL ES Ausgabe

Die Videodaten mittels OpenGL ES darzustellen ist für den Programmierer aufwändiger zu implementieren als mit speziell dafür ausgelegten Bibliotheken. OpenGL ES ist darauf ausgelegt, komplexe 3D-Szenen in Echtzeit darzustellen. Bei einer Videodarstellung mit OpenGL ES muss das Video daher auf ein Objekt im 3D-Raum projeziert werden, damit es dargestellt werden kann. Der dazu verwendete Code wirkt deshalb ein wenig unübersichtlich.

Dieser Code wird von einem Timer, dessen Geschwindigkeit in der Config.h eingestellt werden kann, regelmäßig ausgeführt. Die Videodaten werden in der Zeile 6 aus dem Dekoder ausgelesen. In den Zeilen 10 und 11 wird die Ausgabe auf den Framebuffer gesetzt. Die Zeilen 13 bis 52 werden dafür nicht benötigt. Sie dienen dazu, das aktuelle Bild so zu verzerren, dass es von OpenGL ES dargestellt werden kann. Diese Größenänderung ist erforderlich, da OpenGL ES nur Texturen akzeptiert, deren Höhe und Weite eine Potenz von 2 ist.

Für die Größenänderung, die in der Zeile 36 stattfindet, wird auf die Quartz Bibliothek zurückgegriffen. Dieser Code wird hier nicht weiter erläutert, da er für eine Videoausgabe nicht

zwingend notwendig ist. Er ermöglicht aber ein Darstellen von Bildern mit beliebiger Größe.

Die Größe der Textur wird in der Config.h unter TEXTURE_WIDTH und TEXTURE_HEIGHT festgelegt, wobei das Bild, je nachdem wie man die Texturgröße wählt, gestreckt oder gestaucht wird. Das Stauchen erfolgt schneller als das Strecken, ist dafür aber mit einem Qualitätsverlust behaftet.

Sobald das Bild als Textur vorhanden ist, kann es nach Belieben in eine andere Größe gestreckt werden. In diesem Fall wird das Bild immer auf die Größe des GUI Elementes getreckt, auf dem es dargestellt wird. Wird die Texturgröße kleiner als das ursprüngliche Bild gewählt, erhält man allerdings einen Qualitätsverlust.

Das eigentliche Zeichnen des Bildes findet erst in der Zeile 55 statt, indem es in die aktuelle Textur geschrieben wird. Danach kann die Textur in den Zeilen 63 bis 65 in den Framebuffer gerendert werden, der dann auf dem Display dargestellt wird.

Quellcode 4.5: *OpenGL ES implementieren*

```
1  - (void)drawView {
2
3    unsigned char *data;
4    int w,h;
5
6    data=[[HomeWindow GetCurrent]Daviko]->get_video_data(&w, &h);
7    if(data==NULL||w==0||h==0)
8      return;
9
10   [EAGLContext setCurrentContext:context];
11   glBindFramebufferOES(GL_FRAMEBUFFER_OES, viewFramebuffer);
12
13   CGDataProviderRef provider=CGDataProviderCreateWithData(NULL,data,w*h*4,NULL);
14   const int bitsPerComponent = 8;
15   const int bitsPerPixel = 4 * bitsPerComponent;
16   const int bytesPerRow = 4 * w;
17   CGColorSpaceRef colorSpaceRef = CGColorSpaceCreateDeviceRGB();
18
19 #if ALPHA_POS==0
20   CGBitmapInfo bitmapInfo = kCGBitmapByteOrderDefault|kCGImageAlphaFirst;
21 #elif ALPHA_POS==3
22   CGBitmapInfo bitmapInfo = kCGBitmapByteOrderDefault;
23 #else
24   Log(TRUE,"EAGLView::drawView() -> #define ALPHA_POS has an unsupported value");
25 #endif
26
27   CGColorRenderingIntent renderingIntent = kCGRenderingIntentDefault;
28   CGImageRef spriteImage= CGImageCreate(w, h, bitsPerComponent,
29                                         bitsPerPixel, bytesPerRow,
```

```
                                        colorSpaceRef, bitmapInfo,
                                        provider, NULL,
                                        NO, renderingIntent);
    CGDataProviderRelease(provider);
    CGColorSpaceRelease(colorSpaceRef);

    spriteImage=resizeCGImage(spriteImage,TEXTURE_WIDTH,TEXTURE_HEIGHT);
    if(spriteImage) {
      const size_t width = CGImageGetWidth(spriteImage);
      const size_t height = CGImageGetHeight(spriteImage);
      GLubyte* spriteData = (GLubyte*) calloc(width * height * 4, sizeof(GLubyte));
      const CGContextRef spriteContext =
                          CGBitmapContextCreate(spriteData, width,
                                                height, 8, width * 4,
                                                CGImageGetColorSpace(spriteImage),
                                                kCGImageAlphaPremultipliedLast);

      CGContextDrawImage(spriteContext, CGRectMake(0.0, 0.0,
                                                   (CGFloat)width,
                                                   (CGFloat)height),
                                                   spriteImage);
      CGContextRelease(spriteContext);
      CGImageRelease(spriteImage);

      //write to OpenGL texture:
      glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height,
                   0, GL_RGBA, GL_UNSIGNED_BYTE, spriteData);
      free(spriteData);
      glEnable(GL_TEXTURE_2D);
      glBlendFunc(GL_ONE, GL_ONE_MINUS_SRC_ALPHA);
      glEnable(GL_BLEND);
    }

    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
    glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
    [context presentRenderbuffer:GL_RENDERBUFFER_OES];

}
```

4.4 Auslesen der Kamera

Das Auslesen der Kamera ist der schwierigste Teil für die Ausgestaltung der Videokonferenz. Das Problem liegt dabei an den unzureichenden Bibliotheken, die Apple zur Verfügung stellt.

Um auf die Kamera zugreifen zu können gibt es die UIImagePickerController-Klasse, die dem Entwickler jedoch nur sehr wenige Möglichkeiten zur Verfügung stellt. Sie legt beim Instanzieren der Klasse ein neues Fenster an, das über die Anwendung gelegt wird. Dieses enthält im Normalfall einen Button zum Auslösen und ein Preview des aktuellen Kamerabildes. Im Anschluss daran verliert der Entwickler die Kontrolle über den weiteren Ablauf. Durch das Betätigen des Auslösers wird ein Event geworfen, welches intern verarbeitet wird und dafür sorgt, dass ein Bild aufgenommen wird. Sobald dieser Vorgang abgeschlossen ist, wird das Fenster der Kamera geschlossen und ein Event wird an den Entwickler zurückgegeben. Erst an dieser Stelle erhält der Entwickler die Kontrolle zusammen mit dem aufgenommenen Foto zurück.

Das iPhone 3GS kann auch Videos aufnehmen, wobei der Ablauf ähnlich ist. Das abschließende Event kommt dabei nicht von der Kamera, sondern vom Benutzer sobald er die Aufnahme stoppt.

Problematisch ist dabei, dass sich das Verhalten nicht beeinflussen lässt. Die Klasse ist daher für eine Videokonferenz nicht zu gebrauchen, da das eigentliche Bild nicht selbst dargestellt werden soll, sondern nur dessen Daten zum Versenden benötigt werden. Mit der offiziellen iPhone API ist es somit nicht möglich, die Kamera direkt anzusprechen und einzelne Frames herauszulesen.

4.4.1 Problemstellung

In der offizielle Dokumentation gibt es keine Lösungsvorschläge und auch in der Fachliteratur gibt es keine Lösungen zu diesem Problem. Jedoch ließen sich im Internet einige Foren finden, in denen andere Entwickler das gleiche Problem hatten und diskutierten. In einem Forum[3] lies sich ein Ansatz finden, der das Problem zum Teil lösen konnte.

Zwar wird auch hier zwangsläufig ein Fenster auf dem Bildschirm geöffnet, jedoch lässt sich dieses Fenster in der Größe und Position verändern.

Die einzige Möglichkeit auf einzelne Frames unmittelbar zuzugreifen, ist das Preview der Kamera, das ausgelesen werden kann. Da die UIImagePickerController-Klasse jeglichen Zugriff auf interne Daten verbietet, muss man eine Möglichkeit finden, auf die internen Daten zuzugreifen ohne die UIImagePickerController-Klasse zu benutzen. Dazu muss jedoch die Struktur der internen Klassen bekannt sein, was nicht der Fall ist, da dem Entwickler der Quellcode nicht zur Verfügung gestellt wird.

Um zumindest den Aufbau der GUI, die das Preview enthält, zu verstehen, wurde eine Funktion entwickelt mit der sich dieser ermitteln lässt:

[3] http://modmyi.com/forums/iphone-ipod-touch-sdk-development-discussion/642261-cameracontroller-3gs-3-0-a-3.html

Quellcode 4.6: *Klassenhierarchie ermitteln*

```
-(void)MyInspectView: (UIView *)theView depth:(int)depth path:(NSString *)path {
  NSMutableString *pad = [NSMutableString stringWithCapacity:1024];
  for (int i=0; i<depth; i++) {
    [pad appendString:@"  "];
  }
  NSLog([NSString stringWithFormat:@"%@Class: %@",pad,[theView description]]);
  for (int i=0; i<[theView.subviews count]; i++) {
    NSString *subPath = [NSString stringWithFormat:@"%@/%d",path,i];
    [self MyInspectView:[theView.subviews objectAtIndex:i]
                  depth:depth+1 path:subPath];
  }
}
```

In der Abbildung 4.4 ist das Ergebnis dieser Funktion für die UIImagePickerController-Klasse als Klassendiagramm dargestellt.

Wie sich erkennen lässt, ist die PLCameraView-Klasse entscheidend dafür verantwortlich, auf die Kamera zugreifen und das Preview auf dem Display darstellen zu können. Diese Klasse wird von Apple intern benutzt, jedoch den Entwicklern nicht zur Verfügung gestellt. Es ist deshalb kaum möglich, diese Klasse zu verwenden, da keine Headerfiles zu ihr bereitgestellt werden. Auch eine öffentliche Dokumentation der Klasse existiert nicht.

