

Effiziente kantenerhaltende Glättung und ihre Anwendung in der Praxis

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Medieninformatik

eingereicht von

Georg Braun

Matrikelnummer 0227225

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Dr. Margrit Gelautz
Mitwirkung: Dipl.-Ing. Dr. Michael Bleyer, Dr. Christoph Rhemann

Wien, 10.09.2011

(Unterschrift Verfasser/in)

(Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Georg Braun
Karl Millöckergasse 11
2542 Kottlingbrunn

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.

Wien, 10.09.2011

(Unterschrift Verfasser/in)

Kurzfassung

Ziel einer Glättung ist die Reduktion von Bildrauschen unter Beibehaltung der Kantenschärfe. Traditionelle Glättungsfiler, wie beispielsweise der Gauß-Filter, reduzieren Bildrauschen, aber auch Kantenschärfe und sind daher für diese Aufgabe nicht gut geeignet. Im ersten Teil dieser Diplomarbeit werden drei kantenerhaltende Glättungsfiler behandelt, welche sowohl Rauschanteile unterdrücken als auch Kanteninformation weitgehend erhalten. Weiterhin wird ausführlich darauf eingegangen, wie traditionelle- und kantenerhaltende Glättungsfiler effizient implementiert werden können. Bis auf eine Ausnahme wird für alle behandelten Filter ein Algorithmus mit konstanter Laufzeit bezüglich der Kernelgröße präsentiert. In einem eigenen Kapitel werden Strategien zur parallelen Implementation der vorgestellten Glättungsfiler auf CUDA-fähigen Grafikkarten des Herstellers NVIDIA präsentiert.

Der zweite Teil der Arbeit zeigt, dass der Anwendungsbereich kantenerhaltender Glättungsfiler keineswegs auf die Entfernung von Bildrauschen beschränkt ist. Es werden zwei interessante Applikationsszenarien vorgestellt, und es wird diskutiert, inwieweit die zuvor vorgestellten kantenerhaltenden Filter zur Lösung der jeweiligen Aufgabenstellung eingesetzt werden können. Als erstes wird das Szenario der interaktiven Bildsegmentierung behandelt. Hier gilt es den Vordergrundbereich eines Bildes mit Hilfe minimaler Benutzereingaben zu extrahieren. Es wird gezeigt, dass Segmentierungsalgorithmen mit integrierter kantenerhaltender Filterung im Vergleich zu State-of-the-Art-Algorithmen (Grab-Cut) nicht nur qualitativ ähnliche Ergebnisse erzielen können, sondern auch deutlich weniger Rechenleistung beanspruchen.

Das andere Anwendungsszenario beschäftigt sich mit der Nachbearbeitung von Tiefenkarten, die durch Stereo-Matching berechnet wurden. Es wird gezeigt, dass durch die Anwendung kantenerhaltender Filter die Qualität der Karten, vor allem in der Nähe von Tiefenunstetigkeiten, deutlich gesteigert werden kann. Weiters wird demonstriert, wie kantenerhaltende Filter bei der Rückskalierung von Tiefenkarten, die aus Geschwindigkeitsgründen von einem verkleinerten Kamerabildpaar berechnet wurden, genutzt werden können. Der vorgestellte Algorithmus skaliert zunächst die verkleinerte Tiefenkarte mit Hilfe eines herkömmlichen Verfahrens (z.B. Nearest Neighbour Upsampling) auf die Größe des ursprünglichen Kamerabildes. Anschließend wird die vergrößerte Tiefenkarte mit einem kantenerhaltendem Filter geglättet, um die Kantenkonvergenz zwischen ursprünglichem Kamerabild und skaliertes Tiefenkarte zu verbessern.

Abstract

The goal of smoothing is to reduce image noise while preserving edges. Traditional smoothing operators such as the well-known gaussian filter reduce noise but also edge sharpness. Hence such filters are not suitable for the task at hand. In the first part of this master thesis three edge-preserving smoothing operators are presented. These filters have the ability to suppress noise while keeping edge information intact. Furthermore we extensively explore way how both traditional and edge-preserving smoothers can be implemented in an efficient way. For all smoothing operators (with one exception) an algorithm with constant runtime regarding the kernel size is presented. In a separate chapter the parallel implementation of smoothing operators on NVIDIA's CUDA-enabled graphics adapters is discussed.

The second part of this thesis we show that the applications of edge-preserving smoothing filters extend beyond the task of noise suppression. Two interesting application scenarios are presented and discussed in detail. First we take a close look at the scenario of interactive image segmentation. Here the goal is to extract an image's foreground region by utilization of minimal user input. We present an algorithm using an edge-preserving smoothing filter, which can not only achieve results of similar quality to Grab-Cut, but is considerably less computationally demanding.

The other scenario deals with post processing of depth maps calculated by stereo matching. We show that by the application of an edge-preserving smoothing filter the quality of such maps can be increased significantly, especially near depth discontinuities. Furthermore, we explore if and how edge preserving filters can be used to scale depth maps that have been calculated from a downscaled pair of camera images for performance reasons back to the size of the original camera image. For this purpose a simple two-step algorithm is introduced. First, the downscaled depth map is upsampled to the size of the original camera image by means of a traditional upscaling algorithm such as Nearest-Neighbour. Second, the scaled depth map will be smoothed by application of an edge-preserving filter. We show that the smoothing operation ensures better convergence of edges between original camera image and upscaled depth map.

Inhaltsverzeichnis

1. Einleitung	1
2. Glättungsfilter	4
2.1 Grundlagen.....	4
2.1.1 Glättung.....	4
2.1.2 Lineare- und nichtlineare Filter	5
2.2 Traditionelle Glättungsfilter	6
2.2.1 Mittelwertfilter	6
2.2.2 Gauß-Filter	8
2.2.3 Median-Filter	11
2.3 Kantenerhaltende Glättungsfilter	13
2.3.1 Bilateral Filter	13
2.3.1.1 Joint Bilateral Filter	14
2.3.2 Bilateral Median-Filter	15
2.3.3 Guided Image Filter	17
2.4 Glättung von Farbbildern	20
3. Sequentielle Implementierung	23
3.1 Mittelwertfilter	23
3.2 Gauß-Filter	27
3.3 Median-Filter	30
3.4 Bilateral-Filter	38
3.5 Bilateral Median-Filter	44
3.6 Guided Image Filter.....	45
4. Parallele Implementierung	48
4.1 CUDA-Architektur.....	48
4.1.1 Motivation	48
4.1.2 C for CUDA vs. OpenCL	49
4.1.3 CUDA-Architektur.....	50
4.2 Mittelwertfilter	57
4.3 Andere Filter.....	64
5. Interaktive Bildsegmentierung	66
5.1 Motivation	66
5.2 Algorithmus	66
5.3 Evaluierung	69
5.4 Paint Selection	81
5.5 Alpha-Matting	84
6. Optimierung von Tiefenkarten in Disparitätsvideos	85
6.1 Motivation	86
6.2 Kantenerhaltende Glättung von Tiefenkarten.....	87
6.3 Upsampling von Tiefenkarten	94
7. Zusammenfassung	100
Literaturverzeichnis	103



Abb. 1:
(a) 7x7 Median Filter
(b) 11x11 Bilateral Filter

1. Einleitung

Traditionell wird Bildrauschen durch Glättung mit Mittelwert-, Gauß- oder Median Filter reduziert. Da diese Filter jedoch auch die Kantenschärfe des Bildes beeinträchtigen sind sie nur bedingt für diese Aufgabe geeignet. Kantenerhaltende Glättungsfiler, wie der in *Abb. 1* gezeigte Bilateral Filter, versuchen hingegen Rauschanteile zu minimieren ohne die Kantenschärfe des ursprünglichen Bildes signifikant zu reduzieren. In dieser Diplomarbeit werden mit dem Bilateral Filter [1], Bilateral Median Filter [2] und Guided Image Filter [3] drei kantenerhaltende Filter präsentiert. In Kapitel 2 werden zunächst die drei genannten traditionellen Glättungsfiler vorgestellt und untersucht weshalb sie Kanteninformationen verfälschen. Aufbauend darauf werden die drei kantenerhaltenden Filter definiert. Weiters wird gezeigt welche qualitativen Unterschiede zwischen allen genannten Filtern bei der Entfernung von Bildrauschen bestehen.

Die Qualität des berechneten Ergebnisses ist eine wichtige, jedoch nicht die einzige entscheidende Eigenschaft eines Filters. In den meisten Anwendungen ist es von Bedeutung, dass Ergebnisse innerhalb einer vorgegebenen Zeitspanne berechnet werden können. In Kapitel 3 wird daher gezeigt, wie traditionelle und kantenerhaltende Filter effizient implementiert werden können. Für jeden Filter, abgesehen vom Guided Image Filter, welcher bereits per Definition eine konstante Laufzeit besitzt, wird zunächst ein naiver Algorithmus mit quadratischer Laufzeit bezüglich der Kernelgröße präsentiert. Dieser erste Algorithmus wird anschließend schrittweise optimiert und die Laufzeiten von naiver und optimierter

Version in einem Diagramm gegenübergestellt. Für alle Filter, abgesehen vom Bilateral Median Filter, wird ein effizienter Algorithmus mit konstanter Laufzeit bezüglich der Kernelgröße präsentiert. Einige der optimierten Algorithmen sind approximative Verfahren. In diesen Fällen wird ebenfalls gezeigt inwieweit sich das berechnete Ergebnis von einer exakten Implementierung unterscheidet. Abgesehen von den $O(1)$ -Approximationen des Gauß- und Bilateral Filters wurden alle vorgestellten Algorithmen im Rahmen dieser Diplomarbeit implementiert.

In den letzten Jahren wurde erkannt, dass die enorme Rechenleistung moderner Grafikkarten nicht nur zum Rendering dreidimensionaler Szenen genutzt werden kann. Führende Grafikkartenhersteller begannen Architekturen zu definieren, die es ermöglichen beliebige Anwendungen auf die Grafikkarte zu verlagern. In Kapitel 4 wird die CUDA-Architektur [4] des Grafikkartenherstellers NVIDIA vorgestellt. CUDA-Karten sind massiv parallele Systeme und es ist daher von größter Wichtigkeit jeden Algorithmus vor der CUDA-Implementation zu parallelisieren. Am Beispiel des Mittelwertfilters wird die Parallelisierung eines konkreten Algorithmus demonstriert. Dabei wird besonders auf die speziellen Eigenschaften und Einschränkungen der CUDA-Architektur Rücksicht genommen, denn nicht jeder parallele Algorithmus läuft effizient auf der Grafikkarte. Im Rahmen dieser Diplomarbeit wurden auch Bilateral Filter und Guided Image Filter in CUDA implementiert, sowie festgestellt welche Laufzeitunterschiede im Vergleich zu den CPU-Implementierungen bestehen.

Die Anwendungsmöglichkeiten kantenerhaltender Filter beschränken sind nicht auf die Minimierung von Bildrauschen. In den Kapiteln 5 (Interaktive Bildsegmentierung) und 6 (Optimierung von Tiefenkarten in Disparitätsvideos) werden zwei weitere interessante Anwendungen vorgestellt. Bei der interaktiven Bildsegmentierung wird ein zu segmentierendes Objekt vom Benutzer grob markiert. Es wird ein Algorithmus vorgestellt der durch kantenerhaltende Filterung alle Pixel ermittelt die Teil des markierten Objektes sind. Kapitel 6 demonstriert, dass Tiefenkarten stereoskopischer Aufnahmen durch kantenerhaltende Glättung optimiert werden können. Es wird gezeigt, dass ein konstantes (Filter)Parameterset ausreicht, um die Mehrheit der Tiefenkarten signifikant zu verbessern. Weiters wird untersucht ob einer der drei kantenerhaltenden Filter besonders für diese Aufgabe geeignet ist. Auch die in Kapitel 5 und 6 vorgestellten Algorithmen wurden im Rahmen dieser Diplomarbeit implementiert.

Kapitel 7 fasst schließlich die wesentlichsten Erkenntnisse der vorhergehenden Abschnitte zusammen und bietet einen kurzen Einblick in weitere interessante Anwendungsgebiete der kantenerhaltenden Filterung, die in der vorliegenden Diplomarbeit aus Platzgründen nicht ausführlich behandelt werden können.



Abb. 2: Minimierung von Bildrauschen durch Unterdrückung hoher Frequenzen

2. Glättungsfilter

In diesem Kapitel wird der Begriff Glättung definiert und untersucht welche Filter besonders für diese Aufgabe geeignet sind.

2.1 Grundlagen

2.1.1 Glättung

Der Begriff Glättung ist in verschiedenen Teilbereichen der Wissenschaft unterschiedlich definiert [5]: In der Mathematik versteht man unter Glättung beispielsweise die Überführung einer Kurve in eine Kurve mit geringerer Krümmung, die möglichst wenig vom Original abweicht. Auch außerhalb der Wissenschaft wird der Begriff Glättung verwendet: Fotografen verstehen darunter beispielsweise Effekte, die gezielt zur unscharfen Abbildung eingesetzt werden. Im Kontext dieser Diplomarbeit ist der Begriff Glättung wie folgt definiert: Entfernung von Bildrauschen unter möglichst geringer Veränderung bestehender Bildstrukturen.

Bildrauschen besteht vorwiegend aus hohen Frequenzen im Ortsfrequenzbereich. Abb. 2 zeigt, dass die Unterdrückung hoher Frequenzen zu einer Minimierung des Bildrauschens führt.

Dieser naive Ansatz führt allerdings nicht zum gewünschten Ergebnis, da auch Kanten weich gezeichnet werden. Hohe Frequenzen enthalten also nicht nur Bildrauschen sondern auch Information. Ein Ziel der Glättung im Sinne dieser Diplomarbeit ist die möglichst geringere Veränderung bestehender Strukturen. Durch einfache Unterdrückung hoher Frequenzen kann dieses Ziel nicht erreicht werden. Traditionelle Glättungsfilter, wie sie in Abschnitt 2.2 vorgestellt werden, approximieren den idealen Tiefpassfilter und reduzieren daher gleichermaßen Bildrauschen wie Kantenschärfe. Kantenerhaltende Glättungsfilter werden in Abschnitt 2.3 vorgestellt.

2.1.2 Lineare- und nichtlineare Filter

Einige Aufgabenstellungen der Bildverarbeitung, wie die Gamma-Korrektur, können mit Punktoperationen gelöst werden. Punktoperationen sind dadurch gekennzeichnet, dass jeder neue Pixelwert aus dem ursprünglichen Pixelwert an derselben Position berechnet wird. Viele Bildverarbeitungsprobleme können jedoch nicht auf diese Weise gelöst werden. Ein Tiefpassfilter kann beispielsweise nicht als Punktoperation implementiert werden, da sich aus einem isolierten Punkt keine Frequenzinformation bestimmen lässt. Filter unterscheiden sich von Punktoperationen dadurch, dass auch Pixelwerte in der Umgebung in die Berechnung einbezogen werden, indem ein Fenster über dem zu berechnenden Pixel zentriert wird. Alle Punkte innerhalb dieses Fensters fließen in die Berechnung ein. Durch Einbeziehung der Umgebung ist es beispielsweise möglich Aussagen über die Frequenz im Ortsfrequenzbereich zu treffen. In dieser Diplomarbeit werden, wenn nicht anders erwähnt, immer Filter mit symmetrischen Fenstergrößen ungerader Dimension betrachtet.

Filter werden in zwei Gruppen unterteilt [6]: Lineare- und nichtlineare Filter. Lineare Filter berechnen die gewichtete Summe aller Pixelwerte innerhalb des Filterfensters. Dies entspricht einer linearen Verknüpfung der entsprechenden Werte. In der Praxis werden lineare Filter oft mit einer aus der Mathematik bekannten Operation berechnet, der linearen Faltung:

$$I'(x, y) = \sum_{i=-\frac{p}{2}}^{\frac{p}{2}} \sum_{j=-\frac{q}{2}}^{\frac{q}{2}} I(x+i, y+j) \cdot H\left(i + \frac{p}{2}, j + \frac{q}{2}\right) \quad (1)$$

I ist das ursprüngliche Bild (Eingabebild), I' das gefilterte Bild (Ausgabebild). x und y definieren die Koordinaten des zu berechnenden Pixels. Die zweidimensionale Matrix H bestimmt die Gewichtung der Pixel innerhalb des Filterfensters. Die Parameter p und q definieren die Dimension von H und somit auch die Größe des Filterfensters. Mittelwertfilter und Gauß-Filter, welche beide in Abschnitt 2.2 vorgestellt werden, sind lineare Filter.

Alle Filter, die Pixelwerte nicht linear miteinander verknüpfen, werden nichtlineare Filter genannt. Sie können nicht durch lineare Faltung berechnet werden. Zur Familie der nichtlinearen Filter zählen beispielsweise sogenannte Rangordnungsfiler, wie der in Abschnitt 2.2 vorgestellte Medianfilter. Ein Rangordnungsfiler sortiert alle Pixelwerte innerhalb des Filterfensters entsprechend ihrer Intensität. Anschließend wird aus der sortierten Gruppe ein Pixelwert gewählt. Ein Minimumfilter wählt beispielsweise den niedrigsten Wert, ein Maximumfilter den höchsten Wert. Der gewählte Wert ist der ausgegebene Pixelwert.

$$H = \begin{bmatrix} \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \\ \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} & \frac{1}{25} \end{bmatrix}$$

Abb. 3: Filterkern 5x5 Mittelwerter

Da die Möglichkeiten linearer Filter begrenzt sind, können viele Aufgabenstellungen nur durch den Einsatz nichtlinearer Filter gelöst werden. Insbesondere sind die in Abschnitt 2.3 vorgestellten kantenerhaltenden Glättungsfilter allesamt nichtlinearer Natur.

2.2 Traditionelle Glättungsfilter

2.2.1 Mittelwertfilter

Der ideale Tiefpassfilter kann durch den Mittelwertfilter approximiert werden. Der Mittelwertfilter ist ein linearer Filter, bei dem alle Elemente des Filterkerns H auf einen konstanten Wert gesetzt werden, nämlich $1 \div (p \cdot q)$. p und q stellen die Dimension des Filterkerns dar. *Abb. 3* zeigt den Filterkern eines 5x5 Mittelwertfilters.

Abb. 4 (nächste Seite) zeigt ein mit Mittelwertfilter geglättetes Bild. Wie erwartet wurden durch die Filterung sowohl Rauschanteile reduziert als auch Kanten weichgezeichnet.

Der Mittelwertfilter besitzt ein ungünstiges Frequenzverhalten [6], da höhere Frequenzen oft weniger stark gedämpft werden als niedrigere. *Abb. 5* (nächste Seite) zeigt das Frequenzverhalten des Mittelwertfilters bei Glättung von Kosinusschwingungen im Bereich von 1 bis 500Hz. Ein idealer Tiefpassfilter würde höhere Frequenzen immer stärker dämpfen als niedrigere Frequenzen. Die Amplitude der gefilterten Kosinusschwingungen sollte also mit steigender Frequenz stetig abfallen. Beim Mittelwertfilter ist dies nur teilweise der Fall. Beispielsweise wurde eine 150Hz Schwingung stärker gedämpft als eine Schwingung mit 200Hz. Es kann also festgehalten werden, dass der ideale Tiefpassfilter durch den Mittelwertfilter sehr schlecht approximiert wird.



