FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Effiziente Tiefenpropagierung in Videos mit GPU-Unterstützung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik

eingereicht von

Manuel Ivancsics
Matrikelnummer 0627658

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuerin: ao. Univ. Prof. Mag. DI Dr. Margrit Gelautz
Mitwirkung: Dipl.-Ing. Mag. Nicole Brosch

Wien, 25.11.2013

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Manuel Ivancsics
Denisgasse 23
1200 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Danksagung

Diese Diplomarbeit wurde durch das Projekt ICT08-019 des Wiener Wissenschafts-, Forschungs- und Technologiefonds (WWTF) gefördert.

Kurzfassung

Die zunehmende Beliebtheit von 3D-Technologien und die kommerzielle Verfügbarkeit von stereoskopischen Bildschirmen geht mit einem wachsenden Bedarf an passenden 3D-Inhalten einher. Die effiziente Generierung solcher Inhalte stellt somit ein aktuelles Problem dar. Eine Lösung dieses Problems ist die Konvertierung von bestehenden 2D-Inhalten zu 3D. Der damit verbundene Arbeitsaufwand von BenutzerInnen und die für die Konvertierung benötigte Zeit sind dabei zwei wichtige Faktoren. Diese Diplomarbeit präsentiert einen optimierten, semi-automatischen Algorithmus zur effizienten 2D-zu-3D-Konvertierung von Videos, der auf einem bereits vorhandenen Konvertierungs-Algorithmus beruht. Im existierenden Konvertierungs-Algorithmus stellen BenutzerInnen im Vorfeld grobe Tiefeninformationen zur Verfügung. Diese Tiefeninformationen werden im Anschluss automatisch mithilfe von Segmentierungs- und Filterungs-Techniken auf das gesamte Video propagiert. Das primäre Ziel dieser Diplomarbeit liegt in der Reduktion der Laufzeit des existierenden Konvertierungs-Algorithmus, bei gleichzeitiger Beibehaltung der Qualität der Ergebnisse. Das wurde durch die Verlagerung von besonders rechenintensiven Teilen des Algorithmus auf die Grafikkarte erreicht, wo eine parallele Abarbeitung stattfindet. Das beinhaltet auch die Optimierung der zu Grunde liegenden Segmentierungs- und Filterungs-Techniken. Die limitierte Kapazität des Grafikkartenspeichers erschwert die parallele Bearbeitung von großen Datenmengen wie Videos. In dieser Diplomarbeit wird ein Lösungsvorschlag für dieses Problem präsentiert, welcher darauf basiert, längere Videos in Subsequenzen aufzuteilen und nacheinander auf der Grafikkarte zu verarbeiten. Evaluierungen zeigen, dass die vorgestellte Optimierung des grundlegenden Konvertierungs-Algorithmus in der Lage ist, Ergebnisse in unverändert hoher Qualität zu generieren. Im Vergleich zur nicht-optimierten Version des Algorithmus wird die dafür benötigte Laufzeit deutlich reduziert. Ein Vergleich mit zwei verwandten Algorithmen zeigt, dass diese von der optimierten Implementierung dieser Diplomarbeit sowohl in Bezug auf die Qualität der Ergebnisse, als auch in Bezug auf die Laufzeit, deutlich übertroffen werden.

Abstract

The commercial availability of stereoscopic displays increases the demand for 3D content. Therefore, fast and robust tools for the generation of such content are of academic and commercial interest. An avenue to generating 3D content is to convert existing 2D content to 3D using specialized software. In this regard, the effort for annotation by conversion specialists and overall computation time are two crucial factors. To this end, this thesis presents an optimized semi-automatic algorithm for efficient 2D-to-3D video conversion that is based on an existing conversion algorithm. In the existing algorithm, the specialist provides sparse depth information. This depth is then propagated to the specified video clip, using segmentation and filtering techniques. The primary goal of this thesis is to reduce the computation time of the existing 2D-to-3D conversion algorithm, while maintaining the quality of its results. To achieve this goal, the most computationally expensive parts of the algorithm are re-implemented on the GPU, where they are processed in parallel. The optimization of the existing conversion algorithm also includes the optimization of the segmentation and filtering stages. The limited capacity of the GPU's onboard memory places limits on the parallel execution of large data sets like videos. This thesis solves this problem by splitting long videos into smaller sequences and processing them sequentially while incorporating methods for handling borders between the sequences. The evaluation shows that the proposed, optimized 2D-to-3D conversion algorithm is capable of generating high-quality results, while significantly reducing execution time compared to the original, un-optimized algorithm. Additional comparisons show that the proposed optimized conversion algorithm significantly outperformed two related conversion algorithms, in terms of both quality and execution time.

Inhaltsverzeichnis

Abkürzungsverzeichnis

CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
LAB	L*a*b-Farbraum
MSE	Mean Squared Error
RGB	Rot-Grün-Blau-Farbraum
SIFT	Scale Invariant Feature Transform

1. Einleitung

1.1. Motivation und Problemstellung

Mit der stetigen Weiterentwicklung von 3D-Bildschirmen und ihrer kommerziellen Verfügbarkeit steigt auch der Bedarf an 3D-Inhalten. Diese zeigen zwei leicht versetzte Ansichten einer Szene, eine für das linke und eine für das rechte Auge. Der Unterschied dieser Bilder wird vom menschlichen Gehirn als Tiefe wahrgenommen. Die Produktion solcher 3D-Inhalte ist zumeist mit großem Zeit- und Kostenaufwand verbunden. Diese Diplomarbeit beschäftigt sich mit der Reduktion dieses Aufwands.

Es gibt verschiedene Ansätze zur Produktion von 3D-Inhalten. In der Regel beinhalten diese Ansätze die Erstellung einer so genannten Tiefenkarte (siehe Abbildung 1.1, nahe: weiß, fern: schwarz), welche die Verschiebung (*Disparität*) zwischen linkem und rechtem Bild speichert. Die Disparität entspricht der relativen Tiefe [1]. Die beiden Begriffe, Tiefe und Disparität, werden in dieser Diplomarbeit synonym verwendet. Die generierten Tiefenkarten können zur automatischen Anpassung von 3D-Inhalten an unterschiedliche Darstellungsmedien oder unter Verwendung von speziellen Rendering-Techniken [2] zur Generierung von zusätzlichen Ansichten (zum Beispiel für das linke und rechte Auge) verwendet werden.

Eine Möglichkeit zur Erstellung von 3D-Inhalten ist die Aufnahme eines Filmes mit mehreren Kameras (oder einer stereoskopische Kamera). Werden zusätzlich Tiefenkarten benötigt, können diese mittels Stereo Matching [3] generiert werden. Ein weiterer Ansatz zur Erstellung von 3D-Inhalten ist die Verwendung von Tiefenkameras (beispielsweise *Time of Flight* [4], *ZCam* [5], *Kinect* [6]), mit denen Tiefeninformationen direkt beim Aufzeichnen des Videomaterials gemessen werden. Diese können anschließend zur Generierung einer zweiten Ansicht verwendet werden. Die Entscheidung, ob monoskopische (2D) oder stereoskopische (3D) Inhalte benötigt werden, muss bei beiden Ansätzen bereits im Vorfeld getroffen werden. Aufgrund der notwendigen zusätzlichen Anschaffung von teurem Equipment, lohnen sich diese Ansätze nur für große Produktionen, bei denen das nötige Budget zur Verfügung steht. Ein Beispiel für eine direkte Aufnahme von zwei Ansichten stellt der Film „Avatar“ dar, dessen offizielle Produktionskosten 230 Mio. US-Dollar betrugen. [7]

Ein anderer Ansatz zur Generierung von 3D-Inhalten ist die Konvertierung von bereits vorhandenen 2D-Inhalten zu 3D. Ziel der 2D-zu-3D-Konvertierung von Videos ist es, für alle Bilder (*Frames*) des Videos jeweils eine Tiefenkarte zu erstellen. Diese Tiefenkarten sollen einerseits von hoher Qualität sein, um anschließend in der 3D-Ansicht einen guten Tiefeneindruck zu ermöglichen. Qualitätsmerkmale sind beispielsweise zeitliche Kohärenz, kontrastreiche Tiefenübergänge an Objekträndern und glatte räumliche und zeitliche Tiefenänderungen. Andererseits soll die Generierung von Tiefenkarten einzelner Frames in möglichst kurzer Zeit erfolgen. Dies ist speziell für die Verarbeitung von größeren Datenmengen wie Videos von Bedeutung. Beispielsweise müssen für Videos mit einer Spieldauer zwischen 90 und 120 Minuten mit einer Framerate von 24 Frames pro Sekunde, zwischen 130.000 und 170.000 Frames bearbeitet werden. Die 2D-zu-3D-Konvertierung kann im Allgemeinen auf drei Arten erfolgen:

1. *Manuell*: Bei der weitgehend manuellen Konvertierungsmethode, *Rotoskopie* [8], werden für jedes Frame computergestützt jeweils zwei Ansichten generiert. Ein Animateur extrahiert zu diesem Zweck Objekte aus einem Frame und manipuliert diese von Hand, um die beiden für

den 3D-Eindruck benötigten Ansichten zu erstellen. Auf diese Weise können qualitativ hochwertige Ergebnisse generiert werden. Dieser Vorgang ist sehr zeitaufwändig und teuer und wird in großen Produktionen mit entsprechend hohem Budget eingesetzt. Die 2D-zu-3D-Konvertierung des Filmes „Titanic“ dauerte beispielsweise 60 Wochen (750.000 Arbeitsstunden) und kostete 18 Mio. US-Dollar [9].

2. *Automatisch:* Die Motivation dieser Ansätze liegt in der Verringerung der benötigten Zeit, die für die 2D-zu-3D-Konvertierung aufgewendet werden muss. Automatische Konvertierungsmethoden [10, 11, 12, 13] generieren ohne Interaktion durch BenutzerInnen aus einem 2D-Video ein 3D-Video. Tiefeninformationen werden zu diesem Zweck aus einem bestehenden 2D-Video ermittelt. Beispielweise können Tiefenwerte aufgrund von Bewegungen der Kamera geschätzt werden, unter der Annahme, dass Objekte im Vordergrund tendenziell größere Bewegungen aufweisen als Objekte im Hintergrund. Solche Annahmen treffen oft nur für bestimmte 2D-Videos zu und beschränken sich somit auf das zu konvertierende 2D-Videomaterial. Das Hauptproblem bei voll-automatischen Methoden ist, dass die Resultate ohne Interaktion seitens der BenutzerInnen schwer zu kontrollieren sind. Fehler in den resultierenden Tiefenkarten sind schwer korrigierbar, was eine aufwändige Nachbearbeitung notwendig machen kann.

3. *Semi-automatisch*: Die Motivation von semi-automatischen Konvertierungsmethoden [1, 14, 15, 16, 17, 18, 19, 20, 21, 22] ist es, die Lücke zwischen automatischen und manuellen Methoden zu schließen. Semi-automatische Methoden basieren auf spärlicher Tiefeninformation, welche vor der eigentlichen Konvertierung von BenutzerInnen zur Verfügung gestellt wird. BenutzerInnen bearbeiten zu diesem Zweck bestimmte Frames eines Videos (*Schlüssel-Frames*), beispielsweise unter Verwendung von externen Bildbearbeitungsprogrammen (beispielsweise Adobe Photoshop), und weisen den Bildobjekten ungefähre Tiefenwerte zu. Diese Tiefenwerte werden anschließend automatisch auf die restlichen Frames des Videos propagiert. BenutzerInnen haben auf diese Weise eine höhere Kontrolle über die Resultate als bei automatischen Methoden. Somit können Tiefenkarten in einer hohen Qualität und in relativ kurzer Zeit generiert werden.

a) b)

Abbildung 1.1: *a) Eingabebild. b) Tiefenkarte. Die Tiefenwerte der Pixel sind als Grauwerte repräsentiert (weiß: nahe, schwarz: fern).*

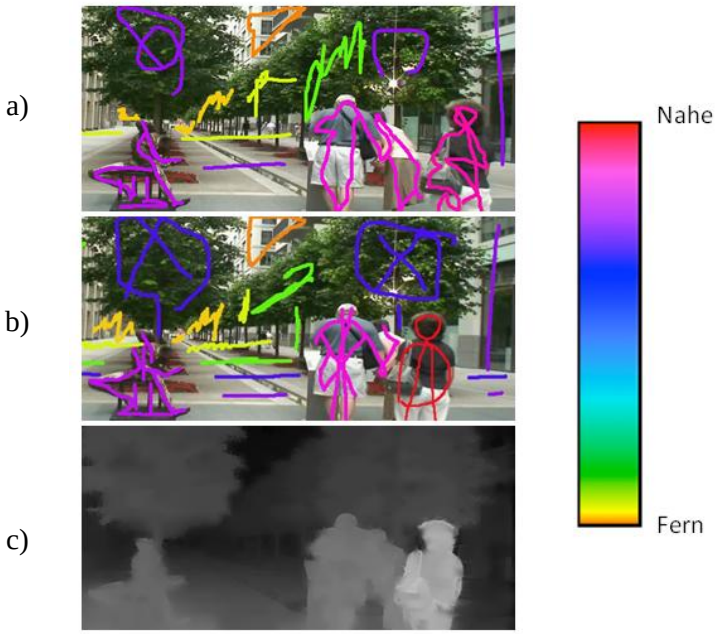


Abbildung 1.2: *BenutzerInnen zeichnen Tiefenwerte in Form von Scribbles in die Schlüssel-Frames des Eingabevideos ein. Diese werden anschließend automatisch auf die restlichen Frames des Videos propagiert. a) Erstes Frame mit Scribbles. b) Letztes Frame mit Scribbles. c) Mit dem Algorithmus dieser Diplomarbeit erstellte Tiefenkarte eines dazwischenliegenden Frames.*

Diese Diplomarbeit konzentriert sich auf die semi-automatischen 2D-zu-3D-Konvertierungsmethoden und die Reduktion des Zeit- und Kostenaufwands. Dazu sollen einerseits der benötigte Rechenaufwand und die Laufzeit der Konvertierung minimiert werden. Andererseits sollen die Dauer und das Ausmaß der Interaktion von BenutzerInnen möglichst gering gehalten werden. Existierende Verfahren [1, 11, 14, 15, 16, 17, 18, 21] reduzieren die von BenutzerInnen benötigte Interaktion zur Vordefinition von wenigen Tiefen, etwa in Form von Scribbles (siehe Abbildung 1.2). Zu etablierten Strategien zur Reduktion der Laufzeit zählen das vorherige Verkleinern des zu konvertierenden Videomaterials [1], das Gruppieren und gemeinsame Verarbeiten von Pixeln [21], oder die parallele Implementierung des Algorithmus auf der Grafikkarte [11, 15]. Diese Diplomarbeit beschäftigt sich mit der effizienten, parallelen Implementierung eines aktuellen Verfahrens zur 2D-zu-3D-Konvertierung, welches bei minimaler Interaktion von BenutzerInnen hoch-qualitative und zeitlich kohärente Tiefenkarten produzieren kann [22]. Die Qualität der Ergebnisse, insbesondere die oft schwer zu erzielenden kontrastreichen Tiefenübergänge an Objekträndern [1], ergibt sich aus einer gleichzeitigen Segmentierung des Videos und Propagierung der Tiefenwerte. Glatte räumliche und zeitliche Tiefenänderungen innerhalb von Objekten werden in [22] durch effizient implementierbare Filteroperationen [23] realisiert. In [22] definieren BenutzerInnen Tiefenwerte sowohl im ersten als auch im letzten Frame eines Videos, wodurch zeitliche Änderungen besser erfasst werden können, als beispielweise durch den Algorithmus von [14].

1.2. Ziele und Beitrag

Das primäre Ziel dieser Diplomarbeit ist es, die Laufzeit eines Algorithmus zur semi-automatischen 2D-zu-3D-Konvertierung [22], bei gleichbleibender Qualität der Ergebnisse, zu verkürzen. Die verringerte Laufzeit führt zu einer erhöhten Skalierbarkeit des Algorithmus, wodurch auch die Konvertierung von längeren Videos ermöglicht wird. Des Weiteren steigert eine Verringerung der

Laufzeit die Nutzbarkeit der Anwendung. Das Hauptproblem liegt dabei in der enormen Datenmenge, die im Zuge der Konvertierung eines Videos bearbeitet werden muss. Einige existierende Methoden versuchen die Laufzeit der Konvertierung durch Reduktion der zu bearbeitenden Datenmenge zu verkürzen, indem beispielsweise die Auflösung des Eingabevideos reduziert wird [1] oder die Pixel eines Frames im Vorfeld in zusammenhängende Regionen gruppiert werden [21]. Andere Methoden beschränken die Verringerung der Konvertierungszeit auf Reduzierung der Zeit, die BenutzerInnen für das Vordefinieren von Tiefen benötigen. Beispielsweise werden in [20] Tiefenkarten für Schlüssel-Frames automatisch generiert und von BenutzerInnen korrigiert, falls die Ergebnisse nicht zufriedenstellend sind. Wiederum andere Methoden verlagern besonders rechenintensive Teile ihrer Algorithmen auf die GPU und können dadurch große Laufzeitverringerungen erzielen [11, 15].

Das Ziel der Verringerung der Laufzeit des Algorithmus aus [22] wird durch eine parallele GPU-Implementierung von besonders rechenintensiven Teilen des Algorithmus erreicht. Serielle Programmabläufe werden zu diesem Zweck auf die Grafikkarte verlagert, wo sie von einer Vielzahl von GPU-Kernen parallel abgearbeitet werden können. Die Grafikkartenprogrammierung erfolgt in dieser Diplomarbeit unter Verwendung von NVIDIA CUDA (*Compute Unified Device Architecture*) [24]. Aufgrund der Voraussetzungen der CUDA-Architektur, besonders in Bezug auf die interne Verwaltung des Grafikkartenspeichers und den Zugriff auf die Speicherbereiche, müssen Teile des Algorithmus aus [22] adaptiert werden, um eine effektive parallele Bearbeitung auf der GPU zu ermöglichen. Die Optimierung des Algorithmus aus [22] geschieht auf zwei Ebenen:

1. *Segmentierung und Propagierung*: Die Laufzeit der Segmentierung und Propagierung in [22] wird durch eine Optimierung des zugrunde liegenden Segmentierungs-Algorithmus aus [25] verringert. Die Laufzeitverringerung resultiert aus parallelen GPU-Implementierungen von besonders rechenintensiven Programmabschnitten des Segmentierungs-Algorithmus. Als zusätzliches Ergebnis wird in dieser Diplomarbeit somit auch die Laufzeit des zu Grunde liegenden Segmentierungs-Algorithmus aus [25] reduziert. Die optimierte Version des Segmentierungs-Algorithmus ist durch die limitierte Kapazität des GPU-Speichers eingeschränkt. Aufgrund dessen kann nur eine limitierte Anzahl von Frames gleichzeitig bearbeitet werden. Längere Videos werden zu diesem Zweck in Subsequenzen aufgeteilt, welche nacheinander auf der GPU bearbeitet werden.

2. *Filterung*: Da die Tiefenwerte, welche im Zuge der Segmentierung an alle Pixel eines Videos zugewiesen werden, abrupte Tiefensprünge enthalten können, werden diese im Verfeinerungsschritt mittels räumlicher und zeitlicher Regularisierung geglättet. Zu diesem Zweck wird eine adaptierte Version des Guided Filters [23], ein kantenerhaltender Glättungsfilter, getrennt auf die einzelnen räumlichen und zeitlichen Regionen des segmentierten Videos angewandt. Zur Optimierung der Nachbearbeitung in [22] wird eine GPU-Version des Guided Filters für Videos erstellt. Diese baut auf einer bereits vorhandenen GPU-Implementierung des Guided Filters für Bilder auf, welche im Zuge der Diplomarbeit aus [26] erstellt wurde. Diese Implementierung wird in dieser Diplomarbeit einerseits für die Verwendung auf Videos erweitert. Andererseits wird eine GPU-Version des Guided Filters zur effektiven regionsweisen Filterung eines Videos implementiert. Wie die GPU-Implementierung der Segmentierung, ist auch die GPU-Implementierung des Guided Filters durch die limitierte Speicherkapazität der GPU eingeschränkt. Längere Videos werden daher in Subsequenzen aufgeteilt, welche nacheinander auf der GPU gefiltert werden.

Die optimierte Implementierung des Algorithmus aus [22] liefert dieselben, qualitativ hochwertigen Ergebnisse wie die bestehende C++-Implementierung. Die Laufzeit wird hingegen deutlich reduziert. Durch die implementierten Lösungen des Problems des limitierten Grafikkartenspeichers ist es zudem möglich, längere Videos in relativ kurzer Zeit zu bearbeiten. Die Evaluierung in Kapitel 7 zeigt in einem direkten Vergleich, dass die optimierte Implementierung die beiden verwandten Algorithmen aus [14] und [1], sowohl in Bezug auf die Qualität der resultierenden Tiefenkarten, als auch auf die benötigte Laufzeit, deutlich übertrifft.

1.3. Aufbau der Arbeit

Diese Diplomarbeit ist in acht Kapitel gegliedert. Kapitel 2 gibt einen Überblick über verschiedene semi-automatische Algorithmen der 2D-zu-3D-Konvertierung. Da das primäre Ziel dieser Diplomarbeit die Reduktion der Laufzeit ist, werden im Zuge dessen auch die von den jeweiligen Algorithmen durchgeführten Strategien zur Reduzierung der Laufzeit diskutiert. In Kapitel 3 werden die Grundlagen der parallelen Grafikkartenprogrammierung unter Verwendung von NVIDIA CUDA [24] diskutiert. Dabei wird auf die wichtigsten Aspekte der CUDA-Architektur eingegangen, welche bei der Entwicklung der parallelen GPU-Implementierungen dieser Diplomarbeit relevant sind. In Kapitel 4 werden die Grundlagen der Grafikkartenprogrammierung mittels CUDA anhand einer effizienten parallelen GPU-Implementierung eines linearen Glättungsfilters, dem Box-Filter, vertieft. Dabei wird auf die Besonderheiten der CUDA-Architektur und die damit verbundenen Herausforderungen, insbesondere in Bezug auf die interne Verwaltung des Grafikkartenspeichers, eingegangen. Der Box-Filter wird im Zuge dessen zur Filterung von Videos erweitert. Dieses Beispiel dient nicht nur der Heranführung von LeserInnen an das Thema, sondern stellt auch die Grundlage späterer Implementierungen dar. Eine Evaluierung der optimierten seriellen GPU-Implementierung des Box-Filters erfolgt anschließend mittels eines Laufzeitvergleiches mit einer vorhandenen CPU-Implementierung [27].

Kapitel 5 behandelt die Grundlagen der Segmentierung eines Videos mit gleichzeitiger Propagierung von Tiefenwerten aus [22]. Der verwendete Segmentierungs-Algorithmus basiert auf dem Algorithmus zur Segmentierung von Videos aus [25]. Dieser stellt wiederum eine Erweiterung des Bild-Segmentierungs-Algorithmus aus [28] dar. Zu Beginn von Kapitel 5 werden die beiden Segmentierungs-Algorithmen, welche dem Algorithmus aus [22] zu Grunde liegen, diskutiert. Danach wird die Segmentierung und gleichzeitige Propagierung von Tiefenwerten aus [22] behandelt. Im Anschluss daran wird diskutiert, wie die Laufzeit des Segmentierungs-Algorithmus aus [22] durch parallele GPU-Implementierungen von besonders rechenintensiven Programmabschnitten verringert werden kann. Die resultierende Laufzeitverringerung der optimierten Version des Algorithmus aus [22] wird anschließend, im Zuge eines Laufzeitvergleichs mit der ursprünglichen Version, demonstriert. Die Verkürzung der Laufzeit des Algorithmus aus [22] wird hauptsächlich durch die Optimierung der Laufzeit des zu Grunde liegenden Segmentierungs-Algorithmus aus [25] erreicht. Als zusätzliches Ergebnis wird somit auch eine optimierte Version des Segmentierungs-Algorithmus aus [25] erstellt.

Kapitel 6 behandelt die Verfeinerung der Tiefenkarten aus [22] unter Verwendung des Guided Filters. Zu Beginn werden die Grundlagen des Guided Filters und eine sequentielle CPU-Implementierung erklärt. Im Anschluss wird eine parallele GPU-Implementierung des Guided Filters für Videos diskutiert. Diese basiert auf der bereits vorhandenen GPU-Implementierung des Guided Filters für Bilder aus [26]. In dieser Diplomarbeit wird diese Implementierung zum Zweck der Filterung von Videos erweitert. Zudem wird eine optimierte Implementierung des Guided Filters zur effizienten

regionsweisen Filterung von Videos diskutiert, welche in dieser Diplomarbeit erstellt wird. Am Ende des Kapitels wird die Laufzeitverringerung der optimierten GPU-Implementierung des Guided Filters, in einem Vergleich zu einer sequentiellen CPU-Implementierung [27], demonstriert.

In Kapitel 7 werden die Ergebnisse und Laufzeiten der in dieser Diplomarbeit erstellten optimierten Implementierung zur segmentierungsbasierten Propagierung von Tiefenwerten in Videos evaluiert. Der optimierte Algorithmus wird zu diesem Zweck auf elf Testvideos angewandt. Die resultierenden Tiefenkarten werden im Zuge eines Vergleichs mit Referenzergebnissen evaluiert. Im Anschluss wird die Aufteilung von längeren Videos in Subsequenzen diskutiert. Dazu wird ein Vergleich der Ergebnisse mit und ohne Aufteilen der Videos durchgeführt. Die Ergebnisse und Laufzeiten der Implementierung dieser Diplomarbeit werden mit den beiden verwandten Propagierungs-Algorithmen aus [1] und [14] verglichen. Dabei wird gezeigt, dass diese sowohl in Bezug auf die Qualität der Ergebnisse, als auch in Bezug auf die Laufzeiten, von der optimierten Implementierung dieser Diplomarbeit deutlich übertroffen werden.

In Kapitel 8 werden die wesentlichen Erkenntnisse dieser Diplomarbeit zusammengefasst und Anregungen für zukünftige Forschungsarbeiten, auch im Bereich der effektiven Bearbeitung von längeren Videos, diskutiert.

2. Verwandte Arbeiten

Dieses Kapitel gibt eine Übersicht verschiedener Algorithmen im Bereich der semi-automatischen 2D-zu-3D-Konvertierung, welche Interaktion durch BenutzerInnen erfordern. Das Ziel dieser Algorithmen ist es, Tiefenwerte, welche von BenutzerInnen für einzelne Bildbereiche in Form von Scribbles [1, 15] oder für gesamte Frames [14, 16] vordefiniert werden, über ein gesamtes Eingabevideo zu propagieren. Das resultierende Tiefenvideo soll plausibel und zeitlich kohärent sein, also keine plötzlichen Änderungen oder Flackern enthalten. Zudem soll die Bearbeitung in möglichst kurzer Zeit erfolgen, um eine effiziente Bearbeitung von Videos zu ermöglichen. Aus diesem Grund werden in dieser Übersicht über Konvertierungsverfahren auch die von den jeweiligen Algorithmen durchgeführten Strategien zur Reduzierung der Laufzeit diskutiert. Das folgende Kapitel diskutiert insbesondere zwei Konvertierungsverfahren: Die Propagierung von Tiefenwerten, basierend auf Videofilterung aus [14] (Abschnitt 2.1) und den Stereo Extraktions-Algorithmus aus [1] (Abschnitt 2.2). In Kapitel 7 werden diese mit der optimierten Implementierung des Algorithmus von [22], welche in dieser Diplomarbeit erstellt wurde, verglichen. Die auf Segmentierung basierende Tiefenpropagierung aus [22], welche die Basis dieser Diplomarbeit darstellt, wird in Kapitel 5 thematisiert.

2.1. Propagierung basierend auf Videofilterung

In der Literatur zur semi-automatischen 2D-zu-3D-Konvertierung findet man verschiedene Algorithmen, welche Tiefenwerte mithilfe von Videofilterung propagieren [14, 16, 18, 22]. Dazu gehört unter anderem der Algorithmus von [14], welcher die vordefinierten Tiefenwerte eines gesamten Schlüssel-Frames iterativ auf Folgeframes propagiert. Die Tiefenwerte für das Schlüssel-Frame können entweder von BenutzerInnen manuell zur Verfügung gestellt werden oder mit einer anderen Methode generiert werden (beispielsweise unter Verwendung des Stereo Matching-Algorithmus aus [3]). In [14] erfolgt die Propagierung der Tiefenwerte mithilfe des Bilateral Filters [29]. Dieser kantenerhaltende Filter glättet Tiefenwerte in einem Filterfenster des Vorgängerframes, basierend auf der Farbähnlichkeit zu einem Pixel im aktuellen Frame. Der geglättete Tiefenwert wird anschließend dem Pixel des aktuellen Frames zugewiesen. Durch diese lokale Propagierung können zwei Probleme auftreten:

- *Mehrdeutigkeit:* Wenn innerhalb eines Filterfensters Pixel enthalten sind, die ähnliche Farben, aber unterschiedliche Tiefenwerte aufweisen (Objekte im Vordergrund haben dieselbe Farbe wie Objekte im Hintergrund), kann dies zu gemischten Tiefenwerten führen, welche eine Mischung aus Vordergrund und Hintergrund darstellen.

- *Neue Farbe*: Dieses Problem tritt auf, wenn das aktuelle Frame eine Farbe enthält, die in den vorherigen Frames nicht vorhanden war (Aufdeckung).

Der Einfluss dieser beiden Probleme wird in [14] reduziert, indem die Tiefenwerte in jedem Frame direkt nach der Propagierung korrigiert werden. Zu diesem Zweck werden Bewegungsvektoren (*Optical Flow*) berechnet und angenommen, dass korrespondierende Tiefenwerte im jeweiligen vorangegangenen Frame keine Fehler enthalten. Die mittels Bilateral Filter propagierten Tiefenwerte werden anschließend durch Kopieren korrespondierender Tiefenwerte des Vorgängerframes korrigiert

(siehe Abbildung 2.1). Treten in einem Frame Fehler auf, die nicht korrigiert werden können, werden diese aufgrund der iterativen Vorgehensweise auf nachfolgende Frames propagiert. Da nur jeweils das erste Frame eines Videos von BenutzerInnen definierte Tiefeninformationen enthält, welche als Grundlage für die Propagierung dienen, kann der Algorithmus aus [14] keine Tiefenänderungen modellieren. Im Vergleich mit der optimierten Implementierung des Algorithmus aus [22] ergab sich unter Verwendung eines PCs mit einem Intel Xenon E5 Prozessor mit 3.6GHz, 32GB RAM und einer NVIDIA GeForce GTX 680 Grafikkarte, für ein Video mit der Auflösung 640 x 480, eine durchschnittliche Laufzeit von 19.29 Sekunden pro Frame (siehe Kapitel 7).

a) b) c) d)

Abbildung 2.1: *Beispielkonvertierung durch* [14]*: a) Frame des Eingabevideos. b) Ground Truth-Tiefenkarte. c) Mittels Bilateral Filter propagierte Tiefenwerte. d) Tiefenwerte nach der Korrektur von* [14].

a) b) c)

Abbildung 2.2: *Beispielkonvertierung durch* [1]*: a) Erstes Frame eines Eingabevideos. b) Letztes Frame. c) Resultierende Disparitätskarte in einem dazwischenliegenden Frame.* [1]

2.2. Propagierung durch globale Optimierung

Der Algorithmus zur 2D-zu-3D-Konvertierung von [1] basiert, wie auch die in dieser Diplomarbeit erstellte optimierte Implementierung des Algorithmus von [22], auf Scribbles im ersten und letzten Frame eines Videos. Pixel, welche sich innerhalb eines Scribbles befinden, erhalten, abhängig von der Farbe der Scribbles, Disparitäten. In der ersten Phase des Algorithmus werden diese auf die restlichen Pixel der markierten Frames propagiert. Das konvertierte erste und das konvertierte letzte Frame werden anschließend dazu benutzt, den Zusammenhang von Tiefe und Bildinhalt zu lernen. Dies erfolgt mittels Training eines *Support Vector Machine (SVM)*-Klassifikators [1] für jede der eingezeichneten Disparitäten. Anschließend werden jene Klassifikatoren mit der niedrigsten Fehlerrate ausgewählt, um Tiefenwerte an einigen Punkten (*Ankerpunkte*) im Video zu definieren. Die Disparitätskarte des restlichen Videos wird anschließend im Zuge eines Optimierungsprozesses (*Least Square Modeling)* generiert. Diese Optimierung basiert auf drei Annahmen:

1. Farblich ähnliche, benachbarte Pixel haben ähnliche Disparitäten.

2. Disparitäten ändern sich kontinuierlich, entsprechend den Bewegungsvektoren des Optical Flow aus [30].

3. Die Propagierung orientiert sich an den eingezeichneten Scribbles und den Tiefenwerten an den Ankerpunkten.

Die Optimierung beinhaltet das Lösen großer Gleichungssysteme (*N* x *N*, bei *N* Pixel im Video). Der Rechenaufwand steigt dabei quadratisch mit der Anzahl der Pixel in einem Video. Dieser Vorgang ist sehr rechenintensiv und kann dementsprechend zu langen Laufzeiten führen. Aus diesem Grund wird in [1] die Auflösung der Eingabevideos vor der Bearbeitung um den Faktor vier verringert. Nach der Konvertierung wird das resultierende Tiefenvideo mithilfe der Methode aus [31] auf die ursprüngliche Auflösung vergrößert. Die nachträgliche Vergrößerung der Tiefenvideos kann, vor allem an Objekträndern, zu ungenauen Disparitätskarten führen (siehe Abbildung 2.2). Der Vergleich mit der optimierten Implementierung dieser Diplomarbeit ergab unter Verwendung eines PCs mit einem Intel Xenon E5 Prozessor mit 3.6GHz, 32GB RAM und einer NVIDIA GeForce GTX 680 Grafikkarte für ein Video mit der Auflösung 640 x 480, eine durchschnittliche Laufzeit von 41.18 Sekunden pro Frame (siehe Kapitel 7).

a) b) c)

Abbildung 2.3: *Beispielkonvertierung durch* [15]*: a) Eingabeframe mit eingezeichneten Scribbles (weiß: vorne, schwarz: hinten). b) Generierte Disparitätskarte (weiß: hinten, schwarz: vorne). Die Objektränder konnten korrekt abgebildet werden. c) Anaglyphen-3D-Ausgabe.* [15]

2.3. StereoBrush

Bei StereoBrush [15] handelt es sich um eine interaktive 2D-zu-3D-Konvertierungsmethode, welche die Tiefenwerte der von BenutzerInnen eingezeichneten Scribbles durch Optimierung (*Least Square Modeling*) propagiert. Ähnlich der in [1] durchgeführten Optimierung, basiert die Propagierung auf der Annahme, dass farblich ähnliche Nachbarpixel ähnliche Disparitäten aufweisen. Im Gegensatz zu verwandten Algorithmen [1, 22] werden Scribble-Disparitäten jedoch nicht direkt auf das Video angewandt, sondern sind in eine Disparitäts-Hypothese eingebettet. Zum einen stellen in [15] Disparitäten der Scribbles ein grobes Konzept für die Nähe der Bildelemente zur Kamera dar und müssen nicht direkt übernommen werden, sondern werden stattessen auf einem Intervall abgebildet. Zum anderen werden die abgebildeten Disparitäten zusätzlich und gleichzeitig im Kontext eines von ihnen generierten stereoskopischen Bildpaares optimiert. Dieser Kontext ermöglicht das Formulieren zusätzlicher Annahmen. Beispielsweise wird angenommen, dass sich Nachbarpixel im gegebenen Farbvideo auch in den neu generierten Ansichten im Allgemeinen nicht weit voneinander wegbewegen. Die einzige Ausnahme dieser Annahme bilden Bildkanten, da große Farbähnlichkeiten von Nachbarpixeln auf verschiedene Objekte und damit gegebenenfalls einhergehende

Disparitätswechsel hindeuten. Durch diese Annahme können glatte Tiefenübergänge und damit Objekt-Deformationen in den neuen Ansichten reduziert werden (siehe Abbildung 2.3). Dieser strukturerhaltende Effekt wird besonders in visuell hervorstechenden Regionen erzwungen (siehe [15] für Details). Ähnlich zu [1] werden die Disparitätskarten sowie die zusätzlichen Ansichten, in einem Optimierungsprozess ermittelt, der auf dem Lösen eines großen Gleichungssystems (N x N, bei N Pixel im Video) beruht. Um den quadratisch mit der Anzahl der Pixel steigenden Rechenaufwand zu reduzieren und eine interaktive Verwendung zu ermöglichen, wurde die Laufzeit des Konvertierungs-Algorithmus durch eine parallele GPU-Implementierung reduziert. BenutzerInnen erhalten dadurch beim Einzeichnen der Scribbles direktes Feedback in Form von Tiefenkarten. Laut [15] konnte somit unter Verwendung einer NVIDIA GeForce GTX 480 GPU für ein Video mit der Auflösung 1080 x 720 eine durchschnittliche Updaterate der Tiefenkarten von zwei Frames pro Sekunde erreicht werden.

a)

b)

c)

Abbildung 2.4: *Beispielkonvertierung durch* [16]*: Zuweisung der Disparitäten in Schlüssel-Frames. a) Frame des originalen Eingabevideos mit eingezeichneten Scribbles für die interaktive Segmentierung von Videoobjekten. b) Ergebnis der Segmentierung der Vordergrundobjekte. Die Farben der Scribbles dienen hier lediglich der Illustration der einzelnen Segmentierungsprobleme und codieren, anders als bei verwandten Algorithmen* [1, 15, 22]*, keine Disparitäten. c) Zuweisung der Disparitäten an die markierten Objekte.* [16]

2.4. Interaktive segmentierungsbasierte Propagierung

Ähnlich zu Rotoskopie (siehe Kapitel 1), implementiert der Propagierungs-Algorithmus aus [16] die Annahme der Konsistenz von Farb- mit Disparitätsähnlichkeiten als interaktive Segmentierung und anschließender segmentweiser Propagierung von vordefinierten Disparitäten. Der Algorithmus besteht im Wesentlichen aus den folgenden drei Schritten:

1. Im ersten Schritt werden die Pixel von Schlüssel-Frames im Zuge einer Übersegmentierung in zusammenhängende Regionen gegliedert. BenutzerInnen definieren durch Einzeichnen von Scribbles Bildobjekte, welche interaktiv aus der Szene ausgeschnitten werden (siehe Abbildung 2.4 a)-b)).

2. BenutzerInnen weisen den ausgeschnittenen Objekten anschließend manuell Disparitäten zu (siehe Abbildung 2.4 c)). Die Zuweisung wird durch vordefinierte Modelle (beispielsweise Tiefenmodelle für Schrägen, Sphären oder Zylinder) erleichtert.

3. Die Disparitäten aus den Schlüssel-Frames werden anschließend, basierend auf Videofilterung (einer Version der in Abschnitt 1.1. vorgestellten Methode [14]), auf die restlichen Frames des Videos propagiert. Im Gegensatz zu [14], werden in [16] nicht nur eines, sondern zwei Schlüssel-Frames berücksichtigt. Das finale Ergebnis ist eine Kombination der propagierten Disparitäten bezüglich der beiden Schlüssel-Frames, welche dem Ziel-Frame am nächsten liegen. Diese Kombination erfolgt in Abhängigkeit von ihrer jeweiligen Entfernung zum Ziel-Frame.

In [16] wird unter Verwendung eines PCs mit einer Intel 2.8GHz CPU und 4GB RAM für ein Video mit einer Auflösung von 720 x 576 eine durchschnittliche Bearbeitungszeit von 18 Sekunden pro Frame angegeben. In diesem Kontext ist zusätzlich anzumerken, dass der Propagierungs-Algorithmus aus [16] im Vergleich zu verwandten Algorithmen [15, 14, 17, 22] mehr Interaktion und Zeit von Seiten der BenutzerInnen erfordert, da sowohl die Segmentierung der Vordergrundobjekte, als auch die Zuweisung der Disparitäten weitgehend manuell erfolgt.

a) b) c) d)

Abbildung 2.5: *Beispielkonvertierung durch* [17]*: Generierung von Tiefenkarten für ein Video von 43 Frames mit Frame 1, 14 und 43 als Schlüssel-Frames. a) Frame 6 des Eingabevideos. b) Resultierende Tiefenkarte für Frame 6. Mit Ausnahme einiger Fehler im Bereich des Kopfes und im Hintergrund wurde die grundlegende Struktur der Szene korrekt repräsentiert. c) Frame 32 des Eingabevideos. d) Resultierende Tiefenkarte für Frame 32. Die Entfernung zu den Schlüssel-Frames 14 und 43 ist größer als bei b), was in einem größeren Fehler resultiert.* [17]

2.5. Konvertierung basierend auf maschinellem Lernen

Die Konvertierungsmethode aus [17] verwendet einen maschinellen Lern-Algorithmus (*MLA*) zur semi-automatischen Generierungen von Tiefenkarten. Durch die Verwendung eines MLA kann die benötigte Zeit zur Genierung von Tiefenkarten, im Vergleich zur manuellen Zuweisung, deutlich reduziert werden. Der MLA kann als Black Box betrachtet werden, die trainiert werden kann, um Beziehungen zwischen einem Set von Eingabewerten zu erkennen und ein entsprechendes Set an Ausgabewerten zu generieren. In [17] werden dem MLA die kartesischen Koordinaten und die Farbwerte eines Pixels übergeben. Als Ausgabe liefert der MLA den Tiefenwert des Pixels. Der MLA besteht aus zwei Phasen:

1. *Training*: In dieser Phase wird der MLA, basierend auf Beispielen, bei denen die Tiefenwerte der Pixel bekannt sind, trainiert. Im Zuge des Trainings wird die interne Konfiguration des MLA dahingehend angepasst, dass Pixel, abhängig von ihren Koordinaten und Farbwerten, den richtigen Tiefenwerten zugeordnet werden können.

2. *Klassifikation*: In dieser Phase werden dem MLA Pixel übergeben, deren Tiefenwerte nicht bekannt sind. Der MLA klassifiziert diese, basierenden auf den im Zuge der Trainingsphase erlernten Relationen zwischen Pixeln und Tiefenwerten.

Nach Training eines Klassifikatiors, erfolgt die 2D-zu-3D-Konvertierung in folgenden zwei Schritten:

1. *Tiefen-Mapping*: In diesem Schritt wird der MLA für Schlüssel-Frames angewandt. Im Zuge dessen werden den Schlüssel-Frames Tiefenwerte zugewiesen.

2. *Tiefen-Tweening*: In diesem Schritt wird für jedes Schlüssel-Frame ein separater MLA trainiert. Die MLAs werden anschließend verwendet, um Tiefenkarten für alle dazwischenliegenden Frames zu generieren.

Die Fehler in der Tiefenkarte eines Frames nehmen mit der Entfernung zum Schlüssel-Frame zu. Das liegt daran, dass sich die Pixel der Schlüssel-Frames, mit denen die entsprechenden MLAs der Schlüssel-Frames trainiert wurden, bei einer größeren Entfernung zu den Schlüssel-Frame stärker von den Pixeln des aktuellen Frames unterscheiden, als bei einer geringen Entfernung (siehe Abbildung 2.5). Die Qualität der resultierenden Tiefenkarten ist abhängig von der Wahl und Anzahl der Schlüssel-Frames in einem Video. Finden zwischen Schlüssel-Frames schnelle Bewegungen statt, kommt es zu größeren Fehlern in den dazwischenliegenden Frames. Durch die Verwendung von MLAs wird laut [17] die Zeit zur Generierung von Tiefenkarten im Vergleich zur manuellen Erstellung deutlich reduziert.

2.6. Segmentweise Propagierung von Tiefenwerten

Im Algorithmus von [21] erfolgt die semi-automatische Generierung von Tiefenkarten für Videos mittels Propagierung von Tiefenwerten durch eine Übersegmentierung der Frames eines Farbvideos. Im Gegensatz zu [16] erfolgt die Segmentierung in [21] automatisch, ohne Intervention von BenutzerInnen. Der Algorithmus verläuft in den folgenden vier Schritten:

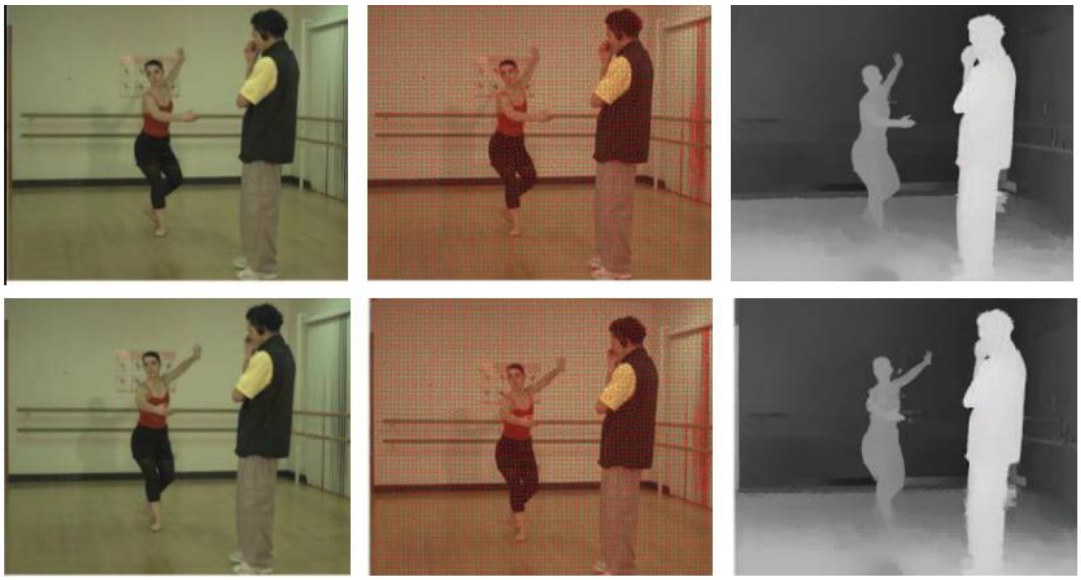
1. Im ersten Schritt generieren BenutzerInnen Tiefenkarten für Schlüssel-Frames eines gegebenen Farbvideos. Diese Generierung der Tiefenkarten erfolgt manuell und mithilfe von Bildbearbeitungs-Programmen (beispielsweise Adobe Photoshop).

2. Die einzelnen Frames werden im zweiten Schritt mithilfe des Segmentierungs-Algorithmus aus [32], in räumlich zusammenhängende Regionen partitioniert. Das Ergebnis dieser Segmentierung ist eine Übersegmentierung, welche aus vielen kleinen Regionen besteht (siehe Abbildung 2.6 b)).

3. Der dritte Schritt stellt einen zeitlichen Bezug zwischen Schlüssel-Frames und Nicht-Schlüssel-Frames her. Dies erfolgt auf Basis der zuvor ermittelten Regionen. Für jede Region eines Nicht-Schlüssel-Frames wird die ähnlichste Region in den Schlüssel-Frames ermittelt.

4. Im vierten und letzten Schritt werden den Regionen von Nicht-Schlüssel-Frames Tiefenwerte ihrer korrespondierenden Regionen in Schlüssel-Frames zugewiesen. Diese Zuweisung wird als Kopiervorgang implementiert. Die entstehende Tiefenkarte des Videos wird anschließend mittels Gauß-Filter [33] geglättet (siehe Abbildung 2.6 c)).

Eine Strategie zur Reduzierung der Laufzeit dieses Algorithmus ist das Generieren von größeren Regionen (weniger Operationen in Schritt 3 und 4). Diese Optimierung erfolgt jedoch auf Kosten der Qualität der resultierenden Tiefenkarten, da diese ebenfalls von der Regionsgröße abhängt [21]. Während kleine Regionen mehr Tiefenabstufungen und -details ermöglichen, sind Tiefenkarten, welche mit größeren Regionen generiert wurden, weniger detailreich. Bei weniger Pixeln pro Region verbessert sich also die Qualität der Ergebnisse bei gleichzeitiger Erhöhung der Laufzeit.



a) b) c)

Abbildung 2.6: *Beispielkonvertierung durch* [21]*: Tiefenwerte eines Schlüssel-Frames (oben) und eines Nicht-Schlüssel-Frame (unten). a) Eingabeframe. b) Übersegmentierung durch* [32]*. Regionsgrenzen sind rot markiert. c) Ergebnis der Propagierung der Tiefenwerte.* [21]

a) b) c) d)

Abbildung 2.7: *Beispielkonvertierung durch* [19]*: a) Frame eines Eingabevideos. b) Generierte Tiefenkarte mit Benutzerinteraktion. c) Frame eines Eingabevideos. d) Generierte Tiefenkarte ohne Benutzerinteraktion.* [19]

2.7. Konvertierung aufgrund von Tiefen-Hinweisen

Die interaktive stereoskopische Video-Konvertierungsmethode aus [19] hebt sich durch ein zusätzliches Modul zur automatischen Schätzung von Tiefeninformation in Videos von den bisher

vorgestellten semi-automatischen 2D-zu-3D-Konvertierungsverfahren ab. Die vorgeschlagenen Tiefenwerte können übernommen, gänzlich verworfen oder korrigiert werden, bevor der eigentliche Propagierungsprozess beginnt. Der Algorithmus, inklusive der erwähnten semi-automatischen Definition der zu propagierenden Tiefenwerte, verläuft grob zusammengefasst in den folgenden drei Schritten:

1. Der erste Schritt umfasst das erwähnte Modul zur automatischen Generierung von Tiefenkarten für Schlüssel-Frames. Die Schätzung der Tiefenkarten erfolgt aufgrund von drei monokularen Tiefen-Hinweisen: Bewegung, Unschärfe und Perspektive. Die Schätzung der Tiefenwerte aufgrund der Bewegung erfolgt unter der Annahme, dass Objekte im Vordergrund tendenziell größere Bewegungen aufweisen als Objekte im Hintergrund. Der Unschärfegrad eines Objekts erlaubt Rückschlüsse über die relative Distanz zu anderen Objekten. Bei Außenaufnahmen, die den Himmel beinhalten, kann die vorhandene Farbinformation Tiefe codieren (grauer Nebel ist im Hintergrund beispielsweise dichter als im Vordergrund). Für eine genauere Beschreibung der automatischen Schätzung der Tiefenwerte siehe [19]. Durch die Kombination der Tiefenkarten der jeweiligen Tiefen-Hinweise können automatisch Tiefenkarten generiert werden.