4.4.2 Headerfiles und Frameworks

Die Frameworks müssen erst einmal zu dem XCode Projekt hinzugefügt werden. Normalerweise liegen die Frameworks in dem Ordner

```
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS3.1.2.sdk/System/Library/Frameworks
```

und können danach benutzt werden. Da es sich bei den benötigten Bibliotheken um inoffizielle Frameworks handelt, liegen sie an einem anderen Ort als die normalen Frameworks und sind in dem Ordner

```
/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS3.1.2.sdk/System/
Library/PrivateFrameworks/
```

zu finden. Benötigt werden CoreSurface.framework und PhotoLibrary.framework. Da zu diesen Bibliotheken keine Headerfiles existieren, muss man sie selbst generieren.

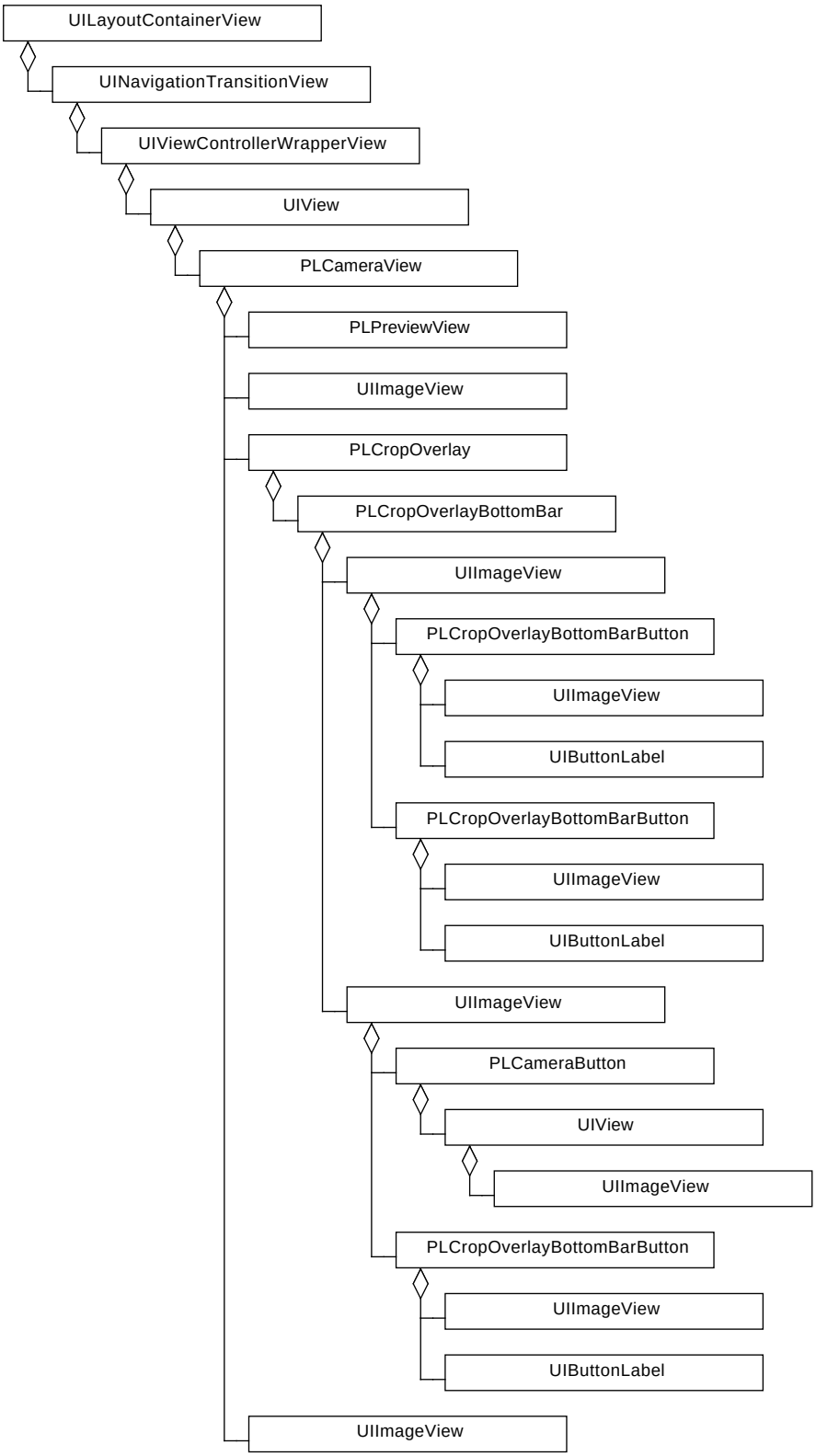

Abbildung 4.4: *UIImagePickerController Hierarchie*

Es gibt ein Open-Source Programm[4], dass aus den kompilierten Bibliotheken die Klassendeklaration erstellen kann. Mit diesem Programm ist es möglich, den Zugriff auf die versteckten Bibliotheken nachträglich zu erhalten.

Der Aufruf sieht z.B. für die PhotoLibrary folgendermaßen aus:

```
./class-dump -H /Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS3.1.2.sdk/
System/Library/PrivateFrameworks/PhotoLibrary.framework/PhotoLibrary
```

Es erstellt im aktuellen Verzeichnis die Headerfiles zu dem angegebenen Framework.

4.4.3 Initialisierung der Kamera

Da nicht mehr die offizielle UIImagePickerController-Klasse benutzt wird, die dem Entwickler sehr viel Arbeit abnimmt, muss die Initialisierung selbst vorgenommen werden.

Quellcode 4.7: *Kamera initialisieren*

```
1  -(void) InitCamera{
2
3    if (!CameraAvailable){
4      return;
5    }
6    cameraController =
7      [(id)objc_getClass("PLCameraController")
8      performSelector:@selector(sharedInstance)];
9    [cameraController setDelegate:self];
10   [cameraController setFocusDisabled:NO];
11   [cameraController setDontShowFocus:YES];
12   [cameraController retain];
13   [cameraController setFocusDisabled:TRUE];
14   [cameraController _setCameraMode:0 force:TRUE];
15
16   UIView *previewView =
17     [cameraController performSelector:@selector(previewView)];
18   [previewView retain];
19
20   //Size and position:
21   CGSize size;
22   size.width=DST_WIDTH;
23   size.height=DST_HEIGHT;
24   CGPoint point;
25   point.x=PLCC_POS_X;
```

[4] www.codethecode.com/projects/class-dump/

```
26    point.y=PLCC_POS_Y;
27    CGRect rect;
28    rect.size=size;
29    rect.origin=point;
30
31    previewView.frame=rect;
32    [window addSubview:previewView];
33    [cameraController performSelector:@selector(startPreview)];
34
35    sleep(2);
36    [window makeKeyAndVisible];
37
38 }
```

Zunächst wird in den Zeilen 6 bis 8 eine Instanz des PLCameraControllers erstellt. Die unübliche Form der Instanziierung liegt an der Verwendung des Singleton-Patterns, mit dem sichergestellt wird, dass immer nur ein PLCameraController existiert und auf die Kamera zugreift.

Da es nicht möglich ist, die Kameradaten direkt zu bekommen, muss ein Element auf der GUI erstellt werden, auf dem die Kamera ein Preview des Bildes darstellen kann. Dies geschieht in Zeile 16, indem ein neues UIView erstellt wird. In den folgenden Zeilen wird die Größe und Position des UIView-Elements festgelegt.

In den Zeilen 32 und 33 wird das UIView-Element zum aktuell geöffneten Fenster hinzugefügt und das Preview der Kamera gestartet.

4.4.4 Wahl des Formates

Zudem ergibt sich ein weiteres Problem. Da die Kameradaten aus dem UIView herausgelesen werden, das den Preview darstellt, erhält man die Daten auch mit der Auflösung des Previews. Weil das Preview aber nicht den ganzen, sondern lediglich einen kleinen Teil des Bildschirmes einnehmen soll, wird auch das Kamerabild kleiner.

Außerdem wird das Bild bei Größenänderungen gestaucht. Das ursprüngliche Format des Previews hat 320x480 Pixel, was einem gedrehten HVGA[5] Format entspricht. Da das iPhone aber nicht gedreht werden soll, ist es schwierig ein geeignetes Format für die Darstellung zu finden.

[5] Half VGA

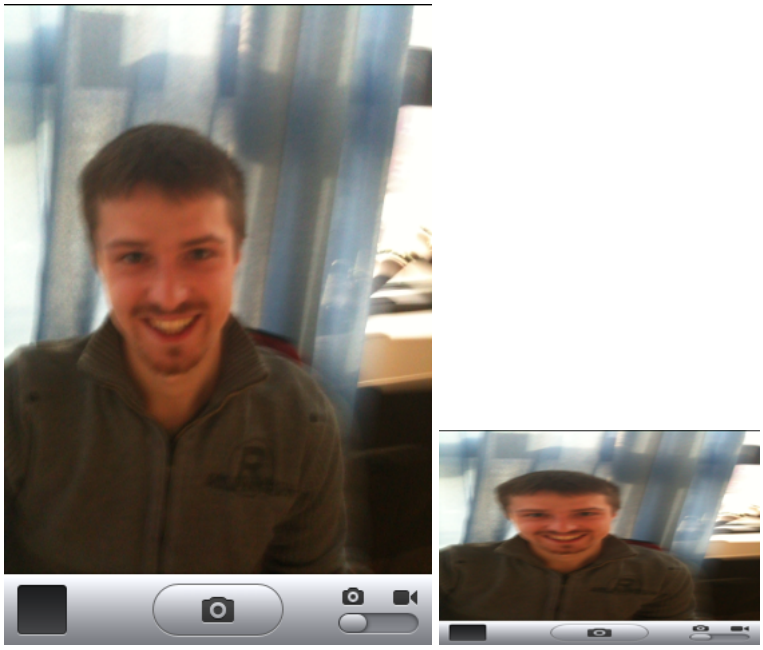

Abbildung 4.5: *Vergleich von 320x480 und 240x160*

Daviko verwendet als Auflösung 240x160 Pixel, was QVGA[6] entspricht. Beim Verändern von 320x480 Pixel auf 240x160 Pixel wird das Bild aber in der Höhe zusammengestaucht und die Bilder sehen verzerrt aus:

$$\begin{aligned} \text{Stauchung der Breite} &= 320/240 = 1,3 \\ \text{Stauchung der Höhe} &= 480/160 = 3 \\ \text{Stauchungsverhältnis} &= 3,0/1,3 = 2,3 \end{aligned}$$

Die Höhe des Bildes wird 2,3 mal stärker gestaucht als dessen Breite, wodurch das Bild viel zu stark verzerrt wird, wie sich in der Abbildung 4.5 ersehen lässt.