Abb. 4: (a) Verrauschtes Originalbild
(b) Ergebnis nach Anwendung des 11x11 Mittelwertfilters

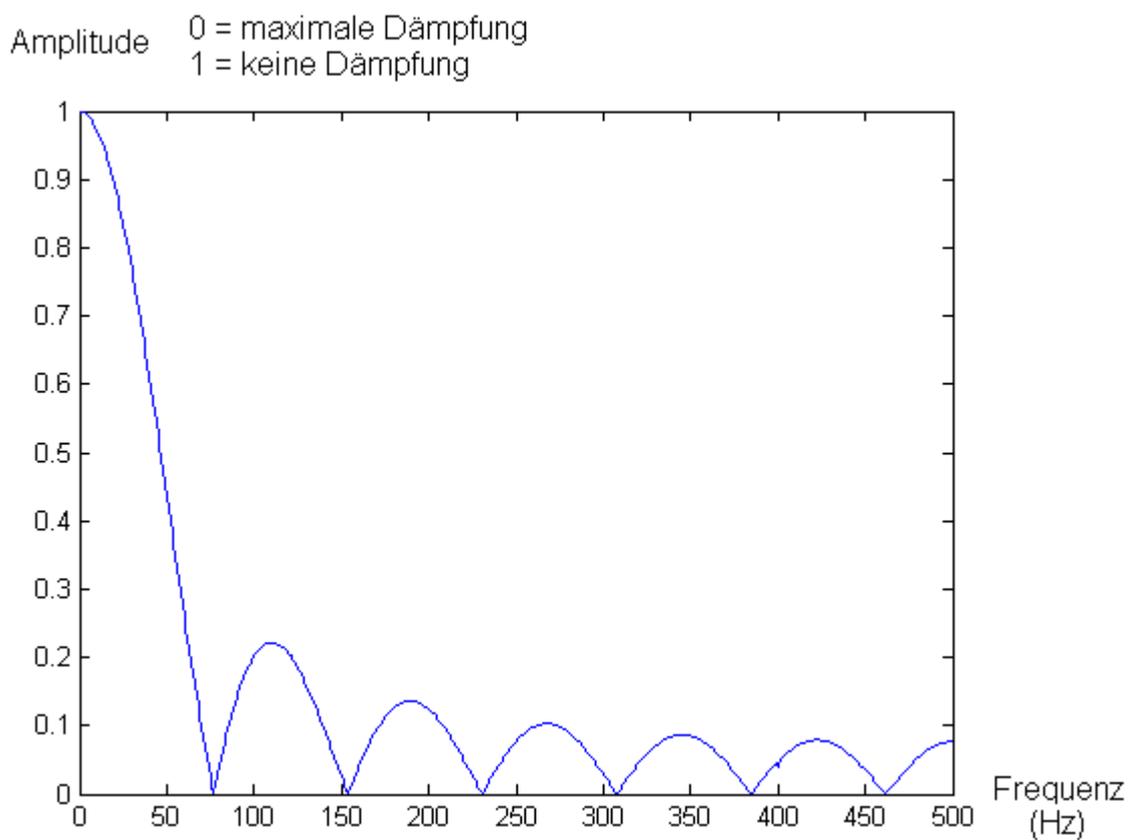


Abb. 5: Frequenzverhalten eines 13x13 Mittelwertfilters, ermittelt durch ein im Rahmen dieser Diplomarbeit erstelltes Programm

Eine weitere unerwünschte Eigenschaft des Mittelwertfilters ist die Anisotropie [6]. Ein Filter ist anisotrop bzw. richtungsabhängig, wenn nicht in alle Richtungen gleich stark gefiltert wird.

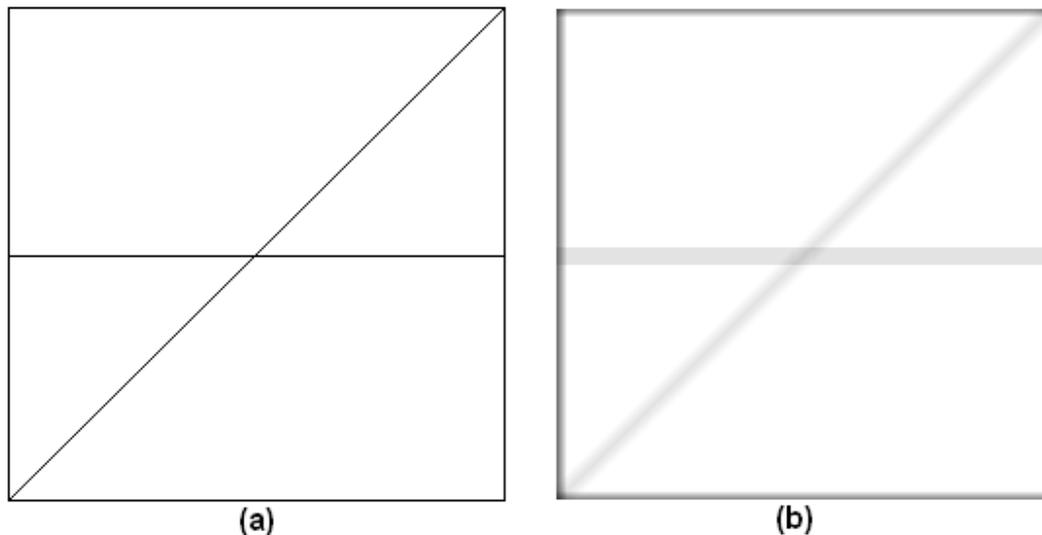


Abb. 6: Anisotropie des Mittelwertfilters,
 (a) Originalbild
 (b) Gefiltertes Bild
 Die horizontale Kante wurde stärker geglättet als die diagonale

Abb. 6 illustriert das anisotrope Verhalten des Mittelwertfilters. Die horizontale Linie wurde stärker weichgezeichnet als die diagonale Linie, obwohl beide im Originalbild gleich stark sind.

2.2.2 Gauß-Filter

Der Gauß-Filter ist eine, im Vergleich zum Mittelwertfilter, verbesserte Approximation des idealen Tiefpassfilters. Die Elemente des Filterkerns werden entsprechend der Gauß-Funktion gewichtet: Je weiter ein Punkt vom Zentrum des Kerns entfernt ist, umso geringer ist seine Gewichtung. *Formel 2* und *3* definieren respektive eindimensionale und zweidimensionale Gauß-Funktion:

$$G_{\sigma} = e^{-\frac{x^2}{2\sigma^2}} \quad (2)$$

$$G_{\sigma} = e^{-\frac{x^2+y^2}{2\sigma^2}} \quad (3)$$

σ ist die Standardabweichung der Gauß-Funktion. Die Standardabweichung legt fest, wie stark die Werte um das Zentrum abfallen. Eine niedrige Standardabweichung bedeutet, dass die Werte schnell abfallen, also entfernte Pixel das Ergebnis nur gering beeinflussen. Bei hoher Standardabweichung fallen die Werte hingegen nur langsam ab. Entfernte Pixel

$$H = \begin{bmatrix} 0.0002 & 0.0033 & 0.0081 & 0.0033 & 0.0002 \\ 0.0033 & 0.0479 & 0.1164 & 0.0479 & 0.0033 \\ 0.0081 & 0.1164 & 0.2832 & 0.1164 & 0.0081 \\ 0.0033 & 0.0479 & 0.1164 & 0.0479 & 0.0033 \\ 0.0002 & 0.0033 & 0.0081 & 0.0033 & 0.0002 \end{bmatrix}$$

Abb. 7: Normalisierter Filterkern des 5x5 Gaußfilters, $\sigma = 0.75$



Abb. 8: (a) Verrauschtes Originalbild
(b) Ergebnis nach Anwendung des 11x11 Gauß-Filters, $\sigma = 2.00$

können daher das Ergebnis signifikant beeinflussen. Im Filterkern werden die Werte der Gauß-Funktion üblicherweise normalisiert, um bei der Glättung den ursprünglichen Wertebereich zu erhalten.

Abb. 7 zeigt den Filterkern eines 5x5 Gauß-Filters. Abb. 8 zeigt ein mit Gauß-Filter gefiltertes Bild. Abb. 9 (nächste Seite) zeigt das Frequenzverhalten des Gauß-Filters. Es ist zu erkennen, dass höhere Frequenzen immer stärker gedämpft werden als niedrigere. Abb. 10 (nächste Seite) illustriert die Isotropie (Richtungsunabhängigkeit) des Gauß-Filters.

Filtergröße und Standardabweichung sollten nicht unabhängig voneinander gewählt werden. Es wird empfohlen beide Werte so zu wählen, dass die Randkoeffizienten des Filterkerns nahe bei 0 liegen, um ein ungünstiges Frequenzverhalten zu vermeiden [6]. Der Gauß-Filter erhält die Kanten etwas besser als der Mittelwertfilter. Die Ergebnisse sind jedoch immer

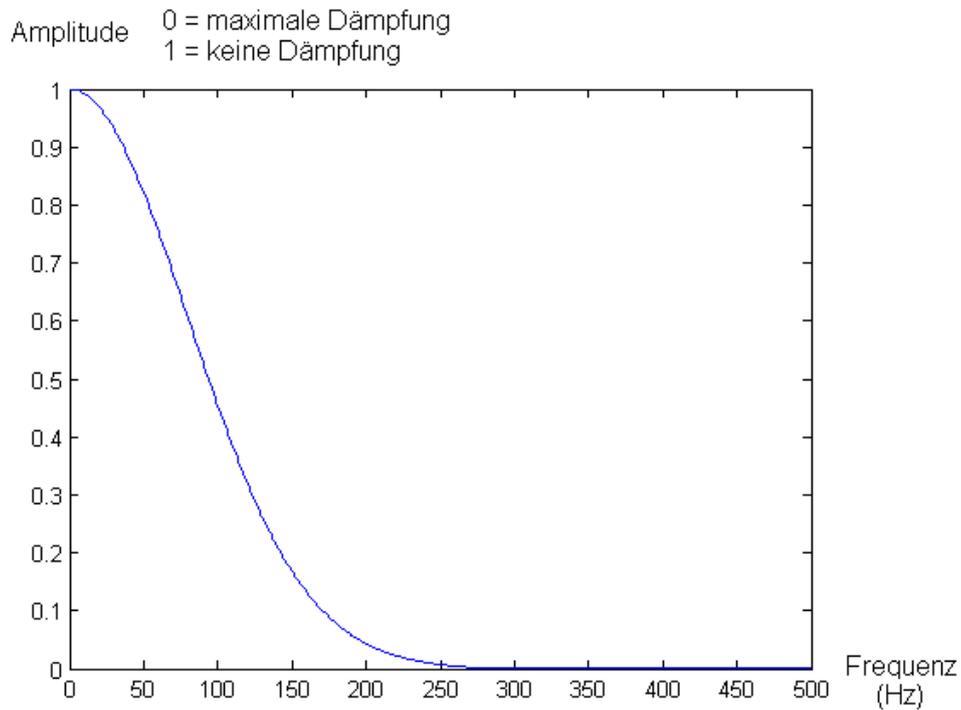


Abb. 9: Frequenzverhalten eines 13x1 Gauß-Filter, $\sigma = 2.00$
Im Vergleich zum Mittelwertfilter fällt die Kurve stetig ab

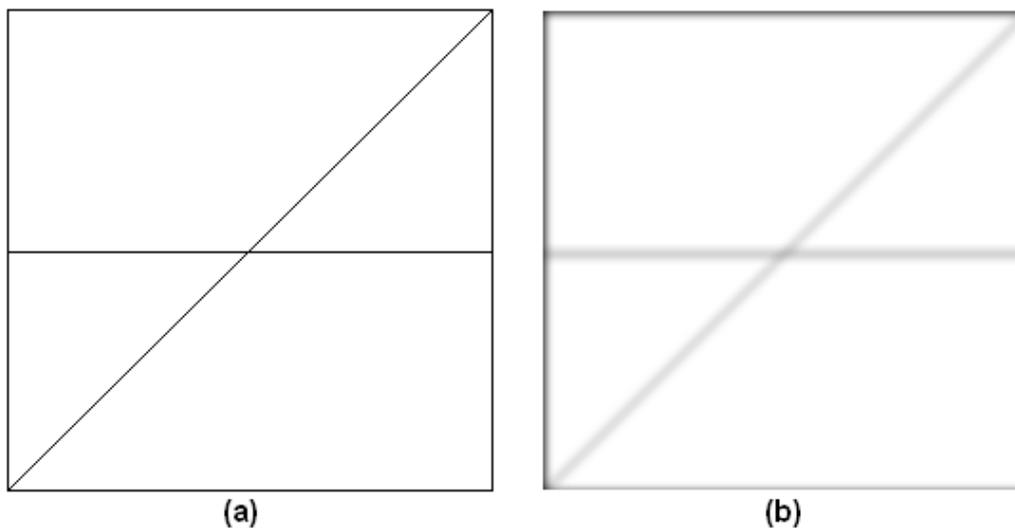


Abb. 10: Isotropie des Gauß-filters
(a) Originalbild
(b) Gefiltertes Bild
horizontale und diagonale Kante wurden gleich stark geglättet

noch nicht zufriedenstellend. Dies liegt daran, dass jeder Tiefpassfilter zwangsläufig Kanten weichzeichnet, da sich diese Information in den hohen Frequenzen befindet.

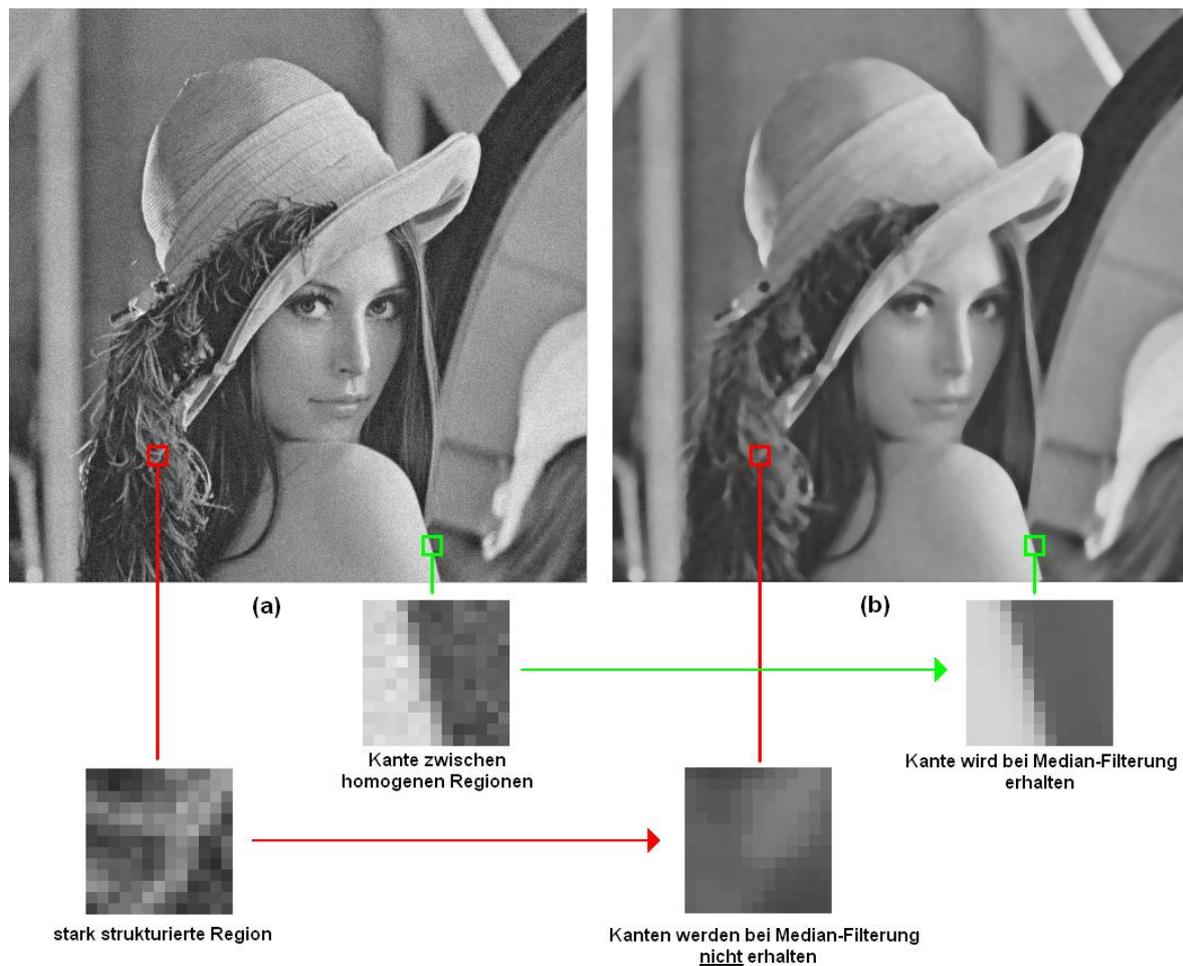


Abb. 11: Kantenerhaltende Funktion des Median-Filters
 (a) Verrauschtes Originalbild
 (b) Ergebnis nach Anwendung des 7x7 Median-Filters

2.2.3 Median-Filer

Beim Median-Filer wird der Wert des gefilterten Pixels auf den Median der Pixel des Filterfensters gesetzt. Er gehört zur Familie der nichtlinearen Rangordnungsfiler, da zur Berechnung des Medians alle Pixel des Filterfensters sortiert werden und anschließend das mittlere Element gewählt wird. Formal ist der Median-Filer wie folgt definiert [6]:

$$I'(x, y) = \text{med}(R_{x,y}) \quad (4)$$

$R_{x,y}$ bezeichnet die Menge aller Helligkeitswerte zentriert auf Pixel (x,y) .

Abb. 11 zeigt ein mit Median-Filer gefiltertes Bild.



Abb. 12: (a) Verrauschtes Originalbild
(b) Ergebnis nach Anwendung des 7x7 Mittelwertfilters
(c) Ergebnis nach Anwendung des 11x11 Gauß-Filters, $\sigma = 2.00$
(d) Ergebnis nach Anwendung des 7x7 Median-Filters

Der Median-Filter besitzt eine eingeschränkte kantenerhaltende Funktion. Treffen größere homogene Regionen aufeinander so werden die Kanten durch den Medianfilter erhalten. In stark strukturierten Regionen können Kanten hingegen nicht erhalten werden. Abb. 12 stellt die Ergebnisse der bisher vorgestellten Glättungsfilter gegenüber. Es ist zu erkennen, dass der 7x7 Medianfilter in etwa gleich viele Rauschanteile entfernt hat wie der 11x11 Gauß-Filter. Viele Kanten wurden jedoch besser erhalten.

Der Median-Filer ist ein Schritt in die richtige Richtung, da Kanten zumindest teilweise erhalten werden. Die im folgenden Abschnitt vorgestellten kantenerhaltenden Glättungsfiler weisen jedoch eine deutlich höhere kantenerhaltende Funktion auf.

2.3 Kantenerhaltende Glättungsfiler

2.3.1 Bilateral Filter

Der bilaterale Filter [1] ist eine Erweiterung des Gauß-Filters bei der zusätzlich zum räumlichen Abstand auch der Intensitätsabstand der Pixel berücksichtigt wird. Die Intensitätsabstände werden durch eine eindimensionale Gauß-Funktion gewichtet. Dadurch fließen Pixel mit hohem Intensitätsabstand nur gering in das Ergebnis ein, während Pixel mit niedrigem Intensitätsabstand das Ergebnis stark beeinflussen. Anders ausgedrückt wird jeder ursprüngliche Intensitätswert innerhalb des Filterfensters sowohl mit dem räumlichen Gewicht als auch dem ermitteltem Intensitätsgewicht multipliziert. Anschließend wird der Ausgabewert durch arithmetische Mittelwertbildung berechnet. Da hohe Intensitätssprünge im Allgemeinen Kanten kennzeichnen, bedeutet eine zusätzliche Intensitätsgewichtung, dass Pixel jenseits einer Kante nur schwach berücksichtigt werden. Daraus ergibt sich, dass Kanten nur geringfügig in angrenzende Regionen einfließen und das gefilterte Bild nur wenig Schärfe verliert. Der Begriff bilateral bezieht sich darauf, dass zwei Arten von Abständen berücksichtigt werden. Formal ist der bilaterale Filter wie folgt definiert:

$$I'(p) = \frac{1}{W} \sum_{q \in R_p} G_{\sigma_s}(\|p - q\|) \cdot G_{\sigma_I}(|I_p - I_q|) \cdot I_q \quad (5)$$

$$W = \sum_{q \in R_p} G_{\sigma_s}(\|p - q\|) \cdot G_{\sigma_I}(|I_p - I_q|)$$

p und q sind zweidimensionale Koordinaten. R_p ist ein auf Pixel p zentriertes Fenster. W ist ein Gewichtungsfaktor um den ursprünglichen Wertebereich beizubehalten. $G_{\sigma_s}(\|m - n\|)$ ist die räumliche Gewichtung eines Pixels. $G_{\sigma_I}(|I_m - I_n|)$ die Intensitätsgewichtung eines Pixels. σ_s und σ_I sind die Standardabweichungen der beiden Gauß-Funktionen, welche durch den Benutzer festgelegt werden.



Abb. 13: (a) Verrauschtes Originalbild
(b) Ergebnis nach Anwendung des 11x11 Bilateral Filters

Die Gewichtung der Abstände erfolgt meist mit der Gauß-Funktion. Dies ist aber nicht zwingend erforderlich. Tatsächlich können beliebige Funktionen zur Gewichtung herangezogen werden. In [2] werden verschiedene Gewichtungsfunktionen evaluiert.

Abb. 13 zeigt ein bilateral gefiltertes Bild. Im Vergleich zum Median-Filter wurden stark strukturierte Regionen, wie Haar und Auge, besser erhalten.

2.3.1.1 Joint Bilateral Filter

Der Joint Bilateral Filter unterscheidet sich vom Bilateral Filter lediglich dadurch, dass die Intensitätsgewichte nicht direkt im zu filternden Bild berechnet werden. Stattdessen wird ein anderes Bild zur Berechnung der Intensitätsabstände herangezogen. Ursprünglich wurde der Joint Bilateral Filter verwendet, um ein Problem der Fotografie zu lösen [7]: Aufnahmen, die in einer schwach beleuchteten Umgebung entstanden sind, besitzen im Allgemeinen einen hohen Rauschanteil. Es ist möglich dieses Rauschen durch bilaterale Filterung mit hohem σ_I zu minimieren, allerdings wird durch die hohe Standardabweichung auch die Kantenschärfe stark reduziert. Eine andere Möglichkeit ist die Verwendung eines Blitzlichts bei der Aufnahme. In diesem Fall ist der Rauschanteil des aufgenommenen Bildes gering. Daher kann ein bilateraler Filter mit niedrigem σ_I verwendet werden und Kanten werden nur

geringfügig beeinträchtigt. Durch den Blitz werden jedoch vorhandene Lichtquellen überdeckt und Intensitäten verfälscht aufgenommen. Dieses Problem kann mit Hilfe des Joint Bilateral Filters gelöst werden: Bei der Aufnahme wird die Szene zweimal fotografiert. Einmal ohne Blitzlicht und einmal mit Blitzlicht. Anschließend wird das ohne Blitz aufgenommene Bild mit Hilfe eines Joint Bilateral Filter gefiltert, wobei zur Berechnung der Intensitätsgewichte das mit Blitz aufgenommene Bild herangezogen wird. Da letzteres nur einen geringen Rauschanteil aufweist, ist es möglich ein entsprechend niedriges σ_I zur Berechnung der Intensitätsgewichte zu verwenden. Durch die Joint Bilaterale Filterung bleiben daher sowohl die gewünschten Intensitäten als auch Kanten weitgehend erhalten.