2. Der zweite Schritt befasst sich mit der interaktiven Verfeinerung der automatisch generierten Tiefenwerte. Ähnlich zu [16] markieren BenutzerInnen zu diesem Zweck Vordergrundobjekte, um diese unter Verwendung der *GrabCut*-Methode [34] aus dem Video zu extrahieren. Die Tiefenwerte der extrahierten Objekte können anschließend erneut geschätzt und manuell korrigiert werden. Das Ergebnis des zweiten Schrittes sind also robuste Tiefenkarten für Schlüssel-Frames, welche zur weiteren Propagierung herangezogen werden können.

3. Im letzten Schritt werden die Tiefenwerte der Schlüssel-Frames auf die restlichen Frames propagiert. Dazu werden die Segmentgrenzen der Objekte (*Konturen*) in den Schlüssel-Frames entsprechend der Objektbewegungen, auf die korrespondierenden Objekte in den folgenden Nicht-Schlüssel-Frames übertragen. Die Konturen werden anschließend unter Verwendung der Methode aus [35] an die Objekte in den Nicht-Schlüssel-Frames angepasst Die Tiefenwerte in den Nicht-Schlüssel-Frames werden, basierend auf den Tiefenwerten in den Schlüssel-Frames, generiert.

In [19] wird für ein Video mit der Auflösung 1280 x 720 eine durchschnittliche Bearbeitungszeit von 8 bis 10 Sekunden pro Frame angegeben. Die Interaktion von BenutzerInnen besteht dabei in der manuellen Nachbearbeitung von falsch berechneten Tiefenwerten für Objekte im Vordergrund (siehe Abbildung 2.7).

2.8. Generierung von Tiefenkarten durch Fusion von Graph Cuts und Random Walks

Der semi-automatische Algorithmus aus [20] zur Generierung von Tiefenkarten für Bilder und Videos kombiniert die beiden Segmentierungs-Algorithmen *Graph Cuts* [36, 37] und *Random Walks* [38]. BenutzerInnen zeichnen zu Beginn Tiefenwerte in Form von Scribbles in Schlüssel-Frames ein. Ausgehend davon werden zwei Tiefenkarten generiert, einmal unter Verwendung des Graph Cuts-Algorithmus und einmal unter Verwendung des Random Walks-Algorithmus.

Der Graph Cuts-Algorithmus löst eine Reihe von binären Segmentierungsproblemen und weist aufgrund der Scribble-Disparitäten Tiefenwerte zu. Pro Scribble entsteht eine binäre Segmentierung in Vordergrund (zum Scribble gehörend) und Hintergrund (nicht zum Scribble gehörend). In diesem Zusammenhang erhalten Vordergrundpixel die Tiefe des aktuellen Scribbles. Zusätzlich zum Segmentierungsergebnis und der Tiefeninformation werden die Kosten (Qualität der Lösung) für die einzelnen Pixel gespeichert. Die Ergebnisse der einzelnen binären Segmentierungen können anschließend zu einer einzigen Tiefenkarte kombiniert werden. Diese Tiefenkarte kann Löcher (Pixel ohne zugewiesene Tiefe) enthalten, welche später gefüllt werden müssen (beispielsweise mittels Interpolation). Diese Situation tritt auf, wenn ein Pixel in keiner der binären Segmentierungsprobleme als Vordergrund erkannt wurde. Umgekehrt kann ein Pixel mehrere Male als Vordergrund erkannt worden sein. In diesem Fall wird der Tiefenwert mit den geringsten Kosten (bessere Lösung) gewählt. Die entstehenden Tiefenkarten enthalten harte Grenzen zwischen Objekten. Innerhalb der Objekte gibt es keine Tiefenvariationen, was dazu führt, dass die Objekte in der 3D-Darstellung aus dem Bild ausgeschnitten wirken (siehe Abbildung 2.8 d)).

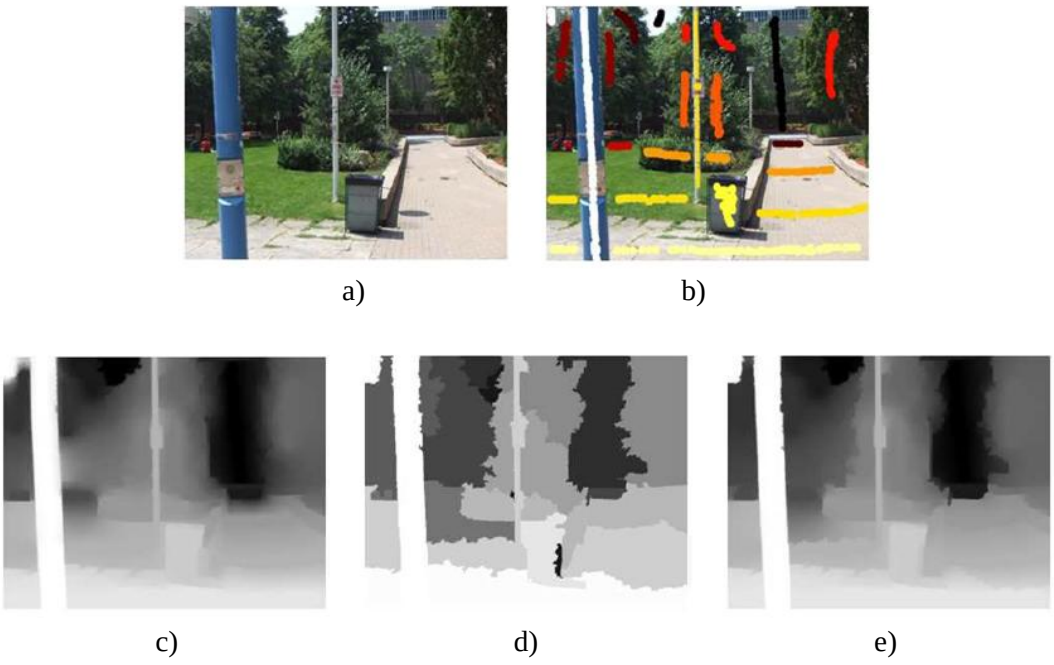


Abbildung 2.8: *Beispielkonvertierung durch* [20]*: a) Eingabeframe. b) Eingabeframe mit eingezeichneten Scribbles. c) Resultierende Tiefenkarte unter Verwendung des Random Walks-Algorithmus* [38]*. d) Resultierende Tiefenkarte unter Verwendung des Graph Cut-Algorithmus* [37]*. e) Kombination der Tiefenkarten aus c) und d).* [20]

Der Random Walks-Algorithmus ist ein Optimierungs-Algorithmus, der zur Propagierung von Informationen wie Tiefenwerten eingesetzt werden kann. Bei spärlich gegeben Tiefenwerten findet diese Methode, unter anderem durch Lösung eines linearen Gleichungssystems (siehe [20, 38] für Details), einen optimalen Tiefenwert für jedes Pixel. Diese optimierten Tiefenwerte sind nicht auf die ursprünglich gegebenen Tiefenwerte beschränkt, sondern können variieren. Während diese Eigenschaft innerhalb von Objekten erwünscht ist (beispielsweise bei schrägen Flächen), kann diese an Objektgrenzen zu unerwünschten, gemischten Tiefen führen (siehe Abbildung 2.8 c)). Um Tiefenkarten in letztgenannten Bereichen zu verbessern, werden in der Random Walks-Optimierung

nicht nur die Tiefen der Scribbles, sondern auch die mittels Gaph Cuts generierte Tiefenkarte berücksichtigt. Diese Kombination aus Graph Cuts und Random Walks resultiert in Tiefenkarten mit kontinuierlichen Tiefenübergängen und klaren Objektgrenzen (siehe Abbildung 2.8 e)).

Für Videos werden auf diese Weise Tiefenkarten für Schlüssel-Frames generiert. Die Tiefenwerte werden anschließend mithilfe eines von zwei Verfahren auf die restlichen Frames des Videos propagiert. Im ersten Verfahren markieren BenutzerInnen Objekte im ersten Frame, für welche sie Tiefenwerte propagieren wollen. Diese Objekte werden anschließend über die folgenden Frames hinweg mithilfe eines Tracking-Verfahrens verfolgt und Tiefenwerte entlang des Bewegungspfades propagiert. Der Nachteil dieses Verfahrens liegt im hohen Rechenaufwand des Tracking-Algorithmus. Aus diesem Grund wird in [20] ein alternatives Verfahren vorgeschlagen, welches auf Optical Flow-Informationen basiert. Die Tiefenwerte der Schlüssel-Frames werden dabei, abhängig von Bewegungsvektoren, auf die nachfolgenden Frames kopiert.

Der Algorithmus zur Generierung von Tiefenkarten wird in [20] im Zuge eines Vergleichs mit dem Algorithmus aus [1] evaluiert. Laut [20] werden dabei eine geringere Laufzeit und qualitativ hochwertigere 3D-Ergebnisse erzielt.

a)

b)

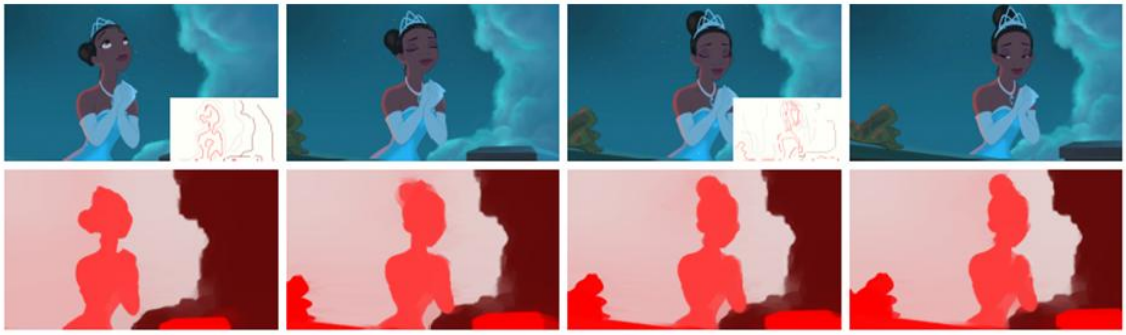


Abbildung 2.9: *Beispielpropagierung von Farben durch* [18]*: a) Eingabeframes mit Scribbles in Schlüssel-Frames (rechter Bildrand). Die Farbe der Scribbles entspricht der neuen Farbe im Ergebnisvideo. b) Ergebnis der Propagierung der Scribbles: Umfärbung des Eingabevideos.* [18]

2.9. Tiefenpropagierung durch Kostenfilterung

Bei der Propagierung durch Kostenfilterung [18, 39, 40] handelt es sich um eine einfache und effiziente Methode zur Lösung von bildbasierten Optimierung-Problemen, wie der Propagierung von Scribble-Tiefen [18] oder interaktiver Segmentierung [39, 40]. Im Speziellen können Optimierungsprobleme mit folgender Energiefunktion minimiert werden [18]:

$$E(J) = E_{data}(J) + \lambda E_{smooth}(J) \tag{2.1}$$

J ist hier die unbekannte Lösung (beispielsweise die gesuchte Tiefenkarte), welche durch die Minimierung der Energiefunktion *E* ermittelt werden soll. E_{data} ist der Daten-Term, welcher abhängig von der konkreten Anwendung variiert. Im Kontext der Tiefenpropagierung ermittelt dieser für jedes Pixel die Kosten, welche die Zuweisung zu einem bestimmten Pixel verursacht. Diese Kosten werden in einem sogenannten Kostenvolumen gespeichert [18]. Der Glattheits-Term E_{smooth} gewährleistet, dass benachbarte Pixel, welche ähnlich zueinander sind, ähnliche Tiefenwerte aufweisen. Um die

aufwändige globale Optimierung [1, 15] zu vermeiden, werden E_{data} und E_{smooth} aufgeteilt und getrennt voneinander gelöst. Nach Generierung des Kostenvolumens, wird der Glattheits-Term E_{smooth} durch die Anwendung eines kantenerhaltenden Filters (beispielsweise Bilateral Filter [41] oder Guided Filter [23]) umgesetzt. Die Glättung der Kosten farblich ähnlicher Nachbarpixel verhindert abrupte Änderungen von Zuweisungen (Tiefensprünge) in farblich ähnlichen Regionen. Bei Anwendung der Kostenfilterung auf Videos wird der verwendete Filter um die Filterung der zeitlichen Dimension erweitert [18, 39]. Um die zeitliche Kohärenz der Ergebnisse zu verbessern, erfolgt die Filterung in [18] entlang vorher ermittelter Bewegungspfade. Die Auswahl der finalen Zuweisung der Pixel, beispielsweise zu Tiefen, erfolgt anhand der minimalen Kosten pro Pixel.

In [18] wird unter Verwendung eines PCs mit einer Intel Core i7 920 CPU mit 2.67 GHz und 12 GB RAM für acht Frames eines Videos mit der Auflösung 1280 x 720, eine durchschnittliche Laufzeit von 0.625 Sekunden angegeben. In [39] wurden für eine optimierte Implementierung unter Verwendung eines PCs mit einer Intel Core 2 Quad CPU mit 2.4 GHZ und einer GeForce GTX 480 Grafikkarte, eine Laufzeit von 0.004 Sekunden pro Frame und Filterdurchlauf für ein Video mit 620 x 360 Pixeln gemessen.

3. Grundlagen von CUDA

In diesem Kapitel werden die Grundlagen paralleler Grafikkartenprogrammierung mittels NVIDIA CUDA (*Compute Unified Device Architecture*) behandelt. Zu Beginn wird erläutert, warum es sinnvoll ist, gewisse Prozeduren parallel auf der Grafikkarte auszuführen und was der Vorteil gegenüber einer seriellen CPU-Implementierung ist. Anschließend wird die grundlegende Architektur einer CUDA-Grafikkarte beschrieben und welche Schritte notwendig sind, um Prozeduren auf der GPU ausführen zu können. Im letzten Abschnitt wird ein Design-Zyklus zur Beschleunigung von einem bestehenden Programmcode beschrieben. Sequentielle Programmteile werden dabei sukzessive auf die GPU verlagert, wo sie parallelisiert werden. Interessierte LeserInnen sind für weiterführende Information über CUDA auf [24, 42, 43, 44] verwiesen.

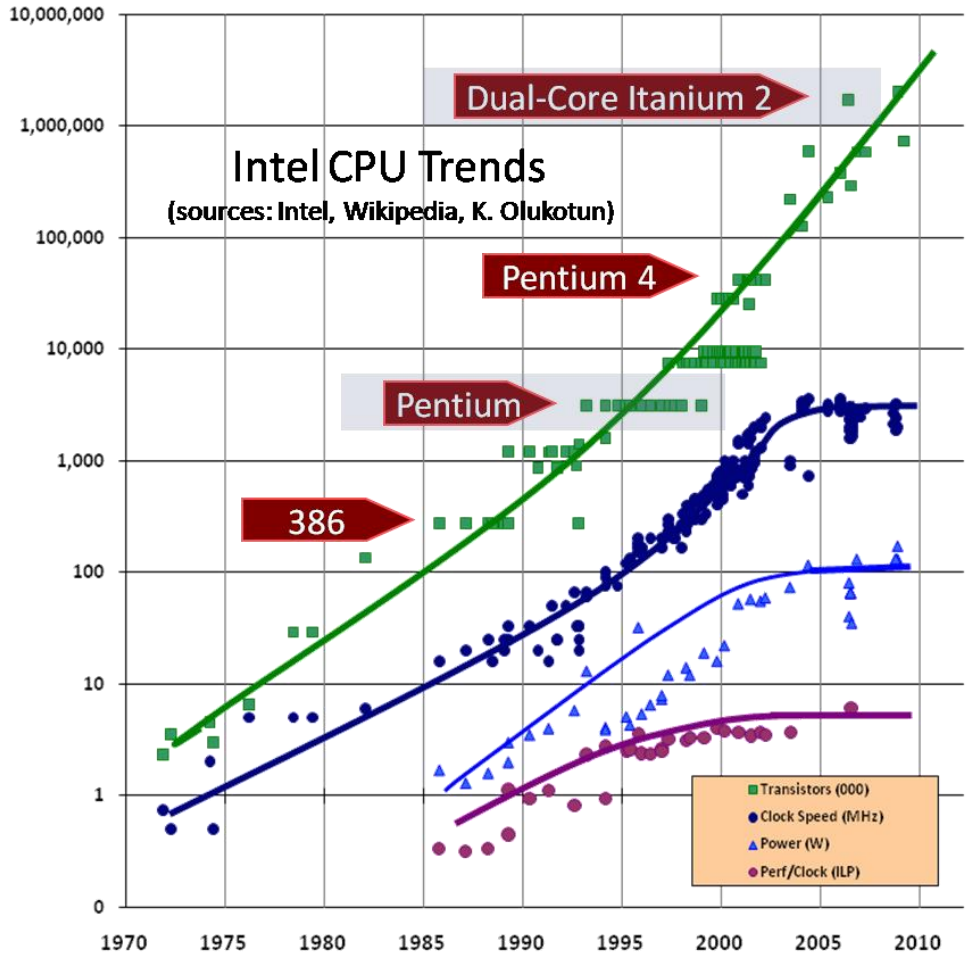


Abbildung 3.1: *Die Anzahl der Transistoren auf einem Chip nahm seit 1970 stetig zu und verdoppelte sich alle zwei Jahre. Aufgrund von physikalischen Einschränkungen begann die Steigerung der Prozessorleistung jedoch ab ca. 2003 zu stagnieren.* [45]

3.1. Motivation

Moore's Law [46] besagt, dass sich die Anzahl der Transistoren auf integrierten Schaltkreisen alle zwei Jahre verdoppelt und somit auch Prozessoren entsprechend leistungsfähiger werden. Auch wenn daraus nicht geschlossen werden kann, dass sich die Leistungsfähigkeit von Prozessoren alle zwei

Jahre verdoppelt, verbessert sich diese dennoch merklich mit jeder Generation (siehe Abbildung 1.1). Die schnelle Verbesserung der Prozessorleistung führte dazu, dass Software, welche traditionellerweise als sequentielles Programm entwickelt wurde, mit jeder Prozessorgeneration schneller ausgeführt werden konnte. Das lag daran, dass diese sequentiellen Programme von einem einzigen Prozessor schrittweise abgearbeitet wurden. Eine Steigerung der Prozessorleistung führte automatisch dazu, dass die Programme schneller liefen. Dieser Trend nahm aber seit 2003 immer mehr ab. Man stieß bei der Entwicklung von integrierten Schaltkreisen langsam an die physikalischen Grenzen. Zu hoher Energieverbrauch und daraus folgende Probleme bei der Wärmeleitung beschränkten die Steigerung der Taktfrequenz und die mögliche Signalleitung während eines Taktes. Die Leistungsfähigkeit von einzelnen Prozessoren stieg im Vergleich zu Vorgängermodellen nur noch unwesentlich. Wie in Abbildung 3.1 zu sehen ist, nahm die Anzahl der Transitoren zwar regelmäßig zu, die Prozessorleistung verbesserte sich aber immer weniger. Prozessorhersteller umgehen dieses Problem durch die Entwicklung von Prozessoren mit mehreren parallel laufenden Kernen (*Cores*). Die Leistungssteigerung wird nicht mehr durch die Verbesserung von einzelnen Prozessoren, sondern durch die parallele Verarbeitung durch mehrere Prozessorcores erreicht. Mehrkernsysteme gehören heute bereits zum Standard. In diesem Zusammenhang stehen SoftwareentwicklerInnen vor dem Problem, dass ihre Programme auf den zukünftigen Generationen von Prozessoren nicht mehr automatisch schneller ausgeführt werden können, da die Leistung von einzelnen Prozessorcores nur unwesentlich wächst. Für sequentielle Programme kann somit nur ein Bruchteil der vorhandenen Prozessorleistung genutzt werden. Aus diesem Grund ist es notwendig, bei der Programmentwicklung die Mehrkernsysteme zu berücksichtigen. Dazu werden Algorithmen parallelisiert und auf mehrere Cores verteilt. [44]

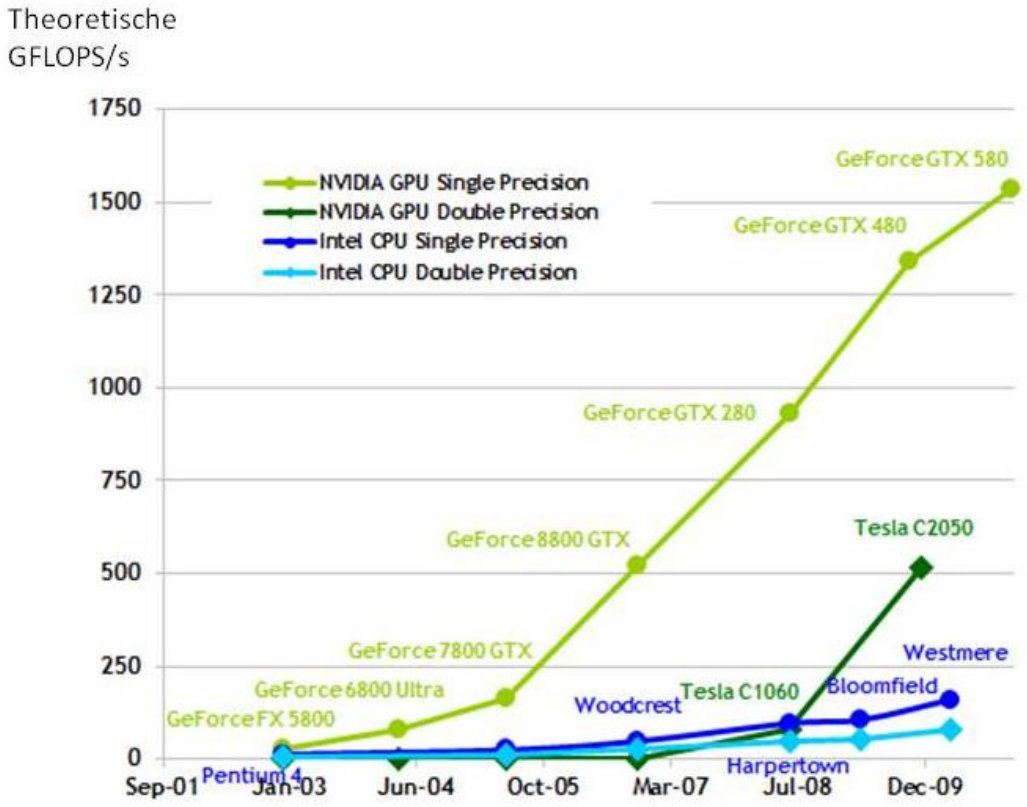


Abbildung 3.2: *Vergleich der Rechenleistung von CPU und GPU. Es werden die theoretisch möglichen Gleitkommaoperationen pro Sekunde (FLOP/s) verglichen.* [24]

Seit 2003 schlägt man bei der Entwicklung von Mehrkernsystemen zwei unterschiedliche Richtungen ein [44]: *Multi-core* und *Many-core*. Multi-core-Systeme haben das Ziel, die Ausführungsgeschwindigkeit von sequentiellen Programmen durch die Verlagerung auf mehrere Cores zu unterstützen. Vier- oder Acht-Core-Prozessoren sind keine Seltenheit mehr. Ein aktuelles Beispiel stellt der Intel® Core™ i7 mit vier Prozessorcores dar. Many-core-Systeme haben das Ziel, parallele Anwendungen möglichst effizient abzuarbeiten. Ein wichtiger Vertreter dieser Kategorie sind die Grafikkarten (GPUs). Grafikkarten besitzen eine Vielzahl von Cores und setzen in großem Maß auf Parallelität. Eine NVIDIA GeForce GTX 260 verfügt zum Beispiel über 192 Cores und eine NVIDIA GeForce GTX 670 über 1344 Cores [47]. So können tausende Punkte eines komplexen geometrischen Objekts gleichzeitig gerendert werden. Erst dadurch ist es möglich, die aufwändige Grafik in Spielen in Echtzeit zu berechnen. Grafikkartenhersteller erkannten, dass diese enorme Rechenleistung auch für Anwendungen außerhalb des Rendering-Bereichs genutzt werden kann. Die Architektur von Grafikkarten wurde dahingehend angepasst, dass beliebige Funktionen auf der GPU ausgeführt werden können. Grafikkarten wurden damit universal programmierbar.

Ein Beispiel für universal programmierbare GPUs (*General Purpose GPUs)* [48] stellen die CUDA-fähigen Grafikkarten von NVIDIA dar. Ein weiterer großer Anbieter von universal programmierbaren GPUs ist ATI/AMD. Grafikkarten unterschiedlicher Hersteller besitzen unterschiedliche Architekturen. Das bedeutet, dass ein für CUDA-Grafikkarten optimiertes Programm nur suboptimal auf ATI/AMD GPUs läuft und umgekehrt. In dieser Diplomarbeit wird die CUDA-Architektur verwendet. Abbildung 3.2 zeigt einen Vergleich zwischen der Leistung von CPU und GPU. Es wird verglichen, wie viele Gleitkommaoperationen pro Sekunde durchgeführt werden können.

Für die Entwicklung von parallelen Algorithmen wird bei CUDA-fähigen Grafikkarten *C for CUDA* verwendet [43]. Dabei handelt es sich um einen C-Dialekt, welcher um Funktionen zur parallelen Implementierung auf der GPU erweitert wurde. Dabei ist anzumerken, dass Programme, welche mit *C for CUDA* entwickelt wurden, ausschließlich auf NVIDIA Grafikkarten lauffähig sind. In dieser Diplomarbeit wurde *C for CUDA* für die Entwicklung verwendet. Alternativ dazu kann zur Entwicklung auf der GPU auch *OpenCL* [49] verwendet werden. Programme, welche mit OpenCL entwickelt wurden, sind sowohl auf NVIDIA, als auch auf ATI/AMD Grafikkarten lauffähig. Aufgrund der unterschiedlichen Architektur ist ein für NVIDIA optimiertes OpenCL-Programm auf einer ATI/ADM Grafikkarte aber nur suboptimal lauffähig und umgekehrt.

3.2. Die CUDA-Architektur

In diesem Abschnitt wird der grundlegende Aufbau der CUDA-Architektur beschrieben. Um nicht zu sehr ins Detail zu gehen, wird nur auf jene Aspekte eingegangen, welche für die Implementierungen in dieser Diplomarbeit von Relevanz sind. Eine detaillierte Beschreibung der CUDA-Architektur kann in den CUDA-Dokumentationen [24, 42, 50] gefunden werden. Diese Diskussion wird durch Codesegmente, in der Programmiersprache *C for CUDA*, unterstützt.

Bei CUDA handelt es sich um ein Paradigma zur parallelen Programmierung, welches 2007 von NVIDIA veröffentlicht wurde. Es dient dazu, allgemeine Anwendungen für die GPU zu entwickeln und diese parallel auszuführen, indem die Programmabläufe auf die Cores der GPU verteilt werden. Abbildung 3.3 zeigt den grundlegenden Aufbau einer CUDA-Grafikkarte. [26, 44, 48]

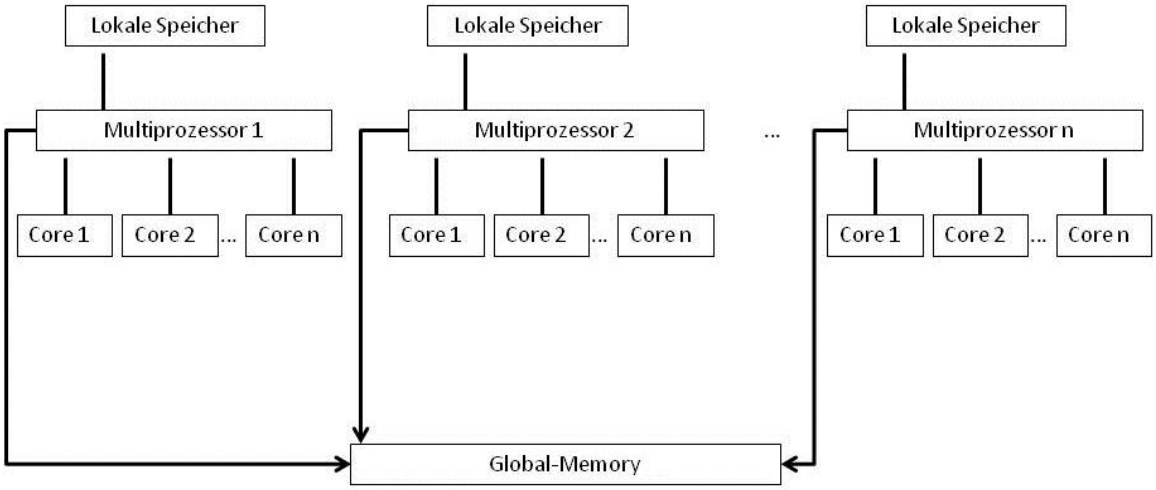


Abbildung 3.3: *Grundlegender Aufbau einer CUDA fähigen Grafikkarte. Nach* [26]

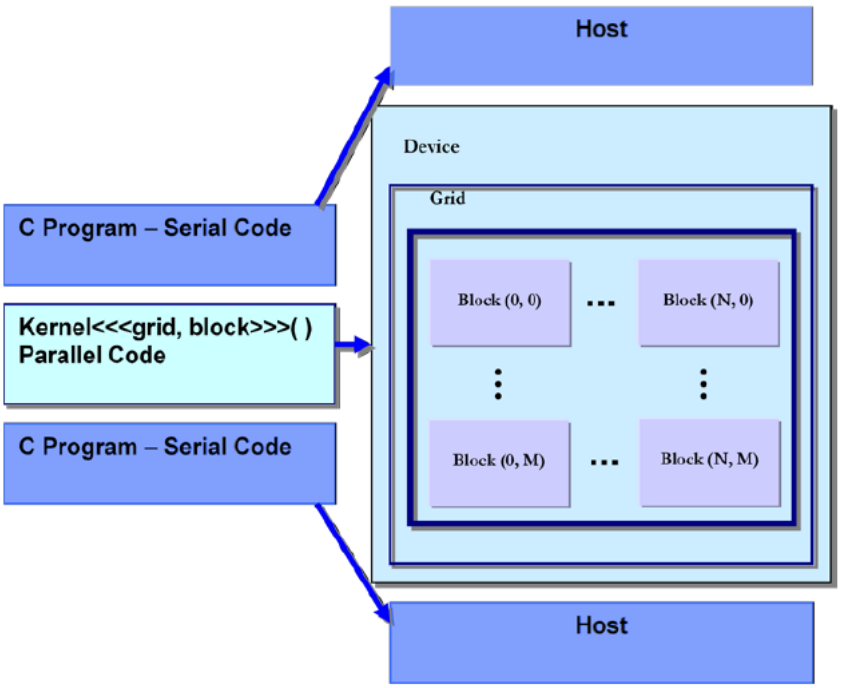


Abbildung 3.4: *Programmfluss eines CUDA-Programms. Serielle Code-Abschnitte werden auf der CPU (Host) ausgeführt, parallele Abschnitte auf der GPU (Kernel).* [48]

Eine CUDA-Grafikkarte besteht aus mehreren parallelen Multiprozessoren. Jeder dieser Multiprozessoren enthält wiederum eine bestimmte Anzahl von Cores. Um eine hohe Effizienz zu erreichen, sollten möglichst alle Cores parallel eingesetzt werden. Auf der Grafikkarte sind zudem unterschiedliche Speicherbausteine enthalten: 1. Hauptspeicher (*Global-Memory*) und 2. lokale Speicherbereiche. Den größten Speicher enthält der Global-Memory. Eine GeForce GTX 260 enthält beispielsweise 768MB Global-Memory, eine GeForce GTX 660ti 2048MB [47]. Alle Threads können auf den Global-Memory zugreifen. Die Zugriffsgeschwindigkeit ist relativ langsam. Außerdem kann es zu Kollisionen kommen, wenn mehrere Threads gleichzeitig auf den Global-Memory zugreifen.

Neben dem Global-Memory hat jeder Multiprozessor lokale Speicherbereiche. Auf die Eigenschaften der unterschiedlichen Speicherbereiche wird in Abschnitt 3.3 genauer eingegangen.

Ein typisches CUDA-Programm folgt einem bestimmten Ablauf, bestehend aus seriellen und parallelen Programmteilen. Abbildung 3.4 illustriert den typischen Programmfluss eines CUDA-Programms. Dabei wird zwischen dem seriellen Programmteil, dem *Host*, und dem parallelen Teil, dem *Kernel*, unterschieden. Der Host-Code wird seriell auf der CPU ausgeführt. Es handelt sich dabei um gewöhnliche C-Funktionen. Beim Kernel handelt es sich um jenen Programmteil, der auf der Grafikkarte ausgeführt wird.

Der Kernel wird wie folgt definiert:

```
__global__ void Kernel(Parameterliste){ … }
```

Das Schlüsselwort *__global__* gibt an, dass es sich um eine Kernel Funktion handelt, welche auf der GPU ausgeführt werden soll. Da Kernelfunktionen keine Werte an den Host zurückgeben können, ist der Rückgabewert der Funktion *void*. Im Kernel können auch Funktionen auf der GPU ausgeführt werden. Diese müssen mit dem Schlüsselwort *__device__* gekennzeichnet werden. Dabei handelt es sich um Funktionen, die nur auf der GPU ausgeführt werden können und auch nur aus dem Kernel aufgerufen werden können. Der Aufruf erfolgt wie bei einer gewöhnlichen C-Funktion. Es können Parameter übergeben und Werte an den Kernel zurückgegeben werden. Device Funktionen werden wie folgt definiert:

```
__device__ Rückgabetyp DeviceFunktion(Parameterliste){ … return Wert;}
```

Der Kernel wird folgendermaßen aufgerufen:

```
Kernel <<1,1>>(Parameter);
```

Der erste Wert in der spitzen Klammer definiert die Anzahl der Instanzen eines Kernels, die gleichzeitig ausgeführt werden sollen. Der Ausdruck *<<1,1>>* bedeutet, dass nur eine einzige Instanz des Kernels erstellt wird. Es handelt sich in diesem Fall um ein sequentielles Programm, welches auf einem einzigen Core eines Multiprozessors ausgeführt wird. Die Stärke der GPU liegt nicht in der Leistungsfähigkeit der einzelnen Multiprozessoren (diese ist wesentlich geringer als die Leistung eines Prozessorcores der CPU), sondern in der Vielzahl an gleichzeitig laufenden Programmteilen. Da serielle Programmteile auf der CPU schneller ausgeführt werden können, sollten nur parallele Programmteile auf der GPU ausgeführt werden und sequentielle Programmteile weiterhin auf der CPU ablaufen. Um einen Kernel parallel abarbeiten zu können, müssen mehrere Instanzen davon erzeugt werden. Wird der Kernel beispielsweise mit *<<256,1>>* aufgerufen, bedeutet das, dass 256 Instanzen des Kernels erzeugt werden, welche gleichzeitig auf unterschiedlichen Cores ausgeführt werden. Eine Kernel-Instanz wird als *Block* bezeichnet. Es ist darauf zu achten, dass die Anzahl der Blöcke die Grenze von 65.535 nicht überschreitet. Die Überschreitung dieses Limits würde zu Fehlern führen [43]. CUDA ermöglicht es, Gruppen von Blöcken in zwei Dimensionen zu erstellen. Es kann für zweidimensionale Problemstellungen, wie beispielsweise das Rechnen mit Matrizen oder Bildbearbeitungs-Algorithmen von Vorteil sein, Blöcke zweidimensional darzustellen, um unnötigen Aufwand beim Indizieren zu vermeiden. Es sei an dieser Stelle aber darauf hingewiesen, dass alle Problemstellungen auch im eindimensionalen Bereich gelöst werden können. Im zweidimensionalen Bereich werden die Blöcke in einem Gitter (*Grid*) geordnet. Ein Kernel-Aufruf mit dem Ausdruck *<<<4,2>,1>>* erzeugt ein Grid mit vier Spalten und zwei Zeilen und enthält somit acht Blöcke.

Abbildung 3.5 veranschaulicht den Aufbau eines Grids und die Zuweisung der Blöcke an die Cores der Multiprozessoren. Zuerst werden vier Blöcke ohne bestimmte Reihenfolge auf die freien Cores der beiden Multiprozessoren verteilt. Dort werden sie parallel abgearbeitet. Nachdem ein Core mit der Bearbeitung eines Blocks fertig ist, wird ihm der nächste Block übergeben. Das geschieht solange, bis alle Blöcke des Grids bearbeitet wurden. [26]

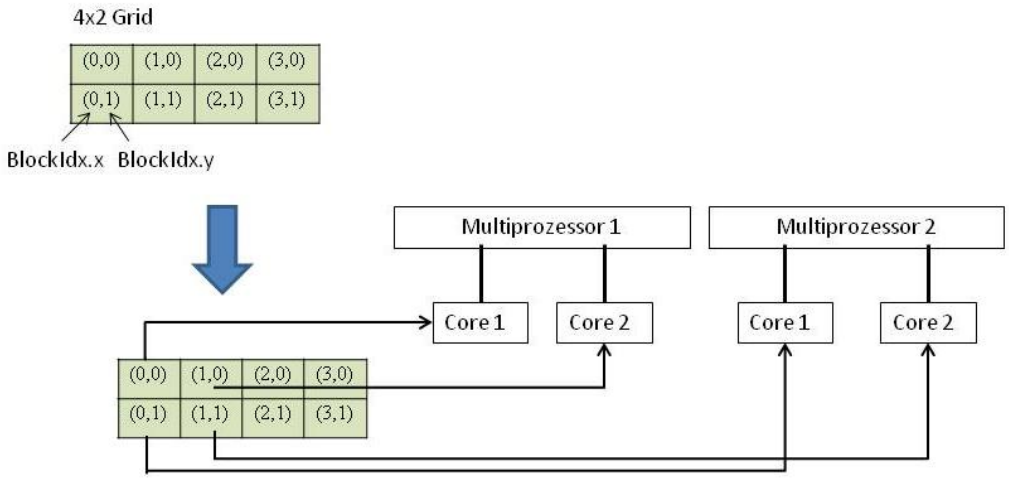


Abbildung 3.5: *Aufbau eines 4x2 Grids in CUDA. Das Grid enthält acht Blöcke, welche mittels BlockIdx.x und BlockIdx.y indiziert werden. Zuerst werden vier beliebige Blöcke auf die Cores verteilt und dort abgearbeitet. Sobald ein Core frei ist, wird ihm der nächste Block übergeben, solange bis alle Blöcke bearbeitet wurden. Nach* [26]

Die CUDA-Architektur stellt vordefinierte Variablen zu Verfügung, um auf die Blöcke zugreifen zu können. Auf einen einzelnen Block kann mithilfe der Variablen *blockIdx.x* und *blockIdx.y* zugegriffen werden, was seiner Position innerhalb des Grids entspricht [43]. In Abbildung 3.5 hat beispielsweise das erste Element der zweiten Zeile im Grid die Indizes blockIdx.x = 0 und blockIdx.y = 1. Ein Block kann demnach folgendermaßen identifiziert werden [43]:

$$bid = blockIdx.x + blockIdx.y * gridDim.x \tag{3.1}$$

gridDim gibt die Dimension des Grids an. Im eindimensionalen Fall genügt *blockIdx.x* zur eindeutigen Identifikation. Da jeder Block eine Instanz desselben Kernels enthält, ist es notwendig zu wissen, an welcher Position in der Datenstruktur man sich gerade befindet.

Der zweite Parameter in den spitzen Klammern beim Aufruf des Kernels steht für die Anzahl an Threads, die pro Block generiert werden sollen. Bei einem CUDA-Thread handelt es sich um eine ausführbare Instanz des Kernels. Auf einem Core können mehrere Threads gleichzeitig ausgeführt werden. Der Ausdruck *<<2,1>>* gibt an, dass zwei Blöcke mit jeweils einem Thread generiert werden. Das bedeutet, dass ein Core jeweils genau eine Instanz des Kernels ausführt. Erhöht man den zweiten Wert, werden entsprechend viele Threads gleichzeitig auf einem Core ausgeführt. Ein Set von Threads, die auf einem Core laufen, wird als *Thread-Block* bezeichnet [48]. Ein Thread wird mithilfe der Variablen *threadIdx* identifiziert. Die Position des aktuellen Threads kann im eindimensionalen Fall mit

$$tid = threadIdx.x + blockIdx.x * blockDim.x \tag{3.2}$$

bestimmt werden [43]. Die Variable *blockDim* enthält die Anzahl der Threads entlang jeder Dimension des Blocks. *blockDim* ist dreidimensional, das heißt, CUDA ermöglicht es, ein zweidimensionales Gitter von Blöcken zu erstellen, wobei jeder Block ein dreidimensionales Array an Threads enthalten kann. Theoretisch wird somit eine Indizierung in fünf Dimensionen ermöglicht (ab Compute Capability 2.0 sogar in sechs Dimensionen, da das Grid ab dann in drei Dimensionen erzeugt werden kann), was aber in der Praxis selten eingesetzt wird [43]. Abbildung 3.6 veranschaulicht den Aufbau des Grids unter der Verwendung von mehreren Threads.

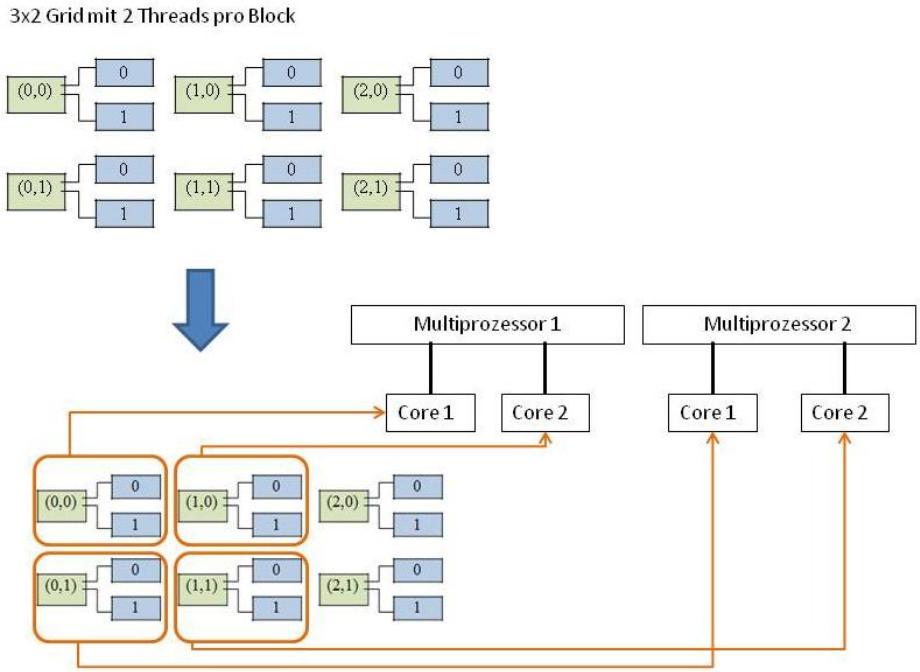


Abbildung 3.6: *3x2 Grid mit zwei Threads pro Block. Die Blöcke werden auf die freien Cores aufgeteilt. Auf jedem Core laufen zwei Threads gleichzeitig. Nach* [26]

Innerhalb eines Thread-Blocks werden jeweils 32 Threads zu einem sogenannten *Warp* zusammengefasst. Alle 32 Threads eines Warps werden gleichzeitig auf einem Core ausgeführt. Die Threads werden nacheinander, entsprechend ihrer ID, zusammengefasst. Das heißt, die Threads innerhalb eines Warps liegen nebeneinander. Wenn in dem Kernel ein Speicherzugriff auf den Global-Memory erfolgt, bedeutet dies, dass 32 Threads gleichzeitig auf den Speicher zugreifen. Damit diese Speicherzugriffe optimal durchgeführt werden können, ist es wichtig zu wissen, wie der Global-Memory aufgebaut ist (Siehe Abbildung 3.7) [26].

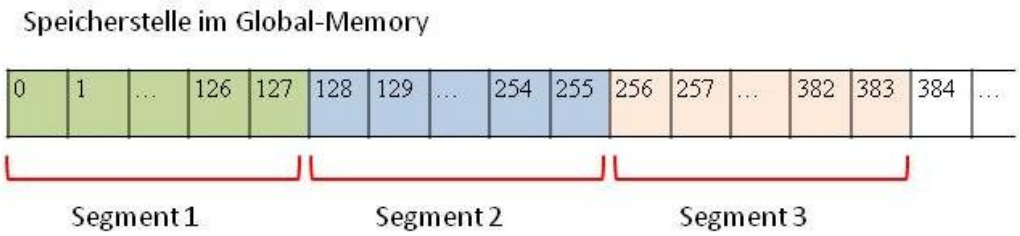


Abbildung 3.7: *Unterteilung der Speicherbereiche im Global-Memory in 128 Byte große Segmente.*

Der Speicherbereich im Global-Memory wird in gleich große Segmente mit der Größe von 128 Byte unterteilt. Auf Speicherbereiche innerhalb eines Segments kann gleichzeitig zugegriffen werden. Auf Speicherbereiche, die in unterschiedlichen Segmenten liegen, kann allerdings nur seriell zugegriffen werden. Es sollte also darauf geachtet werden, dass im Falle eines Speicherzugriffs möglichst alle Threads eines Warps auf Speicherbereiche innerhalb desselben Segments zugreifen, da in diesem Fall alle Speicherzugriffe parallel erfolgen können. Speicherzugriffe innerhalb eines Warps können wie folgt aussehen:

- *Alle Threads greifen auf dieselbe Speicherstelle zu:*
  In diesem Fall wird nur ein einziger Speicherzugriff ausgeführt. Das Ergebnis wird dann an alle Threads geschickt. Im Falle eines Lesezugriffs ist dieser Fall ideal, da nur ein einziges Mal gelesen werden muss. Wollen aber mehrere Threads auf dieselbe Speicherstelle schreiben, kommt es zu Konflikten zwischen den Threads. Schreiben mehrere Threads auf dieselbe Speicherstelle, kann nur ein Thread diesen Schreibzugriff durchführen, wobei nicht definiert ist, welcher Thread das ist. [24]

- *Alle Threads greifen auf Speicherstellen innerhalb desselben Segments zu:*
  Alle Speicherzugriffe können parallel erfolgen.

- *Die Threads greifen auf Speicherstellen innerhalb von zwei Segmenten zu:*
  Die Zugriffe auf jeweils ein Segment können parallel erfolgen. Zugriffe auf zwei unterschiedliche Segmente erfolgen seriell. Es werden zuerst die Zugriffe auf das erste und anschließend die Zugriffe auf das zweite Segment durchgeführt.

- *Die Threads greifen auf mehr als zwei Segmente zu:*
  Die Zugriffe innerhalb eines Segments erfolgen parallel. Die Segmente werden dann wieder seriell bearbeitet. Im schlechtesten Fall erfolgen alle Zugriffe eines Warps in unterschiedliche Segmente, was einer kompletten Serialisierung der Speicherzugriffe entspricht.