Da sich das Verzerren des Bildes nicht verhindern lässt, fällt die Entscheidung auf ein quadratisches Format, das zwar unüblich ist, aber besser wirkt als ein hochkantiges Format. Damit das Preview nicht zu viel Platz auf dem Display beansprucht, sollte es nicht breiter als 160 Pixel sein. Aus technischen Gründen, auf die in Abschnitt 4.5 detaillierter eingegangen wird, wurde ein Format von 160x160 Pixel ausgewählt. Das Ergebnis zeigt die Abbildung 4.6.

[6] Quarter VGA

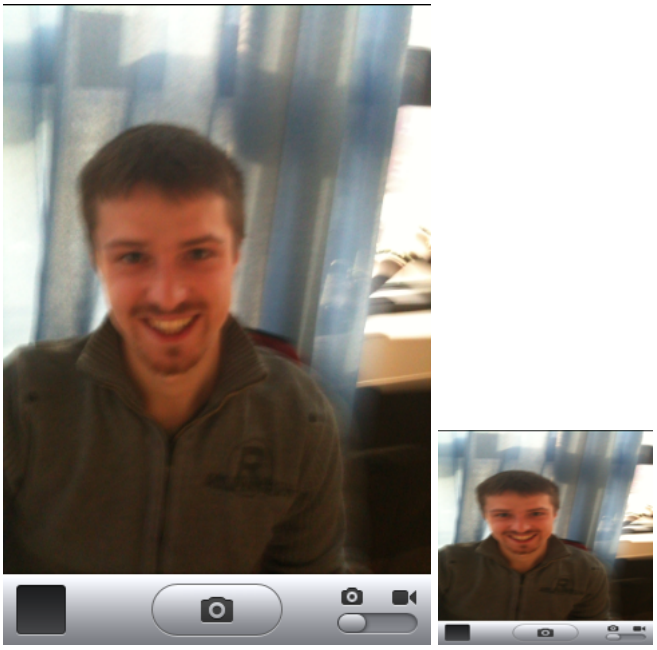

Abbildung 4.6: *Vergleich von 320x480 und 160x160*

Prinzipiell kann das Format aber in der Config.h beliebig eingestellt werden, solange Höhe und Breite je nach Enkodereinstellung durch 8, 16 oder 32 teilbar sind.

4.4.5 Auslesen der Kameradaten

Um an die aktuellen Kameradaten zu kommen, werden sie direkt von dem Display gelesen.

Quellcode 4.8: *Kamera auslesen*

```
1  - (void)frame{
2
3    if ([homewindow IsSending]==false){
4      return;
5    }
6
7    CoreSurfaceBufferRef* coreSurfaceBuffer = (const void**)
8            [cameraController performSelector:@selector(_createPreviewIOSurface)];
9
10   if (!coreSurfaceBuffer){
11     Log(FALSE,
12       "MainWindow::frame() -> Couldn not create the CoreSurfaceBufferRef");
13     return;
```

```
14    }
15    Surface* surface=[[Surface alloc]initWithCoreSurfaceBuffer:coreSurfaceBuffer];
16    [surface lock];
17
18    const unsigned int src_height = surface.height;
19    const unsigned int src_width = surface.width;
20    const unsigned int alignmentedBytesPerRow = (src_width * 4);
21    const unsigned int dst_width = DST_WIDTH;
22    const unsigned int dst_height = DST_HEIGHT;
23    const unsigned char *src_pixels = (unsigned char*)surface.baseAddress;
24
25    getMiddle((Color*) src_pixels, src_width, src_height,
26              (Color*) dst_pixels, dst_width, dst_height);
27
28    if(!IncommingPixels){
29      IncommingPixels=(unsigned char*)malloc(alignmentedBytesPerRow*src_height);
30    }
31    colorConvRGB32toYUV420(IncommingPixels, dst_width,dst_height,
32                           0, 0, dst_pixels, dst_width,dst_height);
33
34    [surface unlock];
35    [surface release];
36    CFRelease(coreSurfaceBuffer);
37
38    [homewindow Daviko]->send_video(IncommingPixels,dst_width, dst_height);
39  }
```

Zunächst wird dafür in der Zeile 7 mit dem PLCameraController ein neues Surface erstellt, das üblicherweise dazu benutzt wird, ein Preview des aktuellen Kamerabildes auf dem Display darzustellen. Zurück bekommt man die Adresse der Bilddaten. Um mit diesen Bilddaten arbeiten zu können, muss ein Surface-Object wie in Zeile 15 angelegt werden.

Es gibt die Alternativen, die Bilddaten für das neue Surface-Objekt zu kopieren, oder darauf zu verzichten und sich die Daten mit anderen Objekten zu teilen. Das Teilen ist schneller, weil es das Kopieren der Daten überflüssig macht. Da es sich dabei allerdings um einen kritischen Bereich handelt, muss das Surface beim Bearbeiten mit lock/unlock geschützt werden.

Durch die getMiddle Funktion kann das Videoformat unabhängig von der Größe auf dem Display gewählt werden. Ist das Preview auf dem Display größer als das gewünschte Darstellungsformat, kann mit dieser Funktion das Darstellungsformat aus der Mitte herausgeschnitten werden.

In der Zeile 31 werden anschließend die Daten in das vom Codec erwartete YUV-Format umgewandelt. Nach der Umwandlung können die Daten in der Zeile 38 gesendet werden.

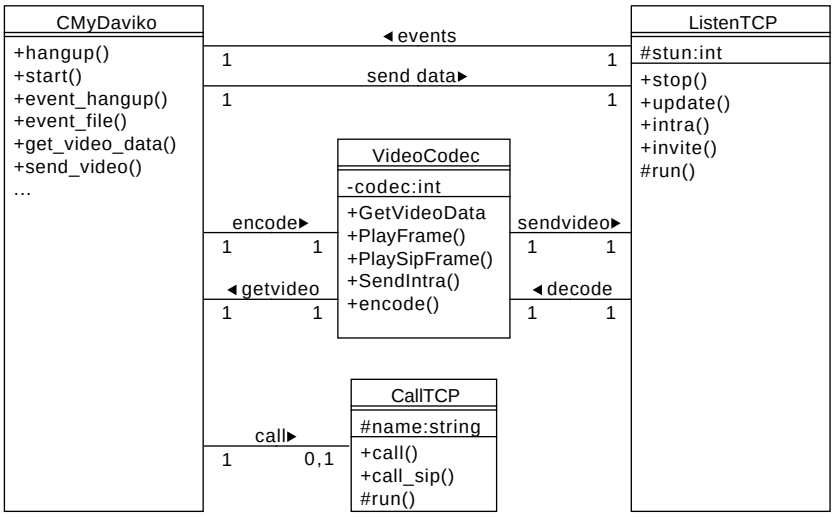

Abbildung 4.7: *VideoCodec Beziehungen*

4.4.6 Softwaredesign

Durch die Probleme beim Auslesen der Kamera muss auch das Design der Software an die neuen Gegebenheiten angepasst werden. In der ursprünglichen Software hat der Encoder-Thread die Daten aus der Kamera herausgelesen und sie anschließend enkodiert und verschickt. Dieser Teil lief daher unabhängig von der restlichen Software.

Weil aber die Kameradaten in der GUI ausgelesen werden müssen, passt das alte Design nicht mehr. Es ist zwar möglich, die Daten herauslesen zu lassen und dann an den Encoder-Thread zu übergeben, dieses Vorgehen würde das Design aber unnötig verkomplizieren.

Das gesamte Enkodieren wird daher aus dem Encoder-Thread in die VideoCodec-Klasse verschoben und so von dem Auslesen der Kamera getrennt. Außerdem gibt es dadurch nur noch eine Klasse für das En- und Dekodieren, auf die allein ListenTCP und CMyDaviko Zugriff haben. Das Ganze ist in der Abbildung 4.7 dargestellt. In CMyDaviko gibt es zudem eine neue Methode, über welche die GUI die Videodaten senden kann:

Quellcode 4.9: *Versenden der Videodaten*

```
1 virtual void send_video(unsigned char *yuv,const int breite, const int hoehe);
```

Durch diese Änderung wird die komplette En- und Decoder-Thread Struktur vollständig aufgelöst.

4.5 Entwicklung einer GUI

Die GUI wurde mit der Windows Mobile API erstellt und war deshalb für das iPhone unbrauchbar. An dieser Stelle war die einzige Möglichkeit eine Neuimplementation der GUI.

Dafür stellt Apple eine eigene API zur Verfügung, Cocoa Touch genannt (Apple, 2009b). Apple hat zum Erstellen von GUIs für das iPhone einen GUI-Builder, mit dem das Erstellen von GUIs durch einfaches Drag&Drop möglich ist. Damit kann man sehen, wie die GUI später auf dem iPhone aussieht, wodurch die gesamte Entwicklung sehr vereinfacht wird. Ein Nachteil von Cocoa Touch ist allerdings dessen geringe Verbreitung. Die GUI ist nur auf dem iPhone lauffähig und müsste bei einer weiteren Portierung erneut angepasst werden.

Eine bessere Lösung wären plattformübergreifende Bibliotheken wie z.B. wxWidgets[7] oder Qt[8], für die es inzwischen auch eine iPhone Portierung gibt.

Verwendet wird Cocoa Touch, da die GUI aufgrund der Größe und des Funktionsumfanges nur wenig Code enthält und bei einer Portierung leicht auszutauschen ist. Außerdem ist Cocoa Touch die einfachste und sicherste Möglichkeit für die Neuimplementierung der GUI, da sie von Apple speziell für das iPhone entworfen wurde.

Eine spätere Umstellung auf Qt wäre aber eine sinnvolle Möglichkeit dafür, die Software unabhängiger vom iPhone zu machen.

