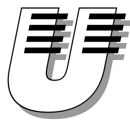

Fachbereich 4: Informatik

Erstellung eines Frameworks zur Filterung von Volumendaten auf der GPU am Beispiel von Videosequenzen

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Computervisualistik

vorgelegt von

Andreas Langs

Erstgutachter: Prof. Dr. Stefan Müller
Institut für Computervisualistik/Arbeitsgruppe Müller

Zweitgutachter: Dipl. Inform. Matthias Biedermann
Institut für Computervisualistik/Arbeitsgruppe Müller

Koblenz, im September 2006

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

....

(Ort, Datum) (Unterschrift)

Inhaltsverzeichnis

Einleitung **1**

I Theoretische Grundlagen **3**

1 Video **3**

1.1 Videonormen . . . 3

1.2 Digitale Videodaten . . . 5

1.2.1 Video-Containerformate . . . 5

1.2.2 Video-Codecs . . . 7

1.3 Probleme in Videobildern . . . 10

1.3.1 Rauschen . . . 10

1.3.2 Interlacing . . . 12

2 Filter **14**

2.1 Kantenerhaltende Glättungsfilter . . . 15

2.1.1 Bilateral Filter . . . 15

2.1.2 Anisotropische Diffusion . . . 17

2.1.3 Filter in der Wavelet Domäne . . . 17

2.2 Volumenfilter . . . 18

2.3 Auswahl des geeigneten Filters . . . 19

2.4 Fehlermaß . . . 19

3 GPGPU **21**

3.1 Programmiersprachen für die GPU . . . 22

3.2 OpenGL Shading Language . . . 23

3.2.1 Programme für die GPU . . . 24

3.2.2 Von der Graphikkarte zum universellen Coprozessor 25

3.3 Filtern auf der GPU . . . 26

4 Applikationen **27**

4.1 Videoapplikationen 27

4.1.1 VirtualDub 27

4.1.2 AviSynth 28

4.1.3 Adobe AfterEffects 28

4.1.4 Blender 30

4.2 Videofilter 31

4.2.1 Neat Video 31

II Praktische Umsetzung **32**

5 Das Framework **32**

5.1 Konzeption 32

5.2 Plexus 33

5.3 Erweiterungen an Plexus 34

5.3.1 Bilddatentypen 37

5.3.2 Video streaming 39

6 Arbeiten mit der GPU **41**

6.1 Verwenden der OpenGL Shading Language 41

6.2 Allgemeine Bildfilter auf der GPU 41

6.3 Volumenfilter auf der GPU 43

6.4 ATI vs. NVIDIA 43

7 Implementierungsdetails **46**

7.1 Entwickelte Devices 46

7.1.1 Devices für die CPU 46

7.1.2 Devices für die GPU 48

7.1.3 Im-/Export Devices 49

7.2 Portabilität 50

III Ergebnisse und Bewertung **51**

8 Messaufbau **51**

8.1 Testvideos 51

8.2 Hardware 51

8.3 Software 53

8.4 Messergebnisse 53

8.4.1 Qualitätsmessungen 53

8.4.2 Geschwindigkeitsmessungen 54

8.5 Bewertung 56

8.5.1 Qualität 56

8.5.2 Geschwindigkeit 58

8.5.3 Gesamtbewertung 58

Zusammenfassung und Ausblick **59**

Abbildungs- und Tabellenverzeichnis **61**

Literatur **62**

Anhang **64**

Einleitung

Ziel dieser Diplomarbeit ist die „Erstellung eines Frameworks zur Filterung von Volumendaten auf der GPU am Beispiel von Videosequenzen“. Die Filterung von Volumendaten ist ein bekannter Vorgang, aber wie passen Video und Volumen zusammen?

Bei vielen Videoaufnahmen, die mit handelsüblichen DV-Camcordern bei wenig Licht aufgenommen werden, also abends bzw. nachts oder in Räumen, findet man ein deutliches Rauschen in den Bildern. Dieses Rauschen ist typisch für alle Bilder, die mit einem beliebigen Signalwandler aufgenommen worden sind, seien es nun CMOS oder CCD Bildwandler in digitalen Fotoapparaten und Camcordern oder entsprechende Sensoren in Magnetresonanztomographie- oder Röntgengeräten. Die elektrischen Signale der Bildwandler sind schwächer, wenn weniger Licht bzw. Strahlung auf ein Bildelement trifft, denn die Signalstärke bildet die empfangene Lichtmenge ab. Wenn nun wenig Licht vorhanden ist, müssen die schwachen Signale verstärkt werden, um so mehr, je weniger Licht empfangen wird. Durch diese Verstärkung der Signale der Bildelemente entsteht das Rauschen — allerdings nicht nur bei digitalen Signalwandlern, sondern ist z. B. in der Fotografie auf Film als „Korn“ bekannt, das, je empfindlicher der Film ist, umso stärker wahrgenommen werden kann.

Da dieses Rauschen bei allen nicht synthetisch erstellten digitalen Bildern ein Problem ist, wird seit dem ersten aufgenommenen digitalen Bild geforscht, wie dieses Rauschen aus den Bildern herausgerechnet werden kann. Aufgrund der Arbeitsweise der Signalwandler wird es jedoch nie gelingen, das Rauschen restlos zu entfernen, ohne neue Artefakte[1] zu erzeugen. Ziel ist, dass diese neuen Artefakte weniger wahrgenommen werden als die entfernte Störung.

Das Rauschen im Bild ist für jedes aufgenommene Bild unterschiedlich, es ist also keine statische Überlagerung mit einem festen Muster. Daher kommt bei Videosequenzen, die ja im Prinzip aus schnell hintereinander abgespielten Einzelbildern bestehen, noch hinzu, dass z. B. in homogenen Flächen das Rauschen einen Eindruck erzeugt als würde die Fläche flimmern. Aus diesem Grund werden die Bilder moderner Kameras mit einem Entstörfilter verbessern. Dabei fügt man jedoch wie beschrieben neue Artefakte hinzu, die ihrerseits separat betrachtet vielleicht weniger wahrgenommen werden als das Rauschen, in der Bildfolge jedoch wiederum ein Flimmern erzeugen. Deshalb wird in der vorliegenden Arbeit ein Entstörfilter um eine zeitliche Komponente ergänzt, die man als dritte Dimension der zweidimensionalen, einzelnen Bilder sehen kann. Die Videosequenz wird

[1] Mit Artefakte werden in der Bildverarbeitung Muster,Störungen oder Verfälschungen bezeichnet, die durch die Bearbeitung des Bildes entstehen und im ungestörten Bild nicht vorhanden sind.

also als Volumen repräsentiert, wobei die Höhe und Breite den Dimensionen eines Bildes und jede Schicht in die Tiefe dem nächsten Bild in der Bildfolge entspricht. Nun wird auf diesem Volumen mit einem Volumenfilter eine Entstörung durchgeführt. Dadurch wird bei der Entstörung nicht nur die Nachbarschaft eines Pixels nach rechts und links, bzw. oben und unten, sondern auch in vorherigen und nachfolgenden Bildern berücksichtigt. Allerdings kann man entlang der Zeit und innerhalb jedes Bildes nicht einfach eine Weichzeichnung durchführen, um das Rauschen zu entfernen, da es sowohl in jedem einzelnen Bild als auch in der Zeit Kanten gibt, die man erhalten möchte.

Diese Diplomarbeit untersucht die Ergebnisse im Vergleich zu herkömmlicher Bild-für-Bild Filterung.

Die Filterung von Volumendaten ist nicht neu und wird vielfach bei natürlicherweise als Volumen vorliegenden Daten, wie Magnetresonanztomographie-Aufnahmen, schon lange angewendet. Deshalb soll das Framework nicht nur auf Videosequenzen beschränkt bleiben, sondern sich prinzipiell auch für die Filterung von solchen Daten eignen.

Weiterhin ist je nach verwendetem Filter die nötige Rechenleistung schon bei 2D-Daten sehr hoch und vervielfacht sich bei der 3D-Repräsentation noch einmal. Außerdem werden beständig neue Generationen von Camcordern mit immer höheren Auflösungen entwickelt, die durch kleinere Sensorelemente auf den Bildwandlern noch stärker als bisherige Geräte vom Rauschen betroffen sind. Der dadurch nötigen höheren Rechenleistung kann im Moment durch schnellere CPUs nicht Rechnung getragen werden. Es hat sich jedoch in letzter Zeit gezeigt, dass die immer mächtiger werdenden GPUs, die Prozessoren auf der Graphikkarte, sich nicht nur sehr gut für die Darstellung von Graphik, sondern auch für die datenparallele Verarbeitung digitaler Signale eignen. Es soll daher in der vorliegenden Arbeit ebenfalls untersucht werden, ob die Verwendung moderner Graphikprozessoren die Filterdauer im Vergleich zur CPU verkürzen kann.

Im weiteren Verlauf dieser Arbeit werden zunächst Einführungen in die verschiedenen Themengebiete gegeben und theoretisch die Probleme und angestrebten Lösungen diskutiert. Die Implementationen der ermittelten Lösungen werden dann im zweiten Teil dargestellt und getestet und die erreichten Ergebnisse präsentiert.

Teil I
Theoretische Grundlagen

1 Video

Der Themenbereich „Video“ ist sehr umfangreich mit vielen ganz unterschiedlichen Ausprägungen. Dazu gehört z. B. die analoge Videoübertragung, angefangen beim amplitudenmodulierten Schwarz-Weiß-Fernsehen bis hin zu modernem HDCP verschlüsseltem und MPEG-4 komprimiertem digitalen HDTV. Es wird daher hier nur auf die für diese Arbeit notwendigen Grundlagen eingegangen und Eigenschaften besprochen, die im Consumerbereich[2] für die Verarbeitung relevant sind. Bereiche wie die analoge Übertragung, Standards in anderen Ländern und Technik aus Fernsehstudios, werden außen vor gelassen.

Diese Arbeit behandelt die Verbesserung von digitalen Videosequenzen, die mit aktuellen handelsüblichen Consumer-DV-Camcordern erstellt wurden. Außerdem sollen die kommenden HDTV-Formate berücksichtigt werden, die zur Zeit im Consumerbereich für hochauflösendes Fernsehen stark im Kommen sind und damit auch bald Einzug in Consumer-Camcordern halten werden.

1.1 Videonormen

In der riesigen Anzahl von Videonormen, die im Laufe immer neuer technischer Errungenschaften entwickelt wurden, sind eine Vielzahl von Eigenschaften rund um Themenbereich „Video“ spezifiziert. Die folgenden beiden Ausschnitte gehen hauptsächlich auf die Repräsentation eines Bildes, eines Frames, ein. Dazu gehören der Farbraum und die räumliche Auflösung, aber auch Eigenschaften des Video-Daten-Stroms.

SDTV $Y'C_BC_R$ 4:2:0 PAL 576i25 Nonsquare Komprimiert

Diese kryptische Beschreibung gibt in etwa an, was an digitalen Daten im PC von einem per FireWire angeschlossenen Consumer-DV-Camcorder ankommt. Die Helligkeits- und Farbwerte der Bilder sind dabei nicht wie im Computergraphikbereich üblich als RGB-Farbwerte codiert, sondern als $Y'C_BC_R$. Dabei steht Y' für Luma mit ${}^{601}Y' = 0.299 \cdot R' + 0.587 \cdot G' + 0.144 \cdot B'$, was in etwa der Helligkeit entspricht und Chroma C_B und C_R für die Farbinformationen als Differenzen zu Blau und Rot mit $C_B = B' - Y'$und $C_R = R' - Y'$. R', G', B'stehen dabei für einen definierten RGB Farbraum.

[2]Mit Geräten aus dem Consumerbereich sind in dieser Arbeit Geräte gemeint, die im gut sortierten Elektronikfachmarkt erworben werden können.

Zusätzlich wird noch 4:2:0 Chroma-Subsampling angewendet. Das bedeutet, dass weniger Farbinformationen als Helligkeitsinformationen gespeichert werden, da das menschliche Auge empfindlich für Helligkeitsunterschiede aber wenig empfindlich für Farbunterschiede ist. So lässt sich die Datenmenge verringern bei nur einer kleinen Qualitätseinbuße. 4:2:0 bedeutet hierbei, dass auf 2×2 Pixeln mit Y' Luma-Information nur eine C_B und C_R Chroma-Information kommt. Außerdem wird durch diese Angabe noch die Art des Samplings innerhalb der 2×2 Pixeln der Chroma-Informationen angegeben.

Diese Eigenschaft und weitere werden als *PAL*[3] Standard bezeichnet. Dazu gehört auch die Auflösung von 720×576 Pixeln bei 50 Halbbildern pro Sekunde. Ein Halbbild, auch Field genannt, besteht vereinfacht gesagt abwechselnd aus nur den geraden oder ungeraden Zeilen des gesamten Bildes. Dieses abwechselnde Übertragen nennt man *interlaced*. Weiterhin ist ein Pixel nicht quadratisch. Dies ist historisch bedingt durch die Art des Samplings des analogen Signals.

Die beschriebenen Eigenschaften sind Eigenschaften der vorhandenen Bildinformationen. Man spricht von diesen Daten als „unkomprimiert“, obwohl durch das Chroma-Subsampling bereits Details entfernt werden. Weitere Details gehen nun durch die eigentliche Kompression verloren. Die Kompression ist sehr ähnlich zur JPEG Kompression, erfüllt aber diesen Standard wegen bestimmter, für die Videoverarbeitung nötige Eigenschaften nicht (wie z. B. eine konstante Bitrate und Verarbeitung von interlaced Frames). Nach der Kompression liegt für Consumer-DV Video eine Datenrate von 25 MBit/sec vor.

HDTV $Y'C_BC_R$ 4:2:2 PAL 1080p24 Square Komprimiert

Der digitale HDTV Standard verwirft viele, aber nicht alle Altlasten der analogen Videobildübertragung zu Gunsten einer Vereinfachung des Standards. Leider hatten viele Firmen bei der Standardisierung mitzureden, und so kann man eigentlich nicht sagen, dass der Standard wesentlich einfacher geworden ist. Einige Eigenschaften, die im Rahmen dieser Arbeit interessant sind, wie z. B. das Pixelseitenverhältnis oder die *interlaced* Übertragung, sind jedoch vereinfacht worden.

HDTV verwendet weiterhin eine $Y'C_BC_R$ Farbcodierung. Diesmal jedoch mit $^{709}Y' = 0.2126 \cdot R' + 0.7152 \cdot G' + 0.0722 \cdot B'$. Die Chroma-Kanäle sind wie bei SDTV definiert. Auch wird wieder ein Chroma-Subsampling, diesmal jedoch mit 4:2:2, verwendet. Das bedeutet, dass auf 2×2 Pixeln mit Y'Luma-Informationen jeweils zwei C_B und C_R Chroma-Informationen kommen, die Farbauflösung also höher ist als bei SDTV. Die Eigenschaften werden wiederum als *PAL*-Standard zusammengefasst. Dabei ist u. a. eine

[3] **P**hase **Al**ternating **L**ine

Auflösung von 1920×1080 Pixeln bei 24 kompletten Bildern festgelegt. Die Übertragung von ganzen Bildern wird als *progressive* bezeichnet.

Die Pixel bei HDTV sind quadratisch und der Datenstrom wird wieder komprimiert. Diesmal mit MPEG-2 mit einem definierten Satz an Parametern. Da durch die Wahl der Kompression und dessen Parameter die Datenrate erheblich beeinflusst werden kann, gibt es für verschiedene Einsatzbereiche nur Empfehlungen. Diese reichen von 27 MBit/s bis zu 270 MBit/s.

Für umfassende Informationen zu den Themen Farbe, Sampling, Codierung usw. im Bereich Video und HDTV sei [Poy03] empfohlen.

1.2 Digitale Videodaten

Nach der kurzen Übersicht über zwei wichtige Videonormen geht es um die Frage, wie die Videobilder und natürlich auch der Ton zum Video in den Dateien auf dem Computer abgelegt werden. Dabei unterscheidet man zwei Komponenten:

1. Container
2. Codec

In einem Container können mit einem Codec komprimierte Video- und Audio-Datenströme abgelegt werden. Der Codec, eine Wortkombination aus ***Co****der* und ***De****coder*, ist das Verfahren, mit dem die Videobilder und die Tonsamples komprimiert sind; der Container bestimmt, wie die komprimierten Video- und Audiodaten und weitere Daten in der Datei strukturiert sind.

1.2.1 Video-Containerformate

Mit Containern wird in der Informatik allgemein eine Struktur bezeichnet, die angibt, wie Inhalte abgelegt sind. Insbesondere sind alle Dateiformate in der Regel eine Art von Container, denn für jedes Dateiformat ist auf eine bestimmte Art festgelegt, wie die Daten abgelegt sind.

In modernen Video-Containern können mehrere Video- und Audio-Datenströme abgelegt sein, verschiedene Untertitel-Datenströme, Kapitelmarken, Menüstrukturen, Timecodes und Synchronisierungsinformationen. Das Zusammenfügen bzw. Trennen der ineinander geschachtelten Datenströme wird Multiplexing bzw. Demultiplexing genannt und ist eine wesentliche Eigenschaft, die die verschiedenen Containerformate unterscheidet. Programme zum Lesen und Demultiplexen der Containerformate werden in der Regel als *Source-Filter* und *Splitter* bezeichnet. Im Gegensatz zu den *Codecs* sind diese Programme eher unbekannt. Im Folgenden wird sich auf eine Auswahl von Video-Containerformaten beschränkt, und nicht auf die riesige Anzahl an Daten-, Audio- oder Multimedia-Containerformaten eingegangen.

AVI Das von Microsoft eingeführte Containerformat *Audio Video Interleaved* ist schon recht alt und daher sehr verbreitet. Um Dateien größer als 2 GB zu unterstützen, wurde von der Firma Matrox 1996 die Erweiterung *OpenDML* eingeführt, die auch als AVI 2.0 bekannt ist. Es unterstützt mehrere Video-, Audio-, MIDI- und Textdatenströme. Ein Nachteil ist, dass es schwierig zu erweitern ist, Fähigkeiten moderner Codecs nicht oder nur über Umwege unterstützt und sich nicht für das Streaming eignet.

Matroska Der offene Containerstandard Matroska, dessen Namen sich von den ineinander schachtelbaren russischen Puppen *Matroschka* ableitet, hat sich zum Ziel gesetzt, alle anderen Videocontainer abzulösen. Dazu setzt er auf die EBML (Extensible Binary Meta Language) auf, eine Binärversion von XML und steht unter der GNU Lesser General Public License. Das Format unterstützt mehrere Video- und Audio-Ströme, Kapitel, flexible Untertitel, Menüs, ist fehlertolerant, gut streambar und unterstützt die Fähigkeiten moderner Codecs wie variable Bitraten. Die Verbreitung dieses Formats ist noch gering.

MPEG-2 Part 1 Der von der ***M****otion* ***P****icture* ***E****xpert* ***G****roup* entworfene Standard MPEG-2 beschreibt neben zwei Containerformaten (Part 1) auch Verfahren zur Kompression von Video- und Audio-Daten (Part 2), weswegen die Trennung von Container und Codec bei MPEG Formaten gerne übersehen wird. Das Containerformat mit dem Namen *Programmstrom* wird auf jeder DVD eingesetzt und ist für alle Übertragungen mit geringen Fehlerraten entworfen worden. Es unterstützt mehrere Video- und Audio-Ströme und die von der DVD bekannten Menüs und Kapitelmarken. Das *Transportstrom* genannte Containerformat verwendet ein anderes Multiplexing und ist für Übertragungen mit hoher Fehlerrate entwickelt worden, wie z. B. die Übertragung von digitalem Fernsehen via DVB-S (Satellit) oder DVB-T (Terrestrisch).

Ogg Ebenso wie Matroska ist Ogg ein offener Containerstandard unter der Leitung der Xiph.Org Foundation, der mit dem Ziel entwickelt wurde, offen und frei von Softwarepatenten zu sein und Multimediainhalte effizient speichern und streamen zu können. Es stellt den Standardcontainer für die von der Xiph.Org Foundation entwickelten Video- und Audio-Codecs Theora und Vorbis dar, ist aber natürlich nicht auf diese beschränkt. Fehlererkennung, Timestamps und ein geringer Datenoverhead zeichnen dieses Format aus. Die Verbreitung von Ogg ist gering, jedoch setzt z. B. die Wikipedia bei Audioinhalten auf dieses Format.

Quicktime Unter dem Namen *Quicktime* wird eine von der Firma *Apple* entwickelte Multimedia-Architektur verstanden. Das Containerformat

Quicktime, auf dem auch das Containerformat von MPEG-4 basiert, ist ein für verschiedenste multimediale Daten entwickeltes Format und daher sehr flexibel einsetzbar. Da es dieses Format ebenso wie AVI schon recht lange gibt, jedoch schon früh für Streaming eingesetzt werden konnte, hat es sich in diesem Bereich sehr stark im Internet durchgesetzt. Außerdem ist es das Standard-Video-Container- und Codec-Format der Betriebssysteme der Firma Apple.

1.2.2 Video-Codecs

Mit *Codec* werden Programme bezeichnet, die Video- oder Audio-Daten in ein bestimmtes Kompressionsformat komprimieren bzw. aus diesem dekomprimieren können. Das Kompressionsformat gibt dabei vor, wie die komprimierten Daten aussehen müssen, es gibt jedoch nicht vor, wie die unkomprimierten Daten in die komprimierte Form umgewandelt werden. Der Algorithmus zum Komprimieren unterscheidet die verschiedenen Codecs, sie produzieren aber alle dem Kompressionsformat entsprechend komprimierte Datenströme. Deshalb können Daten, die mit einem bestimmten Codec komprimiert wurden, von einem anderen Codec dekomprimiert werden, wenn beide das gleiche Kompressionsformat unterstützen.