Formal ist der Joint Bilateral Filter wie folgt definiert:

$$I'(p) = \frac{1}{W} \sum_{q \in R_p} G_{\sigma_s}(\|p - q\|) \cdot G_{\sigma_I}(\|J_p - J_q\|) \cdot I_q \quad (6)$$

$$W = \sum_{q \in R_p} G_{\sigma_s}(\|p - q\|) \cdot G_{\sigma_I}(\|J_p - J_q\|)$$

I ist das zu filternde Bild, I' das gefilterte Bild, J das Bild in dem die Intensitätsabstände berechnet werden.

In den Kapiteln 5 und 6 werden weitere Anwendungen des Joint Bilateral Filters vorgestellt.

2.3.2 Bilateral Median-Filter

Ein Nachteil des Bilateral Filters ist, dass bei zunehmenden σ_I nicht nur mehr Rauschanteile entfernt werden, sondern auch Kanten verstärkt in angrenzende Regionen einfließen. Dieses unerwünschte Verhalten kann reduziert werden, indem das arithmetische Mittel durch einen gewichteten Median ersetzt wird [2]: Jedem Intensitätswert innerhalb des Filterfensters wird ein Gewicht zugeordnet, welches sich aus dem Produkt des ermitteltem räumlichen- und Intensitätsgewichtes ergibt. Anschließend wird der gewichtete Median berechnet.

Abb. 14 (nächste Seite) zeigt ein mit Bilateral Median-Filter gefiltertes Bild. Im direkten Vergleich zum Bilateral Filter ist jedoch keine erhöhte Kantenschärfe erkennbar. *Abb. 15* (nächste Seite) zeigt, dass der Bilateral Median-Filter dem Bilateral Filter in bestimmten Situationen überlegen ist. Der Bilateral Filter führt eine gewichtete Mittelwertbildung durch, wodurch Punkte jenseits der Kante, wenn auch nur mit geringer Gewichtung, ebenfalls in das



Abb. 14: (a) Ergebnis nach Anwendung des 11x11 Bilateral Filters
(b) Ergebnis nach Anwendung des 11x11 Bilateral Median-Filters

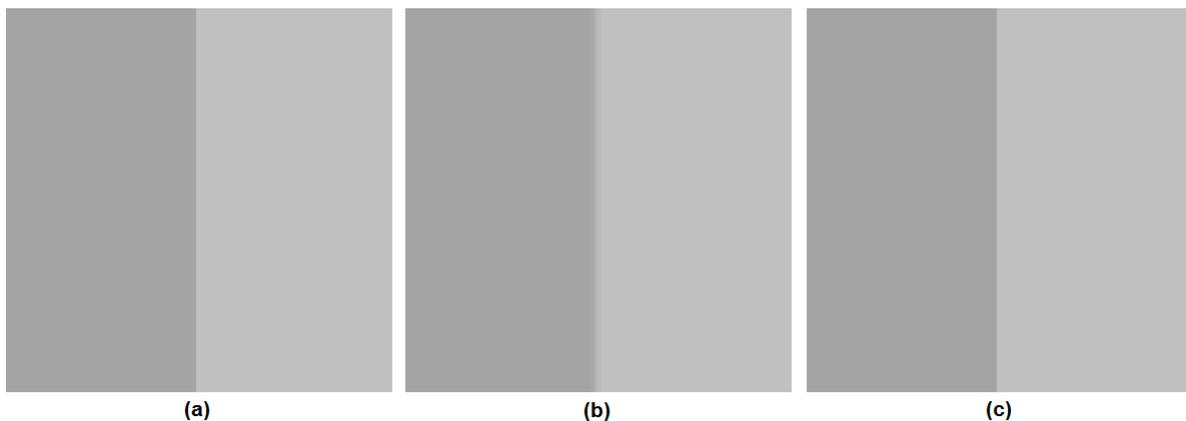


Abb. 15: (a) Originalbild
(b) Ergebnis nach Anwendung des 5x5 Bilateral Filters
(c) Ergebnis nach Anwendung des 5x5 Bilateral Median-Filters

Ergebnis einfließen. Dadurch wurden in diesem Beispiel Intensitäten in unmittelbarer Kantennähe aneinander angeglichen. Beim Bilateral Median-Filter blieb die Kantenschärfe hingegen vollständig erhalten. Natürlich wäre diese Kante auch bei Einsatz eines (nicht bilateralen) Median Filter erhalten geblieben. Der Bilateral Median-Filter kann daher auch als Kombination der Vorteile von Bilateral- und Median-Filter gesehen werden, da die kantenerhaltenden Eigenschaften von Median- und Bilateral-Filter zu einem Filter verschmolzen werden. Bei der Filterung natürlicher Bilder unterscheiden sich die Ergebnisse von Bilateral Filter und Bilateral Median Filter jedoch oft nicht signifikant, wie bereits in Abb. 14 gezeigt.



Abb. 16: (a) Varianz einer homogenen Region
(b) Varianz einer Region mit Kanten

2.3.3 Guided Image Filter

Bilateral Filter und Bilateral Median-Filter liefern zufriedenstellende Ergebnisse. Eine effiziente Implementation ist jedoch schwierig. Insbesondere können hochauflösende Bilder auf derzeit verbreiteter Hardware nur durch approximative Varianten des Bilateral Filters in Echtzeit gefiltert werden (siehe Kapitel 3). Jede Approximation führt jedoch zu mehr oder weniger sichtbaren Ungenauigkeiten. Dieser Umstand war die Hauptmotivation bei der Entwicklung des Guided Image Filters [3]. Es sollte ein Filter erschaffen werden, der hinsichtlich Qualität den Bilateral Filter erreicht, sowie gleichzeitig auf derzeit vorhandener Hardware effizient in exakter Form implementiert werden kann.

Um die Funktionsweise des Guided Image Filters zu veranschaulichen soll zunächst ein unvollständiger Algorithmus vorgestellt werden, welcher die dahinterliegende Idee verdeutlicht, ohne mit Details abzulenken:

Der Guided Image Filter entscheidet für jeden Bildpunkt, ob er sich in einem homogenen Bereich oder einem Bereich mit Kante(n) befindet. Diese Entscheidung erfolgt anhand der Varianz der Pixel innerhalb des Filterfensters. In einer homogenen Region (siehe Abb. 16(a)) weichen die Pixel des Fensters nur gering vom Mittelwert ab. Die Varianz ist daher niedrig. Befindet sich in der Nähe des Pixels hingegen eine Kante (Abb. 16(b)), so ist die Varianz entsprechend hoch. In homogenen Bereichen wird das Rauschen durch arithmetisches Mitteln der Intensitäten reduziert. Bereiche mit Kanten bleiben hingegen unangetastet, da eine Mittelwertbildung die Kanteninformation verfälschen würde.

Tatsächlich trifft der Guided Image Filter jedoch keine binäre Entscheidung, sondern ermittelt anhand der Varianz wie stark Mittelwert und ursprünglicher Intensitätswert in das Ergebnis einfließen. Bei niedriger Varianz (homogener Bereich) fließt der Mittelwert stark in das Ergebnis ein, der ursprüngliche Intensitätswert nur schwach. Ist die Varianz hingegen hoch

(Bereich mit Kanten), so ist die Situation genau umgekehrt: Der ursprüngliche Intensitätswert dominiert das Ergebnis und wird vom Mittelwert nur leicht beeinflusst. Es ist ein Kompromiss, um auch in Bereichen mit Kanten eine gewisse Glättung, auf Kosten der Kantenschärfe, zu erreichen. Dieses Verhalten wird in *Formel 7* festgehalten.

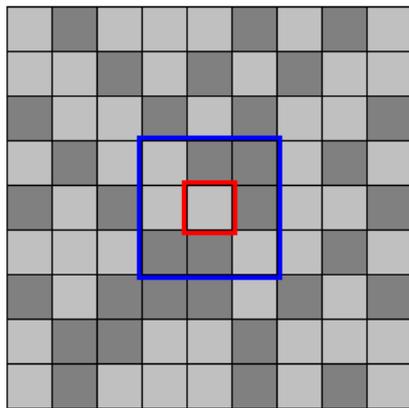
$$q_i = a_k I_i + b_k \quad (7)$$

$$a_k = \frac{1}{|\omega|} \frac{\sum_{j \in \omega_k} I_j^2 - \mu_k^2}{\sigma_k^2 + \varepsilon} \quad b_k = \mu_k - a_k \mu_k$$

I ist das zu filternde Bild, q das gefilterte Bild. i definiert die Koordinaten des zu filternden Pixels, weiters gilt $k = i$. ω_k ist ein auf k zentriertes Fenster, $|\omega|$ die Größe des Fensters (in Pixel), μ_k der Mittelwert aller Pixel in ω_k , σ_k^2 die Varianz in ω_k . ε ist ein Regularisierungsparameter (siehe nächster Absatz).

Im Ausdruck $q_i = a_k I_i + b_k$ wird der ursprüngliche Intensitätswert I_i mit dem Ergebnis eines Ausdrucks (a_k) multipliziert und anschließend das Ergebnis eines weiteren Ausdrucks (b_k) addiert. Das Ergebnis des Ausdrucks a_k steigt mit der Varianz: Bei hoher Varianz (Bereich mit Kanten) ist a_k hoch, bei niedriger Varianz (homogener Bereich) hingegen nahe 0. Mit dem Regularisierungsparameter ε kann gesteuert werden, welche Varianzbereiche als hoch bzw. niedrig klassifiziert werden. Seine Funktion entspricht in etwa dem σ_I des Bilateral Filters. Das Ergebnis des Ausdrucks b_k besteht aus dem Mittelwert der Pixel innerhalb des Filterfensters (μ_k), welcher entsprechend dem Ergebnis des Ausdrucks a_k abgeschwächt wird: Je höher a_k ist umso niedriger wird b_k und umgekehrt. Dies entspricht genau dem zuvor in Textform beschriebenen Verhalten: In einem Bereich mit hoher Varianz liegt a_k nahe bei 1, b_k nahe bei 0. Der ursprüngliche Pixelwert fließt also stark in das Ergebnis ein. Bei homogenen Bereichen liegt hingegen a_k nahe bei 0 und b_k nimmt den Mittelwert der Region an. In diesem Fall fließt der Mittelwert stark in das Ergebnis ein, der homogene Bereich wird geglättet.

Diese vereinfachte Variante des Guided Image Filters soll nun so erweitert werden, dass $k = i$ nicht mehr gilt: In einem ersten Schritt werden a_k und b_k für alle möglichen Pixelpositionen k berechnet. Bei einem 1 Megapixel-Bild werden beispielsweise jeweils 1.000.000 a_k und b_k berechnet. In einem zweiten Schritt werden die Ausgabewerte berechnet, wobei in den gefilterten Wert einer konkreten Pixelposition i nur jene a_k und b_k einfließen, deren auf k zentriertes Fenster den zu filternden Pixel umfasst. Es gibt keinen



Guided Image Filter bei einem 3x3 Fenster

Schritt 1:

Berechne a_k und b_k für jede Pixelposition k

In diesem Beispiel werden jeweils 81 a_k und b_k berechnet.

Schritt 2:

Berechne die Ausgabewerte

In den gefilterten Wert des rot umrandeten Pixel fließen beispielsweise alle a_k und b_k des blau umrandeten Bereichs ein.

Abb. 17: Guided Image Filter bei einem 3x3 Fenster

Grund anzunehmen, dass a_k und b_k bei $k = i$ "korrekter" ist als im Fall $k \neq i$. Daher ist es naheliegend alle a_k und b_k innerhalb des Filterfensters mit gleichem Gewicht zu berücksichtigen. Ein solcher Filter wird in *Formel 8* definiert.

$$q_i = \bar{a}_i I_i + \bar{b}_i \quad (8)$$

$$\bar{a}_i = \frac{1}{|\omega|} \sum_{k \in \omega_i} a_k \quad \bar{b}_i = \frac{1}{|\omega|} \sum_{k \in \omega_i} b_k$$

$$a_k = \frac{\frac{1}{|\omega|} \sum_{j \in \omega_k} I_j^2 - \mu_k^2}{\sigma_k^2 + \varepsilon} \quad b_k = \mu_k - a_k \mu_k$$

\bar{a}_i ist der Mittelwert aller a_k , die in die zu filternde Pixelposition einfließen, \bar{b}_i der Mittelwert aller b_k , die in die zu filternde Pixelposition einfließen. ω_k ist ein auf Pixel k zentriertes Fenster, ω_i ein auf i zentriertes Fenster.

Abb. 17 zeigt eine grafische Zusammenfassung des erweiterten Guided Image Filters.

Der vollständige Guided Image Filter unterscheidet sich nur in einem Punkt: Neben dem zu filternden Bild wird noch ein weiteres Bild, das sogenannte Guidance Image, in den Algorithmus einbezogen. Das Guidance Image, welches zur Orientierung dient, erfüllt dieselbe Funktion wie das „zweite“ Bild des Joint Bilateral Filters (das Bild in dem die

Intensitätsabstände berechnet werden). In bestehenden Anwendungen kann daher der Joint Bilateral Filter durch den Guided Image Filter ersetzt werden, um die Verarbeitungsgeschwindigkeit zu erhöhen. Sind Guidance Image und zu filterndes Bild hingegen identisch, so liefert der Guided Image Filter zum Bilateral Filter vergleichbare Ergebnisse. Folglich kann der Guided Image Filter auch den Bilateral Filter bestehender Anwendungen ersetzen. *Formel 9* definiert den vollständigen Guided Image Filter:

$$q_i = \bar{a}_i I_i + \bar{b}_i \quad (9)$$

$$\bar{a}_i = \frac{1}{|\omega|} \sum_{k \in \omega_i} a_k \quad \bar{b}_i = \frac{1}{|\omega|} \sum_{k \in \omega_i} b_k$$

$$a_k = \frac{\frac{1}{|\omega|} \sum_{j \in \omega_k} I_j p_j - \mu_k \bar{p}_k}{\sigma_k^2 + \varepsilon} \quad b_k = \bar{p}_k - a_k \mu_k$$

I ist das zu filternde Bild, p das Guidance Image, q das gefilterte Bild. μ_k definiert den Mittelwert von I in ω_k , σ_k^2 die Varianz von I in ω_k , \bar{p}_k den Mittelwert von p in ω_k . Alle weiteren Variablen erfüllen dieselbe Funktion wie zuvor (siehe Formel 8, Seite 19).

Abb. 18 (nächste Seite) zeigt ein mit Guided Image Filter gefiltertes Bild. Im Vergleich zum Bilateral Filter sind jedoch nur minimale Unterschiede feststellbar. Der Vorteil des Guided Image Filters liegt vielmehr in der signifikant höheren Verarbeitungsgeschwindigkeit (siehe Kapitel 3).

Abschließend soll ausdrücklich betont werden, dass Ergebnisse des Guided Image Filters weder denen des Bilateral Filters noch denen des Joint Bilateral Filters exakt entsprechen. Der Guided Image Filter ist ein eigenständiger Algorithmus dessen Ergebnisse jedoch im Allgemeinen dem Bilateral Filter bzw. Joint Bilateral Filter sehr ähnlich sind. Ergebnisse der vorgestellten Filter werden auch in den Kapiteln 5 und 6 gegenübergestellt.

2.4 Glättung von Farbbildern

Bisher wurde von Graustufenbildern ausgegangen, welche aus einem Kanal bestehen. In vielen Fällen wird jedoch mit Farbbildern gearbeitet, die drei Kanäle beinhalten (RGB , $YC_B C_R$ oder andere Farbräume mit 3 Komponenten). Eine Möglichkeit ist die getrennte Verarbeitung aller Kanäle: In diesem Fall wird jede Komponente des Farbraums als



Abb. 18: (a) Ergebnis nach Anwendung des 11x11 Bilateral Filters
(b) Ergebnis nach Anwendung des 9x9 Guided Image Filters, $\varepsilon = 0.004$

eigenständiges Bild betrachtet. Der Filter wird also dreimal aufgerufen. Diese Methode ist einfach zu implementieren, erreicht jedoch oft nicht das bestmögliche Ergebnis, da viele Kanten nicht in allen Kanälen diskriminativ sind. In diesem Abschnitt sollen verbesserte Farbbildvarianten von Joint Bilateral Filter und Guided Image Filter vorgestellt werden. Es wird davon ausgegangen, dass lediglich das Guidance Image (bzw. beim Joint Bilateral Filter das Bild in dem die Farbabstände berechnet werden) drei Kanäle beinhaltet. Zu filterndes Bild und Ergebnis bestehen weiterhin nur aus Grauwerten. Farbfilter dieser Art werden in vielen Anwendungsgebieten eingesetzt (siehe Kapitel 5 und 6).

Im Fall des Joint Bilateral Filters wird die Farbumstellung durchgeführt, in dem der Intensitätsabstand durch einen Farbabstand ersetzt wird: Anstatt wie bisher die absolute Differenz zweier Intensitäten zu berechnen (siehe *Formel 5*, Seite 13), werden beide Farben als dreidimensionale Koordinaten interpretiert. Der Farbabstand entspricht dem euklidischen Abstand dieser Koordinaten.

Die Farbversion des Guided Image Filters wird bereits im Originalpaper [3] definiert:

$$\begin{aligned}
 q_i &= a_i^T I_i + \bar{b}_i \\
 a_k &= (\Sigma_k + \varepsilon U)^{-1} \cdot \left(\frac{1}{|\omega|} \sum_{i \in \omega_k} I_i p_i - \mu_k \bar{p}_k \right) \\
 b_k &= \bar{p}_k - a_k^T \mu_k
 \end{aligned} \tag{10}$$

Σ_k ist die 3x3 Kovarianzmatrix von I in ω_k , U die 3x3 Identitätsmatrix. Alle weiteren Variablen erfüllen dieselbe Funktion wie bisher (siehe *Formel 9*, Seite 20).

3. Sequentielle Implementierung

In diesem Kapitel wird gezeigt wie traditionelle- und kantenerhaltende Glättungsfiler effizient implementiert werden können. Abgesehen von den $O(1)$ -Approximationen des Gauß- und Bilateral Filters wurden alle vorgestellten Algorithmen im Rahmen dieser Diplomarbeit implementiert.

3.1 Mittelwertfilter

Ein naiver Algorithmus zur Mittelwertbildung besteht aus der direkten Implementation der zweidimensionalen linearen Faltung:

```

01:  for y = 1:image_height:
02:      for x = 1:image_width:
03:          
$$I'(x, y) = \sum_{i=-\frac{p}{2}}^{\frac{p}{2}} \sum_{j=-\frac{q}{2}}^{\frac{q}{2}} I(x+i, y+j) \cdot H(i + \frac{p}{2}, j + \frac{q}{2})$$


```

Pseudocode 1: Naiver Mittelwertfilter

image_height und *image_width* definieren respektive Höhe und Breite des zu filternden Bildes. Zeile 03 definiert die zwei-dimensionale lineare Faltung (siehe Seite 4), deren Variablen zur besseren Übersicht an dieser Stelle nochmals wiederholt werden: I ist das ursprüngliche Bild (Eingabebild), I' das gefilterte Bild (Ausgabebild). x und y definieren die Koordinaten des zu berechnenden Pixels. Die Parameter p und q definieren die Dimension des Filterkerns. Der zweidimensionale Filterkern H bestimmt die Gewichtung der Nachbarpixel. Das Problem des naiven Algorithmus liegt in dieser Zeile, da die Zahl der für die Faltung eines Pixel benötigten Schleifendurchläufe quadratisch mit der Größe des Filterkerns steigt (siehe Abb. 19, nächste Seite).