Da die Software nicht die komplette Funktionalität der PC-Version haben sollte, wurden nur folgende Elemente gebraucht:

1. Eine Videoausgabe vom Gesprächspartner
2. Eine Videoausgabe der eigenen Kamera
3. Ein Schalter für den ausgehenden Stream
4. Ein Schalter für den eingehenden Stream
5. Ein Call/Hangup Button
6. Ein Quit Button
7. Ein Pop-up für eingehende Anrufe

Da die Videoausgaben schon in den Abschnitten 4.3.7 und 4.4 beschrieben wurden, wird an dieser Stelle nicht weiter darauf eingegangen.

Schwierigkeiten bereitet allerdings die Kamera, weil das zu übertragende Format, wie in Abschnitt 4.4 beschrieben, in gleicher Größe auf dem Display dargestellt werden muss. Da

[7] www.wxwidgets.org
[8] www.qt-iphone.com

das Preview jedoch erst zur Laufzeit auf dem Display erscheint, muss vorher die genaue Position und Größe ausgerechnet und mit den restlichen Elementen abgestimmt werden.

4.6 Audioimplementierung

Apple stellt für den Zugriff auf die Soundhardware vier Bibliotheken zur Verfügung, die sich im Einsatzbereich und im Grad der Abstraktion unterscheiden.

CoreAudio ist das Framework auf unterster Ebene und greift über HAL[9] direkt auf die Hardware zu. CoreAudio ist ein sehr mächtiges Framework, das den Programmierer kaum einschränkt. Der Nachteil liegt in der Komplexität. Um z.B. eine einfache Mp3-Datei abzuspielen, ist es ungeeignet, da der Aufwand dazu in keinem Verhältnis steht. Möchte man komplexe Soundapplikationen entwickeln wie z.B. einen Synthesizer, ist das CoreAudio-Framework die einzige Möglichkeit dieses zu realisieren.

Zusätzlich kann das Framework durch das AudioUnit-Framework ergänzt werden. Mit diesem Framework können Plug-ins geschrieben werden, die dann von jedem Programm genutzt werden können. So können z.B. Filter und Effekte einheitlich entwickelt werden.

AudioToolbox befindet sich über CoreAudio und ist wesentlich einfacher zu benutzen. Es können ohne großen Aufwand Dateien und Ströme geladen und abgespielt werden. Der Programmierer muss allerdings selbst dafür sorgen, dass die Daten geladen und danach regelmäßig an die Soundkarte übergeben werden.

AVFoundation ist ein Framework, das noch weiter abstrahiert als AudioToolbox. Es können Dateien oder Daten aus einem Buffer abgespielt werden. Außerdem werden viele Funktionen unterstützt, die normalerweise zum Abspielen von Musik benötigt werden wie z.B. Spulen oder Wiederholungen eines Titels. Apple empfiehlt dieses Framework daher für einfache Aufgaben, während von Streaming oder positionsabhängigen Stereoeffekten, wie sie z.B. in Spielen auftreten, abgeraten wird.

OpenAL wird häufig in Spielen eingesetzt, da es direkt auf diesen Anwendungsbereich ausgelegt wurde. Es werden z.B. positionsabhängige Stereoeffekte unterstützt, die von den vorherigen Frameworks nicht unterstützt und daher vom Programmierer implementiert werden müssen.

[9] Hardware abstraction layer

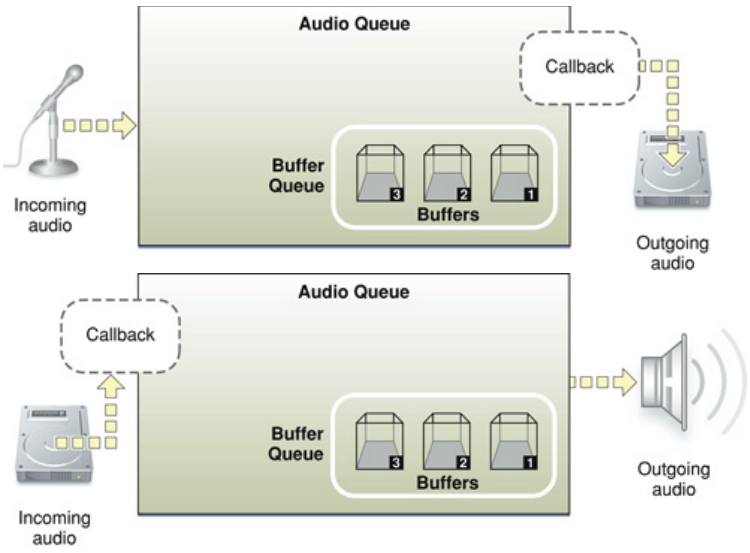

Abbildung 4.8: *Audio Queues (Quelle: developer.apple.com)*

4.6.1 Auswahl des Frameworks

Als Framework wird für dieses Projekt AudioToolbox ausgewählt. Der Grund dafür ist die einfache Implementierung und ausreichende Funktionalität. AudioUnit ist in diesem Zusammenhang zu wenig spezialisiert und daher unnötig kompliziert. Klangeffekte und Filter werden für die Videokonferenz zunächst nicht benötigt, weshalb dieses Framework nicht verwendet wird.

Das AVFoundation-Framework bietet keine vernünftige Möglichkeit einen Datenstrom abzuspielen, da es auf das einfache Abspielen von Dateien spezialisiert ist.

OpenAL wird nicht verwendet, da es auf Spiele und komplexere Multimediaanwendungen ausgelegt ist und daher zunächst konfiguriert werden müsste, weil es eigentlich nicht darauf ausgelegt ist, einen Datenstrom abzuspielen.

AudioToolbox bietet zwar nicht so viele Möglichkeiten wie AudioUnit und ist nicht so leicht zu benutzen wie AVFoundation, bildet aber einen guten Kompromiss aus beiden.

4.6.2 Funktionsweise der Auido Queues

Das AudioToolbox-Framework benutzt Audio Queues, um Audiodaten abzuspielen oder aufzunehmen. Sie stellen eine Verbindung zur Hardware her, erledigen die Speicherverwaltung und integrieren bei Bedarf Codecs. Hauptsächlich besteht eine Audio Queue aus einer

Callback-Funktion und einer Liste (Buffer Queue) aus mehreren Buffern, in die die Audiodaten eingefügt werden können.

Beim Abspielen von Audiodaten wird die Callback-Funktion vom Betriebssystem aufgerufen, sobald ein Buffer aus der Liste frei zum Beschreiben ist. In der Callback-Funktion kann dieser vom Programmierer mit den abzuspielenden Audiodaten befüllt und an das Betriebssystem zurückgegeben werden.

Beim Aufnehmen von Audiodaten werden die Buffer in der Liste vom Betriebssystem befüllt und danach über die Callback-Funktion an den Programmierer übergeben. Dieser kann die Daten aus den Buffern herauslesen und danach für ein erneutes Befüllen an das Betriebssystem zurückgeben.

Die Struktur einer AudioQueue und der Ablauf beim Aufnehmen und Wiedergeben ist in Abbildung 4.8 dargestellt.

4.6.3 Audioeingabe

Bevor man das Mikrofon benutzen kann, müssen einige Einstellungen gemacht werden, um die Daten im gewünschten Format zu erhalten. Als Format wird PCM[10] mit einer 16 kHz Samplerate verwendet. Auf Stereo wird verzichtet, um die Datenmenge möglichst gering zu halten. Die Initialisierung sieht wie folgt aus:

Quellcode 4.10: *Mikrofon initialisieren*

```
1
2 - (id) initWithURL: fileURL
3 {
4   self = [super init];
5   if (self != nil)
6   {
7     audioFormat.mSampleRate         = 16000.00;
8     audioFormat.mFormatID           = kAudioFormatLinearPCM;
9     audioFormat.mFormatFlags        = kAudioFormatFlagIsSignedInteger |
10                                     kAudioFormatFlagIsPacked;
11    audioFormat.mFramesPerPacket    = 1;
12    audioFormat.mChannelsPerFrame   = 1;
13    audioFormat.mBitsPerChannel     = 16;
14    audioFormat.mBytesPerPacket     = 2;
15    audioFormat.mBytesPerFrame      = 2;
16
17    AudioQueueNewInput (&audioFormat,recordingCallback,
18                        self, NULL,NULL,0,&queueObject);
19
```

[10] Puls-Code-Modulation

```
20      UInt32 sizeOfRecordingFormatASBDStruct = sizeof (audioFormat);
21      AudioQueueGetProperty (queueObject,kAudioQueueProperty_StreamDescription,
22                             &audioFormat,&sizeOfRecordingFormatASBDStruct);
23
24      [self enableLevelMetering];
25    }
26    return self;
27  }
```

In den Zeilen 7 bis 15 wird das Format eingestellt, in dem der Strom ankommen soll. Sobald neue Daten vom Mikrofon aufgenommen werden, kann mit einem Event darauf reagiert werden. Die Funktion, die dabei aufgerufen werden soll, wird in Zeile 17 festgelegt. In diesem Fall wird die Funktion recordingCallback aufgerufen, die für das Befüllen der Audiobuffer zuständig ist und wie folgt aussieht:

Quellcode 4.11: *Aufnehmen*

```
1  static void recordingCallback (void *inUserData,AudioQueueRef inAudioQueue,
2                                 AudioQueueBufferRef inBuffer,
3                                 const AudioTimeStamp *inStartTime,
4                                 UInt32 inNumPackets,
5                                 const AudioStreamPacketDescription *inPacketDesc)
6  {
7
8    AudioRecorder *recorder = (AudioRecorder *) inUserData;
9
10   if (inNumPackets > 0)
11   {
12     int offset=1280-buffersize;
13     memcpy(buffer+buffersize, inBuffer->mAudioData, offset);
14     GetCurrentSoundOut()->Encode( (short*)(buffer) );
15     buffersize=0;
16
17     char* data=(char*)inBuffer->mAudioData+offset;
18     int datasize=inBuffer->mAudioDataByteSize-offset;
19     int remaining=datasize;
20     for(int i=0;i<(datasize-1280);i+=1280){
21       GetCurrentSoundOut()->Encode( (short*)(data+i) );
22       remaining-=1280;
23     }
24
25     if(remaining>0){
26       memcpy(buffer,data+datasize-remaining, remaining);
27       buffersize=remaining;
28     }
29   }
30
31   if ([recorder isRunning])
```

```
32    {
33      AudioQueueEnqueueBuffer (inAudioQueue, inBuffer, 0, NULL);
34    }
35  }
```

In inBuffer->mAudioDataByteSize wird dem Programmierer mitgeteilt, wie viele Bytes vom Mikrofon aufgenommen wurden. Da der Codec zum Enkodieren 1280 Bytes erwartet, müssen überschüssige Bytes in einem Buffer gespeichert und beim nächsten Aufruf mit verschickt werden. Die Verwaltung des Buffers geschieht in den Zeilen 11 bis 14 und 24 bis 27. Auf den Audiobuffer, der die vom Mikrofon aufgenommenen Daten enthält, kann mit inBuffer->mAudioData zugegriffen werden, wobei die Anzahl der Bytes in inBuffer->mAudioDataByteSize steht.