In der Regel definieren die Kompressionsformate verlustbehaftete Kompressionsmethoden, da die Datenrate von unkomprimiertem Video sehr hoch ist. Durch die Kompression kann Speicherplatz und Übertragungsbandbreite gespart werden, indem nicht wahrnehmbare Details weggelassen werden. Ausgehend von Verfahren, die dem der JPEG Kompression von Bildern ähnlich sind, wurden im Laufe der Entwicklung immer stärker komprimierende Verfahren entwickelt, deren Komplexität aber ebenfalls deutlich zunahmen. So kann bei einigen Kompressionsformaten durch die Spezifizierung von verschiedenen *Profilen* die Komplexität beeinflusst werden. Dies ist bei Anwendungen hilfreich, bei denen nur eine geringe Rechenleistung zum Dekodieren zur Verfügung steht, wie dies z. B. bei PDAs oder Handys der Fall ist.

Umgangssprachlich ist die Trennung zwischen *Kompressionsformat* und *Codec* nicht immer eindeutig. So produzieren z. B. der *XviD* und der *DivX* Video-Codec jeweils dem Kompressionsformat MPEG-4 Part 2 entsprechende Datenströme.

Es folgt eine kurzer Überblick über einige wichtige und verbreitete Kompressionsformate und es werden Codecs angegeben, die dieses Format unterstützen. Der unter Windows verbreitete Source-Filter/Splitter/Codec *ffdshow* setzt auf der *ffmpeg* Bibliothek, einer plattform-unabhängigen Bibliothek zum Lesen und Schreiben aller relevanten Containerformate und Komprimieren und Dekomprimieren aller relevanter Kompressionsformate, auf. Er ist somit in der Lage, alle aufgeführten Container- und Kompressionsformate zu lesen und zu schreiben.

DV Das DV Format beschreibt neben Bandkassettenmaßen, Bandschreibverfahren und vielen weiteren Eigenschaften rund um die Aufzeichnung und Übertragung von digitalem Video auch ein Kompressionsformat. Bei diesem Format wird jedes Bild separat und nicht in Abhängigkeit von vorhergehenden und nachfolgenden Bildern komprimiert . Die typische Datenrate, die je nach Bildinhalt leicht variieren kann, liegt bei ca. 25 MBit/sec. Im Verhältnis zu der Datenrate von unkomprimiertem YUY2 (YUV bzw. $Y'C_BC_R$ mit 4:2:2 Sampling) bei 25 Frames/sec von 167,5 MBit/sec ergibt sich ein Kompressionsverhältnis von ca. 6:1. Wegen der hohen zu erwartenden Fehlerrate bei Band-basierten Aufzeichnungsverfahren verwendet das DV Kompressionsformat ein Reed-Solomon Fehlererkennungs- und -korrekturverfahren.

Bekannte Codecs sind *Cedocida*[4] und freie Codecs von *Canopus* und *Panasonic*.

HuffYUV Das vom gleichen Autor wie AviSynth (siehe 4.1.2) entwickelte Kompressionsformat ist im Gegensatz zu allen anderen hier vorgestellten Formaten verlustfrei. Der zu diesem Format gehörende Codec arbeitet sehr schnell, so dass eine Kompression in Echtzeit möglich ist. Das Format wurde zur Ablösung von unkomprimiertem YUV-Video entwickelt. Der Original Codec *HuffYUV*[5] kann nur YUY2, UYVY (wie YUY2 mit gedrehten Bytes) und RGB Eingabedaten komprimieren, wobei die Geschwindigkeit bei RGB geringer als bei YUY2 und UYUV ist. Es besteht zudem die Möglichkeit, RGB Daten nach YUY2 zu konvertieren und dann zu komprimieren. Bei der Konvertierung tritt jedoch durch das geringere Sampling der Farbkanäle ein Verlust auf. Das Format komprimiert ebenso wie das DV Format jedes Bild separat.

Viele Videobearbeitungspipelines arbeiten im YUY2 Format, nicht zuletzt weil auch die MPEG Formate diesen Farbraum verwenden. Dadurch eignet sich dieser Codec sehr gut, um Zwischenergebnisse verlustfrei speichern zu können.

MPEG-2 Part 2 / H.262 Der MPEG-2 Standard wurde 1994 eingeführt. Er fand große Verbreitung, da er auf DVDs und beim DVB (Digital Video Broadcasting) eingesetzt wird. Dem Format liegt die *Group-of-Pictures* (GOP) genannte Struktur zu Grunde (Abb. 1). Sie ist aus I-, P-, B- und D-Frames zusammengesetzt. Dieses Verfahren wird Intra-Frame-Kodierung genannt:

I-Frames sind komplette, für sich allein stehende Bilder.

P-Frames sind Differenzinformationen zu vorhergehenden I- oder P-Frames.

[4] `http://www-user.rhrk.uni-kl.de/dittrich/cedocida/`
[5] `http://neuron2.net/www.math.berkeley.edu/benrg/`

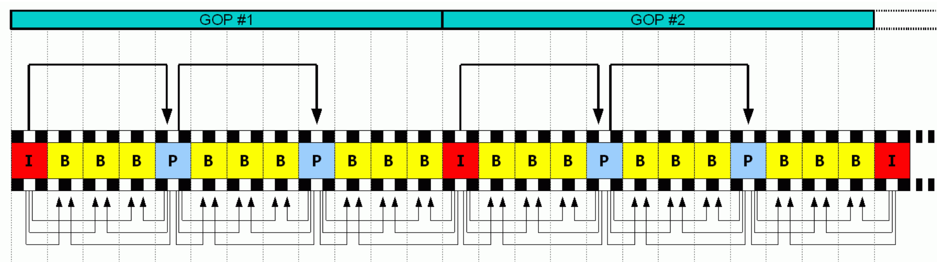

Abbildung 1: Group-of-Pictures [Wika]

B-Frames sind Differenzinformationen zu vorhergehenden und/oder nachfolgenden I- und P-Frames.

D-Frames dienen zum schnellen Vorlauf.

Die gewünschte Datenrate kann anhand verschiedener Parameter eingestellt werden, es werden allerdings in der Regel einige Grenzen vom übertragenden Medium vorgegeben, z. B. maximal 10,08 MBit/sec inkl. Audio bei Video-DVDs.

Bekannte Codecs sind *TMPGEnc* und *bbmpeg*.

MPEG-4 Part 2 (ASP) / H.263 Der MPEG-4 Standard wurde 1998 als Nachfolger von MPEG-2 eingeführt und benutzt ebenso wie dieses als grundlegende Struktur die GOP. Der Part 2 (ASP - ***A**dvanced **S**imple **P**rofile*) beschreibt das von den verbreiteten Codecs *DivX* und *XviD* verwendete Kompressionsformat. Da während der Entwicklung dieser Codecs höhere Kompressionsraten mit bestimmten Elementen erreicht wurden, die nicht dem Standard entsprechen, gibt es von diesen Codecs Erweiterungen und Profile, die dann jedoch Codec- und nicht Kompressionsformat spezifisch sind.

Aufgrund der hohen Verbreitung dieses Kompressionsformats erschienen viele Standalone-DVD-Player mit der Möglichkeit, auch solche Videodateien wiederzugeben. Durch die verschiedenen Erweiterungen der Codecs ist jedoch nicht immer sichergestellt, dass alle Videodateien auf allen Playern einwandfrei abgespielt werden können.

MPEG-4 Part 10 (AVC) / H.264 Der Wunsch nach einem Kompressionsformat, das die neu entwickelten Verfahren enthält, mündete 2003 in dem Kompressionsformat *MPEG-4 Part 10 **A**dvanced **V**ideo **C**oding* und ist auch unter der ITU-Bezeichnung H.264 (***I**nternational **T**elecommunication **U**nion*) bekannt.

Mit der zunehmenden Verfügbarkeit von Codecs für dieses Format steigt auch die Verbreitung. Dies liegt auch an der ca. doppelt so hohen Codiereffizienz gegenüber MPEG-4 Part 2 und ca. dreifachen Effizienz gegenüber

Abbildung 2: Probleme in Videobildern: a) Interlacing (vergrößert), b) Dunkelstrom-Rauschen (verstärkt)

MPEG-2 Part 2 und der breiten Verfügbarkeit von Prozessoren, die schnell genug sind, dieses Format zu dekodieren. Bekannte Codecs sind der kommerzielle *Nero Digital* und der offene *x264*.

Dieses Format ist eines der obligatorischen Video-Kompressionsformate der kommenden HD-DVD und Blu-ray Disk, außerdem wird es in den Mobilfernsehstandards DVB-H und DMB verwendet. Es ist also zu erwarten, dass dieses Format eine ebenso große Verbreitung wie MPEG-2 Part 2 und MPEG-4 Part 2 erreichen wird.

1.3 Probleme in Videobildern

Videobilder, die mit einer Kamera aufgenommen werden, sind mit diversen Fehlern behaftet, die durch die Natur der Analog-Digital-Wandlung entstehen. Dazu gehören u. a. alle Arten von Rauschen (siehe 1.3.1). Diese Fehler sind in computergenerierten Bildern, z. B. in Animationsfilmen, gewöhnlich nicht vorhanden. Ein weiteres Problem in der digitalen Videoverarbeitung kann das *Interlacing* darstellen, das von der Wahl der Videonorm abhängig ist.

1.3.1 Rauschen

Das Problem des Rauschens tritt hauptsächlich bei der Videoaufnahme von dunklen Szenen auf. Die Bildsensoren, die in allen Arten von digitalen Einzelbild- und Videokameras eingesetzt werden, funktionieren nach dem

gleichen Prinzip. Der Bildsensor verfügt über viele sehr kleine, lichtempfindliche Zellen mit einer Kantenlänge ab aktuell 5 μm, die den resultierenden Pixeln entsprechen. Während der Belichtung wird die Energie des eintreffenden Lichts in die Zellen übertragen. Nach der Belichtung wird diese Energie ausgelesen und entfernt (Reset). Da diese Ladungen sehr klein sind, werden sie verstärkt und schließlich von einem Analog-Digital-Wandler in diskrete Werte zerlegt.

Bei diesem Vorgang gibt es verschiedene Stellen, an denen Störungen auftreten, die schließlich zu dem führen, was der Betrachter als Rauschen wahrnimmt. Zum einen entstehen in den lichtempfindlichen Zellen Ladungen, die nicht durch das eintreffende Licht verursacht werden. Dies ist u. a. durch die starke Miniaturisierung bedingt und für alle Arten von Halbleitern völlig normal. Diese Ladungen werden als Dunkelstrom bezeichnet, da sie ein Signal ohne Licht, also im Dunklen, verursachen. Dieser Dunkelstrom verursacht nur eine sehr geringe Ladung in den Zellen, deshalb tritt er hauptsächlich in dunklen Bildbereichen zu Tage. Die Ladungen in einer mit viel Licht belichteten Zelle ist wesentlich höher als der Dunkelstrom, weshalb dieser nicht erkennbar ist.

Dieser Effekt tritt besonders bei dunklen Szenen auf, weil in diesen insgesamt wenig Licht vorhanden ist und die geringe Menge an Ladung insgesamt mehr verstärkt werden muss, um ein ausreichend starkes Signal zur Weiterverarbeitung zu erhalten. In der digitalen Fotografie wird diese Verstärkung durch den einstellbaren ISO-Wert beeinflusst, bei digitalen Videokameras kann diese Verstärkung in der Regel durch den *Gain*-Wert beeinflusst werden. Dabei wird, so lange es geht, nur minimal verstärkt (0dB[6]) und zu viel Licht durch Schließen der Blende reduziert. Diese minimale Verstärkung ist auch bei der vergleichsweise starken Ladung der Zellen bei viel Licht für die Weiterverarbeitung notwendig. Dabei ist diese Verstärkung so gewählt, dass bis zu einer gewissen unteren Lichtmengengrenze ein rein schwarzes Bild entsteht, in dem kein Dunkelstrom zu erkennen ist. Möchte man Aufnahmen unterhalb dieser Grenze erstellen, muss man den *Gain*-Wert erhöhen. Dadurch kann man die Szene auch bei wenig Licht erkennen, jedoch nimmt mit Erhöhung des *Gain*-Werts, und damit der Verstärkung, das Rauschen zu. Dies ist in Abb. 2b zu erkennen, in der die Szene nur von einem Teelicht beleuchtet wird, und bei der die Aufnahme mit +18 dB verstärkt werden musste.

Ein weiterer Störfaktor bei der Verstärkung ist die Verstärkung selbst. Bei jeglicher Art von Verstärkung erkauft man diese mit Störungen, die in das verstärkte Signal einfließen. Diese Störungen sind dabei um so größer, je mehr verstärkt wird.

[6] dB (Dezibel) ist eine Maßeinheit zur Beschreibung eines Pegels oder Verstärkung. Es ist der zehnte Teil des dekadischen Logarithmus des Verhältnisses zweier Größen wie z. B. Eingangsleistung zu Ausgangsleistung.

1.3.2 Interlacing

Analoge Videoaufnahmen wurden hauptsächlich in Zeilensprungverfahren (*Interlaced*) aufgenommen und abgelegt. Dabei baut sich ein vollständiges Bild aus zwei Halbbildern auf. Ein Halbbild besteht dabei aus allen Bildzeilen mit gerader Nummer, das andere Halbbild aus Bildzeilen mit ungerader Nummer. Werden für das erste Halbbild die ungeraden Bildzeilen verwenden, spricht man von Upper- oder Top Field Übertragung, andernfalls von Bottom- oder Lower-Field Übertragung. Diese Verfahren wurde in der Anfangszeit der Fernsehübertragung insbesondere für Darstellungsgeräte mit Bildröhre entwickelt. Es hat zwei wesentliche Vorteile:

1. Das Flimmern der Darstellung wird reduziert und gleichzeitig wird die Bewegungsauflösung verbessert, da das Bild 50 mal in der Sekunde abgetastet bzw. aufgebaut wird,

2. gleichzeitig wird dabei nur die Hälfte der Bandbreite benötigt, da im Prinzip nur 25 vollständige Bilder übertragen werden.

Mit diesen Vorteilen erkaufte man sich allerdings auch einige Nachteile, die vor allem bei heutigen Wiedergabegeräten zum Tragen kommen:

1. Zeilenflimmern: Waagerechte Strukturen „tanzen“ hoch und runter, da sie genau zwischen zwei Zeilen liegen und mal in der oberen, mal in der unteren Bildzeile erscheinen.

2. Treppenbildung an schrägen Kanten und kammartige Ausfransungen in Standbildern bei Bewegungen (siehe 2a).

Für Geräte, die auf Bildröhren basieren, war das Zeilensprungverfahren ideal geeignet, da die leuchtenden Bildpunkte nacheinander, zeilenweise von einem Elektronenstrahl zum Leuchten angeregt wurden, und genau ein Halbbild lang leuchteten, bevor sie wieder von dem Elektronenstrahl angeregt werden mussten.

LCD- und Plasma-Panels, die die Bildröhren kontinuierlich aus dem Markt drängen, benötigen jedoch einen progressiven Bildaufbau, d.h. es muss immer das komplette Bild dargestellt werden. Interlaced Videobilder müssen also vor der Darstellung durch de-interlacing Algorithmen in progressive Videobilder umgerechnet werden, bevor sie dargestellt werden können.

Die neuen HDTV Videonormen berücksichtigen diese neuen technischen Entwicklungen und spezifizieren progressive Videobildübertragung, beinhalten jedoch zusätzlich aus Kompatibilitätsgründen auch interlaced Formate.

Da im Zeilensprungverfahren vorliegende Videobilder für die Verarbeitung besondere Herausforderungen und Probleme beinhalten, werden in

dieser Arbeit nur progressive Videobilder berücksichtigt. In Halbbildern vorliegendes Videomaterial wird vor der Verarbeitung in vollständige Bilder umgewandelt.

2 Filter

Filter nennt man in der Bildverarbeitung Operationen auf Rasterbildern[7], die einen Bildpunkt (einen Pixel[8]) eines Eingabebildes in einen Pixel eines Ausgabebildes transformieren. Man unterscheidet drei Klassen von Operationen, die sich durch die Anzahl der Pixel, die für die Transformation nötig sind, unterscheiden.

- **Punktoperatoren** Der Grau- bzw. Farbwert des Zielpixels ist nur vom Grau- bzw. Farbwert des Ausgangspixel und evtl. dessen Position abhängig. Beispiele für Punktoperatoren sind Histogrammäqualisation, Histogrammspreizung und Schwellwertfilter.
- **lokale Operatoren** Bei den lokalen Operatoren wird der neue Grau- bzw. Farbwert auf Basis der Nachbarpixel in einer begrenzten örtlichen Region ermittelt. Dabei unterscheidet man noch zwischen linearen und nichtlinearen Filtern. Beispiele für lineare Filter sind Mittelwertfilter und Gaußfilter, für nichtlineare Filter der Medianfilter und die morphologischen Filter Erosion und Dilatation.
- **globale Operatoren** Das Zielpixel ist bei den globalen Operatoren von allen Pixeln des Ausgangsbildes abhängig. Dazu wird z. B. das Bild vom Ortsraum in einen Frequenzraum transformiert, in dem nicht mehr die Farbwerte sondern die Frequenz- und Phasenanteile der zugrunde liegenden Punktfolge abgelegt sind. Dabei ist jeder Punkt im Frequenzraum von allen Punkten im Ortsraum abhängig und somit werden auch alle Pixel im Ortsraum nach einer Rücktransformation von allen Punkten im Frequenzraum beeinflusst. Typische Operatoren sind Beschneidungen von Werten im durch die Fourier-Transformation erzeugten Frequenzraum.

Mit verschiedenartig gestalteten Filtern können die vielfältigsten Veränderungen eines Bildes erzeugt werden. In der Bildverarbeitung ist es jedoch häufig das Ziel, ein gestörtes Signal zu verbessern. Daher gibt es eine sehr große Anzahl an Glättungsfiltern, die versuchen aus Signalen, oder im konkreten Fall aus Bildern, Störungen zu entfernen. Dabei können Informationen über die Art der Störung bei der Wahl des Filters entscheidend sein. Weiterhin beeinflusst auch die Verwendung des Bildes nach der Filterung die Wahl des Filters. In den meisten Fällen wird jedoch ein Glättungsfilter gesucht, der ein „Rauschen“ aus einem Bild entfernt und dabei „Kanten“ erhält, das Bild also nicht einfach nur weichzeichnet.

[7]Ein Rasterbild ist ein Darstellungsform eines Bildes, bei der das Bild rasterförmig in Bildpunkte unterteilt wird. Jeder Bildpunkt besteht in der Regel aus einer Farbe und ist rechteckig.

[8]Kunstwort aus engl. Picture (→Pics, →**Pix**) und **El**ement. Kleinste Einheit in einem Rasterbild.[Wikb]

Ein Beispiel der einfachen, *nicht* kantenerhaltenden Vertreter der Weichzeichnungsfilter ist der Gaußfilter aus der Klasse der lokalen Operatoren. Er berechnet einen gewichteten Durchschnitt von Pixeln in der Umgebung eines gegebenen Pixels, wobei sich das Gewicht mit zunehmendem Abstand verringert. Diese Art von Filter funktioniert sehr gut auf Regionen bei denen sich die Pixelwerte nur langsam ändern, oder anders gesagt die Region eine niedrige (Farb-)Wertveränderungs-Frequenz hat. Sie versagt jedoch an „Kanten" im Bild, also an Stellen an denen sich die Pixelwerte schnell ändert, also eine hohe Frequenz vorliegt. Diese Kanten werden dann ebenfalls wie das Rauschen weichgezeichnet, die Kante wird also unscharf. Deshalb spricht man bei diesem Filtertyp von Tiefpass-Filtern: Tiefe Frequenzen (langsame Farbwertänderungen) bleiben erhalten, hohe Frequenzen (schnelle Farbwertänderungen, z. B. Kanten) werden herausgefiltert.

Filter funktionieren nicht nur auf zweidimensionalen Bildern, sondern auch auf dreidimensionalen Volumen. Der Unterschied zwischen Bild- und Volumenfiltern ist dabei nur der *Filterkern*. Dieser gibt an, wie sich der aktuelle Pixel in Abhängigkeit von der eigenen Position und dem eigenen Wert und den Positionen und den Werten der umgebenden Pixel ändern muss. Der Filterkern beschreibt also den Filteroperator. Der Filterkern eines Volumenfilters bezieht nun nicht nur die *Ebene* in der der Pixel liegt, sondern das *Volumen,* in dem der Voxel[9] liegt, in die Berechnung ein. Der Vorgang des Filterns wird auch als Faltung des Bildes bzw. des Volumens mit dem Filterkern bezeichnet.

2.1 Kantenerhaltende Glättungsfilter

Da die Entrauschung von Bildern eine elementare Funktion in der Bildverarbeitung ist, wurde viel Forschung betrieben, um kantenerhaltende Glättungsfilter zu entwickeln. Im Folgenden ist eine Übersicht über verschiedene Filter und deren Eigenschaften gegeben.