Im Fall, dass ein Warp still steht, weil er beispielsweise auf einen Speicherzugriff warten muss, kann inzwischen ein anderer Warp ausgeführt werden, der keinen Speicherzugriff benötigt. Es ist besser, eine höhere Anzahl an Threads pro Block zu generieren, damit möglichst wenig Leistung durch stillstehende Warps verloren geht. Die maximale Anzahl der Threads, die pro Block generiert werden können, hängt von der Compute Capability der Grafikkarte ab [24]. Auf Grafikkarten mit Compute Capability 1.0 bis 1.3 können 512 Threads pro Block generiert werden. Ab Compute Capability 2.0 können 1024 Threads pro Block erzeugt werden. Will man also 1024 Threads gleichmäßig auf acht Cores verteilen, müssen 128 Threads generiert werden, womit vier Warps (mit je 32 Threads) pro Core laufen. [26]

3.3. Speicherbereiche

Wie bereits erwähnt, gibt es neben dem Global-Memory noch weitere Speicherbereiche auf einer CUDA-fähigen Grafikkarte. Abbildung 3.8 visualisiert den Aufbau der Speicherbausteine und zeigt, auf welche Elemente CPU und GPU zugreifen können. Es wird zwischen zwei Arten von Speicherbereichen unterschieden: 1. globale und 2. lokale Speicher. Zu den globalen Speichern zählen *Global-Memory*, *Constant-Memory* und *Texture-Memory*. Bei diesen drei Speichern handelt es sich um denselben physikalischen Speicher, nämlich den Hauptspeicher der Grafikkarte. Die Speicherbereiche unterscheiden sich lediglich durch die Caching-Algorithmen und die

Zugriffsmodelle. Zu den lokalen Speichern zählen *Register-Memory*, *Local-Memory* und *Shared-Memory*. [24]

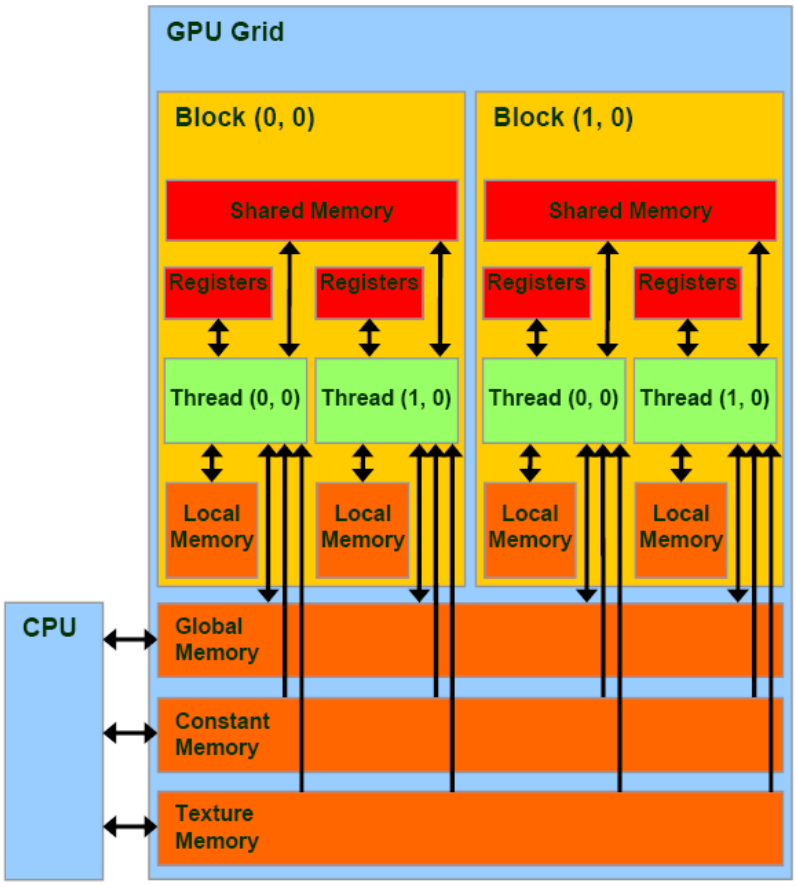


Abbildung 3.8: *Speicherbereiche einer CUDA-fähigen Grafikkarte.* [51]

Global-Memory

Der Global-Memory stellt den Hauptspeicher der Grafikkarte dar. Er hat von allen Speicherbausteinen die größte Kapazität. Wie in Abbildung 3.8 zu sehen ist, kann sowohl die CPU als auch die GPU auf den Global-Memory zugreifen. Er stellt somit die Verbindung zwischen CPU und GPU her. Alle Daten, die auf der GPU bearbeitet werden, müssen vorher vom Hauptspeicher des PCs auf den Global-Memory der GPU kopiert werden. Es handelt sich um einen globalen Speicherbereich, das heißt, jeder Thread kann jederzeit darauf zugreifen. Der Nachteil liegt in der langsamen Zugriffszeit, welche darauf zurückzuführen ist, dass beim Zugriff kein Caching möglich ist. An dieser Stelle sei angemerkt, dass beim Lesezugriff ab Compute Capability 2.0 Caching möglich ist. Sobald eine Schreiboperation auf den Global-Memory durchgeführt wird, sind die Daten im Cache jedoch nicht mehr gültig. Wie bereits erwähnt, muss bei Speicherzugriffen durch Threads auf die Segmentierung des Global-Memory Rücksicht genommen werden. Speicherbereiche, welche im Global-Memory reserviert wurden, bleiben standardmäßig über die gesamte Laufzeit der Anwendung bestehen. Werden sie nicht mehr benötigt, müssen sie manuell freigegeben werden.

Constant-Memory

Der Constant-Memory stellt einen Teilbereich des Global-Memory dar. Er dient dazu, Variablen mit einem konstanten Wert zu definieren. Im Gegensatz zum Global-Memory kann nur die CPU auf den Constant-Memory schreiben. Deshalb müssen konstante Variablen vom Host definiert und in den Constant-Memory kopiert werden. Kernel-Funktionen können diese ausschließlich auslesen. Daten werden hier global definiert, das heißt, sie sind von jedem Thread aus zugänglich. Die Lebensdauer der Daten läuft über die gesamte Laufzeit der Anwendung. Werden die Konstanten nicht mehr benötigt, müssen sie manuell freigegeben werden. Der Vorteil des Constant-Memory liegt darin, dass Caching bei Leseoperationen möglich ist. Der Nachteil liegt in der geringen Speichergröße von nur 64KB.

Texture-Memory

Der Texture-Memory stellt einen Teilbereich des Global-Memory dar. Genau wie der Constant-Memory, können Kernel-Funktionen nur aus dem Texture-Memory lesen, aber nicht auf ihn schreiben. Datenstrukturen, die im Texture-Memory verwendet werden sollen, müssen vom Host an diesen gebunden werden. Der Kernel kann auf die so erstellten Texturen mithilfe von *Texture Fetches* [24] zugreifen. Der Vorteil liegt darin, dass Leseoperationen auf Daten, die räumlich nahe beieinander liegen, schneller erfolgen können als beim Zugriff auf den Global-Memory. Das liegt daran, dass diese pro Multiprozessor in 8KB große Blöcke gecached werden können. Texturen können ein-, zwei- oder dreidimensional erstellt werden. Die Verwendung des Texture-Memory kann zu Performancesteigerungen führen, wenn auf Datenstrukturen nur lesend zugegriffen werden soll, da Texture Fetches schneller durchgeführt werden können als Leseoperationen auf den Global-Memory. Ändern sich die entsprechenden Daten, müssen sie im Texture-Memory neu definiert werden.

Register-Memory

Jeder Core eines Multiprozessors hat einen eigenen Speicherbereich im Register-Memory. Dieser wird für lokale Variablen verwendet, welche innerhalb von Threads definiert werden. Jeder Thread kann exklusiv auf diesen Speicherbereich zugreifen. Die Daten bleiben nur über die Lebenszeit des Threads im Register-Memory und werden wieder freigegeben, sobald der Thread beendet wurde. Die Zugriffe auf Register-Memory erfolgen schneller als auf den Global-Memory.

Local-Memory

Werden in einem Thread Variablen angelegt, die größer sind als der für ihn reservierte Bereich im Register-Memory, werden diese auf den Local-Memory verlagert. Diese Variablen können exklusiv nur von jenem Thread verwendet werden, in welchem sie definiert wurden. Die Zugriffszeit auf den Local-Memory entspricht jener auf den Global-Memory. Nach der Lebensdauer des Threads werden die reservierten Speicherbereiche automatisch wieder freigegeben.

Shared-Memory

Es handelt sich dabei um einen lokalen Speicherbereich auf jedem Multiprozessor. Jeder Multiprozessor hat einen eigenen Shared-Memory. Die Zugriffzeit auf den Shared-Memory ist (abhängig von der jeweiligen Grafikkarte) in etwa um den Faktor 100 schneller als der Zugriff auf den

Global-Memory [51]. Der Shared-Memory wird verwendet, um Daten innerhalb von Threads auszutauschen. Daten können nur zwischen Threads ausgetauscht werden, die auf demselben Core liegen. Der Nachteil des Shared-Memory besteht in seiner geringen Größe. Die maximale Größe des Shared-Memory pro Multiprozessor liegt zwischen 16KB (bis Compute Capability 1.3) und 48KB (ab Compute capability 2.0). [24]

3.4. Work Flow

Parallele Code-Abschnitte, also Kernel-Funktionen, werden auf der GPU ausgeführt. Da die Grafikkarte nicht auf den Hauptspeicher des Computers zugreifen kann, ist es notwendig, alle Daten, die auf der GPU verwendet werden sollen, auf den Global-Memory der GPU zu kopieren. Das Ergebnis muss anschließend wieder zurück auf den Hauptspeicher kopiert werden, wenn es von der CPU weiterverwendet werden soll. Es gibt zwei Möglichkeiten, Parameter an den Kernel zu übergeben [43]:

1. Will man einen Parameter nur zum Zweck des Lesens an den Kernel übergeben (beispielsweise Variablen für Bildbreite und Bildhöhe), kann die Übergabe genauso erfolgen wie bei einer normalen C-Funktion. Die notwendigen Kopiervorgänge zwischen Host und Kernel werden automatisch durchgeführt.

2. Die Übergabe von Zeigern an die GPU, beispielsweise auf ein Array mit Eingabedaten, ist nicht möglich, da der entsprechende Zeiger an eine Stelle im Hauptspeicher zeigt, auf den die GPU keinen Zugriff hat. In diesem Fall ist es notwendig, den benötigten Speicher im Global-Memory der GPU zu reservieren und das gewünschte Array vom Hauptspeicher auf den reservierten Speicher im Global-Memory zu kopieren. Da der Kernel keine Werte an den Host zurückgeben kann, müssen Daten, die danach von der CPU weiterbearbeitet werden sollen, im Anschluss an die Berechnungen im Kernel wieder zurück auf den Hauptspeicher kopiert werden.

Im zweiten Fall sieht der Ablauf im Allgemeinen so aus [43]:

- *Speicher auf dem Global-Memory reservieren:*
  Die Reservierung des Speichers kann mit der CUDA-API-Funktion *cudaMalloc* erfolgen.

- *Daten vom Hauptspeicher auf den Grafikkartenspeicher kopieren:*
  Das Kopieren erfolgt mit der CUDA-API-Funktion *cudaMemcpy*. Mit dem Parameter *cudaMemcpyHostToDevice* gibt man an, dass es sich um einen Kopiervorgang zwischen Hauptspeicher und Grafikkartenspeicher handelt.

- *Kernel ausführen:*
  Der Programmcode wird in mehreren Threads parallel abgearbeitet. Das Ergebnis wird anschließend in einem Array auf dem Global-Memory abgelegt.

- *Ergebnisse zurück auf den Hauptspeicher kopieren:*
  Das Kopieren erfolgt mit der CUDA-API-Funktion *cudaMemcpy*. Da zu diesem Zeitpunkt vom Device auf den Host kopiert wird, lautet der letzte Parameter *cudaMemcpyDeviceToHost.*

- *Reservierte Speicherbereiche freigeben:*
  Der reservierte Speicherbereich im Global-Memory muss anschließend wieder freigegeben werden. Die CUDA-API-Funktion dafür ist *cudaFree*. Da der Global-Memory in seiner Kapazität sehr eingeschränkt ist, ist es besonders wichtig, keine unnötigen Speicherbereiche reserviert zu halten. Speicherbereiche sollten freigegeben werden, sobald sie nicht mehr benötigt werden.

Da die Bandbreite zwischen Hauptspeicher und Global-Memory verhältnismäßig gering ist, sollte versucht werden, die Kopiervorgänge zwischen Host und Kernel möglichst gering zu halten. Es kann daher sinnvoll sein, auch sequentielle Programmteile auf der GPU auszuführen, da der aufwändige Kopiervorgang und das Reservieren und Freigeben von Speicherbereichen mehr Zeit in Anspruch nehmen kann, als durch die schnellere Abarbeitung von seriellen Programmteilen auf der CPU gewonnen wird. [43]

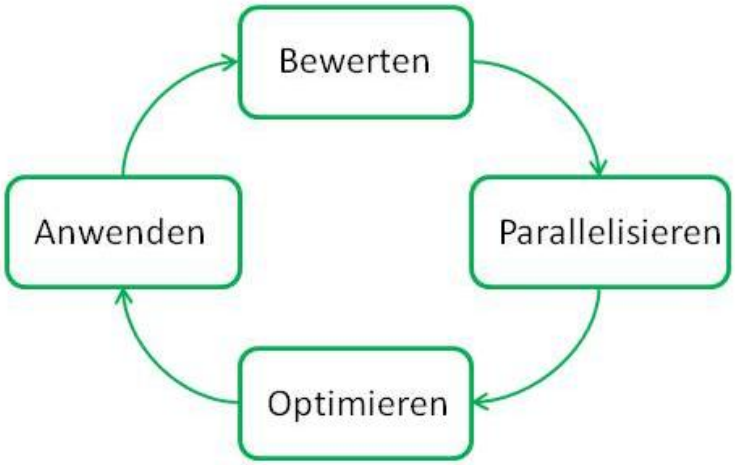


Abbildung 3.9: *APOD-Design-Cycle zur Optimierung von bestehendem Code mittels Parallelisierung. Nach* [42]

3.5. APOD-Design-Zyklus

NVIDIA stellt im *Best Practice Guide* [42] das Design Modell *APOD* vor. APOD steht für *Assess* (Bewerten) *Parallelize* (Parallelisieren) *Optimize* (Optimieren) *Deploy* (Anwenden). Ziel dieses Modells ist es, bestehende Anwendungen und Algorithmen mithilfe von parallelen CUDA-Implementierungen zu beschleunigen. Der Ablauf des APOD-Design-Zyklus wird in Abbildung 3.9 veranschaulicht.

Bewerten

Im ersten Schritt gilt es jene Teile der Anwendung zu finden, welche für einen großen Teil der Laufzeit verantwortlich sind. Hat man diese gefunden, gilt es zu evaluieren, ob sich die Laufzeit dieser Teile durch eine parallele Implementierung verringern lässt. Dazu muss bewertet werden, an welchen Stellen eine parallele Abarbeitung möglich ist. Wenn für Berechnungen Ergebnisse aus vorherigen Schritten benötigt werden, ist eine serielle Abarbeitung notwendig.

Parallelisieren

Hat man die rechenintensiven Teile des Codes identifiziert und bewertet, welche Schritte parallel erfolgen können, kann man damit beginnen, die Bearbeitungsschritte auf die GPU zu verlagern und parallel abzuarbeiten. Abhängig von den Anforderungen des originalen Codes kann das beispielsweise durch die Verwendung von vorhandenen optimierten Bibliotheken, wie *cuBLAS*, *cuFF* oder *Thrust* [52] erfolgen. Sequentielle Implementierungen können auch äquivalent auf die GPU verschoben werden, wo sie parallel auf mehreren Cores anstatt schrittweise in einer Schleife ausgeführt werden. Eine parallele Implementierung kann allerdings auch eine aufwändige Umstrukturierung des originalen Codes erfordern.

Optimieren

Nachdem die Parallelisierung eines Programmteils durchgeführt wurde, kann die Performance durch Optimierung der Prozeduren weiter verbessert werden. Bestimmte Abläufe können beispielsweise weiter beschleunigt werden, indem auf Aspekte der CUDA Architektur Rücksicht genommen wird, wie beispielsweise die spezielle Art der Speicherverwaltung auf der GPU.

Anwenden

Hat man die GPU Beschleunigung von Programmkomponenten fertig gestellt, kann man die Resultate mit jenen des originalen Codes vergleichen. Danach kann der Zyklus wieder von vorne beginnen und der nächste Programmteil behandelt werden.

4. Box-Filter

In diesem Kapitel wird der Box-Filter (Mittelwert-Filter) vorgestellt. Dieser lineare Glättungsfilter stellt die Basis für die Entwicklung weiterer Algorithmen dar. Die in diesem Abschnitt beschriebenen Box-Filter-Implementierungen kommen in Kapitel 6 im Zuge der Implementierung des Guided Filters [23] zur Anwendung. Dieses Kapitel beschreibt die Grundlagen der linearen Filterung im Allgemeinen, und des Box-Filters im Speziellen. Insbesondere werden eine sequentielle CPU-Implementierung und eine parallele GPU-Implementierung des Box-Filters diskutiert. Die grundlegende Funktionsweise der Algorithmen wird anhand der Filterung von Bildern beschrieben. Im Anschluss daran wird jeweils eine Erweiterung zur Filterung von Videos diskutiert, welche im Zuge dieser Diplomarbeit entwickelt wurde. Die Laufzeiten der unterschiedlichen Box-Filter-Implementierungen werden anschließend miteinander verglichen.

4.1. Grundlagen

Ziel des Box-Filters ist es, durch Weichzeichnung (Glättung) Rauschen aus Bildern zu entfernen, ohne die ursprüngliche Bildgeometrie zu verändern [33]. So eine Filterung ist im Allgemeinen dadurch gekennzeichnet, dass das Ergebnis nicht aus einem einzigen Ursprungspixel berechnet wird, sondern aus einer Menge von Pixeln aus dem Originalbild. Die Glättung eines Bildes erfolgt zum Beispiel durch das Ersetzen der Pixel mit dem Durchschnittswert der benachbarten Pixel. Im Falle des Box-Filters wird dabei ein rechteckiges Filterfenster mit dem jeweiligen zu berechnenden Pixel im Mittelpunkt verwendet. Die Größe des Filterfensters definiert, wie viele ursprüngliche Pixel zur Berechnung des neuen Pixelwertes herangezogen werden. Je größer das Filterfenster gewählt wird, desto stärker ist die Weichzeichnung (siehe Abbildung 4.1).

a) b) c)

Abbildung 4.1: *Anwendung des Box-Filters. a) originales Eingabebild. b) Eingabebild gefiltert mit einem 3 x 3 Box-Filter. Das Bildrauschen nimmt durch die Weichzeichnung ab. c) Eingabebild gefiltert mit einem 8 x 8 Box-Filter. Durch das größere Filterfenster wird das Bild stärker geglättet. Es wird unscharf.*

Filter werden im Allgemeinen in lineare und nicht lineare Filter eingeteilt. Der Unterschied besteht in der Verknüpfung der Pixel. Bei linearen Filtern werden die Pixelwerte innerhalb des Filterfensters in linearer Form, beispielsweise durch eine gewichtete Summe, miteinander verknüpft. Das Filterfenster wird durch eine Matrix von Filterkoeffizienten spezifiziert, dem sogenannten *Filterkern H(i,j).* Die Größe des Filterkerns entspricht der Größe des Filterfensters. Seine Werte definieren das Gewicht, mit

dem das entsprechende Pixel multipliziert wird. Das Gewicht definiert, wie stark der Wert des Pixels in die Mittelwertberechnung einfließt. Beim Box-Filter haben alle Pixel innerhalb des Filterfensters dasselbe Gewicht. Im Falle eines 3 x 3 = 9 Filterfensters werden also alle neun Pixel mit demselben Gewicht multipliziert, nämlich $\frac{1}{p \cdot q}$, wobei *p* und *q* die Anzahl der Pixel des Filterkerns *H* darstellen:

$$H(i,j) = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} \tag{4.1}$$

i und *j* sind die Koordinaten der Gewichte innerhalb des Filterkerns. Beim 3 x 3 Box-Filter wird zur Berechnung des Mittelwertes eines Pixels wie folgt vorgegangen:

- Der Filterkern *H(i,j)* (Formel (4.1)) wird über das Eingangsbild *I* gelegt, sodass der Koordinatenursprung auf dem aktuellen Pixel *I(u, v*) liegt.

- Nun werden alle Pixel mit dem entsprechenden Filterkoeffizienten multipliziert (im Falle des 3 x 3 Box-Filters mit $\frac{1}{9}$) und die jeweiligen Ergebnisse aufsummiert.

- Die resultierende Summe wird an die entsprechende Position im Ergebnisbild *Q(u,v)* gespeichert.

Die Filterung wird durch lineare Faltung des Eingabebildes *I* mit dem Filterkern *H* durchgeführt. Die lineare Faltung ist wie folgt definiert [33]:

$$\mathrm{Q}(u,v) = \sum_{i=-\frac{p}{2}}^{\frac{p}{2}} \sum_{j=-\frac{q}{2}}^{\frac{q}{2}} I(u+i, v+j) H(i+\frac{p}{2}, j+\frac{q}{2}) \tag{4.2}$$

Die lineare Faltung (*) kann verkürzt wie folgt geschrieben werden [33]:

$$\mathrm{Q} = I * H \tag{4.3}$$

Hier entspricht *I* dem Eingabebild und *Q* dem gefilterten Ausgabebild. *u* und *v* sind die Koordinaten des zu berechnenden Pixels. *i* und *j* dienen zur Bestimmung der Position der Nachbarpixel. *H* ist der Filterkern mit der Dimension *p* x *q*. Die lineare Filterung hat folgende Eigenschaften [33]:

- *Kommutativität:*
  Das Ergebnis der Faltung von *I* mit *H* führt zum gleichen Ergebnis wie eine Faltung von *H* mit *I* [33]*:*

$$I * H = H * I \tag{4.4}$$

- *Linearität:*
  Wird ein Bild mit einem Skalar α multipliziert, dann multipliziert sich auch das gefilterte Ausgabebild um diesen Faktor [33]:

$$(I \cdot \alpha) * H = I * (\alpha \cdot H) = \alpha \cdot (I * H) \tag{4.5}$$

- *Assoziativität:*
  Die Reihenfolge mehrerer Faltungen kann verändert werden, ohne dass sich das Ergebnis ändert. Das Ergebnis bleibt, unabhängig von der Reihenfolge der Filteroperationen, dasselbe. Es können mehrere Filter beliebig zu neuen Filtern zusammengefasst werden [33]:

$$I * (Hx * Hy) = (I * Hx) * Hy \tag{4.6}$$

4.2. Sequentielle Implementierung

Eine naive Implementierung des Box-Filters für ein zweidimensionales Bild besteht aus der direkten Anwendung einer zweidimensionalen linearen Faltung auf jedes Pixel des Eingabebildes (Formel (4.2)). Der Nachteil dieses naiven Ansatzes liegt in der Anzahl der benötigten Operation pro Bildelement. Im Falle eines Box-Filters mit einem Filterkern der Dimension 3 x 3 werden für jeden Bildpunkt 3 x 3 = 9 Operationen benötigt. Im Falle eines 5 x 5 Filterkerns sind 5 x 5 = 25 Operationen notwendig. Es ist zu erkennen, dass die Anzahl der benötigten Operationen pro Bildpunkt proportional mit der Größe des Filterkerns steigt.

Im Gegensatz zu einer zwei- oder sogar dreidimensionalen Faltung wächst die Anzahl der benötigten Operationen bei einer eindimensionalen Faltung lediglich linear zur Filtergröße. Werden zwei oder mehrere lineare Faltungen hintereinander ausgeführt, wächst der Aufwand weiterhin linear zur Größe des Filterkerns. Die Assoziativität der linearen Faltung macht es möglich, eine zweidimensionale Faltung durch zwei hintereinander ausgeführte eindimensionale Faltungen zu ersetzen. Da die Reihenfolge, in welcher Filteroperationen ausgeführt werden, nicht von Bedeutung ist, können beliebig viele eindimensionale Filter zu einem mehrdimensionalen Filter zusammengefasst werden. Es ist somit möglich, einen zweidimensionalen Filter *H* in zwei eindimensionale Filter *Hx* und *Hy* zu zerlegen [33]:

$$H = Hx * Hy \tag{4.7}$$

Die Möglichkeit, mehrdimensionale Filter in mehrere eindimensionale Filter zerlegen zu können, wird als *Separierbarkeit* [33] bezeichnet. Formel (4.8) gibt ein Beispiel dafür, wie ein 3 x 3 Box-Filter in zwei eindimensionale Filter separiert werden kann.

$$H = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} \qquad Hx = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix} \quad Hy = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} \quad H = Hx * Hy \tag{4.8}$$

Filterung von Bildern

Der separierte Box-Filter benötigt im Falle eines 5 x 5 Filterkerns nur 5 + 5 = 10 Operationen pro Bildelement und erfordert somit erheblich weniger Rechenaufwand als eine zweidimensionale Faltung. Die Filterung eines Bildes mittels Box-Filter erfolgt somit in zwei Schritten [33]:

1. Filterung der Spalten des Bildes.

2. Filterung der Zeilen des gefilterten Ergebnisses aus dem ersten Schritt.

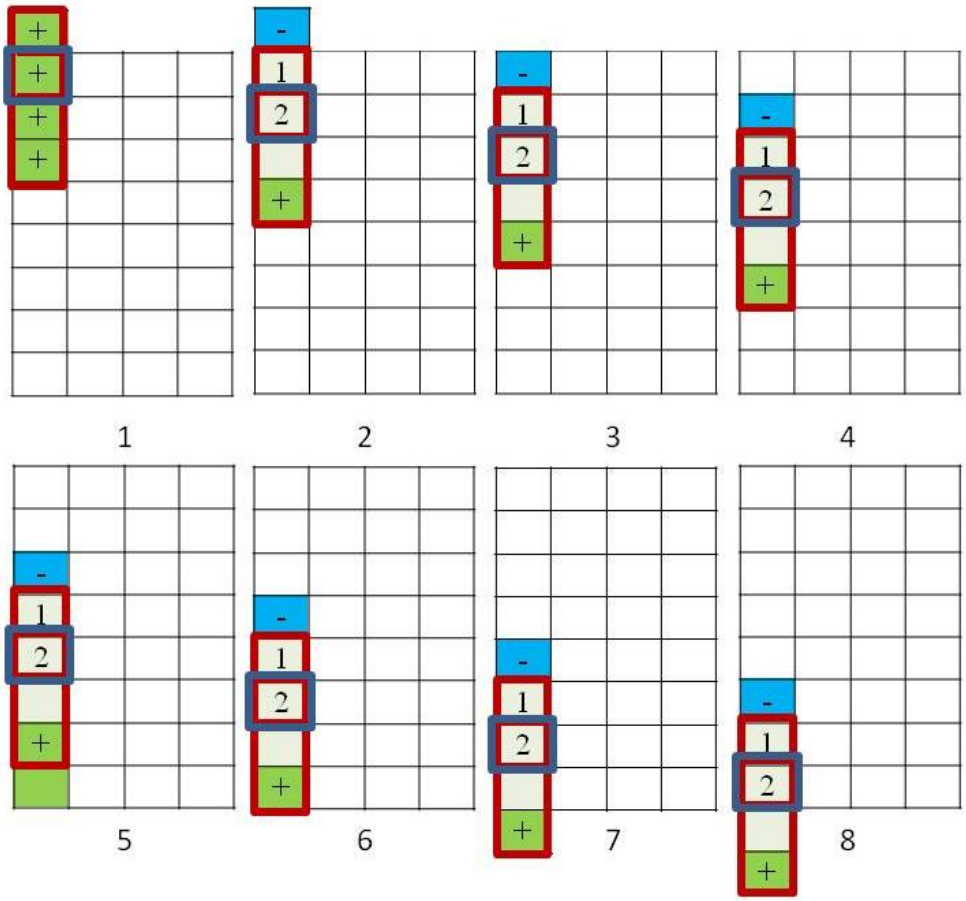


Abbildung 4.2: *Filterung der ersten Spalte eines Bildes. Das Filterfenster (dargestellt durch das rote Fenster) bewegt sich Pixel für Pixel vom oberen zum unteren Bildrand. Der Mittelwert des aktuellen Pixels (2) wird berechnet, indem vom Mittelwert des vorangegangenen Pixels (1) der Wert des aus dem Filterfenster herausgefallenen Pixels (blau, -) subtrahiert und der Wert des neu hinzugekommenen Pixels (grün, +) addiert wird. Es wird in jedem Schritt (mit Ausnahme des ersten Pixels) ein Gewicht subtrahiert und ein Gewicht addiert. Das Resultat wird anschließend durch die Größe des Filterfensters dividiert.*

Die Effizienz des Box-Filters kann weiter gesteigert werden, indem die *Sliding Window*-Technik [53] angewandt wird. Bei der Berechnung des Mittelwertes zweier benachbarter Pixel werden nahezu dieselben Nachbarpixel herangezogen. Es ist somit nicht notwendig, den Mittelwert an jedem Bildpunkt neu zu berechnen. Stattdessen können die bereits berechneten Mittelwerte der Nachbarpixel verwendet werden, um den neuen Mittelwert zu ermitteln. Die Funktionsweise der Sliding Window-

Technik wird in Abbildung 4.2 anhand der Filterung der Spalten eines Bildes mit einem 2 x 2 Filterkern demonstriert. Listing 4.1 zeigt den Programmcode der Spaltenfilterung und Abbildung 4.2 illustriert den Ablauf bei der Filterung der ersten Bildspalten des Eingabebildes.

Eingabewerte: Eingabebild *I*, Ausgabebild *Q*, Bildbreite *w*, Bildhöhe *h*, Filterradius *r*

```
0    void boxfilter_y(float *I, float *Q, int w, int h, int r)
1    {
2     for (int x = 0; x < w; x++) {
3          float sum = I[0];
4
5          for (int y = 0; y < r + 1; y++) {
6                sum += I[y * w + x];
7          }
8
9          Q[x] = sum / (r * 2);
10
11         for(int y = 1; y < h; y++) {
12             if(y - r > 0)
13                sum -= I[(y - r) * w + x];
14             else
15                sum -= I[y * w];
16
17             Q[y * w + x] = sum / (r * 2);
18
19             if(y + r < h)
20                sum += I[(y + r) * w + x];
21             else
22                sum += I[(h - 1) * w + x]
23         }
24    }
25   }
```

Listing 4.1: *Sequentieller Programmcode zum Filtern der Spalten eines Bildes (entspricht der Implementierung in* [27]*).*

Der in Listing 4.1 und Abbildung 4.2 beschriebene Algorithmus filtert das Bild Spalte für Spalte. Die Größe des Filterfensters ist in diesem Beispiel mit vier Pixeln festgelegt. Der Algorithmus verläuft von links nach rechts, bis die rechte Bildkante erreicht wurde (Listing 4.1, Zeile 2). In jedem Schleifendurchlauf wird eine Spalte gefiltert. Das Filterfenster bewegt sich Pixel für Pixel von oben nach unten, bis der Mittelwert des untersten Pixels berechnet wurde. Zur Filterung einer Spalte wird wie folgt vorgegangen:

1. Zuerst wird der Mittelwert für das erste Pixel (am oberen Bildrand) berechnet (siehe Abbildung 4.2, Schritt 1; Listing 4.1, Zeile 5 - 7). Das Filterfenster wird auf das erste Pixel zentriert. Anschließend wird jeder Wert, der sich innerhalb des Filterfensters befindet, schrittweise addiert. Der Mittelwert ergibt sich nach Division mit der Fenstergröße (Listing 4.1, Zeile 9). Dieser wird an der entsprechenden Position im Ausgabebild *Q* gespeichert. Das Filterfenster ragt über den oberen Bildrand hinaus. Da außerhalb der Bildränder keine Bildwerte zur Berechnung des Mittelwertes vorhanden sind, werden die Bildränder getrennt behandelt. Es gibt verschiedene Strategien zur Randbehandlung. Drei Beispiele dafür sind [33]:

- Bildpunkten außerhalb des Bildbereiches wird ein konstanter Wert zugewiesen, beispielsweise grau oder schwarz (Abbildung 4.3a)).

- Randpixel werden außerhalb des Bildes fortgesetzt. Anstatt einen konstanten Wert anzunehmen, werden zur Berechnung die Intensitätswerte der Randpixels verwendet (Abbildung 4.3b)).

- Pixel außerhalb des Bildbereiches wiederholen sich zyklisch von der gegenüberliegenden Seite des Bildes (Abbildung 4.3c))

Der erste Ansatz führt zu einer Verfälschung der Ergebnisse an den Bildrändern, da es zwischen den Randpixeln und den Pixeln außerhalb des Bildbereiches zu einer sprunghaften Änderung der Intensitätswerte kommt. In dieser Diplomarbeit wurde der zweite Ansatz zur Randbehandlung gewählt. In diesem werden sprunghafte Intensitätsänderungen vermieden. In der Sliding Window-Technik wird diese Strategie zur Randbehandlung umgesetzt, indem der erste Bildpunkt ein weiteres Mal addiert wird (Listing 4.1, Zeile 3).

2. Zur Berechnung des aktuellen Pixels wird der Mittelwert des vorangegangenen Pixels verwendet (siehe Abbildung 4.2, Schritt 2 - 6). Von diesem wird der Wert jenes Pixels subtrahiert, das aus dem Filterfenster herausfällt (Abbildung 4.2, blau) und der Wert jenes Pixels addiert, das hinzukommt (Abbildung 4.2, grün; Listing 4.1, Zeile 11 - 23). Den Mittelwert erhält man anschließend, indem man das Resultat durch die Größe des Filterfensters dividiert. Für den Fall, dass das Filterfenster über den oberen Bildrand hinaus ragt, wird der Wert des Randpixels verwendet (Listing 4.1, Zeile 14 - 15). Dieser Schritt stellt den Hauptteil der Filterung der Spalte dar.

3. Der untere Bildrand wird, ebenso wie der obere Bildrand, getrennt behandelt. Ragt das Filterfenster über den Bildrand hinaus, wird der Wert des Randpixels verwendet (Listing 4.1, Zeile 21 – 22; Abbildung 4.2, Schritt 7 - 8).

a) b) c)

Abbildung 4.3: *Randbehandlung beim Box-Filter. a) Für die Bildpunkte außerhalb des Bildbereiches wird ein konstanter Wert angenommen (beispielsweise grau). b) Bildpunkte außerhalb des Bildbereiches erhalten den Wert des entsprechenden Randpixels. c) Bildpunkte außerhalb des Bildes werden zyklisch von der gegenüberliegenden Seite des Bildes fortgesetzt.* [33]

Im Anschluss an die Filterung der Spalten werden die Zeilen des Bildes auf die gleiche Art gefiltert. Jede Bildzeile wird dabei getrennt behandelt. Der Algorithmus wird solange durchgeführt, bis alle

Zeilen des Bildes gefiltert wurden. Durch eine solche Implementierung wird immer ein Gewicht subtrahiert und eines addiert, unabhängig von der Größe des Filterkerns. Die Laufzeit des Filters ist somit konstant bezüglich der Größe des Filterkerns. Die ersten Pixelwerte am oberen Bildrand werden getrennt ermittelt (siehe Abbildung 4.2, Schritt 1). [26]

Eingabewerte: Eingabebild *I*, Ausgabebild *Q*, Bildbreite *w*, Bildhöhe *h*, Filterradius *r*

Device Funktion

```
1   __device__ void
2   d_boxfilter_float_yb(float *I, float *Q, int w, int h, int r)
3   {
4    float sum = I[0];
5
6    for (int y = 0; y < r + 1; y++) {
7         sum += I[y * w + x];
8    }
9
10   Q[x] = sum / (r * 2);
11
12   for(int y = 1; y < h; y++) {
13       if(y - r > 0)
14          sum -= I[(y - r) * w + x];
15       else
16          sum -= I[y*w];
17
18       Q[y * w + x] = sum / (r * 2);
19
20       if(y + r < h)
21          sum += I[(y + r) * w + x];
22       else
23          sum += I[(h - 1) * w + x]
24   }
25  }
```

Kernel

```
26  __global__ void
27  d_boxfilter_y_global(float *I, float *Q, int w, int h, int r)
28  {
29   unsigned int x = __umul24(blockIdx.x, blockDim.x) + threadIdx.x;
30
31   if(x > w)
32      return;
33
34   d_boxfilter_y(&I[x], &Q[x], w, h, r);
35  }
```

Kernel-Aufruf

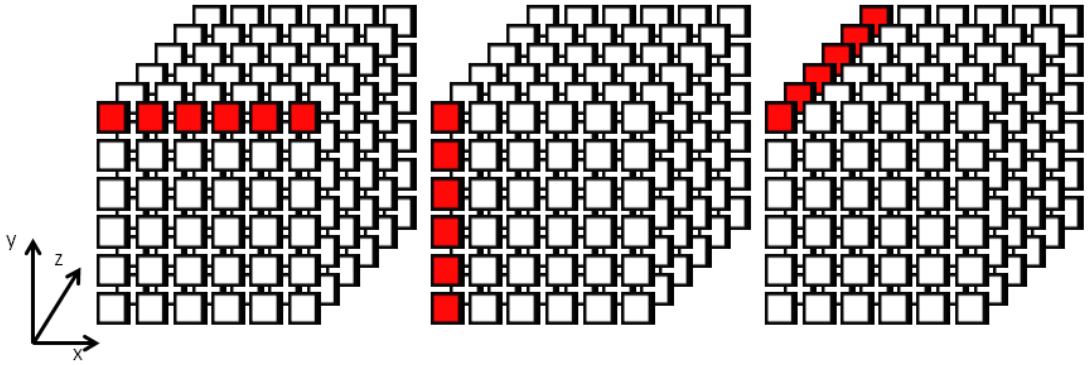
```
36  d_boxfilter_y_global<<< w / 128, 128>>>(I, Q, w, h, r);
```

Listing 4.2: *CUDA-Kernel der Spaltenfiltung des Box-Filters.* [27]

Filterung von Videos

Soll statt eines Bildes ein Video gefiltert werden, ist zusätzlich eine Filterung über mehrere Frames hinweg notwendig. Das Video kann dabei als dreidimensionale Datenstruktur betrachtet werden (siehe Abbildung 4.4). Die Filterung der Spalten und Zeilen der einzelnen Frames kann analog zur Filterung

in Listing 4.1 durchgeführt werden. Statt eines Bildes werden mehrere Bilder nacheinander gefiltert. Zusätzlich werden die Bildpunkte in zeitlicher Dimension gefiltert, um die temporäre Kohärenz zwischen den Frames zu gewährleisten. Die zeitliche Filterung kann analog zur Zeilen- und Spaltenfilterung durchgeführt werden. Der Programmcode aus Listing 4.1 bezieht sich auf die Filterung eines Bildkanals. Im Falle eines RGB-Bildes wird der Box-Filter auf jeden Farbkanal getrennt angewandt.



a) b) c)

Abbildung 4.4: *Ein Video kann als dreidimensionale Datenstruktur betrachtet werden. a) Filterung der Zeilen. b) Filterung der Spalten. c) zeitliche Filterung über mehrere Frames.*

4.3. Parallele Implementierung

In diesem Kapitel wird eine parallele GPU-Implementierung des in Abschnitt 4.2 vorgestellten Box-Filters für Bilder und Videos beschrieben. Durch die Parallelisierung von Teilen des Box-Filters wird die Effizienz im Vergleich zur sequentiellen CPU-Implementierung gesteigert. Die Implementierung des Box-Filters, welche im Zuge dieser Diplomarbeit erstellt wurde, baut auf einer CUDA-Implementierung von NVIDIA auf, welche im NVIDIA GPU Computing SDK (Version 4.1) [27] enthalten ist. In dieser Diplomarbeit wurde der vorhandene Algorithmus zur Verwendung für Videos adaptiert.

Filterung von Bildern

Aufgrund der Separierbarkeit des Box-Fiters können die Spalten und Zeilen eines Bildes, unabhängig voneinander, nacheinander gefiltert werden. Wie bei der vorgestellten CPU-Implementierung (Listing 4.1), ist die Reihenfolge der Filterungen nicht von Bedeutung. Für die parallele CUDA-Implementierung ist es vorteilhaft, zuerst die Spalten zu filtern. Dies hängt mit der Organisation der Speicherzugriffe auf den Global-Memory der Grafikkarte zusammen (siehe Kapitel 3). Um die Filterung effizient durchführen zu können, muss eine möglichst hohe Parallelität des Algorithmus erreicht werden. Im Idealfall würde das für die GPU-Implementierung des Box-Filters bedeuten, dass jedes Pixel in einem eigenen Thread bearbeitet wird. Unter Verwendung der Sliding Window-Technik ist das allerdings nicht möglich, da für die Berechnung des Mittelwertes eines Pixels der Ausgabewert des unmittelbaren Vorgängers bereits vorhanden sein muss. Wie in Abbildung 4.2 zu sehen ist, fließt bei der Berechnung des Mittelwertes an einer konkreten Position nur der darüberliegende Wert der aktuellen Spalte ein. Das Ergebnis ist somit von anderen Spalten unabhängig und die Spalten können von der GPU somit parallel bearbeitet werden. Dazu wird für jede Spalte ein eigener Thread erstellt, in

welchem sie sequentiell gefiltert wird. Der Ablauf zur Filterung einer Spalte entspricht dem in Abbildung 4.2 dargestellten Vorgehen. [26]

Listing 4.2 zeigt den Programmcode des CUDA-Kernels zur Spaltenfilterung des Box-Filters [27]. Die parallele GPU-Implementierung entspricht im Wesentlichen der sequentiellen CPU-Implementierung in Listing 4.1. Der Unterschied liegt in der parallelen Filterung der einzelnen Bildspalten. Im Gegensatz zur sequentiellen CPU-Implementierung werden die Spalten nicht nacheinander in einer Schleife abgearbeitet. Stattdessen wird jede Spalte in einem eigenen Thread auf der GPU bearbeitet. Die Bearbeitung der Spalten eines Bildes durch die einzelnen Threads ist in Abbildung 4.5 illustriert.

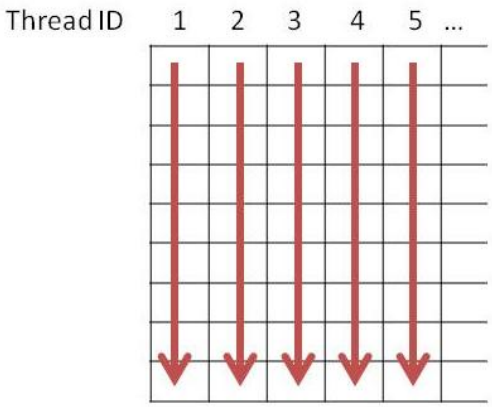


Abbildung 4.5*: In jedem Thread wird eine Bildspalte gefiltert. Die einzelnen Threads werden parallel ausgeführt.*

Der Aufruf des Kernels erfolgt mit den Parametern *<<ceil(w/128), 128>>* (Listing 4.2, Zeile 36). Es werden 128 Threads pro Block erzeugt. Die Anzahl der Threads pro Block wurde im Zuge von empirischen Test ermittelt. Unter Verwendung einer NVIDIA GeForce GTX 660ti wurde mit 128 Threads pro Block im Vergleich zu einer anderen Thread-Anzahl (beispielsweise 64 oder 256) eine geringere Laufzeit erreicht. Wie bereits in Kapitel 3 erwähnt, werden jeweils 32 Threads zu einem Warp zusammengeschlossen. Die Anzahl der Threads pro Block wird demzufolge als ein Vielfaches von 32 gewählt, damit für den Fall, dass ein Warp stillsteht, inzwischen ein anderer Warp ausgeführt werden kann und Leerläufe vermieden werden. Da pro Block 128 Threads instanziert werden, wird die Anzahl der Blöcke um die Anzahl der Threads pro Block reduziert, um nicht zuviele Threads zu instanzieren. Bei einer Bildbreite von 512 Pixeln werden mit diesen Parametern vier Blöcke mit jeweils 128 Threads generiert. Da nicht vorausgesetzt werden kann, dass die Anzahl der Spalten genau ein Vielfaches von 128 ist, wird das Ergebnis aufgerundet, da sonst die Möglichkeit besteht, dass Spalten am rechten Bildrand nicht mehr bearbeitet werden. Im Falle einer Bildbreite, die kein Vielfaches von 128 ist, werden mehr Threads generiert als notwendig. Im Kernel wird daher eine Abfrage eingebaut werden, die verhindert, dass der Filter über die Bildbreite hinaus läuft (Listing 4.2, Zeile 31 - 32) und sichergestellt, dass kein Zugriff auf Spalten stattfindet, die nicht vorhanden sind. In der jeweilige Instanz des Kernels wird der Beginn der aktuellen Bildspalte, welche gefiltert werden soll, ermittelt (Listing 4.2, Zeile 29). Im Kernel wird eine Device-Funktion aufgerufen, welche das Filtern einer Bildspalte implementiert. Dieser wird jeweils das erste Pixel der entsprechenden Spalte übergeben. In jeder Instanz des Kernels erfolgt die Bearbeitung einer Spalte seriell. Damit Speicherzugriffe parallel erfolgen können, müssen alle Threads eines Warps auf dasselbe Segment im Global-Memory zugreifen. Speicherzugriffe auf unterschiedliche Segmente des Global-Memory

werden seriell durchgeführt (siehe Abbildung 4.6). Die Threads 0 – 31 des ersten Warps greifen im ersten Schritt auf die Bildelemente 0 bis 31 zu, die Threads 32 – 63 des zweiten Warps auf die Bildelemente 32 – 63, und so weiter. In der ersten Bildzeile ist somit sichergestellt, dass die Threads eines Warps jeweils auf benachbarte Speicherelemente des Global-Memory zugreifen. Ab der Bearbeitung der nächsten Bildzeile kann es vorkommen, dass Threads eines Warps auf zwei unterschiedliche Segmente zugreifen. Das ist dann der Fall, wenn die Bildbreite kein Vielfaches der Threads ist, die in einem Warp laufen, also 32. Die Threads von *Warp 5*, dem ersten Warp des zweiten Blocks (siehe Abbildung 4.6), greifen auf Speicherstellen in unterschiedlichen Segmenten zu. Die Zugriffe auf *Segment 1* und *Segment 2* können daher nur seriell erfolgen. Dieses Problem lässt sich lösen, indem vorausgesetzt wird, dass die Bildbreite aller zu filternden Bilder ein Vielfaches von 32 beträgt. Somit ist gewährleistet, dass alle Threads eines Warps auf dasselbe Segment im Global-Memory zugreifen. Bilder mit einer Bildbreite, die kein Vielfaches von 32 ist, werden vorher entsprechend vergrößert, indem die fehlenden Bildbereiche mit einem konstanten Wert aufgefüllt werden (*Padding*). Die hinzukommenden Bildspalten dienen lediglich der Optimierung der Speicherzugriffe und werden nicht gefiltert. [26]

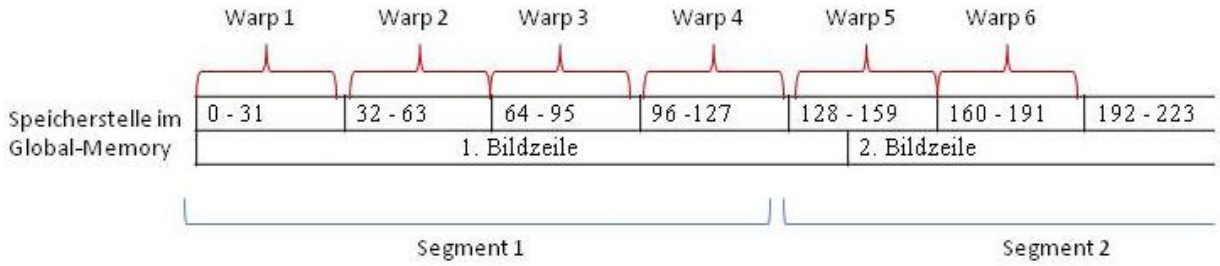


Abbildung 4.6: *Die Threads von Warp 1 bis Warp 4 greifen auf dasselbe Segment im Global-Memory zu (Segment 1). Warp 5 (der erste Warp des zweiten Blockes) greift auf Speicherstellen innerhalb von zwei Segmenten des Global-Memory zu (Segment 1 und Segment 2). Die Speicherzugriffe innerhalb eines Segments können parallel erfolgen. Zugriffe auf zwei unterschiedliche Segmente erfolgen nacheinander.*

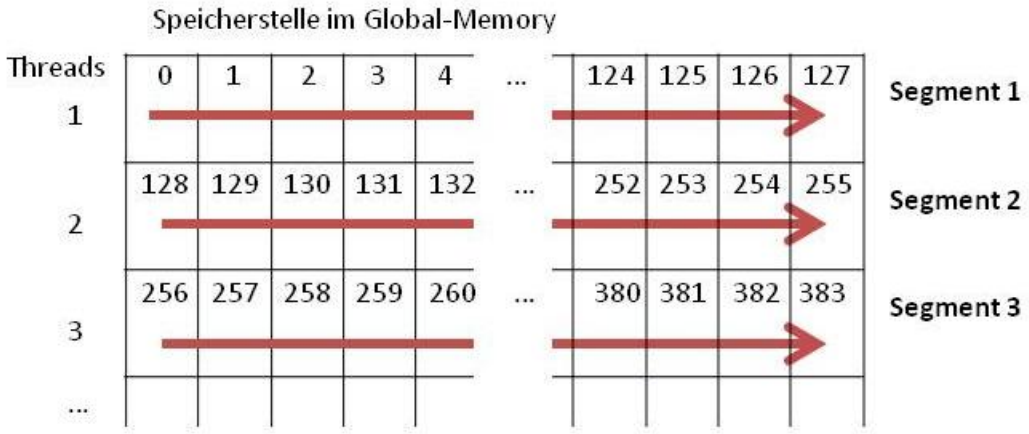


Abbildung 4.7: *Wird der Zeilenfilter analog zum Spaltenfilter umgesetzt, greift zwangsweise jeder Thread auf unterschiedliche Segmente im Global-Memory zu. In diesem Fall erfolgen alle Speicherzugriffe seriell.*

Nachdem die Spalten des Eingabebildes gefiltert wurden, werden auch die Zeilen gefiltert. Bei der sequentiellen Implementierung wurde dies analog zur Spaltenfilterung durchgeführt. Für die GPU-Implementierung würde das bedeuten, dass jede Bildzeile in einem eigenen Thread gefiltert wird. Speicherzugriffe auf den Global-Memory können nur parallel ausgeführt werden, wenn alle Threads eines Warps auf dasselbe Segment zugreifen. Bei einer 1:1 Umsetzung (*GPU* nach Abbildung 4.11) der sequentiellen Implementierung der Zeilenfilterung wäre das nicht der Fall. Bei einer Bildbreite von 128 Pixel würde zwangsweise jeder Thread eines Warps auf ein anderes Segment im Global-Memory zugreifen. Daraus folgt, dass sämtliche Speicherzugriffe eines Warps serialisiert werden müssen. Abbildung 4.7 veranschaulicht dieses Problem: In der ersten Spalte greift *Thread 1* auf *Speicherstelle 0* zu, *Thread 2* auf *Speicherstelle 128*, und so weiter. Alle Zugriffe erfolgen somit in ein anderes Segment des Global-Memory.