Das eigentliche Enkodieren und Verschicken der 1280 Byte Pakete geschieht in Zeile 20. Die restlichen Bytes, die kein 1280 Byte Paket mehr füllen können, werden im Buffer gespeichert und beim nächsten Aufruf verschickt.

In Zeile 31 wird der Audiobuffer dem Betriebssystem für ein erneutes Befüllen zurückgegeben.

4.6.4 Audioausgabe

Der Ablauf der Audioausgabe funktioniert ähnlich wie die Audioeingaben. Das Betriebssystem ruft eine Rückruffunktion auf, in der der Programmierer einen übergebenen Buffer mit einem PCM-Signal beschreiben kann. Zuvor muss die Audioausgabe wie folgt konfiguriert werden:

Quellcode 4.12: *Audioausgabe initialisieren*

```
1  -(void)initAudio {
2    AudioSessionInitialize(NULL,NULL,NULL,NULL);
3    UInt32 sessionCategory = kAudioSessionCategory_PlayAndRecord;
4    AudioSessionSetProperty(kAudioSessionProperty_AudioCategory,
5                            sizeof (sessionCategory), &sessionCategory );
6    UInt32 UseSpeaker= kAudioSessionProperty_OverrideCategoryDefaultToSpeaker;
7    AudioSessionSetProperty(kAudioSessionProperty_OverrideCategoryDefaultToSpeaker,
8                            sizeof (UseSpeaker), &UseSpeaker );
9
10   in.mDataFormat.mSampleRate        = SAMPLERATE;
11   in.mDataFormat.mFormatID          = kAudioFormatLinearPCM;
12   in.mDataFormat.mFormatFlags       = kLinearPCMFormatFlagIsSignedInteger |
13                                       kAudioFormatFlagIsPacked;
14   in.mDataFormat.mBytesPerPacket    = 2;
15   in.mDataFormat.mFramesPerPacket   = 1;
16   in.mDataFormat.mBytesPerFrame     = 2;
```

```
17    in.mDataFormat.mChannelsPerFrame = 1;
18    in.mDataFormat.mBitsPerChannel   = 16;
19    in.frameCount                    = FRAMES;
20
21    AudioQueueNewOutput( &in.mDataFormat, AQBufferCallback, &in,
22                         NULL, NULL, 0, &in.queue);
23    UInt32 bufferBytes = FRAMES*2;
24    for (int i=0; i< BUFFERS; i++) {
25      AudioQueueAllocateBuffer(in.queue, bufferBytes, &in.mBuffers[i]);
26      AQBufferCallback (&in, in.queue, in.mBuffers[i]);
27    }
28
29    AudioQueueSetParameter(in.queue, kAudioQueueParam_Volume, 1.0);
30    AudioQueueStart(in.queue, NULL);
31  }
```

Bei einer Videokonferenz muss das Verhalten des iPhones zunächst auf die Anforderungen der Videokonferenz angepasst werden. Dies geschieht in den Zeilen 2 bis 8. Beim Zugriff auf das Mikrofon wird die Audiowiedergabe unterbrochen, da üblicherweise Sprache aufgenommen wird und ansonsten vom Lautsprecher übertönt werden würde. Bei der Videokonferenz ist dieses Verhalten allerdings notwendig und wird daher in den Zeilen 3 und 4 aktiviert. Außerdem wird festgelegt, welcher Lautsprecher zur Wiedergabe verwendet wird.

In den Zeilen 10 bis 19 wird das Format des Stroms angegeben. Nachdem in Zeile 21 eine neue Audioqueue angelegt und eine Rückruffunktion angegeben wurde, muss in Zeile 25 und 26 der nötige Speicher reserviert werden und die Rückruffunktion einmalig aufgerufen werden. Nach der Initialisierung wird der Aufruf vom Betriebssystem übernommen, sobald ein Buffer zum Befüllen zur Verfügung steht.

Die Callback-Funktion sieht wie folgt aus:

Quellcode 4.13: *Audioausgabe*

```
1  static void AQBufferCallback(void *in, AudioQueueRef inQ,
2                               AudioQueueBufferRef outQB)
3  {
4    char* data=(char*)outQB->mAudioData;
5    if(running){
6      for(int i=0;i<AMOUNT_OF_PCM;i++){
7        GetCurrentSoundOut()->GetFrame((short*)(data+i*1280));
8      }
9    }else{
10     memset(data,0,AMOUNT_OF_PCM*1280);
11   }
12   outQB->mAudioDataByteSize=AMOUNT_OF_PCM*1280;
13   AudioQueueEnqueueBuffer (inQ,outQB, 0 ,NULL);
14 }
```

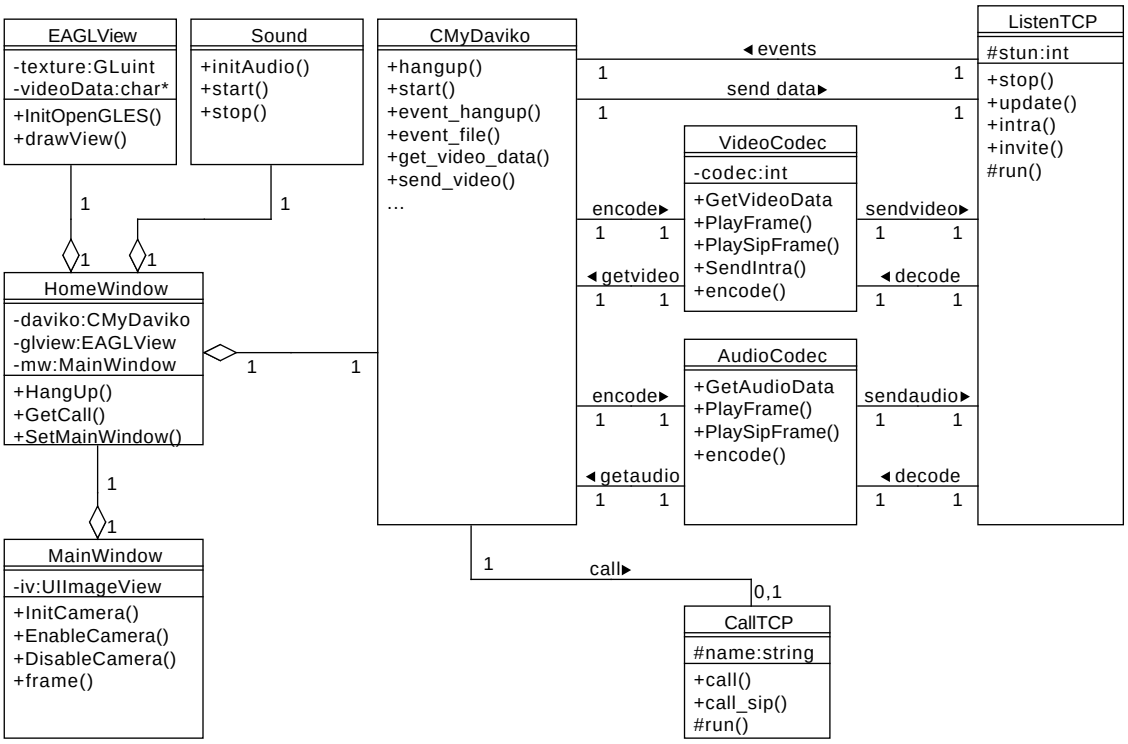

Abbildung 4.9: *Klassendiagramm der Videokonferenz*

Bei jedem Aufruf wird in Zeile 7 eine bestimmte Anzahl von 1280 Byte Paketen dekodiert und in den Buffer geschrieben. Die Anzahl kann mit AMOUNT_OF_PCM festgelegt werden und ist in diesem Fall auf 10 gesetzt. In Zeile 13 wird der gefüllte Buffer dem Betriebssystem übergeben, das die Audiodaten dann abspielt.

Die Audioausgabe funktioniert, allerdings gibt es noch einige Probleme mit der Qualität, weshalb sich die Sprache sehr unsauber anhört. Hier sind noch weitere Feinabstimmungen notwendig, um eine gute Sprachqualität zu erreichen.

4.7 Resultat

Das Klassendiagramm der fertigen Software ist in der Abbildung 4.9 dargestellt.

Der interne Aufbau der Software wurde grundlegend verändert und an die Besonderheiten des iPhones angepasst. Die größten Änderungen wurden an dem Codec vorgenommen. Im Gegensatz zur ursprünglichen Software läuft das En- und Dekodieren nicht mehr in einem

seperaten Thread. Außerdem wurden beide Aufgaben in einer Klasse VideoCodec zusammengefasst.

Um die Kamera ansprechen zu können, musste dieser Teil des Codes in die GUI verlagert werden, was große Auswirkungen auf das Design der Software hat. Die Videoausgabe wurde verändert, um das Design möglichst einheitlich zu gestalten. Weitere große Änderungen wurden an der GUI vorgenommen, die komplett neu gestaltet wurde.

5 Ergebnis, Test, Evaluation

Das Resultat dieser Arbeit ist eine Videokonferenzsoftware auf dem iPhone. Das Ergebnis der fertigen Software kann man in der Abbildung 5.1 sehen. Die Software wurde dabei an die besonderen Gegebenheiten auf dem iPhone angepasst, indem an einigen Stellen das Design der Software verändert wurde.