2.1.1 Bilateral Filter

Der Bilateral Filter wurde 1998 von C. Tomasi und R. Manduchi vorgestellt [TM98]. Die grundlegende Idee beim Bilateral Filter ist, im *Ortsbereich* (Domain) und im *Wertebereich* (Range) zu arbeiten. Der *Ortsbereich* beschreibt den räumlichen Abstand, die *Nähe* zweier Pixel in einem Bild, wohingegen der *Wertebereich* den Abstand der Pixelwerte, die *Ähnlichkeit* zweier Pixel beschreibt. Die Ähnlichkeit sollte dabei die wahrgenommene Ähnlichkeit beschreiben.

Traditionelle Filter gewichten Pixelwerte nach dem räumlichen Abstand: Der Einfluss nimmt mit größerer *räumlicher Entfernung* ab. Beim Bilateral

[9]Kunstwort aus engl. **Vo**lume und Pi**xel.** Kleinste Einheit in einem Volumen.

Filter wird auch im *Wertebereich* gefiltert. Hier nimmt der Einfluss mit der *Unähnlichkeit der Pixelwerte* ab.

Filter die im *Wertebereich* arbeiten sind nicht-linear, weil die Gewichtung von den Pixelwerten und nicht von der Pixelposition abhängt. Es reicht aber nicht, nur im *Wertebereich* zu filtern, erst die Kombination mit dem *Ortsbereich* bringt das gewünschte kantenerhaltende Ergebnis.

Formal wird der Bilateral Filter beschrieben durch:

$$k(x) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} c(\xi, x)\, s(f(\xi), f(x))\, d\xi$$

mit der geometrischen Nähe $c(\xi, x)$ zwischen dem Nachbarschaftsmittelpunkt x und einem nahen Punkt ξ und der photometrischen Ähnlichkeit $s(f(\xi), f(x))$ zwischen dem Nachbarschaftsmittelpunkt x und einem nahen Punkt ξ. c arbeitet also im *Ortsbereich* und s im *Wertebereich*. Der Pixelwert bei x wird durch einen Durchschnittswert von *nahen* und *ähnlichen* Pixeln ersetzt.

Ein einfacher und wichtiger Fall der Abstandsfunktion im *Ortsbereich* ist der gaußgewichtete Euklidische Abstand:

$$c(\xi, x) = e^{-\frac{1}{2}\left(\frac{d(\xi,x)}{\sigma_d}\right)^2}$$

mit dem Euklidischen Abstand:

$$d(\xi, x) = d(\xi - x) = \|\xi - x\|$$

Die Abstandsfunktion im *Wertebereich* ist analog definiert:

$$s(\xi, x) = e^{-\frac{1}{2}\left(\frac{\delta(f(\xi),f(x))}{\sigma_r}\right)^2}$$

mit einem geeigneten Abstand für die beiden Intensitätswerte ϕ und f:

$$\delta(\phi, f) = \delta(\phi - f) = \|\phi - f\|$$

Dies kann im einfachen Fall der Betrag der absoluten Pixelwerte sein.

Die Werte σ_d und σ_r sind Parameter, durch die das Filterergebnis gesteuert werden kann. Mit σ_d kann die Stärke der Tiefpassfilterung gesteuert werden. Durch σ_r wird beeinflusst, ob eher ähnliche oder unähnliche Pixelwerte kombiniert werden, oder anders gesagt, $\sigma_r$bestimmt, wie stark die Unähnlichkeit sein darf, damit das Pixel noch verwendet werden darf.

Der Bilateral Filter ist in seiner ursprünglichen Form nicht separierbar, in [PvV05] wurde jedoch eine separierte Version vorgestellt. 2D-Filter sind dann separierbar, wenn das Ergebnis der hintereinander folgenden Ausführung zweier 1D Filter entspricht. Durch die Separierung kann der Berechnungsaufwand der Faltung verringert werden.

2.1.2 Anisotropische Diffusion

Ein älteres Verfahren zur kantenerhaltenden Glättung von Bildern ist die Anisotropische Diffusion [BSMH98]. Dabei wird Rauschen entfernt, indem das Bild durch eine partielle Differenzialgleichung geändert wird: An jedem gegebenen Punkt wird die lokale Varianz gemessen und die Pixelwerte nach dieser Varianz gewichtet.

Dazu wird z. B. im einfachsten Fall die *Diffusions-* oder *Wärmeleitungsgleichung* verwendet. Das Ergebnis der Anwendung der *Wärmeleitungsgleichung* entspricht dem des Gauß-Filters. Um nun eine kantenerhaltende Glättung zu erreichen, muss die *Diffusionsgleichung* durch eine Funktion ersetzt werden, die die Diffusion über Kanten hinweg aufhält.

Diese *Diffusionsgleichungen* sind immer von einem Parameter der Zeit abhängig und somit arbeitet diese Art von Algorithmen iterativ. Da die *Diffusionsgleichungen* partielle Differenzialgleichungen sind, leiden diese Algorithmen außerdem unter Stabilitätsproblemen und sind sehr berechnungsaufwändig.

2.1.3 Filter in der Wavelet Domäne

In den 1990er Jahren wurde die Wavelet-Transformation populär. Sie dient ebenso wie z. B. die Fourier-Transformation dazu, ein Signal von einem Ortsraum in einen anderen Raum zu transformieren, im Falle der Fourier-Transformation in den Frequenzraum. Bei der Fourier-Transformation bleibt jedoch nur die Information über die Frequenzamplitude erhalten, die Zeitinformation geht verloren. Dies folgt aus der Heisenbergschen Unschärferelation, nach der die Position und der Impuls eines sich bewegenden Teilchens nicht genau bestimmt werden kann.

Gegenüber den Sinus- und Cosinus-Funktionen der Fourier-Transformation besitzen Wavelets Frequenz- und Zeitlokalität. Dadurch können im Wavelet-Raum andere Analysen als im Fourier-Raum durchgeführt werden. Es gibt verschiedene Wavelet-Funktionen, wie z. B. das einfache und schon 1909 vorgeschlagene Haar-Wavelet oder das Daubechies D4 Wavelet, die sich z. B. in ihrer Stetigkeit und Differenzierbarkeit unterscheiden.

Nachdem ein Bild, oder allgemein ein Signal, in die Wavelet-Domäne überführt wurde, gibt es nun viele Möglichkeiten, das enthaltene Rauschen zu entfernen. So gibt es Verfahren, die durch Grenzwerte die Koeffizienten des Wavelets beschränken oder den Wiener-Filter [BV04] verwenden. Der Wiener-Filter stellt das ungestörte Signal durch Bestimmen des *minimalen mittleren quadratischen Fehlers* zwischen dem gegebenen Signal und Wissen über das Rauschen her.

2.2 Volumenfilter

Filter im allgemeinen sind wie erwähnt nicht auf Bilder beschränkt. In der Regel spricht man von der Filterung von Signalen, wobei dann ein Bild als 2D-Signal verstanden wird. Für Filter ist die Anzahl der Dimensionen des zu filternden Raums nicht beschränkt. Solange der Filterkern beschreibt, wie die verschiedenen Dimension in das Ergebnis einfließen, oder die Anwendung des Filterkerns auf eine für ihn passende Anzahl an Dimensionen beschränkt ist, können beliebige n-dimensionale Datenräume gefiltert werden.

Volumenfilterung für Abbildungen werden sehr häufig im medizinischen Bereich verwendet, da viele bildgebende Verfahren 3D-Datensätze generieren. Gleichzeitig sind die Daten von Magnetresonanztomographie- oder Computertomographie-Geräten stark verrauscht, da es technisch schwierig ist, starke Magnetfelder zu erzeugen, bzw. man die Röntgenstrahlenbelastung gering halten möchte.

Die Möglichkeit 3D-Filter auf Videosequenzen anzuwenden, wird nur selten in entsprechenden Videoapplikationen angeboten. Entsprechende Optionen werden meistens „temporale Filterung“ genannt. Volumenfilter auf Video anzuwenden, könnte jedoch, in Anbetracht folgender Darstellung, gute Ergebnisse liefern: Man könnte die Funktionsweise von 2D-Entrauschungsfiltern beschreiben in dem man sagt, dass der Filter versucht, homogene Flächen zu finden und das Rauschen durch Weichzeichnen der Fläche zu entfernen. Das Problem dabei ist, die homogenen Flächen, bzw. die sie eingrenzenden Kanten, auf der Ebene zuverlässig zu erkennen und diese nicht mit weichzuzeichnen.

Wenn man sich nun eine Videosequenz vorstellt, in der die Kamera eine feste Position hat, und der Bildinhalt sich nur geringfügig von Bild zu Bild ändert, liegen homogene Flächen in der Tiefe vor. Ein Pixel, das immer die gleiche Abbildung einer Szene darstellt, und seinen Farbwert dabei geringfügig ändert, tut dies wahrscheinlich, weil der Farbwert durch ein Rauschen verfälscht wird. Wenn man nun über die Zeit eine Weichzeichnung durchführt, wird das Rauschen entfernt. Natürlich ist die Szene bei Videoaufnahmen selten statisch. So benötigt man auch für die Tiefe im Videovolumen, die zeitliche Dimension, eine Kantendetektion. Eine Kante in der Zeit ist dann ein Objekt das sich bewegt und in einem Bild sich in einem Pixel befindet und im nächsten Bild in einem anderen Pixel. Über diese Kanten in der Zeit hinweg, darf dann natürlich auch nicht weichgezeichnet werden, da es sonst zu einem Geister-Effekt kommt. Die Bewegung des Objektes zeichnet sich schemenhaft vor und nach der eigentlichen Bewegung ab. Da Video in der Regel eine Bildwiederholrate von 25 und mehr Bildern pro Sekunde hat, erscheinen selbst schnelle Bewegungen bei einer Bild für Bild Betrachtung langsam, wenn nicht gar statisch.

Nach dieser Überlegung müsste ein 3D-Kantenerhaltender-Weichzeichnungsfilter auf einem Videovolumen das Rauschen besser entfernen können,

als sein 2D-Pendant. Aus dem selben Grund müssten weiterhin Artefakte, die durch eine reine 2D-Filterung entstehen, verringert werden. In Abschnitt III werden die erreichten Ergebnisse besprochen.

2.3 Auswahl des geeigneten Filters

Da es in dieser Arbeit nicht um einen Vergleich verschiedener kantenerhaltender Glättungsfilter geht, ist eine Beschränkung auf die Implementierung *eines* Filters sinnvoll. D. Barash hat in [Bar00] gezeigt, das eine Generalisierung des *Bilateral Filters* und der *Anisotropischen Diffusion* möglich ist, und eine gemeinsame Sicht auf beide Ansätze ermöglicht. Somit führen beide Ansätze bei der Wahl entsprechender Parameter zum gleichen Ergebnis.

Filter in der Wavelet-Domäne sind ein interessanter und aktueller Ansatz zur kantenerhaltenden Glättung. Ihre Stärken sind irreguläre und adaptive Darstellungen, jedoch sind solche Strukturen für die GPU nicht geeignet.

Da der Algorithmus des *Bilateral Filters* einfach und nicht iterativ ist, bietet er sich für eine performante Implementierung auf der GPU an. Weiterhin wurde in [PvV05] gezeigt, dass eine schnelle „quasi-separierte“ Version des *Bilateral Filters* eine gute Näherung des originalen *Bilateral Filters* liefert.

2.4 Fehlermaß

Um den Erfolg eines Entrauschungsfilters zu messen, benötigt man ein Fehlermaß. Damit kann man je nach Art des Fehlermaßes auf drei Arten die Güte eines gefilterten Bildes bestimmen:

1. *Full-reference*: Man besitzt neben dem gefilterten Bild auch ein Referenzbild, zu dem ein Abstand, der Fehler, bestimmt werden kann.

2. *No-reference*: Es existiert nur das gefilterte Bild. Aus diesem allein wird ein Maß, die Qualität, bestimmt.

3. *Reduced-reference*: Man besitzt von dem Referenzbild einige Eigenschaften, anhand derer man das gefilterte Bild beurteilen kann.

In der Praxis liegen häufig keine Referenzbilder vor, da es z. B. keinen Bildwandler gibt, der bei dunklen Szene ein rauschfreies Bild aufnehmen kann, mit dem ein Entrauschungsfilter auf einer verrauschten Aufnahme verglichen werden könnte. Jedoch gibt es keine zuverlässigen *no-reference* Messmethoden, so dass in der Regel Tests nach der *full-reference* Methode durchgeführt werden, da für diese anschauliche und einfache Algorithmen existieren. Eine unverrauschte Aufnahme wird also mit künstlichem Rauschen verschlechtert, und ein Entrauschungsfilter versucht, die unverrauschte Referenzaufnahme wiederherzustellen.

Ein Beispiel für die *full-reference* Methode ist der *Mean-Squared-Error* (*MSE*), der den Durchschnitt der quadrierten Differenzen der Pixelwerte zwischen dem gefilterten Bild und dem Referenzbild darstellt. Mit dieser Methode ist auch das *peak signal-to-noise ratio* (*PSNR*) verwandt. Diese Methoden haben eine klare physikalische Bedeutung, sind einfach zu berechnen und können gut in Optimierungsverfahren eingesetzt werden. Sie berücksichtigen aber in keinster Weise die menschliche Wahrnehmung. Deshalb sind sie für die Beurteilung der Qualität von Bildern schlecht geeignet.

Ein Fehlermaß, dass die menschliche Wahrnehmung berücksichtigt, ist das *structural similarity measure* (SSIM), das örtliche Muster von Pixelwerten zwischen gefiltertem Bild und Referenzbild vergleicht. Es wurde 2004 von Zhou Wang [WBSS04] vorgestellt. Er zeigte, dass dieses Fehlermaß im Vergleich zu MSE und anderen Fehlermaßen einem objektiv ermittelten Fehlermaß besser entspricht.

Da es in dieser Arbeit um die Verbesserung der wahrgenommenen Bildqualität geht, wird für die Bewertung der Ergebnisse dieser Arbeit das SSIM Fehlermaß verwendet.

3 GPGPU

Die Abkürzung GPGPU steht für **G**eneral **P**urpose Computation on **G**raphics **P**rocessing **U**nit und beschreibt damit die Berechnung von allgemeinen Problemen auf der Graphikkarte statt auf der CPU [OLG+05].

Die ersten Graphikkarten, bzw. Zusatzkarten, für den Heimgebrauch besaßen eine Verarbeitungspipeline mit fest vorgegebenen Berechnungsschritten (Fixed-Function-Pipeline). Sie boten die erste Möglichkeit, um die Darstellung von dreidimensionalen Modellen zu beschleunigen. Der Prozessor auf der Graphikkarte, die GPU, bekam eine Reihe von Parametern wie Position und Farbe von Geometriepunkten, Informationen über Lichtquellen und Kameraparametern und berechnete anhand dieser Daten Pixelwerte und stellte diese dar. Durch diesen spezialisierten Prozessor konnten dreidimensionale Welten mit einer Qualität und Geschwindigkeit dargestellt werden, die bis dahin nur von nicht-spezialisierten Hochleistungscomputern oder speziellen Graphikcomputern erreicht wurde. Da diese Graphik- und Zusatzkarten vom Preis und den Fähigkeiten her auf den Computerspieler ausgerichtet waren und von diesen sehr gut angenommen wurden, konnten sich diese Karten schnell und in hoher Menge im Markt etablieren. Dies führte zu einer starken und schnellen Entwicklung in diesem Bereich, aber auch zu einem erbitterten Konkurrenz- und Preiskampf, so dass heute nur noch die beiden Firmen NVIDIA und ATI (seit 08/2006 AMD) eine Rolle spielen. Die Entwicklung der Graphikkarten ist mittlerweile so weit, dass Spezialcomputer nicht mehr an die Leistung moderner Graphikkarten heranreichen.

Da durch die Fixed-Function-Pipeline das Aussehen, bzw. Renderergebnis der erzeugten Welten stark vorgegeben und von den Anwendungsentwicklern kaum zu beeinflussen war, ging man im Laufe der Entwicklung dazu über, Teile der Pipeline durch programmierbare Stufen zu ergänzen. So konnte z. B. von dem standardmäßig vorgegebenen Gouraud-Shading [Gou98] abweichend ein anderes Shading-Verfahren wie Phong-Shading [Pho75] implementiert werden. Diese programmierbaren Stufen weckten rasch das Interesse daran, mit ihrer Hilfe nicht graphikspezifische Berechnungsaufgaben zu lösen.

Dieses Interesse wurde auch dadurch geschürt, dass die GPU SIMD[10] Berechnungen mit vielen parallel arbeitenden Einheiten (aktuell bis zu 24 Pixel-Pipelines) durchführt und daraus ihre hohe Geschwindigkeit bezieht. Diese Architektur der GPU ist grundlegend anders als die der klassischen CPU. Die aktuellen Entwicklungen der GPU und CPU gehen allerdings im Augenblick in einigen Bereichen aufeinander zu. Der starre Pipelinegedanke der GPU wird aufgelöst und flexiblere Lese- und Schreibmöglichkeiten aus

[10] engl. **S**ingle **I**nstruction **M**ultiple **D**ata. Beschreibt die gleichzeitige Anwendung einer Rechenoperation auf mehreren Daten, z. B. die gleichzeitig stattfindende, komponentenweise Addition zweier Vektoren, die aus jeweils vier float Werten bestehen.

und in den Speicher eingeführt und andererseits erhalten moderne CPUs immer mächtigere SIMD Einheiten und mehrere Kerne mit eigenem und gemeinsam genutzten Speicher. Es ist daher wahrscheinlich, dass in einigen Jahren der Begriff GPGPU verschwinden wird, da ein oder mehrere Prozessorkerne mit verschiedenen Verarbeitungskonzepten die heute getrennten Aufgaben der CPU und GPU vereinen werden.

Heute jedoch wird der GPGPU-Gedanke von den Graphikkartenherstellern aktiv unterstützt. So gehen aktuelle Trends in die Richtung, mehrere Graphikkarten (SLI / CrossFire) pro Rechner und mehr GPUs pro Graphikkarte einzusetzen. Einerseits natürlich um die Berechnungsmenge der immer komplexer werdenden dreidimensionalen Welten aufzuteilen und somit die visuelle Qualität weiter zu steigern, andererseits um eine weitere GPU zur Berechnung von z. B. Physiksimulationen (Havok FX™) zu verwenden.

3.1 Programmiersprachen für die GPU

Mit der Möglichkeit Teile der Graphikkartenpipeline programmieren zu können, kam auch der Wunsch nach praktischen Programmiersprachen auf. So war es in der Anfangszeit nur mit einer zum jeweiligen Graphikchip passenden Assemblersprache möglich, eigene Programme, auch Shader genannt, in der Pipeline zu definieren. Im Jahre 2002 wurde mit den OpenGL Erweiterungen `GL_ARB_vertex_program` und `GL_ARB_fragment_program` eine Assemblersprache für die Vertex- und Fragmentbearbeitung standardisiert, die auf jedem Graphikchip, der diese Erweiterungen unterstützt, funktioniert.

Assemblersprachen haben den Nachteil, dass sie unintuitiv zu schreiben und lesen sind. Aus diesem Grund entwickelten sich rasch einige Hochsprachen für die Programmierung der Shader. Mit den Hochsprachen wurden Compiler entwickelt, die die Programme in die Maschinensprache der jeweiligen Graphikchips übersetzten. So war es nun möglich, gut wartbare Programme zur Ausführung auf der GPU zu entwickeln. Einige Beispiele für Shader-Hochsprachen sind:

- HLSL - High Level Shading Language - Microsoft [Bly06]
- Cg - C for Graphics - NVIDIA [MGA03]
- GLSL - OpenGL Shading Language - OpenGL ARB [Ros04]

Diese Hochsprachen haben alle eine den Programmiersprachen C und C++ ähnlichen Syntax. Da die Fähigkeit zur Programmierung der Graphikkarte in erster Linie zur Beeinflussung der Darstellung, dem Shading, von dreidimensionalen Graphiken dient, finden sich viele Konzepte aus der *RenderMan Shading Language* wieder [Ros04]. Auch wenn RenderMan viele Jahre älter

ist als moderne Shader-Sprachen und für das Offline-Rendering[11] entwickelt wurde, nutzt man heute diese Erfahrung, und ist sogar in der Lage, viele damals mit RenderMan erzeugte Effekte in Echtzeit zu berechnen.

Von einigen geringen Unterschieden in der Syntax abgesehen, unterscheiden sich HLSL / Cg und GLSL an der Stelle, an der die Programme kompiliert werden. Während bei HLSL / Cg mit einem Compiler Binärcode erzeugt wird, der von dem Graphiktreiber in die GPU geladen wird, übernimmt bei GLSL der Graphiktreiber selbst das Kompilieren. So können die jeweiligen Graphikchiphersteller den Compiler auf die jeweiligen Besonderheiten der Chips optimieren. Ein Nachteil ist jedoch, dass der Quelltext der Shaderprogramme in der Applikation abgelegt ist. Hersteller von kommerziellen Produkten können somit ihr geistiges Eigentum und damit evtl. innovative Entwicklungen nicht ohne weiteres schützen. Es ist jedoch auch bei Binärdaten möglich durch Disassemblierung an den Quelltext zu gelangen.