Optimierung der Zeilenfilterung

Zur Lösung des Problems der Optimierung der Zeilenfilterung des Box-Filters wurden verschiedene Ansätze vorgeschlagen [26, 27]. Die Implementierung des Box-Filters von NVIDIA [27] (*GPU Text* nach Abbildung 4.11) optimiert die Speicherzugriffe auf den Global-Memory bei Filterung der Bildzeilen durch die Verwendung von Texturen. Bilddaten werden als zweidimensionale Textur im Texture-Memory abgelegt. Zugriffe auf den Texture-Memory können schneller durchgeführt werden als Zugriffe auf den Global-Memory, wenn die Bildpunkte, räumlich gesehen, nahe beieinander liegen. Das Problem der seriellen Speicherzugriffe auf den Global-Memory kann auf diese Weise umgangen werden. Der Nachteil dieses Vorgehens liegt darin, dass Kernel-Funktionen nur lesend auf den Texture-Memory zugreifen können. Die gebildeten Mittelwerte können nicht direkt vom Kernel auf diesen Speicherbereich geschrieben werden. Stattdessen werden die Daten vom Host in den Texture-Memory kopiert. Ändern sich die Bilddaten, müssen die Texturen neu definiert werden. Dieser Ansatz ist durch die Anzahl der hintereinander durchgeführten Box-Filter limitiert. Bei mehr als einer Iteration werden die Ergebnisse nach jedem Filtervorgang auf den Host zurück kopiert. Für den nächsten Filtervorgang werden die gefilterten Daten erneut vom Host an den Texture-Memory gebunden.

In [26] wird ein anderer Ansatz verfolgt, um beim Filtern der Bildzeilen die Serialisierung der Speicherzugriffe auf den Global-Memory zu verhindern (*GPU Trans* nach Abbildung 4.11). Dieser Ansatz betrachtet das Bild als zweidimensionale Matrix. Diese zweidimensionale Matrix kann transponiert werden (das bedeutet, die Zeilen werden zu Spalten und die Spalten zu Zeilen). Nach der Transposition ist es möglich, anstatt der Zeilen noch einmal die Spalten zu filtern. Im Anschluss daran kann die ursprüngliche Orientierung des Bildes durch eine erneute Transposition wieder hergestellt werden. Um eine Matrix zu transponieren, werden die x und y Koordinaten jedes Elementes vertauscht. Eine effektive Matrixtransposition auf der GPU ist nicht trivial, da die Reihenfolge der Speicherzugriffe auf den Global-Memory bei parallelen CUDA-Implementierungen zu berücksichtigen ist (siehe Abbildung 4.7). Im Zuge der Transposition einer zweidimensionalen Matrix wird der Bildpunkt an der Position (1,2) an die Position (2,1) verschoben. Aus der Sicht der Speicherorganisation wird der Bildpunkt an der *Speicherstelle 1* mit jenem Bildpunkt an der *Speicherstelle 128* vertauscht. Die beiden Bildpunkte liegen jeweils in einem anderen Segment im Global-Memory, einer in *Segment 1*, der andere in *Segment 2*. Der Bildpunkt an der Position (1,3) wird entsprechend von *Speicherstelle 2* (*Segment 1*) an die *Speicherstelle 256* (*Segment 3*) kopiert. Wenn jeder Tauschvorgang in einem eigenen Thread durchgeführt wird, lesen alle Threads eines Warps Pixel an benachbarten Speicherstellen (*Thread 1* an *Speicherstelle 0*, *Thread 2* an *Speicherstelle 1*, und so weiter) und greifen nur auf Elemente im selben Segment zu. Anschließend

schreiben alle Threads eines Warps in unterschiedliche Segmente (*Thread 1* in *Segment 2*, *Thread 2* in *Segment 3*, und so weiter). Die Schreibzugriffe werden demzufolge alle serialisiert.

Der NVIDIA Computing SDK enthält eine optimierte CPU-Implementierung zur Transposition von zweidimensionalen Matrizen [54]. Die Daten werden dabei blockweise in den Shared-Memory kopiert und dort schrittweise transponiert. Diese Funktion zur Matrizentransposition setzt voraus, dass die Dimension der zu transponierenden Matrix ein Vielfaches von 32 ist, damit die Zugriffe auf den Shared-Memory optimal erfolgen können. Darum werden, wenn nötig, sowohl die Bildbreite als auch die Bildhöhe durch Padding auf ein Vielfaches von 32 erweitert. Beim Filtern der Spalten ist zu berücksichtigen, dass nur bis zum unteren Bildrand des originalen Bildes gefiltert wird, und nicht bis in den Bildbereich, der durch das Padding hinzugekommen ist. Die Bildpunkte an den Rändern des originalen Bildes würden ansonsten verfälscht werden, da die zusätzlichen Intensitätswerte in den Mittelwert der Randpixel mit einbezogen werden würden. Die in Abschnitt 4.2 erwähnte Randbehandlung würde erst im dazugekommen Bildbereich zum Tragen kommen. Dasselbe gilt bei der anschließenden Filterung der Zeilen. Die parallele GPU-Implementierung des Box-Filters, welche im Zuge dieser Diplomarbeit erstellt wurde (*GPU Trans*), basiert auf der Optimierung der Zeilenfilterung aus [26].

Filterung von Videos

In dieser Diplomarbeit wurde die CUDA-Implementierung des zweidimensionalen Box-Filters (*GPU Trans*) zur Filterung von Videos erweitert. Beim Filtern von Videos wird analog zur Filterung von Bildern vorgegangen. Anstatt eines einzigen Bildes werden mehrere Frames eines Videos gefiltert. Die Frames eines Videos liegen im zweidimensionalen Raum untereinander und können wie ein einziges großes Bild behandelt werden. Bei der Spaltenfilterung ist darauf zu achten, dass jeder Thread genau eine Spalte eines Bildes bearbeitet. Beim Wechsel zwischen zwei Frames werden die Indizes entsprechend angepasst. Zusätzlich zum Filtern der Spalten und der Zeilen wird nun auch in zeitlicher Dimension gefiltert. Das kann analog zur Filterung der Spalten durchgeführt werden (siehe Abbildung 4.8).

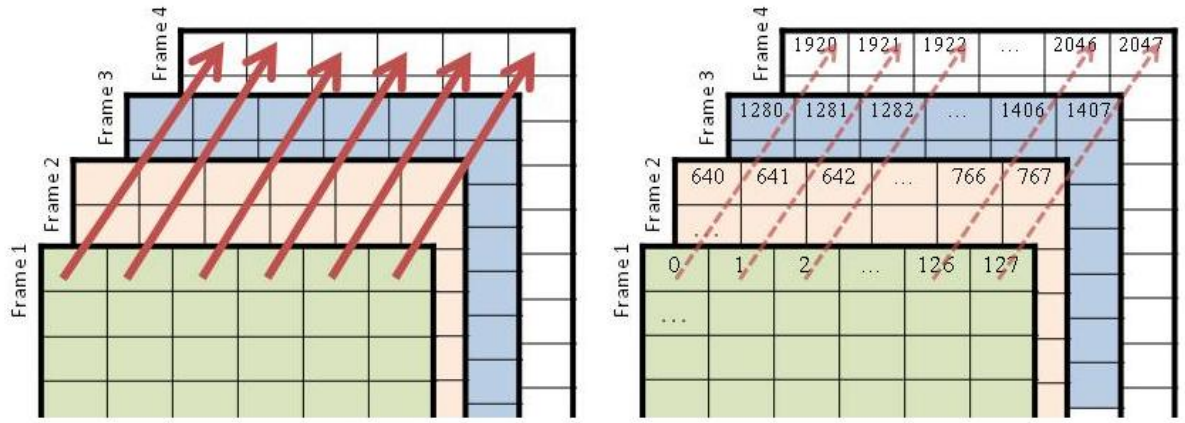


Abbildung 4.8: *Die Filterung in zeitlicher Dimension kann analog zur Filterung der Bildspaltung durchgeführt werden. Dabei kommt erneut die Sliding Window-Technik zum Einsatz* [53]. *Da sowohl die Bildbreite als auch die Bildhöhe auf ein Vielfaches von 32 erweitert wurden, ist die Bedingung erfüllt, dass jeweils alle 32 Threads eines Warps auf Speicherstellen in denselben Segmenten im Global-Memory zugreifen.*

Im Hauptspeicher werden die Bilder in einem zweidimensionalen Array untereinander abgelegt. Zur Bearbeitung auf der Grafikkarte werden die Bilddaten in einem einzigen eindimensionalen Array in den Global-Memory kopiert. Abbildung 4.9 visualisiert die Speicherreihenfolge der Bilddaten.

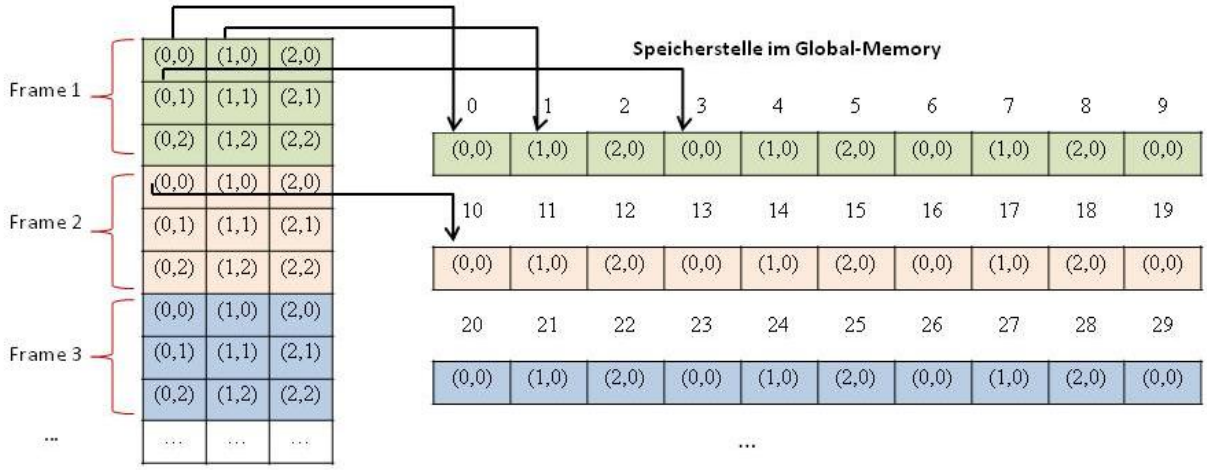
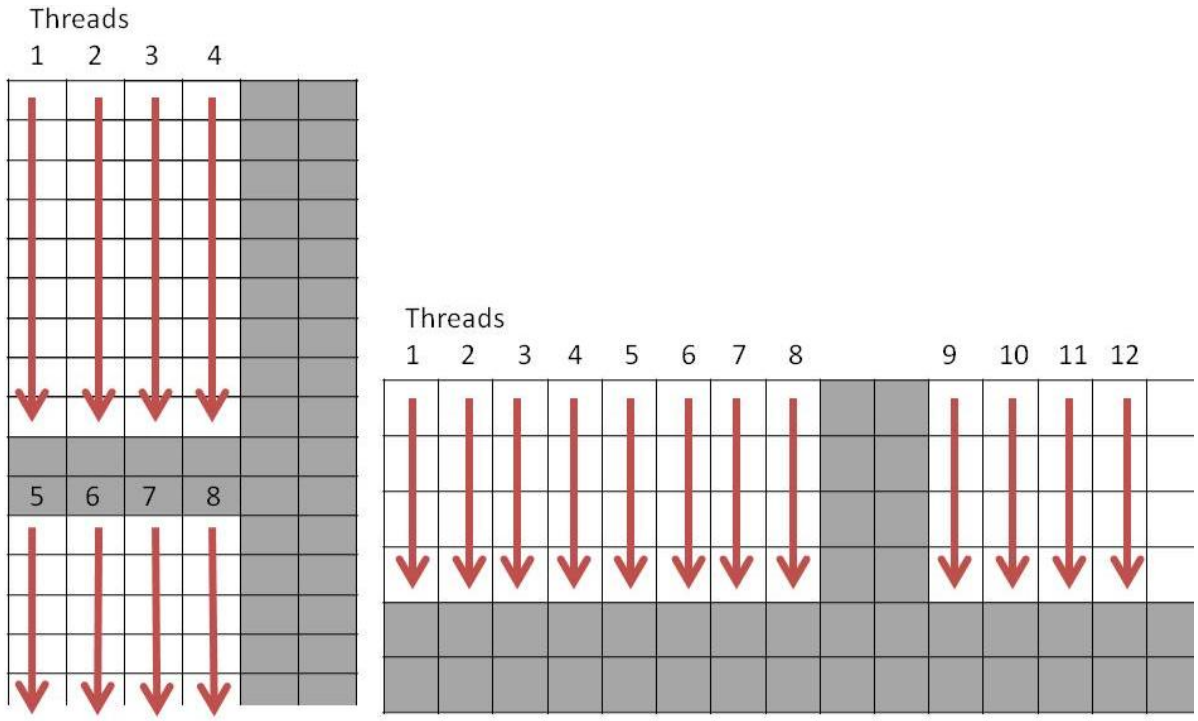


Abbildung 4.9: *Speicheranordnung mehrerer Frames im Global-Memory der Grafikkarte.*



a) b)

Abbildung 4.10: *a) Spaltenfilter für mehrere Frames. Jene Bildbereiche, welche durch Padding hinzugekommen sind, werden bei der Filterung übersprungen. b) Spaltenfilterung der einzelnen Frames nach der Transposition.*

Der Box-Filter wird in drei Schritten durchgeführt: Im ersten Schritt werden die Spalten der einzelnen Frames gefiltert (siehe Abbildung 4.10a)). Die Spaltenfilterung kann für alle Frames parallel

ausgeführt werden, wobei jede Spalte in einem eigenen Thread bearbeitet wird. Da die Breite und Höhe der Bilder auf ein Vielfaches von 32 erweitert wurden (siehe Abschnitt 4.3), sind die dadurch entstandenen Bildbereiche zu berücksichtigen, damit an der Grenze zwischen zwei Frames die richtigen Indizes gewählt werden. Jene Threads, welche *Frame 2* bearbeiten, erhalten als Input die oberste Bildzeile von *Frame 2*. Die zusätzlichen Bildbereiche werden übersprungen. Im zweiten Schritt werden die Zeilen der Frames gefiltert (siehe Abbildung 4.10b)). Um die Optimierung aus [26] verwenden zu können, werden alle Frames zuerst transponiert. Die Bilddaten können als eine einzige zweidimensionale Matrix betrachtet werden, in welcher die einzelnen Frames untereinander angeordnet sind. Diese Bildmatrix kann analog zu einem zweidimensionalen Bild transponiert werden. Anschließend wird der Spaltenfilter erneut auf die einzelnen Frames angewandt. Da hier anstatt eines einzigen Bildes mehrere Bilder gleichzeitig bearbeitet werden, wird die Indizierung der einzelnen Threads angepasst. Im Gegensatz zur ersten Spaltenfilterung, befinden sich innerhalb der Spalten der einzelnen Frames keine durch Padding hinzugekommenen Bildpunkte. Es sind somit keine Index-Sprünge in vertikaler Richtung nötig. Der zusätzliche dritte Schritt filtert in zeitlicher Dimension (siehe Abbildung 4.8). Das erfolgt analog zur Filterung der Spalten, mit dem Unterschied, dass sich das Filterfenster über alle Frames hinweg bewegt, beginnend bei den Pixeln des ersten Frames. Jeder Durchlauf erfolgt in einem eigenen Thread. Da sowohl Breite als auch Höhe der einzelnen Bilder auf ein Vielfaches von 32 angepasst wurden, sind die Speicherzugriffe auf den Global-Memory beim zeitlichen Filtern bereits optimiert. Wie in Abbildung 4.8 zu sehen ist, liegen die Speicherbereiche, die von den einzelnen Threads bearbeitet werden, direkt nebeneinander. Da die Bildbreite ein Vielfaches von 32 ist, erfolgt auch beim Wechsel zwischen zwei Bildzeilen kein Zugriff auf mehr als ein Segment. Dasselbe gilt für den Wechsel zwischen zwei Frames, da auch die Bildhöhe entsprechend angepasst wurde. Es ist somit sichergestellt, dass alle Threads eines Warps auf dasselbe Segment im Global-Memory zugreifen.

4.4. Laufzeitvergleich der Implementierungen

Die Grafik in Abbildung 4.11 zeigt einen Vergleich der Performance von unterschiedlichen Implementierungen des Box-Filters. Ausgeführt wurden diese auf einer NVIDIA GeForce GTX 660Ti. Gefiltert wurde ein 8-Bit Bild mit der Auflösung 1024x1024. Es werden folgende Versionen des Box-Filters verglichen:

1. *CPU*: Eine sequentielle CPU-Implementierung, wie sie in Abschnitt 4.2 beschrieben wurde.
   Laufzeit: 15 Millisekunden

2. *GPU*: Eine parallele GPU-Implementierung ohne Optimierung der Speicherzugriffe beim Filtern der Bildzeilen. Dabei handelt es sich um eine 1:1 Umsetzung der sequentiellen CPU-Version auf der GPU.
   Laufzeit: 3.22 Millisekunden

3. *GPU Text*: Eine parallele GPU-Implementierung, bei der die Speicherzugriffe beim Zeilenfiltern unter Verwendung des Texture-Memory der Grafikkarte optimiert werden. Bei dieser Version handelt es sich um die Box-Filter-Implementierung von NVIDIA, welche im NVIDIA Computing SDK 4.1 enthalten ist [27].
   Laufzeit: 1.86 Millisekunden

4. *GPU Trans*: Eine parallele GPU-Implementierung mit Optimierung der Speicherzugriffe mittels Transponieren der Bildmatrix und erneutem Spaltenfiltern. Dabei handelt es sich um jene Version des Box-Filters, welche in [26] beschrieben wird.
   Laufzeit: 1.29 Millisekunden

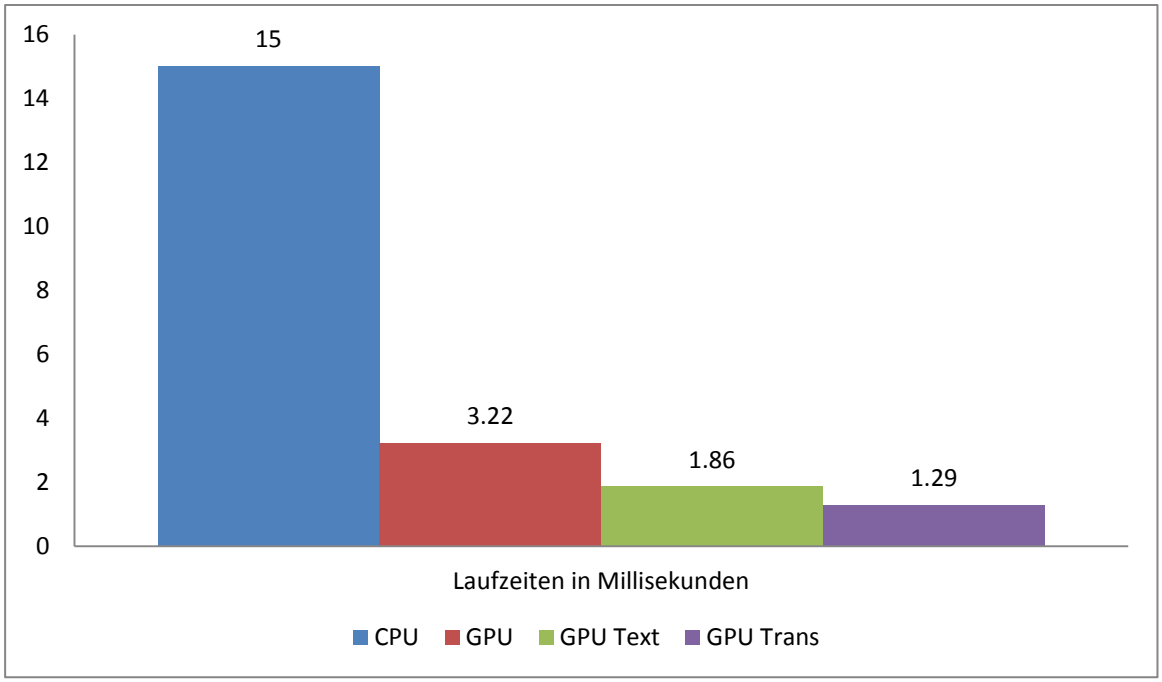


Abbildung 4.11: *Vergleich der Laufzeiten des Box-Filters bei Filterung eines 8-Bit Bildes mit der Auflösung 1024x124 auf einer NVIDIA GeForce GTX 660ti.*

Die Laufzeitvergleiche zeigen, dass die parallelen GPU-Implementierungen signifikant weniger Zeit benötigen als die serielle CPU-Implementierung. Die Laufzeiten von *GPU Text* und *GPU Trans* liegen deutlich unter jener von *GPU*, was auf die Optimierung der Zeilenfilterung zurückzuführen ist. Der Unterschied zwischen den Laufzeiten von *GPU Text* und *GPU Trans* fällt hingegen gering aus. Der Vorteil von *GPU Trans* liegt in der geringeren Anzahl an Kopiervorgängen zwischen Kernel und Host, was vor allem bei mehreren Iterationen des Box-Filters von Vorteil ist.

5. Graphen-basierte Segmentierung und Propagierung

Der Algorithmus aus [22] zur semi-automatischen Propagierung von Disparitäten in Videos basiert auf dem Algorithmus zur Segmentierung von Bildern aus [28]. Dieser wird in [25] zur Segmentierung von Videos in zeitliche und räumliche Regionen erweitert. Ziel der Segmentierung von Bildern und Videos ist es, diese in Regionen zu partitionieren, die in bestimmten Kriterien homogen sind (beispielsweise im Farbwert der enthaltenen Pixel). In diesem Kapitel werden die Grundlagen des Segmentierungs-Algorithmus, auf welchem der Algorithmus aus [22] basiert, behandelt. In Abschnitt 5.1 wird die grundlegende Funktionsweise des Segmentierungs-Algorithmus aus [28] beschrieben. Abschnitt 5.2 diskutiert die Erweiterung zur Segmentierung von Videos aus [25]. Im Anschluss daran wird in Abschnitt 5.3 der Algorithmus aus [22] beschrieben, welcher im Zuge der Segmentierung Disparitäten propagiert, welche zuvor von BenutzerInnen in Form von Scribbles zur Verfügung gestellt wurden. Abschnitt 5.4 diskutiert, wie die Laufzeit des Segmentierungs-Algorithmus aus [22] verringert werden kann, indem Teile des Algorithmus parallel auf der GPU ausgeführt werden. Abschnitt 5.5 präsentiert die daraus resultierende Laufzeitverringerung im Zuge eines Laufzeitvergleiches der optimierten und nicht optimierten Version des Algorithmus aus [22].

a) b)

Abbildung 5.1: *a) Eingabebild. b) Pixel mit ähnlichen Farbwerten werden im Zuge der Segmentierung in Regionen gruppiert. Zur Visualisierung wird jeder Region im Bild ein zufälliger Farbwert zugewiesen.*

5.1. Segmentierung von Bildern

Ziel der Bildsegmentierung ist es, Pixel in einem Eingabebild, welche ähnliche Farbwerte aufweisen, in zusammenhängende Regionen zu gruppieren (siehe Abbildung 5.1). Der Segmentierungs-Algorithmus aus [28] verwendet zu diesem Zweck eine Graphen-basierte Repräsentation des Bildes. Zu diesem Zweck wird das Bild als ungerichteter Graph $G = (V, E)$ repräsentiert, in welchem jedes Pixel des Eingabebildes einem Knoten $v_i \in V$ entspricht. Benachbarte Knoten v_i und v_j sind durch Kanten $e_{ij} \in E$ miteinander verbunden (siehe Abbildung 5.2). Jede Kante e_{ij} hat ein Gewicht w_{ij}, welches als Maß für die Verschiedenheit (*dissimilarity*) zwischen den verbundenen Knoten v_i und v_j dient. Das Gewicht einer Kante ergibt sich aus der absoluten Differenz der Intensitäten der verbundenen Knoten [28]:

$$w_{ij} = \left| I(v_i) - I(v_j) \right| \tag{5.1}$$

$I(v_i)$ entspricht der Intensität des Knotens v_i. Ein hohes Gewicht bedeutet, dass die verbundenen Knoten nur wenig Ähnlichkeit aufweisen, ein niedriges Gewicht bedeutet, dass die verbundenen

Knoten eine hohe Ähnlichkeit aufweisen. Vor der Berechnung der Kantengewichte kann das Eingabebild mittels Gauß-Filter [33] geglättet werden, um Aliasing-Artefakte zu entfernen. Eine Segmentierung *S* entspricht einer Gruppierung der Knoten *V* in zusammenhängende Regionen $R \in S$. Eine Region bildet somit einen Teilgraphen *G´(V, E´,)* welcher im Graphen *G(V, E)* enthalten ist. Im Allgemeinen sind Kanten innerhalb einer Region durch ein niedriges Kantengewicht gekennzeichnet, während Kanten zwischen Regionen ein hohes Gewicht aufweisen.

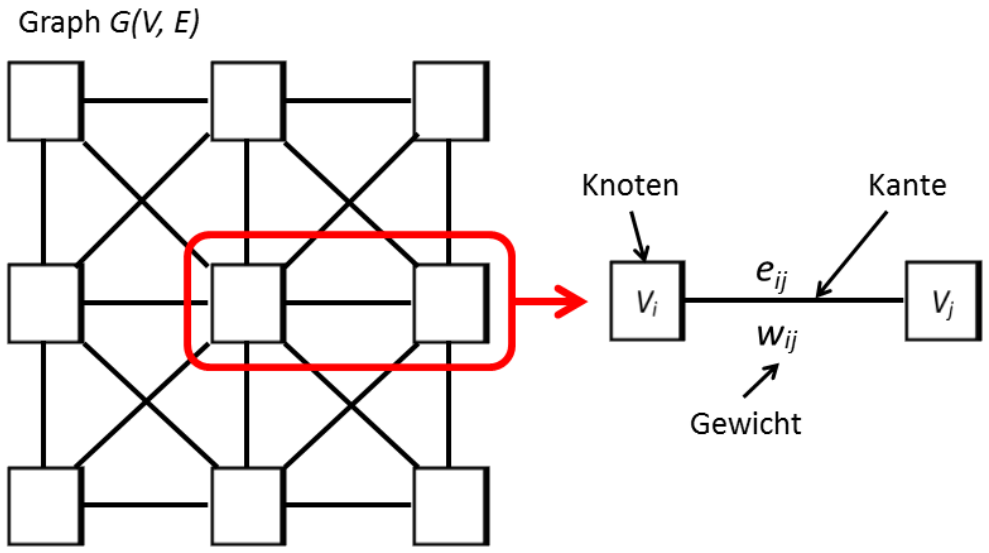
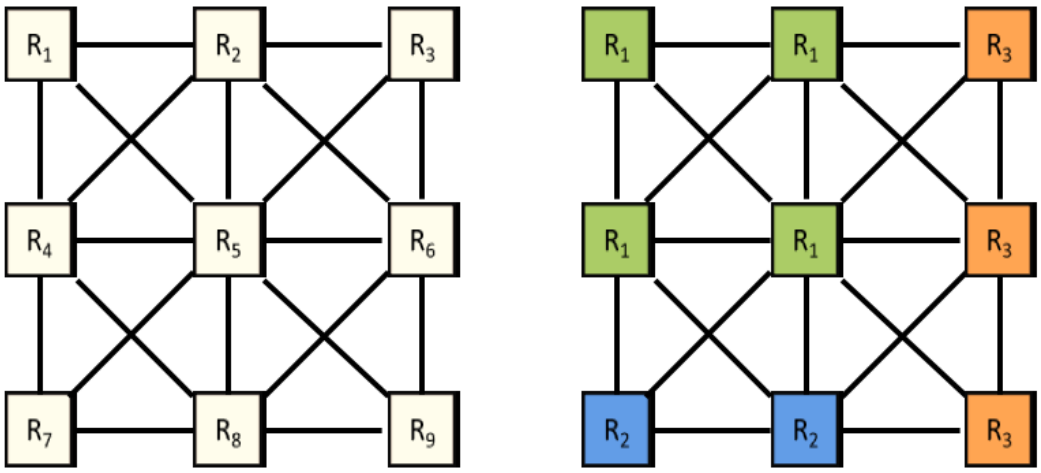


Abbildung 5.2: *Die Pixel eines Bildes werden als ein ungerichteter Graph repräsentiert. Benachbarte Pixel werden durch Kanten miteinander verbunden. Jede Kante erhält ein Gewicht, welches die Verschiedenheit der verbundenen Pixel repräsentiert.*



a) b)

Abbildung 5.3: *a) Zu Beginn bildet jeder Knoten im Graphen eine eigene Region. b) Im Zuge der Segmentierung werden ähnliche Regionen schrittweise zusammengefasst.*

Zu Beginn der Segmentierung bildet jeder Knoten des Graphen eine eigene Region (siehe Abbildung 5.3 a)). In [28] werden automatische Schwellwerte definiert, um zu beurteilen, ob zwischen zwei

benachbarten Bildregionen eine Bildkante existiert. Ist das nicht der Fall, werden die Regionen miteinander verbunden (siehe Abbildung 5.3 b)). Im Speziellen werden die Kantengewichte benachbarter Regionen mit den Differenzen innerhalb der Regionen (*interne Differenz*) verglichen. Sind die internen Differenzen zweier Regionen R_i und R_j größer als das Kantengewicht w_{ij}, werden die Regionen miteinander verbunden. Die interne Differenz *Int(R)* einer Region *R* ist als das größte Kantengewicht w_{ij} innerhalb des minimalen Spannbaumes *MST(R,E)* (siehe Abbildung 5.4) der Region definiert [28]:

$$\mathrm{Int}(R) = \max_{e \in MST(R,E)} w_{ij} \tag{5.2}$$

Ein Spannbaum *T* eines Graphen *G* ist ein Baum, für den gilt: *V(T) = V(G)* und *E(T) ⊆ E(G)*. Der Spannbaum *T* enthält somit dieselben Knoten wie der ursprüngliche Graph *G* und eine Teilmenge der Kanten von *G*. Ein minimaler Spannbaum von *G* ist ein Spannbaum, bei dem die Summe aller Kantengewichte minimal ist [55]. Der minimale Spannbaum bildet eine Region *R* innerhalb der Gesamtmenge von Kanten im Graphen mit minimalen Kantengewichten. Jede andere Region mit derselben Kantenanzahl innerhalb des Graphen enthält zumindest eine Kante mit einem Kantengewicht, das größer oder gleich der internen Differenz *Int(R)* der Region *R* ist. Die interne Differenz entspricht somit der maximalen Farbdifferenz (also dem größten Kantengewicht *w*) innerhalb der Region *R* [28].

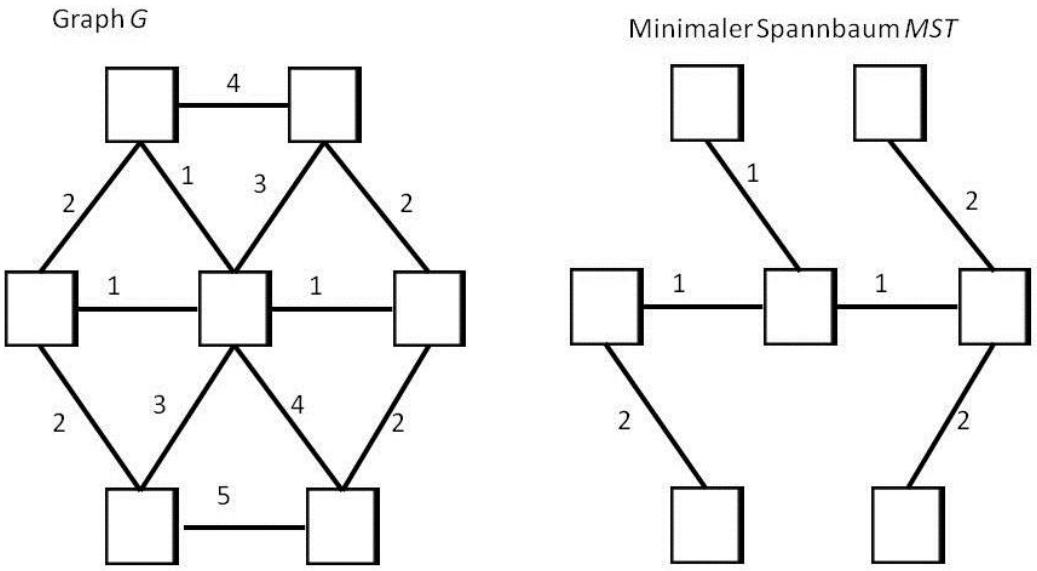


Abbildung 5.4: *a) Graph G mit unterschiedlichen Kantengewichten. b) Minimaler Spannbaum MST des Graphen G. MST enthält dieselben Knoten wie G und eine Teilmenge der Kanten von G. Die Summe der Kantengewichte in MST ist minimal. Nach* [55]

Mithilfe der internen Differenzen wird überprüft, ob es Hinweise auf eine Grenze zwischen zwei, durch eine Kante e_{ij} verbundene Regionen R_i und *Rj* gibt. Zu diesem Zweck wird das Kantengewicht w_{ij} mit den internen Differenzen $Int(R_i)$ und $Int(R_j)$ verglichen. Überschreitet w_{ij} eine der beiden internen Differenzen, wird angenommen, dass es eine Grenze zwischen den Regionen gibt [28]:

$$wenn\ w_{ij} \leq Int(R_i) \wedge w_{ij} \leq Int(R_j) \rightarrow\ verbinden$$
$$wenn\ w_{ij} > Int(R_i) \vee w_{ij} > Int(R_j) \rightarrow\ nicht\ verbinden \quad (5.3)$$

Da zu Beginn der Segmentierung alle Knoten eine eigene Region bilden, gibt es zu diesem Zeitpunkt keine Kanten innerhalb der Regionen. Es ist daher in diesem Fall nicht möglich, die interne Differenz einer Region zu berechnen. In [28] wird dieses Problem gelöst, indem die interne Differenz einer Region von der Regionsgröße abhängig gemacht wird. Zu diesem Zweck wird die interne Differenz einer Region R_i um einen Schwellwert $\tau(R_i)$ erweitert:

$$\tau(R_i) = \frac{k}{|R_i|} \quad (5.4)$$

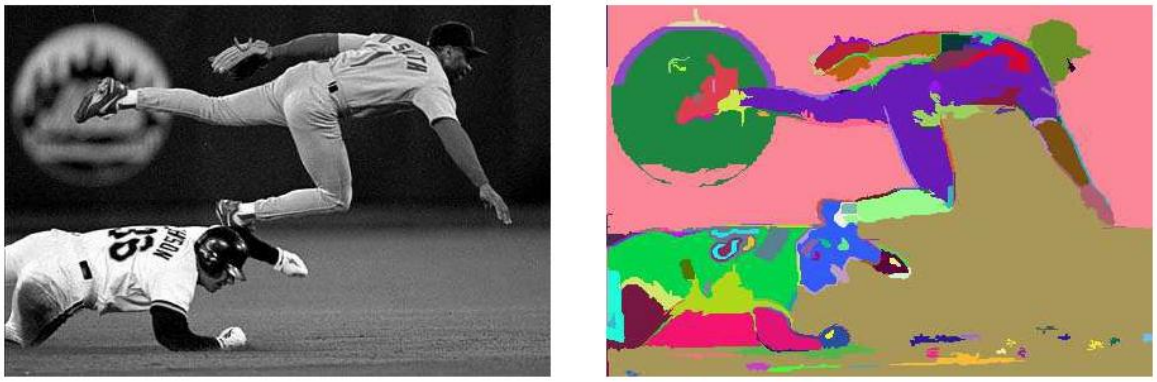
Hier gibt $|R_i|$ die Größe einer Region R_i an. Bei k handelt es sich um einen konstanten Parameter, welcher die Größe der entstehenden Regionen beeinflusst. Der Schwellwert $\tau(R_i)$ kontrolliert, wie groß das Kantengewicht im Verhältnis zu den internen Differenzen sein muss, damit angenommen werden kann, dass es eine Grenze zwischen den beiden Regionen gibt. Je größer k gewählt wird, desto größer muss das Kantengewicht im Vergleich zu den internen Differenzen sein, damit eine Grenze zwischen den Regionen erkannt wird.

Im Algorithmus aus [28] wird die Segmentierung eines Graphen $G = (V, E)$ mit n Knoten und m Kanten in drei Schritten generiert:

1. Begonnen wird mit der Segmentierung S^0, in der jeder Knoten v_i eine eigene Region darstellt.

2. Die Kanten E werden in aufsteigender Reihenfolge sortiert.

3. Für alle $q = 1 \ldots m$ wird S^q aus S^{q-1} ermittelt. Wenn sich zwei verbundene Knoten v_i und v_j in unterschiedlichen Regionen von S^{q-1} befinden und das Kantengewicht w_{ij} kleiner ist als die internen Differenzen $Int(R_i)$ und $Int(R_j)$, werden die beiden Regionen miteinander verbunden. Ansonsten bleiben die beiden Regionen bestehen ($S^q = S^{q-1}$).



a) b)

Abbildung 5.5: *Segmentierung eines Grauwertbildes mit der Auflösung 432 x 294. a) Eingabebild. b) Segmentierungsergebnis. Jeder Region wurde eine zufällig gewählte Farbe zugewiesen. Die Region des Rasens enthält Teile der Wand, was auf die langsamen Änderungen in den Intensitäten dieser beiden Bildregionen zurückzuführen ist. (σ = 0.8, k =300).* [28]

Die Laufzeit des Segmentierungs-Algorithmus für einen Graphen mit *m* Kanten beträgt *O(m log m)* [28]. Abbildung 5.5 zeigt das Ergebnis der Segmentierung eines Grauwertbildes. Wie im Segmentierungsergebnis in Abbildung 5.5 b) zu sehen ist, enthält die Region des Rasens Teile der Wand im Hintergrund. Das ist darauf zurückzuführen, dass zwischen diesen beiden Regionen eine langsame, fließende Änderung in den Intensitätswerten stattfindet. Aus diesem Grund wurde im Zuge der Segmentierung keine Grenze zwischen den Regionen gefunden. [28]

5.2. Segmentierung von Videos

In [25] wird der Segmentierungs-Algorithmus aus [28] zur Segmentierung von Videos erweitert. Zu Beginn wird ein volumetrischer Pixel-Graph *G(V, E)* erzeugt, in welchem jedes Pixel *i* des Videos einem Knoten $v_i \in V$ im Graphen entspricht. Die Knoten sind durch zeitliche und räumliche Kanten $e_{ij} \in E$ miteinander verbunden. Räumliche Kanten verbinden benachbarte Knoten innerhalb eines Frames. Zeitliche Kanten verbinden benachbarte Knoten in aufeinanderfolgenden Frames (siehe Abbildung 5.6). Jede Kante hat ein Gewicht w_{ij}, welches abhängig von der Ähnlichkeit der Farbwerte der verbundenen Pixel v_i und v_j ist.

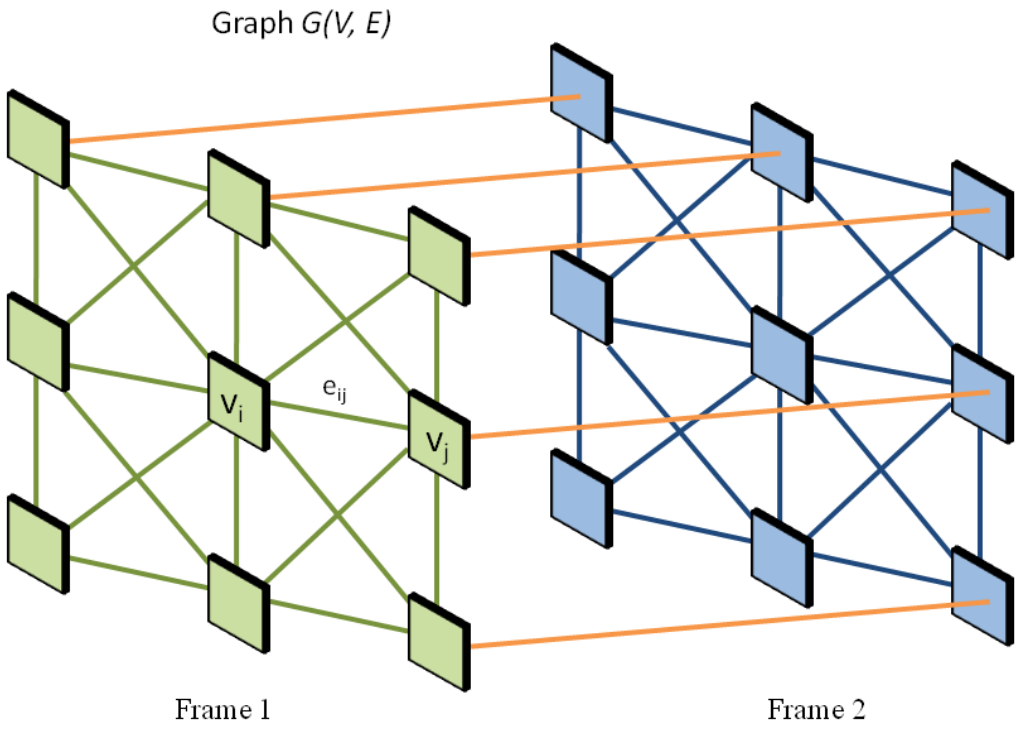


Abbildung 5.6: *Volumetrischer Pixel-Graph. Benachbarte Pixel innerhalb desselben Frames werden mittels räumlicher Kanten miteinander verbunden, benachbarte Pixel in aufeinanderfolgenden Frames durch zeitliche Kanten.*

Theoretisch könnte der Segmentierungs-Algorithmus aus [28] direkt auf den volumetrischen Pixel-Graphen angewandt werden. Jedoch ergeben sich daraus laut [25] folgende Probleme:

1. Der Schwellwert τ aus Formel (5.4) steuert die Größe der Regionen, welche im Zuge der Segmentierung entstehen. Die Erhöhung von τ führt zu größeren Regionen im

Segmentierungsergebnis, aber auch zu inkonsistenten Regionsgrenzen. Wird τ kleiner gewählt, ergibt sich eine konsistente Übersegmentierung mit einer geringen Regionsgröße. In [25] wird die gewünschte durchschnittliche Regionsgröße erst im Anschluss an die Segmentierung gewählt.

2. Die interne Differenz $Int(R_i)$ einer Region R_i unterscheidet zuverlässig zwischen homogenen und texturierten Bildbereichen. Da $Int(R_i)$ der maximalen Farbdifferenz innerhalb der Region entspricht (siehe Abschnitt 5.1), wird dieses Maß mit steigender Größe der Regionen jedoch zunehmend unzuverlässiger bezüglich der Unterscheidung zwischen ähnlichen Regionen. In [25] wird daher anstatt eines Maßes, das auf einzelnen Pixeln basiert, ein Maß verwendet, das auf Regionen basiert.

In der ersten Phase der Segmentierung wird der volumetrische Pixel-Graph im Zuge einer Übersegmentierung unter Anwendung des Segmentierungs-Algorithmus aus [28] mit einem kleinen Wert für τ (in [25] τ = 0.02) in zeitliche und räumliche Regionen gruppiert. Zusätzlich wird die Annahme getroffen, dass Regionen eine minimale Größe C_{min} haben. Regionen, welche diese Mindestgröße unterschreiten, werden so lange um Knoten mit aufsteigendem Kantengewicht erweitert, bis die minimale Größe erreicht ist.

Im zweiten Teil der Segmentierung wird ein Regionen-Graph erzeugt, in welchem jede der im Zuge der Übersegmentierung entstandenen Regionen einem Knoten entspricht (siehe Abbildung 5.8). Benachbarte Regionen sind durch die Kanten e_{ij} miteinander verbunden. Für jede Kante wird ein Kantengewicht w_{ij} berechnet. Dazu wird von jeder der Regionen ein Histogramm im L*a*b Farbraum [56] (in weiterer Folge mit LAB abgekürzt) mit 20 Klassen (*Bins*) entlang jeder Dimension erstellt. Das LAB-Histogramm einer Region hat demzufolge 20 x 20 x 20 = 8000 Bins [25]. Die LAB-Histogramme dienen zum Vergleich der Ähnlichkeiten von benachbarten Regionen. Als Ähnlichkeitsmaß wird die χ^2 Distanz zwischen den LAB-Histogrammen P und Q der verbundenen Regionen verwendet [57]:

$$w_{ij} = \frac{1}{2}\sum_{b}\frac{(P_b - Q_b)^2}{(P_b + Q_b)} \tag{5.5}$$

P_b und Q_b entsprechen dem normalisierten Wert des Bins b der Histogramme P und Q. Im Gegensatz zu lokalen Ähnlichkeitsmaßen pro Pixel, wie sie im Segmentierungs-Algorithmus für Bilder zum Einsatz kommen (siehe Abschnitt 5.1), liefern Ähnlichkeitsmaße, welche auf Histogrammen basieren, mehr Informationen zur Verteilung der Farbwerte innerhalb der entsprechenden Regionen. Beispielsweise haben texturierte Bildregionen eher flache Histogramme mit weit gestreuten Werten, während homogene Bildbereiche durch wenige hohe Werte und eine geringe Streuung gekennzeichnet sind (siehe Abbildung 5.7). Listing 5.1 enthält den Programmcode zur Berechnung der χ^2 Distanz zwischen zwei LAB-Histogrammen P und Q. Die LAB-Histogramme der entsprechenden Regionen werden Bin für Bin in einer Schleife miteinander verglichen. Für die Berechnung der χ^2 Distanz zwischen zwei LAB-Histogrammen mit 20 Bins pro Dimension werden somit 8000 Schleifendurchläufe durchgeführt. Die Berechnung der χ^2 Distanzen stellt somit den rechenintensivsten Teil des Segmentierungs-Algorithmus dar.

Der Segmentierungs-Algorithmus wird im nächsten Schritt auf den entstandenen Regionen-Graphen angewandt. Dieser Vorgang wird iterativ wiederholt, wodurch eine Segmentierungs-Hierarchie von räumlichen und zeitlichen Segmentierungen entsteht, welche in einer Baumstruktur gespeichert wird.