Im Vergleich zum vorherigen Design ist nicht mehr CMyDaviko für die Kamera oder die Darstellung auf dem Display zuständig. Um einen Videostream zu empfangen oder zu senden, bietet CMyDaviko einfach zu benutzende Funktionen an, über die CMyDaviko mit der Kamera und Videodarstellung kommuniziert. Dies ermöglicht ein leichtes Auswechseln der einzelnen Komponenten, ohne Änderungen an CMyDaviko vornehmen zu müssen.

Um die Software auf ihre Tauglichkeit zu prüfen, müssen Zeitmessungen durchgeführt werden. Zunächst wird dafür die maximale Enkodierleistung, die auf dem iPhone möglich ist, ermittelt. Dazu wird eine Videosequenz aus einer Datei geladen und enkodiert. Als Videosequenz wird eine Standardtest-Sequenz (Foreman) verwendet, mit der sich die gemessenen Leistungen von Codecs und Geräten mit anderen vergleichen lassen. Für das Enkodieren der 300 Frames benötigt das iPhone 21,45 Sekunden was einer Framerate von 14fps entspricht.

Interessanter ist allerdings, wie sich die Software im laufenden Betrieb verhält. Dafür wird eine Testumgebung aufgebaut, die das Verhalten der Software in unterschiedlichen Situationen misst. Für jede Zeitmessung werden 1000 Messwerte zugrunde gelegt, um den Durchschnitt und Median zu ermitteln. Der Median ist von Interesse, da er im Gegensatz zum Durchschnitt nicht so anfällig für Werte ist, die stark von den restlichen abweichen. Beim Durchschnitt und Median werden zusätzlich die Standardabweichungen angegeben. Außerdem werden die Minimal- und Maximalwerte angegeben um einen Überblick zu bekommen, wie stark das Zeitverhalten vom Durchschnitt abweichen kann. Speziell beim Maximum kann es zu erheblichen Abweichungen kommen, da die Komponenten nicht isoliert gemessen werden und daher von anderen Komponenten beeinflusst werden können. Es ist z.B. möglich, dass ein Thread von einem höher priorisierten Thread verdrängt wird und sich dies auf die Messergebnisse auswirkt.

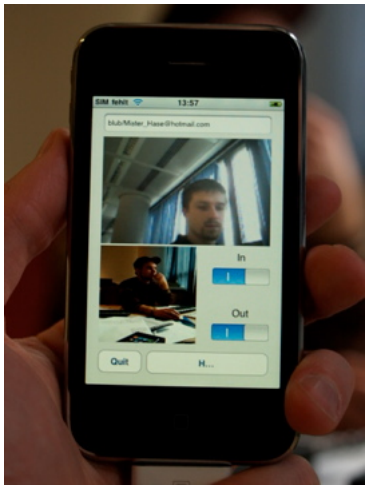

Abbildung 5.1: *Ergebnis der Videokonferenz*

Im Einzelnen wurden folgende zeitintensive Aufgaben untersucht:

- Dekodieren
- Videoausgabe mit OpenGL ES
- Verschicken eines Frames
- Enkodieren

5.1 Dekodierzeit

Die Zeit für das Dekodieren ist nicht konstant, sondern ändert sich entsprechend dem Aufbau des Videos und der Stärke der Kompression der einzelnen Daten. Weitere Faktoren sind mögliche andere Threads oder Prozesse, die auch auf dem Prozessor arbeiten. Daher sind die gemessenen Daten nur grobe, für Vergleichszwecke aber ausreichende Richtwerte.

Für diese Arbeit werden vier unterschiedliche Szenarien untersucht, die innerhalb einer Konferenz auftreten können. Zunächst wird ein Video untersucht, das nur aus schwarzen Bildern besteht. Der Enkoder hat hierfür nicht viel zu berechnen, weil es keine Bewegungen oder farbigen Elemente auf dem Bild gibt. Anschließend wird ein Standbild untersucht, auf dem zwar nahezu keine Bewegungen stattfinden, aber farblich unterschiedliche Areale vorhanden sind. Kleinere Bewegungen im Video können allerdings durch Bildrauschen auftreten.

Tabelle 5.1: *Dekodierzeiten*

Szenario	**arith. Mittel** *in ms*	**Median** *in ms*	**Maximum** *in ms*	**Minimum** *in ms*
Schwarzes Bild	$5,782 \pm 1,634$	$4,977 \pm 1,455$	81,216	3,771
Standbild	$7,165 \pm 2,269$	$6,070 \pm 2,010$	58,907	4,033
Normal	$11,834 \pm 4,004$	$10,527 \pm 3,847$	73,908	5,074
Stark bewegt	$30,580 \pm 7,300$	$30,235 \pm 7,294$	66,210	5,197

Danach werden zwei weitere Videos untersucht, auf denen Bewegungen stattfinden. Hier wird noch einmal zwischen normalen und starken Bewegungen unterschieden. Für die normalen Bewegungen wird ein sich leicht bewegender Kopf als typisches Motiv einer Videokonferenz gewählt. Für die starken Bewegungen wird das Kamerabild verwackelt und es werden schnelle Handbewegungen ausgeführt.

In der Tabelle 5.1 sind die Zeiten aufgelistet. Dabei ist zu entnehmen, dass das Dekodieren sehr schnell abläuft. Selbst bei einem Bild mit Bewegung wäre eine theoretische Framerate von 28,63 fps möglich. Auffällig ist die Abhängigkeit der Zeit von der Bewegung der Objekte. Je mehr Bewegungen stattfinden, desto länger dauert die Dekodierung des Bildes.

Betrachtet man die Maxima, dann fällt auf, dass sie deutlich größer sind als der Durchschnitt und dass das größte Maximum beim schwarzen Bild aufgetreten ist. Man erkennt am Durchschnitt, der deutlich näher am Minimum als am Maximum liegt, dass solche extremen Abweichungen nur sehr selten auftreten. Die Maxima entstehen beim Dekodieren eines Intra-Frames, weshalb ein vollständiges Bild dekodiert werden muss und daher die benötigte Dekodierzeit unabhängig von dem Inhalt ist.

Zusätzlich haben die unterschiedlichen Priorisierungen der Threads einen Einfluss auf die Dekodierzeit. Der Dekoder-Thread hat die niedrigste Priorität und kann von den anderen Threads verdrängt werden. Daher variieren die Maxima sehr stark bei jeder Messung.

Zusammenfassend lässt sich sagen, dass das Maximum zwar zeigt, wie stark die einzelnen Messwerte abweichen können, aber keine Aussage über die eigentliche Dekodierzeit macht.

5.2 Videoausgabe mit OpenGL ES

Die Darstellung auf dem Display ist unabhängig von dem darzustellenden Bild. Die Daten werden nach dem Dekodieren in einem Buffer gespeichert. Dieser wird regelmäßig von

OpenGL ES vollständig ausgelesen und auf dem Display dargestellt. Hierbei sind geringe Schwankungen möglich, wenn der Prozessor zugleich von anderen Threads oder Prozessen genutzt wird. Außerdem kann, wie in Abschnitt 4.3.7 beschrieben, die Größe der Textur, in die die Daten geschrieben werden sollen, eingestellt werden.

Zu diesem Vorgang können nur zwei Messungen gemacht werden, da die Texturgröße eine Potenz von 2 sein muss und sich dadurch nicht viel Spielraum bei der Größenwahl ergibt. Im Vergleich der Texturgrößen, die in der Tabelle 5.2 aufgelistet sind, wird deutlich, dass

Tabelle 5.2: *OpenGL ES*

Auflösung *in Pixel*	**arith. Mittel** *in ms*	**Median** *in ms*	**Maximum** *in ms*	**Minimum** *in ms*
128x128	$10,775 \pm 1,254$	$10,564 \pm 1,238$	$38,259$	$8,366$
256x256	$28,913 \pm 4,018$	$26,912 \pm 3,544$	$174,691$	$22,857$

sich die Zeit beim Benutzen einer 256x256 Textur mehr als verdoppelt. Allerdings ist der Qualitätsverlust bei der 128x128 Textur größer, da hier ein Bild, das eine Größe von 240x160 Pixel hat, auf 128x128 Pixel zusammengestaucht wird.

5.3 Erstellen und Verschicken eines Frames

In diesem Messversuch wird die Zeit gemessen, die für die Verarbeitung eines Frames benötigt wird. Dazu zählt das Auslesen der Kamera, das anschließende Enkodieren und abschließend das Versenden über das Netzwerk. An den Zeiten, die in Tabelle 5.3 zu sehen

Tabelle 5.3: *Sendezeiten*

Szenario	**arith. Mittel** *in ms*	**Median** *in ms*	**Maximum** *in ms*	**Minimum** *in ms*
Schwarzes Bild	$34,340 \pm 3,471$	$32,494 \pm 3,042$	$95,011$	$28,608$
Standbild	$69,404 \pm 9,889$	$67,812 \pm 9,795$	$113,784$	$54,839$
Normal	$81,125 \pm 8,885$	$79,478 \pm 8,683$	$132,661$	$63,652$
Stark bewegt	$78,950 \pm 9,329$	$73,856 \pm 8,953$	$139,702$	$62,688$

sind, ist überraschend, dass sie nicht von dem Bild abhängig zu sein scheinen. Bis auf das komplett schwarze Bild lieferten alle Messungen ähnliche Werte. Das würde aber den Sinn des Codecs verfehlen, der bei bewegungslosen Bildern eigentlich weniger zu leisten hat.

Da in dieser Messung auch Einflüsse vom Auslesen und Verschicken der Daten kommen können, wurde das Enkodieren in dem Abschnitt 5.4 gesondert betrachtet.

5.4 Enkodierzeit

Aber auch bei der Messung der Enkodierung scheinen sich die Zeiten nicht groß zu unterscheiden.