Nachdem sich die Hochsprachen für die Shaderprogrammierung durchgesetzt hatten und die GPU zunehmend für die Berechnung nicht graphikspezifischer Probleme verwendet wird, ist der Wunsch nach komplexen, wahlfrei zugreifbaren Datenstrukturen vorhanden. Mit ihren Ursprüngen in der Computergraphik gelegen, sind die grundlegenden Datenstrukturen die Texturen. Diese können 1D, 2D und 3D sein, und zwischen einer und vier Komponenten mit unterschiedlichen Auflösungen haben. Der Zugriff auf die Texturen ist für sequentielles Lesen optimiert. Für die Lösung allgemeiner Probleme sind aber Datenstrukturen wie *Stack* und *Baum* nötig. Anstatt sich jedes Mal eine Abbildung von einem Stack auf eine Textur zu überlegen und zu implementieren, wäre eine Sammlung generischer Datenstrukturen nützlich. Eine derartige, der STL (Standard Template Library) ähnlichen Bibliothek wurde von Lefohn et al. in [LKS+06] vorgestellt. Eine Implementierung dieser „GPU-STL" mit dem Namen *Glift* lag zum Zeitpunkt dieser Arbeit jedoch noch nicht vor.

3.2 OpenGL Shading Language

Für diese Arbeit wurde zur Programmierung der Graphikkarte die OpenGL Shading Language gewählt. GLSL ist sehr gut in OpenGL integriert, da sie von dem gleichen Konsortium wie OpenGL spezifiziert wird und dabei auf größtmögliche Kompatibilität und Integration geachtet wird. Weiterhin steht die Spezifikation der Sprache für jeden öffentlich bereit und durch die große Anzahl an Unternehmen im OpenGL ARB ist eine langfristige Weiterentwicklung wahrscheinlich. Dem gegenüber werden die Sprachen Cg und HLSL nur von jeweils einer Firma kontrolliert. Dies hat den Vorteil, dass

[11] Offline-Rendering bezeichnet Rendertechniken, die nicht echtzeitfähig sind, sondern durchaus mehrere Stunden Rechenzeit pro Bild benötigen.

neue Entwicklungen schnell in die Sprache einfließen können, andererseits viele Erweiterungen in unterschiedlicher Richtung vielleicht eher schnell als sorgfältig integriert werden. Dagegen geht die Integration von GLSL in die Treiber der Graphikkartenhersteller eher langsam vonstatten. Weiterhin ist eine Verbreitung von GLSL und Cg über verschiedene Plattformen vorhanden. So gibt es GLSL und Cg Unterstützung für Microsoft Windows, Apple Mac OS X und Linux, wohingegen HLSL nur für Microsoft Windows und in einer ähnlichen Form für die Xbox 360 verfügbar ist.

3.2.1 Programme für die GPU

Wird die GPU für allgemeine Berechnungen verwendet, wird in der Regel das *Stream-Processing* Modell zur Anschauung verwendet. Bei diesem Modell gibt es Ein- und Ausgabedaten *(Streams)* auf denen eine Anzahl an Operationen *(Kernel)* durchgeführt werden. Wichtig dabei ist die Datenunabhängigkeit und -lokalität, d. h. die Ausgabedaten hängen nur von den Eingabedaten ab und es können keine weiteren Daten von einem Kernel in den nächsten übergeben werden. Durch diese Einschränkungen kann eine effiziente Speicherverwaltung und ein hoher Grad an Parallelität erreicht werden. Ein Beispiel hierfür ist das sog. *Latency-Hiding* von Speicherzugriffen. Dabei werden Zugriffe auf den Texturspeicher im Verhältnis zu Berechnungsaufgaben so sortiert, dass die Wartezeit auf die Texturdaten mit Berechnungsaufgaben ausgefüllt wird. Wird die GPU als Stream-Prozessor verwendet, gelten genau diese Einschränkungen, es gibt jedoch mit fortschreitender Entwicklung immer mehr Ausnahmen von diesen Beschränkungen.

Da die GPU jedoch ursprünglich nicht als Stream-Prozessor gedacht war, sondern als Graphik-Prozessor, unterscheidet sich die Nomenklatur zu dem *Stream-Processing* Paradigma. Es lässt sich jedoch eine einfache Übersetzung anwenden: Die *Stream* genannten Datenströme sind das, was als *Texturen* bezeichnet wird. Auf diese werden *Shader* angewendet, die die *Kernel* repräsentieren. Bei der GPU unterscheidet man zusätzlich zwischen *Vertex*- und *Fragment*-Shadern. Die Unterscheidung betrifft im Fall der GPU die Fähigkeiten, die die *Kernel* beim Zugriff auf Eingabestreams und Eigenschaften der Ausgabestreams haben. Weiterhin ist es bei der GPU möglich, dass der *Vertex*-Shader zusätzlich einige wenige Daten an den *Fragment*-Shader weitergeben kann.

Bei der Programmierung ist zu berücksichtigen, dass nicht beliebig separat *Vertex*- und *Fragment*-Shader auf die GPU geladen werden können. Vielmehr müssen die Shader zuerst kompiliert werden, und diese Shader-Objekte zu einem *Shader-Programm* zusammen gelinkt werden. Von diesem *Shader-Programm* kann dann eins, oder für Multipass-Rendering mehrere, auf die GPU geladen werden.

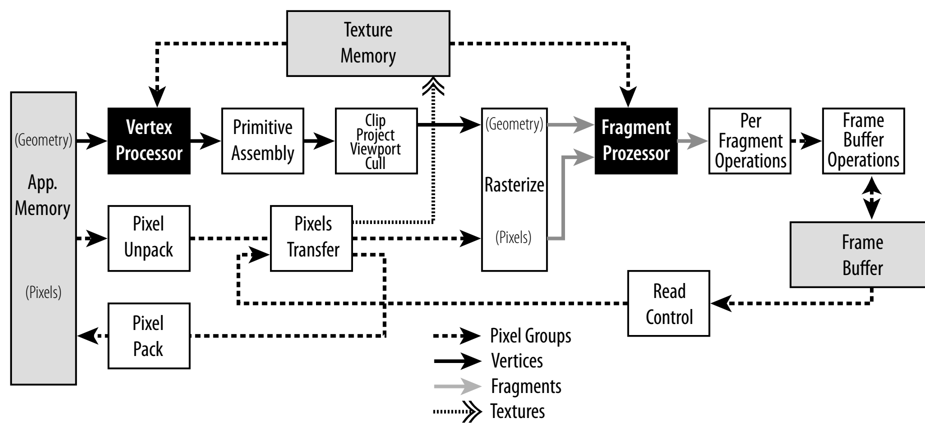

Abbildung 3: OpenGL Pipeline mit programmierbaren Stufen (schwarz), die an die Stellen der Fixed-Funktion Stufen gerückt sind.

3.2.2 Von der Graphikkarte zum universellen Coprozessor

Die ursprüngliche OpenGL Pipeline sah vor, die Geometrie und Texturen, die dargestellt werden sollen, in den Speicher der Graphikkarte zu laden, alle Vertices der Geometrie entsprechend der Kameraeinstellungen zu transformieren, zu beleuchten und zu texturieren und das Ergebnis in den Framebuffer zu schreiben und anzuzeigen. Dieser Ablauf ermöglicht natürlich nicht, die GPU als allgemein verwendbaren Coprozessor zu benutzen, da es keinen Sinn macht, die Ergebnisse der Berechnungen auf dem Monitor darzustellen. Vielmehr werden die Ergebnisse zur weiteren Verarbeitung wieder im Hauptspeicher benötigt.

Deshalb wurde die Pipeline so erweitert, dass auch der Transfer der Pixel vom Framebuffer in den Hauptspeicher möglich ist. Die OpenGL Pipeline ist in Abb. 3 dargestellt. Man kann dort auch das *Stream-Processing* Paradigma erkennen. Der Vertex Prozessor hat als Ein- und Ausgabe Vertices, die im weiteren Verlauf durch den Rasterisierer Fragmente ergeben. Diese dienen dem Fragment Prozessor als Eingabe, als Ausgabe liefert er die Pixel, die nach weiterer Verarbeitung im Framebuffer landen. Man kann auch erkennen, dass das *Stream-Processing* Paradigma etwas aufgeweicht wurde. So ist es beiden Prozessoren *lesend* möglich, auf den Texturspeicher zuzugreifen.

Um die Verwendung als universellen Coprozessor weiter zu vereinfachen, ist es weiterhin möglich, als Framebuffer eine Textur zu verwenden und in diese zu schreiben. Somit müssen die Daten nicht zuerst aus dem Framebuffer in den Hauptspeicher geschrieben und von dort wieder zurück auf die Graphikkarte als Textur geladen werden. Diese Möglichkeit wurde erst als sog. *pBuffer* umgesetzt und mit den *Frame-**B**uffer-**O**bjects* weiter generalisiert, beschleunigt und vereinfacht.

Die FBOs wurden Anfang 2005 als eine der umfangreichsten OpenGL Erweiterung eingeführt und zusammen mit der OpenGL Shading Language im aktuellen OpenGL 2.0 in den Standard übernommen.

3.3 Filtern auf der GPU

Die hohe parallele Datenverarbeitungsleistung der GPU bietet sich zum Filtern von Bildern und Volumen an. So gibt es seit OpenGL 1.2 Funktionen zum Filtern des Pixel Transfers im *Imaging Subset*. Man kann lineare 1D- und 2D-Filterkerne definieren und deren Anwendung beim *Pixel Transfer*[12] aktivieren. Vom *Imaging Subset* wird jedoch keine Filterung von Volumen angeboten und bei den Filterkernen ist man auf lineare Kerne beschränkt. Eine Übersicht über die Bildverarbeitungsfunktionen von OpenGL gibt Rost in [Ros96].

Nahezu unbeschränkt in der Art der Filter ist man, wenn man die Filteroperation in einem Shaderprogramm realisiert. Dazu bietet sich der *Fragment-Shader* an. Durch das Streaming Konzept der GPU kann man während der Anwendung eines *Fragment-Shader-Programms* auf ein Fragment nicht auf die Nachbarfragmente zugreifen. Deshalb ist der Ablauf beim Filtern folgender: Das zu filternde Bild wird als Textur auf die Graphikkarte geladen. Danach wird in einen Framebuffer mit gleicher Größe wie die Textur ein *Quad*, eine quadratische Fläche mit vier Eckpunkten, gerendert. Dabei kann der Framebuffer entweder der Front- bzw. Backbuffer sein, oder, wenn man ein Bild filtern und nicht anzeigen will, ein FBO und somit wieder eine Textur sein. Nun wird für jedes Fragment, das einem Pixel in der Ausgangstextur entspricht, das *Fragment-Shader-Programm* aufgerufen. Das Programm kann nun auf jedes Pixel, die bei FBOs immer auch Texel[13] entsprechen, in der Ausgangstextur lesend zugreifen, seine Berechnung durchführen und dem aktuellen Fragment den entsprechenden Farbwert zuweisen. So entsteht im Framebuffer das gefilterte Bild.

Dieses Vorgehen funktioniert auch fast genauso für Volumen. Dabei muss das FBO eine 3D-Textur repräsentieren. Dann kann in diese Textur ebenenweise geschrieben werden, nicht in einem Durchgang für alle Voxel, da in OpenGL das Rendern in ein Volumen nicht definiert ist. Allerdings hat sich herausgestellt, dass selbst das ebenenweise Schreiben in eine 3D-Textur von keinem aktuellen Graphikkartentreiber unterstützt wird (siehe 6.3).

[12] Der *Pixel Transfer* findet bei folgenden OpenGL Operationen statt: glCopyPixels, glDrawPixels, glReadPixels und glTexImage1D bzw. glTexImage2D.

[13] Kunstwort aus engl. **Tex**tur und **El**ement.

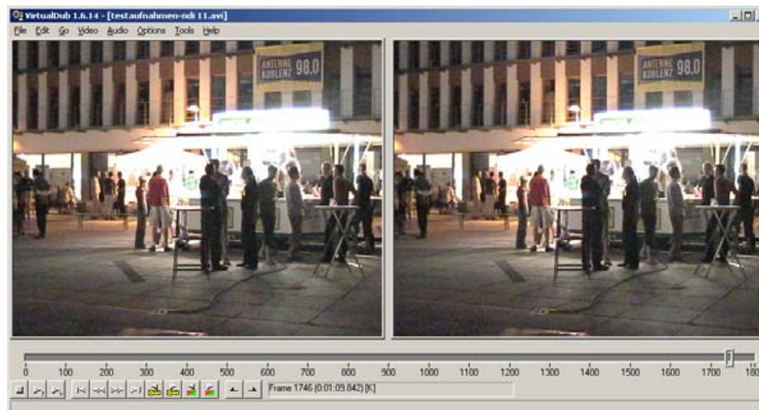

Abbildung 4: VirtualDub GUI

4 Applikationen

Es gibt für den PC eine recht große Auswahl an Videoapplikationen, die sich jedoch in der Regel mit ihren Funktionen auf einen bestimmten Bereich beschränken. So gibt es die Gruppe von Programmen für das Compositing, die im Prinzip die Funktionen von Bildbearbeitungsprogrammen wie Photoshop für den Umgang mit Film statt mit einzelnen Bildern erweitern. Dann gibt es die Programme, die hauptsächlich für den Schnitt von Filmsequenzen benutzt werden können. Zum Schluss gibt es dann noch eine Vielzahl an Tools, die Aufgaben wie z. B. Formatkonvertierung und einfache Filterung durchführen. Im Folgenden wird eine kurze Übersicht über einige Videoapplikationen und deren Verwendungsmöglichkeiten gegeben.

Alle vorgestellten Applikationen bieten eine eigene Plugin-Schnittstelle, über die Funktionalität von externen Entwicklern in die Programme eingebunden werden kann. Dazu gehören insbesondere Filter zur Bearbeitung der Videobilder.

4.1 Videoapplikationen

4.1.1 VirtualDub

Das von Avery Lee unter der GNU General Public License entwickelte Videoschnittprogramm *VirtualDub*[14] (Abb. 4) bietet nur einen kleinen Funktionsumfang, aber eine hohe Geschwindigkeit. Es wurde hauptsächlich entwickelt, um schnell und unkompliziert AVI Dateien zu schneiden, zu filtern und neu zu komprimieren. Das Schneiden beschränkt sich auf das Entfernen von Frames und Anhängen von weiteren AVI Dateien. Das dokumentierte Filtersystem, das durch einen Plugin-Mechanismus um neue Filter

[14] `http://www.virtualdub.org`

erweitert werden kann, ist insofern beschränkt, dass es immer nur ein Frame an den Filter geben kann und ein Frame von dem Filter verlangt. Es ist somit nicht ohne weiteres möglich, zeitabhängige Filter zu implementieren.

Eine Aufgabe, die das Programm mit unerreichter Geschwindigkeit und im Batch-Modus für viele verschiedene Dateien bearbeiten kann, ist z. B. das Deinterlacen und Ändern der Größe von DV Videos und Komprimieren des Videostroms mit *XviD* und des Audiostroms mit *MP3* zu Archivierungszwecken.

4.1.2 AviSynth

AviSynth[15] ist ein OpenSource FrameServer der unter der GNU General Public License entwickelt wird. Als FrameServer besitzt AviSynth keine eigene GUI, die gesamte Funktionalität wird über eine Skript-Datei gesteuert. Diese Skript-Datei (Endung .avs) dient als Ersatz für eine Videodatei als Eingabedateien für andere Videoapplikationen. Dabei verhält sich die Skript-Datei aus Sicht des Videoprogramms wie eine AVI-Video Datei.

Mit Hilfe der von *AviSynth* definierten Skriptsprache ist es möglich komplexe Abläufe, auch mit Bedingungen, zu beschreiben. Dabei wird zuerst eine Quelle für Video und Audio Daten angegeben (Datei, Stream) und dann eine Reihe von Filtern definiert. Diese Filter können zum einen traditionelle Filterungen durchführen, es können mit ihnen aber auch Frames gelöscht, Audioströme ergänzt oder entfernt, also Videoschnittaufgaben übernommen werden.

Durch die große Community rund um das Projekt gibt es eine große Anzahl an Filtern, die im Programmpaket mitgeliefert werden oder im Internet verfügbar sind. Da sich mit dem Programm sehr einfach längere Verarbeitungsabläufe realisieren lassen, wird es gerne zur Restaurierung von digitalisiertem Filmmaterial von alten VHS-Bändern oder abgefilmten 8mm Aufnahmen verwendet. Für diese Anwendung gibt es auch sehr viele „Verbesserungsfilter“ die in der Orts- und/oder Zeit-Domäne arbeiten. Ebenfalls stehen Filter zur Verfügung, die die GPU zur Beschleunigung verwenden.

4.1.3 Adobe AfterEffects

Das Compositing Programm *AfterEffects*[16] (Abb. 5) von Adobe bringt eine Vielzahl von Filtern in Form von Plugins mit, u. a. den Entrauschungsfilter „Körnung entfernen“. Dieser Filter stammt ursprünglich aus dem Plugin-Paket „grain surgery v2“ von der Firma „Visual Infinity“ und ist seit *AfterEffects* 6.5 Bestandteil des Adobe Produkts.

[15] `http://www.avisynth.org`
[16] `http://www.adobe.com/products/aftereffects/`

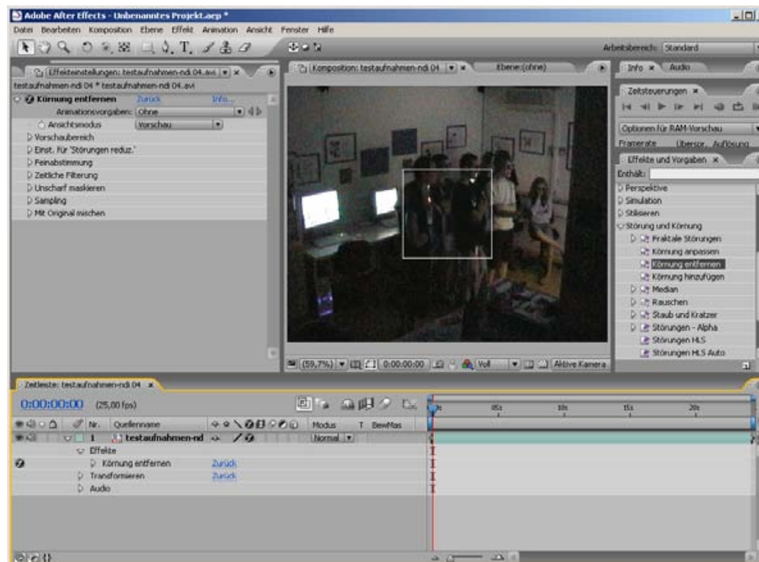

Abbildung 5: Adobe AfterEffects 7.0 GUI

Der Algorithmus, der diesem Filter zu Grunde liegt, ist nicht bekannt, da es sich um ein kommerzielles Produkt handelt. Man kann jedoch erkennen, dass er iterativ arbeitet, da man die Anzahl der Durchgänge einstellen kann. Weiterhin versucht er homogene Flächen zu finden, um aus ihnen Charakteristiken des Rauschens zu ermitteln. Dieser Filter bietet sehr viele Einstellungsmöglichkeiten für den Benutzer, da es ein Ziel des Filters ist, nicht das Rauschen bestmöglich automatisch zu entfernen, sondern gerade dem Benutzer die Möglichkeit zu geben, zu entscheiden, wie das Endergebnis aussehen soll. Dazu gehören Parameter für die Filterung von einzelnen RGB Kanälen, die Chrominanz, Sampling, Nachschärfen, Beitrag über die Zeit und weitere Parameter. Dieser Filter verwendet bei seiner Arbeit die CPU. Filter die die GPU zur Beschleunigung verwenden, werden erst seit der aktuellen Version langsam eingeführt.

Adobe stellt eine gute Dokumentation der Plugin-Schnittstelle zur Verfügung, die auch in anderen Programmen verwendet wird, wie z. B. in *Adobe Premiere Pro* oder *Apple Final Cut Pro*. Dadurch gibt es ein großes Angebot von verschiedensten Filtern von Drittherstellern. Außerdem bestünde die Möglichkeit, dass das in dieser Arbeit entwickelte Framework zusammen mit den Filtern an *Adobe AfterEffects* angebunden und genutzt werden könnte. Dies ist allerdings eine Arbeit für zukünftige Weiterentwicklungen.

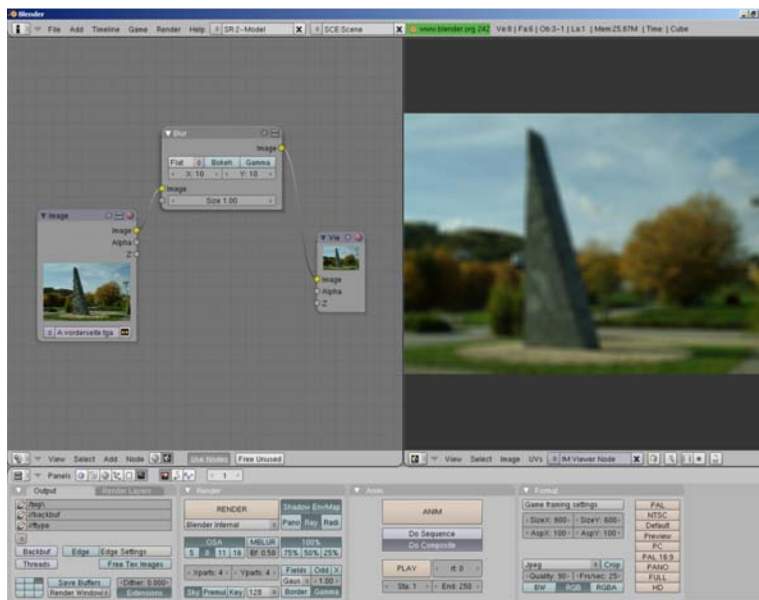

Abbildung 6: Blender GUI

4.1.4 Blender

Blender[17] (Abb. 6) entstand ursprünglich als kommerzielles 3D-Modellierungs- und Animationsprogramm. Mit der Version 2.0 im Jahr 2000 erhielt es eine integrierte Game-Engine. Nach dem Konkurs der Entwicklungsfirma *Not-a-Number* konnten durch eine Spendenaktion die Rechte am Quellcode durch dessen große Community erworben werden. Dieser wird nun unter der GNU General Public License unter dem Dach der nicht kommerziellen *Blender Foundation* weiter gepflegt. Mit der Version 2.42 wurde ein Node basiertes Compositing Modul integriert.