Bei einer eindimensionalen linearen Faltung (siehe *Formel 10*) würde die Zahl der für die Faltung benötigten Schleifendurchläufe hingegen lediglich linear mit der Filtergröße zunehmen [6].

$$I'(x) = \sum_{i=-\frac{p}{2}}^{\frac{p}{2}} I(x+i) \cdot H(i + \frac{p}{2}) \quad (10)$$

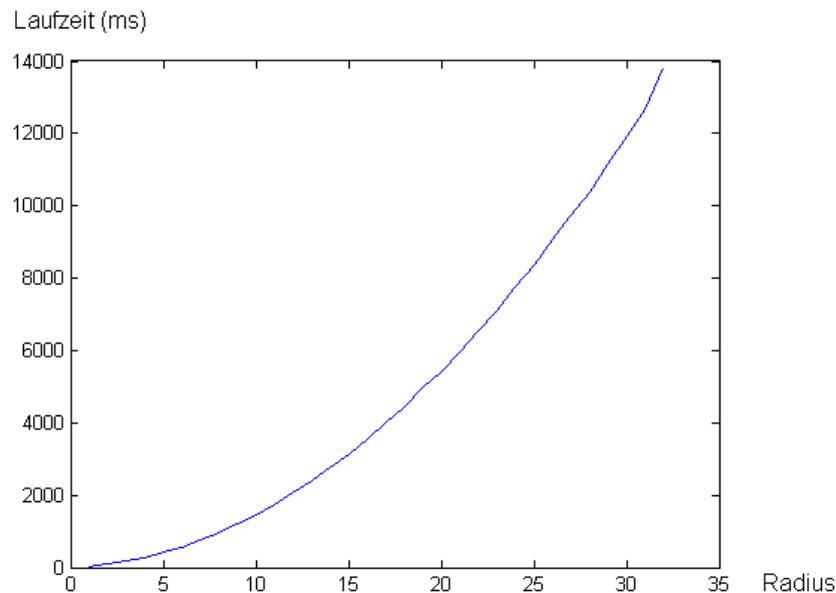


Abb. 19: Quadratische Laufzeit des im Rahmen dieser Diplomarbeit implementierten naiven Mittelwertfilters, Radius: 1 - 32, Größe des gefilterten Bildes: 1280 x 720, CPU: Core 2 Duo E6750, 2.66GHz

Es ist leicht zu erkennen, dass ein Algorithmus, der zwei oder mehrere eindimensionale Faltungen hintereinander ausführt, weiterhin linear bezüglich der Größe des Filterkerns bleibt. Es stellt sich also die Frage, ob es möglich ist zwei eindimensionale Filterkerne zu definieren, die sequentiell angewendet, ein zur zweidimensionalen Faltung identisches Ergebnis liefern. Aufgrund der mathematischen Eigenschaften der linearen Faltung [6] ist dies der Fall wenn gilt:

$$H = H_x * H_y \quad (11)$$

H ist ein zweidimensionaler Filterkern, H_x und H_y zwei eindimensionale Filterkerne, * die Matrizenmultiplikation.

Im konkreten Fall des Mittelwertfilters werden, wie bereits in Abschnitt 2.2.1 beschrieben, alle Elemente des Filterkerns H auf den konstanten Wert $1 \div (m \cdot n)$ gesetzt, wobei m und n den Dimensionen des Filterkerns entsprechen. Um Formel 11 zu erfüllen wird H_x als Spaltenvektor der Dimension n definiert, H_y als Zeilenvektor der Dimension m . Alle Elemente von H_x werden auf den konstanten Wert $1 \div n$ gesetzt. Analog wird allen Elementen des Kerns H_y der Wert $1 \div m$ zugewiesen. Dieses Prinzip der Teilung kann auf Mittelwertfilter beliebiger Größe angewendet werden. Der Mittelwertfilter ist daher

$$H = \begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix} \quad H_x = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix} \quad H_y = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \end{bmatrix} \quad H = H_x * H_y$$

Abb. 20: Separierung des 3x3 Mittelwertfilters

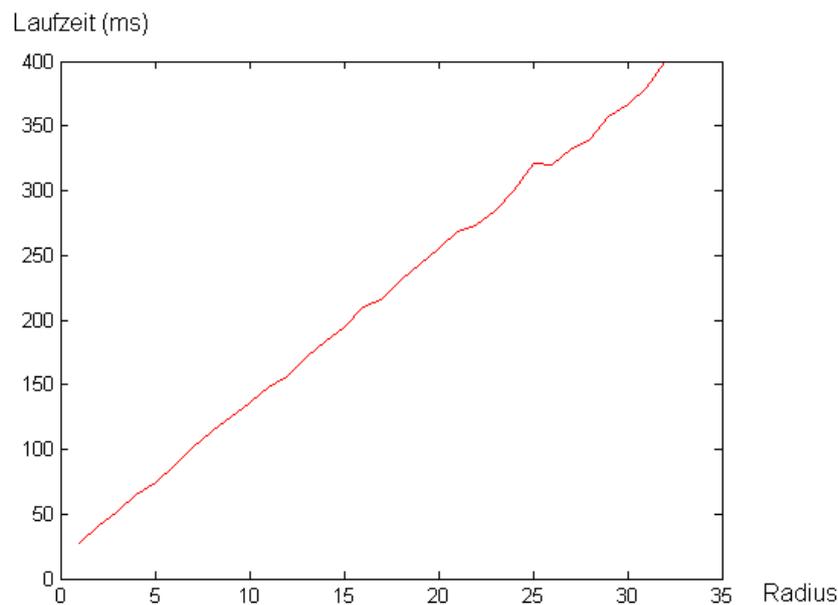


Abb. 21: Lineare Laufzeit des separierten Mittelwertfilters, Radius 1 – 32, Größe des gefilterten Bildes: 1280 x 720, CPU: Core 2 Duo E6750, 2.66GHz

separierbar. Abb. 20 demonstriert die Separation eines 3x3 Mittelwertfilter. Aufgrund der Assoziativität der linearen Faltung ist es irrelevant mit welchem der beiden Kerne zuerst gefaltet wird. Abb. 21 zeigt die Laufzeit des separierten Mittelwertfilters.

Der separierte Mittelwertfilter bietet einen deutlichen Performancegewinn gegenüber der naiven Implementierung. Um die Performance weiter zu steigern, kann er zu einem Mittelwertfilter mit konstanter Laufzeit bezüglich der Kernelgröße modifiziert werden. Dieser O(1)-Mittelwertfilter beruht auf folgender Idee [8]: Werden die Mittelwerte zweier benachbarter Pixelpositionen berechnet, so fließen nahezu dieselben Intensitätswerte des Originalbildes in die beiden Ergebnisse ein (siehe Abb. 22, nächste Seite). Es ist daher nicht notwendig, den Mittelwert an jeder Pixelposition durch Faltung neu zu berechnen. Stattdessen kann das Ergebnis der zuvor berechneten Pixelposition angepasst werden, indem das Gewicht der herausgefallenen Position subtrahiert und das Gewicht der neu

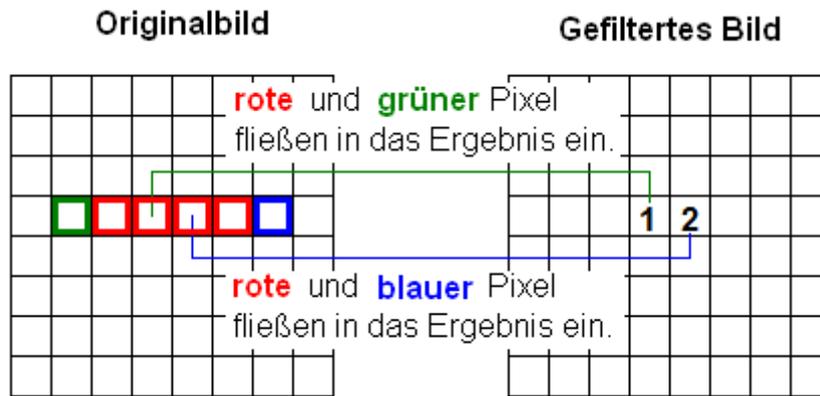


Abb. 22: Berechnung des Mittelwerts an Position 2 bei 5x1 Filterkern
 Vom bereits berechneten Mittelwert der Position 1 wird der grün umrandete Pixel subtrahiert und der blau umrandete Pixel addiert.

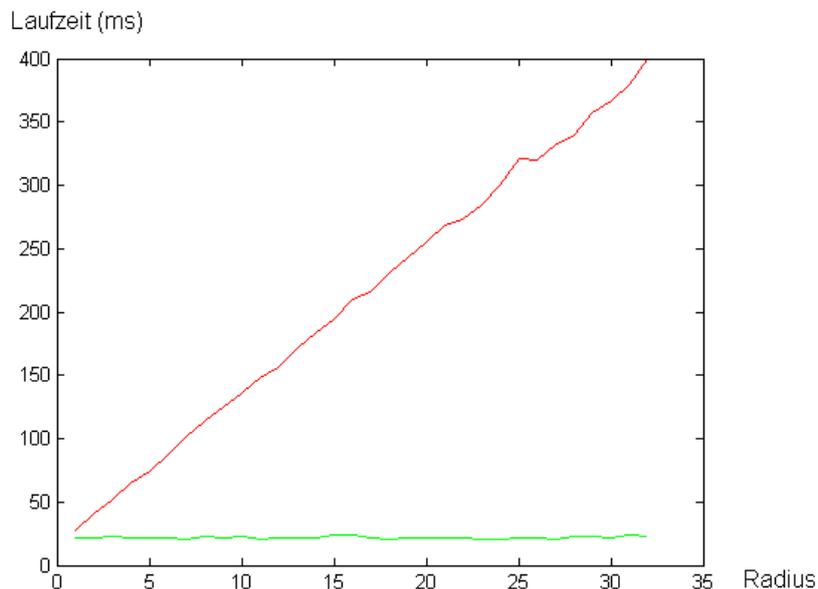


Abb. 23: Laufzeit des $O(1)$ -Mittelwertfilters (grün),
 Laufzeit des linearen Mittelwertfilters (rot),
 Radius 1 – 32, Größe des gefilterten Bildes: 1280 x 720,
 CPU: Core 2 Duo E6750, 2.66GHz

hinzugekommenen Position addiert wird. Das Gewicht einer konkreten Pixelposition wird ermittelt indem der entsprechende Intensitätswert durch die Größe des Filterfensters dividiert wird. Da unabhängig von der Filtergröße jeweils ein Gewicht subtrahiert und ein Gewicht addiert wird, besitzt dieser Filter eine konstante Laufzeit bezüglich der Kernelgröße. Lediglich an der ersten Pixelposition jeder Zeile/Spalte (je nachdem ob in X- oder Y-Richtung gefiltert wird) muss der Ausgabewert durch lineare Faltung berechnet werden.

Abb. 23 zeigt den Performancegewinn des $O(1)$ -Mittelwertfilters.

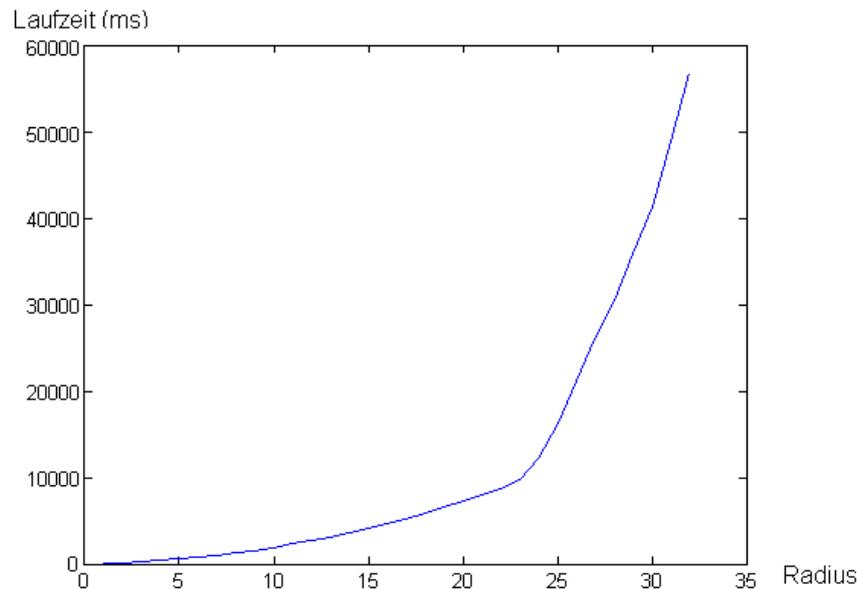


Abb. 24: Laufzeit des naiven Gauß-Filters, Radius: 1 - 32, Größe des gefilterten Bildes: 1280 x 720, CPU: Core 2 Duo E6750, 2.66GHz

3.2 Gauß-Filter

Der Gauß-Filter ist, wie der Mittelwertfilter, ein linearer Filter und kann daher ebenfalls naiv durch zweidimensionale lineare Faltung implementiert werden. Der Faltungskern des Gauß-Filters wurde bereits in Abschnitt 2.2.2 (Seite 8) definiert. *Abb. 24* zeigt die Laufzeit einer naiven Implementation. Wie bereits beim Mittelwertfilter ist ein quadratischer Anstieg bezüglich der Kernelgröße zu erkennen.

Analog zum Mittelwertfilter kann auch beim Gauß-Filter durch Separation ein lineares Laufzeitverhalten bezüglich der Kernelgröße erreicht werden. Die für die beiden eindimensionalen Faltungen benötigten Kerne werden folgendermaßen aufgebaut, um *Formel 11* (Seite 24) zu erfüllen: H_x ist ein Spaltenvektor, dessen Dimension der Anzahl der Zeilen des zweidimensionalen Filterkerns entspricht. H_y ein Zeilenvektor, dessen Dimension der Anzahl der Spalten des zweidimensionalen Filterkerns entspricht. Die Werte beider Vektoren werden entsprechend der eindimensionalen Gauß-Funktion (siehe *Formel 2*, Seite 8) gesetzt, deren Standardabweichung von der ursprünglichen zweidimensionalen Gauß-Funktion übernommen wird. *Abb. 25* (nächste Seite) demonstriert die Separation eines 3x3 Gauß-Filter mit $\sigma = 0.5$. *Abb. 26* (ebenfalls nächste Seite) zeigt die Laufzeit des separierten Gauß-Filters. Wie zu erwarten war, ist die Performance im Vergleich zur naiven Variante signifikant gestiegen.

$$H = \begin{bmatrix} 0.0113 & 0.0838 & 0.0113 \\ 0.0838 & 0.6196 & 0.0838 \\ 0.0113 & 0.0838 & 0.0113 \end{bmatrix} \quad H_x = \begin{bmatrix} 0.1065 \\ 0.787 \\ 0.1065 \end{bmatrix}$$

$$H_y = [0.1065 \quad 0.787 \quad 0.1065] \quad H = H_x * H_y$$

Abb. 25: Separierung des 3x3 Gauß-Filter, $\sigma = 0.5$

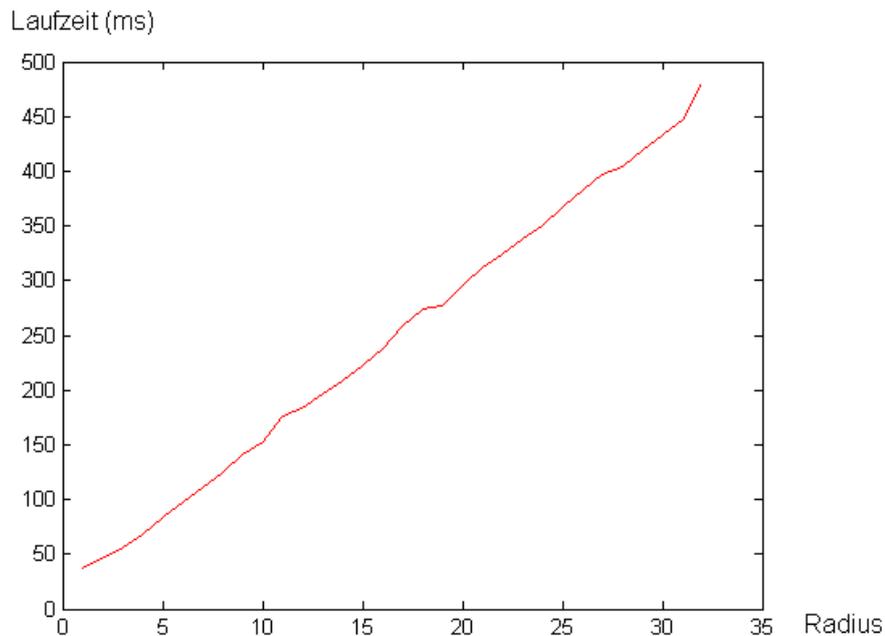


Abb. 26: Lineare Laufzeit des separierten Gauß-Filters, Radius 1 – 32, Größe des gefilterten Bildes: 1280 x 720, CPU: Core 2 Duo E6750, 2.66GHz

Eine weitere Optimierung des Gauß-Filters ist schwierig. Insbesondere ist es nicht möglich eine konstante Laufzeit bezüglich der Kernelgröße zu erzielen. Um dennoch eine Verbesserung zu erreichen, versuchen Deriche [9] und Van Vliet et al. [10] den eindimensionalen Gauß-Filter durch rekursive IIR-Filter (Infinite Impulse Response Filter) zu approximieren. Rekursiv bedeutet im Kontext von Filtern, dass auch Ausgabewerte vorhergehender Schritte zur Berechnung des aktuellen Ausgabewerts genutzt werden. Eine Strategie, die bereits beim $O(1)$ -Mittelwertfilter erfolgreich eingesetzt wurde. Die von Deriche und Van Vliet et al. vorgeschlagenen Approximationen des Gauß-Filters weisen eine konstante Laufzeit bezüglich der Kernelgröße auf.

Bei der Gauß-Approximation nach Deriche wird jede Bildzeile zweimal durchlaufen. Zunächst in einem Vorwärtsdurchgang von links nach rechts. Anschließend in einem

Rückwärtsdurchgang in entgegengesetzter Richtung. Die Anzahl der in die Filterung einbezogenen vorhergehenden Ausgabewerte kann beliebig gewählt werden. Durch die Einbeziehung zusätzlicher Ausgabewerte kann die Approximation auf Kosten der Geschwindigkeit verbessert werden. In der Praxis werden häufig zwei vorhergehende Ausgabewerte benutzt, um einen guten Kompromiss zu erzielen. Diese Version wird in *Formel 12* definiert. Varianten, die eine höhere Anzahl von Ausgabewerten einsetzen, können der Arbeit von Deriche [9] entnommen werden.

Vorwärtsdurchgang (von links nach rechts): (12)

$$y[n] = a_0 \cdot x[n] + a_1 \cdot x[n-1] - b_1 \cdot y[n-1] - b_2 \cdot y[n-2]$$

Rückwärtsdurchgang (von rechts nach links):

$$y[n] = a_2 \cdot x[n+1] + a_3 \cdot x[n+2] - b_1 \cdot y[n+1] - b_2 \cdot y[n+2]$$

Konstanten:

$$a_0 = \frac{(1 - e^{-\alpha})^2}{1 + 2\alpha \cdot e^{-\alpha} - e^{-2\alpha}}$$

$$a_1 = a_0 \cdot (\alpha - 1) \cdot e^{-\alpha}$$

$$a_2 = a_0 \cdot (\alpha + 1) \cdot e^{-\alpha}$$

$$a_3 = -a_0 \cdot e^{-2\alpha}$$

$$b_1 = -2 \cdot e^{-\alpha}$$

$$b_2 = e^{-2\alpha}$$

wobei $\alpha = 1.695 \div \sigma$

x ist das zu filternde Bild, y enthält nach dem Vorwärtsdurchgang ein Zwischenergebnis, nach dem Rückwärtsdurchgang das gefilterte Bild. n definiert die aktuelle Position. Im Vorwärtsdurchgang iteriert n von 0 bis Zeilenende, im Rückwärtsdurchgang umgekehrt von Zeilenende bis 0. An den Rändern erfolgen Zugriffe auf Pixel außerhalb des Bildbereichs. Für diese Pixel wird in Praxis häufig der konstante Wert 0 angenommen. a_0, a_1, a_2, a_3, b_1 und b_2 sind Konstanten, deren Wert ausschließlich von der gewählten Standardabweichung σ abhängt.

Beim herkömmlichen Gauß-Filter werden, wie in Abschnitt 2.2.2 (Seite 8) beschrieben, zwei Parameter festgelegt, Standardabweichung σ und Kernelgröße. Um ein ungünstiges Frequenzverhalten zu vermeiden, wird dabei die Kernelgröße in der Regel so gewählt, dass

die Randkoeffizienten des Filterkerns nahe bei 0 liegen. Rekursive Gauß-Filter nach Deriche verwenden lediglich die Standardabweichung σ . Durch die Elimination des Parameters Kernelgröße wird erreicht, dass die Laufzeit ausschließlich von der Auflösung des zu filternden Bildes abhängt.