Tabelle 5.4: *Enkodierzeiten*

Szenario	**arith. Mittel** *in ms*	**Median** *in ms*	**Maximum** *in ms*	**Minimum** *in ms*	**QF**
Schwarzes Bild	$12,708 \pm 1,836$	$12,095 \pm 1,772$	44,773	9,636	20
Standbild	$29,878 \pm 2,063$	$28,999 \pm 1,775$	59,376	25,556	20
Normal	$47,974 \pm 1,789$	$47,571 \pm 1,735$	80,720	43,502	29
Stark bewegt	$45,995 \pm 3,483$	$45,099 \pm 3,448$	86,932	38,863	26

Die Ursache dafür ist indirekt der Quantisierungsfaktor. Der Quantisierungsfaktor wirkt sich auf die Kompressionsstärke des Codecs aus. Ein hoher Quantisierungsfaktor von max. 29 bedeutet eine geringere Qualität, aber auch kleinere Pakete, wodurch eventuell die verfügbare Bandbreite schlechter ausgelastet wird. Ein kleiner Quantisierungsfaktor bedeutet eine bessere Qualität, aber größere Pakete, wofür die Bandbreite unter Umständen nicht ausreichen könnte. Die Software kann den Quantisierungsfaktor zur Laufzeit regulieren und so auf die Bandbreite reagieren.

Im Codeausschnitt 5.1, der aus der ursprünglichen Software übernommen wurde, ist zu sehen wie der Quantisierungsfaktor, im Code eps_value genannt, an die Übertragungsgeschwindigkeit angepasst wird. In der Zeile 7 wird der neue Quantisierungsfaktor aufgrund des alten Wertes ausgerechnet. Damit die Qualität nicht beliebig verringert werden kann, wird in Zeile 8 der Maximalwert auf 29 gesetzt.

Quellcode 5.1: *Enkodieren der ursprünglichen Software*

```
1  int MAX_QPARAM=31;
2  int l=0;
3  if (eps_value<27) l=1;
4
5  Encode( codec, yuv, eps_value,
6          imode, 2, slicelen, l,
7          0, encode_code, &slices,
8          &len, &yuv_reco);
```

```
9
10   eps_value = EncoGetNewQparam( eps_value, len, imode );
11   eps_value = ( eps_value > (MAX_QPARAM-2) )? (MAX_QPARAM-2) : eps_value;
```

Bei einem stark bewegten Bild benötigt der Codec länger zum Enkodieren der Bilder, wodurch weniger Pakete entstehen, was zu mehr freier Bandbreite führt. Der Codec reagiert darauf, indem er den Quantisierungsfaktor senkt, um so die Qualität der Bilder zu erhöhen und die Bandbreite besser auszunutzen. Das führt allerdings wiederum zu noch längeren Enkodierzeiten. Der Codec reagiert also auf schwer zu enkodierende Bilder mit einer Qualitätssteigerung, was den Prozessor stärker belastet. Daher wird in der zweiten Zeile bei Werten die kleiner als 27 sind, der Codec entlastet, indem ein optionaler Filter im Codec ausgestellt wird.

Diesen Vorgang kann man auch in der Tabelle 5.4 beobachten. Bei dem Standbild und dem schwarzen Bild wird der Quantisierungsfaktor gesenkt, da nur wenig Daten übertragen werden und daher die Qualität erhöht werden kann. Bei einem normalen Video wird die Bandbreite bestmöglich ausgenutzt, weshalb der Quantisierungsfaktor auf das Maximum gesetzt wird, um eine hohe Kompression zu erhalten. Bei einem stark bewegten Video wird der Quantisierungsfaktor heruntergesetzt, weshalb der Filter im Codec ausgestellt wird und so ähnliche Enkodierzeiten wie bei einem normalen Video erreicht werden.

5.5 Abstimmen der Software aufgrund der Messergebnisse

Um die Videokonferenz möglichst flüssig in beide Richtungen zu realisieren, muss die Software abgestimmt werden. Dazu muss die Zeitvorgabe für jede einzelne Aufgabe so eingestellt werden, dass sie zur Bearbeitung ausreicht ohne die anderen Aufgaben zu blockieren.

Zunächst wird das Bearbeiten der ankommenden Frames betrachtet, da deren Anzahl und Qualität von der Gegenstelle abhängig sind. Als erstes wurde dazu die durchschnittliche Anzahl der ankommenden Frames gemessen. Die Messung ergab ca. 16 fps, wobei dieser Wert stark schwanken kann. Der Codec ist in der Lage, auf die Auslastung der Bandbreite zur reagieren, indem er die Framerate anpasst. Diese Messung wurde über einen Zeitraum von 60 Sekunden durchgeführt. Dabei wurde ein normal bewegtes Video als eingehender Datenstrom verwendet, während das iPhone für die ausgehenden Daten ein Standbild verwendet. Versendet man mit dem iPhone ein normal bis stark bewegtes Bild, wird die ankommende Framerate auf 11 herabreguliert. Für die weiteren Rechnungen wird zur Sicherheit von 16 fps ausgegangen.

Jedes Paket muss erst entpackt und dekodiert werden. Dieser Vorgang ist zwingend notwendig, da ein Verwerfen einzelner Pakete die GoP Struktur unterbrechen und so zu Fehlern beim Dekodieren führen würde.

Es ergeben sich folgenden Zeiten:

$$\begin{aligned} \text{Normal} &= 16 * 11,834ms = 189,344ms \\ \text{Bewegt} &= 16 * 30,580ms = 489,280ms \end{aligned}$$

Hinzu kommt die Zeit, die OpenGL ES für das Darstellen des Videos benötigt. Diese Zeit ist relativ konstant und kann nur anhand der verwendeten Texturgröße verändert werden. Da aber die Darstellung unabhängig von den ankommenden Paketen ist, kann die Framerate hier gesondert eingestellt werden.

Daraus ergibt sich die Tabelle 5.5.

Tabelle 5.5: *Dekodierzeiten in ms insgesamt*

Szenario	Texturgröße	fps									
		10	11	12	13	14	15	16	17	18	19
Normal	128x128	300	311	322	333	344	355	366	377	388	399
	256x256	480	509	538	567	596	625	654	683	712	741
Bewegt	128x128	600	611	622	633	644	655	666	677	688	699
	256x256	780	809	838	867	896	925	954	983	1012	1041

Es ist erkennbar, dass die 256x256 Textur vor allem bei hohen Frameraten sehr viel Zeit beansprucht weshalb die 128x128 Textur verwendet wird. Die Wahl der OpenGL ES Framerate fiel auf 15fps, da sie ein Video ohne Verzögerungen darstellt und die 655ms Rechenzeit bei bewegten Bildern für die Darstellung vertretbar ist. Um eine geeignete Einstellung auswählen zu können, müssen die Zeiten des Enkodierens und Dekodierens aufeinander abgestimmt werden.

Die maximale Framerate für die Kamera bei einem stark bewegten Bild berechnet sich daraus wie folgt:

$$\text{max. fps} = (1000ms - 655ms)/80ms = 4,31$$

Dabei wurde die Enkodierzeit aus der Tabelle 5.3 genommen, da hier die Sendezeiten mit berücksichtigt werden.

Bei allen Berechnungen wurde bisher vom ungünstigsten Fall ausgegangen, weshalb 4,31 fps das Minimum der möglichen Framerate ist. Die Kamera wird über einen Timer ausgelesen, der selbständig einzelne Frames verwirft, wenn die Berechnung zu lange dauert. Daher kann die Framerate noch erhöht werden, ohne dass die Gefahr besteht die Software zu überlasten. Um einen geeigneten Wert für die Framerate zu ermitteln, wird sie langsam erhöht und empirisch ausgewertet.

5.6 Überprüfung und Vergleich mit anderen Geräten

Als geeigneter Wert hat sich 12 fps herausgestellt. Zur Überprüfung der Einstellungen kann das Programm Instruments genutzt werden, mit dem sich allerlei Informationen über das Betriebssystem anzeigen lassen. In der Abbildung 5.2 ist die Systemlast bei unterschiedlichen Szenarien dargestellt.

Die vier Graphen haben folgende Bedeutung:

Systemlast gibt an, wie stark der Prozessor vom Betriebssystem beansprucht wird.

Benutzerlast gibt ähnlich wie bei der Systemlast die Last an, wie stark die Anwendung des Benutzers den Prozessor beansprucht.

Gesamtlast setzt sich aus der Systemlast und der Benutzerlast zusammen. Sie gibt an, wie stark der Prozessor insgesamt beansprucht wird.

Virtueller Speicher wird den Anwendungen vom Betriebssystem zur Verfügung gestellt. Da der benötigte Speicher schon beim Starten der Videokonferenz dem Betriebssystem mitgeteilt wird, finden nur kleine Veränderungen zur Laufzeit statt.

Die Systemlast ändert sich bei den unterschiedlichen Szenarien kaum und liegt konstant bei ca. 30%. Wie erwartet, verursacht das Enkodieren mehr Prozessorlast als das Dekodieren. Alles zusammen lastet den Prozessor nahezu vollständig aus, was auf eine gut eingestellte Konfiguration hinweist.

Verglichen mit dem Asus P735, das mit 520 MHz 10 fps enkodiert und 15 fps dekodiert, liefert das iPhone Ergebnisse, die den Erwartungen entsprechen (vgl. Cycon u. a., 2008, S. 1).

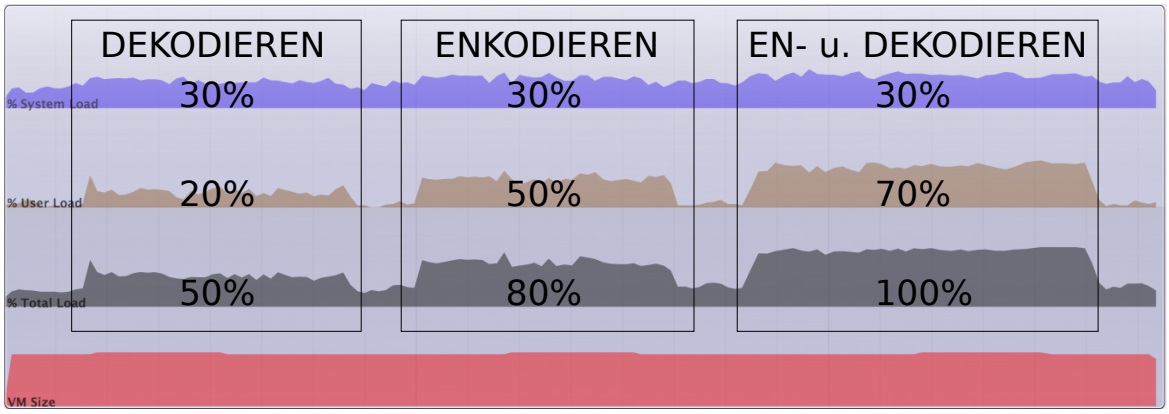

Abbildung 5.2: *Systemlast(blau), Benutzerlast(braun), Gesamtlast(schwarz), Speicher(rot)*

6 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, eine Videokonferenz für das iPhone zu portieren, was auch gelungen ist. Die Entwicklung war eine besondere Herausforderung, da hierzu C++ mit Objective-C gemischt werden und die Software stark auf Ausführungsgeschwindigkeit optimiert werden musste. Die Kombination von C++ und Objective-C verlief meistens problemlos, solange man einige Regeln beim Mischen der Sprachen beachtete. Die Software an die Gegebenheiten des iPhones anzupassen erwies sich als schwierig, da viele Bibliotheken, die in der ursprünglichen Software verwendet wurden, nicht vorhanden waren.