Das Compositing Modul kann separat verwendet werden, indem Knoten verwendet werden, die Bilddaten aus Dateien laden und in Dateien speichern können. Seine Stärke spielt es jedoch aus, wenn es in direkter Integration mit dem restlichen System verwendet wird. So stehen Renderpasses und die kompletten Informationen über den 3D-Raum zum Compositing zur Verfügung.

In dieser ersten Version ist die Anzahl der verfügbaren Manipulationsknoten noch gering und es stehen hauptsächlich grundlegende Funktionen wie Gaußfilterung, Tonwertkorrektur und logische und mathematische Funktionen zur Verfügung.

[17] `http://www.blender.org`

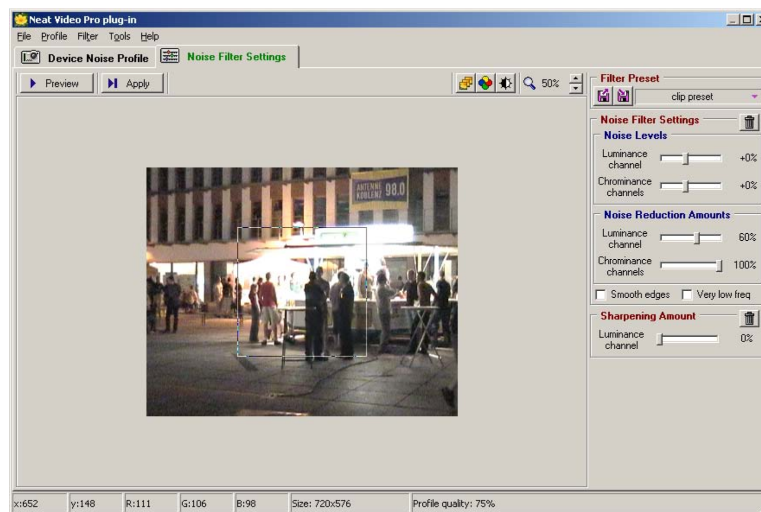

Abbildung 7: Neat Video GUI

4.2 Videofilter

4.2.1 Neat Video

Das Plugin *Neat Video*[18] (Abb. 7) steht für die Schnittstellen von *Adobe AfterEffects*, *Adobe Premiere* und *VirtualDub* zur Verfügung. Dieser Filter ist zum Entfernen von digitalem Rauschen, analogem Korn und Kompressionsartefakten entwickelt worden. Er basiert auf der gleichen Technik wie der Bildfilter *Neat Image* von den gleichen Entwicklern. Er wurde in der Videoversion um eine temporale Komponente erweitert.

Die verwendeten Algorithmen sind nicht bekannt. Das Verfahren ist zweistufig: In der ersten Phase wird ein Rauschprofil erstellt. Dieses kann automatisch aus dem zu filternden Bild gewonnen werden, oder über eine definierte Prozedur spezifisch für ein bestimmtes Aufnahmegerät erstellt werden. Nachdem das Rauschprofil erstellt wurde, kann das Video gefiltert werden. Dabei kann man das Ergebnis anhand einiger Parameter wie z. B. der Stärke der Filterung oder Nachschärfung beeinflussen.

[18] `http://www.neatvideo.com`

Teil II
Praktische Umsetzung

Die praktische Umsetzung dieser Arbeit wurde mit Microsoft Visual Studio 2005[19] in der Programmiersprache C++ und mit der OpenGL API[20] durchgeführt. Dadurch entsprechen im Folgenden Beispiele für Funktions- und Parameternamen dieser Syntax. Jedoch sind die Namen so eindeutig, dass deren Pendant in anderen Programmiersprachen oder bei Verwendung einer anderen Graphik API ohne Probleme zu finden ist.

5 Das Framework

5.1 Konzeption

Diese Arbeit umfasst die Erstellung eines Frameworks. Was aber ist ein *Framework*, und worin unterscheidet es sich beispielsweise von einer *Klassenbibliothek*?

Ein *Framework* im Bereich der objektorientierten Programmierung ist eine kontextspezifische Sammlung von Klassen und Interfaces und ein Regelwerk zu deren Verwendung. Dabei unterscheidet man grob zwischen *Whitebox-Framework* und *Blackbox-Framework*. Das *Whitebox-Framework* stellt nur wenige konkret implementierte Klassen und viele Interfaces und Regeln bereit. Um mit einem *Whitebox-Framework* eine Anwendung erstellen zu können, muss der Entwickler sehr viel Funktionalität selbst programmieren.

Das *Blackbox-Framework* stellt dagegen eine Vielzahl von konkreten Klassen und Funktionalität bereit und die Anwendungsentwicklung hat ihren Schwerpunkt im Zusammensetzen der Module. Dadurch ist in der Regel eine schnellere Anwendungsentwicklung möglich.

Eine *Klassenbibliothek* hingegen stellt, unabhängig von konkreten Anwendungen, Klassen zur universellen Verwendung zur Verfügung. Die Interfaces sind ebenfalls wohl definiert, die Art der Verwendung in einem größeren Rahmen wird aber nicht vorgegeben.

Ein Beispiel für eine *Klassenbibliothek* ist die STL (Standard Template Library). Sie stellt für die Programmiersprachen C++ grundlegende Datenstrukturen mit passenden Algorithmen zur Verfügung. Das von IBM entwickelte Open-Source *Framework Eclipse* dient zur Entwicklung von Rich-Client-Applikationen[21]. Beispiele für Anwendungen, die mit diesem Frame-

[19] `http://www.microsoft.com/germany/msdn/vstudio/`

[20] `http://www.opengl.org/`

[21] Rich-Client: Softwarekonzept bei der die Anwendung und die Daten auf dem Computer des Benutzers liegen und verarbeitet werden. Es ist das Gegenteil des Thin-Client Konzeptes, bei dem die Anwendung auf einem Server läuft und der Computer des Benutzers nur für die Darstellung der Daten verwendet wird.

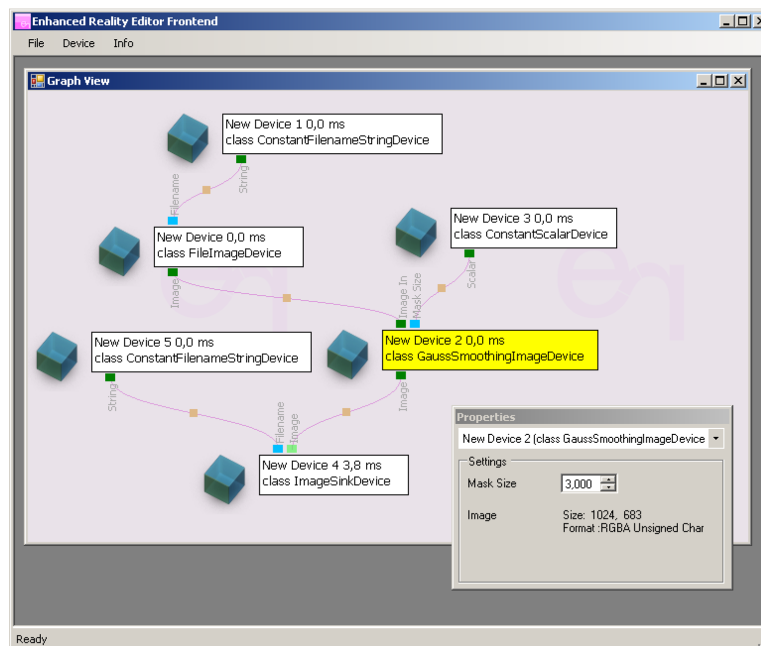

Abbildung 8: Einfacher Graph mit Plexus

work erstellt wurden ist die Eclipse Entwicklungsumgebung oder der *IBM Workspace Client*, der Nachfolger von *Lotus Notes*. Ein weiteres Beispiel für ein Framework ist *Tinymail*, welches alle nötigen Komponenten bereitstellt, um ein E-Mail-Programm zu entwickeln. Dabei kann man sich je nach verfügbarer Performance der Zielplattform die nötige und gewünschte Funktionalität nach Bedarf zusammenstellen.

Da der Schwerpunkt dieser Arbeit auf der Implementierung eines Bildfilters auf der GPU liegt und nicht auf der Entwicklung eines Frameworks an sich, wurde als Grundlage das *Plexus-Framework* verwendet und um Komponenten erweitert, die für die Filterung von Bildern, Volumen und Video nötig waren. Aus Sicht dieser Arbeit wurde das Plexus-Framework als *Whitebox-Framework* angesehen, da nur eine geringe Basisfunktionalität verwendet wurde. Die große Anzahl an vorhandenen, konkret implementierten Modulen, z. B. für die Verarbeitung von Meshes und HDR Kameras, wurde nicht benötigt, könnte das *Plexus-Framework* aus einer anderen Anforderung heraus aber auch zu einem *Blackbox-Framework* werden lassen.

5.2 Plexus

Beim ersten Kontakt präsentiert sich Plexus zuerst einmal als Editor für eine Art Datenflussdiagramm (siehe Abb. 8). In diesem Editor ist es möglich, verschiedene Komponenten, *Devices* genannt, auf einer Ebene anzuordnen und über Verbindungen einen Datenfluss zwischen den Devices herzustellen.

Ein solcher Graph könnte beispielsweise folgende Funktion haben (siehe Abb. 8):

1. Ein Device lädt ein Bild aus einer Datei und stellt es zur Verfügung (FileImageDevice)

2. Das nächste Device wendet einen Gauß-Filter auf das Bild an (GaussSmoothingImageDevice)

3. Als letztes wird dieses gefilterte Bild nun wieder in eine Datei geschrieben (ImageSinkDevice)

Plexus ist aber nicht der Editor, sondern die Komponente, die den Graphen bildet. Plexus ist somit verantwortlich für den Fluss der Daten zwischen den Devices und die Möglichkeit, einen Graph als XML Datei speichern zu können. Dabei werden die Typen der Daten nicht von Plexus vorgegeben. Alleine die Devices sind dafür verantwortlich, die Datentypen, für die sie Eingänge bereitstellen, verarbeiten zu können.

Der Editor (Enhanced Reality Editor Frontend) in Verbindung mit den vorhandenen Devices könnte als *Blackbox-Framework* angesehen werden, da durch einfaches Hinzufügen von Devices und Verbindungen zu einem Graph verschiedene Anwendungen realisiert werden können, wohingegen Plexus, ohne seine große Sammlung an Devices, als *Whitebox-Framework* eine begrenze Kernfunktionalität bietet und eine Reihe von Interfaces beschreibt.

Neben dem Editor gibt es im Augenblick noch eine sog. Standalone Anwendung, die im XML-Format gespeicherte Graphen laden und ablaufen lassen kann. Weiterhin wäre z. B. eine Anbindung von Plexus an AfterEffects als Plugin denkbar, bei der aus AfterEffects heraus XML Graphen geladen und als Filter verwendet werden können.

Das Plexus-Framework wird von Tobias Ritschel im Rahmen des Enhanced Reality Projekts[22] an der Universität Koblenz-Landau entwickelt. Es dient hauptsächlich zum schnellen Erstellen von Prototypen zur Demonstration von neuartigen Algorithmen und Verfahren im Rahmen von Virtual Reality und Augmented Reality. Das Plexus-Framework wurde ursprünglich auf der Microsoft Windows Plattform entwickelt, ist aber mittlerweile auch unter Linux lauffähig. Der Editor und die Standalone Anwendung waren zum Zeitpunkt dieser Arbeit noch nicht auf andere Plattformen portiert.

5.3 Erweiterungen an Plexus

Teil dieser Arbeit war es, das Plexus-Framework um Komponenten zu erweitern, die das Filtern von 2D- und 3D-Daten auf der CPU und auf der GPU ermöglichen.

[22] `http://er.uni-koblenz.de`

Die im Rahmen dieser Arbeit entwickelten Devices werden in 7.1 im Detail vorgestellt. Hier wird nun ein Überblick über das Vorgehen bei der Entwicklung der Devices gegeben.

Datentypen und Datenfluss

Zuerst wurden die vorhandenen Datenstrukturen analysiert. Es existierte bereits der `Image` Datentyp, der für die Repräsentation von Bildern auf der CPU vorgesehen war und in den `Texture2D` Datentyp konvertiert werden konnte, der für die Repräsentation und Speicherung von Bildern auf der GPU vorgesehen war. Dieser Datentyp hat 10 verschiedene Ausprägungen, die sich in der Zahl und Auflösung der Farbkanäle unterscheiden. Diese Ausprägungen werden in 5.3.1 genauer vorgestellt. Der `Texture2D` Datentyp wurde um die Möglichkeit erweitert, wieder in den `Image` Datentyp transferiert werden zu können. Für diesen standen Methoden bereit, um ihn mit Daten aus Bilddateien verschiedener Formate zu füllen. Die Funktion, den Inhalt eines `Image` Datentyps in eine Datei zu schreiben, wurde ergänzt. Diese Lese- und Schreibfunktionalität wird durch die *DevIL*[23]-Bibliothek zur Verfügung gestellt. Weiterhin wurde jeweils ein Device entwickelt, das eine Videodatei lesen und pro Aktualisierungsschritt des Graphen ein `Image` zur Verfügung stellen, bzw. ein `Image` konsumieren und in eine Videodatei schreiben kann (siehe 7.1.3). Dafür wurde die *ffmpeg*[24]-Bibliothek verwendet.

Der `Volume` Datentyp zur Repräsentation von Volumen auf der CPU und ein Funktion, um ihn mit Volumendaten aus RAW Dateien zu füllen, war vorhanden, die `Texture3D` Erweiterung für die Verarbeitung von Volumen auf der GPU wurde ergänzt. Da ein Video nicht komplett als Volumen in den Speicher der Graphikkarte geladen werden kann, wurde eine Möglichkeit zum kontinuierlichen, bildweisen Transfer auf die Graphikkarte implementiert. Dieses Verfahren wird in 5.3.2 beschrieben. Zusammenfassend wurden folgende Erweiterungen an vorhandener Funktionalität vorgenommen:

- Transfer von Bilddaten von der Graphikkarte in den Hauptspeicher
- Speichern eines `Image`-Objektes in eine Datei
- Repräsentation von Volumen in einer `Texture3D` auf der Graphikkarte

Nachdem die Datentypen und Funktionen zum Übertragen von Bilder aus Dateien in ein Volumen, auf die Graphikkarte und zurück in eine Datei vorhanden waren, wurde die Kontrolle des Datenflusses betrachtet. In Plexus können beliebige Datentypen fließen. Dazu gehören z. B. neben den

[23] `http://openil.sourceforge.net/`
[24] `http://ffmpeg.mplayerhq.hu/`

eben vorgestellten Typen für Bilder und Volumen noch folgende häufig verwendete Datentypen:

- **Scalar** Ein einfacher *float* Wert
- **Vec2** Ein Vektor aus zwei *float* Werten
- **Vec3** Ein Vektor aus drei *float* Werten
- **Matrix44** Eine Matrix aus 4×4 *float* Werten
- **String** Eine Zeichenfolge

Außerdem können sehr einfach weitere Typen für spezielle Anwendungen ergänzt werden wie z. B. *Mesh*, *SIFT Feature* oder *Directional Light Cloud*. Jedes Device besitzt eine beliebige Anzahl an Eingangs- und/oder Ausgangsparametern die einen dieser Datentypen akzeptieren. Diese Ein- und Ausgänge werden als *Slot* bezeichnet.

Das Plexus-Framework sieht vor, dass, sobald sich ein Device in einem definierten Zustand befindet, dieses Device seine Daten an das Nächste weitergibt. Befindet sich ein Device in einem undefinierten Zustand oder geht in diesen Zustand über, gibt das Device keine Daten an seine Nachfolger weiter und alle weiteren Devices werden ebenfalls undefiniert. Ein Device geht in den undefinierten Zustand über, weil z. B. ein Eingangsparameter ungültig wurde und somit ein Slot undefiniert ist. Die einzige Möglichkeit den Datenfluss zu steuern besteht also darin, ein Device in einen ungültigen Zustand zu bringen. Diese Form der Kontrolle reicht jedoch für die Filterung von Videos aus. Sobald das entsprechende Device eine Videodatei geöffnet hat, befindet es sich in einem definierten Zustand und stellt pro Aktualisierungsschritt des Graphen ein neues `Image` an seinem Ausgangsslot zur Bearbeitung bereit. Befindet sich kein weiteres Bild in dem Videostrom, geht das Device in einen undefinierten Zustand über und die Verarbeitung ist beendet, der Datenfluss versiegt.

CPU Filter

Als Nächstes wurden verschiedene Filter zur Ausführung auf der CPU entwickelt (siehe 7.1.1). Diese dienen zum Vergleich der GPU Implementationen der Filter hinsichtlich der Korrektheit des Ergebnisses und der Geschwindigkeit. Diese Filter waren für den `Image` Typ der Gauß-Filter, der separierte Bilateral Filter und der Rangordnungs-Filter und für den `Volume` Typ der Gauß-Filter und der separierte Bilateral Filter.

Verschiedene Tools

Im Laufe der Erweiterung des Plexus-Frameworks wurden verschiedene Devices entwickelt, um die Ergebnisse der Filter beurteilen zu können (siehe Abschnitt 7.1.1). Dies ist zum einen ein Device das zwei `Images` als Eingabe hat und ein `Image` ausgibt, das aus der linken Hälfte des ersten und der rechten Hälfte des zweiten `Image` besteht. So ist ein direkter Vergleich zwischen zwei Bildern möglich.

Des Weiteren wurde ein Device erstellt, dass ein `Image` gefüllt mit Zufallswerten erzeugt. Dieses ist nötig, um die Qualität der Filter anhand eines *Full-reference* Fehlermaßes vergleichen zu können. Dieser Vergleich kann mit den erstellten Devices, die das SSIM und MSE Fehlermaß nutzen, durchgeführt werden.

GPU Filter

Als Letztes wurden die GPU Varianten der Filter implementiert (siehe 7.1.2). Für den 2D-Fall wurde der Gauß-Filter und der separierte Bilateral Filter implementiert, für den 3D-Fall der separierte Bilateral Filter. Der letztgenannte Filter unterscheidet sich von der CPU Variante dahingehend, dass er eine `Texture3D` als Eingabe bekommt und eine `Texture2D` als Ausgabe produziert. Er erzeugt also nur eine Schicht aus dem Volumen. Dies liegt an der Einschränkung, nicht direkt in 3D-Texturen schreiben zu können. Jedoch ist dies für das Filtern von Videosequenzen nicht von Nachteil, sondern ergänzt sich mit dem Konzept des Streamings des Videos in das Volumen, welches in Abschnitt 5.3.2 vorgestellt wird.

5.3.1 Bilddatentypen

Es gibt verschiedene Möglichkeiten die Farbwerte von Bildern abzuspeichern. Diese unterscheiden sich in der Anzahl der Farbkanäle, und in der Auflösung jedes Kanals. So ist es bei der Mehrzahl an Bildern aus digitalen Kameras üblich, drei Kanäle mit den Farben Rot, Grün und Blau und einer Auflösung von jeweils 256 unterschiedlichen Werten (8 Bit) zu erhalten. Medizinische Bildgebungsverfahren produzieren aber andererseits Bilder mit nur einem Kanal, der in der Regel als Grauabstufung dargestellt wird, mit einer Auflösung von aktuell 4096 Werten (12 Bit) und 65536 Werten (16 Bit). HDR[25]-Kameras wiederum liefern die drei RGB Kanäle mit einer Auflösung von zur Zeit jeweils 16 Bit, wobei die Abbildung des eintreffenden Lichts auf die 65536 möglichen Werte unpraktischerweise nicht linear ist. Mit fortschreitender Entwicklung wird nicht nur die Ortsauflösung sondern auch die Auflösung der Farbkanäle weiter steigen. So werden schon heute synthetische HDR Bilder mit 32 Bit Auflösung verwendet.