Eine zweidimensionale Approximation kann implementiert werden indem die in Formel 12 definierten IIR-Filter in einem zweiten Schritt entlang der Bildspalten angewendet werden. *Abb. 27* (nächste Seite) vergleicht den Gauß-Filter mit der zweidimensionalen Approximation nach Deriche. Die Intensitäten des Differenzbildes wurden mit dem Faktor 16 multipliziert, um Unterschiede visuell wahrnehmbar zu machen. Es ist zu erkennen, dass die Deriche-Approximation im Vergleich zur exakten Implementierung nur minimale Unterschiede aufweist. Da diese geringen Abweichungen oft bedeutungslos sind, können bestehende Gauß-Filter in vielen Anwendungen durch Approximationen nach Deriche ersetzt werden. Die Laufzeit der Deriche-Implementation auf einem mit 1280x720 Punkten aufgelösten Bild beträgt auf dem Rechner des Autors dieser Diplomarbeit 52ms, unabhängig von der gewählten Standardabweichung. Im Vergleich zum separierten Gauß-Filter (vergleiche *Abb. 26*, Seite 28) ist eine signifikant höhere Geschwindigkeit feststellbar.

3.3 Median-Filter

Der Median-Filter ist, im Gegensatz zu Mittelwert- und Gauß-Filter, nicht linear und kann daher nicht durch Faltung implementiert werden. Eine naive Implementierung berechnet den Median einer Pixelposition durch aufsteigendes Sortieren der Intensitäten innerhalb des Filterfensters. Der mittlere Wert der sortierten Folge entspricht dem Median. Die Laufzeit bezüglich der Kernelgröße hängt vom gewählten Sortierverfahren ab, liegt jedoch in jedem Fall über $O(n^2)$, da die Anzahl der zu sortierenden Pixel quadratisch mit der Kernelgröße steigt. *Abb. 28* (nächste Seite) zeigt das Laufzeitverhalten der im Rahmen dieser Diplomarbeit erstellten Implementation, welche das QuickSort-Sortierverfahren benutzt.

Huang [11] definiert in seiner bereits 1981 veröffentlichten Arbeit einen Algorithmus mit linearer Laufzeit bezüglich der Kernelgröße. Er bestimmt das Histogramm aller Intensitäten der Nachbarschaftsregion. Da ein Histogrammaufbau zwangsweise zu geordneten Intensitäten führt, kann so der Aufruf einer teuren Sortierfunktion umgangen werden.



Abb. 27: (a) Verrauschtes Originalbild
 (b) Differenz der beiden gefilterten Bilder
 (c) Ergebnis nach Anwendung des 11x11 Gauß-Filters, $\sigma = 2.00$
 (d) Ergebnis nach Anwendung der Gauß-Approximation nach Deriche, $\sigma = 2.00$

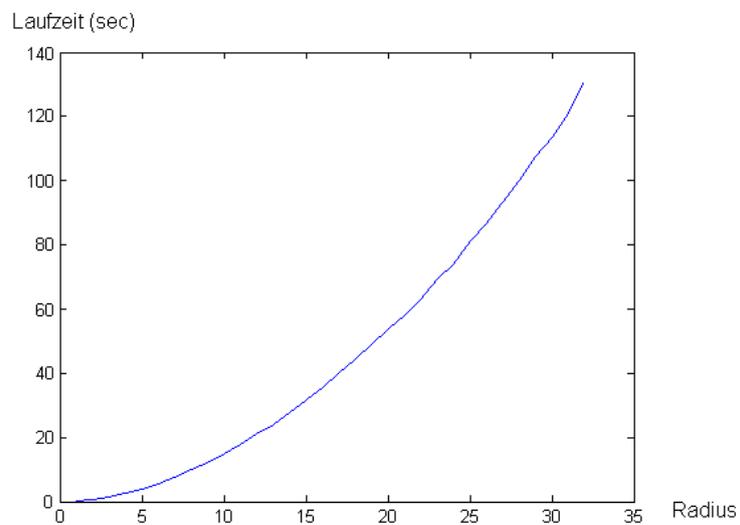


Abb. 28: Laufzeit des naiven Median-Filters,
 Radius: 1 - 32, Größe des gefilterten Bildes: 1280 x 720,
 CPU: Core 2 Duo E6750, 2.66GHz

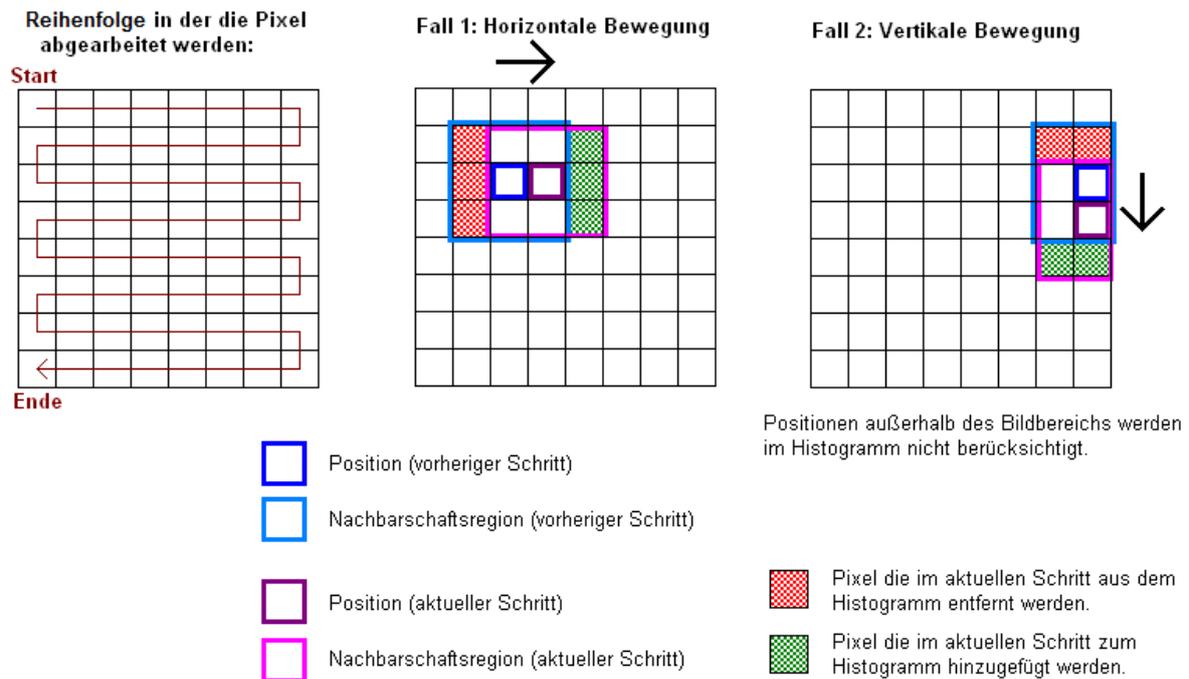


Abb. 29: $O(n)$ Median-Filter, Radius = 1

Im Gegenzug müssen allerdings, um den Median zu bestimmen, alle Histogramm-Bins der Reihe nach akkumuliert werden, bis die Medianposition erreicht oder überschritten wurde. Aus der sortierten Folge der naiven Implementierung konnte der Median hingegen direkt (ohne Schleife) ermittelt werden. Die Laufzeit der Akkumulationsschleife ist jedoch konstant bezüglich der Kernelgröße, da die Zahl der Histogramm-Bins ausschließlich von Zahl der möglichen Intensitätsstufen determiniert wird. Dennoch führt diese Modifikation noch nicht zur erwünschten linearen Laufzeit, da die Anzahl der Nachbarschaftspixel weiterhin quadratisch mit dem Radius steigt. Um das Ziel zu erreichen bedient sich Huang einer Idee, die bereits vom $O(1)$ -Mittelwertfilter bekannt ist. Dort wurde, anstatt den Mittelwert für jeden Ausgabepixel neu zu berechnen, die Summe des vorherigen Schrittes übernommen und aktualisiert. Dieselbe Idee kann auf den Median-Filter angewendet werden: anstatt das Histogramm für jeden Pixel komplett neu aufzubauen wird das Histogramm des vorherigen Schrittes aktualisiert. Dieses Verfahren wird in Abb. 29 demonstriert.

Nur für den ersten Pixel muss das Histogramm komplett aufgebaut werden. Anschließend werden pro besuchter Pixelposition jeweils $2 \cdot \text{radius} + 1$ Elemente aus dem Histogramm entfernt und hinzugefügt. Die Pixel werden in einer schlangenförmigen Reihenfolge abgearbeitet, um nicht am Zeilenende auf den Anfang der nächsten Zeile springen zu müssen (andernfalls müsste das Histogramm zu Beginn jeder Bildzeile komplett neu aufgebaut werden). Die Laufzeit ist bezüglich der Kernelgröße linear, da die Anzahl der Elemente, die dem Histogramm hinzugefügt- bzw. daraus entfernt werden, linear mit

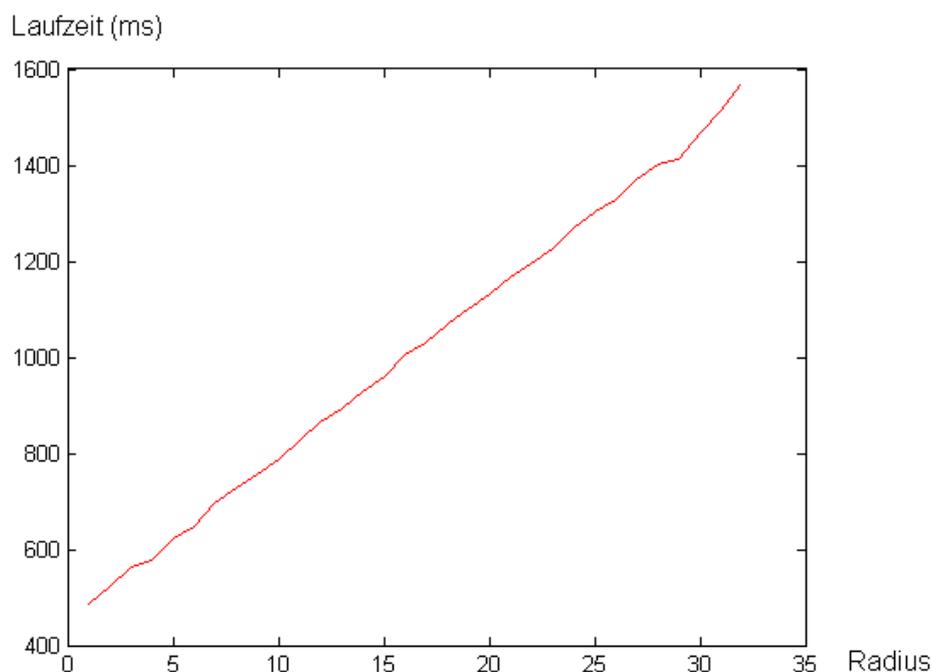


Abb. 30: Lineare Laufzeit des Algorithmus von Huang, Radius 1 – 32, Größe des gefilterten Bildes: 1280 x 720, CPU: Core 2 Duo E6750, 2.66GHz

der Kernelgröße wächst. Abb. 30 visualisiert die lineare Laufzeit von Huangs Median-Algorithmus. Im Vergleich zur naiven Implementierung ist die Performance signifikant gestiegen.

Huang empfiehlt die Verwendung einer optimierten Version seines Akkumulierungsalgorithmus. Im Normalfall müssen bei einem 8-Bit Grauwertbild im Durchschnitt 128 Bins aufsummiert werden, um den Median zu erreichen. Da der Median zweier benachbarter Pixel ähnlich ist wäre es vorteilhaft bei der Position des zuletzt berechneten Medians mit der Suche beginnen zu können. Dies wird durch folgende Veränderungen erreicht:

1. Für den ersten Pixel wird wie bisher das vollständige Histogramm aufgebaut und der Median durch schrittweise Aufsummierung ab dem ersten Bin gesucht. Nachdem der Median gefunden wurde, wird zusätzlich die Nummer des Median-Bins in einer Variablen gespeichert (*medBin*). Die Anzahl der Elemente vor dem Median-Bin wird ebenfalls in einer Variablen abgelegt (*count*).
2. Für jeden weiteren Pixel werden die Histogramme wie gehabt aktualisiert. Zusätzlich muss bei jedem Einfügen bzw. Entfernen eines Histogrammelements die Variable *count* angepasst werden, wenn sich das betroffene Bin vor *medBin* befindet.

3. Die Variable *count* zeigt wie viele Elemente sich vor *medBin* befinden. Es ist daher möglich festzustellen, ob sich der Median vor *medBin* befindet oder nicht. Je nachdem wird bei der Medianbestimmung eine andere Suchrichtung gewählt:
 - Sollte sich der Median vor *medBin* befinden, so wird in absteigende Bin-Richtung gesucht, beginnend bei *medBin - 1*. Es werden solange Bins von *count* abgezogen bis die Medianposition *unterschritten* wurde.
 - Sollte sich der Median *nicht* vor *medBin* befinden, so wird in aufsteigende Bin-Richtung gesucht, beginnend bei *medBin*. Es werden solange Bins zu *count* addiert, bis die Medianposition *erreicht oder überschritten* wurde.

Nachdem der Median gefunden wurde, wird die Variable *medBin* auf die entsprechende Bin-Nummer gesetzt und *count* auf die Anzahl der Elemente vor dem Median-Bin gesetzt.

Bei der Implementation des Algorithmus in Visual C++ 2008 kam es zu einem interessanten Problem: Die Laufzeit war langsamer als die Variante ohne optimierten Akkumulationsalgorithmus. Dieses Problem tritt auf, wenn beim Compilervorgang der Parameter *Optimization* auf *Maximize Speed* oder *Full Optimization* gesetzt wird. Wurde *Optimization* hingegen in *beiden Testprogrammen* auf *Disabled* gesetzt, so war die Variante mit optimiertem Akkumulationsalgorithmus deutlich schneller. Allerdings führt die Einstellung *Optimization Disabled* zu einem deutlichen Performanceeinbruch, der durch den optimierten Akkumulationsalgorithmus nicht ausgeglichen werden kann. Verantwortlich für dieses Problem sind die zusätzlichen *if-Abfragen* in *Schritt 2* (siehe Aufzählung oben). Diese behindern den Microsoft Compiler bei der Optimierung. Da die Optimierungsstrategien von Visual C++ 2008 nicht öffentlich einsehbar sind, war es nicht möglich, das Problem zu beheben. Aus diesem Grund wird an dieser Stelle auch auf eine graphische Darstellung der Performance verzichtet. Es ist fraglich, ob diese Optimierung auf gängigen Hardware-Plattformen tatsächlich einen Geschwindigkeitsvorteil bringt.

Lange Zeit galt der von Huang entwickelte Algorithmus als State-of-the-Art. Seit einigen Jahren gibt es jedoch Algorithmen, die eine verbesserte Laufzeit vorweisen können [12][13][14][15]. In dieser Diplomarbeit wird der Algorithmus von Perreault et al. [15]

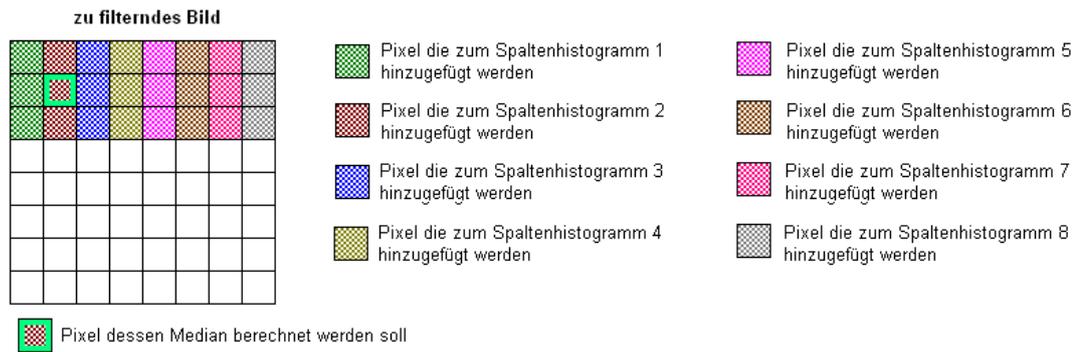


Abb. 31: Initialisierung der Spaltenhistogramme in einem 8x8 Bild, Radius des Median-Filters: 1

vorgestellt, welcher eine konstante Laufzeit bezüglich der Kernelgröße aufweist. Verantwortlich für die lineare Laufzeit des Huang-Algorithmus ist die Histogrammaktualisierung, da in jedem Schritt jeweils $2 \cdot radius + 1$ Pixel entfernt und hinzugefügt werden müssen. Der $O(1)$ -Algorithmus von Perreault und Hebert, welcher ebenfalls mit Histogramm arbeitet, umgeht dieses Problem durch Verwendung mehrerer Histogramme. Im folgenden Algorithmus werden zur besseren Übersichtlichkeit nur jene Pixel gefiltert, deren Nachbarschaft komplett im Bildbereich enthalten ist:

1. Für jede Spalte des Bildes wird ein Histogramm aufgebaut, wobei in jedem Histogramm nur die obersten $2 \cdot radius + 1$ Pixel der jeweiligen Spalte berücksichtigt werden (siehe *Abb. 31*). Dieser Schritt besitzt eine lineare Laufzeit bezüglich der Kernelgröße, wird jedoch nur einmal zur Initialisierung ausgeführt. Als erstes soll der Median des linken, oberen Pixels ermittelt werden.
2. Die ersten $2 \cdot radius + 1$ Spaltenhistogramme werden zu einem sogenannten Kernel-Histogramm zusammengefügt (siehe *Abb.32*, nächste Seite). Dieser Schritt besitzt eine lineare Laufzeit bezüglich der Kernelgröße, wird jedoch nur einmal pro Bildzeile ausgeführt.
3. Der Median des aktuellen Pixels wird aus dem Kernel-Histogramm ermittelt. Dieser Schritt ist konstant bezüglich der Kernelgröße, da die Anzahl der Histogramm-Bins ausschließlich von der Zahl der möglichen Intensitätsstufen abhängt.

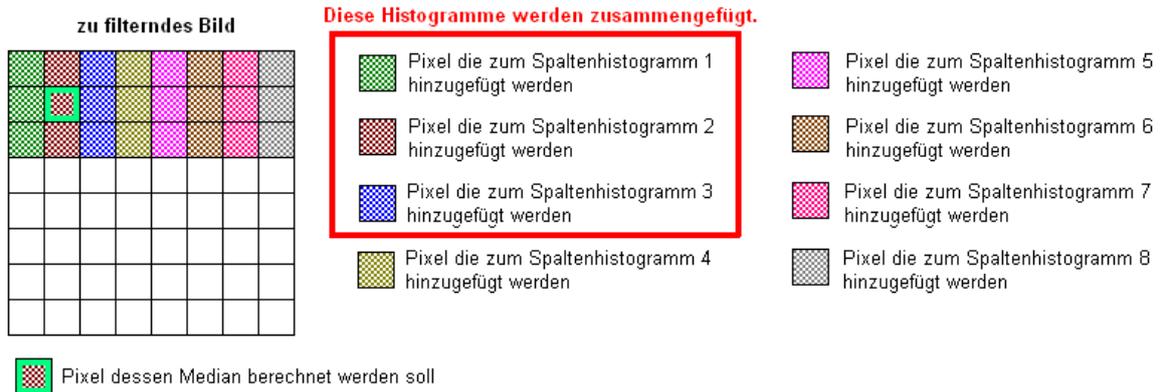


Abb. 32: Initialisierung des Kernel-Histogramms in einem 8x8 Bild, Radius des Median-Filters: 1



Abb. 33: Aktualisierung des Kernel-Histogramms in einem 8x8 Bild bei Wechsel zu umrandeten Position, Radius des Median-Filters: 1

- Als nächstes soll der Median des rechten Nachbarpixels berechnet werden. Dazu wird zunächst das Kernel-Histogramm aktualisiert. Es wird jeweils ein Spaltenhistogramm vom Kernel-Histogramm subtrahiert und addiert, sodass das aktualisierte Spaltenhistogramm alle Pixel der Nachbarschaft des neuen aktuellen Pixels abdeckt (siehe Abb. 33). Dieser Schritt ähnelt der Histogrammaktualisierung nach Huang. Es besteht jedoch ein entscheidender Unterschied: Hier werden Histogramme subtrahiert/addiert. Die Berechnungen sind daher konstant bezüglich der Kernelgröße.
- Berechnung des Medians (siehe auch Schritt 3).