6.1 Entwicklungsumgebung

Die Programme, die Apple den Programmierern für die Entwicklung zur Verfügung stellt, sind gut aufeinander abgestimmt und lassen sich gut bedienen. Die Entwicklungsumgebung XCode von Apple ist klar strukturiert und übersichtlich aufgebaut. Ihre Benutzung ist sehr intuitiv und es gibt eine Menge Programme, die dem Anwender bei der Entwicklung helfen. Für das iPhone liefert Apple bereits viele Bibliotheken, die der Programmierer nutzen kann. Diese stellen die wichtigsten Funktionen zu Verfügung, um ein Programm möglichst schnell und einfach zu entwickeln.

Aber auch die Schwierigkeiten, die im Laufe der Arbeit auftraten, hatten ihren Ursprung oft bei den Programmen und Bibliotheken, die Apple den Entwicklern zur Verfügung stellt. Zunächst war es nicht einfach, Programme für das iPhone zu schreiben, da Apple das Entwickeln nur für registrierte Entwickler und für registrierte Geräte zulässt. Die Registrierung ist ein umständlicher Vorgang, der einmal erledigt werden muss, um dann beliebig viele Programme für ein registriertes Gerät schreiben zu können.

Beim Entwickeln eines Programmes wird es erst problematisch, wenn man versucht an Informationen oder Funktionen zu gelangen, die von Apple nicht offiziell zur Verfügung gestellt werden. Es gibt z.B. nur sehr wenige GUI Elemente, die zur Gestaltung einer Oberfläche genutzt werden können. Das sichert zwar ein einheitliches Look&Feel, schränkt den Programmierer allerdings sehr ein.

6.2 Portierung

Bei der Portierung mussten zunächst alle Windows-spezifischen Datentypen ersetzt werden. Danach konnte die Struktur der Software analysiert und darauf aufbauend ein Konzept für die Portierung entworfen werden. Weil viele Bibliotheken nicht für das iPhone existierten, musste für diese ein Ersatz gefunden und das Design an deren neue Eigenschaften angepasst werden.

Bei der Implementierung erforderte die Verknüpfung von Objective-C und C++ ein durchdachtes Konzept, da es bei diesen Übergängen für den Programmierer Einschränkungen gibt. Es ist z.B. nicht möglich, eine Obective-C Klasse von einer C++ Klasse erben zu lassen.

Das größte Problem dieser Arbeit war es die Kamera anzusprechen, wie in Abschnitt 4.4 beschrieben. Die fehlenden Dokumentationen und Headerfiles haben dazu geführt, dass die Daten der Kamera nur über Umwege ausgelesen werden konnten und erforderten viel Verständnis für den Aufbau der Cocoa Bibliothek. Abschließend lässt sich sagen, dass die Realisierung der Portierung gelungen ist, aber weiter verbessert werden kann.

6.3 Weiterentwicklungen

Aufbauend auf der Software könnte die Videokonferenz auf dem iPhone noch verbessert werden. Zur Zeit basiert die Software auf einer älteren Version von Daviko. Daher wäre es zunächst sinnvoll, den von mir verwendeten Code durch den neuesten von Daviko zu ersetzen. Eventuell ist es auch möglich, die Performance der Software noch weiter zu steigern.

Wahrscheinlich wird auch das iPhone irgendwann einmal eine Frontkamera besitzen. Spätestens dann dürfte eine API für die Kamera herausgebracht werden, mit der das Auslesen einfacher wird. Der Austausch der Kamera sollte aufgrund des neuen Designs kein Problem sein und bedarf nur geringer Änderungen der Software.

Eine weitere Entwicklung wäre auch bei der GUI denkbar. Es geht dabei hauptsächlich um das Design, da es nicht dem Look&Feel des iPhones entspricht. Zunächst sollte aber die weitere Entwicklung des iPhones abgewartet werden. Wenn die nächsten Generationen mit einer Frontkamera ausgestattet sind, ergeben sich neue Möglichkeiten bei der Gestaltung, da dann auf die umständliche Lösung beim Auslesen der Kameradaten verzichtet werden kann.

Literaturverzeichnis

[Apple 2009a] APPLE: *iPhone OS Reference Library*. 2009. – URL http://developer.apple.com/iphone/library/documentation/MediaPlayer/Reference/MediaPlayer_Framework/MediaPlayer_Framework.pdf. – Zugriffsdatum: 27.11.2009

[Apple 2009b] APPLE: *iPhone OS Reference Library*. 2009. – URL http://developer.apple.com/iphone/library/navigation/index.html#section=Frameworks&topic=Cocoa%20Touch%20Layer. – Zugriffsdatum: 27.11.2009

[Asus 2009] ASUS: *Asus 565 Specification*. 2009. – URL http://www.asus.de/product.aspx?P_ID=6UJWYSd5UByyAv7G&content=specifications. – Zugriffsdatum: 10.01.2010

[babylon 2010] BABYLON: *Portierung Definition*. 2010. – URL http://woerterbuch.babylon.com/portierung/. – Zugriffsdatum: 10.04.2010

[Cycon u. a. 2008] CYCON, Hans L. ; SCHMIDT, Thomas C. ; HEGE, Gabriel ; WÄHLISCH, Matthias ; MARPE, Detlev ; PALKOW, Mark: Peer-to-Peer Videoconferencing with H.264 Software Codec for Mobiles. In: JAIN, Ramesh (Hrsg.) ; KUMAR, Mohan (Hrsg.): *WoWMoM08 – The 9th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks – Workshop on Mobile Video Delivery (MoViD)*. Piscataway, NJ, USA : IEEE Press, June 2008, S. 1–6. – URL http://dx.doi.org/10.1109/WOWMOM.2008.4594916

[Ghanbari, M. 2003] GHANBARI, M.: *Standard Codecs: Image Compression to Advanced Video Coding (Telecommunications)*. Institution Electrical Engineers, 2003. – ISBN 0852967101

[Handley u. a. 2006] HANDLEY, M. ; JACOBSON, V. ; PERKINS, C.: *SDP: Session Description Protocol*. RFC 4566 (Proposed Standard). Juli 2006. – URL http://www.ietf.org/rfc/rfc4566.txt

[Rosenberg u. a. 2008] ROSENBERG, J. ; MAHY, R. ; MATTHEWS, P. ; WING, D.: *Session Traversal Utilities for NAT (STUN)*. RFC 5389 (Proposed Standard). Oktober 2008. – URL http://www.ietf.org/rfc/rfc5389.txt

[Rosenberg u. a. 2002] ROSENBERG, J. ; SCHULZRINNE, H. ; CAMARILLO, G. ; JOHNSTON, A. ; PETERSON, J. ; SPARKS, R. ; HANDLEY, M. ; SCHOOLER, E.: *SIP: Session Initiation Protocol*. RFC 3261 (Proposed Standard). Juni 2002 (Request for Comments). – URL http://www.ietf.org/rfc/rfc3261.txt. – Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630

[Samsung 2009] SAMSUNG: *Samsung i8000 Omnia II*. 2009. – URL http://www.samsung.com/at/consumer/detail/productPreviewRead.do?model_cd=BGT-I8000/M2&group=mobile-phone&type=mobile-phone&subtype=mobile-phone&subsubtype=#. – Zugriffsdatum: 10.01.2010

[Schulzrinne u. a. 2003] SCHULZRINNE, H. ; CASNER, S. ; FREDERICK, R. ; JACOBSON, V.: *RTP: A Transport Protocol for Real-Time Applications*. RFC 3550 (Standard). Juli 2003 (Request for Comments). – URL http://www.ietf.org/rfc/rfc3550.txt. – Updated by RFC 5506

[SDL 2009a] SDL: *About*. 2009. – URL http://www.libsdl.org/index.php. – Zugriffsdatum: 25.11.2009

[SDL 2009b] SDL: *Download*. 2009. – URL http://www.libsdl.org/download-1.2.php. – Zugriffsdatum: 19.11.2009

[SDL 2009c] SDL: *Google Summer of Code*. 2009. – URL http://www.libsdl.org/gsoc-2008.php. – Zugriffsdatum: 08.11.2009

[SDL 2009d] SDL: *SVN*. 2009. – URL http://www.libsdl.org/svn.php. – Zugriffsdatum: 25.11.2009

[Trick 2007] TRICK, Frank: *SIP, TCP/IP und Telekommunikationsnetze: Next Generation Networks und VoIP*. 3. Auflage. Oldenburg : Oldenbourg Wissenschaftsverlag, 2007. – ISBN 3-486-58228-3

[www.anandtech.com 2009] WWW.ANANDTECH.COM: *The iPhone 3GS Hardware Exposed & Analyzed*. 2009. – URL http://www.anandtech.com/gadgets/showdoc.aspx?i=3579&p=2. – Zugriffsdatum: 29.11.2009

Inhalt der beiliegenden CD

/**PDF**/ Die Bachelorarbeit im PDF Format.

/**Code**/**Videokonferenz**/ Der Quellcode zur Videokonferenz.

/**Code**/**Foreman**/ Der Quellcode mit dem die Enkodiergeschwindkeit der Foreman Videosequenz gemessen werden kann.

/**Messergebnisse**/ Die Messergebnisse im Rohformat.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §25(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 29. April 2010	
Ort, Datum	Unterschrift