[25]**H**igh-**D**ynamic-**R**ange

	Grau	RGB	RGBA
unsigned char (8 Bit, Integer)	✗	✗	✗
short (16 Bit, Integer)	✗		
half (16 Bit, Gleitkommazahl)	✗	✗	✗
float (32 Bit, Gleitkommazahl)	✗	✗	✗

Tabelle 1: Bilddatentypen in Plexus

Anstatt nun alle verschiedenen Eingabedaten in einem universellen Datentyp unterzubringen, bietet das Plexus-Framework an, für die unterschiedlichen Anforderungen unterschiedliche Datentypen zu verwenden. Diese unterscheiden sich in der Art die Farbwerte zu speichern, haben aber Parameter wie z. B. die Auflösung gemeinsam. Dadurch kann ein optimierter Speicherverbrauch erreicht werden. Außerdem können die Fähigkeiten der CPUs und GPUs optimal ausgenutzt werden. So benötigen beispielsweise algebraische Operationen auf der CPU mit float-Werten mehr Zeit als mit int-Werten. Außerdem können bei ganzzahligen Werten schnelle Shiftoperationen verwendet werden. Weiterhin stehen auf der GPU manche Operatoren nicht für alle Datentypen zur Verfügung. Das liegt einfach daran, dass die GPU für einige bestimmte Datentypen optimiert ist, wohingegen die CPU als universeller Prozessor alle Datentypen verarbeiten kann; manche jedoch mit geringerer Geschwindigkeit. Die 10 von dem Plexus-Framework zur Verfügung gestellten Bilddatentypen sind in Tabelle 1 angegeben.

Die OpenGL API gibt für die Verwendung auf der GPU ein ganze Reihe verschiedener Datentypen vor. Dabei wird zusätzlich unterschieden zwischen der Reihenfolge der Farbkanäle und speziellen Kanälen für Stencil- und Depth-Werte. Außerdem wird unterschieden zwischen der Repräsentation der Pixel im Hauptspeicher und der Darstellung auf der Graphikkarte. Des Weiteren gibt es verschiedene Varianten, bei denen die unterschiedlichen Farbkanäle unterschiedliche Auflösungen haben. Diese große Anzahl an Datentypen kommt durch die gewünschte, starke Kompatibilität bei fortwährender Entwicklung zustande. Im Plexus-Framework stehen Abbildungen für die 10 bereitgestellten Datentypen auf die entsprechenden Datentypen der Graphikkarte zur Verfügung. Dadurch ist der einfache Transfer von Bilddaten zwischen der CPU und der GPU gewährleistet. Die Vielzahl an Datentypen auf der GPU kann insofern ein Problem darstellen, als dass einige Graphikkartentreiber Konvertierungen beim Transfer der Bilddaten auf die GPU vornehmen müssen und es dadurch zu Geschwindigkeitseinbußen kommen kann.

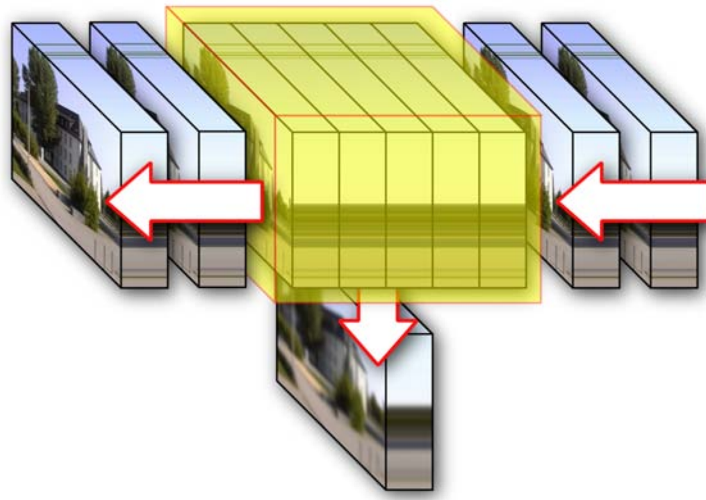

Abbildung 9: Durch das stationäre Filter-Fenster *streamen* die Videobilder. Pro Filterschritt werden in diesem Beispiel in das Ergebnis eines Bildes fünf Bilder des Videostroms einbezogen.

5.3.2 Video streaming

Das zu filternde Video soll auf der Graphikkarte als Volumen abgelegt werden, um beim Filtern zeitliche Zusammenhänge auszunutzen. Da aber Videosequenzen beliebiger Länge gefiltert werden sollen, können die Videosequenzen nicht komplett in den Speicher der Graphikkarte geladen werden. Obwohl aktuell bis zu 512 MB Speicher zur Verfügung steht, fänden darin nur ca. 200 Frames in PAL Auflösung oder nur ca. 40 Frames in HDTV1080 Platz, wenn man versuchen würde die komplette Videosequenz in den Speicher der Graphikkarte zu laden. Durch Limitierungen der GPU müsste das Volumen zweimal im Speicher liegen, einmal zum Lesen und einmal zum Schreiben. Damit ergibt sich für PAL:

$$(512\,MB/2)\,/\,(720 \cdot 576 \cdot 3\,Byte/Frame) \approx 200\,Frames$$

Das entspricht bei einer Bildwiederholrate von 25 Bilder/s eine Länge von 8 Sekunden, bei HDTV1080 nicht einmal 2 Sekunden.

Um Videos beliebiger Länge zu filtern, wird das Video auf die Graphikkarte *gestreamt* (siehe in Abb. 9) und als Ergebnis kein Volumen, sondern nur jeweils eine Schicht erzeugt. Währenddessen befinden sich immer nur so viele Videobilder im Graphikkartenspeicher wie das Volumen tief ist. Dabei sollte die Tiefe des Volumen mit $Tiefe = 2 \cdot R_D + 1$ so gewählt werden, dass sie den Tiefenradius R_D des Filters berücksichtigt.

Soll der Filter beispielsweise zum aktuellen Frame noch 2 Frames vor und hinter dem aktuellen Frame berücksichtigen (Filter-Tiefen-Radius = 2), so sollte das Volumen eine Tiefe von 5 haben. Der Filter bearbeitet nun in

einem Bearbeitungsschritt die mittlere Schicht in dem Volumen und bezieht die drei davor und dahinter liegenden Schichten in die Filterung mit ein. Das Ergebnis ist die gefilterte, mittlere Schicht. Nach dem Filtern werden alle Schichten im Volumen um eine Stelle nach vorne verschoben. Damit fällt die vorderste Schicht aus dem Volumen heraus und die hinterste, nun leere Schicht wird mit dem nächsten Frame aus dem Videostrom gefüllt. Das Filterfenster bleibt also stationär im Volumen und das Video „strömt" durch es hindurch. Durch dieses Vorgehen ist zugleich die Randbehandlung in der Tiefe vereinfacht, da nur am Anfang und Ende des Videos, wenn das Volumen noch leer ist oder leer wird, mit Wiederholungen des ersten bzw. letzten Frames das Volumen aufgefüllt werden muss.

So ist selbst für Video in HDTV1080 Auflösung bei einer Tiefe von 5 nur:

$$\underbrace{((1920 \cdot 1080 \cdot 3\, Byte/Frame) \cdot 5)}_{Eingabeschichten} + \underbrace{(1920 \cdot 1080 \cdot 3)}_{Ausgabeschicht} \approx 38\, MB$$

Speicher auf der Graphikkarte nötig, bzw ein maximaler Filterradius bei 512 MB Graphikkartenspeicher von 40 möglich.

6 Arbeiten mit der GPU

6.1 Verwenden der OpenGL Shading Language

Die OpenGL Shading Language ist seit OpenGL 2.0 [SA04] Bestandteil der OpenGL Spezifikation. Wie bei allen Bestandteilen von OpenGL sollte vor deren Verwendung geprüft werden, ob die nötige Funktionalität vorhanden ist. Dies geschieht mit dem Befehl:

```
glGetString(GL_EXTENSIONS);
```

Man erhält eine lange Zeichenkette mit allen verfügbaren Funktionalitäten. Um die OpenGL Shading Language verwenden zu können, müssen folgende Extensions vorhanden sein:

```
GL_ARB_Shader_Objects
GL_ARB_Vertex_Shader
GL_ARB_Fragment_Shader
GL_ARB_Shading_Language_100
```

Wenn diese Zeichenketten vorhanden sind, weiß der Graphikkartentreiber mit den entsprechenden OpenGL Befehlen etwas anzufangen. Das Vorhandensein bedeutet nicht zwangsweise, dass die Funktionalität durch die Hardware unterstützt wird. Den Herstellern der Treiber und Hardware ist es überlassen, ob die Funktionen durch die Hardware beschleunigt, oder ob sie durch eine Softwareemulation realisiert werden. Auf diese Tatsache wird in Abschnitt 6.4 eingegangen.

6.2 Allgemeine Bildfilter auf der GPU

Durch ihren Ursprung in der Computergraphik bietet die Graphikkarte einige Eigenschaften, die für die Entwicklung von Bildfiltern von Vorteil sind und die Implementierung vereinfachen. Ein Punkt der bei der Implementierung von Filtern berücksichtigt werden muss ist die *Randbehandlung*. Da ein Filter zur Berechnung des aktuellen Pixels die Pixel in der Nachbarschaft berücksichtigt, müssen diese Nachbarpixel vorhanden sein, oder für die Fälle der Pixel am Rand und in den Ecken müssen Regeln existieren, wie die nicht vorhandenen Nachbarpixel ersetzt werden. Implementierungen von Filtern für die CPU müssen die Randbehandlung explizit berücksichtigen, wenn sie einigermaßen optimiert sind und nicht auf sehr abstrakten Bildklassen, die die Randbehandlung übernehmen, arbeiten. Dagegen müssen für die Filter auf der GPU nur die entsprechenden Parameter für die Randbehandlung gesetzt werden und diese wird, durch die Hardware unterstützt, durchgeführt. Die beiden wichtigsten Arten der Randbehandlung der GPU sind:

- **GL_CLAMP** Für Zugriffe auf Bereiche, die außerhalb des Bildes liegen, wird der letzte, bzw. erste Farbwert der innerhalb des Bildes liegt, verwendet.
- **GL_REPEAT** Der Bereich außerhalb des Bildes verhält sich so, als ob das Bild um das eigentliche Bild herum noch 8 mal schachbrettartig liegen würde.

Diese Arten der Randbehandlung werden auch in CPU Implementierungen verwendet. In der Computergraphik können dadurch große Flächen, die ein sich wiederholendes Muster haben, durch eine kleine, speichersparende Textur mit einer Oberfläche versehen werden (GL_REPEAT), oder es kann gerade ein sich wiederholendes Muster vermieden werden (GL_CLAMP).

Einen weiteren Parameter den OpenGL für Textur-, bzw. Bildzugriffe anbietet, ist der des Texturfilters. Durch die perspektivische Verzerrung bei der Darstellung von 3D-Szenen, erfolgt der Texturzugriff nicht pixelgenau mit ganzzahligen Pixelpositionen, sondern in einem auf den Bereich $0 \dots 1$ normierten Texturkoordinatensystem. Deshalb wird beim Texturzugriff in der Standardeinstellung der Pixelwert aus der Umgebung interpoliert. Die Interpolationsmethode wird dabei Texturfilter genannt. Für die Darstellung von 3D-Szenen gibt es eine ganze Reihe unterschiedlicher Interpolationsmethoden, die verschiedene Probleme beheben und den Berechnungsaufwand minimieren. Bei der Verwendung der GPU für Bildfilter ist in der Regel ein pixelgenauer Zugriff gegeben und eine Interpolation nicht nötig. Die beiden wichtigen Texturfilter sind in diesem Zusammenhang:

- **GL_NEAREST** Es wird der zur angegebenen Position in Manhattandistanz[26] nächstgelegene Pixel verwendet.
- **GL_LINEAR** Es wird der gewichtete Mittelwert der vier Pixel verwendet, die der angegebenen Position am nächsten liegen.

Der GL_LINEAR Texturfilter ist insofern zu beachten, als dass er in Verbindung mit der Randbehandlung GL_CLAMP bei OpenGL unter Umständen nicht das gewünschte Ergebnis liefert. Wird beim Zugriff auf ein Pixel in einer Textur nicht die Position der Mitte des Pixels, sondern der Rand angegeben, und liegt das Pixel am Rand der Textur, wird im GL_LINEAR Modus ein Mittelwert zwischen dem Randpixelwert und einer Rahmenfarbe[27], die der Textur zugewiesen werden kann, zurückgegeben. Damit

[26] Die Manhattandistanz beschreibt auf einem schachbrettartigen Muster diejenige Distanz, die sich von einem Feld zu einem anderen Feld nur durch waagerechte und senkrechte Bewegungen ergibt.

[27] Der Rahmen, den man für Texturen definieren kann, ist beispielsweise im Zusammenhang mit Texturen, die wegen ihrer Größe gekachelt werden müssen, nützlich. Dann kann der Rahmen einer Textur die Pixel der benachbarten Textur enthalten und somit ist eine Texturfilterung auch an den Nähten zwischen Texturen möglich, ohne dass zur Interpolation an den Nähten Werte aus zwei verschiedenen Texturen gelesen werden müssen.

das eigentlich gewünschte Verhalten eintritt, nämlich dass keine Randfarbe, sondern nur Pixel innerhalb der Textur verwendet werden, muss der Randbehandlungsmodus GL_CLAMP_TO_EDGE verwendet werden.

6.3 Volumenfilter auf der GPU

Volumenfilter könnten prinzipiell mit den gleichen Eigenschaften wie Bildfilter auf der GPU implementiert werden. Die OpenGL Spezifikationen schränken jedoch eine direkte Umsetzung ein. So ist es nicht möglich in ein FBO zu rendern, das eine 3D-Textur repräsentiert, da dass 3D-Rasterisieren in einen 3D-Framebuffer nicht definiert ist. Die OpenGL Pipeline ist darauf ausgelegt eine 3D-Szene in ein Bild zur rasterisieren, nicht in ein Volumen, da dafür einige Konzepte wie z. B. das Clipping geändert werden müssten.

Um dennoch in einen 3D-Framebuffer rendern zu können, gibt die FBO-Extension vor, das beim Anfügen der 3D-Textur an das FBO mit dem Befehlt `glFramebufferTexture3DEXT()` zusätzlich die Tiefenebene angeben werden muss, in die gerendert werden soll. Obwohl viele aktuelle Graphikkarten die `GL_EXT_framebuffer_object` Extension unterstützen, wird der Befehl zum Setzen der 3D-Textur bei allen verfügbaren Graphikkarten zur Zeit mit einem Fehler quittiert. Die einzige Möglichkeit, die zur Zeit genutzt werden kann, um in ein 3D-FBO zu renden, besteht darin, in ein 2D FBO zur renden und die 2D-Textur danach mit dem `glCopyTexSubImage3D()` Befehl in das 3D FBO zu kopieren.

6.4 ATI vs. NVIDIA

Die beiden großen Graphikkartenhersteller ATI und NVIDIA bringen in sehr kurzen Zeitabständen immer neue Generationen an Graphikkarten auf den Markt, mit dem Werbeversprechen, immer die aktuellsten und zukünftigen Graphik APIs zu unterstützen. Allerdings muss man dabei sehr genau hinschauen, welches Features von welcher Graphikkarte denn nun wirklich unterstützt wird. So gibt es einerseits große Unterschiede in der Hardwareunterstützung verschiedener Features bei den beiden verbreitetsten Graphik-APIs OpenGL und Direct3D, und andererseits auf den verschiedenen Plattformen Windows, Linux und Mac. Außerdem kommt es nicht selten vor, dass bestimmte Features nicht sorgfältig in den Graphikkartentreibern implementiert sind, und somit nicht immer das gewünschte Ergebnis eintritt. Während der Implementierung bei diese Arbeit von verschiedener Funktionalität für die GPU, die nur unter Windows stattfand, sind insbesondere Unterschiede zwischen den Graphikkarten der Firmen ATI und NVIDIA aufgetreten, die zu Testzwecken zur Verfügung standen. Diese Unterschiede werden nun im Folgenden dokumentiert:

Unterstützung von Extensions

Die einfache Erweiterungsmöglichkeit von OpenGL durch Extensions hat dazu geführt, dass von verschiedenen Firmen Erweiterungen definiert wurden, die speziell für die Graphikkarten dieser Firmen geeignet waren. Um der Konkurrenz nun keinen Wettbewerbsvorteil zu gewähren, haben die konkurrierenden Firmen ebenfalls versucht, diese Extensions zu unterstützen. Allerdings sieht der OpenGL Standard keinen Zwang zur Unterstützung von fremden und sogar eigene, vom OpenGL ARB[28] standardisierte Erweiterungen, vor. Die ARB-Extensions werden in der Regel in der nächsten OpenGL Version in den zentralen Bestandteil von OpenGL aufgenommen, und müssen dann von der API bereitgestellt, aber nicht unbedingt durch die Hardware beschleunigt werden, wenn der Treiber vorgibt eine bestimmte OpenGL Version zu unterstützen. Aus diesem Grund unterstützen nicht alle Graphikkarten[29] alle existierenden[30] OpenGL Extensions.

So steht z. B. die `GL_ARB_texture_non_power_of_two` Extension auf keiner aktuellen ATI Graphikkarte (Stand 09/2006) zur Verfügung, wohingegen NVIDIA diese Erweiterung seit April 2004 mit der Einführung der GeForce 6 Generation unterstützt (siehe auch Abschnitt 7.2).

OpenGL Shading Language Compiler

Da beide Graphikkartenhersteller den OpenGL 2.0 Standard unterstützen, liefern beide Hersteller in ihren Treibern Compiler für die GLSL. Während der Compiler von NVIDIA auch Syntax-Konstrukte akzeptiert, die nicht dem Standard entsprechen, aber eigentlich von der Intention klar sind, hält sich der ATI Compiler exakt an den Standard. Für eine größtmögliche Kompatibilität der Shaderprogramme ist die exakte Einhaltung der Spezifikation unerlässlich.

Beide Compiler besitzen große Defizite bei der Umsetzung aktueller Sprachkonstrukte. Obwohl aktuelle GPUs in der Hardware Schleifen in den Programmen unterstützen, generiert der NVIDIA Compiler Assemblercode, in dem die Schleifen *unrolled* sind, also durch wiederholtes Hintereinanderkopieren des Schleifeninhaltes ersetzt sind. Da in dem implementierten Bilateral Filter Schleifen über die Pixel in der Nachbarschaft vorkommen, ist die Nachbarschaftsgröße durch die maximale Anzahl an Instruktionen, die ein kompilierter Fragment-Shader haben darf, begrenzt.

Die Situation ist im Fall des Bilateral Filters auf ATI Graphikkarten noch ungünstiger. Der GLSL Compiler in den ATI Treibern ist nicht in der Lage,

[28] **A**rchitecture **R**eview **B**oard

[29] Eine Übersicht darüber, welche Graphikkarte welche Extensions unterstützt, und welche OpenGL Beschränkungen vorgegeben werden gibt es unter: `http://www.delphi3d.net/hardware/`

[30] Eine Liste mit allen definierten Extensions wird von der Firma sgi gepflegt: `http://oss.sgi.com/projects/ogl-sample/registry/`

Arrayzugriffe mit Variablen zu verarbeiten. Der folgende Programmcode kann nicht auf der GPU ausgeführt werden, und es wird eine äußerst langsame Softwareemulation verwendet:

```
int array[5];
int i = 0;
array[i] = 3;
```

Der folgende Programmcode kann von dem ATI Compiler so übersetzt werden, dass er durch die GPU ausgeführt werden kann:

```
int array[5];
array[0] = 3;
```

Durch diese Einschränkung konnte der Bilateral Filter mit vertretbarem Aufwand nur für einen einzelnen, festen Filterradius von zum Beispiel 3 implementiert werden. Da Bildfilter häufig auf einer variablen Nachbarschaft arbeiten, eigenen sich ATI-Graphikkarten nur sehr bedingt für die Implementierung von Bildfiltern oder ähnlichen Algorithmen mit GLSL. Da die aktuellen GPUs von NVIDIA und ATI alle Sprachkonstrukte des Shader Modell 3, auf dem GLSL basiert, beherrschen, ist es sehr schade, dass diese Funktionalität nicht in allen Shader Programmiersprachen zur Verfügung steht. Beide Firmen haben die erwähnten Defizite wie *unrolling* und statischen Arrayzugriff in anderen Shadersprachen nicht.

Vorgegebene Beschränkungen

Über vorgegebene OpenGL-Konstanten können Beschränkungen für verschiedenste Parameter der API-Befehle angegeben werden. Dazu gehören z. B. GL_MAX_TEXTURE_SIZE und GL_MAX_3D_TEXTURE_SIZE. Während bei NVIDIA Graphikkarten 2D-Texturen eine maximale Größe von 4096×4096 Pixel haben dürfen, sind die 3D-Texturen auf $512 \times 512 \times 512$ Pixel beschränkt. Dies schränkt die Volumenfilterung von Videosequenzen auf Auflösungen kleiner als PAL erheblich ein. ATI Graphikkarten hingegen erlauben 3D-Texturen bis $2048 \times 2048 \times 2048$ Pixel, jedoch sind 2D-Texturen auf 2048×2048 Pixel beschränkt. Diese Grenzen lassen die Volumenfilterung von Videosequenzen bis zur höchsten HDTV1080 Auflösung zu. Geschwindigkeitsmessungen zeigen jedoch, dass die höchstmöglichen Auflösungen nicht die zu erwartenden Verarbeitungsgeschwindigkeiten erreichen (siehe Abschnitt 8.4.2).

7 Implementierungsdetails

Im Rahmen dieser Arbeit wurde das Plexus-Framework um Komponenten erweitert, die es zu einem Framework zur Filterung von Videosequenzen als Volumen auf der GPU macht. In Abschnitt 5.3 wurde das allgemeine Vorgehen besprochen, hier folgen nun einige konkrete Details zu den entwickelten Komponenten, den Devices. Einige Devices dienen nicht zum konkreten Messen von Geschwindigkeit oder Qualität im Rahmen dieser Arbeit, sondern dienen zukünftigen Entwicklern als Einstiegsbeispiele und zum Vertrautwerden mit dem Framework.