Die minimale Größe C_{min} der Regionen, sowie der Parameter τ werden in jeder Iteration um einen konstanten Faktor s (in [25] s = 1.1) erhöht. Die Regionsgrößen erhöhen sich demzufolge mit jeder Hierarchiestufe, wobei Regionsgrenzen erhalten bleiben (siehe Abbildung 5.9). Durch die Kombination der Übersegmentierung des volumetrischen Pixel-Graphen und der iterativen Segmentierung des Regionen-Graphen können zeitliche und räumliche Regionen erhalten werden, welche temporär kohärent sind. [25]

a) b)

Abbildung 5.7*: a) Homogene Bildregionen sind durch Histogramme mit wenigen hohen Werten gekennzeichnet. b) Texturierte Bildbereiche haben flache Histogramme mit weit gestreuten Werten.*

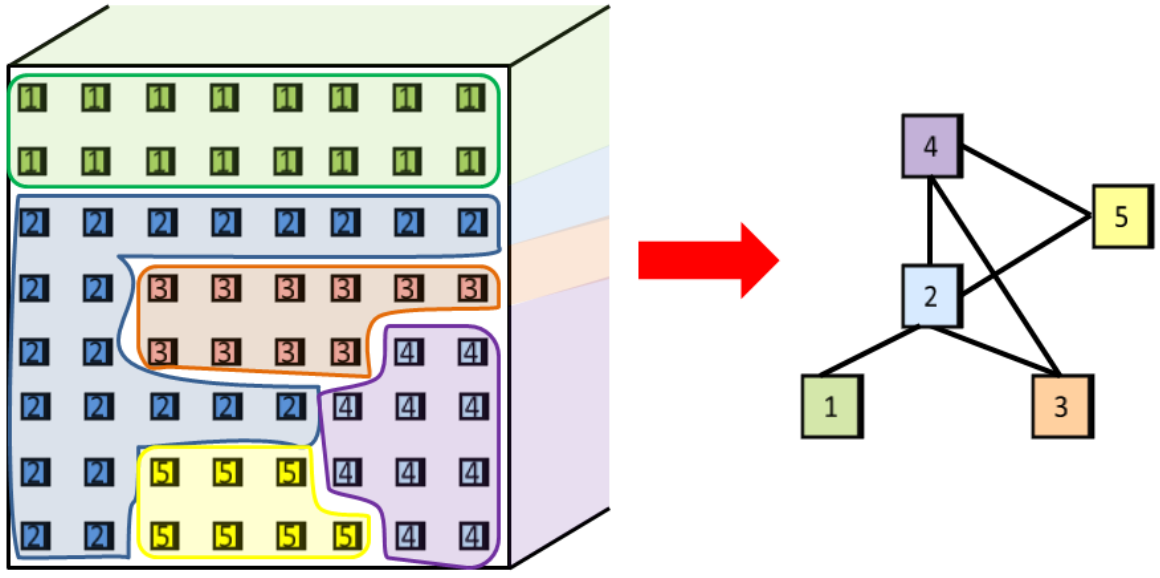


Abbildung 5.8: *Jeder Knoten im Regionen-Graphen entspricht einer Region, welche im Zuge der Übersegmentierung entstanden ist. Zeitlich und räumlich benachbarte Regionen sind durch Kanten miteinander verbunden.*

a)

b)

c)

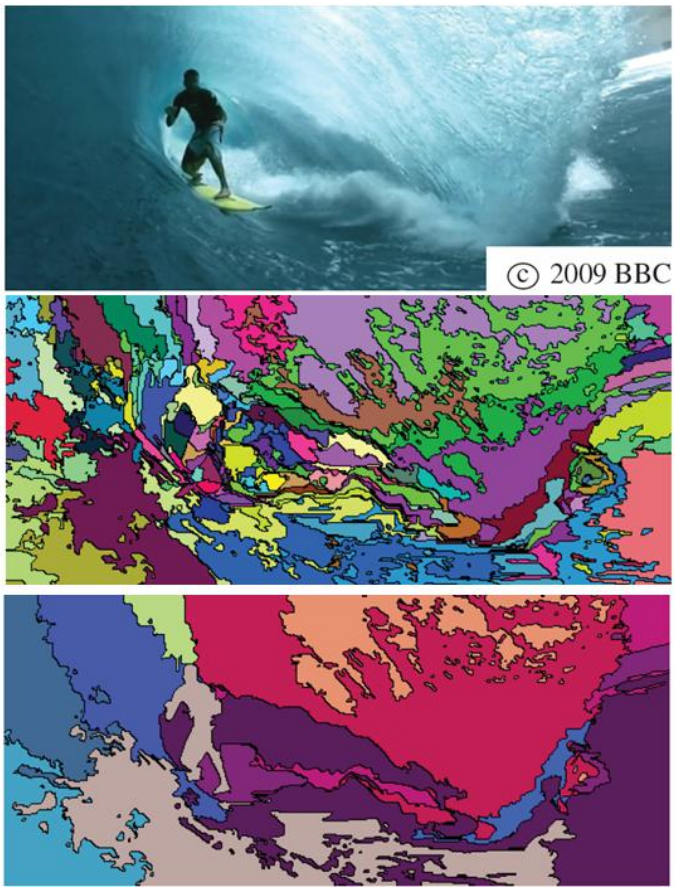


Abbildung 5.9: *a) Frame des Eingabevideos. b) Ergebnis der Übersegmentierung. c) Segmentierungsergebnis nach mehreren Iterationen.* [25]

Eingabewerte: LAB-Histogramme *P* und *Q* der verbundenen Regionen
Ausgabewert: Normalisierte χ^2 Distanzen zwischen den LAB-Histogrammen

```
1    double getNormChiQDistance(histogram *P, LABdescriptor *Q){
2        double dist=0;
3        double sum_bins=0;
4        double sub_bins=0;
5        int bin = 20;
6
7        for(int i=0;i<bins*bins*bins;i++){
8
9            sum_bins = (P->getCount(i)/(P->getSize())
10                    +(Q->getCount(i)/(Q->getSize());
11
12           sub_bins = (P->getCount(i)/(P->getSize())
13                    -(Q->getCount(i)/(Q->getSize()));
14
15           if(sub_bins != 0 && sum_bins != 0)
16               dist += square(sub_bins)/sum_bins;
17       }
18
19       return dist/2;
20   }
```

Listing 5.1: *Programmcode zur Berechnung der χ^2 Distanzen zwischen den normalisierten LAB-Histogrammen P und Q. Nach* [25]

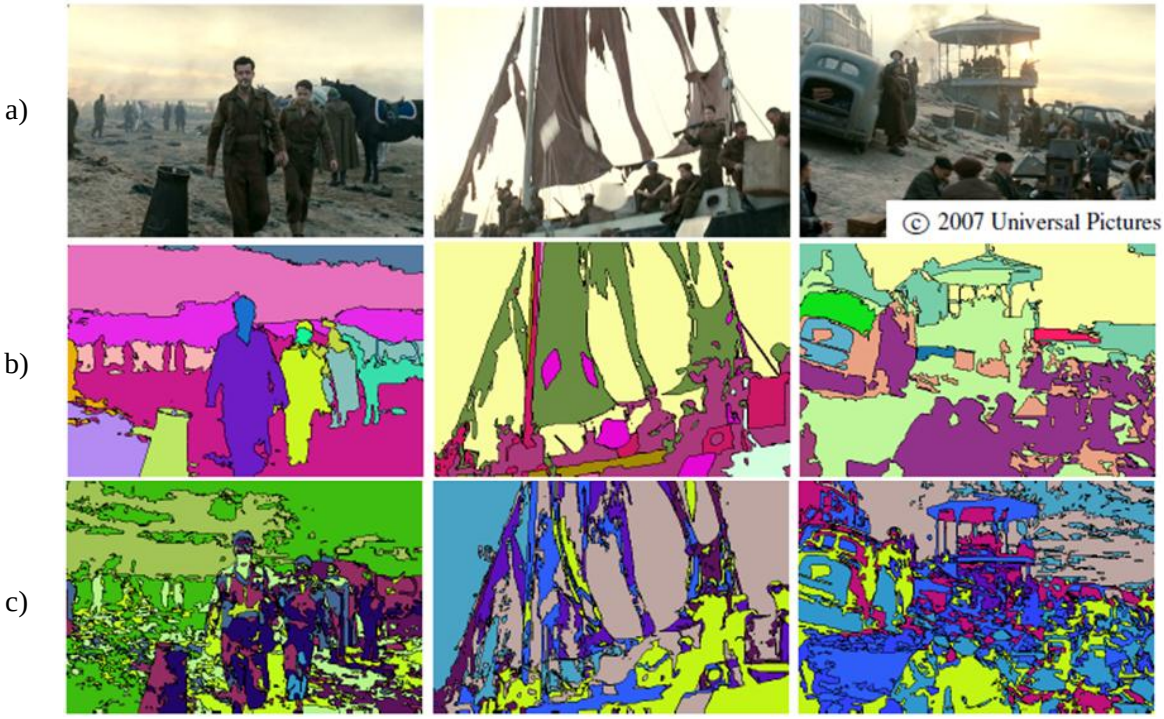


Abbildung 5.10: *a) Originale Frames des Eingabevideos. b) Segmentierungsergebnisse unter Verwendung von Optical Flow aus* [58]. *c) Segmentierungsergebnisse ohne Optical Flow.* [25]

Optical Flow

Durch die zusätzliche Verwendung von Bewegungsinformationen (*Optical Flow*) können die Segmentierungsergebnisse verbessert werden (siehe Abbildung 5.10). In [25] wird zu diesem Zweck Optical Flow aus [58] verwendet. Der Optical Flow findet in zwei Bereichen Anwendung:

1. Anstatt jedes Pixel des volumetrischen Pixel-Graphen mit seinen direkten Nachbarn in benachbarten Frames zu verbinden, wird es mit seinen Nachbarn entlang der Bewegungsvektoren verbunden. Finden zwischen Frames Bewegungen statt, fließen diese somit direkt in die Struktur des Pixel-Graphen ein.

2. Der Optical Flow wird als zusätzliches Merkmal zur Berechnung der Kantengewichte e_{wr} verwendet. Zusätzlich zu den LAB-Histogrammen werden für alle Regionen Flow-Histogramme erstellt. Diese haben jeweils 16 Bins, wobei jedes Bin mit dem Wert der Flow-Vektoren aufgerechnet wird. Die Berechnung der χ^2 Distanz zwischen zwei Flow-Histogrammen erfolgt analog zur Berechnung der χ^2 Distanz zwischen LAB-Histogrammen (siehe Listing 5.1). Die Regionen-Kantengewichte e_{wr} berechnen sich in diesem Fall durch eine Kombination der χ^2 Distanzen der normalisierten LAB-Histogramme $d_c \in [0,1]$ und der normalisierten Flow-Histogramme $d_f \in [0,1]$. Die resultierenden Gewichte e_{wr} sind nahe 0 für Regionen mit ähnlichen Optical Flow- und Farbwerten, anderenfalls sind sie nahe 1 [25]:

$$e_{wr} = (1 - (1 - d_c)(1 - d_f))^2 \tag{5.6}$$

Abbildung 5.10 zeigt einen Vergleich der Segmentierungsergebnisse mit (siehe Abbildung 5.10 b)) und ohne Verwendung von Optical Flow (siehe Abbildung 5.10 c)). Die Abbildung zeigt, dass die zusätzliche Verwendung von Optical Flow zu konsistenteren Ergebnissen führt, was vor allem an den Regionsgrenzen zu sehen ist. In weiterer Folge wird davon ausgegangen, dass Optical Flow zur Berechnung der Kantengewichte e_{wr} verwendet wird.

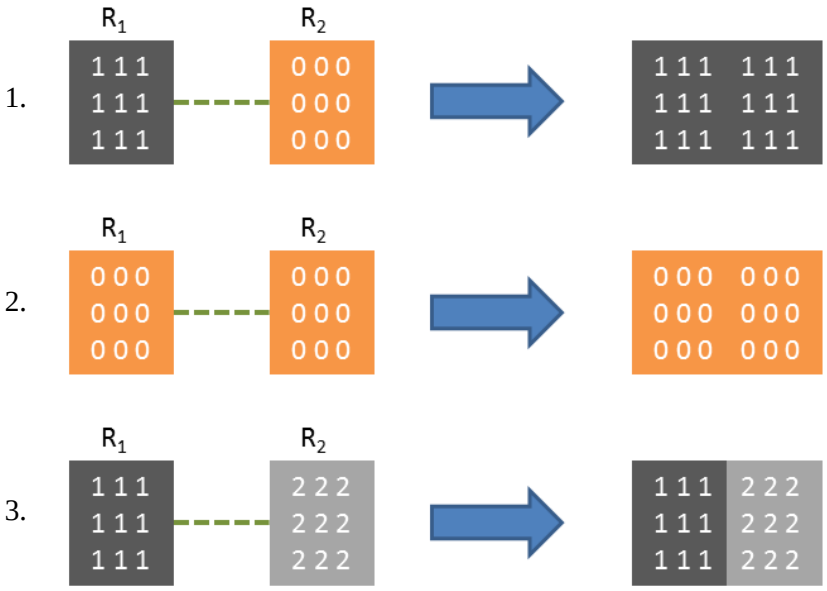


Abbildung 5.11: *Propagierungsregeln bei der Kombination von zwei Regionen R_1 und R_2.*

5.3. Propagierung von Disparitäten in Videos

In [22] wird der Segmentierungs-Algorithmus aus [25] um die semi-automatische Propagierung von Disparitäten in Videos erweitert. BenutzerInnen zeichnen dazu Scribbles im ersten und letzten Frame (*Schlüssel-Frames*) des zu konvertierenden Videos ein. Die Farbe der Scribbles codieren die Disparitäten der Pixel, welche in ihnen enthalten sind (siehe Kapitel 1). Jedem Pixel, das sich innerhalb eines Scribbles befindet, wird eine konkrete Disparität zugewiesen. Die Disparitäten werden im Zuge des Segmentierungs-Algorithmus (siehe Abschnitt 5.2) über das gesamte Video propagiert. In der ersten Phase entspricht jedes Pixel einer eigenen Region. Werden zwei Regionen miteinander verbunden, werden gleichzeitig die Disparitäten der entsprechenden Pixel propagiert. Abbildung 5.12 illustriert die Ergebnisse der Propagierung im Zuge des Segmentierungs-Algorithmus. Die Propagierung zwischen zwei Regionen R_1 und R_2 erfolgt anhand folgender Regeln (siehe Abbildung 5.11):

1. Enthält R_1 eine Disparität und R_2 nicht, wird die Disparität von R_1 nach R_2 propagiert. Es resultiert eine Region, in welcher alle Pixel die Disparität aus R_1 enthalten.

2. Enthalten weder R_1 noch R_2 eine Disparität, resultiert daraus eine Region ohne Disparität.

3. Enthält sowohl R_1 als auch R_2 eine Disparität, behalten alle beteiligten Pixel ihre Disparitäten. Eine Region kann demzufolge mehr als eine Disparität enthalten. Das wird mit einem Beispiel

in Abbildung 5.12 illustriert. Die blaue Region im rechten Bildbereich in Abbildung 5.12 e) wurde in der Disparitätskarte in Abbildung 5.12 f) in Subregionen mit unterschiedlichen Disparitäten geteilt, wodurch beispielsweise die Disparitäten der Lampe erhalten blieben (rot markiert).

In der zweiten Phase wird ein Regionen-Graph generiert, in welchem jede einzelne der Regionen, die in der vorangegangenen Phase entstandenen sind, einem Knoten entspricht. Im Gegensatz zum Algorithmus aus [25] haben die Kanten des Regionen-Graphen jeweils zwei Kantengewichte:

1. Ein Pixel-Kantengewicht $e_{wp,}$ welches der Ähnlichkeit der Farbwerte zwischen den Pixeln an den Regionsgrenzen entspricht (es handelt sich dabei um das Kantengewicht *w* aus Abschnitt 5.2)

2. Ein Regionen-Kantengewicht e_{wr}, welches anhand der χ^2 Distanzen der LAB- und Flow-Histogramme aus Formel (5.6) berechnet wird.

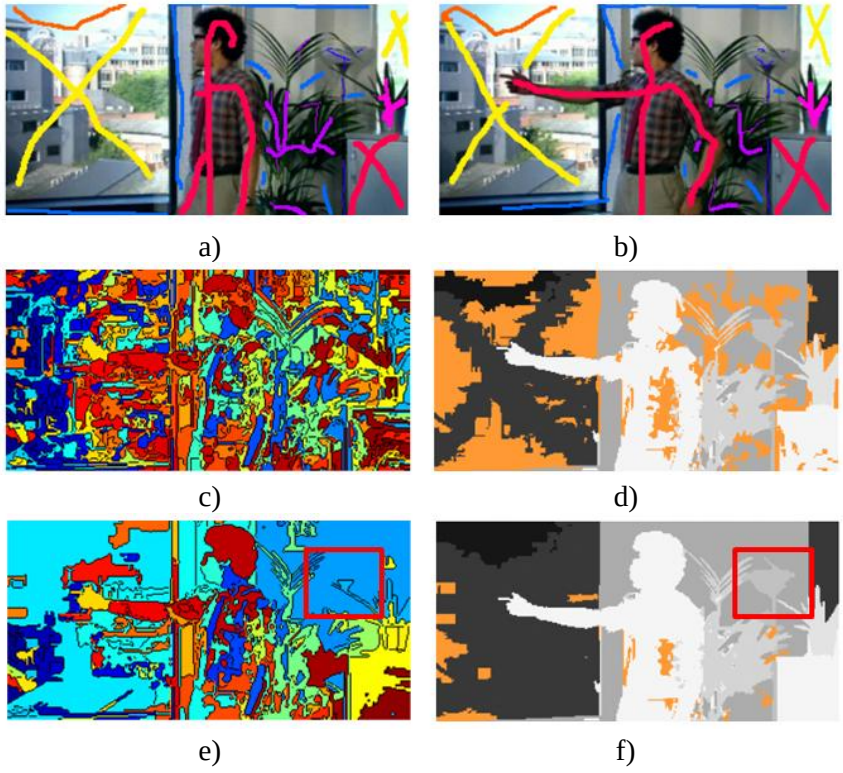


Abbildung 5.12: *Propagierung von Disparitäten im Zuge der Segmentierung. a) Scribbles des ersten Frames. b) Scribbles des letzten Frames. c) Ergebnis der Übersegmentierung. d) Ergebnis der Propagierung im Zuge der Übersegmentierung. Regionen ohne Disparitäten sind orange gekennzeichnet. e) Segmentierungsergebnis nach Segmentierung des Regionen-Graphen. f) Ergebnis der Propagierung im Zuge der Segmentierung des Regionen-Graphen. Regionen ohne Disparitäten (orange) wird anschließend eine Disparität zugewiesen. Nach* [22]

Der Segmentierungs-Algorithmus wird iterativ auf den Regionen-Graphen angewandt. Die Propagierung der Disparitäten erfolgt im Zuge dessen analog zur vorigen Phase. Enthält eine Region mehrere unterschiedliche Disparitäten, wird jene Disparität des ähnlichsten Randpixels zwischen den

Regionen propagiert. Zu diesem Zweck werden die Kanten vor Beginn der Segmentierung zuerst nach ihren Regionen-Kantengewichten e_{wr} und anschließend nach ihren Pixel-Kantengewichten e_{wp} sortiert. Regionen, welche am Ende des Segmentierungs-Algorithmus keine Disparität enthalten, wird durch iterative Kombination von Kanten mit aufsteigendem Kantengewicht eine Disparität zugewiesen. Das Ergebnis der Propagierung wird in einem weiteren Schritt optimiert. Dieser Nachbearbeitungsschritt wird in Kapitel 6 behandelt.

5.4. Optimierung der Segmentierung und Propagierung

Dieser Abschnitt präsentiert eine optimierte Version der C++-Implementierung aus [22], in welcher die Laufzeit verringert wird, indem Teile des zugrunde liegenden Segmentierungs-Algorithmus aus Abschnitt 5.2 auf der GPU ausgeführt werden. Im Folgenden wird diskutiert, wie die einzelnen Schritte des Segmentierungs-Algorithmus im Zuge dessen optimiert werden. Mit Ausnahme von Schritt 3 wird die Optimierung durch eine Verlagerung der einzelnen Teilalgorithmen auf die GPU erreicht, wo diese parallel bearbeitet werden. Mit Ausnahme einiger simpler Änderungen, welche aufgrund von Voraussetzungen der CUDA-Architektur notwendig sind, bleiben die Algorithmen dabei unverändert. Die Optimierung in Schritt 3 ist hingegen komplexer und wird daher im nachfolgenden Abschnitt genauer behandelt.

1. *Generierung des Pixel-Graphen:*
   Zur Generierung des volumetrischen Pixel-Graphen wird jedes Pixel mit seinen räumlichen und zeitlichen Nachbarn durch Kanten verbunden, wobei für jede Kante ein Kantengewicht berechnet wird, das der Differenz der Farbwerte der verbundenen Pixel entspricht. Die Bearbeitung eines Pixels ist unabhängig von den anderen Pixeln im Video. Es können somit alle Pixel parallel bearbeitet werden. Die Kanten zwischen einem Pixel und seinen Nachbarn und die entsprechenden Kantengewichte werden für jedes Pixel in einem eigenen Thread auf der GPU berechnet. Die entstehenden Kanten werden anschließend im Pixel-Graphen gespeichert.

2. *Segmentierung des Pixel-Graphen:*
   Die Segmentierung des Pixel-Graphen erfolgt durch iterative Kombination von immer größer werdenden Bildregionen (siehe Abschnitt 5.1). In jedem Segmentierungsschritt werden die Ergebnisse des vorangegangenen Schrittes benötigt. Dieser Teil des Algorithmus eignet sich aus diesem Grund nicht für eine parallele GPU-Implementierung.

3. *Generierung des Regionen-Graphen:*
   Bei der Generierung des Regionen-Graphen handelt es sich um den rechenintensivsten Teil des Segmentierungs-Algorithmus. Aus einer Optimierung dieses Abschnitts folgt somit eine Optimierung des gesamten Segmentierungs-Algorithmus. Die Optimierung der Generierung des Regionen-Graphen wird im anschließenden Abschnitt diskutiert.

4. *Iterative Segmentierung des Regionen-Graphen:*
   Der Regionen-Graph wird iterativ, das heißt durch wiederholte Anwendung von Schritt 2 und 3, segmentiert. Die Generierung des Regionen-Graphen erfolgt parallel auf der GPU (Schritt 3). Die iterative Segmentierung des Regionen-Graphen wird seriell auf der CPU durchgeführt (Schritt 2).

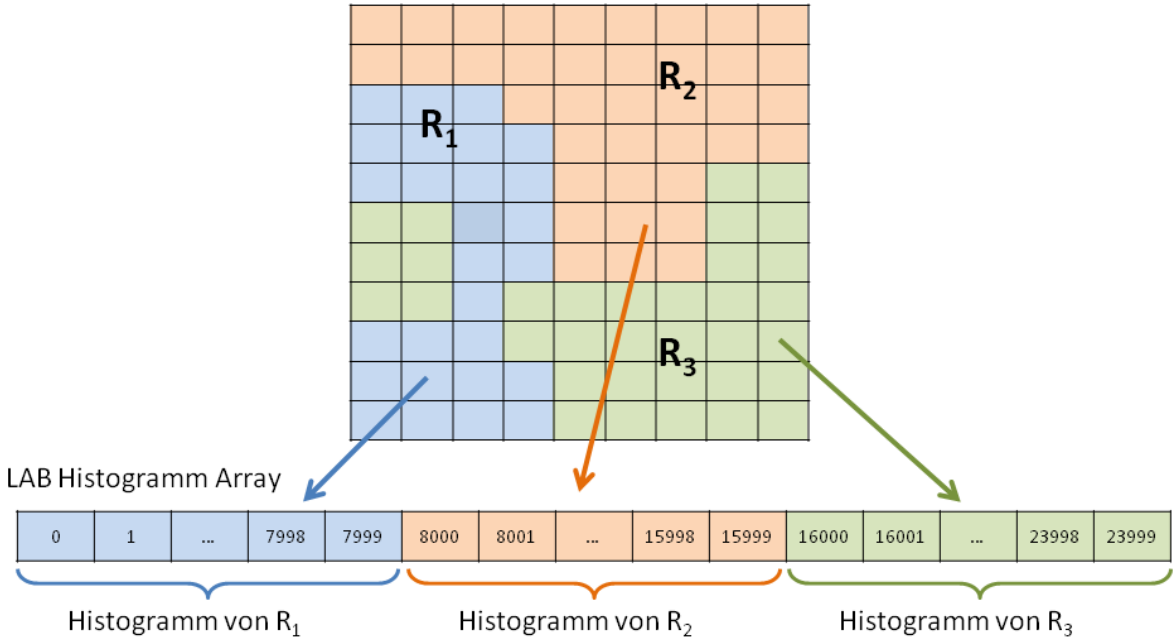


Abbildung 5.13: *Die LAB-Histogramme aller Regionen werden nacheinander in einem eindimensionalen Array auf dem Global-Memory der GPU gespeichert. Jedes LAB-Histogramm hat 20 x 20 x 20 = 8000 Bins.*

Generierung des Regionen-Graphen

Die Generierung des Regionen-Graphen erfolgt im Wesentlichen in fünf Schritten: 1. Berechnung der LAB- und Flow-Histogramme der Regionen des Videos. 2. Generierung der Kanten des Regionen-Graphen. 3. Berechnung der normalisierten χ^2 Distanzen der LAB- und Flow-Histogramme der verbundenen Regionen. 4. Berechnung der Kantengewichte e_{wr}.

Als Erstes werden die LAB- und Flow-Histogramme der Regionen generiert. Die Erstellung der LAB-Histogramme erfolgt parallel auf der GPU, wobei jedes Pixel in einem eigenen Thread bearbeitet wird. Zu diesem Zweck wird das gesamte Video vom RGB-Farbraum in den LAB-Farbraum konvertiert. Die Konvertierung erfolgt parallel auf der GPU, wobei jedes Pixel in einem eigenen Thread konvertiert wird. Da das resultierende LAB-Farbvideo in jeder Iteration benötigt wird, bleibt es während des gesamten Segmentierungs-Algorithmus auf dem Global-Memory der GPU gespeichert. Auf diese Weise ist nur ein einziger Konvertierungsvorgang notwendig. Anschließend werden die LAB-Histogramme für alle Regionen des Videos gebildet. Die LAB-Histogramme der einzelnen Regionen werden nacheinander in einem eindimensionalen Array im Global-Memory gespeichert (siehe Abbildung 5.13). Um die Pixel den richtigen Histogrammen zuordnen zu können, wird im Vorfeld ein Array erstellt, welches für jedes Pixel im Video angibt, zu welcher Region es gehört. Für jedes Pixel wird, abhängig von seinem LAB-Farbwert, in einem eigenen Thread das entsprechende Bin im LAB-Histogramm jener Region erhöht, zu welcher das Pixel gehört. In Abbildung 5.13 erfolgt die Erhöhung der Bins für Pixel in Region R_1 beispielsweise im ersten Histogramm (blau). Da alle Pixel parallel behandelt werden, kommt es beim Erhöhen der Bins zu Konflikten bei den Speicherzugriffen, da mehrere Threads gleichzeitig auf dieselbe Speicherstelle im Global-Memory schreiben. Wie in Kapitel 3 erwähnt, kann im Fall eines Schreibkonflikts zwischen zwei oder mehreren Threads nur ein Thread die Schreiboperation durchführen. Die Schreibzugriffe müssen bei der Erstellung der Histogramme somit synchronisiert werden. Die CUDA-Architektur stellt zu diesem

Zweck atomare Funktionen (*atomics*) zur Verfügung [24]. Die Verwendung dieser Funktionen stellt sicher, dass die Schreibzugriffe mehrerer Threads auf den Global-Memory synchronisiert werden. Zum Aufsummieren der Histogramm-Bins wird die Funktion *atomicAdd* [24] verwendet. Die CUDA-Architektur stellt zudem Funktionen zur parallelen Berechnung von Gleitkomma-Operationen zur Verfügung, unter anderem zur Multiplikation (*__fmul_rn*) und Division (*__fdividef*) von Gleitkommazahlen [24]. Diese wurden in dieser Diplomarbeit verwendet, um die Laufzeit dieser Operationen zu beschleunigen und somit eine Laufzeitreduktion der einzelnen Threads zu erreichen. Die Erstellung der Flow-Histogramme erfolgt analog zu den LAB-Histogrammen.

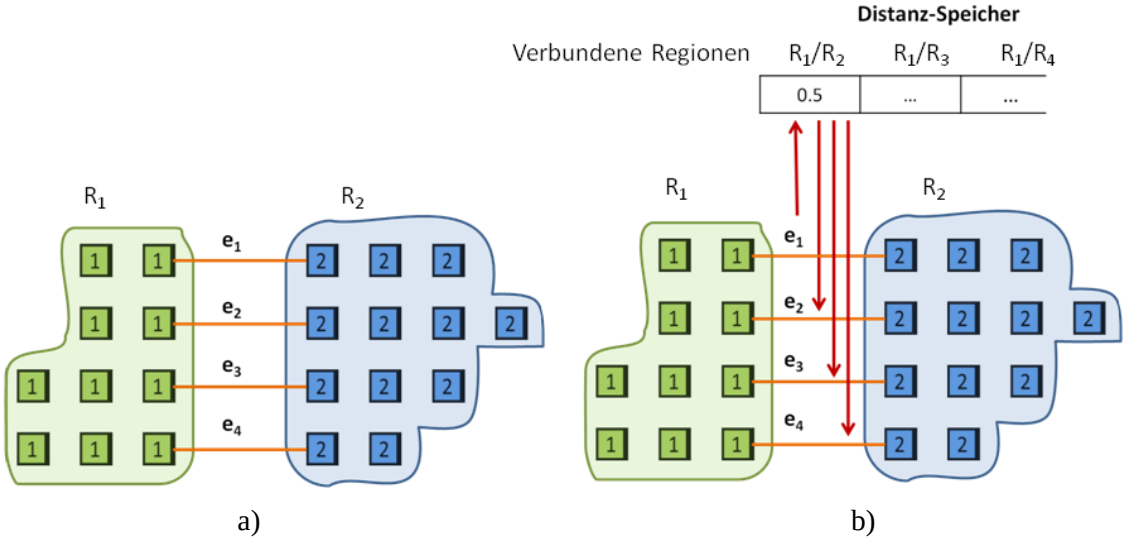


Abbildung 5.14: *a) Die Regionen R_1 und R_2 sind durch die Kanten e_1 bis e_4 verbunden. Zur Ermittlung der Kantengewichte werden die χ^2 Distanzen zwischen LAB- und Flow-Histogrammen der beiden Regionen berechnet. In der Implementierung von* [22] *werden die Distanzen für jede Kante getrennt ermittelt. b) In der optimierten Implementierung werden die Distanzen nur für eine der Kanten berechnet und anschließend im Distanz-Speicher zwischengespeichert. Zur Berechnung der Kantengewichte der restlichen Kanten werden die Distanzen anschließend aus dem Distanz-Speicher geladen. Die Anzahl der benötigten Distanzberechnungen wird somit minimiert.*

Im zweiten Schritt erfolgt die Generierung des Regionen-Graphen. Die Generierung der Kanten des Regionen-Graphen erfolgt unabhängig voneinander und kann somit parallel auf der GPU ausgeführt werden. Für jede Kante werden zwei Kantengewichte e_{wp} und e_{wr} berechnet. Das Gewicht e_{wp}, welches der Farbdifferenz zwischen den verbundenen Pixel entspricht, wurde bereits im Zuge der Segmentierung des Pixel-Graphen ermittelt (siehe Abschnitt 5.2) und kann ohne weitere Berechnungen übernommen werden. Das Gewicht e_{wr} (siehe Formel (5.6)) ergibt sich aus einer Kombination der normalisierten χ^2 Distanzen der LAB- und Flow-Histogramme der verbundenen Regionen. Wie in Abschnitt 5.2 diskutiert, stellt die Berechnung dieser Distanzen den rechenintensivsten Teil des Segmentierungs-Algorithmus dar. Die Optimierung, welche in dieser Diplomarbeit erstellt wurde, erfolgt einerseits durch die Minimierung der Anzahl der benötigten Distanzberechnungen und andererseits durch die Optimierung der Laufzeiten der einzelnen Distanzberechnungen. In der Implementierung von [22] wird das Kantengewicht e_{wr} im Zuge der Erstellung des Regionen-Graphen für alle Kanten getrennt berechnet. Beispielsweise werden in Abbildung 5.14 a) für zwei Regionen R_1 und R_2 die χ^2 Distanzen für alle Kanten e_1 bis e_4, welche die Knoten aus R_1 und R_2 miteinander verbinden, ermittelt. Da die resultierenden Distanzen für alle

Kanten e_1 bis e_4 identisch sind, genügt es jedoch, die Distanzberechnungen ein einziges Mal durchzuführen. Aus diesem Grund werden die bereits berechneten Distanzen in einem Array zwischengespeichert (*Distanz-Speicher*). Distanzen zwischen Regionen, welche bereits berechnet wurden, können somit aus dem Distanz-Speicher geladen werden und müssen nicht erneut berechnet werden (siehe Abbildung 5.14 b)). Der Distanz-Speicher wird in der Größe der maximalen Kombinationen zwischen den Regionen angelegt. Die Größe M des Distanz-Speichers kann für ein Video mit n Regionen mithilfe der Gauß'schen Summenformel [59] ermittelt werden:

$$M = n^2 - \frac{n(n+1)}{2} \tag{5.7}$$

Für ein Video mit vier Regionen beträgt die Größe des Distanz-Speichers somit $M = 6$ (siehe Abbildung 5.15). Der Index $d(i, j)$ der Distanz zwischen zwei Regionen R_i und R_j wird wie folgt berechnet:

$$d(i,j) = i + jM - \frac{(j^2 + j)}{2} \tag{5.8}$$

Die Generierung der Kanten des Regionen-Graphen erfolgt parallel auf der GPU, wobei jede Kante in einem eigenen Thread erstellt wird. Im Zuge der Kantengenerierung wird im Distanz-Speicher indiziert, welche Distanzen zur Berechnung der Kantengewichte benötigt werden. Der Distanz-Speicher wird zu diesem Zweck mit dem Wert -10 initialisiert (siehe Abbildung 5.15). Für jede Kante, welche zwei Regionen R_i und R_j miteinander verbindet, wird die entsprechende Speicherstelle $d(i, j)$ im Distanz-Speicher mit -1 indiziert. Die negativen Werte im Distanz-Speicher dienen ausschließlich dazu, zu kennzeichnen, welche Distanzen zu berechnet sind und fließen in keine der Distanzberechnungen ein. Die Ermittlung der Distanzen erfolgt im Anschluss parallel auf der GPU, wobei jeweils eine Distanzberechnung pro Thread durchgeführt wird. Im Zuge dessen werden alle Distanzen berechnet, welche im Distanz-Speicher den Wert -1 enthalten. Die Anzahl der benötigten Distanzberechnungen wird auf diese Weise minimiert. Um die Laufzeit der einzelnen Distanzberechnungen zu verringern, wird die Normalisierung (siehe Listing 5.1, Zeile 9-13) der Bin-Werte aller Histogramme vor der Distanzberechnung parallel auf der GPU vorberechnet. Die normalisierten Bin-Werte werden im Global-Memory zwischengespeichert und im Zuge der Distanzberechnungen ausgelesen. Im Anschluss an die Distanzberechnungen werden die Kantengewichte e_{wr} des Regionen-Graphen anhand von Formel (5.6) ermittelt. Die Berechnung der Kantengewichte erfolgt parallel auf der GPU, wobei jede Kante in einem eigenen Thread behandelt wird. Die vorher berechneten χ^2 Distanzen der LAB- und Flow-Histogramme werden im Zuge dessen aus dem Distanz-Speicher ausgelesen.

Zur weiteren Optimierung der Laufzeiten wurden Gleitkommaoperationen auf der GPU mit einfacher Genauigkeit (*single-precision* [24]) durchgeführt. In der Implementierung von [22] wurden die Berechnungen mit doppelter Genauigkeit (*double-precision* [24]) durchgeführt. Die Ergebnisse können aus diesem Grund von den Ergebnissen aus [1] abweichen, was auf daraus resultierende Rundungsfehler zurückzuführen ist. Der Grund, warum in dieser Diplomarbeit Gleitkommaoperationen auf der GPU mit einfacher Genauigkeit durchgeführt wurden, liegt einerseits darin, dass Berechnungen mit doppelter Genauigkeit in CUDA von Grafikkarten mit Compute Capability unter 1.3 nicht unterstützt werden und Gleitkommaoperationen in diesem Fall automatisch mit einfacher Genauigkeit durchgeführt werden. Grafikkarten ab Compute Capability 1.3 unterstützen Gleitkommaberechnungen mit doppelter Genauigkeit. Diese benötigen allerdings wesentlich länger als Berechnungen mit einfacher Genauigkeit (siehe [24]).

Regionen	R_1/R_2	R_1/R_3	R_1/R_4	R_2/R_3	R_2/R_4	R_3/R_4
Speicherstelle	0	1	2	3	4	5
Wert	-10	-1	-1	-10	-1	-1

Abbildung 5.15: *Distanz-Speicher für ein Video mit vier Regionen. Es gibt sechs mögliche Kombinationen zwischen den Regionen. Für jede dieser Kombinationen wird ein Eintrag im Distanz-Speicher erstellt. Jene Distanzen, welche für die Berechnung der Kantengewichte benötigt werden, enthalten den Wert -1. In diesem Beispiel werden die Distanzen R_1/R_3, R_1/R_4, R_3/R_4 und R_3/R_4 berechnet.*

Ergebnisse

Abbildung 5.16 und Abbildung 5.17 zeigen Ergebnisse der in dieser Diplomarbeit erstellten Implementierungen zur segmentierungsbasierten Propagierung von Disparitäten in Videos. Die Implementierung, die im Zuge dieser Diplomarbeit erstellt wurde, stellt eine Optimierung bezüglich der Laufzeit der C++-Implementierung des Algorithmus von [22] dar. Die resultierenden Disparitätskarten enthalten keine feinen Bilddetails (wie Haare), wie in Abbildung 5.16 c) im Bereich des Gesichts zu sehen ist. Zudem können abrupte zeitliche Disparitätsänderungen enthalten sein. Aus diesem Grund werden die Ergebnisse dieses ersten Bearbeitungsschrittes im darauffolgenden Schritt, welcher in Kapitel 6 diskutiert wird, weiter verbessert. Weitere Ergebnisse sind in Kapitel 7 zu finden.

a) b) c)

Abbildung 5.16: *Ergebnis der segmentierungsbasierten Propagierung für ein Video mit 10 Frames mit einer Auflösung von 340 x 170. a) Scribbles des ersten und letzten Frames des Videos. b) Frame 1, Frame 5 und Frame 10 des Eingabevideos. c) Resultierende Disparitätskarten (k = 0.02, s = 1.1, c_{min}= 110, 10 Iterationen).*

a) b) c)

Abbildung 5.17: *Ergebnisse für ein Video mit 20 Frames mit einer Auflösung von 600 x 255. a) Scribbles des ersten und letzten Frames. b) Frame 1, Frame 10 und Frame 20 des Eingabevideos. c) Resultierende Disparitätskarten (k = 0.02, s = 1.1, c_{min}= 200, 10 Iterationen).*

Aufteilen von Videos

Aufgrund der eingeschränkten Speicher-Kapazität des Global-Memory der Grafikkarte können nur eine bestimmte Anzahl an Frames, abhängig von der Auflösung des Videos, gleichzeitig auf der GPU bearbeitet werden. Um die Segmentierung und Propagierung von längeren Videos durchführen zu können, wird in der Implementierung, welche in dieser Diplomarbeit erstellt wurde, wie folgt vorgegangen: Überschreitet die Größe des Videos (in Pixel) einen Schwellwert, wird das Video in Subsequenzen aufgeteilt. Die Subsequenzen werden nacheinander bearbeitet und anschließend zu einem einzigen Video zusammengefasst. Um einen möglichst konsistenten Übergang zwischen den Subsequenzen zu gewährleisten, wird jeweils ein Frame als Überlappung eingeplant, unter der Annahme, dass zwischen den Frames eine lineare Bewegung stattfindet. Das letzte Frame einer Subsequenz stellt zugleich das erste Frame der folgenden Subsequenz dar (siehe Abbildung 5.18). Im Zuge der Segmentierung der ersten Subsequenz wird jedem Pixel ein Regions-Index zugeordnet, welcher angibt, zu welcher Region das entsprechende Pixel gehört. Beim Aufteilen werden diese Regions-Indizes für alle Pixel des letzten Frames einer Subsequenz an die entsprechenden Pixelpositionen des ersten Frames der Folgesequenz kopiert. (siehe Abbildung 5.19). Damit die Disparitäten über das gesamte Video propagiert werden können, werden jeweils im ersten und im letzten Frame einer Subsequenz die vordefinierten Scribble-Disparitäten benötigt. Diese stehen allerdings nur im ersten und im letzten Frame des gesamten Videos zur Verfügung, wo sie von BenutzerInnen eingezeichnet wurden (siehe Abschnitt 5.3). Um die Scribble-Disparitäten in einem dazwischenliegenden Frame zu erhalten, werden die Disparitäten aus dem ersten und letzten Frame auf das dazwischenliegende Frame propagiert (siehe Abbildung 5.20). Die Disparitäten dieser Schlüssel-Frames fließen dabei mit unterschiedlicher Gewichtung ein, abhängig von ihrer Entfernung zum dazwischenliegenden Frame. Im Beispiel in Abbildung 5.20 ist der Abstand von *Frame 5* zu beiden Schlüssel-Frames gleich groß. Die Disparitäten fließen somit mit gleicher Gewichtung ein.

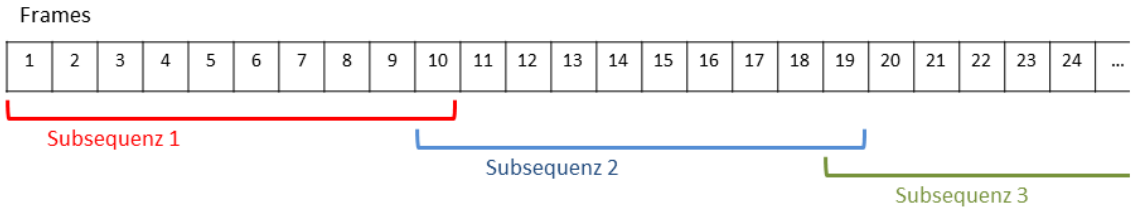


Abbildung 5.18: *Längere Videos werden in Subsequenzen aufgeteilt, welche nacheinander bearbeitet werden. Zwischen den Subsequenzen gibt es jeweils eine Überlappung von einem Frame.*

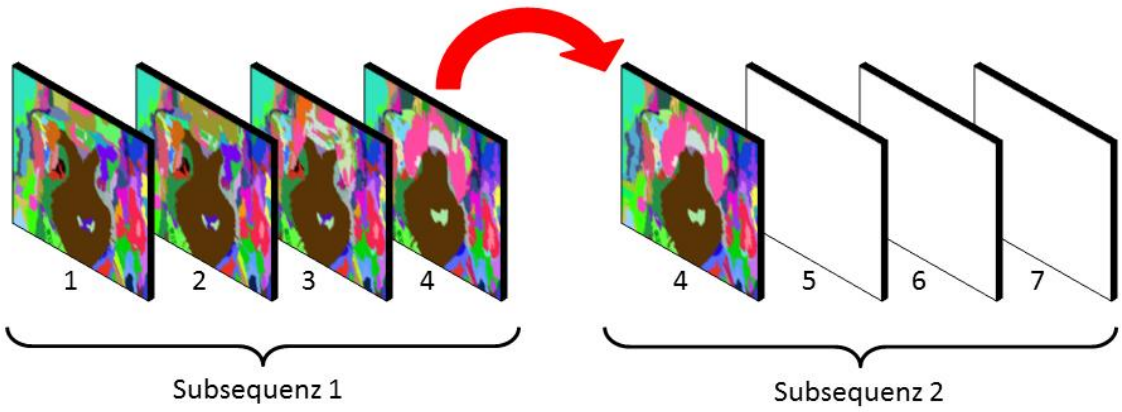


Abbildung 5.19: *Die Regions-Indizes (hier mit zufälligen Farben codiert) werden pixelweise vom letzten Frame einer Subsequenz auf das erste Frame der folgenden Subsequenz kopiert, um einen konsistenten Übergang zwischen den Subsequenzen zu gewährleisten.*

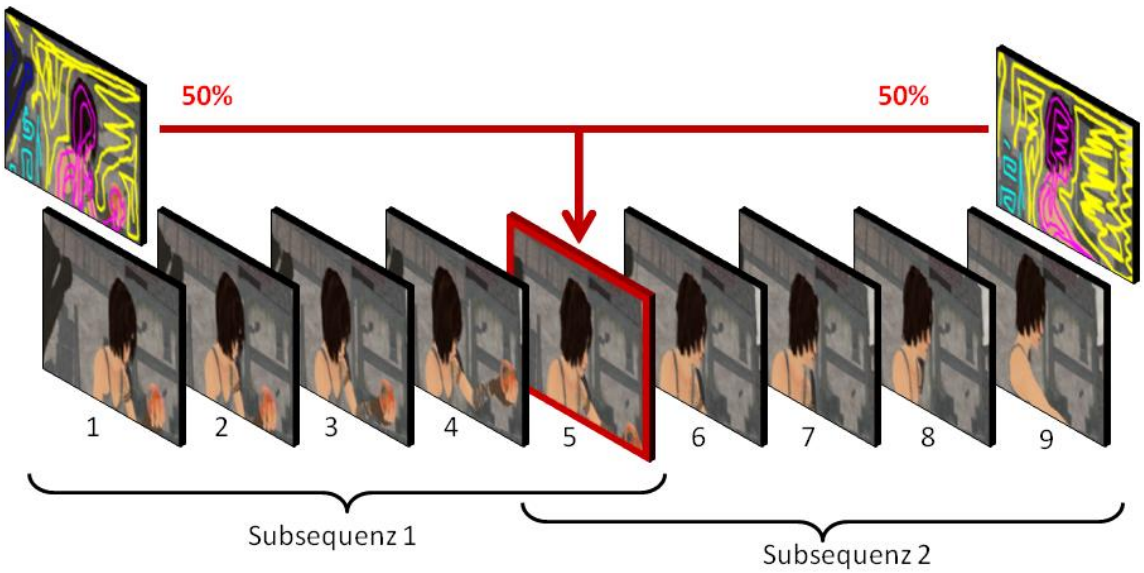


Abbildung 5.20: *Um die Disparitäten über das gesamte Video propagieren zu können, werden Scribble-Disparitäten in Frame 5 benötigt. Um diese zu erhalten, werden die vorgegebenen Disparitäten aus dem ersten und letzten Frame des Videos auf Frame 5 propagiert. Der Abstand von Frame 5 zu beiden Schlüssel-Frames ist gleich groß. Die Disparitäten fließen somit zum gleichen Teil in Frame 5 ein.*

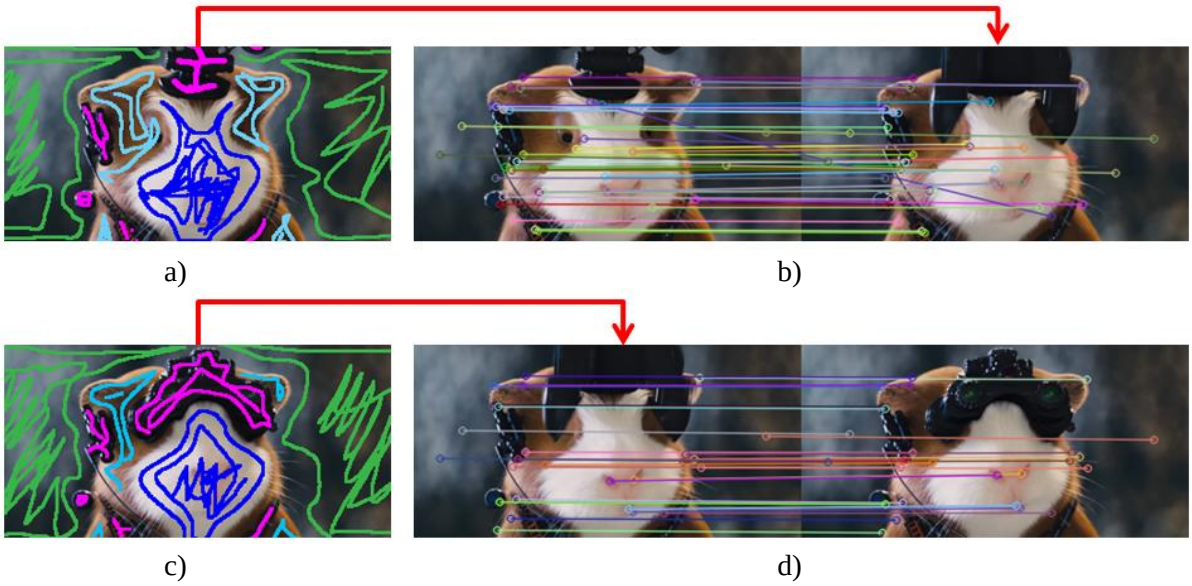


Abbildung 5.21: *Propagierung der Disparitäten von den Schlüssel-Frames auf ein dazwischenliegendes Frame. a) Scribbles des ersten Frames. b) Gefundene Matches von SIFT-Features zwischen Frame 1 und Frame 6. Die Features sind im Bild durch Kreise gekennzeichnet. Korrespondierende Features sind durch Linien miteinander verbunden. Die Disparitäten werden entlang der Matches von Frame 1 auf Frame 6 propagiert. c) Scribbles des letzten Frames. d) Matches zwischen Frame 6 und Frame 10. Die Disparitäten werden von Frame 10 auf Frame 6 propagiert.*

Die Propagierung der Scribble-Disparitäten erfolgt mittels *Feature Matching* von SIFT-Features (*Scale Invariant Feature Transform*) [60] der beteiligten Frames. Zur Extraktion und zum Matching von SIFT-Features werden Funktionen aus dem *Feature2D*-Framework [61] verwendet, welches in der freien Programmbibliothek OpenCV (*Open Source Computer Vision*) [62] enthalten ist. Interessierte LeserInnen sind für weiterführende Informationen über die Funktionsweise von SIFT auf [60] verwiesen. Für ein Video mit 10 Frames erfolgt die Propagierung der Disparitäten von *Frame 1* und *Frame 10* auf das dazwischenliegende *Frame 6* in drei Schritten:

1. Die SIFT- Features werden aus *Frame 1*, *Frame 6* und *Frame 10* extrahiert.

2. Die SIFT-Features zwischen *Frame 1* und *Frame 6* werden gematched (siehe Abbildung 5.21 b)). Dabei werden für alle Features aus *Frame 1* korrespondierende Features in *Frame 6* gesucht. Es werden ausschließlich robuste Matches verwendet, deren Distanz unterhalb eines Schwellwertes liegt (Distanz bezieht sich hier auf eine metrische Distanz, beispielsweise die Hamming-Distanz, und nicht auf die Distanz zwischen Pixelpositionen [61]). Es wird nun für jedes Feature in *Frame 6* das Pixel an der Position des korrespondierenden Features in *Frame 1* betrachtet. Enthält das Pixel eine Disparität, wird diese an die Pixelposition des korrespondierenden Features in *Frame 6* kopiert. Um möglichst viele Disparitäten zu erhalten, wird zusätzlich die Nachbarschaft des betreffenden Pixels betrachtet (beispielsweise in einem Radius von 10 Pixeln). Um zu verhindern, dass im Zuge dessen ungültige Disparitäten kopiert werden, wird die Differenz der Farbwerte zwischen den korrespondierenden Pixeln in *Frame 6* und *Frame 1* berechnet. Liegt diese unter einem Schwellwert (beispielsweise < 0.1) werden die Disparitäten kopiert.