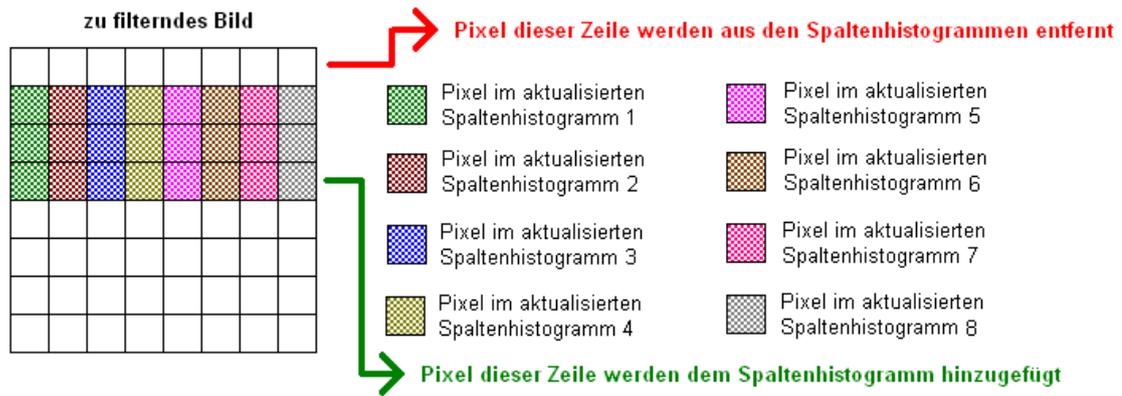


Abb. 34: Aktualisierung der Spaltenhistogramme in einem 8x8 Bild bei Wechsel auf die 3. Bildzeile, Radius des Median-Filters: 1

6. Fortfahren bei Schritt 4 wenn der rechte Zeilenrand noch nicht erreicht wurde. Ansonsten fortfahren bei Schritt 7 wenn die letzte Bildzeile noch nicht gefiltert wurde. Andernfalls terminiert der Algorithmus.
7. Fortfahren mit dem Pixel am linken Bildrand der nächsten Bildzeile. Zunächst werden alle Spaltenhistogramme aktualisiert. Es wird jeweils ein Wert subtrahiert und addiert, so dass alle Spaltenhistogramme über der neuen Zeile „zentriert“ sind (siehe Abb. 34). Dieser Schritt ist konstant bezüglich der Kernelgröße. Anschließend wird mit Schritt 2 fortgefahren.

Der beschriebene Algorithmus hat einen entscheidenden Nachteil: Das Zusammenfügen der Spaltenhistogramme zum Kernel-Histogramm (Schritt 4) ist zwar konstant bezüglich der Kernelgröße, allerdings ist der Aufwand sehr hoch. Bei jedem Positionswechsel muss jeweils ein Histogramm addiert und subtrahiert werden, um das Kernel-Histogramm aktuell zu halten. Im Falle eines 8-Bit Grauwertbildes entspricht dies $2 \cdot 2^8 = 512$ Rechenoperationen. Huangs linearer Filter benötigt bei der Histogrammaktualisierung hingegen lediglich $2 \cdot 2 \cdot radius + 1$ Rechenschritte. Es ist zu erkennen, dass der konstante Filter bei Radien bis 127 mehr Rechenschritte benötigt ($2 \cdot 2^8 > 2 \cdot 2 \cdot 127 + 1$). Um dieses Problem zu umgehen empfehlen Perreault et al. die Verwendung von Multilevel-Histogrammen: Anstatt eines Spaltenhistogramms mit 256 Bins wird ein zweistufiges Spaltenhistogramm aufgebaut. Die untere Stufe enthält weiterhin 256 Bins. Die obere Stufe enthält hingegen 16 Bins, die jeweils 16 aufeinanderfolgende Bins der unteren Stufe zusammenfassen (siehe Abb. 35, nächste Seite). Bei Aufbau und Aktualisierung der Spaltenhistogramme ist somit die doppelte Anzahl an Operationen notwendig, da jeweils beide Stufen aktualisiert werden müssen (Schritt 1, Schritt 7). Dieser Aktualisierungsvorgang erfolgt jedoch nur einmal pro Bildzeile. Der kritische

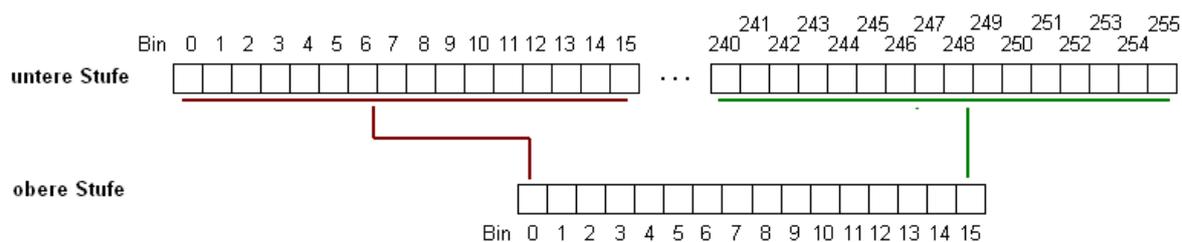


Abb. 35: Multilevel-Histogramm

Schritt 4 kann dagegen mit Hilfe der Multilevel-Histogramme signifikant beschleunigt werden, da in den meisten Fällen eine hohe Anzahl von Bins der oberen Stufe den Wert 0 aufweist. In einer Schleife wird über die 16 Bins der oberen Stufe iteriert. Nur für jene Bins, deren Wert ungleich 0 ist, werden die entsprechenden 16 Bins der unteren Stufe durchlaufen. Die Anzahl der benötigten Operationen wird stark reduziert. Weiters empfiehlt es sich das Kernel-Histogramm ebenfalls als Multilevel-Histogramm derselben Form zu speichern, da der Median im Kernel-Histogramm berechnet wird (Schritt 3, Schritt 5). Dieser Vorgang kann erheblich beschleunigt werden, indem zunächst ermittelt wird in welches Bin der oberen Stufe der Median fällt. Anschließend muss nur der entsprechende 16 Bin weite Bereich der unteren Stufe durchsucht werden. Somit reduziert sich die Anzahl der notwendigen Operationen im Durchschnitt von 128 auf 16 (8 Zugriffe in der oberen Stufe + 8 Zugriffe in der unteren Stufe). Es entsteht jedoch ein zusätzlicher Aufwand bei Aktualisierung des Kernel-Histogramms, da jeweils beide Histogrammstufen aktuell gehalten werden müssen.

Abb. 36 (nächste Seite) vergleicht die Laufzeiten von $O(1)$ und $O(n)$ Median-Filter. Erst bei größeren Filterkernen war der $O(1)$ Filter überlegen. Perreault et al. stellen in ihrer Arbeit einige weitere Optimierungen des $O(1)$ Median-Filter vor, die jedoch im Rahmen dieser Diplomarbeit nicht implementiert wurden. Ihre Messungen ergaben, dass eine hochoptimierte parallelisierte Version mit SIMD-Instruktionen einem optimierten $O(n)$ -Filter auch bei kleinen Filterkernen überlegen ist.

3.4 Bilateral-Filter

Die naive Implementierung des Bilateral Filters besteht aus der direkten Implementierung von Formel 5 (Seite 13). Da die Zahl der möglichen Intensitätsunterschiede beschränkt ist (in einem 8-Bit Grauwertbild gibt es bspw. 256 Möglichkeiten), empfiehlt es sich bei Ermittlung der Intensitätsgewichtung eine Look-Up Tabelle zu verwenden, anstatt die Gauß-Funktion

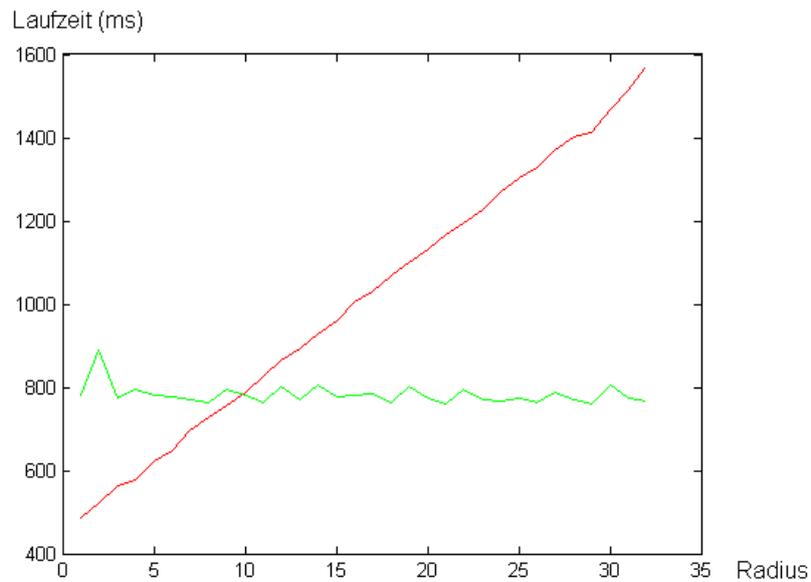


Abb. 36: Laufzeit des $O(n)$ -Algorithmus von Huang (rot),
 Laufzeit des $O(1)$ -Algorithmus von Perreault und Hebert (grün)
 Radius 1 – 32, Größe des gefilterten Bildes: 1280 x 720,
 CPU: Core 2 Duo E6750, 2.66GHz

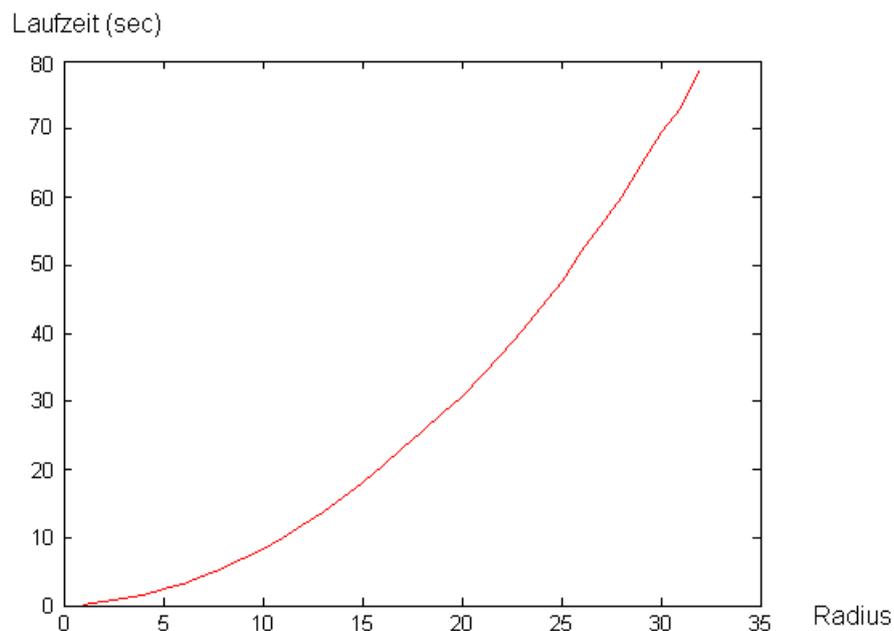


Abb. 37: Laufzeit des naiven Bilateral Filter,
 Radius 1 – 32, Größe des gefilterten Bildes: 1280 x 720
 CPU: Core 2 Duo E6750, 2.66GHz

wiederholt zu evaluieren. Weiters kann auch zur Ermittlung der räumlichen Gewichtung eine Look-Up Tabelle eingesetzt werden, da alle möglichen räumlichen Abstände bekannt sind (sie ergeben sich aus der Größe des Filterkerns). *Abb. 37* zeigt die Performance des naiven Algorithmus mit Look-Up Tabellen.

Bilateral Filter
(Formel 5, Seite 13)

$$I'(m) = \frac{1}{W} \sum_{n \in R_m} G_{\sigma_s}(\|m-n\|) \cdot G_{\sigma_I}(\|I_m - I_n\|) \cdot I_n$$

$$W = \sum_{n \in R_m} G_{\sigma_s}(\|m-n\|) \cdot G_{\sigma_I}(\|I_m - I_n\|)$$

I	...	zu filterndes Bild
I'	...	gefiltertes Bild
m	...	Koordinaten des zu filternden Pixels
R_m	...	Nachbarschaftsregion von m
$G_{\sigma_s}(\ m-n\)$...	räumliche Gewichtung
$G_{\sigma_I}(\ I_m - I_n\)$...	Intensitätsgewichtung

Algorithmus von Yang

```

01: für jede Intensitätsstufe i = 0..255
02:   for y = 1:height
03:     for x = 1:width
04:       diff = abs(i - I[x,y])
05:       W[x,y] = table[diff]
06:       J[x,y] = I[x,y] * table[diff]
07:     W = Gσs(W)
08:     J = GσI(J)
09:   for y = 1:height
10:     for x = 1:width
11:       if (I[x,y] == i)
12:         I'[x,y] = J[x,y] / W[x,y]
    
```

width	...	Breite des zu filternden Bildes
height	...	Höhe des zu filternden Bildes
I[w][h]	...	zu filterndes Bild (8-Bit, Grauwert)
I'[w][h]	...	gefiltertes Bild (8-Bit, Grauwert)
W[w][h]	...	
J[w][h]	...	temporäre Arrays (Floating-Point)
table[256]	...	Look-Up Tabelle zur Gewichtung der Intensitätsdifferenzen (Floating-Point)

Abb. 38: (a) Bilateral Filter
(b) O(1)-Algorithmus von Yang

Da der Bilateral Filter aufgrund seiner kantenerhaltenden Eigenschaften sehr nützlich sein kann, wurden zahlreiche optimierte Algorithmen entwickelt [16][17][18][19][20]. In dieser Diplomarbeit soll der Algorithmus von Yang et al. [21] vorgestellt werden, welcher eine konstante Laufzeit bezüglich der Kernelgröße aufweist.

Das Verhalten des Algorithmus wird über zwei Parameter gesteuert die bereits vom Bilateral Filter bekannt sind, σ_s und σ_I . σ_s definiert die Standardabweichung der beiden Gauß-Filter (Zeile 07 und 08, Abb. 38(b)). σ_I wird zum Aufbau der Look-Up Tabelle für die Intensitätsgewichtung (table) benötigt, welche wiederum in den Zeilen 05 und 06 verwendet wird. Der folgende Absatz beschreibt den bereits in Abb. 38(b) definierten Algorithmus von Yang et al. in Worten.

Die äußerste Schleife (Zeile 01) iteriert über alle möglichen Intensitätsstufen i . Innerhalb eines Schleifendurchlaufs werden die Ausgabewerte aller Pixel berechnet, die im zu filternden Bild die Intensität i aufweisen. Im ersten Durchlauf werden also alle Pixel mit Intensität 0 gefiltert, im zweiten Durchlauf alle Pixel mit Intensität 1, usw. Innerhalb jedes

Schleifendurchläufe werden schrittweise zwei Arrays J und W aufgebaut. J soll den **hellblau umrandeten Teil** der Formel des Bilateral-Filters entsprechen, W den **dunkelgelben Teil** (siehe *Abb. 38(a)*). Um die Arrays zu befüllen wird zunächst in einer Schleife über alle Pixel des zu filternden Bildes iteriert (Zeile 02 – 03). Für jeden Bildpunkt wird der absolute Intensitätsabstand zu i ermittelt (Zeile 04). In Zeile 05 werden aus den Intensitätsabständen mit Hilfe der Look-Up Tabelle die entsprechenden Intensitätsgewichte ermittelt und W befüllt. Zu diesem Zeitpunkt entspricht W dem **rot umrandeten Teil** der Formel des Bilateral Filters (*Abb. 38(a)*). In Zeile 06 wird analog J aufgebaut, welches zu diesem Zeitpunkt dem **dunkelblauen Teil** entspricht (*Abb. 38(a)*). J unterscheidet sich von W lediglich dadurch, dass die ermittelten Intensitätsgewichte mit den ursprünglichen Intensitätswerten multipliziert wurden. In Zeile 07 wird W mit Hilfe eines Gauß-Filters der Standardabweichung σ_s geglättet. Yang et al. empfehlen eine Gauß-Approximation nach Deriche einzusetzen (siehe Abschnitt 3.2), um eine konstante Laufzeit bezüglich der Kerngröße zu gewährleisten. Um die Auswirkung dieser Anweisung zu verstehen, muss bedacht werden, dass in jedem Durchlauf der äußersten Schleife (Zeile 01) nur Bildpunkte der Intensität i gefiltert werden sollen. An diesen Positionen werden durch den Gauß-Filter die benachbarten Intensitätsgewichte entsprechend ihres räumlichen Abstandes gewichtet und aufsummiert. Das Ergebnis entspricht dem **dunkelgelb umrandeten Teil** (*Abb. 38(a)*). An Positionen, deren Intensität ungleich i im zu filternden Bild ist, liefert die Gauß-Filterung hingegen kein verwertbares Ergebnis, da die Intensitätsabstände (Zeile 04) in Bezug auf i ermittelt wurden. Diese Bildpunkte werden in anderen Durchläufen der äußeren Schleife gefiltert. In Zeile 08 wird analog durch Gauß-Filterung von W der **hellblau umrandete Teil** berechnet (*Abb. 38(a)*). In einer weiteren Schleife (Zeile 09 – 10) werden alle Bildpunkte des Originalbildes, deren Intensität i entspricht (Zeile 11), im gefilterten Bild I' auf J / W (dem Wert des Pixels nach bilateraler Filterung, siehe *Abb.38(a)* **grau umrandeter Bereich**) gesetzt. Nach dem letzten Durchlauf der äußersten Schleife (Zeile 01) enthält I' das bilateral gefilterte Bild. Der Algorithmus kann leicht zum Joint Bilateral Filter erweitert werden, indem in den Zeilen 04 und 11 auf ein anderes Bild zugegriffen wird.

Ein Problem des Algorithmus ist der hohe Aufwand. In einem 8-Bit Grauwertbild wird die äußere Schleife 256-mal durchlaufen. Die beiden Gauß-Filter (Zeile 07 und 08) müssen also pro Bild jeweils 256-mal aufgerufen werden. Um dieses Problem zu umgehen, empfehlen Yang et al. eine Approximation. Anstatt über alle möglichen Intensitätswerte zu iterieren, wird die äußerste Schleife auf einige wenige beschränkt, z.B. 0, 17, 34, 51, 68, 85, 102, 119, 136, 153, 170, 187, 204, 221, 238 und 255. Im ersten Schleifendurchlauf werden wie bisher alle Pixel mit Intensität 0 gefiltert. Im zweiten Durchlauf werden hingegen alle Pixel mit Intensitäten von 1 bis 17 gefiltert, jedoch werden nur Pixel mit Intensität 17 exakt berechnet.

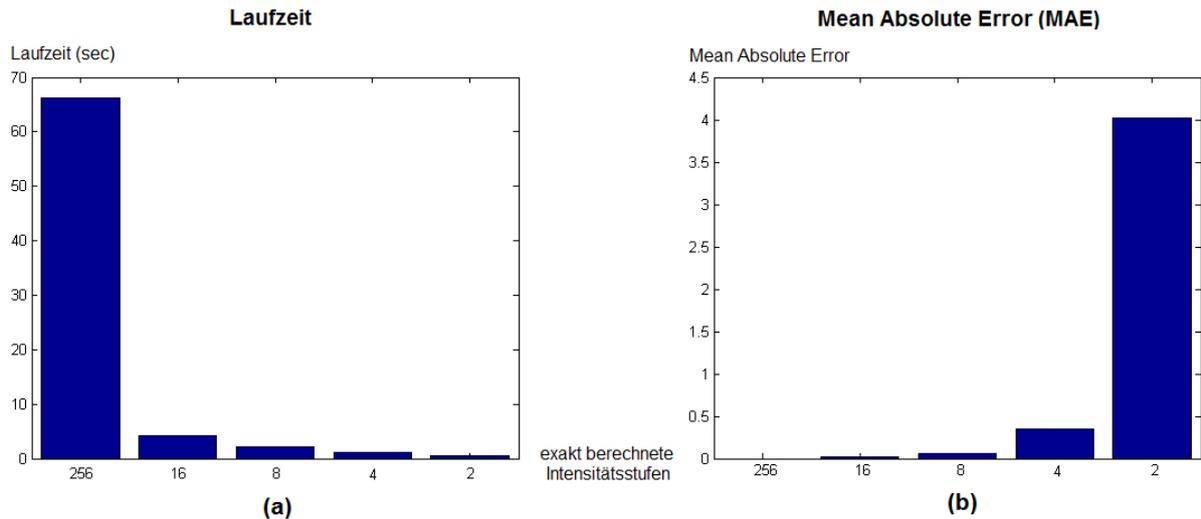


Abb. 39: (a) Laufzeit von Yangs Bilateral Filter
 (b) Mean Absolute Error von Yangs Bilateral Filter
 Größe des gefilterten Bildes: 1280 x 720
 CPU: Core 2 Duo E6750, 2.66GHz

Pixel, deren Intensität im Bereich 1 bis 16 liegt, werden hingegen durch lineare Interpolation gefiltert. Um dieses Verhalten zu erreichen, wird Zeile 12 wie folgt angepasst:

$$12: \quad I'[x,y] = \frac{[J[x,y] * \alpha + J2[x,y] * (1-\alpha)]}{W[x,y] * \alpha + W2[x,y] * (1-\alpha)}$$

$J2$ und $W2$ entsprechen J und W des vorherigen Schleifendurchlaufs. Am Ende jedes Schleifendurchlaufs muss also J nach $J2$ und W nach $W2$ kopiert werden. α wird entsprechend der Intensität des zu filternden Pixels so gewählt, dass eine lineare Interpolation zwischen J und $J2$ bzw. W und $W2$ stattfindet. Wenn die Intensität des zu filternden Pixels beispielsweise bei 1 liegt, wird α auf 0.05 (1 / 17) gesetzt, bei 16 hingegen auf 0.94 (16 / 17). Bei einer Intensität von 17 erhält α den Wert 1.0 (17 / 17). In diesem Fall ergibt sich das Ergebnis direkt aus J und W . Für alle Intensitätsstufen größer 17 wird in den folgenden Schleifendurchläufen analog vorgegangen. Durch die lineare Interpolation treten selbstverständlich Fehler auf. *Abb. 39* untersucht verschiedene Interpolationsstufen hinsichtlich Laufzeit und Abweichung vom exakten Ergebnis. *Abb. 40* (nächste Seite) vergleicht Ergebnisse unterschiedlicher Interpolationsstufen visuell.