7.1 Entwickelte Devices

Es wurde eine Reihe von Devices entwickelt, die zur Filterung von Bildern und Volumen dienen, und Devices die eine Bewertung der Ergebnisse ermöglichen oder erleichtern.

7.1.1 Devices für die CPU

Die folgenden Devices verwenden für alle Berechnungen die CPU und arbeiten entsprechend auf den Plexus-Datentypen `Image` und `Volume`. Sie wurden stets als Vorläufer, zum Vergleich und zur Kontrolle des Ergebnisses für die GPU Variante entwickelt.

SeparableBilateralSmoothingImageDevice Die separierte Variante des Bilateral Filters für Bilder wird von diesem Device zur Verfügung gestellt. Es arbeitet auf Bildern des `RGBAUnsignedChar` Typs. Als Parameter können der Maskenradius (Mask Radius HW), der Sigma-Wert für die Gaußgewichtung der Position (Sigma D) und der Sigma-Wert für die Gaußgewichtung des Pixelwerts (Sigma R) beeinflusst werden. Für die genaue Bedeutung der Parameter siehe Abschnitt 2.1.1.

RankOrderImageDevice Das Device stellt ein Rangordnungsfilter für Bilder des `fRGBAUnsignedChar` Typs zur Verfügung. Als Parameter kann der Maskenradius (Mask Radius) und der Rang zwischen 0 und 1 (Rank) angegeben werden. Ein Rang von 0 entspricht dem Minimum-Filter, der Rang 1 dem Maximum-Filter und ein Rang von 0,5 dem Median-Filter.

VerticalCombineImageDevice Dieses Device liefert ein Bild, das die Kombination zweier `fRGBAUnsignedChar` Bilder darstellt. Dabei wird von Bild A die linke Hälfte und von Bild B die rechte Hälfte zu einem neuen Bild kombiniert. Dies kann zum direkten visuellen Vergleich eines gefilterten und ungefilterten Bildes verwenden werden, wie in Abb. 10 dargestellt.

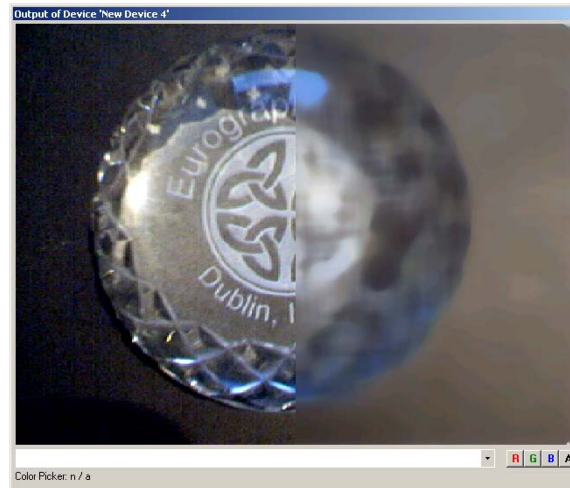

Abbildung 10: Vertikale Kombination eines ungefilterten (links) und eines mit dem RankOrderImageDevice (Median, rechts) gefilterten Webcam-Bildes.

SimpleGaussSmoothingVolumeDevice Eine einfache, unoptimierte, jedoch separierte Variante des Gaußschen Weichzeichnungsfilters für Volumen wird von diesem Device bereitgestellt. Es verarbeitet Volumen vom `fRGBAUnsignedChar` Typ und gibt ein weichgezeichnetes Volumen weiter. Als Parameter kann man den Filterradius in der Ebene und in der Tiefe separat angeben.

SeparableBilateralSmoothingVolumeDevice Dieses Device ist die Erweiterung des SeparableBilateralSmoothingImageDevice für Bilder auf Volumen. Ergänzend besitzt es als Parameter den Maskenradius für die Tiefe (Mask Radius D). Die beiden Sigma-Werte werden sowohl für die Filterung in der Ebene als auch in der Tiefe verwendet. Ebenso wie das Gauß-Volumen-Device verarbeitet dieses Device Volumen vom `fRGBAUnsignedChar` Typ und gibt ein entrauschtes Volumen weiter.

Texture2DToImageDevice Zum anschließenden Abspeichern von Videosequenzen oder auch von Bildern transferiert dieses Device eine `Texture2D` von der Graphikkarte zurück in ein `Image` in den Hauptspeicher. Es besitzt keine Parameter. Zusammen mit dem ImageToTexture2DDevice stehen alle Möglichkeiten zur Verfügung, um Bilddaten beliebig zwischen der CPU und der GPU transferieren zu können.

StreamingImageToVolumeDevice Dieses Device ist für das in Abschnitt 5.3.2 beschriebene Verfahren zum Transferieren der Bildsequenz eines Videos in ein Volumen zuständig. Das erstellte Volumen hat in Höhe und Breite die gleiche Abmessung wie das eintreffende Bild, die Tiefe

des Volumens wird über einen Parameter gesteuert. Das Device kopiert in jedem Schritt alle Ebenen im Volumen eine Position nach vorne und kopiert das eintreffende Bild an die hinterste Stelle im Volumen.

SSIMErrorImageScalarDevice Zum Bewerten der Qualität der Filterergebnisse ermittelt dieses Device einen Wert, einen Skalar, für den wahrgenommenen Unterschied zweier Bilder nach der in 2.4 erwähnten Methode.

SimpleNoiseImageDevice Für die Ermittlung eines Fehlermaßes nach der *Full-reference* Methode stellt dieses Device Bilder mit Rauschen bereit. Jedes Pixel im Bild erhält einen zufälligen Wert in einem durch die Parameter Min und Max festlegbaren Bereich. Die Größe des zu erstellenden Bildes ist ebenfalls durch einen Parameter einstellbar. In jedem Aktualisierungsdurchlauf des Graphen wird ein neues Rauschbild im Format fRGBAUnsignedChar erstellt, unabhängig davon, ob sich die Eingabeparameter geändert haben. Dieses Bild kann dann durch ein weiteres Device mit einem unverrauschten Bild kombiniert werden.

7.1.2 Devices für die GPU

Nach den CPU Varianten wurde von einigen Filtern GPU Versionen entwickelt. Diese arbeiten auf den Plexus-Datentypen `Texture2D` und `Texture3D`. Sie sind alle als reine Fragment-Shader-Programme realisiert.

SimpleGaussSmoothingLayerDevice Dieses Device implementiert den Gaußschen Weichzeichnungsfilter für 2D-Texturen. Es stellt eine einfache, unoptimierte und nicht separierte Variante zur Verfügung und war der erste implementiert GPU Filter. Für jedes zu filternde Pixel wird auf der GPU zuerst die Filtermaske mit den Gaußgewichten berechnet und dann die Pixel in der Nachbarschaft mit diesen gewichtet aufsummiert. Dieses Vorgehen wurde für verschiedene Tests während der Entwicklung gewählt. Alle weiteren Filter bekommen diese oder ähnliche Masken mit vorberechneten Gewichten als Uniform-Parameter[31] übergeben.

SeparableBilateralSmoothingLayerDevice Der GPU Pendant zum SeparableBilateralSmoothingImageDevice wird durch dieses Device realisiert. Er besitzt die gleichen Parameter wie die CPU-Version. Die Implementierung von diesem Device und seinem Gegenstück für die 3D-Textur besitzt eine Besonderheit: Die 1D-Gaußmaske für die Gewichtung der Pixelwerte wird an das Fragment-Programm nicht als

[31] Uniform-Parameter sind für die Übergabe von Variablen an das Shader-Programm geeignet, die sich nicht pro Vertex oder pro Fragment, sondern beispielsweise pro Renderdurchgang ändern.

Array von float-Werten übergeben, sondern als 1D-Textur dem Shader-Programm zur Verfügung gestellt. Dies liegt an der Beschränkung, dass alle Arrayzugriffe, im Gegensatz zu Texturzugriffen, zur Kompilierzeit bekannt sein müssen. Im Fall des Bilateral Filters wird der Arrayzugriff durch den Pixelwert erst während der Laufzeit festgelegt. Diese Beschränkung wird durch die Compiler, nicht durch die Spezifikation von GLSL vorgegeben.

SeparableBilateralStreamingSmoothingVolumeLayerDevice Im Kern stellt dieses Device die Funktionalität zur Verfügung, die durch diese Arbeit untersucht werden soll. Es stellt die 3D-Textur-Variante des SeparableBilateralSmoothingVolumeDevice dar. In Ergänzung zu dessen Parametern kann bei der GPU Version der Sigma D und Sigma R Wert zusätzlich für die Tiefe, getrennt von denen für die Ebene, angegeben werden. Dies hatte sich durch die Art der Implementierung angeboten. Im Gegensatz zu der CPU Version verarbeitet diese Device eine 3D-Textur und gibt als Ergebnis eine 2D-Textur aus. Dieses Vorgehen wurde für ein optimales Zusammenspiel mit der Video streaming-Technik (siehe Abschnitt 5.3.2) und zur Umgehung des Problems des Schreiben in ein Volumen auf der GPU (siehe Abschnitt 6.3) gewählt.

7.1.3 Im-/Export Devices

Für die Erweiterung des Plexus-Framework zu einem Framework für die Volumenfilterung von Videosequenzen war die Erstellung einiger Devices zum Lesen und Schreiben von Video- und Bildsequenzen nötig.

ImageSinkDevice Zum Schreiben von Bilddateien benötigt diese Device als Parameter einen Dateinamen und das zu schreibende Bild. Es stellt keine Ausgaben zur weiteren Verwendung im Graph bereit. Die eigentliche Schreibfunktionalität wird von der DevIL-Bibliothek zur Verfügung gestellt. Das Format der Bilddatei wird durch die Erweiterung des angegebenen Dateinamens bestimmt.

FFmpegImageDevice Dieses Device kapselt die Lesefunktionen der ffmpeg Bibliothek. Es nimmt als Eingabeparameter einen Dateinamen und stellt ein Bild bereit. In jedem Aktualisierungsschritt des Graphen wird das nächste Bild aus dem Videostrom decodiert und bereitgestellt. Ist das Ende des Video erreicht, wechselt das Device in einen undefinierten Zustand und stoppt somit die weitere Verarbeitung im Graphen.

FFmpegSinkDevice Analog zum ImageSinkDevice nimmt dieses Device einen Dateinamen und ein Bild, und schreibt dieses in einen Videostrom einer Videodatei. Die erzeugte Videodatei ist eine unkomprimierte AVI-Datei. Sobald Daten an den beiden Eingängen des Devices

anliegen, wird die Videodatei geöffnet und die ankommenden Bilder in der Videodatei abgelegt. Wird einer der Eingänge ungültig, wird die Videodatei mit nötigen Headerdaten versehen und geschlossen.

7.2 Portabilität

Einige in Abschnitt 6.4 beschriebene Beschränkungen verschiedener Graphikkarten führten zu Implementierungsvarianten, auch Code-Pfade genannt, die die Portabilität, insbesondere zwischen den Graphikkarten der Hersteller ATI und NVIDIA, sicherstellen. Die wichtigste Variation betrifft die fehlende `GL_ARB_texture_non_power_of_two` Erweiterung bei ATI Graphikkarten. Da keine Videonorm Auflösungen, die einer Zweierpotenz entsprechen, definiert, werden im Fall einer vorhandenen ATI Graphikkarte, alle Bilder, die in den Graphikkartenspeicher transferiert werden sollen, auf die nächst höhere Auflösung, die einer Zweierpotenz entspricht, vergrößert und die ursprüngliche Größe des Bildes gespeichert. Dadurch kann das Bild beim Transfer von der GPU auf die CPU wieder auf die entsprechende Auflösung verkleinert werden.

Teil III
Ergebnisse und Bewertung

8 Messaufbau

8.1 Testvideos

Für die verschiedenen Vergleiche wurden vier Videoaufnahmen erstellt, die typische Szenen aus dem Heimvideobereich zeigen. Der Schwerpunkt bei den Videos liegt auf dunklen Szenen, da die Rauschentfernung bewertet werden soll. Es wurden jedoch auch unverrauschte Szenen untersucht, da diese möglichst unverfälscht wiedergegeben werden sollen. Alle Aufnahmen wurden mit dem Ein-Chip Camcorder NV-DS35 von Panasonic erstellt. Alle angegebenen Videos befinden sich in originaler und gefilterter Version auf der beiliegenden DVD.

uni_hell 01.avi Aufnahmen auf dem Universitätsgelände bei Sonnenschein. Diese Bilder sind weitestgehend rauschfrei. Sie enthalten neben homogenen Flächen auch feine, bewegte Strukturen (Blätter der Bäume), die unverändert bleiben sollten.

kerze 18db.avi Diese Aufnahme zeigt ein Objekt mit feinen Strukturen (Laserkristall), das nur durch ein Teelicht beleuchtet wird. Bei dieser Aufnahme wurde der größtmöglichste Gain Wert von 18 dB verwendet.

ndi 04.avi Aufnahmen während einer Labordemo im Mixed Reality Labor bei der *Nacht der Informatik* an der Universität. Bei dieser Aufnahme wurde ebenfalls ein Gain Wert von 18 dB verwendet.

Zusammenschnitt.avi Eine Videosequenz mit den drei zuvor genannten Aufnahmen und zusätzlich Aufnahmen u. a. mit einem Gain Wert von 9 dB und Text vor verrauschter schwarzer Fläche.

Alle Videos wurden mit VirtualDub mit dem Filter „deinterlace“ mit dem Parameter „Blend fields together“ in einen progressiven Videostrom umgewandelt und als unkomprimiertes Video abgespeichert. Für den Geschwindigkeitsvergleich zwischen dem „3D Bilateral Filter“ und dem „Körnung entfernen“ Filter in AfterEffects wurde eine auf 512×512 Pixel beschnittene Version des `Zusammenschnitt.avi` Videos erstellt.

8.2 Hardware

Für die Messungen wurden unterschiedliche Computer mit unterschiedlicher Hardwareausstattung verwendet. Insbesondere unterscheiden sich

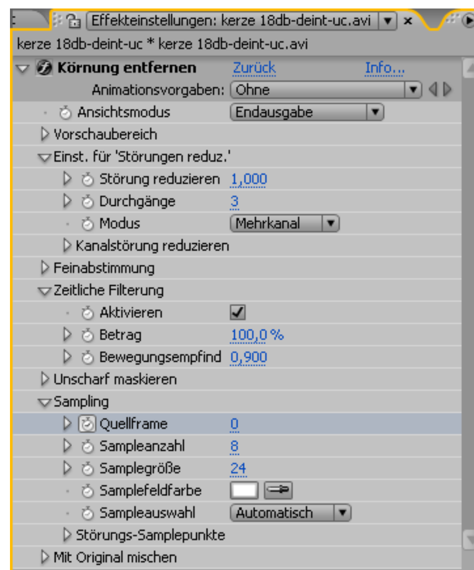

Abbildung 11: AfterEffects: Einstellungen des Filters „Körnung entfernen“

die Generationen der GPU, Graphikkartenschnittstellen (AGP und PCIe) und Hersteller. Um die Tabellen mit den Messergebnissen übersichtlich zu halten, soll hier eine Übersicht über die verwendeten Computer und deren Hardwareausstattung gegeben werden[32]:

- **“Chap“** AMD Athlon 64 3200+, 2 GB Ram,
  ATI Radeon X1600 Pro 256 MB, AGP
- **“Chip“** AMD Athlon 64 3200+, 2 GB Ram,
  NVIDIA GeForce 6600 GT 256 MB, AGP
- **“Bruce“** AMD Sempron 3400+, 2 GB Ram,
  NVIDIA GeForce 6800 XT 512 MB, PCI-Express
- **“Sid“** 2x Intel Xeon 3 GHz, 2 GB Ram,
  NVIDIA GeForce 6800 GT 256 MB, AGP
- **“Bob“** AMD Athlon 64 X2 4400+, 2 GB Ram,
  NVIDIA GeForce 7800 GTX 256 MB, PCI-Express

Alle Messungen wurden unter dem Microsoft Windows XP Betriebssystem mit aktuellen Graphikkartentreibern durchgeführt.

[32]Die Rechner „Bruce“, „Sid“ und „Bob“ wurden von der AG Computergraphik bereitgestellt, und sind mit deren Netzwerknamen aufgeführt.

8.3 Software

Plexus

Mit dem Plexus Editor wurden für alle Messungen entsprechende Graphen erstellt. Für den „2D Bilateral Filter“ wurden folgende Parameter gewählt:

Mask Radius HW: 3	Sigma D: 50	Sigma R: 3

Der „3D Bilateral Filter“ wurde mit folgenden Parametern verwendet:

Mask Radius HW: 3	Sigma D HW:3	Sigma R HW: 3
Mask Radius D: 3	Sigma D D: 5	Sigma R D: 3

Bei beiden Filtern hängt die Geschwindigkeit nur von den Radius Parametern ab, da die Sigma-Parameter nur die Werte in vorberechneten Tabellen beeinflussen.

AfterEffects

Für den Qualitätsvergleich wurde das `uni_hell 01.avi` Video mit dem AfterEffects Filter „Körnung entfernen“ gefiltert. Für die Filterparameter wurden die in Abb. 11 dargestellten Standardwerte verwendet, mit der Ausnahme, dass das ganze Bild gefiltert wird und nicht nur ein kleiner Vorschaubereich innerhalb des Bildes und der Filter wurde so eingestellt, dass er automatisch in jedem Bild die Störungsinformationen analysiert.

Neat Video

Das Neat Video Plugin wurde aus AfterEffects heraus für den Qualitätsvergleich verwendet. Es wurden ebenfalls alle Standardeinstellungen beibehalten. Ein Störungsprofil wurde automatisch im ersten Frame erstellt, welches zur Filterung von allen weiteren Frames verwendet wurde. Im Gegensatz zu dem AfterEffects Filter „Körnung entfernen“ konnte die Störungsinformation nicht in jedem Bild automatisch erfasst werden, da der Filter in einigen Frames diese Aktion mit einer Fehlermeldung abbrach.

8.4 Messergebnisse

8.4.1 Qualitätsmessungen

Für den objektiven Qualitätsvergleich wurden die beiden in Abschnitt 2.4 vorgestellten Fehlermaße SSIM und MSE verwendet. Es wurde nach der *Full-reference* Methode der Abstand zwischen zwei Videos bestimmt. Das eine Video war das unverrauschte Referenzvideo `uni_hell 01.avi`. Für das zweite Video wurde mit dem SimpleNoiseImageDevice und dem AdditionBlendImageDevice eine verrauschte Version des Referenzvideos erstellt. Dieses verrauschte Video wurde dann mit den entsprechenden Filtern bearbeitet und der Fehler zwischen dem gefilterten Video und dem Referenzvideo wurde bestimmt. Dazu wurde der Fehler zwischen jedem Bild

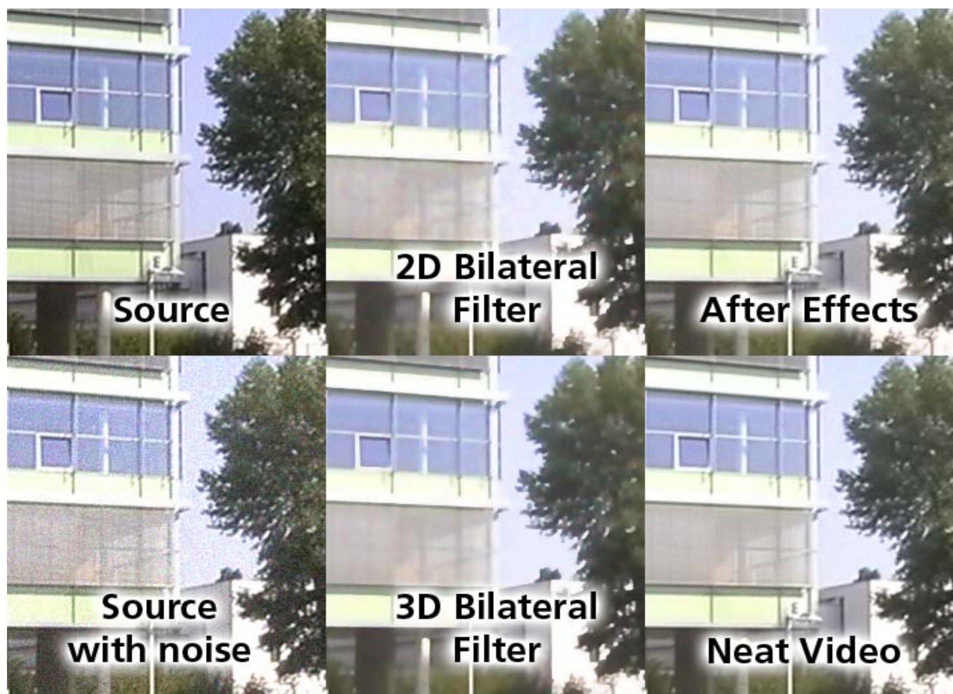

Abbildung 12: Beispiel aus der Qualitätsmessung

	SSIM	MSE
ungefiltert	0,707	508,5
gefiltert mit „2D Bilateral Filter“	0,898	412,6
gefiltert mit „3D Bilateral Filter“	0,899	410,6
gefiltert mit AfterEffects „Körnung entfernen“	0,893	414,1
gefiltert mit Neat Video	0,913	387,0

Tabelle 2: Qualitätsvergleich

im Video ermittelt und über alle Fehlerwerte wurde als Ergebniswert das arithmetische Mittel gebildet. Die Ergebnisse sind in Tabelle 2 zu finden. Abb. 12 zeigt beispielhaft sechs entsprechenden Ausschnitte . Beim SSIM Fehlermaß steht der Wert 1 für exakte Übereinstimmung, der Wert 0 für größtmögliche Unähnlichkeit. Der MSE-Wert 0 steht für exakte Übereinstimmung, die obere Schranke wird durch den Wertebereich eines Farbkanals vorgegeben (Bsp.: $(256 - 0)^2 = 65536$).