3. Die Disparitäten zwischen *Frame 10* und *Frame 6* werden analog zum zweiten Schritt propagiert (siehe Abbildung 5.21 d)). Für Pixel in *Frame 6*, welchen im zweiten Schritt bereits Disparitäten zugewiesen wurden, wird der gewichtete Mittelwert zwischen den Disparitäten aus *Frame 1* und *Frame 10* verwendet. Die Disparitäten aus *Frame 1* fließen dabei zu 45% (Abstand zu *Frame 6* = 6), jene aus *Frame 10* zu 55% (Abstand zu *Frame 6* = 4) in den Mittelwert ein (siehe Abbildung 5.22 b)).

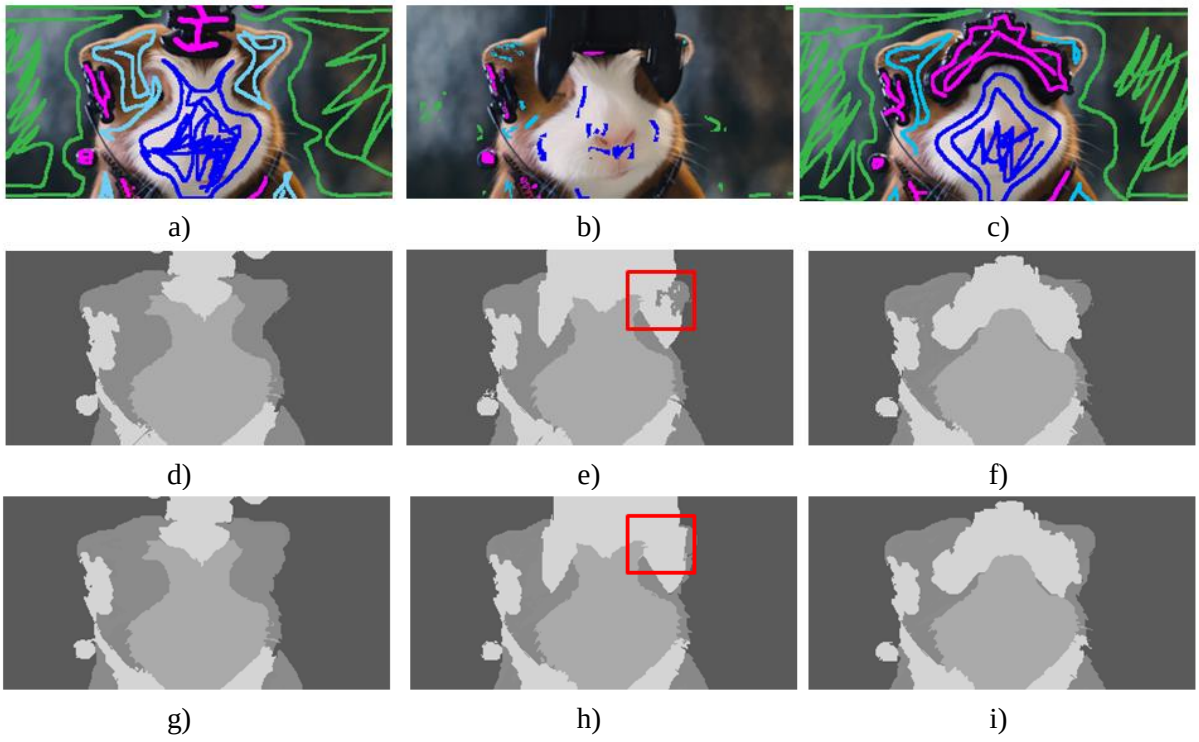


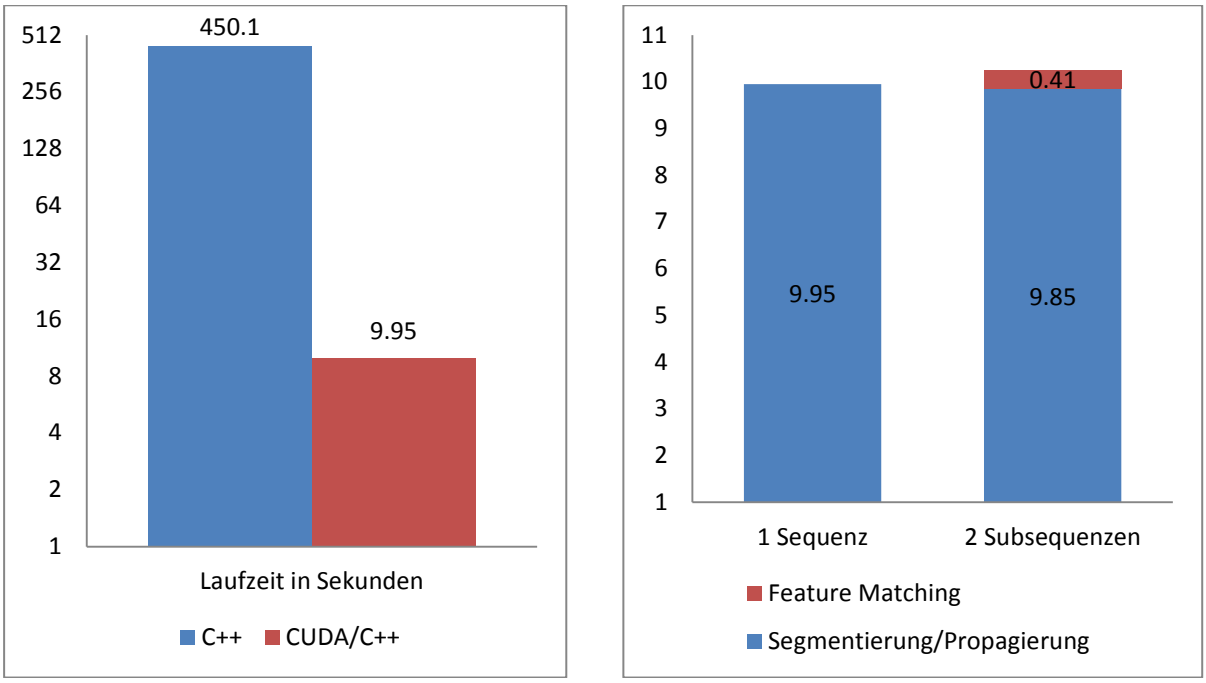
Abbildung 5.22: *a) Scribbles von Frame 1 des Videos aus Abbildung 5.21. b) Scribbles von Frame 6, welche aus Frame 1 und Frame 10 propagiert wurden. c) Scribbles von Frame 10. d)-e) Ergebnis der Segmentierung und Propagierung von Frame 1, Frame 6 und Frame 10 eines Videos von 10 Frames, welches in zwei Subsequenzen unterteilt wurde. g) –h) Ergebnis der Segmentierung und Propagierung desselben Videos als Ganzes. Durch die Aufteilung des Videos und die Propagierung der Disparitäten im dazwischenliegenden Frame kommt es in manchen Bildregionen zu Fehlern (im Bild rot markiert). Diese kommen zustande, da im Zuge des Feature Matching in diesem Bildbereich keine Übereinstimmungen gefunden werden konnten und demzufolge auch keine Disparitäten propagiert wurden.*

In diesem Lösungsansatz ist die Qualität der Ergebnisse der Propagierung von Disparitäten über mehrere Subsequenzen hinweg, durch die Qualität der Ergebnisse der Methoden zum Feature Matching beschränkt. Werden zu wenige oder falsche Matches gefunden, kommt es in den betroffenen Bildregionen zu Fehlern bei der Propagierung der Disparitäten. In Abbildung 5.22 wurden beispielsweise im rot markierten Bildbereich keine Matches gefunden und demzufolge auch keine Disparitäten propagiert. Aus diesem Grund kann es in diesen Bereichen zu Propagierungsfehlern kommen. Die Qualität der Ergebnisse kann durch Veränderung der Parameter des Feature Matching beeinflusst werden. Werden zu wenige Matches gefunden, kann der Schwellwert zur Auswahl von robusten Matches erhöht werden, wodurch mehr Matches erhalten werden. Dadurch wird auch die

Wahrscheinlichkeit von falschen Matches erhöht. Bei diesem Vorgehen zum Aufteilen längerer Videos handelt es sich um eine einfache Lösung. An dieser Stelle besteht somit Raum für weitere Entwicklungen.

5.5. Laufzeitvergleiche

In diesem Abschnitt wird die Laufzeit der CUDA/C++-Implementierung des Algorithmus zur segmentierungsbasierten Propagierung von Disparitäten in Videos, welche in dieser Diplomarbeit erstellt wurde (siehe Abschnitt 5.4), mit der ursprünglichen C++-Implementierung aus [22] verglichen. Beide Implementierungen wurden dazu auf einem PC mit einem Intel Xenon E5 Prozessor mit 3.6GHz, 32GB RAM und einer NVIDIA GeForce GTX 680 Grafikkarte ausgeführt.



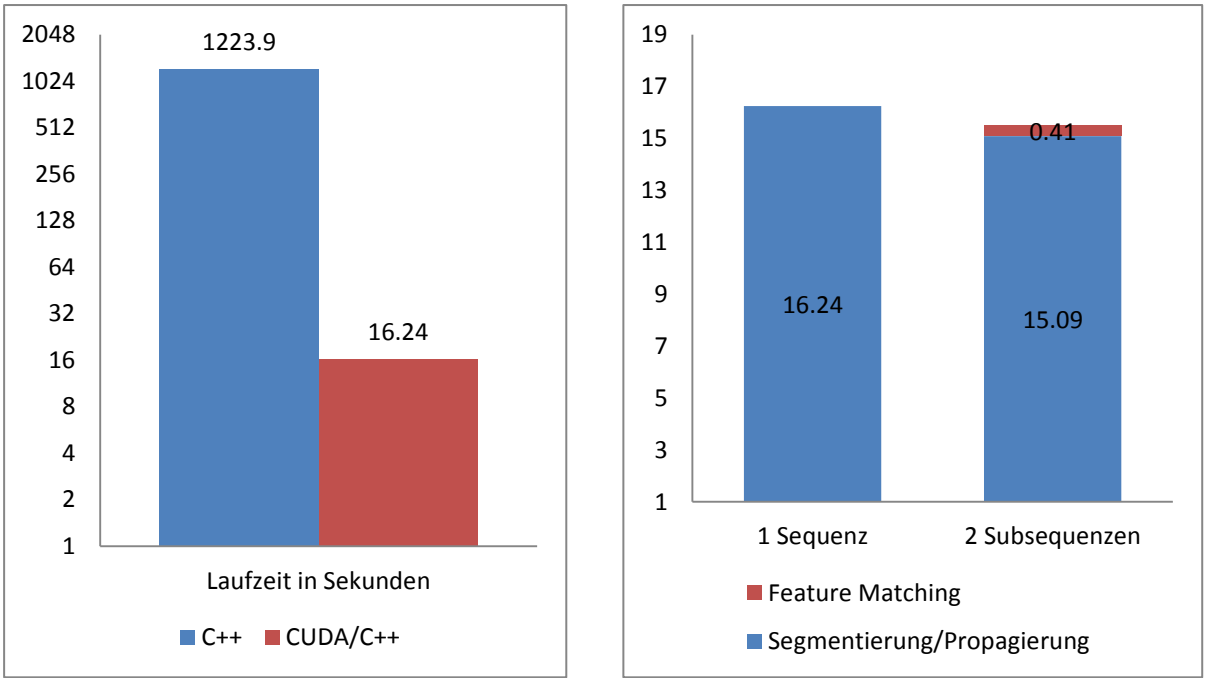
a) b)

Abbildung 5.23: *a) Laufzeitvergleich zwischen der CUDA/C++-Implementierung und der C++-Implementierung zur segmentierungsbasierten Propagierung von Disparitäten in Videos. Es wurde ein Video von 20 Frames mit der Auflösung 600 x 255 bearbeitet (k = 0.02, s = 1.1, c_{min} = 110, 1 Iteration). b) CUDA/C++-Laufzeit zur Bearbeitung eines Videos als Ganzes im Vergleich zur Aufteilung in zwei Subsequenzen.*

Für den ersten Laufzeitvergleich wurde ein RBG-Farbvideo von 20 Frames mit einer Auflösung von 600 x 255 in einer Iteration bearbeitet. Die Laufzeit der CUDA/C++-Version ist mit 9.95 Sekunden um den Faktor 45.24 schneller als die C++-Version, welche 450.1 Sekunden benötigt (siehe Abbildung 5.23 a)). Die Laufzeitverringerung ergibt sich hauptsächlich aus der Parallelisierung der Generierung des Regionen-Graphen (siehe Abschnitt 5.4, Schritt 3), da es sich dabei um den rechenintensivsten Teil des Segmentierungs-Algorithmus handelt. Bei der Bearbeitung desselben Videos in zwei Subsequenzen ergibt sich für die CUDA/++-Version mit 10.26 Sekunden eine höhere Laufzeit als bei der Bearbeitung des Videos als Ganzes (siehe Abbildung 5.23 b)). Die Laufzeiterhöhung ist auf den zusätzlichen Aufwand zur nicht-optimierten Propagierung von

Disparitäten mittels Feature Matching zurückzuführen (siehe Abschnitt 5.4). Die Laufzeit der Segmentierung/Propagierung verringert sich hingegen durch das Aufteilen des Videos um 0.1 Sekunden, da die Bearbeitung der beiden Subsequenzen auf der GPU insgesamt schneller durchgeführt werden kann als die Bearbeitung des Videos als Ganzes. Das ist darauf zurückzuführen, dass die Anzahl der Regionen in den Subsequenzen geringer ist als im Video als Ganzes. Es müssen somit im Zuge der Berechnung der Kantengewichte weniger Distanzvergleiche durchgeführt werden, was in einer geringeren Laufzeit resultiert.

Für den zweiten Laufzeitvergleich wurden beide Implementierungen auf dasselbe Video wie beim ersten Vergleich ausgeführt. Diesmal wurde das Video in fünf Iterationen bearbeitet. Die Laufzeit der CUDA/C++-Version ist mit 16.24 Sekunden im Vergleich zur Bearbeitung in einer Iteration um den Faktor 1.63 gestiegen (siehe Abbildung 5.24). Die Laufzeit der C++-Version hat sich mit 1223.9 Sekunden hingegen um den Faktor 2.72 erhöht. Der Laufzeitunterschied zwischen der CUDA/C++-Version und der C++-Version erhöht sich demzufolge mit der Anzahl der durchgeführten Iterationen. Der Vergleich zwischen der Laufzeit der Bearbeitung zweier Subsequenzen und der Bearbeitung eines einzigen Videos zeigt, dass sich die Laufzeit der Segmentierung und Propagierung durch das Aufteilen in Subsequenzen auf 15.09 Sekunden verringert. Da die Propagierung der Disparitäten mittels Feature Matching, unabhängig von der Anzahl der Iterationen, nur einmal ausgeführt werden muss, ergibt sich eine Gesamtlaufzeit von 15.5 Sekunden.



a) b)

Abbildung 5.24: a) *Laufzeitvergleich zwischen der C++-Version und der CUDA/C++-Version. Es wurde ein Video von 20 Frames mit der Auflösung 600 x 255 bearbeitet (k = 0.02, s = 1.1, c_{min} = 110, 5 Iterationen). b) CUDA/C++-Laufzeit zur Bearbeitung eines Videos als Ganzes im Vergleich zur Aufteilung in zwei Subsequenzen.*

6. Kohärente Regularisierung der Disparitätskarten

Nachdem die Segmentierung durchgeführt wurde, ist jedem Pixel im Video eine Disparität zugeordnet. Die Disparitätskarten können abrupte Tiefensprünge enthalten, da die auf Segmentierung beruhende Propagierung der Disparitäten (Kapitel 5) auf Zuweisung gegebener Disparitäten zu nicht annotierten Pixeln beruht. Enthält beispielsweise eine Region im ersten Frame eine hohen Disparität und im letzten Frame eine niedrige Disparität, kann es zu abrupten zeitlichen Disparitätssprüngen kommen (siehe Abbildung 6.1c) und d)). Die hohe Disparität des ersten Frames dominiert am Anfang des Videos, die niedrige Disparität des letzten Frames dominiert am Ende des Videos. Diese abrupten Übergänge können mittels räumlicher und zeitlicher Regularisierung der Disparitäten geglättet werden (siehe Abbildung 6.1e) und f)). Diese Regularisierung wird in [22] durch einen effizienten, kantenerhaltenden Glättungsfilter implementiert, welcher auf dem *Guided Filter* [23] basiert. Dieser Filter mittelt Disparitäten von farblich ähnlichen Pixeln einer Region.

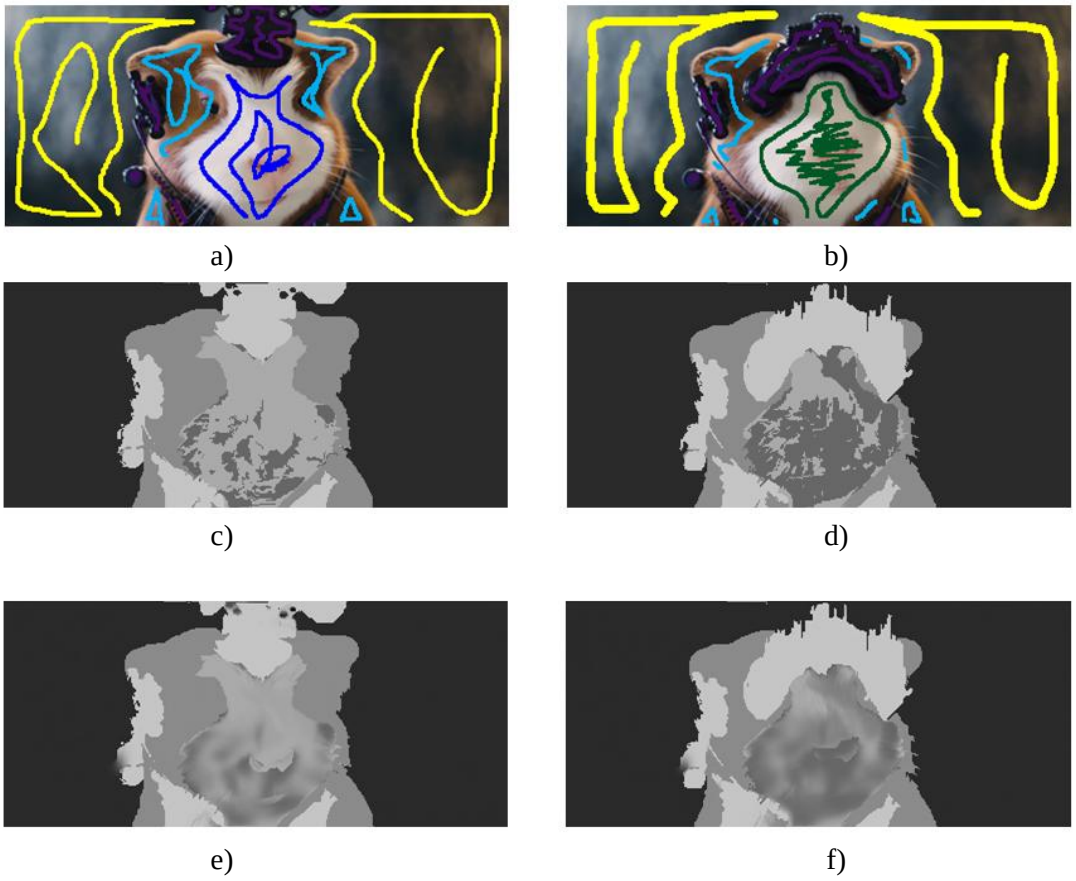


Abbildung 6.1*: a) Erstes Frame eines Videos von 10 Frames, in welchem die Disparitäten mittels Scribbles annotiert sind. b) Letztes Frame mit Scribbles. Im Bereich des Gesichts enthält das erste Frame eine hohe Disparität, das letzte Frame eine niedrige Disparität. Darum kommt es im dazwischenliegenden Frame in diesem Bereich zu abrupten Disparitätsprüngen. c) Disparitätskarte von Frame 2 nach der Segmentierung. d) Disparitätskarte von Frame 6. e) Disparitätskarte von Frame 2 nach regionsweiser Filterung. f) Disparitätskarte von Frame 6 nach regionsweiser Filterung. Die zeitlichen Disparitätssprünge im Bereich des Gesichts wurden geglättet.*

In den unbearbeiteten Tiefenkarten werden feine Details (beispielsweise Haare) nicht immer von der Segmentierung erfasst. In einem optionalen Nachbearbeitungsschritt kann der Guided Filter zur Verbesserung feiner Details an Regionsgrenzen herangezogen werden. Der für Videos erweiterte Guided Filter wird dazu auf die gesamte Disparitätskarte angewandt. Die Disparitäten werden weiter geglättet, wobei die Kanten, welche durch zeitliche und räumliche Änderungen zwischen den Frames zustande gekommen sind, erhalten bleiben.

Da beide Filterschritte, die räumliche und zeitliche Regularisierung und die optionale Nachbearbeitung, auf dem Guided Filter basieren, diskutiert dieses Kapitel zu Beginn die Grundlagen dieses Filters (Abschnitt 6.1). Anschließend werden eine serielle CPU-Implementierung (Abschnitt 6.2) und eine parallele GPU-Implementierung des Guided Filters (Abschnitt 6.3), welche in dieser Diplomarbeit erstellt wurde, vorgestellt. Im Anschluss an die jeweilige Implementierung wird eine Erweiterung zur Filterung von Videos beschrieben. Neben der Erweiterung des Guided Filters zur Anwendung auf Videos stellt dieses Kapitel zwei GPU-Implementierungen zur regionsweisen Regularisierung vor (Abschnitt 6.4). In Abschnitt 6.5 werden die unterschiedlichen Implementierungen des Guided Filters anhand der benötigten Laufzeit verglichen.

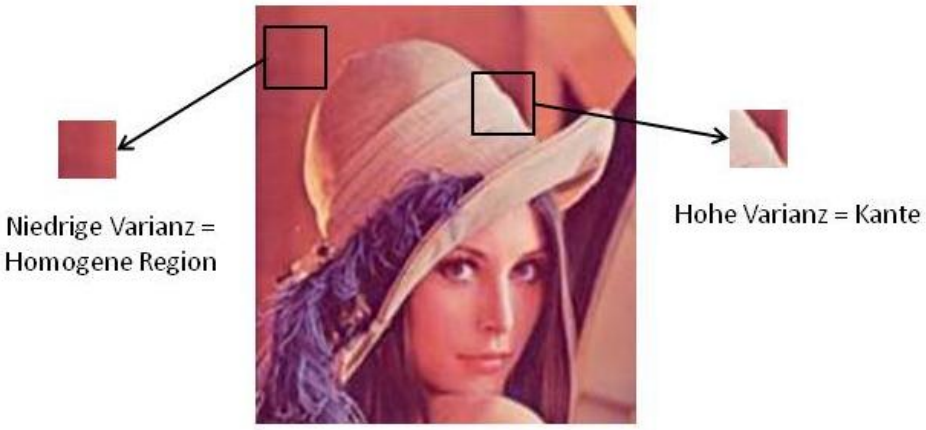


Abbildung 6.2: *Homogene Regionen sind durch eine niedrige Varianz gekennzeichnet, das heißt, die Intensitätswerte innerhalb der Region sind ähnlich. Regionen, die eine Bildkante enthalten, zeichnen sich durch eine hohe Varianz aus. Nach* [26]

6.1. Grundlagen des Guided Filters

Beim Guided Filter [23] handelt es sich um einen kantenerhaltenden Glättungsfilter, welcher zur Glättung von Bildern entwickelt wurde. Der Guided Filter glättet das Eingabebild und erhält gleichzeitig die Bildkanten. Der Filter verwendet dazu einerseits das zu filternde Eingabebild und andererseits ein Leitbild (*Guidance Image*). Beim Leitbild kann es sich um das Eingabebild selbst handeln. Dieses Konzept ähnelt dem des Joint Bilateral Filters [29, 41]. Das Eingabebild wird aufgrund des Leitbildes verändert. Regionen im Eingabebild, denen homogene Regionen im Leitbild entsprechen, werden geglättet. Regionen im Eingabebild, denen Regionen mit Kanten im Leitbild entsprechen, bleiben erhalten. Die Entscheidung, ob eine Region homogen ist oder eine Kante enthält, wird aufgrund der Varianz der darin liegenden Pixel im Leitbild bestimmt. Diese gibt an, wie sehr die Intensitätswerte der Pixel innerhalb einer Region gestreut sind [63]. Sind die Intensitätswerte innerhalb einer Region sehr ähnlich, ist die Varianz niedrig und es wird angenommen, dass es sich um eine homogene Region handelt (siehe Abbildung 6.2, links). Unterscheiden sich die Intensitätswerte

innerhalb einer Region stark, ist die Varianz hoch und es wird angenommen, dass es sich um eine Kante handelt (siehe Abbildung 6.2, rechts).

Der Guided Filter basiert auf zwei Annahmen: 1. Der Guided Filter ist ein lokales lineares Modell zwischen dem Leitbild und dem Ausgabebild. 2. Die Differenz zwischen dem Eingabebild und dem Ausgabebild ist minimal [23]:

1. Das gefilterte Ergebnisbild ist eine lineare Transformation des Leitbildes in einem Filterfenster w_k, welches auf das Pixel k zentriert ist [23]:

$$q_i = a_k I_i + b_k \ , \forall i \in w_k \tag{6.1}$$

   I entspricht dem Leitbild und q dem gefilterten Ausgabebild. I_i entspricht der Position des Pixels i im Leitbild. a_k und b_k sind lineare Koeffizienten, welche aus dem Leitbild ermittelt werden.

2. Es wird angenommen, dass die Differenz zwischen dem Eingabebild p und Ausgabebild q minimal ist [23]:

$$(q - p)^2 \rightarrow minimal \tag{6.2}$$

Die beiden Annahmen können gemeinsam in folgender Energiefunktion pro Filterfenster w_k minimiert werden [23]:

$$E(a_k, b_k) = \sum_{i \in w_k} ((a_k I_i + b_k - p_i)^2 + \varepsilon a_k^2) \tag{6.3}$$

Der Parameter ε ist ein Regulierungsparameter, welcher verhindert, dass a_k zu groß wird. Die Minimierung von Formel (6.3) erfolgt lokal mittels linearer Regression [63, 64], woraus sich die Koeffizienten a_k und b_k ergeben [23]:

$$a_k = \frac{\frac{1}{|w|}\sum_{i \in w_k} I_i p_i - \mu_k \bar{p}_k}{\sigma_k^2 + \epsilon} \tag{6.4}$$

$$b_k = \bar{p}_k - a_k \mu_k \tag{6.5}$$

w_k ist ein auf k zentriertes Filterfenster. $|w|$ gibt die Anzahl der Pixel an, welche innerhalb dieses Fensters liegen. $\bar{p}_k$ ist der Mittelwert aller Pixel des Eingabebildes p im Filterfenster w_k. μ_k ist der Mittelwert und σ_k^2 ist die Varianz aller Pixel des Leitbildes I im Filterfenster w_k. Das lineare Modell wird auf alle lokalen Filterfenster w_k im Eingabebild angewandt. Da ein Pixel i in mehr als einem Filterfenster w_k enthalten ist (siehe Abbildung 6.3a)) und für jedes Filterfenster unterschiedliche Koeffizienten a_k und b_k berechnet werden, ergeben sich für jedes Pixel i mehrere unterschiedliche Ausgabewerte q_i. Zur Berechnung des finalen Ausgabewertes wird der Mittelwert aller möglichen Ausgabewerte q_i verwendet [23]:

$$q_i = \frac{1}{|w|} \sum_{k:i \in w_k} (a_k I_i + b_k) \tag{6.6}$$

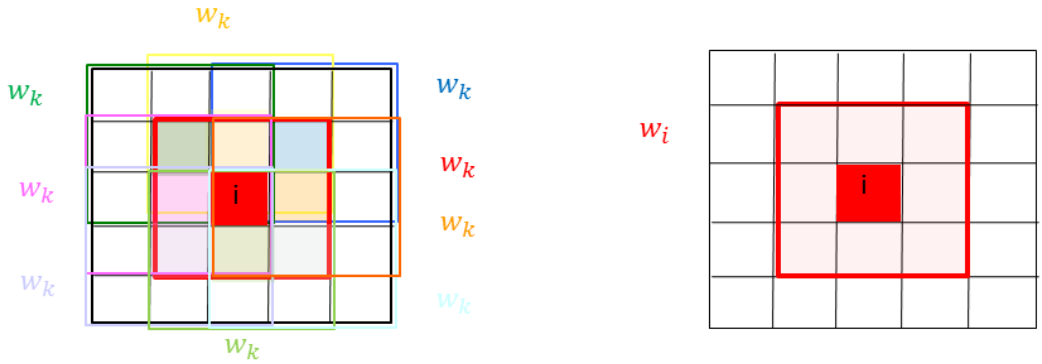
Die Berechnung der Mittelwerte der Ausgabepixel in jedem Filterfenster ist äquivalent zur Mittelwertbildung der Koeffizienten *a* und *b* im Filterfenster w_i (siehe Abbildung 6.3b)) [23]:

$$\bar{a}_i = \frac{1}{|w|} \sum_{k \in w_i} a_k \qquad \bar{b}_i = \frac{1}{|w|} \sum_{k \in w_i} b_k \tag{6.7}$$

$\bar{a}_i$ ist der Mittelwert aller Koeffizienten a_k, $\bar{b}_i$ der Mittelwert aller Koeffizienten b_k, welche in die zu filternde Pixelposition *i* einfließen. w_i ist ein auf das Pixel *i* zentriertes Filterfenster. Die Berechnung des gefilterten Ausgabebildes q_i an der Stelle *i* wird anschließend wie folgt durchgeführt [23]:

$$q_i = \bar{a}_i I_i + \bar{b}_i \tag{6.8}$$

Das Ergebnis des Koeffizienten a_k hängt von der Varianz σ_k^2 der Bildregion ab (siehe Abbildung 6.2). Bei niedriger Varianz (homogene Bildregion) ist der Wert von a_k gering, bei hoher Varianz (Bildkante) ist der Wert hoch. Der Koeffizient b_k berechnet sich aus dem Mittelwert aller Pixel innerhalb des Filterfensters und wird durch das Ergebnis von a_k verringert. Je größer der Wert von a_k ist, desto geringer ist der Wert von b_k. Der Regulierungsparameter ε definiert den Schwellwert, ab welchem die Varianz als hoch oder gering klassifiziert wird. Ist die Varianz größer als ε, liegt a_k nahe bei 1 und b_k nahe bei 0. Der ursprüngliche Wert des Pixels fließt somit zu einem größeren Teil in das Ergebnis ein als der Mittelwert, was dazu führt, dass Bildkanten erhalten bleiben. Ist die Varianz kleiner als ε, liegt a_k nahe bei 0 und b_k nahe bei 1. Der Mittelwert hat somit einen größeren Einfluss auf das Ergebnis als der ursprüngliche Pixelwert, woraus folgt, dass homogene Regionen geglättet werden (siehe Abbildung 6.4). Der Vorteil des Guided Filters gegenüber verwandten Filtern (beispielsweise Joint Bilateral Filter [29, 41]) liegt in der schnelleren Laufzeit des Guided Filters, bei gleichbleibender Qualität der Ergebnisse (siehe [23]). Die Laufzeit des Guided Filters für ein Eingabebild mit *N* Pixeln beträgt *O(N)*. Die Laufzeit ist konstant bezüglich der Größe des Filterkerns.



a) b)

Abbildung 6.3*: a) Das Pixel i und das Filterfenster, in dessen Mittelpunkt i liegt, sind rot markiert. Das Pixel i ist in mehreren überlappenden Filterfenstern w_k enthalten. Der Ausgabewert an der Stelle i variiert mit den verschiedenen Koeffizienten der unterschiedlichen Filterfenster. Der finale Ausgabewert wird durch die Bildung des Mittelwertes aller möglichen Ausgabewerte q_i berechnet. b) Anstatt des Mittelwertes der Ausgabewerte in allen Filterfenstern w_k, kann der Mittelwert der Koeffizienten a und b im Filterfenster w_i verwendet werden.*

Neben der Erhaltung der Kanten bei der Glättung eines Bildes und der effizienten Implementierbarkeit hat der Guided Filter noch eine weitere positive Eigenschaft. Aufgrund des lokalen linearen Modells zur Berechnung des Ausgabebildes (Formel (6.8)) ist es möglich, die Struktur des Leitbildes auf das Eingabebild zu übertragen (siehe Abbildung 6.5). Diese Eigenschaft des Guided Filters ermöglicht beispielsweise Anwendungen wie *Feathering/Matting* [65], *Dehazing* [66] oder filterbasierte *Stereo Matching* Methoden [40, 67].

a) b)

Abbildung 6.4: *a) Eingabebild. b) Gefiltertes Ausgabebild unter Verwendung eines 8 x 8 Guided Filters mit ε = 0.004. Als Eingabebild und Leitbild wurde eine 8-Bit Graustufenbild mit der Auflösung 1024 x 1024 verwendet. Das Bild wurde in den homogenen Bildbereichen geglättet, Bildkanten blieben erhalten.*

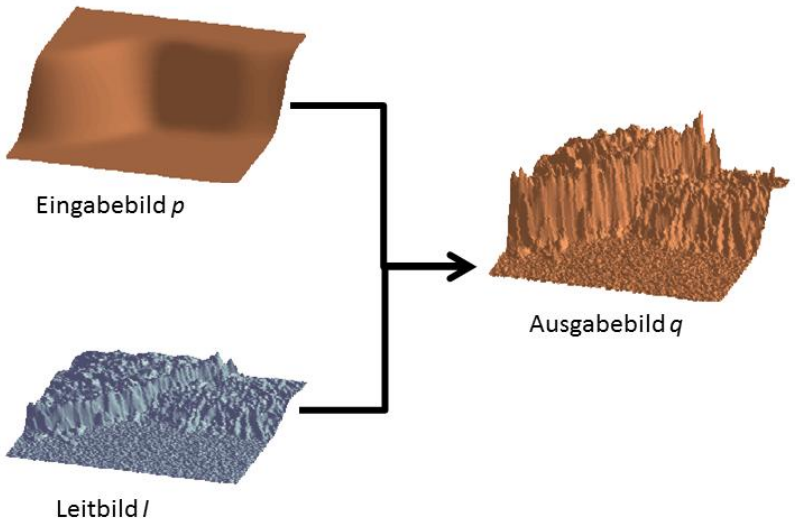


Abbildung 6.5: *Der Guided Filter ermöglicht es, Strukturen aus dem Leitbild I auf das Ausgabebild q zu übertragen, selbst wenn im Eingabebild p keine Strukturen enthalten sind. Nach* [23]

Erweiterung zur Filterung von Farbbildern

Bei der Verwendung von farbigen Eingabebildern *p* kann der Guided Filter auf jeden Farbkanal getrennt angewandt werden. Jede Komponente des Farbraumes wird dabei als eigenständiges Bild betrachtet. Nach der Filterung werden die Kanäle wieder zu einem einzigen Ergebnisbild *q* zusammengefügt. Für den Fall, dass das Leitbild *I* mehrere Farbkanäle hat, werden die Formeln (6.4), (6.5) und (6.8) wie folgt adaptiert [23]:

$$a_k = (\Sigma_k + \varepsilon U)^{-1} * (\frac{1}{|w|}\sum_{i \in w_k} I_i p_i - \mu_k \bar{p}_k) \tag{6.9}$$

$$b_k = \bar{p}_k - a_k^T \mu_k \tag{6.10}$$

$$q_i = \bar{a}_i^T I_i + \bar{b}_i \tag{6.11}$$

Σ_k ist die Kovarianzmatrix des Leitbildes *I* im Filterfenster w_k. *U* ist die Identitätsmatrix. w_k ist ein auf *k* zentriertes Filterfenster, $|w|$ gibt die Anzahl der Pixel an, welche innerhalb dieses Fensters liegen. $\bar{p}_k$ ist der Mittelwert aller Pixel des Eingabebildes *p* im Filterfenster w_k. μ_k ist der Mittelwert aller Pixel des Leitbildes im Filterfenster. In Farbbildern sind mehr Kanteninformationen enthalten als in Grauwertbildern, weshalb sie sich besser zur Verwendung als Leitbilder eignen.

6.2. Sequentielle Implementierung

Dieser Abschnitt diskutiert eine sequentielle CPU-Implementierung des Guided Filters, welcher eine konstante Laufzeit bezüglich der Größe des Filterfensters aufweist. Der grundlegende Ablauf wird anhand eines Guided Filters zur Filterung von Grauwertbildern beschrieben. Listing 6.1 zeigt den Pseudocode dieses Filters.

Eingabewerte: Leitbild *I*, Eingabebild *p*, Radius *r* des Filterfensters, Regulierungsparameter ε
Ausgabewert: Gefiltertes Bild *q*

```
1    mean_I =  boxfilter(I)
2    mean_p =  boxfilter(p)

3    corr_I =  boxfilter(I. * I)
4    corr_Ip = boxfilter(I. * p)
5    var_I = corr_I   - mean_I .* mean_I
6    cov_Ip = corr_Ip - mean_I .* mean_p

7    a = cov_Ip ./ (var_I + ε)
8    b = mean_p - a .* mean_I

9    mean_a = boxfilter(a)
10   mean_b = boxfilter(b)
11   q = mean_a .* I + mean_b
```

Listing 6.1: *Pseudocode eines Guided Filters für Grauwertbilder. Nach* [23]

Die sequentielle Implementierung eines Guided Filters für Bilder (Listing 6.1) verläuft in vier Schritten:

1. Im ersten Schritt werden die lokalen Mittelwerte μ_k (Formel (6.4) und (6.5)) für jedes Filterfenster w_k des Leitbildes I und die lokalen Mittelwerte $\bar{p}_k$ jedes Filterfensters w_k des Eingabebildes p berechnet. Die Mittelwerte können jeweils effizient mit dem Box-Filter (siehe Kapitel 4) berechnet werden (Listing 6.1, Zeile 1 und 2).

2. Im zweiten Schritt werden die lokalen Varianzen des Leibildes (Formel (6.4)) und die Kovarianzen von Leitbild und Eingabebild (Formel (6.4)) berechnet. Die Berechnung der Varianzen erfolgt in Listing 6.1 in Zeile 3 und Zeile 5, die Berechnung der Kovarianzen wird in Zeile 4 und Zeile 6 durchgeführt.

3. Im dritten Schritt werden die Koeffizienten a_k und b_k berechnet (Formel (6.4) und (6.5)). Die jeweilige Berechnung erfolgt in Listing 6.1 in Zeile 7 und Zeile 8. Es ist zu beachten, dass die Division in Listing 6.1 in Zeile 7 pixelweise durchgeführt wird und die Berechnung von a_k den Regulierungsparameter ε enthält.

4. Wie in Abschnitt 6.1 diskutiert, werden die Koeffizienten a_k und b_k für die Berechnung des gefilterten Ergebnisbildes gemittelt. In der sequentiellen Implementierung werden diese Mittelwerte effizient mittels Box-Filter berechnet (Listing 6.1, Zeile 9 und 10). Die anschließende Berechnung des Ergebnisbildes erfolgt in Zeile 11 durch direktes Einsetzen der Koeffizienten in Formel (6.8).

Die pixelweise Multiplikation (Listing 6.1, Zeile 3-6, 8 und 11), die pixelweise Division (Listing 6.1, Zeile 7), Subtraktion (Listing 6.1, Zeile 5, 6 und 8) und Addition (Listing 6.1, Zeile 11) sind von der Größe des Filterkerns unabhängig und beeinträchtigen die konstante Laufzeit des Algorithmus nicht. Der rechenintensivste Teil des Guided Filter-Algorithmus ist die Mittelwertbildung mittels Filterung mit dem Box-Filter (Listing 6.1, Zeilen 1-4, 9 und 10). Die Laufzeit des Guided Filters hängt somit maßgeblich von der Laufzeit des verwendeten Box-Filters ab. Daraus folgt, dass eine Optimierung des Box-Filters zu einer Verringerung der Laufzeit des Guided Filters führt. Die Laufzeit der optimierten Box-Filter-Implementierung, welche in Kapitel 4 vorgestellt wurde, ist konstant bezüglich der Größe des Filterfensters. Wird diese Version des Box-Filters zur Mittelwertberechnung beim Guided Filter verwendet, folgt daraus eine konstante Laufzeit des Guided Filters *(O(N))*. Diese Angabe bezieht sich auch auf die Filterung von Farbbildern. [23]

Erweiterung zur Filterung von Videos

Der vorgestellte Guided Filter für Bilder kann für die Verwendung zur Filterung von Videos erweitert werden. Dazu wird der Algorithmus auf ein Video angewandt. Statt eines zweidimensionalen Box-Filters für Bilder wird ein dreidimensionaler Box-Filter für Videos, wie in Kapitel 4 beschrieben, zur Mittelwertbildung verwendet. Die restlichen Bearbeitungsschritte (Listing 6.1, Zeile 5-8 und 11) können unverändert durchgeführt werden.

6.3. Parallele Implementierung

Die Laufzeit des Guided Filters kann durch eine parallele GPU-Implementierung weiter verringert werden. Dazu werden die einzelnen Bearbeitungsschritte aus Listing 6.1 auf die GPU verlagert und dort parallel ausgeführt. Wie in Abschnitt 6.2 erwähnt, stellt die Filterung mittels Box-Filter den aufwändigsten Bearbeitungsschritt des Guided Filters dar. In allen anderen Bearbeitungsschritten werden pixelweise Rechenoperationen durchgeführt. Diese sind unabhängig voneinander und können demzufolge parallel in jeweils einem eigenen Thread auf der GPU ausgeführt werden. In Kapitel 4 wurde eine effiziente GPU-Implementierung des Box-Filters vorgestellt. Diese findet nun in der parallelen GPU-Implementierung des Guided Filters Anwendung. Die Optimierung des Box-Filters wirkt sich direkt auf die Laufzeit des Guided Filters aus. Die Verarbeitungsschritte aus Listing 6.1 werden schrittweise ausgeführt, wobei jeweils eine Zeile des Pseudocodes parallel auf der GPU berechnet wird.

Erweiterung zur Filterung von Videos

In dieser Diplomarbeit wurde die parallele GPU-Implementierung des Guided Filters aus [26] adaptiert, um für die Filterung von Videos verwendet werden zu können. Zur Mittelwertberechnung wurde die parallele GPU-Implementierung des dreidimensionalen Box-Filters aus Kapitel 4 verwendet. Alle weiteren Bearbeitungsschritte können analog zur Filterung von Bildern ausgeführt werden.

Die Leistungsfähigkeit der GPU-Implementierung ist durch die Kapazität des Global-Memorys der GPU beschränkt. So können nur eine bestimmte Anzahl von Frames, abhängig von ihrer Auflösung und der Kapazität des Global-Memory, gleichzeitig auf der GPU gefiltert werden. Sollen längere Videos gefiltert werden, können diese in Subsequenzen aufgeteilt werden. Diese Subsequenzen werden nacheinander an die GPU übergeben und dort jeweils parallel bearbeitet. Der Guided Filter für Videos filtert sowohl in räumlicher als auch in zeitlicher Dimension. Damit es durch die Aufteilung der Filterung nicht zu Fehlern in der zeitlichen Filterung kommt, wird ein entsprechend großer Überlappungsbereich zwischen den Subsequenzen eingeplant (siehe Abbildung 6.6). Die Anzahl der überlappenden Frames entspricht dem Radius des zeitlichen Filterfensters. Bei einem zeitlichen Filterfenster mit dem Radius $r_t = 4$ gibt es beispielsweise eine Überlappung von vier Frames zwischen den Subsequenzen. Das Filterergebnis wird durch diese Aufteilung nicht verändert.

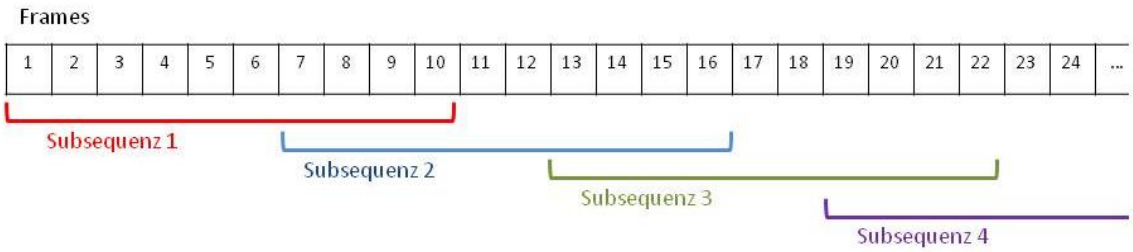
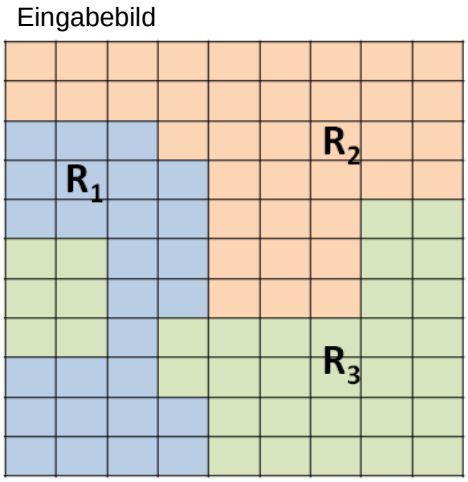


Abbildung 6.6: *Videos, die zu groß sind, um als Ganzes auf der GPU bearbeitet werden zu können, werden in Subsequenzen aufgeteilt. Um Fehler in der zeitlichen Filterung zu vermeiden, gibt es zwischen den Subsequenzen einen Überlappungsbereich in der Größe des Radius des zeitlichen Filterfensters r_t. In diesem Fall hat das Filterfenster den Radius $r_t = 4$. Die Überlappung beträgt somit vier Frames.*

6.4. Regionsweiser Guided Filter

Um die Veränderung der Disparitäten, die sich aus der in Kapitel 5 beschriebenen Propagierung ergeben, über mehrere Frames hinweg interpolieren zu können, wird der Guided Filter auf jede räumliche und zeitliche Region angewandt. Die Regionen werden unabhängig voneinander gefiltert. Der Guided Filter verwendet als Eingabevideo die gesamte Disparitätskarte des Eingabevideos. Als Leitvideo dient das originale Farbvideo. Eine zusätzliche Regionsmaske gibt an, zu welchen Regionen die einzelnen Pixel gehören. Dank der kantenerhaltenden Eigenschaften des Guided Filters werden Regionen in der Disparitätskarte geglättet, wenn die entsprechenden Pixel in dem originalen Video ähnliche Farbwerte aufweisen. Disparitäten an Bildkanten im Farbvideo bleiben erhalten. Durch die Anwendung eines regionsweisen Guided Filters können Disparitäten zeitlich über mehrere Frames hinweg geglättet werden, ohne Werte außerhalb der jeweiligen Regionen zu beeinflussen.



Regionsmaske

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	**1**	**1**	0	0	0	0	0	0
1	**1**	**1**	**1**	0	0	0	0	0
1	**1**	**1**	**1**	0	0	0	0	0
0	**1**	**1**	**1**	0	0	0	0	0
0	**1**	**1**	**1**	0	0	0	0	0
0	**1**	**1**	0	0	0	0	0	0
1	**1**	**1**	0	0	0	0	0	0
1	**1**	**1**	**1**	0	0	0	0	0
1	**1**	**1**	**1**	0	0	0	0	0

a) b)

Abbildung 6.7: *a) Frame des Eingabevideos. Die Regionen im Eingabebild sind farblich gekennzeichnet (Region R_1: blau, Region R_2: orange, Region R_3: grün). b) Regionsmaske für R_1 (blau). In der Regionsmaske sind die Pixel der aktuellen Region mit 1 gekennzeichnet. Bei der Filterung werden nur Pixel der aktuellen Region berücksichtigt.*

Farbwerte

4	4	6	2
4	4	5	3
1	1	1	3
1	2	2	1
1	2	2	1

.