256 Intensitätsstufen exakt berechnet



16 Intensitätsstufen exakt berechnet



8 Intensitätsstufen exakt berechnet



4 Intensitätsstufen exakt berechnet



2 Intensitätsstufen exakt berechnet

Abb. 40:

Visueller Vergleich unterschiedlicher Interpolationsstufen,
Größe des gefilterten Bildes: 512 x 512

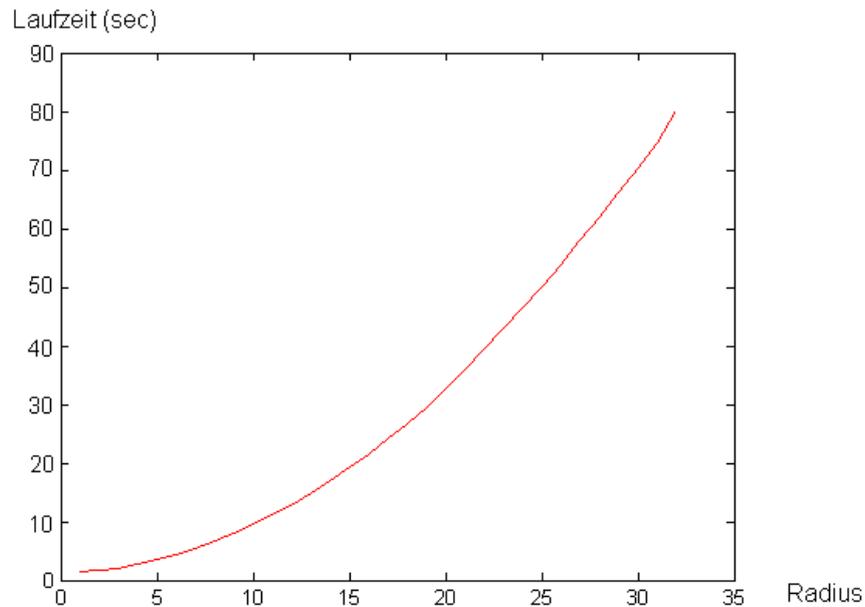


Abb. 41: Laufzeit des Bilateral Median-Filters,
Radius 1 – 32, Größe des gefilterten Bildes: 1280 x 720
CPU: Core 2 Duo E6750, 2.66GHz

3.5 Bilateral Median-Filter

Bei der Implementierung sollte die Sortierfunktion durch ein Histogramm ersetzt werden (siehe auch Huangs Median Filter, Seite 30). Im Gegensatz zu Huangs Median-Filter muss das Histogramm jedoch für jede Bildposition komplett neu aufgebaut werden. Der Bilateral Median-Filter mit Histogramm besitzt daher eine quadratische Laufzeit bezüglich der Kernelgröße (die Größe des Fensters steigt quadratisch mit dem Radius des Filterkerns).

Abb. 41 zeigt die Laufzeit des Bilateral Median-Filters mit Histogrammen.

Eine weitere Optimierung des Bilateral Median-Filters ist ausgesprochen schwierig. Insbesondere sind dem Autor dieser Diplomarbeit keine Algorithmen bekannt, die es ermöglichen, hochauflösende Bilder auf derzeit verbreiteter Hardware in Echtzeit zu filtern. Da die Qualität des Bilateral Median-Filters in den meisten Fällen nur minimal über der des Bilateral Filters liegt (siehe auch *Abb. 14*, Seite 16), wird in der Praxis häufig der wesentlich schnellere Bilateral Filter bevorzugt.

3.6 Guided Image Filter

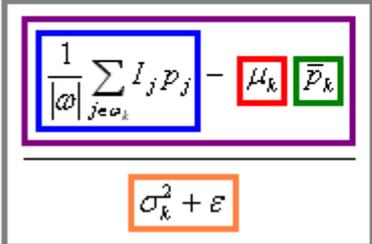
He et al. [3] definieren in ihrer Arbeit nicht nur den Guided Image Filter, sondern beschreiben auch eine Implementierung mit konstanter Laufzeit bezüglich der Kernelgröße. Da dieser optimierte Algorithmus gleichzeitig mit dem Guided Image Filter veröffentlicht wurde, wäre es nicht sinnvoll in dieser Diplomarbeit die Laufzeit einer naiven Implementierung zu messen. Stattdessen soll sofort zur Beschreibung der optimierten Variante übergegangen werden. Zur besseren Übersichtlichkeit hier noch einmal die Definition des Guided Image Filters (siehe auch Formel 9, Seite 20):

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

$$\bar{a}_i = \frac{1}{|\omega|} \sum_{k \in \omega_i} a_k$$

$$\bar{b}_i = \frac{1}{|\omega|} \sum_{k \in \omega_i} b_k$$

$$a_k = \frac{\frac{1}{|\omega|} \sum_{j \in \omega_k} I_j p_j - \mu_k \bar{p}_k}{\sigma_k^2 + \epsilon}$$

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


Nun zur Definition des $O(1)$ -Algorithmus: Wie bereits in Abschnitt 2.3.3 (Seite 17) beschrieben, berechnet der Guided Image Filter zunächst für jeden Pixel des zu filternden Bildes a_k . Es ist daher naheliegend, alle a_k in einem zweidimensionalen Array a zusammenzufassen. In diesem Fall kann a durch folgenden Algorithmus in konstanter Zeit bestimmt werden:

1. Mittelwertfilter auf das zu filternde Bild (I): Das gefilterte Bild wird als $mean_I$ abgelegt. Der Mittelwertfilter kann konstant bezüglich der Kerngröße implementiert werden (siehe 3.1). **Roter Bereich.**
2. Mittelwertfilter auf das Guidance Image (p): Das gefilterte Bild wird als $mean_p$ abgelegt. **Grüner Bereich.**
3. Multipliziere I pixelweise mit p und wende einen Mittelwertfilter auf das Ergebnis an. Das gefilterte Bild wird als $mean_Ip$ gespeichert. Die pixelweise Multiplikation zweier Bilder ist vom Filterradius unabhängig und beeinträchtigt daher nicht die konstante Laufzeit des Algorithmus. **Blauer Bereich.**

4. Berechne den Zähler des Ausdrucks durch pixelweises Multiplizieren und Subtrahieren ($mean_Ip - mean_I \cdot mean_p$). Das Ergebnis wird als Array cov_Ip gespeichert. Subtraktion und Multiplikation sind vom Filterradius unabhängig. **Violetter Bereich.**

5. Berechnung der Varianzen von I (σ_k^2). Es ist zu beachten, dass die Varianz einer Zahlenmenge auch durch Bildung der Differenz von Mittelwert der quadrierten Zahlen und quadrierten Mittelwert der ursprünglichen Zahlen berechnet werden kann: $\sigma_k^2 = \mu_{k^2} - (\mu_k)^2$. Um die Varianzen zu berechnen, werden also zunächst alle Pixel in I quadriert und anschließend ein Mittelwertfilter auf das Ergebnis angewendet. Das gefilterte Bild wird als $mean_II$ gespeichert. Anschließend werden die Varianzen σ_k^2 berechnet ($\sigma_k^2 = mean_II - mean_I \cdot mean_I$). Wie bereits in Schritt 4 erwähnt, sind Subtraktionen und pixelweise Multiplikationen vom Filterradius unabhängig. Gemäß Definition wird der Regularisierungsparameter ε zu den berechneten Varianzen addiert. Das Ergebnis wird im Array var_I abgelegt. **Oranger Bereich.**

6. Berechne a durch elementweise Division von cov_Ip und var_I . Analog zur Multiplikation ist auch die elementweise Division unabhängig von der Größe des Filterkerns. Nach Abarbeitung von Schritt 6 wurde daher a in konstanter Laufzeit bezüglich der Kernelgröße berechnet. **Grauer Bereich.**

Gleichermaßen werden alle b_k in einem Array b zusammengefasst. Die Werte werden berechnet, indem $mean_I$ mit I elementweise multipliziert und das Ergebnis anschließend von $mean_p$ subtrahiert wird ($b = mean_p - I \cdot mean_I$). Offensichtlich ist auch die Berechnung von b konstant bezüglich der Kernelgröße.

Im nächsten Schritt werden die Mittelwerte der Nachbarschaften aller a_k und b_k berechnet (\bar{a}_i, \bar{b}_i). In der O(1)-Implementierung werden diese Mittelwerte in zwei Arrays \bar{a} und \bar{b} zusammengefasst. Es ist leicht zu erkennen, dass \bar{a} und \bar{b} einfach durch Anwendung des Mittelwertfilters auf die bereits zuvor berechneten Arrays a bzw. b gebildet werden können. Die Berechnung von \bar{a} und \bar{b} ist somit ebenfalls konstant bezüglich der Filtergröße.

Im letzten Schritt wird das gefilterte Bild q wie folgt berechnet: $q = \bar{a} - I \cdot \bar{b}$. Elementweise Multiplikation und Subtraktion sind wie bereits mehrfach erwähnt konstant bezüglich der

Filterkerngröße. Somit können alle Einzelschritte des Algorithmus in konstanter Zeit ausgeführt werden.

Mit diesem Algorithmus kann ein 1280x720 großes Graustufenbild in 87.89ms gefiltert werden (Implementation des Autors dieser Diplomarbeit auf einem Core 2 Duo E6750, 2.66GHz). Eine im Vergleich zum Bilateral Filter beeindruckende Performancesteigerung: Yangs $O(1)$ -Algorithmus benötigte selbst bei lediglich zwei exakt berechneten Intensitätsstufen 590ms, die exakte Variante sogar 66.27sec (siehe auch *Abb. 39*, Seite 42). Bereits in Abschnitt 2.3.3 wurde gezeigt, dass der Guided Image Filter hinsichtlich der Qualität dem Bilateral Filter nicht nachsteht. Anders ausgedrückt können durch den Guided Image Filter qualitativ gleichwertige Ergebnisse in wesentlich kürzerer Zeit berechnet werden. Durch Multicore- und GPU-Implementation (letztere werden in Kapitel 4 behandelt) kann die Performance weiter verbessert werden.

4. Parallele Implementierung

In diesem Kapitel wird die CUDA-Architektur [4] des Grafikkartenherstellers NVIDIA vorgestellt und anhand des Mittelwertfilters die Parallelisierung eines konkreten Algorithmus demonstriert. Im Rahmen dieser Diplomarbeit wurden auch Bilateral Filter, Joint Bilateral Filter und Guided Image Filter in CUDA implementiert. Die Laufzeiten dieser Implementierungen werden ebenfalls präsentiert.

4.1 CUDA-Architektur

4.1.1 Motivation

Lange Zeit herrschte eine für den Programmierer angenehme Situation. Mit jeder neuen Generation arbeiteten Prozessoren Maschineninstruktionen schneller ab, wodurch bestehende Programme automatisch die höhere Rechenleistung kommender Prozessorgenerationen ausnutzen konnten. Aufgrund verschiedener physikalischer Probleme, wie beispielsweise der Wärmeableitung, können heute konstruierte Prozessoren Maschinenbefehle nur unwesentlich schneller abarbeiten als die jeweilige Vorgängergeneration [22]. Prozessorhersteller umgehen dieses Problem durch die Integration mehrerer parallel arbeitender Kerne in einem Prozessorgehäuse. Mit jeder neuen Prozessorgeneration wächst die Anzahl der Kerne. CPUs mit zwei Kernen sind heutzutage Standard, Quad-Core und Achtkern-CPU's auch im Consumer-Bereich keine Seltenheit mehr. Der Programmierer steht jedoch vor einem ihm bisher unbekanntem Problem: Bestehende Programme sind sequentiell und können daher nur auf einem Kern ausgeführt werden. Nur ein Bruchteil der vorhandenen Prozessorleistung wird ausgenutzt. Daher ist es wichtig bereits bei der Programmentwicklung Mehrkernsysteme zu berücksichtigen, indem Algorithmen parallelisiert und auf Kerne verteilt werden.

Neben der CPU befindet sich in modernen PCs eine weitere Komponente mit mehreren Kernen: die Grafikkarte. Grafikkarten besitzen schon seit langer Zeit eine Vielzahl von Kernen, da Berechnungen im Vertex- und Pixel-Shader massiv parallel ausgeführt werden. Durch diese parallele Ausführung können beispielsweise mehrere tausend Punkte eines komplexen geometrischen Objekts parallel gerendert werden. Nur so ist es möglich die High-End Grafik aktueller Spiele in Echtzeit zu berechnen. Vor einigen Jahren erkannten Grafikkartenhersteller, dass diese enorme Rechenleistung auch für Anwendungen außerhalb des Rendering-Bereichs genutzt werden kann [22]. Daher wurden die Architekturen so angepasst, dass beliebige Funktionen auf den Kernen der Grafikkarten ausgeführt werden können. Der Hersteller NVIDIA bezeichnet solche Karten als CUDA-fähig. Andere Hersteller

wie ATI/AMD bieten ebenfalls universal programmierbare Grafikkarten an. Aufgrund der ständig steigenden Verbreitung moderner Grafikkarten ist damit zu rechnen, dass in Zukunft viele Programmierer Algorithmen auf die Grafikkarte auslagern werden.

4.1.2 C for CUDA vs. OpenCL

Grafikkarten unterschiedlicher Hersteller besitzen verschiedene Architekturen mit unterschiedlichen Eigenschaften [23][24]. In der Praxis bedeutet das, dass beispielsweise ein für die CUDA-Architektur (NVIDIA-GPUs) optimiertes Programm auf ATI/AMD Karten nur suboptimal läuft und umgekehrt. In dieser Diplomarbeit wird die CUDA Architektur beschrieben und darauf eingegangen wie die in Abschnitt 3 vorgestellten Glättungsalgorithmen effizient auf CUDA-fähigen Grafikkarten implementiert werden können. Die Entscheidung für CUDA wurde getroffen, da sowohl auf den Testrechnern des Instituts als auch in den persönlichen PCs des Autors dieser Diplomarbeit ausschließlich NVIDIA-Karten eingebaut waren.

Genauso wie der Programmierer bei der Entwicklung von herkömmlichen Programmen die Wahl zwischen verschiedenen Programmiersprachen hat, kann auch bei der Entwicklung von Programmen für CUDA-fähige Grafikkarten zwischen den Alternativen C for CUDA [23] und OpenCL [25] gewählt werden. C for CUDA ist ein C-Dialekt der von NVIDIA um einige, für die parallele Programmierung essentielle, Konstrukte erweitert wurde. Die Programme sind nur auf NVIDIA-Grafikkarten lauffähig. OpenCL ist hingegen ein offener von einem Konsortium gemanagter Standard und nicht auf NVIDIA-Grafikkarten beschränkt. Das bedeutet insbesondere, dass OpenCL Programme auch auf ATI/AMD Grafikkarten lauffähig sind. Es soll an dieser Stelle noch einmal ausdrücklich darauf hingewiesen werden, dass Grafikkarten unterschiedlicher Hersteller verschiedene Architekturen besitzen. In der Praxis bedeutet das, dass ein für NVIDIA-GPUs optimiertes OpenCL Programm zwar auf einer ATI/AMD-Karte läuft, aber nicht mit optimaler Geschwindigkeit. Ein weiterer Punkt ist, dass OpenCL aufgrund seiner Vielseitigkeit zu komplexeren Programmcode führt. Beispielsweise werden in C for CUDA Funktionen, die später auf der Grafikkarte ausgeführt werden sollen, bereits beim Kompilieren des Programms in Maschinensprache übersetzt. In OpenCL müssen sie hingegen zur Laufzeit über API-Funktionen kompiliert werden [25], da Grafikkarten verschiedener Hersteller unterschiedliche Maschinensprachen verwenden.

Wie bereits erwähnt, beschränkt sich diese Diplomarbeit auf die CUDA-Architektur, d.h. es wird in Worten und Pseudocode beschrieben wie Algorithmen auf dieser Hardware effizient implementiert werden können. In einigen Fällen ist es zum besseren Verständnis nötig,

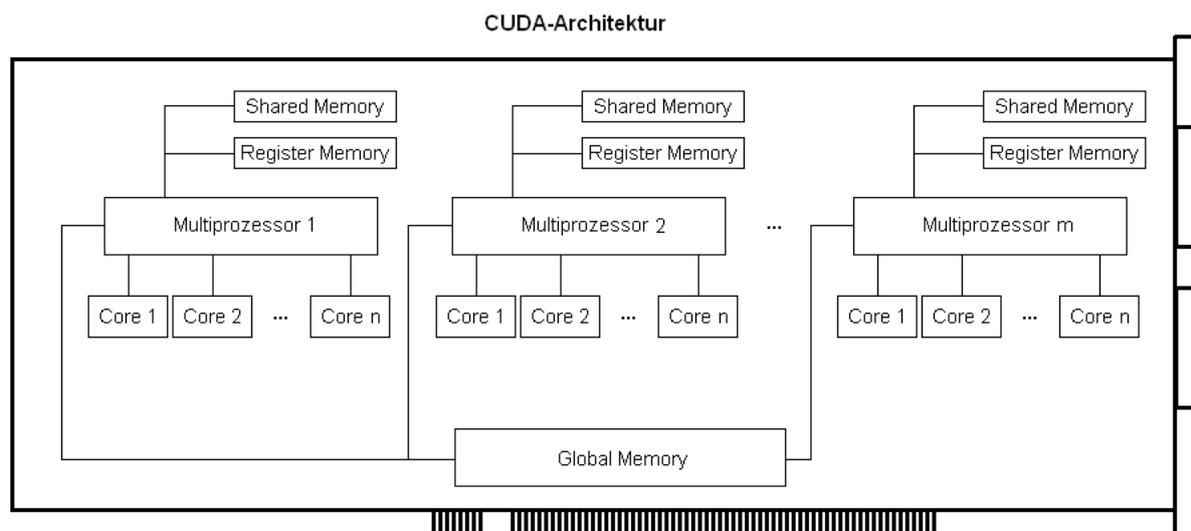


Abb. 42: CUDA-Architektur

einzelne Zeilen von tatsächlichem Code wiederzugeben. In diesem Fall wird C for CUDA verwendet. Es soll jedoch ausdrücklich betont werden, dass derzeit (Mai 2011) unklar ist, ob sich einer der beiden Standards (C for CUDA bzw. OpenCL) durchsetzen kann oder beide parallel bestehen bleiben.

4.1.3 CUDA-Architektur

Bevor zur Implementierung übergegangen wird, ist es notwendig den grundsätzlichen Aufbau der CUDA (Compute Unified Device Architecture)-Architektur zu erklären. Die folgende Einführung kann selbstverständlich nicht alle Einzelheiten umfassen, da dies sowohl den Rahmen der Diplomarbeit sprengen würde als auch didaktisch ungeschickt wäre. Vielmehr werden jene Aspekte genauer betrachtet, die zur späteren Implementierung der Glättungsalgorithmen von besonderer Bedeutung sind. Abb. 42 zeigt eine vereinfachte schematische Darstellung einer CUDA-fähigen Grafikkarte.

Jede CUDA-fähige Grafikkarte enthält eine bestimmte Anzahl von Multiprozessoren (MPs). Eine GeForce 8800 GTS512 besitzt beispielsweise 16 MPs, eine GTX 260 hingegen 24 [23]. Jeder Multiprozessor enthält eine bestimmte Anzahl von Kernen. Bei der GeForce 8800 GTS512 und der GeForce GTX 260 sind beispielsweise 8 Kerne pro Multiprozessor vorhanden [23]. Insgesamt beinhalten die beiden Grafikkarten also respektive 128- bzw. 192-Kerne. Um eine hohe Geschwindigkeit zu erreichen, muss der Programmierer möglichst alle vorhandenen Kerne parallel einsetzen.

Weiters enthält eine CUDA-fähige Grafikkarte mehrere Speicherbausteine. Das Global-Memory ist der Bereich mit der höchsten Kapazität. Eine GeForce 8800 GTS512 enthält beispielsweise 512MB-, eine GeForce GTX 260 sogar 768MB Global-Memory [23]. Alle Kerne können auf das Global-Memory zugreifen, wobei gewisse Einschränkungen beachtet werden müssen. Das Global-Memory besitzt eine relativ langsame Zugriffsgeschwindigkeit und es treten Kollisionen auf, wenn mehrere Kerne gleichzeitig auf das Global-Memory zugreifen. Um diese Probleme zu umgehen, werden zwei weitere Speichertypen eingesetzt, die getrennt für jeden Multiprozessor vorhanden sind. Das Register-Memory wird für Variablen verwendet, wobei nur der jeweilige Core auf die von ihm im Register-Memory angelegten Variablen zugreifen kann. Jedem Core eines Multiprozessors wird also ein eigener exklusiver Bereich im Register-Memory zugewiesen. Das Shared-Memory wird oft verwendet, um Daten zwischen einzelnen Threads auszutauschen. Eine genauere Beschreibung folgt später.