8.4.2 Geschwindigkeitsmessungen

Bei den Geschwindigkeitsmessungen der entwickelten Filter wurden keine Videos gefiltert, sondern es wurde das SimpleNoiseImageDevice verwendet, um für jeden Aktualisierungsschritt im Graph eine neues Bild zu erstellen;

Auflösung	Chip	Chap	Bruce	Sid	Bob
256×256	72 ms	72 ms	72 ms	63 ms	66 ms
512×512	297 ms	297 ms	293 ms	258 ms	266 ms
1024×1024	1203 ms	1203 ms	1183 ms	1050 ms	1070 ms
2048×2048	4845 ms	4845 ms	4763 ms	4263 ms	4310 ms
4096×4096	-	-	20321 ms	16239 ms	18156 ms

Tabelle 3: Filterung mit dem „2D Bilateral Filter“ auf der CPU

Auflösung	Chip	Chap	Bruce	Sid	Bob
256×256	6 ms	5 ms	5 ms	3 ms	2 ms
512×512	13 ms	17 ms	17 ms	11 ms	5 ms
1024×1024	50 ms	67 ms	63 ms	41 ms	17 ms
2048×2048	195 ms	266 ms	250 ms	162 ms	68 ms
4096×4096	-	-	-	34306 ms	760 ms

Tabelle 4: Filterung mit dem „2D Bilateral Filter“ auf der GPU

das gefilterte Bild wurde dann angezeigt. Dadurch wurde sichergestellt, dass die Messergebnisse nicht durch Lese- und Schreiboperationen auf Videodateien verfälscht werden. Die angegebenen Zeiten geben nur die Verarbeitungsdauer des jeweiligen Filter-Devices an. Alle angegebenen Bildgrößen haben Kantenlängen, die einer Zweierpotenz entsprechen. Dadurch wird ein fairer Vergleich zwischen Graphikkarten der Hersteller ATI und NVIDIA möglich

In Tabelle 3 sind die Zeiten für die Filterung mit dem „2D Bilateral Filter“ auf der CPU für verschiedene Auflösungen angegeben. Der Rechner „Chap“ kann das Ergebnis wegen der ATI Graphikkarte nicht darstellen, der Rechner „Chip“ bricht die Darstellung mit einer Fehlermeldung ab, obwohl die Darstellung möglich sein sollte. Tabelle 4 enthält die Zeiten für GPU Implementierung des „2D Bilateral Filter“. Die höchste Auflösung kann von der ATI Graphikkarte nicht erreicht werden (siehe Abschnitte 6.4), die NVIDIA Karten in den Rechnern „Chip“ und „Bruce“ lieferten kein brauchbares Ergebnis.

Die Tabelle 5 enthält die Zeiten für die Filterung mit dem „3D Bilateral

Auflösung	Chip	Chap	Bruce	Sid	Bob
$256 \times 256 \times 8$	240 ms	240 ms	239 ms	201 ms	214 ms
$512 \times 512 \times 8$	990 ms	990 ms	973 ms	921 ms	878 ms
$1024 \times 1024 \times 8$	3996 ms	3996 ms	3947 ms	3769 ms	3555 ms
$2048 \times 2048 \times 8$	16088 ms	16088 ms	15918 ms	15402 ms	14308 ms

Tabelle 5: Filterung mit dem „3D Bilateral Filter“ auf der CPU

Auflösung	Chip	Chap	Bruce	Sid	Bob
$256 \times 256 \times 8$	112 ms	70 ms	116 ms	66 ms	36 ms
$512 \times 512 \times 8$	345 ms	160 ms	305 ms	179 ms	108 ms
$1024 \times 1024 \times 8$	-	627 ms	-	-	-
$2048 \times 2048 \times 8$	-	38800 ms	-	-	-

Tabelle 6: Filterung mit dem „3D Bilateral Filter“ auf der GPU

	Bob	Chip
„2D Bilateral Filter“	52 ms	85 ms
„3D Bilateral Filter“	169 ms	489 ms
AfterEffects „Körnung entfernen“	2360 ms	3874 ms
Neat Video	466 ms	716 ms

Tabelle 7: Geschwindigkeitsvergleich

Filter“ auf der CPU und in Tabelle 6 sind die entsprechenden Zeiten für den „3D Bilateral Filter“ auf der GPU angegeben. Die Auflösungen ≥ 1024 konnten nur von der ATI Graphikkarte bearbeitet werden.

Für den Geschwindigkeitsvergleich der Gesamtverarbeitungsdauer zwischen dem AfterEffects Filter „Körnung entfernen“, Neat Video und dem „3D Bilateral Filter“ wurde eine auf 512×512 Pixel beschnittene Version den `ndi 04.avi` Videos verwendet. Dadurch konnten die Geschwindigkeitsvergleiche auf den gleichen Rechnern stattfinden. So war auch ein Vergleich zwischen der schnellsten verfügbaren GPU und CPU möglich. Die Ergebnisse sind in Tabelle 7 zu finden.

8.5 Bewertung

8.5.1 Qualität

Die Qualitätsmessungen in Tabelle 2 zeigen mit den Fehlermaßen SSIM und MSE, wenn auch nicht sehr deutlich, dass die 3D Filterung ein besseres Ergebnis als die 2D Filterung erzielt. Beim Betrachten der gefilterten Videos wird jedoch eine sehr deutliche Überlegenheit sichtbar. So ist in den homogenen Flächen der 2D Filterung ein deutliches Flimmern wahrnehmbar, das bei der 3D Filterung nicht vorhanden ist. In Abb. 12 ist die Überlegenheit des 3D Filters kaum zu erkennen, nur das Flimmern deutet sich durch eine ganz leichte Fleckigkeit in der grauen Fläche an. Die bessere Filterleistung des „3D Bilateral Filters“ kann in Abb. 13, die das `ndi 04.avi` Video zeigt, deutlicher erkannt werden. Die hellen Flächen sind deutlich homogener als bei dem Ergebnis des „2D Bilateral Filters“. Da für diese Videosequenz jedoch kein Referenzvideo (aus Gründen die in Abschnitt 2.4 beschrieben sind) vorhanden ist, kann kein subjektives Fehlermaß bestimmt werden.

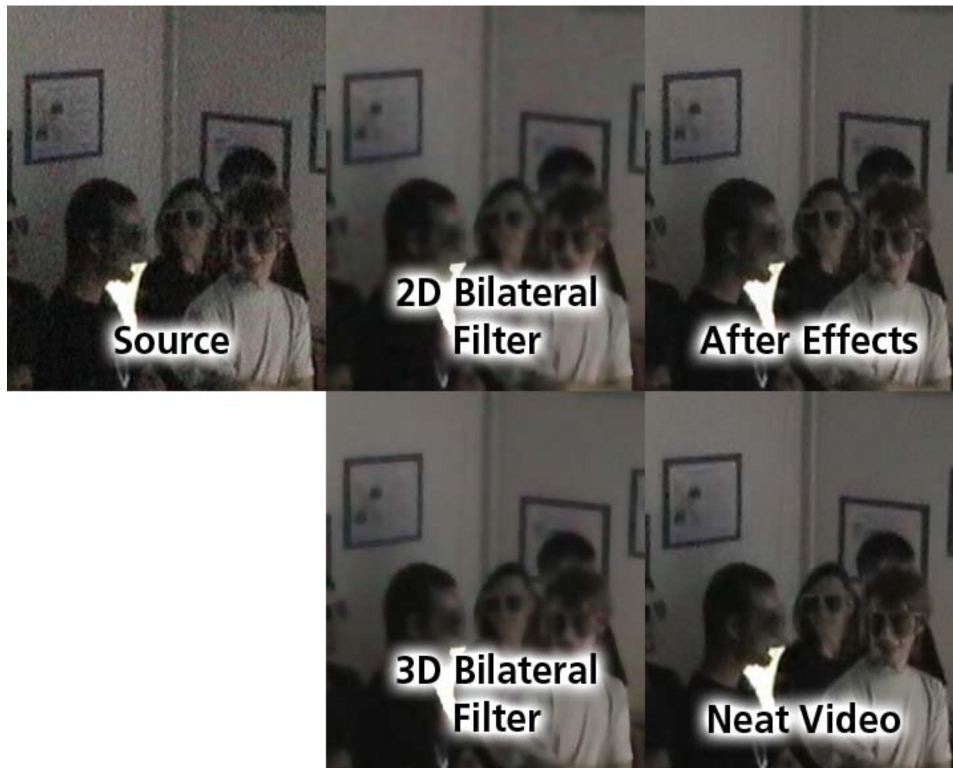

Abbildung 13: Qualitätsvergleich aus dem Video `ndi 04.avi`

An den unterschiedlichen Ausprägungen der Ergebnissen in Abb. 12 und Abb. 13 lässt sich erkennen, das die Wahl des Filters bzw. der Filterparameter deutlich das Ergebnis beeinflussen kann. Es ist auch festzuhalten, dass bei allen Filtern die Standardparameter gewählt wurden. Es ist durchaus möglich, dass durch die Wahl anderer Parameter ganz andere Ergebnisse erreicht worden wären.

Ein weiterer Punkt erschwert die Beurteilung der Qualitätsmessungen. Zwischen dem besten und dem schlechtesten Filterergebnis nach dem SSIM Fehlermaß in Tabelle 2 liegen nur 0,02 Punkte, nach dem MSE Fehlermaß 27,1 Punkte. Bei diesen geringen Unterschieden ist es schwierig zu behaupten, ein Filter sei besser als ein anderer. Alle Filter erreichen jedoch eine deutliche Verbesserung gegenüber dem ungefilterten Video. Jedoch ist auch die subjektive Beurteilung der Filterergebnisse wegen deren hohen Ähnlichkeit schwierig. Insofern kann man durchaus sagen, dass insbesondere das SSIM Fehlermaß die wahrgenommene Ähnlichkeit abbildet. Vielleicht werden zukünftige Entwicklungen das SSIM Fehlermaßes um eine zeitliche Komponente ergänzen, um die Aussage weiter zu verbessern.

8.5.2 Geschwindigkeit

Lohnt sich der Einsatz der GPU zur Filterung von Bildern? Diese Frage lässt sich nach den in dieser Arbeit gewonnenen Ergebnissen eindeutig mit „Ja" beantworten. So ist z.B. der schnellste „2D Bilateral Filter" auf einem 2048×2048 Pixel großen Bild auf der GPU über 60 mal schneller als auf der schnellsten CPU. Generell sind alle implementierten GPU Filter schneller als deren CPU Varianten. Jedoch sind diese Ergebnisse mit Vorsicht zu genießen. Sie spiegeln nur die konkret in dieser Arbeit implementierten Filter wieder. Es ist sicherlich möglich, sowohl die CPU als auch die GPU Versionen zu optimieren. Andererseits sind alle Varianten in ihrem gemessenen Zustand nach dem gleichen Kenntnisstand optimiert, so das ein Vergleich innerhalb dieser Versionen durchaus legitim ist.

Eine weitere Fragestellung ist, inwieweit der Graphikkartenbus die Filtergeschwindigkeit beeinflusst. Dazu kann man die Ergebnisse der beiden Computer „Bruce" und „Sid" vergleichen. Sie haben bei eine ähnlich gute Graphikkarte, jedoch ist „Bruce" mit den neueren und schnelleren PCI-Express Bus ausgestattet, im Gegensatz zu „Sid", der die Graphikkarten über den AGP anbindet. Der PCI-Express Bus kann laut Spezifikation Daten sowohl zur Graphikkarte hin, als auch im Besonderen von der Graphikkarte zurück in den Hauptspeicher schneller transferieren als AGP. Da für die Filterung permanent Bilder zwischen Hauptspeicher und Graphikkartenspeicher transferiert werden müssen, könnte ein Vorteil für den PCI-Express Bus vermutet werden. Die ermittelten Ergebnisse stützen diese Annahme nicht. Bei allen Messungen erzielt „Sid" eine etwas kürzere und damit bessere Filterungszeit als „Bruce", trotz des vermeintlich langsameren Buses. Dies liegt vermutlich an einer doch geringfügig schnelleren Graphikkarte. Daraus lässt sich schließen, dass der PCI-Express Bus im Gegensatz zu AGP bei der Verwendung aktueller Graphikkarten zur Filterung von Bildern keine Vorteile bringt.

8.5.3 Gesamtbewertung

Das beste Filterergebnis wird von *Neat Video* erzielt. Jedoch ist der zweitplatzierte „3D Bilateral Filter" auf aktuellen Graphikkarten mehr als zweimal schneller. Der von der Qualität her drittplatzierte „2D Bilateral Filter" ist gegenüber dem letztplatzierten AfterEffects Filter „Körnung entfernen" sogar 45 mal schneller. Da zur Zeit GPUs im Gegensatz zu CPUs von Generation zu Generation wesentlich deutlichere Leistungssteigerungen erfahren, ist zu erwarten, dass GPU basierte Bildfilter in Zukunft im Vergleich zu CPU basierten Bildfilter noch deutlichere Geschwindigkeitsvorsprünge ermöglichen.

Zusammenfassung und Ausblick

Diese Arbeit thematisiert die Problematik verrauschter Videoaufnahmen. Dazu wurde eine Einführung in die Verarbeitung von Videodaten auf dem Computer und deren Probleme wie das Rauschen gegeben. Zur Entfernung des Rauschens wurde ein Überblick über verfügbare kantenerhaltende Weichzeichnungsfilter gegeben und es wurde die Volumenfilterung von Videosequenzen als mögliche Verbesserung vorgestellt. Darauf folgte die Auswahl des Bilateral Filters als aussichtsreichster Kandidat für einen qualitativ hochwertigen und dennoch schnellen Filter zur Rauschentfernung. Danach wurden Messmethoden und Verfahren zum Vergleichen der Ergebnisse angegeben. Um eine zügige Filterung zu erreichen, wurde der Einsatz der Graphikkarte als universeller Coprozessor diskutiert und die Auswahl der *OpenGL Shading Language* zur Programmierung der GPU begründet. Die Grundlagen wurden mit einem Überblick über verfügbare Videoverarbeitungsprogramme und Videofilter vervollständigt.

Im zweiten Teil dieser Arbeit wurde ein Konzept für eine praktische Umsetzung des im ersten Teil vorgestellten Ansatzes der Volumenfilterung von Videosequenzen entwickelt. Es wurde sich für eine Erweiterung des Plexus-Framework zu einem Framework für die Filterung von Bildern und Videos auf der CPU und der GPU entschieden und die nötigen Erweiterungen besprochen. Dabei wurde insbesondere auf die nötigen Konzepte zur Volumenfilterung von Videosequenzen eingegangen. Danach wurden die Einschränkungen und Möglichkeiten, die bei der Verwendung der GPU zur Filterung von Bildern und Volumen zu erwarten sind, diskutiert. Darauf folgte eine Übersicht über die für Plexus erstellten Devices und Probleme, die bei deren Entwicklung auftraten. Der praktische Teil dieser Arbeit endete mit einer Reihe von Messungen der erzielten Qualität und Geschwindigkeit der umgesetzten Filtermöglichkeiten und einer Bewertung dieser Ergebnisse.

Es konnte gezeigt werden, dass eine qualitativ hochwertige und gleichzeitig schnelle Filterung erreicht werden kann. Außerdem wurde der teilweise deutliche Geschwindigkeitsgewinn durch Ausnutzung der Graphikkarte gezeigt. Der Vorteil der Filterung einer Videosequenz als Volumen statt Bild für Bild konnte ebenfalls durch Beispiele und Messergebnisse belegt werden. Das um Funktionen zur Filterung von Bilder und Volumen auf der CPU und GPU erweiterte Plexus-Framework steht für weitergehende Arbeiten und Analysen in diesem Bereich zur Verfügung.

Für die Zukunft blieben auch einige Herausforderungen und Probleme bestehen. Nachdem Computersysteme mit mehr als einer CPU oder mehreren Kernen pro CPU ihre Geschwindigkeitsvorteile belegt haben, erscheinen auch immer häufiger Graphikkarten, die über mehr als eine GPU verfügen

oder per *SLI* und *CrossFire* genannte Techniken zusammengeschaltet werden können. Es stellt sich somit die Frage, ob und wie gut, mehrere GPUs die Filterung von Bilddaten beschleunigen können. Außerdem sollten Techniken untersucht werden, mit denen Volumen größer als $512 \times 512 \times 512$ Pixeln auf Graphikkarten der Firma NVIDIA gefiltert werden können, um bei Videosequenzen in PAL oder HDTV Auflösung nicht nur auf Graphikkarten des Herstellers ATI angewiesen zu sein, da diese ebenfalls Einschränkungen besitzen.

Weiterhin sollte untersucht werden, ob sich mit anderen und komplexeren Filteralgorithmen bessere Ergebnisse erzielen lassen und ob sich diese ebenfalls für eine Beschleunigung durch die GPU eignen.

Abbildungsverzeichnis

1 Group-of-Pictures 9
2 Probleme in Videobildern 10
3 OpenGL Pipeline 25
4 VirtualDub GUI 27
5 Adobe AfterEffects 7.0 GUI 29
6 Blender GUI 30
7 Neat Video GUI 31
8 Einfacher Graph mit Plexus 33
9 Streaming video 39
10 VerticalCombineImageDevice 47
11 AfterEffects: Einstellungen des Filters „Körnung entfernen“ 52
12 Beispiel aus der Qualitätsmessung 54
13 Qualitätsvergleich aus dem Video `ndi 04.avi` 57

Tabellenverzeichnis

1 Bilddatentypen in Plexus 38
2 Qualitätsvergleich 54
3 Filterung mit dem „2D Bilateral Filter“ auf der CPU 55
4 Filterung mit dem „2D Bilateral Filter“ auf der GPU 55
5 Filterung mit dem „3D Bilateral Filter“ auf der CPU 55
6 Filterung mit dem „3D Bilateral Filter“ auf der GPU 56
7 Geschwindigkeitsvergleich 56

Literatur

[Bar00] Danny Barash. Bilateral filtering and anisotropic diffusion: Towards a unified viewpoint. *Third International Conference on Scale-Space and Morphology*, pages 273–280, 2000.

[Bly06] David Blythe. The Direct3D 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[BSMH98] M. J. Black, G. Sapiro, D. Marimont, and D. Heeger. Robust anisotropic diffusion. *IEEE Trans. on Image Processing*, 7(3):421–432, 1998.

[BV04] V. Bruni and D. Vitulano. Old movies noise reduction via wavelets and wiener filter. In V. Skala, editor, *Journal of WSCG*, volume 12, Plzen, Czech Republic, feb 2004. UNION Agency - Science Press.

[Gou98] Henri Gouraud. Continuous shading of curved surfaces. pages 87–93, 1998.

[Hyt05] Heli T. Hytti. Characterization of digital image noise properties based on raw data. *Image Quality and System Performance III, eds. Luke C. Cui, Yoichi Miyake. Proc. Of SPIE-IS&T Electronic Imaging, SPIE Vol. 6059, 60590A*, 2005.

[LKS+06] Aaron Lefohn, Joe M. Kniss, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Transactions on Graphics*, 25(1):60–99, January 2006.

[MGA03] W. Mark, S. Glanville, and K. Akeley. Cg: A system for programming graphics hardware in a C-like language, 2003.

[Mul] MultimediaWiki. YCbCr 4:2:2. `http://wiki.multimedia.cx/index.php?title=YCbCr_4:2:2&oldid=5995` (abgerufen am 26. September 2006).

[OLG+05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[Pho75] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, 1975.

[Poy03] Charles Poynton. *Digital Video and HDTV Algorithms and Interfaces*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[PvV05] T.Q. Pham and L.J. van Vliet. Separable bilateral filtering for fast video preprocessing. In *ICME 2005: IEEE International Conference on Multimedia & Expo, 2005*, page 4, Washington, DC, USA, 2005. IEEE Computer Society.

[Ros96] Randi J. Rost. Using opengl for imaging. *SPIE Medical Imaging '96 Image Display Conference*, 1996.

[Ros04] Randi J. Rost. *OpenGL®Shading Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[SA04] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 2.0). Technical report, Silicon Graphics, Inc., 2004.

[TM98] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *ICCV '98: Proceedings of the Sixth International Conference on Computer Vision*, page 839, Washington, DC, USA, 1998. IEEE Computer Society.

[WBSS04] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[Wika] Wikipedia: Die freie Enzyklopädie. Bild:GOP.gif. `http://de.wikipedia.org/w/index.php?title=Bild:GOP.gif&oldid=19014304` (abgerufen am 26. September 2006).

[Wikb] Wikipedia: Die freie Enzyklopädie. Pixel. `http://de.wikipedia.org/w/index.php?title=Pixel&oldid=21372255` (abgerufen am 26. September 2006).

[Wil97] Jan Willamowius. Framework-Evolution am Beispiel eines Frameworks zur Steuerung von Tk-Anlagen. Master's thesis, 1997.

Anhang