Maske

0	0	0	0
0	0	0	0
1	1	0	0
1	1	1	1
1	1	1	1

=

Ergebnis

0	0	0	0
0	0	0	0
1	1	0	0
1	2	2	1
1	2	2	1

Abbildung 6.8: *Das Überspringen von Bildbereichen, welche sich nicht in der aktuellen Region befinden, erfolgt durch Multiplikation der Farbwerte mit der Regionsmaske. Pixel außerhalb der aktuellen Region werden mit 0 multipliziert und fließen somit nicht in die Mittelwertberechnung ein.*

In einer naiven Implementierung des regionsweisen Guided Filters wird jede Region der Disparitätskarte einzeln in einem eigenen Filterschritt gefiltert. Der Guided Filter verwendet die vollständige Disparitätskarte als Eingabevideo. Die Regionen werden im Zuge der Segmentierung

erzeugt (siehe Abbildung 6.7a), Regionen sind farblich gekennzeichnet) und pro Region wird eine Regionsmaske erstellt (siehe Abbildung 6.7b)). Der Guided Filter filtert nur jene Pixel, welche in der Regionsmaske mit 1 gekennzeichnet sind. Nachdem alle Regionen gefiltert wurden, werden die Ergebnisse wieder zu einem einzigen Video zusammengefasst.

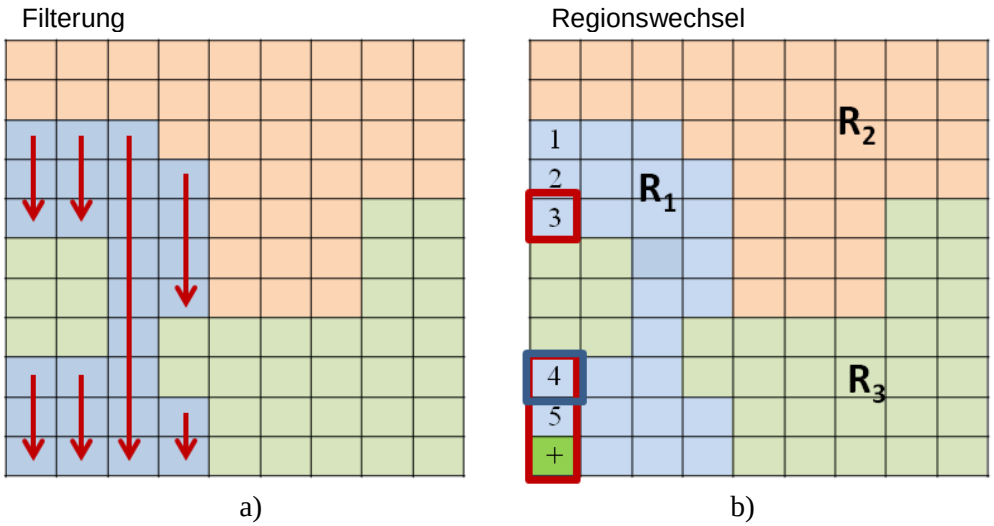
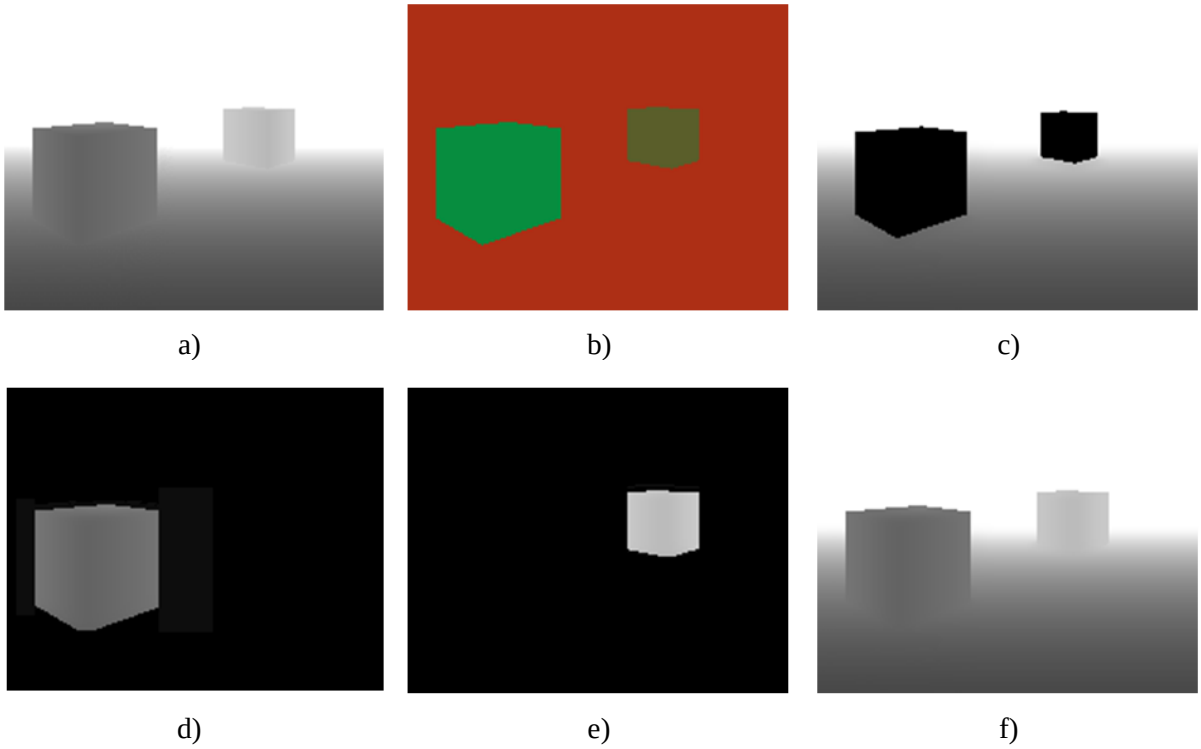


Abbildung 6.9: *a) Spaltenfilterung der ersten Region mittels des regionsweisen Box-Filters. Es werden nur Pixel berücksichtigt, die sich in der aktuellen Region R_1 (blau) befinden. b) Regionswechsel zwischen R_1 und R_2. Es fließen nur Pixel in die Mittelwertberechnung ein, die sich in R_1 befinden (Pixel 1 bis 6).*

In der naiven Implementierung des regionsweisen Guided Filters ist der Ablauf des regionsweisen Box-Filters zur Filterung der Bildspalten für eine Region R_1 wie folgt (siehe Abbildung 6.9): Vor der Anwendung des Box-Filters wird das gesamte Eingabevideo mit der Regionsmaske multipliziert. Dies hat eine Maskierung der Pixel außerhalb der aktuellen Region zur Folge (siehe Abbildung 6.8). Durch die Maskierung wird bei Anwendung der Sliding Window-Technik für Pixel außerhalb der aktuellen Region durch die Multiplikation mit 0 auch jeweils 0 zur Laufsumme addiert. Im Falle eines Regionswechsels fließen aus diesem Grund nur Pixel innerhalb des Filterfensters in die Mittelwertberechnung ein, welche sich in der Region R_1 befinden. Die Anzahl der in die Mittelung einfließenden Pixel ist kleiner als die Größe des Filterfensters. Aus diesem Grund wird die Laufsumme zur Mittelwertberechnung anschließend nicht durch die Größe des Filterfensters dividiert, sondern durch die Anzahl der Pixel im Filterfenster, welche zur aktuellen Region gehören.

Die Filterung der Bildzeilen und die zeitliche Filterung erfolgen analog dazu. So entsteht für jede Region ein eigenes gefiltertes Ergebnisvideo. Die regionsweisen Ergebnisse werden anschließend zu einem einzigen Ergebnis zusammengefasst. Die restlichen Bearbeitungsschritte des Guided Filters (siehe Listing 6.1, Zeile 5-8, 11) werden analog dazu für jede Region getrennt durchgeführt. Der Nachteil dieses naiven Lösungsansatzes ist, dass für jede Region getrennt gefiltert wird. Wird beispielsweise ein Video von 10 Frames mit jeweils einer Auflösung von 1024 x 1024 und 20 Regionen verarbeitet, wird der Guided Filter 20 Mal auf das gesamte Video angewandt. Bei einer Laufzeit von 0.015 Sekunden pro Frame (siehe Abbildung 6.16) ergibt das eine Gesamtlaufzeit von 3 Sekunden. Die Laufzeit steigt somit mit der Anzahl der Regionen.

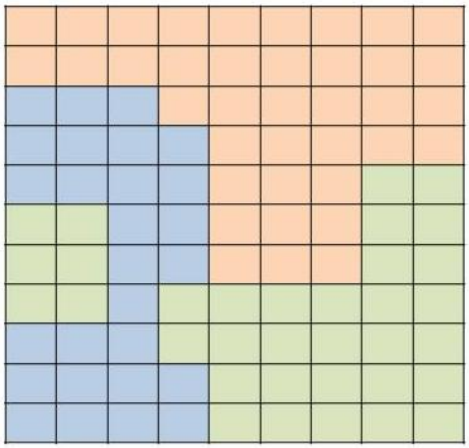


a) b) c)

d) e) f)

Abbildung 6.10: *Regionsweise Filterung eines Bildes a) mit drei Regionen. Jede Region wird in einem separaten Bearbeitungsschritt gefiltert. Bildbereiche, welche sich nicht in der aktuellen Region befinden, werden durch Multiplikation mit der jeweiligen Regionsmaske enfernt (in den Abbildungen sind diese Bildbereiche schwarz) und fließen somit nicht in die Mittelwertberechnung ein. Die drei gefilterten Teilergebnisse werden anschließend zu einem Ergebnisbild kombiniert. a) Eingabebild (Disparitätskarte). b) Farblich gekennzeichnete Regionen des Eingabebildes. c) Filterung der ersten Region (Hintergrund). d) Filterung der zweiten Region (großer Würfel). e) Filterung der dritten Region (kleiner Würfel). f) Kombination der Zwischenergebnisse aus c) – e).*

Eingabebild



Regionsmaske

2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2
1	1	1	2	2	2	2	2	2
1	1	1	1	2	2	2	2	2
1	1	1	1	2	2	2	3	3
3	1	1	1	2	2	2	3	3
3	1	1	1	2	2	2	3	3
3	1	1	3	3	3	3	3	3
1	1	1	3	3	3	3	3	3
1	1	1	1	3	3	3	3	3
1	1	1	1	3	3	3	3	3

a) b)

Abbildung 6.11: *a) Frame des Eingabevideos. b) Regionsmaske des Eingabeframes. Sie speichert für jedes Pixel des Eingabevideos eine ID, die angibt, welcher Region es zugeordnet ist.*

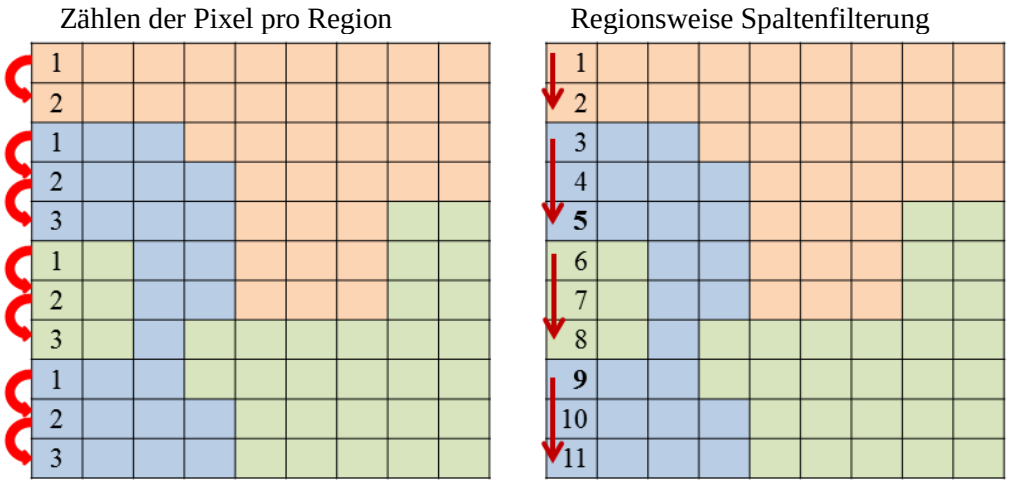


Abbildung 6.12: *a) In jedem Schritt wird zuerst die Anzahl der Pixel in jeder Teilregion ermittelt. b) Anschließend werden die Pixel jeder Teilregion schrittweise gefiltert (Pixel 1 und 2 im ersten Schritt, 3 bis 5 im zweiten Schritt, 6 bis 8 im dritten Schritt und 9 bis 11 im vierten Schritt). Jede Teilregion wird als eigenständiges Bild behandelt und die Ränder der Teilregionen werden als Bildränder betrachtet. Die Randbehandlung erfolgt wie in Kapitel 4 beschrieben. Die Teilregionen werden unabhängig voneinander gefiltert.*

Optimierung des regionsweisen Guided Filters

Im Zuge dieser Diplomarbeit wurde eine optimierte Version des regionsweisen Guided Filters entwickelt. Die Laufzeit wird verkürzt, indem alle Regionen in einem einzigen Durchlauf des Guided Filters gefiltert werden. Wie bei der naiven Implementierung basiert auch die optimierte Implementierung auf einem Eingabevideo, einem Leitvideo und einer Regionsmaske (siehe Abbildung 6.11). Im Gegensatz zu der Regionsmaske aus Abbildung 6.7, welche ausschließlich dazu diente, die aktuelle Region zu kennzeichnen, werden hier alle Pixel des Eingabevideos einer Region zugeordnet. Die Regionsmaske speichert für jedes Pixel eine ID, welche es eindeutig einer Region zuordnet. Die optimierte Implementierung des regionsweisen Guided Filters hat das Ziel, die Laufzeit der naiven Implementierung zu verringern. Dies kann durch Verringerung der für die Filterung benötigten Durchgänge erreicht werden. Um die Filterung in einem einzigen Durchgang durchzuführen, wird die Spaltenfilterung in zwei Schritten durchgeführt:

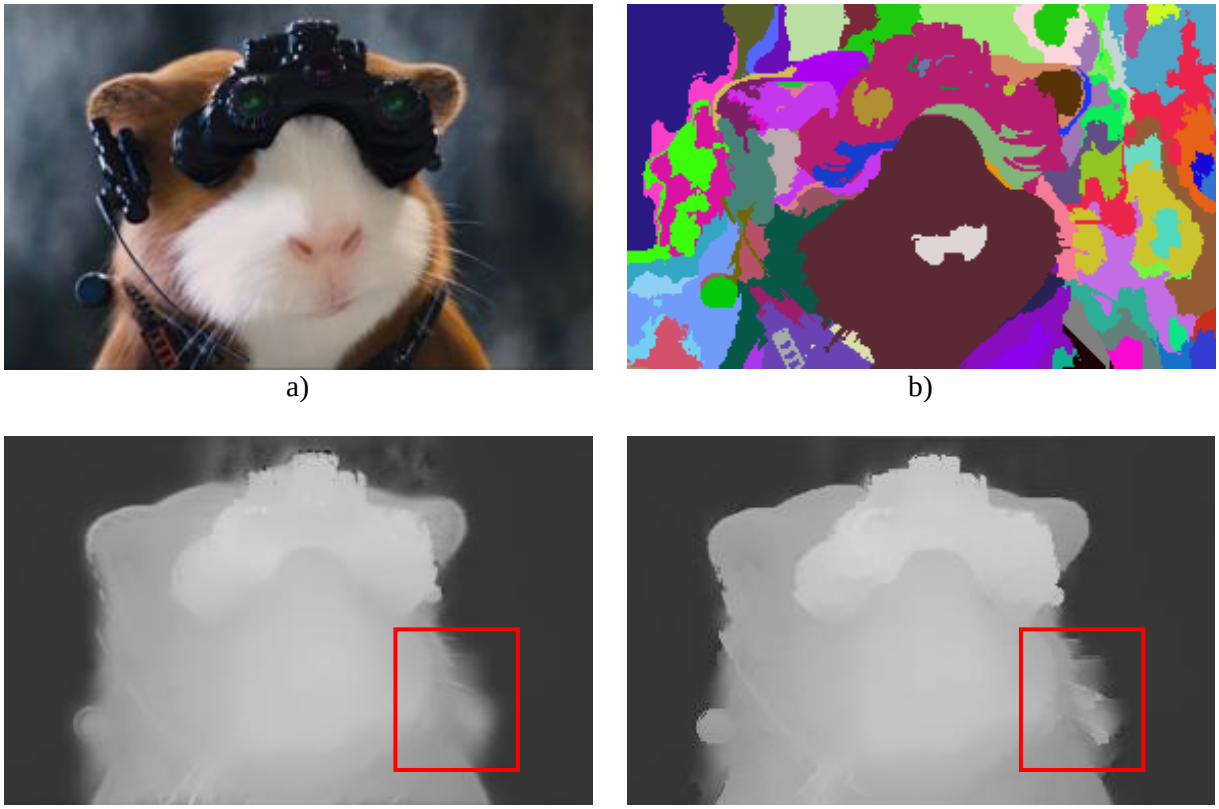
1. Ermittlung der Anzahl der Pixel pro Regionsabschnitt (Abbildung 6.12 a)). Dazu wird in einer Schleife ein Zähler so lange erhöht, bis ein Regionswechsel stattfindet. Der Zähler definiert somit die Höhe des entsprechenden Regionsabschnitts.

2. Filterung des entsprechenden Regionsabschnitts (Abbildung 6.12 b)). Die Teilregion wird bis zur in Schritt 1 bestimmten Höhe gefiltert. Das letzte Pixel einer Teilregion wird in diesem Schritt als unterer Rand betrachtet.

Dieser Vorgang wird solange wiederholt, bis die gesamte Bildspalte gefiltert wurde. Jede Teilregion wird wie ein eigenständiges Bild behandelt. Die Ränder der Regionen werden als Bildränder betrachtet und entsprechend der Randbehandlung, welche in Kapitel 4 beschrieben wurde, explizit bearbeitet. Die berechneten Laufsummen vor dem Regionswechsel fließen nicht in die Berechnung des

Mittelwertes ein. Beispielsweise wird in Abbildung 6.12 b) *Pixel 5* als unterer Randpixel der entsprechenden Teilregion betrachtet und *Pixel 9* als oberes Randpixel einer neuen Teilregion. Es fließen somit keine Pixelwerte der oberen Teilregion in die Mittelwertberechnungen der unteren Teilregion ein.

Der Zeilenfilter und der zeitliche Filter des Box-Filters werden analog zum Spaltenfilter adaptiert. Die restlichen Berechnungsschritte des Guided Filters (siehe Listing 6.1, Zeile 5-8, 11) erfordern keinerlei Adaptierung, da die pixelweisen Operationen unabhängig von der Segmentierung durchgeführt werden können. Im Vergleich der optimierten Implementierung des regionsweisen Guided Filters mit der naiven Implementierung konnten in unseren Experimenten keine sichtbaren Auswirkungen auf das Filterergebnis festgestellt werden.



a) b)

c) d)

Abbildung 6.13: *Vergleich zwischen Guided Filter und regionsweisem Guided Filter. a) Frame 8 eines Eingabevideos von 10 Frames. b) Regionen in dem Bild. Die IDs der Regionen sind farblich gekennzeichnet. c) Disparitätskarte nach Filterung mit dem Guided Filter. d) Disparitätskarte nach der Filterung mit dem regionsweisen Guided Filter ($r = 4$, $r_t = 4$, $\varepsilon = 0.0016$).*

Optionaler Nachbearbeitungsschritt

In einem optionalen Nachbearbeitungsschritt kann die gesamte Disparitätskarte ein weiteres Mal mittels Guided Filter geglättet werden. Als Leitvideo wird das originale Farbvideo verwendet. Das Eingabevideo ist die Disparitätskarte. Durch die Anwendung des Guided Filters für Videos bleiben Kanten, welche durch räumliche und zeitliche Änderungen zwischen Frames entstanden sind, erhalten.

Texturierte Regionen mit konstanten Disparitäten behalten ihre Disparitäten. Homogene Regionen mit unterschiedlichen Disparitäten werden geglättet. Aufgrund der strukturübertragenden Eigenschaft des Guided Filter können Disparitäten von feinen Bildstrukturen (beispielweise Haaren) verbessert werden (siehe Abbildung 6.14d)).

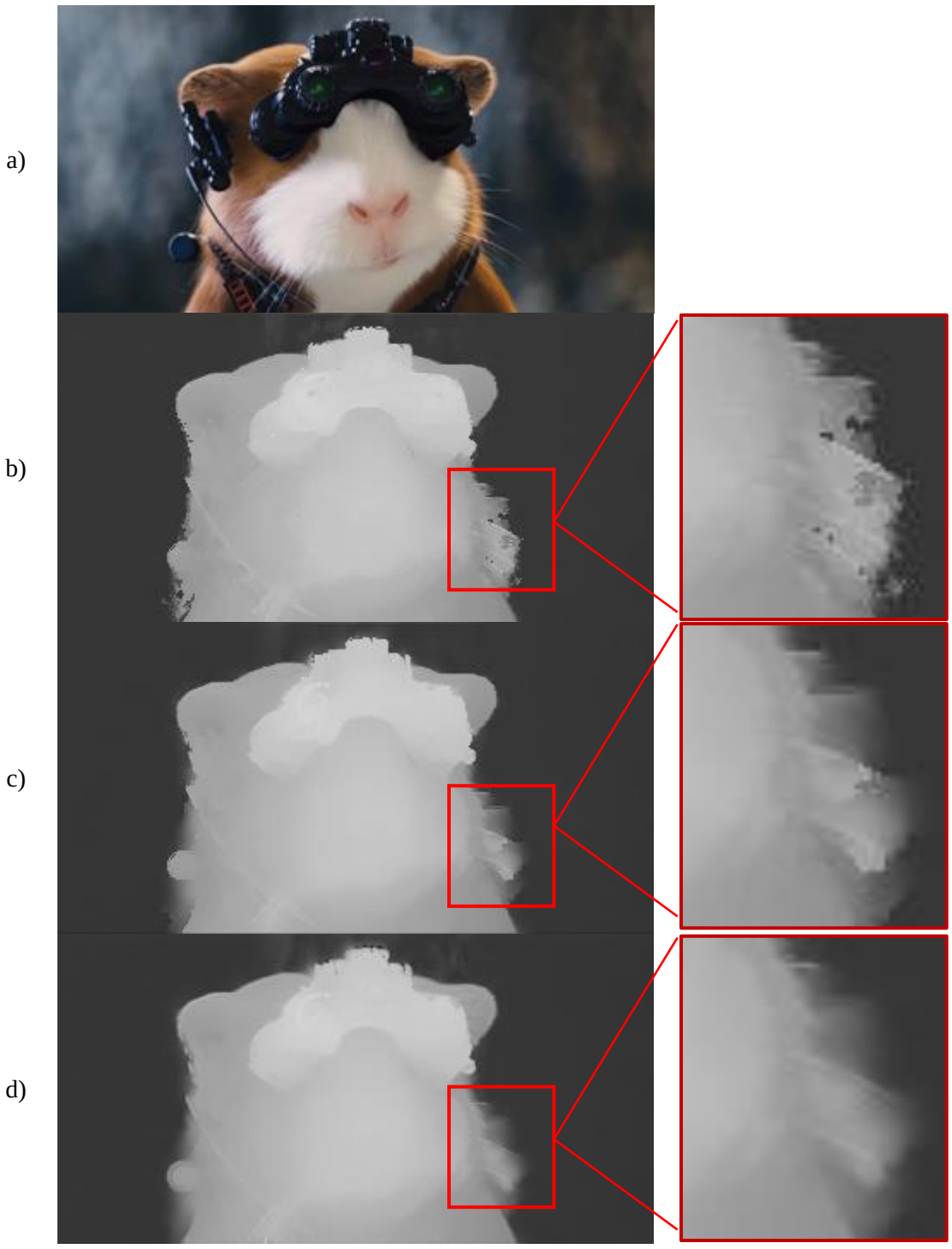


Abbildung 6.14: *a) Frame des Eingabevideos. b) Ergebnis nach der Segmentierung (Disparitätskarte). c) Ergebnis nach der Regularisierung ($r = 4$, $r_t = 5$, $\varepsilon = 0.0016$). d) Ergebnis nach der optionalen Nachbearbeitung ($r = 2$, $r_t = 1$, $\varepsilon = 0.00016$).*

a)

b)

c)

d)

Abbildung 6.15: a) *Frame des Eingabevideos. b) Ergebnis nach der Segmentierung (Disparitätskarte). b) Ergebnis nach der Regularisierung ($r = 4$, $r_t = 9$, $\varepsilon = 0.0016$). c) Ergebnis nach der optionalen Nachbearbeitung ($r = 2$, $r_t = 1$, $\varepsilon = 0.00016$).*

Ergebnisse

Abbildung 6.13 zeigt einen Vergleich zwischen einer Filterung mit dem Guided Filter (Abbildung 6.13c)) und einer Filterung mit dem regionsweisen Guided Filter (Abbildung 6.13d)). Das Filterergebnis der regionsweisen Filterung ist an den Regionsgrenzen schärfer. Homogene Bereiche innerhalb der Regionen wurden geglättet, die Kanten an den Regionsgrenzen blieben erhalten. Die

regionsweise Filterung ermöglicht die räumliche und zeitliche Regularisierung der Disparitäten, wodurch abrupte Disparitätssprünge vermieden werden können. Um zeitlich konsistente Disparitäten zu erhalten, wird im Regularisierungsschritt ein großes zeitliches Filterfenster verwendet (der Durchmesser des Filterfensters hat die Größe, die der Hälfte der Frames des Videos entspricht, die gefiltert werden soll). Das mit dem Guided Filter gefilterte Ergebnis (Abbildung 6.13c)) enthält mehr feine Details (beispielsweise Haare). Im optionalen Nachbearbeitungsschritt werden diese durch die Anwendung des Guided Filters auf die Disparitätskarten des gesamten Eingabevideos hinzugefügt (siehe Abbildung 6.14 und Abbildung 6.15). Da der Guided Filter im Nachbearbeitungsschritt, im Gegensatz zum Regularisierungsschritt, auf das gesamte Eingabevideo angewandt wird (unabhängig von den Regionen) wird ein kleineres Filterfenster verwendet (zum Beispiel drei Frames). Mehr Ergebnisse sind im Kapitel 7 zu finden.

6.5. Laufzeitvergleiche

In diesem Abschnitt werden die Laufzeiten der unterschiedlichen Implementierungen des Guided Filters miteinander verglichen. Für die Laufzeitvergleiche wurden die Algorithmen auf einem PC mit einem Intel Core i7 3.6GHz, 16GB RAM und einer NVIDIA GeForce GTX 660ti ausgeführt.

Vergleich: Sequentieller und paralleler Guided Filter für Bilder

Abbildung 6.16 zeigt einen Vergleich zwischen der sequentiellen MATLAB-Implementierung aus [23] und einer parallelen GPU-Implementierung eines Guided Filters für Bilder. Gefiltert wurden ein 8-Bit Graustufenbild mit der Auflösung 1024 x 1024. Als Leitbild wurde ein 24-Bit Farbbild (RGB) mit derselben Auflösung verwendet. Die parallele GPU-Implementierung läuft mit einer Laufzeit von 15.7 Millisekunden um den Faktor 906.6 schneller als die sequentielle MATLAB-Implementierung mit einer Laufzeit von 14.234 Sekunden. Der Laufzeitunterschied ist auf die Parallelisierung des Guided Filters zurückzuführen, wobei die Verwendung der optimierten GPU-Version des Box-Filters den größten Einfluss auf die Laufzeit der parallelen GPU-Implementierung des Guided Filters hat.

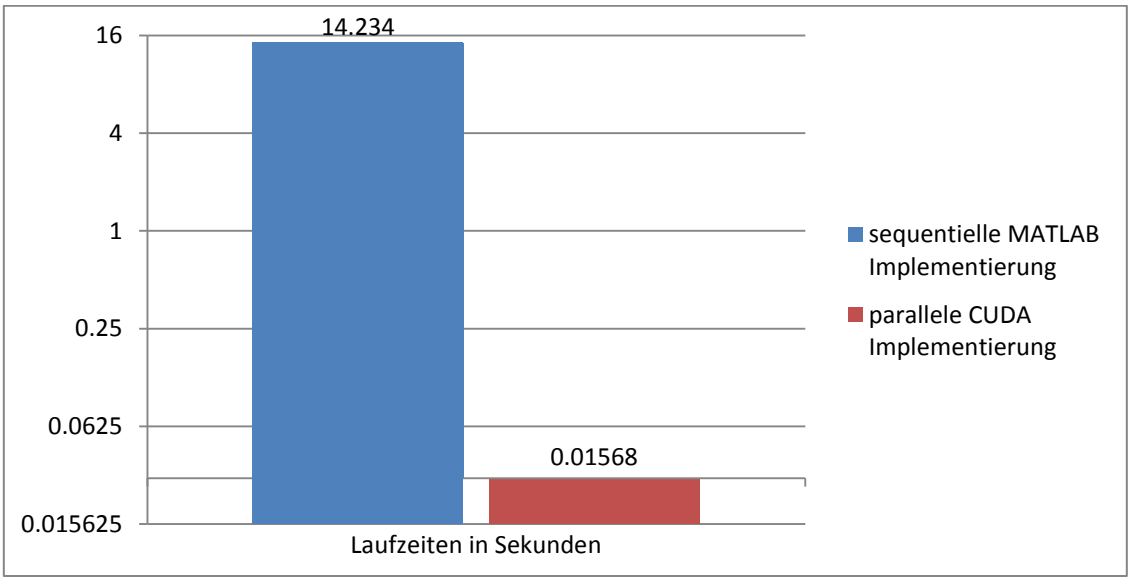


Abbildung 6.16: *Vergleich der Laufzeiten der sequentiellen MATLAB-Implementierung des Guided Filters für Bilder aus* [23] *und der parallelen CUDA Implementierung aus Abschnitt 6.3. Gefiltert wurde ein 8-Bit Grauwertbild mit der Auflösung 1024 x 1024 mit einem entsprechenden Farbbild derselben Auflösung als Leitbild (r = 4, ε = 0.0016).*

Vergleich: Sequentieller und paralleler Guided Filter für Videos

Es wurden eine sequentielle MATLAB-Implementierung und eine parallele GPU-Implementierung eines Guided Filters zur Bearbeitung von Videos verglichen. Für den Vergleich wurde jeweils ein 8-Bit Graustufenvideo von 10 Frames mit der Auflösung 1024 x 1024 gefiltert. Als Leitvideo wurde ein Farbbild mit denselben Dimensionen verwendet. Der Laufzeitvergleich in Abbildung 6.17 zeigt bei beiden Implementierungen eine Steigerung linear zur Anzahl der Frames.

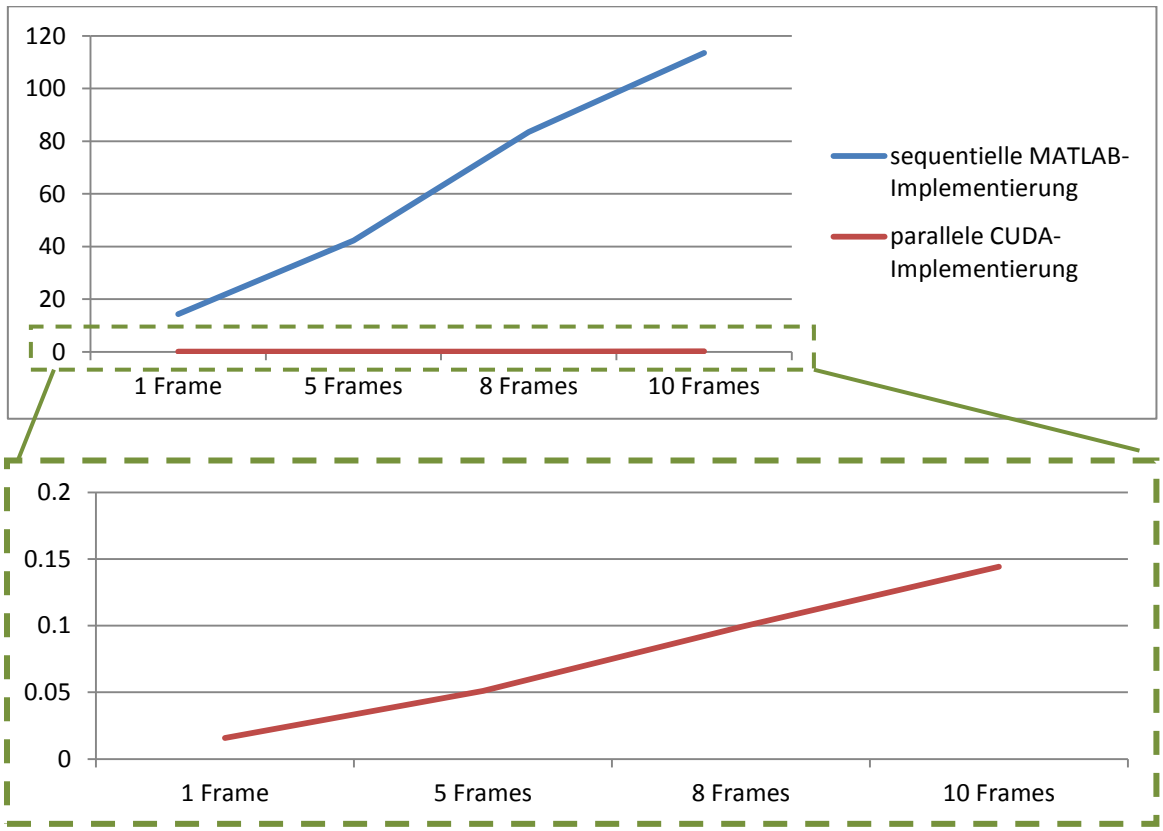


Abbildung 6.17: *Vergleich der sequentiellen MATLAB-Implementierung und der parallelen GPU-Implementierung eines Guided Filters für Videos. Gefiltert wurde ein 8-Bit Grauwertvideo mit der Auflösung 1024 x 1024. Als Leitvideo wurde ein 24-Bit Farbbild mit derselben Auflösung verwendet. Die Laufzeiten beider Versionen steigen konstant mit der Anzahl der Frames. Um das sichtbar zu machen, werden die Laufzeitergebnisse der parallelen GPU-Version in der unteren Grafik in einem verringerten Zeitraum betrachtet ($r = 4$, $r_t = 2$, $\varepsilon = 0.0016$).*

Vergleich: Guided Filter und regionsweiser Guided Filter für Videos

Abbildung 6.18 zeigt einen Vergleich der Laufzeiten des Guided Filters für Videos und der optimierten Implementierung des regionsweisen Guided Filters für Videos. Für den Laufzeitvergleich wurden die Disparitätskarten eines Videos von 10 Frames mit der Auflösung 424 x 229 gefiltert. Als Leitvideo wurde das originale Farbvideo verwendet. Im Vergleich zur naiven Implementierung des regionsweisen Guided Filters steigt die Laufzeit der optimierten Version nur leicht mit der Anzahl der Regionen (siehe Abbildung 6.18). Die erhöhte Laufzeit des optimierten regionsweisen Guided Filters hängt mit der adaptierten Version des Box-Filters zusammen. Die Laufzeit des adaptierten Box-Filters erhöht sich durch den zusätzlichen Aufwand beim Wechsel zwischen den Regionen. Im Laufzeitvergleich ist zu sehen, dass die Laufzeit der optimierten Version des regionsweisen Guided

Filters abhängig von der Anzahl der Regionen höher ist als die Laufzeit der Version des Guided Filters ohne Berücksichtigung der Regionen (siehe Abschnitt 6.3).

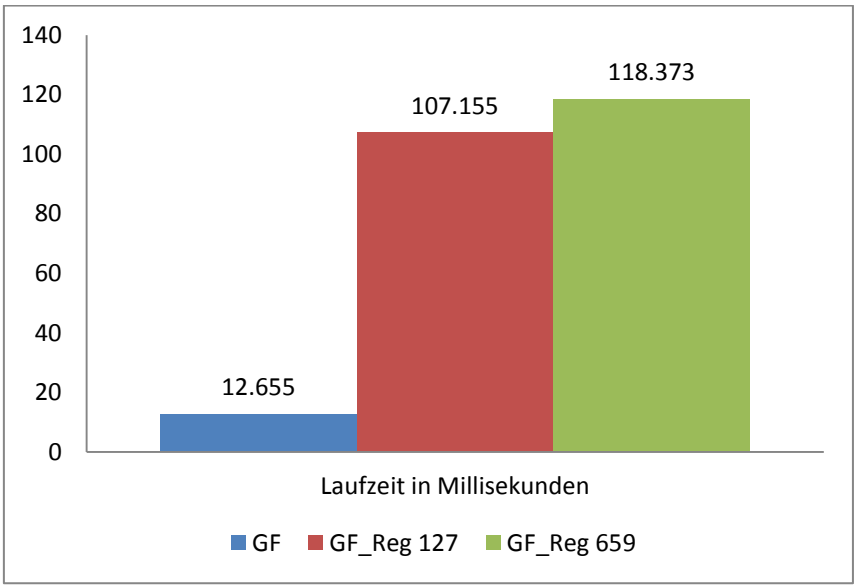


Abbildung 6.18: *Performancevergleichen zwischen Guided Filter (GF) und regionsweisem Guided Filter für ein Eingabevideo von 10 Frames mit 127 Regionen (GF_Reg 127) und 659 Regionen (GF_Reg 659).*

7. Resultate und Laufzeiten

Dieses Kapitel zeigt Resultate und Laufzeiten der in dieser Diplomarbeit erstellten optimierten Implementierung zur segmentierungsbasierten Propagierung von Disparitäten in Videos. Die Resultate werden in diesem Kapitel evaluiert. Bei den Disparitätskarten, welche in diesem Kapitel angegeben werden, handelt es sich um die finalen Endergebnisse der segmentierungsbasierten Propagierung nach durchgeführter Regularisierung und zusätzlicher Nachbearbeitung. Für die letzten beiden Bearbeitungsschritte wurden konstante Parameter für den Guided Filter verwendet (Regularisierung: $r = Frames/4$, $r_t = 10$, $\varepsilon = 0.0016$. Nachbearbeitung: $r = 2$, $r_t = 4$, $\varepsilon = 0.0016$). Die Resultate in diesem Abschnitt wurden auf einem PC mit einem Intel Xenon E5 Prozessor mit 3.6GHz, 32GB RAM und einer NVIDIA GeForce GTX 680 Grafikkarte generiert. Abschnitt 7.1 enthält die resultierenden Disparitätskarten nach Anwendung des Algorithmus auf elf Testvideos. Diese werden im Zuge eines Vergleiches mit Disparitäten aus Referenzlösungen evaluiert. Im Zuge dessen wird die Auswirkung der Verwendung von externen Optical Flow Informationen diskutiert und ein Vergleich der Ergebnisse mit und ohne Verwendung von Optical Flow durchgeführt. Abschnitt 7.2 evaluiert die Aufteilung von langen Videos in Subsequenzen. Dazu werden die Qualität der Disparitätskarten und die Laufzeiten mit und ohne Aufteilen der Videos verglichen. In Abschnitt 7.3 wird ein Vergleich der Ergebnisse der Implementierung dieser Diplomarbeit mit Ergebnissen verwandter Propagierungs-Algorithmen durchgeführt.

Nr.	Name	Frames	Auflösung	TS	Iter	k	C_{min}
1	Stadt	19	699 x 282	1	1	0.04	200
2	Parade	11	689 x 282	1	4	0.04	110
3	Schloss	10	702 x 278	1	1	0.02	100
4	Treppe	20	702 x 278	1	1	0.04	200
5	Fußball	21	669 x 282	2	4	0.04	200
6	Kind	21	600 x 338	2	1	0.02	200
7	Kopf	81	600 x 330	4	1	0.10	300
8	Interview	101	600 x 480	6	1	0.02	200
9	Tsukuba 50-66	17	640 x 480	1	1	0.02	200
10	Tsukuba 380-397	18	640 x 480	1	1	0.02	200
11	Tsukuba 1-100	100	640 x 480	8	1	0.02	200

Tabelle 7.1: *Testvideos, welche zur Evaluierung verwendet wurden. Die Einträge enthalten jeweils die Nummerierung und den Namen der Testvideos, die Anzahl der Frames, ihre Auflösung, die Anzahl der Teilsequenzen, in welchen das Video bearbeitet wurde (TS), sowie die Parameter des Segmentierungs-Algorithmus: Iterationen (Iter), k und minimale Regionsgröße (C_{min})(siehe Kapitel 5).*

Testvideos

Tabelle 7.1 gibt einen Überblick über die Testvideos, welche im Zuge der Evaluierung verwendet werden. Im ersten und letzten Frame jedes Videos wurden im Vorfeld Scribbles eingezeichnet. Diese blieben für alle Tests in diesem Kapitel unverändert. Bei den Videos 1 bis 5 handelt es sich um dieselben Testvideos, die auch in [22] zur Evaluierung verwendet wurden. Die Disparitätskarten, welche für diese Videos als Referenzlösungen verwendet werden, wurden mithilfe des Stereo Matching-Algorithmus aus [3] generiert. Die Videos 6 bis 8 stammen von [68]. Sie enthalten Referenz-Disparitäten für eine Reihe von Schlüssel-Frames, aber im Gegensatz zu den restlichen

Testvideos nicht für alle Frames der Videos. Für die Berechnung der Abweichung von der Referenzlösung wurden daher nur jene Frames verwendet, für welche Referenz-Disparitäten zur Verfügung stehen. Bei den Testvideos 9 bis 11 handelt es sich um Auszüge aus dem *Tsukuba Stereo Dataset* [69, 70], bei dem Referenz-Disparitäten für das gesamte Video zur Verfügung stehen.

7.1. Evaluierung der Disparitätskarten

In diesem Abschnitt werden die resultierenden Disparitätskarten des Algorithmus zur segmentierungsbasierten Propagierung von Disparitäten in Videos mit den Referenz-Disparitäten verglichen. Zu diesem Zweck werden jeweils im ersten und letzten Frame eines Eingabevideos Referenz-Disparitäten an den Positionen der Scribbles eingetragen. Die Qualität der resultierenden Disparitätskarten wird durch Messung der mittleren quadratischen Abweichung *MSE* (*mean squared error*) von den Referenz-Disparitäten evaluiert [63]:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2 \tag{7.1}$$

Hier entspricht n der Anzahl der Pixel im Eingabevideo. Y_i entspricht der Disparität an der Pixelposition i im Eingabevideo, $\hat{Y}_i$ der entsprechenden Referenz-Disparität. Zum Zweck der besseren Lesbarkeit wurde der *MSE* in dieser Evaluierung mit dem Faktor 100 multipliziert. Die Abweichung zu den Referenz-Disparitäten wird zusätzlich mithilfe von Fehlerbildern visualisiert, in welchen der Fehler durch Grauwerte dargestellt wird (siehe Abbildung 7.1 d)). Je heller ein Pixel im Fehlerbild ist, desto größer ist die Abweichung an dieser Bildposition. Die Helligkeit und der Kontrast der Fehlerbilder wurde nachträglich jeweils um 20% erhöht, um die Abweichungen zu den Referenzlösungen besser sichtbar zu machen.

Video	MSE x 100		Laufzeit				
	Mit OF	Ohne OF	Seg/Pro	FM	Reg	Fein	Gesamt
Stadt	**0.08**	0.09	12.37	0.00	0.78	0.38	**13.53**
Parade	**0.12**	0.13	10.05	0.00	0.36	0.14	**10.55**
Schloss	**0.17**	0.17	6.69	0.00	0.34	0.13	**7.16**
Treppe	**0.12**	0.12	11.73	0.00	0.78	0.22	**12.73**
Fußball	**0.08**	0.10	34.57	2.98	1.56	0.44	**39.55**
Kind	**0.07**	0.07	27.76	1.06	2.15	0.49	**31.46**
Kopf	**0.44**	0.52	45.20	3.76	1.86	0.88	**51.69**
Interview	**0.37**	0.43	94.17	6.52	8.19	2.97	**111.85**
Tsukuba 50-66	**0.02**	0.02	15.24	0.00	1.11	0.27	**16.61**
Tsukuba 380-397	**0.16**	0.18	15.96	0.00	1.12	0.36	**17.44**
Tsukuba 1-100	**0.11**	0.11	96.71	6.67	8.09	2.36	**113.82**

Tabelle 7.2: *MSE (um den Faktor 100 skaliert) und Laufzeiten in Sekunden. Der MSE wird jeweils mit und ohne Verwendung von Optical Flow angegeben. Die Gesamtlaufzeit gliedert sich jeweils in die Laufzeit für: 1. die Segmentierung/Propagierung (Seg/Pro), 2. Feature Matching, im Fall, dass das Video aufgeteilt wird (FM), 3. die Regularisierung der Disparitäten (Reg) und 4. die Nachbearbeitung (Fein).*

Um die Auswirkungen der zusätzlichen Verwendung von Optical Flow zu analysieren, wurden die Testvideos in diesem Abschnitt jeweils mit und ohne Verwendung von Optical Flow bearbeitet. Der Optical Flow wurde zu diesem Zweck, wie auch in [22] und [25], mithilfe des Algorithmus aus [30] berechnet.

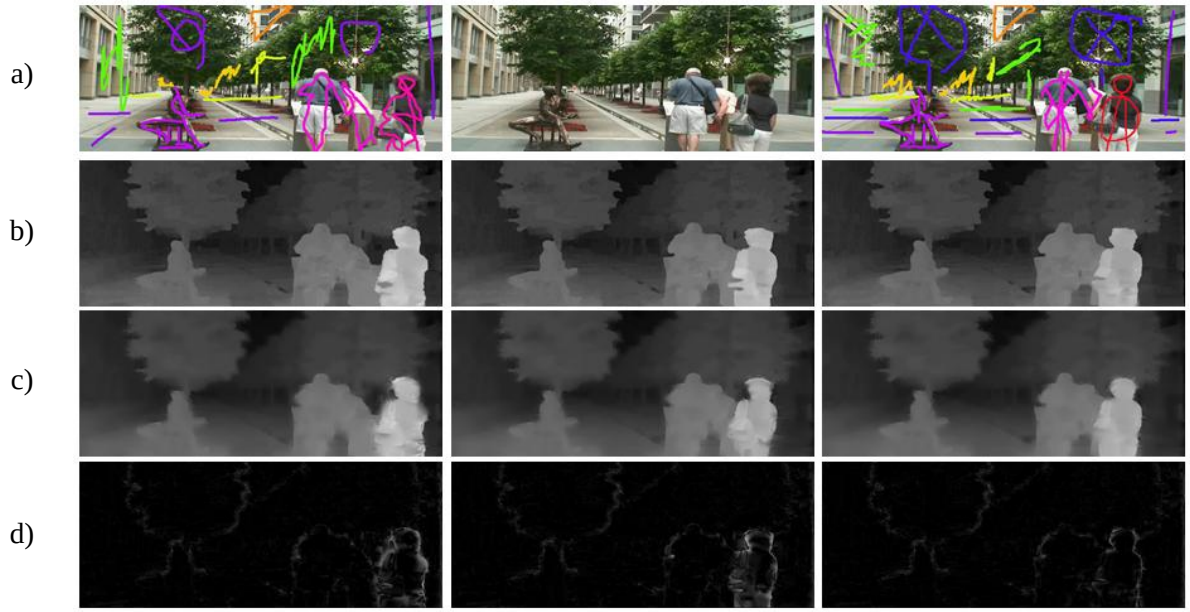


Abbildung 7.1: *Ergebnisse des Testvideos „Stadt“. Von links nach rechts: Frame 1, Frame 9, Frame 18. a) Eingabeframes. b) Referenzlösung. c) Ergebnis der segmentierungsbasierten Propagierung. d) Fehler zur Referenzlösung (MSE* x *100 = 0.08). Abweichungen von der Referenzlösung treten hauptsächlich an den Objektgrenzen von sich bewegenden Objekten auf.*

Abbildung 7.2: *Ergebnisse des Testvideos „Tsukuba 50-66“. Von oben nach unten: Frame 1, Frame 9, Frame 17. a) Eingabeframes. b) Referenzlösung. c) Ergebnis der segmentierungsbasierten Propagierung. d) Fehler zur Referenzlösung (MSE* x *100= 0.02). Im Video findet eine leichte Kamerabewegung statt.*

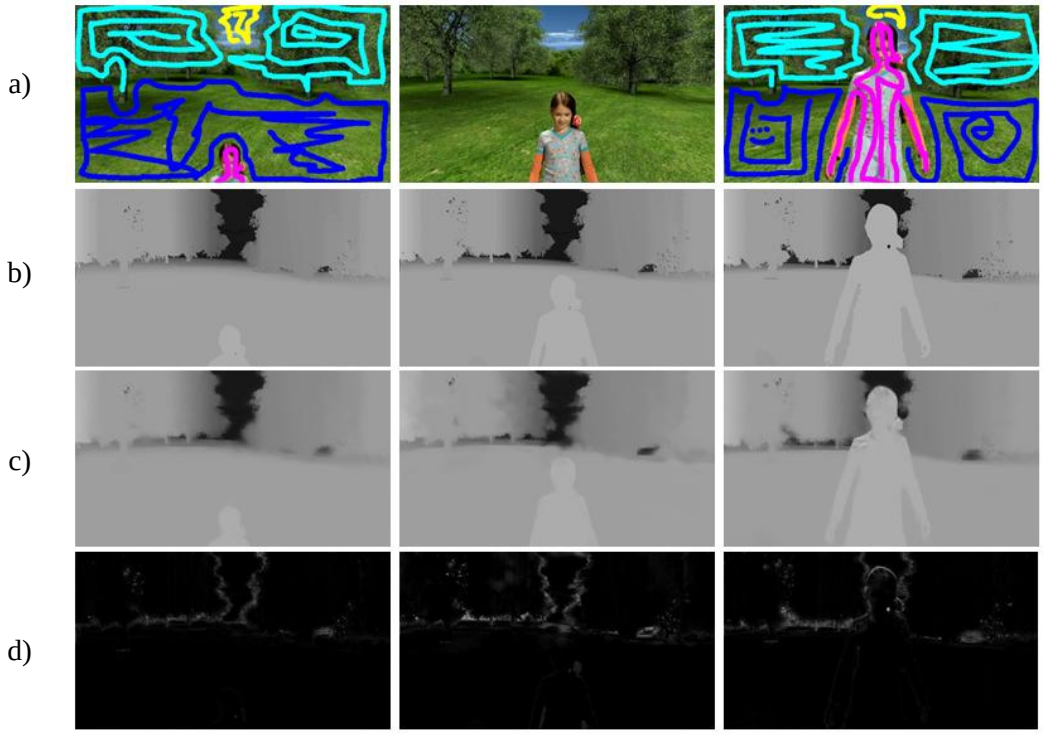
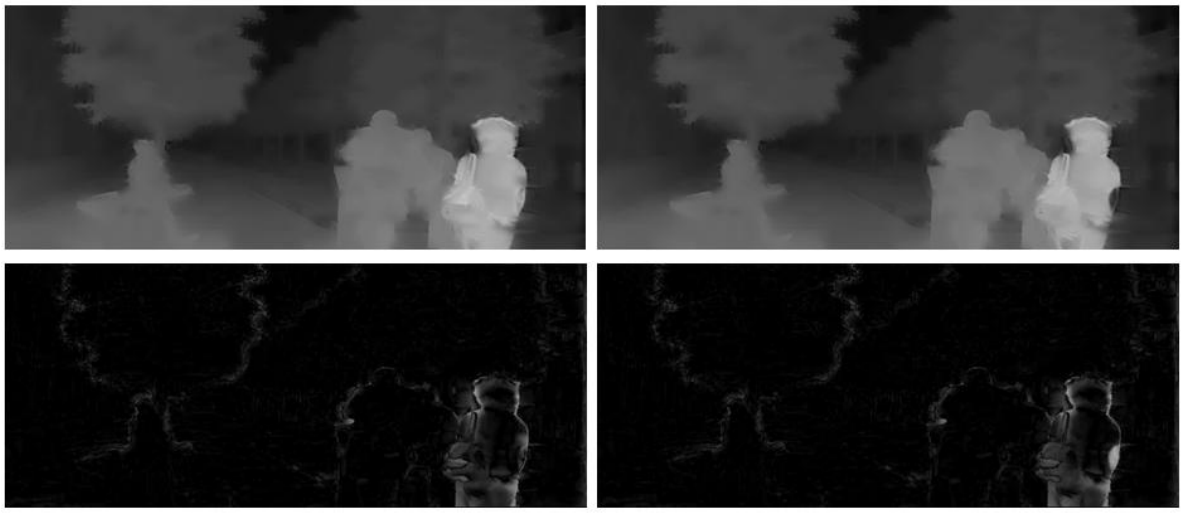


Abbildung 7.3: *Ergebnisse des Testvideos „Kind“. Von links nach rechts: Frame 1, Frame 21, Frame 41. a) Eingabeframes. b) Referenzlösung. c) Ergebnis der segmentierungsbasierten Propagierung. d) Fehler zur Referenzlösung (MSE* x *100= 0.07). Es findet eine Kamerabewegung und ein Zoom statt.*

Abbildung 7.4: *Ergebnisse des Testvideos „Interview“. Von oben nach unten: Frame 1, Frame 51, Frame 101. a) Eingabeframes. b) Referenzlösung. c) Ergebnis der segmentierungsbasierten Propagierung. d) Fehler zur Referenzlösung (MSE* x *100= 0.37). Aufgrund der geringen Farbunterschiede an den Objektgrenzen kommt es beispielsweise im Bereich des Fensters zu Fehlern.*



a) b)

Abbildung 7.5: *a) Resultat eines Frames des Testvideos „Stadt“ unter Verwendung von Optical Flow (MSE* x *100 = 0.08). b) Resultat ohne Optical Flow (MSE* x *100= 0.09). Im Bereich der Frau im Vordergrund kommt es im Ergebnis ohne Optical Flow zu Fehlern.*

a) b)

Abbildung 7.6: *a) Resultat eines Frames des Testvideos „Fußball“ unter Verwendung von Optical Flow (MSE* x *100= 0.08). b) Resultat ohne Optical Flow (MSE* x *100= 0.10). Im Video finden rasche Bewegungen der Spieler statt. Im Ergebnis ohne Optical Flow kommt es im Bereich der sich bewegenden Spieler zu Fehlern.*

Tabelle 7.2 und Abbildung 7.1 – 7.6 zeigen Ergebnisse dieser Evaluierung. Wie beispielsweise in Abbildung 7.1 zu sehen ist, genügen bereits wenige Scribbles in den Schlüssel-Frames, um eine zeitlich konsistente Propagierung der Disparitäten zu erhalten, welche nahe an der Referenzlösung liegt. Abweichungen treten hauptsächlich an den Regionsgrenzen auf. Auch bei Bewegungen von Kamera und Bildobjekten werden qualitativ hochwertige Disparitätskarten generiert. Limitierungen konnten in Videos beobachtet werden, in denen geringe Farbunterschiede an den Regionsgrenzen

auftreten (siehe Abbildung 7.4). Aus den Laufzeitergebnissen aus Tabelle 7.2 ergibt sich für ein Video mit einer Auflösung von 653 x 363 eine durchschnittliche Laufzeit von 1.02 Sekunden pro Frame. Aus den *MSE*-Werten in Tabelle 7.2 und den Vergleichen in Abbildung 7.5 und Abbildung 7.6 ist ersichtlich, dass die Verwendung von Optical Flow zu Verbesserungen führen kann. Diese fallen in Anbetracht des zusätzlichen Rechenaufwandes, welcher durch die Berechnung des Optical Flow entsteht, jedoch gering aus (siehe Tabelle 7.2). Wie auch in [22] erwähnt, kann daher aufgrund des geringen Einflusses auf die Qualität der Ergebnisse, auf die Verwendung von Optical Flow verzichtet werden.

a)

b)

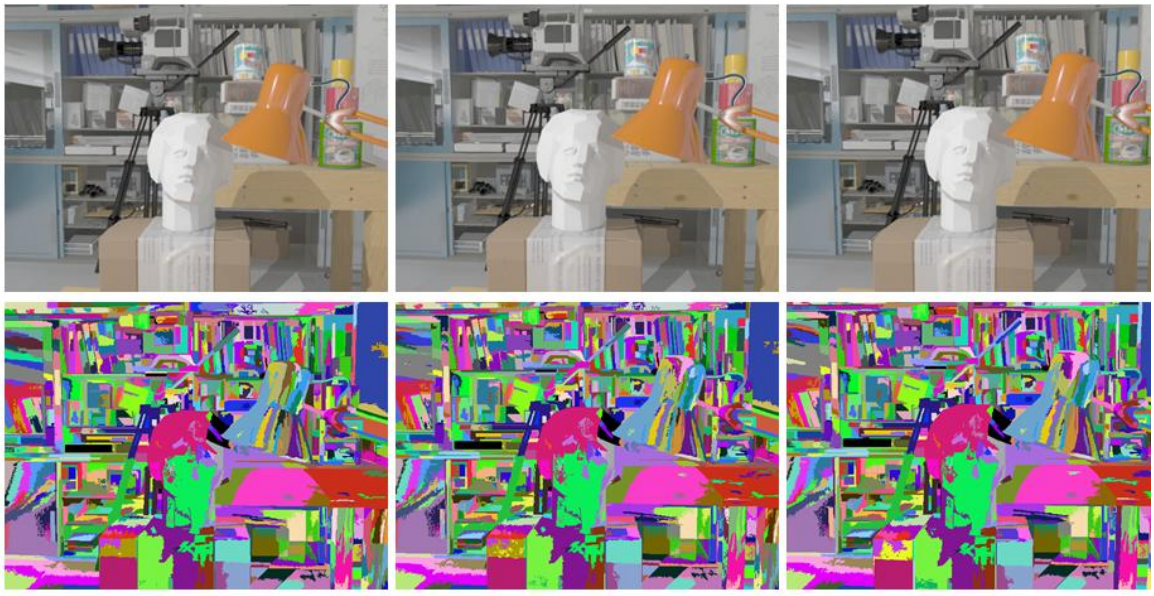


Abbildung 7.7: *Segmentierungsergebnisse des Videos „Tsukuba 50-66“. Von links nach rechts: Frame 1, Frame 9, Frame 17. a) Eingabeframes. b) Segmentierungsergebnis.*

Im Zuge der Optimierung, welche in dieser Diplomarbeit durchgeführt wurde, wurde auch die Laufzeit des Segmentierungs-Algorithmus aus [25] verringert. Die optimierte Implementierung des Algorithmus zur segmentierungsbasierten Propagierung von Disparitäten in Videos liefert daher als sekundäres Ergebnis auch eine Segmentierung des Eingabevideos (siehe Abbildung 7.7). Der Fokus dieser Diplomarbeit liegt auf der effizienten Generierung von Disparitätskarten. Daher wurde im Zuge der Segmentierung weniger Wert auf die Qualität der Segmentierungsergebnisse gelegt, als auf die Qualität der Ergebnisse der Propagierung.

7.2. Vergleich der Bearbeitung von Videos in Subsequenzen mit der Bearbeitung als Ganzes

In diesem Abschnitt werden die Auswirkungen des Aufteilens eines Videos in Subsequenzen diskutiert. Zu diesem Zweck wurden sechs Testvideos jeweils in zwei und in drei Subsequenzen unterteilt, welche anschließend getrennt bearbeitet wurden. Die Ergebnisse werden mit den Ergebnissen der Bearbeitung der entsprechenden Videos ohne Aufteilung verglichen. Für diesen Vergleich wurden nur Testvideos (aus Tabelle 7.1) verwendet, welche klein genug sind, um in einem einzigen Durchlauf bearbeitet werden zu können. Wie anhand des *MSE* in Tabelle 7.2 und der Ergebnisframes in Abbildung 7.8 und Abbildung 7.9 zu sehen ist, steigt die Abweichung zur

Referenzlösung der Testvideos mit der Anzahl der Subsequenzen, was, wie in Kapitel 5 beschrieben, daran liegt, dass es durch die Propagierung der Disparitäten auf die Schlüssel-Frames der Subsequenzen zu Fehlern kommen kann. Die Qualität der Ergebnisse hängt in diesem Fall von den Ergebnissen des Feature Matching-Algorithmus ab. Aus Tabelle 7.3 ist zu entnehmen, dass die Propagierung der Disparitäten mittels Feature Matching für computergenerierte Testvideos robuster ist als für reale Videoaufnahmen. Im Gegensatz zu den anderen Testvideos verschlechtert sich der *MSE* der computergenerierten Videos „Tsukuba 50-66“ und „Tsukuba 380-397“ durch das Aufteilen nicht. Das Video „Treppe“ stellt eine Ausnahme zu den restlichen Testvideos dar. Der *MSE* der Ergebnisse verschlechtert sich zwar durch das Aufteilen in Subsequenzen, jedoch ist der *MSE* bei drei Subsequenzen geringer als bei zwei Subsequenzen. Das liegt daran, dass das Feature Matching in diesem Testvideo beim Aufteilen in drei Subsequenzen für die beiden neuen Schlüssel-Frames (*Frame 4* und *Frame 7*) robuster ist als bei jenem Schlüssel-Frame beim Aufteilen in zwei Subsequenzen (*Frame 6*). Tabelle 7.3 zeigt, dass die Laufzeit durch das Aufteilen mit der Anzahl der Subsequenzen steigt, was auf den zusätzlichen Aufwand zur Propagierung der Disparitäten auf die neuen Schlüssel-Frames der Subsequenzen zurückzuführen ist.

Video	MSE x 100			Laufzeit		
	1 TS	2 TS	3 TS	1 TS	2 TS	3 TS
Stadt	**0.08**	0.11	0.16	**12.61**	13.24	14.01
Parade	**0.12**	0.17	0.27	**10.55**	11.39	14.34
Treppe	**0.12**	0.27	0.21	**12.73**	12.87	13.51
Schloss	**0.17**	0.17	0.20	**7.16**	8.83	10.71
Tsukuba 50-66	**0.02**	0.02	0.02	**16.61**	17.10	18.20
Tsukuba 380-397	**0.16**	0.16	0.16	**17.44**	18.12	19.07

Tabelle 7.3: *MSE (um den Faktor 100 skaliert) und Laufzeiten in Sekunden für die Bearbeitung der Testvideos als Ganzes (1 TS), in zwei Subsequenzen (2 TS) und in drei Subsequenzen (3 TS).*

Abbildung 7.8: *Ergebnisse eines Frames des Videos „Stadt“ mit und ohne Aufteilen in Subsequenzen. Links: Abweichung zur Referenzlösung. Rechts: Resultierende Disparitäten. a) Ergebnis des Videos als Ganzes (MSE x 100= 0.08). b) Ergebnis bei einer Aufteilung in zwei Subsequenzen (MSE x 100 = 0.11). c) Ergebnis bei einer Aufteilung in drei Subsequenzen (MSE x 100= 0.16). Im Zuge des Aufteilens kommt es im Bereich der Frau im Vordergrund zu Fehlern, welche mit der Anzahl der Subsequenzen zunehmen.*

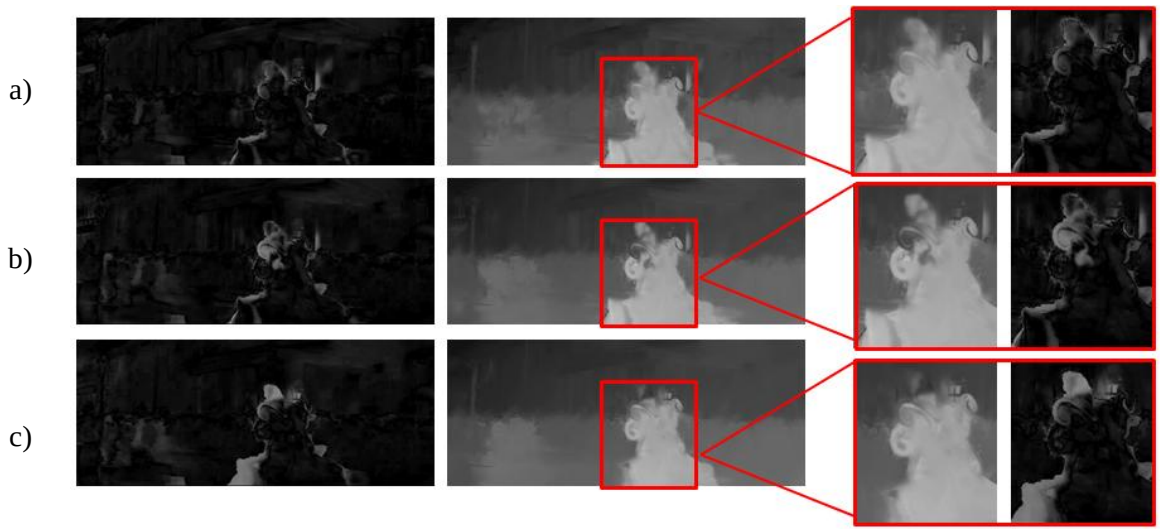


Abbildung 7.9: *Ergebnisse eines Frames des Videos „Parade“ mit und ohne Aufteilen in Subsequenzen. Links: Abweichung zur Referenzlösung. Rechts: Resultierende Disparitäten. a) Ergebnis des Videos als Ganzes (MSE* x *100= 0.12). b) Ergebnis bei einer Aufteilung in zwei Subsequenzen (MSE* x *100= 0.17). c) Ergebnis bei einer Aufteilung in drei Subsequenzen (MSE* x *100 = 0.27).*

7.3. Vergleich mit anderen Algorithmen

In diesem Abschnitt wird die optimierte Implementierung des Algorithmus zur segmentierungsbasierten Propagierung von Disparitäten in Videos, welche in dieser Diplomarbeit erstellt wurde, mit zwei anderen Algorithmen zur Propagierung von Disparitäten in Videos verglichen: 1. dem Propagierungs-Algorithmus aus [14] und 2. dem Stereo Extraktions-Algorithmus aus [1]. Für diesen Vergleich wurden jeweils unsere Re-Implementierungen dieser beiden Algorithmen verwendet. Um einen möglichst exakten Vergleich durchführen zu können, wurden alle Algorithmen auf dieselben Testvideos mit demselben Input angewandt. Im Algorithmus von [14] werden jeweils die Disparitäten im gesamten ersten Frame des Videos benötigt, welche anschließend mithilfe des Bilateral Filters auf die nachfolgenden Frames propagiert werden. Für diesen Vergleich wurden jeweils die Disparitäten der Referenzlösung für das gesamte erste Frame jedes Testvideos verwendet. Im Gegensatz dazu wurden in der Implementierung aus dieser Diplomarbeit die Disparitäten aus der Referenzlösung nur an den Positionen der Scribbles in den Schlüssel-Frames verwendet. Der Stereo Extraktions-Algorithmus aus [1] benötigt ebenfalls Scribbles im ersten und letzten Frame eines Videos, welche die Disparitäten codieren. Diese werden durch die Lösung eines linearen Gleichungssystems auf die restlichen Frames des Videos propagiert. Um zeitlich konsistente Disparitäten zu erhalten, werden in [1] Optical Flow-Informationen aus [30] verwendet. Für den Vergleich mit der Implementierung aus dieser Diplomarbeit wurden jeweils dieselben Testvideos mit denselben Scribbles in den Schlüsselframes sowie dieselben Optical Flow-Informationen verwendet. An den Positionen der Scribbles wurden jeweils die Disparitäten aus der Referenzlösung verwendet. Wie aus Tabelle 7.4 ersichtlich ist, erzeugt der Algorithmus zur segmentierungsbasierten Propagierung von Disparitäten, für alle Testvideos Disparitätskarten mit einer wesentlich geringeren Abweichung zur Referenzlösung als unsere Re-Implementierungen der Algorithmen aus [14] und [1]. In Abbildung 7.11 ist beispielsweise zu sehen, dass die Vorwärtsbewegungen der Frau im Vordergrund vom Algorithmus aus [14] nicht korrekt erfasst werden, was zu einer erhöhten Abweichung von der Referenzlösung führt. In unserer Re-Implementierung wird keine explizite Randbehandlung

durchgeführt, was dazu führt, dass der Randbereich, abhängig von der Größe des Filterfensters des Bilateral Filters, abgeschnitten wird, was in Abbildung 7.11 b) am schwarzen Balken im unteren Bildbereich zu sehen ist. In diesen Bildbereichen kommt es daher zu einer erhöhten Abweichung von der Referenzlösung. In dem Video aus Abbildung 7.12 findet weniger Bewegung statt, was beim Algorithmus aus [14], mit Ausnahme der Bildränder, zu einer geringeren Abweichung von der Referenzlösung führt. Wie in Abbildung 7.11 und Abbildung 7.12 zu sehen ist, weichen die Disparitäten der Ergebnisse unserer Re-Implementierung des Algorithmus von [1] zu einem höheren Grad von den Disparitäten der Referenzlösung ab, was in einem entsprechend höheren *MSE* resultiert. Der Vergleich in Tabelle 7.3 zeigt, dass die Laufzeit der optimierten Implementierung des Algorithmus zur segmentierungsbasierten Propagierung von Disparitäten mit durchschnittlich 0.98 Sekunden pro Frame (für ein Video mit der Auflösung 640 x 480) deutlich geringer ist, als die Laufzeit unserer Re-Implementierung (in C++) des Algorithmus aus [14], welche durchschnittlich 19.29 Sekunden benötigt, und der Laufzeit unserer Re-Implementierung (in MATLAB) des Algorithmus aus [1], mit durchschnittlich 41.18 Sekunden.

Video	MSE x 100			Laufzeit		
	SBDP	[14]	[1]	SBDP	[14]	[1]
Stadt	**0.08**	1.92	12.15	**13.53**	191.86	378.44
Parade	**0.12**	2.06	8.38	**10.55**	110.85	787.25
Fußball	**0.08**	1.09	3.54	**39.55**	558.52	620.70
Schloss	**0.17**	1.81	9.26	**7.16**	111.72	379.83
Treppe	**0.12**	1.38	3.90	**12.73**	242.40	216.38
Tsukuba 50-66	**0.02**	0.34	7.24	**16.61**	327.32	783.28
Tsukuba 380-397	**0.16**	1.78	11.62	**17.44**	347.68	652.62

Tabelle 7.4: *Vergleich zwischen dem Algorithmus zur segmentierungsbasierten Propagierung von Disparitäten in Videos (SBDP) und zwei anderen Algorithmen: 1. dem Algorithmus zur Propagierung von Disparitäten mittels Bilateral Filter aus* [14] *und 2. dem Stereo Extraktions-Algorithmus aus* [1].

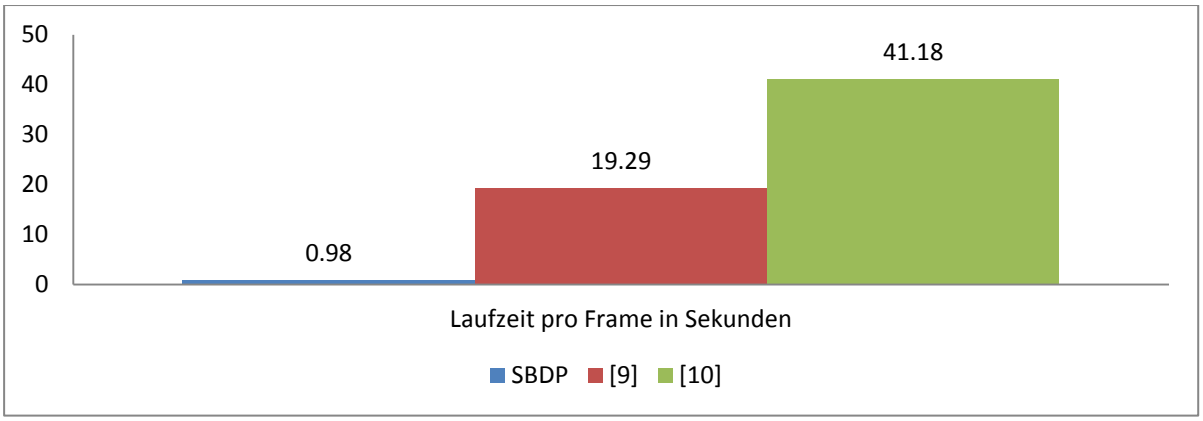


Abbildung 7.10: *Vergleich der durchschnittlichen Laufzeit pro Frame zwischen der segmentierungsbasierten Propagierung von Disparitäten (SBDP), dem Propagierungs-Algorithmus aus* [14] *und dem Stereo Extraktions-Algorithmus aus* [1]. *Für den Vergleich wurde die durchschnittliche Laufzeit pro Frame (in Sekunden) für die beiden Testvideos „Tsukuba 50-66“ und „Tsukuba 380-397“, mit jeweils einer Auflösung von 640 x 480, berechnet.*

a)

b)

c)

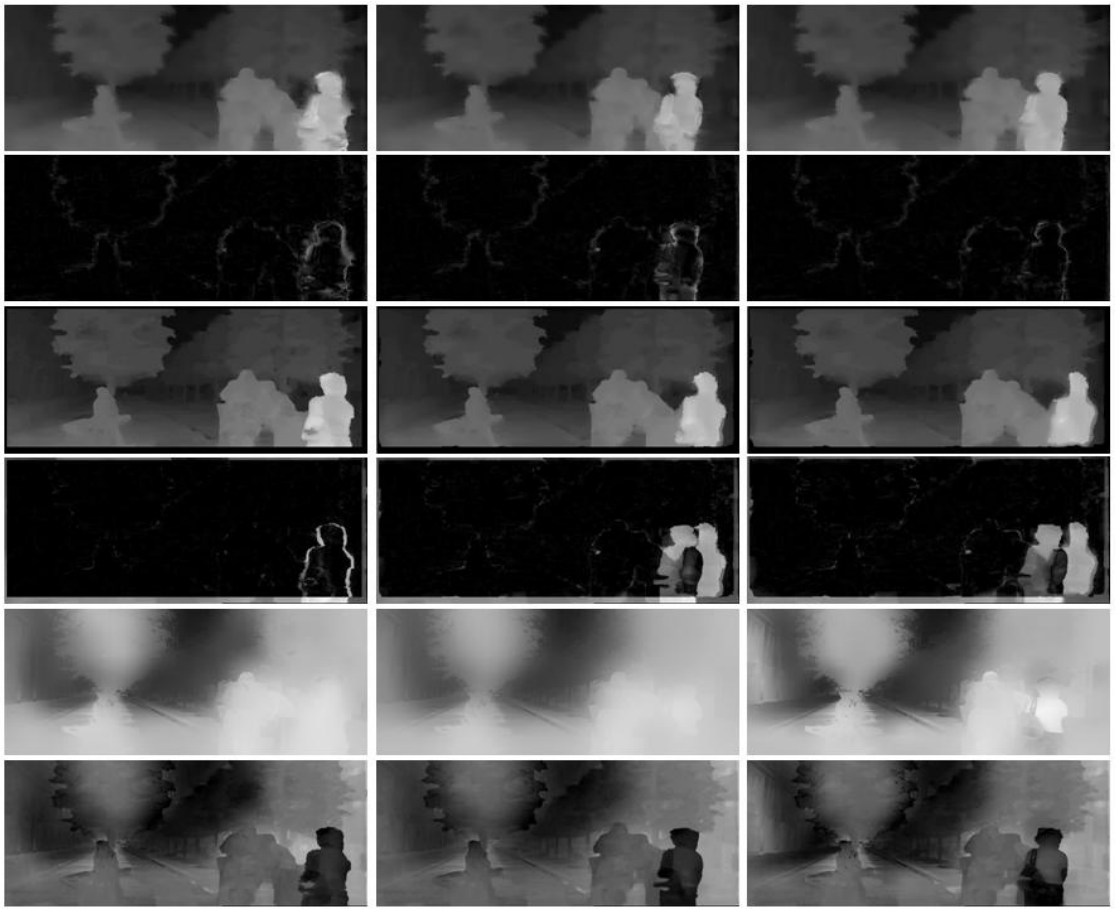
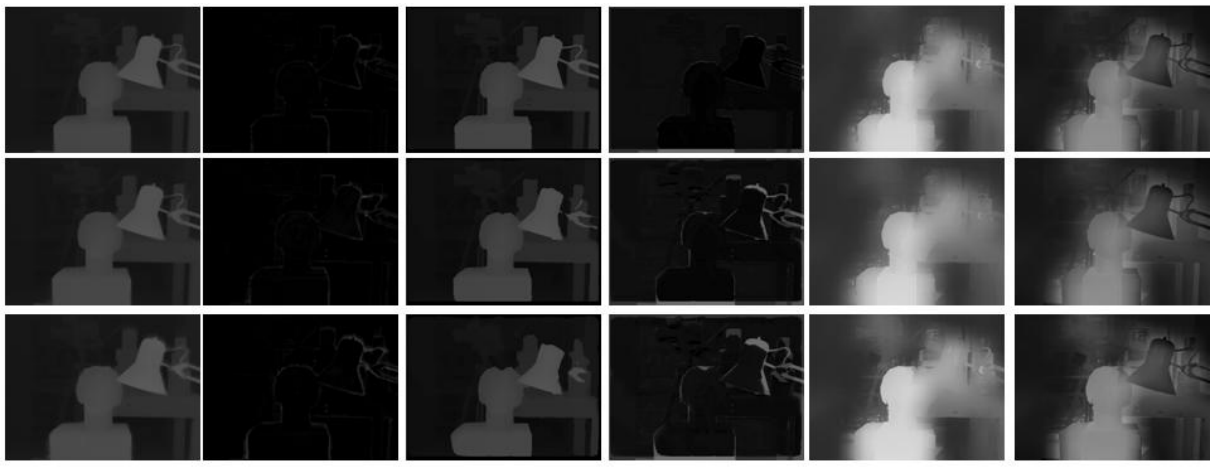


Abbildung 7.11: *Ergebnisse des Testvideos „Stadt“. Von links nach rechts: Frame 1, Frame 9, Frame 18. Die erste Zeile enthält jeweils die resultierende Disparitätskarte, die zweite Zeile den Fehler zur Referenzlösung. a) Ergebnis der segmentierungsbasierten Propagierung (MSE* x *100 = 0.08). b) Ergebnis des Algorithmus aus* [14] *(MSE* x *100 = 1.92). c) Ergebnis des Algorithmus aus* [1] *(MSE* x *100 = 12.15).*



a) b) c)

Abbildung 7.12: *Ergebnisse des Testvideos „Tsukuba 50-66“. Von oben nach unten: Frame 1, Frame 9, Frame 17. Die erste Spalte enthält jeweils die resultierende Disparitätskarte, die zweite Spalte den Fehler zur Referenzlösung. a) Ergebnis der segmentierungsbasierten Propagierung (MSE* x *100 = 0.02). b) Ergebnis des Algorithmus aus* [14] *(MSE* x *100 = 0.34). c) Ergebnis des Algorithmus aus* [1] *(MSE* x *100 = 7.24).*

8. Zusammenfassung

In dieser Diplomarbeit wird eine optimierte Version des Algorithmus aus [22] zur semi-automatischen 2D-zu-3D-Konvertierung vorgestellt. Das Hauptziel liegt dabei in der Verringerung der Laufzeit, bei unveränderter Qualität der Ergebnisse. Qualitätsmerkmale sind dabei beispielsweise zeitliche Kohärenz, kontrastreiche Tiefenübergänge an Objekträndern und glatte räumliche und zeitliche Tiefenänderungen. Dieses Ziel wird durch die Verlagerung von besonders rechenintensiven Programmteilen auf die Grafikkarte erreicht. Die Grafikkartenprogrammierung erfolgt dabei unter Verwendung von NVIDIA CUDA [24].

Zu Beginn klärt diese Diplomarbeit Grundlagen der CUDA-Architektur und gibt eine Übersicht über verwandte 2D-zu-3D-Konvertierungsmethoden und ihre Strategien zur Reduzierung der Laufzeit. Die wichtigsten Aspekte der CUDA-Architektur werden anschließend anhand einer effizienten parallelen GPU-Implementierung des Box-Filters vertieft. Die optimierte Implementierung des Box-Filters wird zudem für die Filterung von Videos erweitert. Die darauf folgenden Kapitel diskutieren die Funktionsweise und die in dieser Diplomarbeit entwickelte Optimierung der Propagierungsschritte aus [22]: 1. segmentierungsbasierte Tiefenpropagierung und 2. filterbasierende Verfeinerung der Tiefenkarten.

Die Optimierung der segmentierungsbasierten Propagierung [22] erfolgt hauptsächlich durch die Parallelisierung des zu Grunde liegenden Segmentierungs-Algorithmus aus [25]. Als zusätzliches Ergebnis wird in dieser Diplomarbeit eine optimierte Version des Algorithmus aus [25] generiert. Des Weiteren wird ein Lösungsansatz für das Problem des limitierten Grafikkartenspeichers vorgestellt. Längere Videos werden zu diesem Zweck in Subsequenzen aufgeteilt, welche nacheinander auf der GPU bearbeitet werden. Im Zuge der Evaluierung wird gezeigt, dass durch die Parallelisierung eine wesentliche Laufzeitverringerung erreicht werden kann, im Vergleich zur ursprünglichen Implementierung aus [22]: Bei einem Video mit 20 Frames und einer Auflösung von 600 x 255 verringert sich die Laufzeit um den Faktor 45.24.

Die Optimierung der filterbasierten Verfeinerung der Tiefenkarten aus [22] erfolgt hauptsächlich durch die Parallelisierung des Guided Filters [23]. Nach einer detaillierten Einführung in die Grundlagen des Guided Filters wird gezeigt, wie dieser durch die Verwendung der optimierten GPU-Implementierung des Box-Filters beschleunigt werden kann. Zu diesem Zweck wird die vorhandene GPU-Implementierung aus [26] zur Filterung von Videos erweitert. Im Anschluss daran wird eine optimierte GPU-Implementierung eines regionsweisen Guided Filters vorgestellt. Laufzeitvergleiche zwischen optimiertem und nicht-optimiertem Filter zeigen eine signifikante Verringerung der Laufzeit: Im Vergleich zur seriellen MATLAB-Implementierung läuft die parallele GPU-Implementierung um den Faktor 906.6 schneller. Das Problem des limitierten Grafikkartenspeichers wird auch hier durch eine Aufteilung von langen Videos in Subsequenzen gelöst.

Das Ergebnis dieser Diplomarbeit ist eine effiziente Implementierung des in [22] vorgestellten Algorithmus. Evaluierungen haben gezeigt, dass die optimierte Version des Algorithmus aus [22] Disparitätskarten mit unverändert hoher Qualität generiert und gleichzeitig eine wesentliche Verringerung der benötigten Laufzeit erreicht. Für ein Video mit der Auflösung 640 x 480 werden durchschnittlich 0.98 Sekunden pro Frame benötigt. Diese Evaluierungen beinhalten auch einen Vergleich mit den beiden verwandten Algorithmen [14] und [1]. Die verwandten Algorithmen können in Bezug auf Qualität der Ergebnisse und Laufzeit von der optimierten Implementierung dieser Diplomarbeit deutlich übertroffen werden. Eine Evaluierung des Lösungsansatzes zur Bearbeitung von

längeren Videos hat gezeigt, dass die Aufteilung eines Videos zu Fehlern führen kann, welche mit der Anzahl der Subsequenzen zunehmen. Zudem stellt die Wahl der Frames, an denen ein Video aufgeteilt wird, einen wichtigen Faktor dar, da die Fehler im Allgemeinen mit zunehmender Entfernung zu den ursprünglichen Schlüssel-Frames gravierender werden. Dieser Faktor wird im Lösungsansatz dieser Diplomarbeit nicht berücksichtigt. An dieser Stelle besteht somit Verbesserungspotential für zukünftige Arbeiten.

Als zusätzlicher Beitrag können die aus dieser Diplomarbeit resultierenden, optimierten Algorithmen zur Segmentierung und kantenerhaltenden Filterung auch unabhängig von dem vorgestellten 2D-zu-3D-Konvertierungsalgorithmus in anderen Applikationen eingesetzt werden. Sowohl die Filterung als auch die Segmentierung stellen fundamentale Methoden im Bereich *Computer Vision* dar, welche oft als Basis für andere Algorithmen verwendet werden. Eine Verwendung der optimierten Implementierungen dieser Algorithmen würde somit auch in anderen Anwendungen zur Verbesserung der Laufzeit führen. Der optimierte Segmentierungs-Algorithmus könnte beispielsweise in segmentierungsbasierten Konvertierungsmethoden [16, 21] Anwendung finden. Die optimierte Version des Guided Filters für Videos könnte zur Laufzeitverringerung von filterbasierten Konvertierungs-Algorithmen [14, 16, 18] eingesetzt werden. Der Guided Filter wird zudem für Anwendungen wie *Feathering/Matting* [65], *Dehazing* [66] oder filterbasierten *Stereo Matching*-Methoden [40, 67] verwendet.

Literaturverzeichnis

[1] M. Guttmann, L. Wolf, und D. Cohen-Or, *Semi-automatic Stereo Extraction from Video Footage*, IEEE International Conference on Computer Vision, S. 136–142, 2009.

[2] C. Fehn und R. S. Pastoor, *Interactive 3-DTV-Concepts and Key Technologies*, Proceedings of the IEEE, Band 94, Ausgabe 3, S. 524–538, 2006.

[3] M. Bleyer und M. Gelautz, *Simple but Effective Tree Structures for Dynamic Programming-based Stereo Matching*, International Conference on Computer Vision Theory and Applications, S. 415–422, 2008.

[4] B. Huhle, S. Fleck, und A. Schilling, *Integrating 3D Time-of-Flight Camera Data and High Resolution Images for 3DTV Applications*, 3DTV Conference, S. 1–4, 2007.

[5] T. H. Kim, H. Jung, K. M. Lee, und S. U. Lee, *Segment-based Foreground Object Disparity Estimation Using Zcam and Multiple-View Stereo*, International Conference on Intelligent Information Hiding and Multimedia Signal Processing, S. 1251–1254, 2008.

[6] J. Smisek, M. Jancosek, und T. Pajdla, *3D with Kinect*, IEEE International Conference on Computer Vision Workshops, S. 1154–1160, 2011.

[7] M. Cieply, *A Movie's Budget Pops from the Screen*, The New York Times, 08. November 2009.

[8] R. Rzeszutek, T. El-Maraghi, und D. Androutsos, *Interactive Rotoscoping Through Scale-space Random Walks*, IEEE International Conference on Multimedia and Expo, S. 1334–1337, 2009.

[9] M. Murphy, P. Jean, und J. Myint, *Inside the 3-D Conversion of 'Titanic'*, The New York Times, 30. März 2012.

[10] L. Liu, C. Ge, N. Zheng, und Q. Li, *Spatio-temporal Adaptive 2D-to-3D Video Conversion for 3DTV*, IEEE International Conference on Consumer Electronics, S. 465–466, 2012.

[11] C.-L. Su, K.-N. Pang, T.-M. Chen, G.-S. Wu, C.-L. Chiang, H.-R. Wen, L.-S. Huang, Y.-H. Hsueh, und S.-Y. Tseng, *A Real-time Full-HD 2D-to-3D Conversion System Using Multicore Technology*, International Conference on Multimedia and Ubiquitous Engineering, S. 273–276, 2011.

[12] H. Wang, L. Zhang, Y. Yang, und B. Liu, *2D-to-3D Conversion based on Depth-from-motion*, International Conference on Mechatronic Science, Electric Engineering and Computer, S. 1892–1895, 2011.

[13] F. Yue, R. Jinchang, und J. Jianmin, *Object-Based 2D-to-3D Video Conversion for Effective Stereoscopic Content Generation in 3D-TV Applications*, IEEE Transactions on Broadcasting, Band 57, Ausgabe 2, S. 500–509, 2011.

[14] C. Varekamp und B. Barenbrug, *Improved Depth Propagation for 2D to 3D Video Conversion using Key-frames*, European Conference on Visual Media Production, S. 1–7, 2007.

[15] O. Wang, M. Lang, M. Frei, A. Hornung, A. Smolic, und M. Gross, *StereoBrush : Interactive 2D to 3D Conversion Using Discontinuous Warps*, Proceedings of the Eighth Eurographics Symposium on Sketch-Based Interfaces and Modeling, S. 47–54, 2011.

[16] X. Cao, Z. Li, und Q. Dai, *Semi-Automatic 2D-to-3D Conversion Using Disparity Propagation*, IEEE Transactions on Broadcasting, Band 57, Ausgabe 2, S. 491–499, 2011.

[17] P. Harman und J. Flack, *Rapid 2D-to-3D Conversion*, Stereoscopic Displays and Virtual Reality Systems IX, S. 78–86, 2002.

[18] M. Lang, O. Wang, T. Aydin, A. Smolic, und M. Gross, *Practical Temporal Consistency for Image-Based Graphics Applications*, ACM Transactions on Graphics, Band 31, Ausgabe 4, S. 34:1–34:8, 2012.

[19] Z. Zhang, C. Zhou, Y. Wang, und W. Gao, *Interactive Stereoscopic Video Conversion*, IEEE Transactions on Circuits and Systems for Video Technology, Band 23, Ausgabe 10, S. 1795–1808, 2013.

[20] R. Phan und D. Androutsos, *Robust Semi-Automatic Depth Map Generation in Unconstrained Images and Video Sequences for 2D to Stereoscopic 3D Conversion*, IEEE Transactions on Multimedia, Band PP, Ausgabe 99, S. 1–30, 2013.

[21] J. Feng, H. Ma, J. Hu, L. Cao, und H. Zhang, *Superpixel Based Depth Propagation for Semi-Automatic 2D-to-3D Video Conversion*, International Conference on Networking and Distributed Computing, S. 157–160, 2012.

[22] N. Brosch, C. Rhemann, und M. Gelautz, *Segmentation-based Depth Propagation in Videos*, ÖAGM Annual Workshop of the Austrian Association for Pattern Recognition, S. 1–8, 2011.

[23] K. He, J. Sun, und X. Tang, *Guided Image Filtering.*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Band 35, Ausgabe 6, S. 1397–409, 2013.

[24] NVIDIA, *CUDA Programming Guide*, 2011. http://www.idt.mdh.se/kurser/dva314/material/GPGPUProgrammingUsingCUDA.pdf. [Aufgerufen am: 04. Juni 2013].

[25] M. Grundmann, V. Kwatra, M. Han, und I. Essa, *Efficient Hierarchical Graph-based Video Segmentation*, IEEE Computer Society Conference on Computer Vision and Pattern Recognition, S. 2141–2148, 2010.

[26] G. Braun, *Effiziente kantenerhaltende Bildglättung und ihre Anwendung in der Praxis*, Diplomarbeit, Technische Universität Wien, Institut für Softwaretechnik und Interaktive Systeme, 2011.

[27] NVIDIA, *NVIDIA GPU Computing SDK*. https://developer.nvidia.com/gpu-computing-sdk. [Aufgerufen am: 12. Juni 2013].

[28] P. F. Felzenszwalb und D. P. Huttenlocher, *Efficient Graph-based Image Segmentation*, International Journal of Computer Vision, Band 59, Ausgabe 2, S. 167–181, 2004.

[29] N. Ahuja, *Real-time O(1) Bilateral Filtering*, IEEE Conference on Computer Vision and Pattern Recognition, Ausgabe 1, S. 557–564, 2009.

[30] A. S. Ogale und Y. Aloimonos, *A Roadmap to the Integration of Early Visual Modules*, International Journal of Computer Vision, Band 72, Ausgabe 1, S. 9–25, 2007.

[31] J. Kopf, M. F. Cohen, D. Lischinski, und M. Uyttendaele, *Joint Bilateral Upsampling*, International Conference and Exhibition on Computer Graphics and Interactive Techniques, 2007.

[32] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, und S. Süsstrunk, *SLIC Superpixels*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Band 34, Ausgabe 11, S. 2274–2282, 2012.

[33] W. Burger und M. J. Burge, *Digitale Bildverarbeitung*, 2. Auflage. Springer Verlag, 2006.

[34] C. Rother, V. Kolmogorov, und A. Blake, *GrabCut-Interactive Foreground Extraction using Iterated Graph Cuts*, ACM Transactions on Graphics, 2004.

[35] C. Li, C. Xu, C. Gui, und M. D. Fox, *Distance Regularized Level Set Evolution and its Application to Image Segmentation.*, IEEE Transactions on Image Processing, Band 19, Ausgabe 12, S. 3243–54, 2010.

[36] Y. Boykov und G. Funka-Lea, *Graph Cuts and Efficient N-D Image Segmentation*, International Journal of Computer Vision, Band 70, Ausgabe 2, S. 109–131, 2006.

[37] Y. Boykov, O. Veksler, und R. Zabih, *Fast Approximate Energy Minimization via Graph Cuts*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Band 23, Ausgabe 11, S. 1222–1239, 2001.

[38] L. Grady, *Random Walks for Image Segmentation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Band 28, Ausgabe 11, S. 1768–1783, 2006.

[39] N. Brosch, A. Hosni, C. Rhemann, und M. Gelautz, *Spatio-temporal Coherent Interactive Video Object Segmentation via Efficient Filtering*, Lecture Notes in Computer Science, Band 7476, S. 418–427, 2012.

[40] A. Hosni, C. Rhemann, M. Bleyer, C. Rother, und M. Gelautz, *Fast Cost-volume Filtering for Visual Correspondence and Beyond*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Band 35, Ausgabe 2, S. 504–11, 2013.

[41] C. Tomasi und R. Manduchi, *Bilateral Filtering for Gray and Color Images*, International Conference on Computer Vision, Band 846, Ausgabe 94305, S. 839–846, 1998.

[42] NVIDIA, *CUDA Best Practices Guide*, 2012. http://www.idt.mdh.se/kurser/dva314/material/GPGPUProgrammingUsingCUDA.pdf. [Aufgerufen am: 16. Juli 2013].

[43] J. Sanders und E. Kondrot, *CUDA by Example*. Addison-Wesley Verlag, 2011.

[44] D. B. Kirk und W. W. Hwu, *In Praise of Programming Massively Parallel Processors : A Hands-on Approach*. Morgan Kaufmann Verlag, 2010.

[45] H. Sutter und J. Larus, *Software and the Concurrency Revolution*, ACM Queue - Multiprocessors, Band 3, Ausgabe 7, S. 54 – 62, 2005.

[46] G. E. Moore, *Moore's Law*, 2013. http://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html. [Aufgerufen am: 16. Juli 2013].

[47] NVIDIA, *CUDA GPUs*, 2013. https://developer.nvidia.com/cuda-gpus. [Aufgerufen am: 04. Juni 2013].

[48] R. Inam, *An Introduction to GPGPU Programming-CUDA Architecture*, 2010. http://www.idt.mdh.se/kurser/dva314/material/GPGPUProgrammingUsingCUDA.pdf. [Aufgerufen am: 04. Juli 2013].

[49] Khronos Group, *The Open Standard for Parallel Programming of Heterogeneous Systems*, 2013. http://www.khronos.org/opencl/. [Aufgerufen am: 04. Juni 2013].

[50] NVIDIA, *NVIDIA CUDA Architecture*, 2009. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf. [Aufgerufen am: 17. Juli 2013].

[51] J. van Oosten, *CUDA Memory Model*. http://3dgep.com/?p=2012. [Aufgerufen am: 12. Juli 2013].

[52] NVIDIA, *CUDA Libraries*. https://developer.nvidia.com/gpu-accelerated-libraries. [Aufgerufen am: 17. Juli 2008].

[53] J. H. McClellan, R. W. Schafer, und M. A. Yoder, *Signal Processing First*. Pearson Verlag, 2003.

[54] G. Ruetsch und P. Micikevicius, *Optimizing Matrix Transpose in CUDA*, 2009. http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf. [Aufgerufen am: 15. Juli 2013].

[55] M. Drmota, G. Gittenberger, G. Karigl, und A. Panholzer, *Mathematik für Informatik*, Berliner S. Heldermann Verlag, 2007.

[56] J.-P. Homann, *Digitales Colormanagement*, 3. Auflage. Springer Verlag, 2007.

[57] O. Pele und M. Werman, *The Quadratic-chi Histogram Distance Family*, European Conference on Computer Vision, Ausgabe 1, S. 749–762, 2010.

[58] M. Werlberger, W. Trobin, T. Pock, A. Wedel, D. Cremers, und H. Bischof, *Anisotropic Huber-L1 Optical Flow*, Proceedings of the British Machine Vision Conference, S. 108.1–108.11, 2009.

[59] H. R. Schwarz und N. Köckler, *Numerische Mathematik*, 8. Auflage. Springer Verlag, 2011.

[60] D. Lowe, *Distinctive Image Features from Scale-invariant Keypoints*, International Journal of Computer Vision, Band 60, Ausgabe 2, S. 91–110, 2004.

[61] OpenCV, *Feature2d Framework*. http://docs.opencv.org/modules/features2d/doc/features2d.html. [Aufgerufen am: 03. Oktober 2013].

[62] OpenCV, *Open Source Computer Vision*. http://opencv.org/. [Aufgerufen am: 03. Oktober 2013].

[63] H.-J. Mittag, *Statistik. Eine interaktive Einführung*, 2. Auflage. Springer Verlag, 2012.

[64] J. O. Rawlings und D. A. Dickey, *Applied Regression Analysis : A Research Tool*, 2. Auflage. Springer Verlag, 1998.

[65] K. He, C. Rhemann, C. Rother, X. Tang, und J. Sun, *A Global Sampling Method for Alpha Matting*, Conference on Computer Vision and Pattern Recognition, S. 2049–2056, 2011.

[66] K. He, J. Sun, und X. Tang, *Single Image Haze Removal Using Dark Channel Prior*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Band 33, Ausgabe 12, S. 2341–2353, 2010.

[67] L. De-Maeztu, S. Mattoccia, A. Villanueva, und R. Cabeza, *Linear Stereo Matching*, International Conference on Computer Vision, Ausgabe 1, S. 1708–1715, 2011.

[68] Broadband Network and Digital Media Lab, *Test Sequences*. http://media.au.tsinghua.edu.cn/2Dto3D/testsequence.html. [Aufgerufen am: 22. Oktober 2013].

[69] S. Martull, M. P. Martorell, und K. Fukui, *Realistic CG Stereo Image Dataset with Ground Truth Disparity Maps*, International Conference on Pattern Recognition, 2012.

[70] M. Peris, *Towards a Simulation Driven Stereo Vision System*, International Conference on Pattern Recognition, S. 1038–1042, 2012.