Funktionen, die auf der Grafikkarte ausgeführt werden können, werden Kerne (engl. Kernels) genannt. In *C for CUDA* werden Kernels wie folgt vereinbart [26]:

```
__global__ void nameDesKernels(ParameterListe) {  
    ...  
}
```

Das Schlüsselwort `__global__` kennzeichnet, dass es sich nicht um eine herkömmliche C-Funktion handelt sondern um einen Kernel für die Grafikkarte. Da Kernels keine Werte an den Aufrufer zurückgeben können, werden sie immer als Typ `void` deklariert. Die Parameterliste innerhalb der runden Klammern entspricht den herkömmlichen C-Funktionen. Nach der Deklaration können Kernels wie folgt aufgerufen werden:

```
nameDesKernels<<<1,1>>>(ParameterListe);
```

Die beiden Zahlen innerhalb der spitzen Klammern geben an wie viele Instanzen des Kernels parallel ausgeführt werden sollen. `<<<1, 1>>>` bedeutet, dass nur eine Instanz des Kernels erstellt wird. Diese läuft auf einem Core eines Multiprozessors während alle anderen Cores im Leerlauf sind. Es handelt sich also um sequentielles Programm. An dieser Stelle

soll darauf hingewiesen werden, dass die Stärke der GPU in der Anzahl der Cores liegt und nicht in deren individueller Geschwindigkeit. Dies ist ein fundamentaler Unterschied zu CPUs, die wenige, sehr schnelle Kerne enthalten. Daher sollten nur parallele Programmteile auf die GPU ausgelagert werden, während sequentielle Blöcke auf der CPU verbleiben. Eine Angabe von `<<<1, 1>>>` ist also nicht sinnvoll. Um die parallele Ausführung mehrerer Kernelinstanzen zu erreichen, müssen die Werte innerhalb der spitzen Klammern genauer betrachtet werden: Der erste Parameter definiert die sogenannte *GridSize* (dt. Größe des Gitters). Das Gitter ist immer zweidimensional, also beispielsweise 4x2, 8x8 oder 1024x768 Elemente groß. Wird, wie im obigen Beispiel, ein Skalarwert angegeben so wird die Y-Dimension des Grids auf 1 gesetzt. Die *GridSize* bestimmt wieviele Kernelinstanzen parallel ausgeführt werden. In einem 8x8-Grid laufen beispielsweise 64 (8*8) Instanzen parallel. Logisch gesehen erstellt die Grafikkarte ein Gitter der festgelegten Größe und weist jedem Element des Gitters eine eindeutige Identifikationsnummer zu. Anschließend werden so viele Blöcke (ein konkretes Element des Grids wird als Block bezeichnet) wie möglich auf die einzelnen Cores der Grafikkarte verteilt. Sobald ein Core mit der Abarbeitung seines zugewiesenen Blocks fertig ist, wird ihm der nächste noch nicht ausgeführte Block übergeben, solange bis alle Blöcke abgearbeitet wurden. Die Identifikationsnummer ist notwendig, da alle Blöcke denselben Kernel ausführen. Jeder Block muss also wissen an welcher Position im Grid er sich befindet, um entscheiden zu können, welche Daten gelesen bzw. geschrieben werden. Ohne Identifikation wäre die parallele Ausführung sinnlos, da alle Blöcke exakt dieselbe Tätigkeit durchführen müssten. Jeder Block kann durch Zugriff auf die Variablen `blockIdx.x` und `blockIdx.y` seine Identifikationsnummer (= Position im Grid) bestimmen. Alle Problemstellungen können mit eindimensionalen Grids gelöst werden. Bei der Bearbeitung von Strukturen, die von Haus aus zweidimensional sind, wie beispielsweise Bilddateien, kann der Programmieraufwand durch Einsetzen von zweidimensionalen Grids jedoch oft erheblich reduziert werden. *Abb. 43* (nächste Seite) demonstriert Gridaufbau und Blockzuweisung.

Der zweite Wert innerhalb der Spitzklammern wurde bisher immer auf 1 gesetzt. Er gibt an wie viele Threads pro Block generiert werden. Eigentlich ist dieser Parameter dreidimensional, aber um das Thema nicht zu komplex werden zu lassen wird er in dieser Diplomarbeit als eindimensionaler Parameter behandelt (zwei- und dreidimensionale Threadstrukturen werden in der Praxis ohnehin äußerst selten eingesetzt). Ein Wert von 1 gibt folglich an, dass für jeden Block des Grids ein Thread generiert wird. Jeder Core führt also genau eine Instanz des Kernels aus. Bei höheren Werten werden entsprechend mehr Threads generiert, die alle weiterhin denselben Kernel ausführen. Ein Wert von 32 bedeutet beispielsweise, dass für jeden Block des Grids 32 Threads generiert werden.

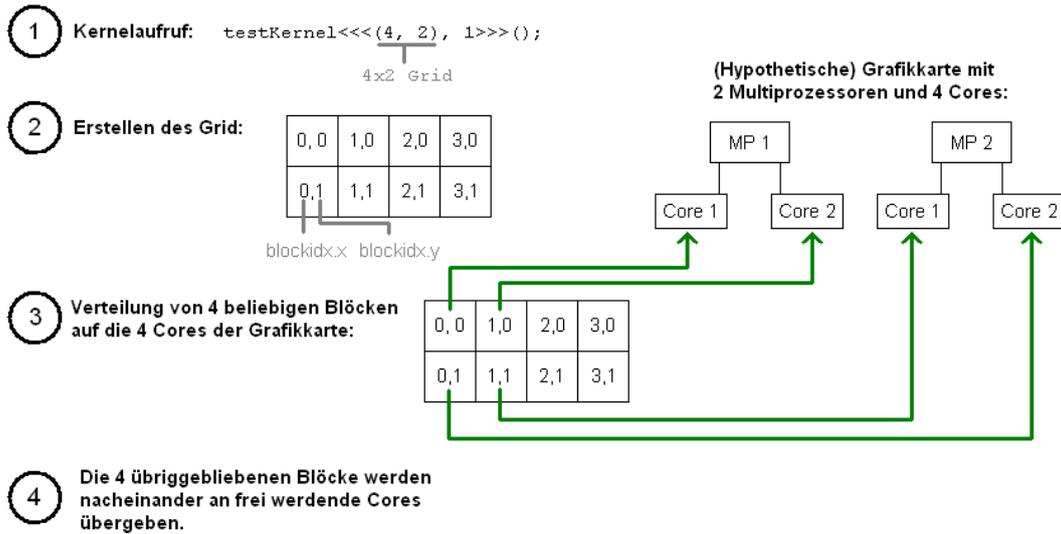


Abb. 43: Gridaufbau und Blockzuweisung in der CUDA-Architektur

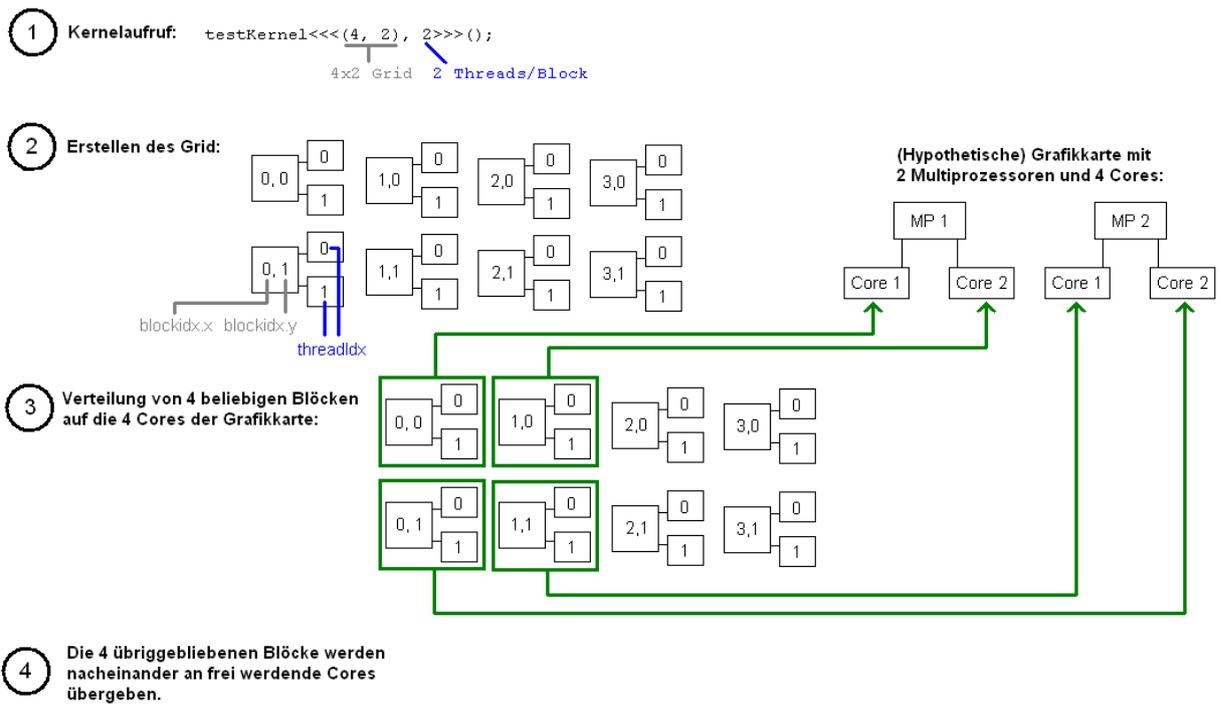
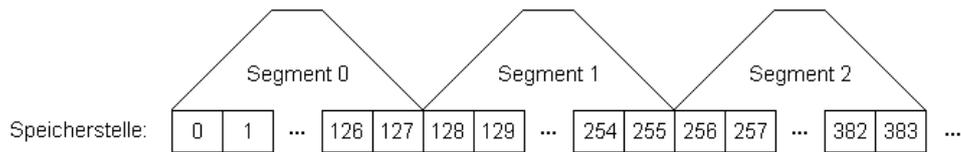


Abb. 44: Threads in der CUDA-Architektur

In diesem Fall führt jeder Core genau 32 Instanzen des Kernels zum gleichen Zeitpunkt aus. Mehrere Threads sind nur sinnvoll, wenn sie durch eine Identifikationsnummer von anderen Threads unterschieden werden können. In C for CUDA kann jeder Thread durch Auslesen der Variable `threadIdx` seine Identifikationsnummer bestimmen. Abb. 44 demonstriert die Verwendung von Threads in der CUDA-Architektur.



Auf einer GeForce 8800 GTS 512 (512MB Global-Memory) ist das Global-Memory beispielsweise in 4.194.304 Segmente unterteilt.

Zugriffsmuster:

- 1 Alle 32 Threads des Warp greifen auf dieselbe Speicherstelle zu.
→ Es wird nur ein Speicherzugriff ausgeführt und das Ergebnis an alle 32 Threads weitergeleitet (Broadcast).
- 2 Alle 32 Threads des Warp greifen auf Speicherstellen im selben Segment zu.
→ Alle 32 Speicherzugriffe können parallel ausgeführt werden.
- 3 Die Speicherzugriffe der 32 Threads verteilen sich auf zwei Segmente.
→ Zuerst werden alle Zugriffe auf das eine Segment parallel durchgeführt, danach alle Zugriffe auf das andere Segment (Serialisierung)
- 4 Die Speicherzugriffe der 32 Threads verteilen sich auf mehr als zwei Segmente.
→ Alle Zugriffe in das selbe Segment werden jeweils parallel ausgeführt

Abb.45: Segmentierung des Global-Memory

Es stellt sich nun die Frage weshalb es sinnvoll ist mehrere Threads pro Core auszuführen. Die Antwort findet sich im Aufbau des Global-Memory. Der gesamte Global-Memorybereich ist in 128 Byte breite Segmente unterteilt. Auf den Inhalt eines Segments kann parallel zugegriffen werden. Auf unterschiedliche Segmente muss hingegen seriell zugegriffen werden. Dieses Verhalten steht eng mit der Threadanzahl in Verbindung: Die GPU fasst jeweils 32 Threads eines Blocks zu einem sogenannten Warp zusammen und führt diese im sogenannten Lockstep aus. Lockstep bedeutet, dass der Instruction Pointer aller 32 Threads des Warps zu jedem Zeitpunkt auf dieselbe Codezeile zeigt. Erst wenn die Codezeile in allen 32 Warps vollständig ausgeführt wurde, kann der Instruction Pointer in allen Threads auf die nächste Codezeile gesetzt werden. Wenn also in einem Kernel ein Speicherzugriff erfolgt, so führen aufgrund der synchronisierten Instruction Pointer immer 32 Threads gleichzeitig einen Speicherzugriff durch. Aufgrund der Segmentierung des Global-Memory können diese Speicherzugriffe parallel ausgeführt werden, wenn sie alle auf dasselbe 128 Byte breite Segment verweisen. Erfolgt hingegen ein Zugriff in unterschiedliche Segmente, so ist eine Serialisierung der Speicherzugriffe notwendig. Im Extremfall könnten die 32 Threads eines Warps auf 32 unterschiedliche Segmente zugreifen. In diesem Fall müssten alle Speicherzugriffe seriell ausgeführt werden. Es ist daher sehr wichtig den Kernel so zu programmieren, dass die 32 Threads eines Warps auf möglichst wenig unterschiedliche Segmente zugreifen. Da die GPU immer Threads mit aufeinanderfolgenden Nummern zu Warps zusammenfasst, sollte der Programmcode so aufgebaut sein, dass die Threads 0 – 31 auf ein Segment zugreifen, Threads 32 – 63 auf ein Segment (aber nicht

notwendigerweise dasselbe wie die Threads 0 – 31), usw. *Abb. 45* zeigt Segmentaufteilung und häufige Zugriffsmuster.

NVIDIA empfiehlt mehr als 32 Threads pro Block zu generieren. Dafür gibt es einen einfachen Grund: Wenn ein Warp „stillsteht“ (weil er beispielsweise auf Speicherzugriff ins Global-Memory wartet), kann oft ein anderer Warp fortgesetzt werden, der in seiner Anweisung keinen Zugriff auf das Global-Memory benötigt. Je höher die Anzahl der Warps umso weniger Coreleistung geht durch stillstehende Warps verloren. Pro Core können maximal 512 Threads generiert werden, wobei zu beachten ist, dass jeder Multiprozessor maximal 1024 Threads verwalten kann. Sollen die 1024 möglichen Threads also gleichmäßig auf die 8 Cores eines Multiprozessors aufgeteilt werden, so müssen pro Core 128 Threads generiert werden. In diesem Fall laufen 4 (128 / 32) Warps pro Core.

Es wurde bereits erwähnt, dass jeder Core einen eigenen Bereich im Register-Memory nutzt. Dieser Bereich wird wiederum unterteilt, so dass jeder Thread einen eigenen Bereich besitzt auf den er exklusiv zugreifen kann. Das Register-Memory wird für Variablen genutzt, die im Thread vereinbart werden. Eine typische Grafikkarte besitzt etwa 32KB Register-Memory pro Multiprozessor [23]. Es ist leicht zu erkennen, dass diese geringe Speichergröße leicht zum Flaschenhals paralleler Implementierungen werden kann: Wird beispielsweise auf einem Multiprozessor die zulässige Maximalzahl von 1024 Threads ausgeführt, so stehen jedem Thread gerade einmal 32 Byte ($32 * 1024 / 1024$) Register-Memory zu Verfügung. Dies entspricht acht 32-Bit Integer Variablen. Da Schleifencounter und andere Konstrukte ebenfalls temporär das Register-Memory belegen, kommt es oft zu der Situation, dass zwangsweise weniger Threads als theoretisch möglich pro Multiprozessor ausgeführt werden müssen, da das Register-Memory sonst überlaufen würde. In der offiziellen CUDA-Dokumentation wird dieses Problem auch „*Register-Pressure*“ genannt [23].

Neben dem Register-Memory gibt es noch einen zweiten lokalen Speicherbereich, das Shared Memory. Jeder Core erhält einen eigenen Bereich im Shared Memory. Im Gegensatz zum Register-Memory wird das Shared-Memory jedoch nicht unter den Threads aufgeteilt, sondern von allen Threads eines Core gemeinsam genutzt. In der Praxis wird das Shared-Memory oft dazu verwendet, Daten zwischen einzelnen Threads (am selben Core) auszutauschen, da ein Austausch über den Umweg Global-Memory deutlich langsamer wäre. Shared-Memory wird angelegt indem einer Variablendeklaration das Schlüsselwort `__shared__` vorangestellt wird (siehe nächste Seite).

```

__global__ void nameDesKernels(ParameterListe) {
    __shared__ int test[128];
}

```

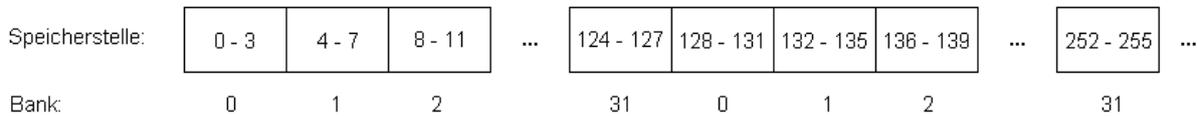


Abb. 46: Banking des Shared-Memory

Im obigen Beispiel wird ein Integer Array `test` angelegt, welches von allen Threads eines Cores gemeinsam benutzt wird. Werden mehrere Cores eines Multiprozessors benutzt, so werden entsprechend viele Kopien des Arrays im Shared Memory angelegt. Jeder Core besitzt also eine eigene Kopie des Arrays zur exklusiven Nutzung.

Wie bereits beim Global-Memory ist es auch bei Zugriffen in das Shared-Memory wichtig, dass die Threads eines Warps in bestimmten Mustern zugreifen, um eine Serialisierung zu vermeiden. Das Shared-Memory ist in 32 Bänke unterteilt, die im 4-Byte Abstand ineinander verzahnt (interleaved) sind (siehe Abb. 46). Auf unterschiedliche Bänke kann parallel zugegriffen werden. Wenn also alle Threads eines Warps auf verschiedene Bänke zugreifen, so können alle 32 Speicheroperationen parallel ausgeführt werden. Andernfalls kommt es zu einer Serialisierung. Ein serialisierter Zugriff in das Shared-Memory ist jedoch im Allgemeinen immer noch schneller als ein paralleler Zugriff ins Global-Memory. CUDA-fähige Grafikkarten besitzen 16KB-48KB Shared-Memory pro Multiprozessor [23].

Da die Grafikkarte nicht direkt auf den Hauptspeicher des Computers zugreifen kann, müssen alle Eingabedaten, die ein Kernel benötigt, vor dessen Aufruf über eine API-Funktion in das Global-Memory der Grafikkarte kopiert werden. Gleichermaßen kann ein Kernel das berechnete Ergebnis nur im Global-Memory der Grafikkarte ablegen. Soll das Ergebnis auf der CPU weiterverarbeitet werden, so muss es zuerst vom Grafikspeicher in den Hauptspeicher kopiert werden. Oft werden folgende Schritte ausgeführt:

1. Global-Memory auf der GPU reservieren (CUDA-API-Funktion: `cudaMalloc`)
2. Ein Array mit den Eingabedaten vom Hauptspeicher in das Global-Memory kopieren (CUDA-API-Funktion: `cudaMemcpy`)
3. Kernel ausführen (das Ergebnis wird in einem Array im Global-Memory abgelegt)
4. Ergebnisarray vom Global-Memory in den Hauptspeicher kopieren (CUDA-API-Funktion: `cudaMemcpy`)

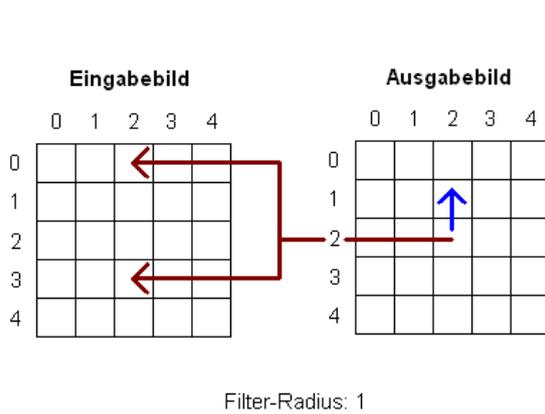
5. Freigeben des allokierten Global-Memory (CUDA-API-Funktion: *cudaFree*)

Da das Kopieren von der/auf die Grafikkarte über den PCI-Express Bus erfolgt, sind diese Operationen relativ langsam. Es ist daher manchmal sinnvoll auch sequentielle Programmteile auf der Grafikkarte auszuführen, um wiederholte Kopiervorgänge zu umgehen.

Nach Studium dieses Kapitels stellt sich der Leser möglicherweise die Frage, wie das Beschriebene umgesetzt werden kann. Aus diesem Grund ist der folgende Abschnitt praktisch orientiert. Er soll demonstrieren wie der in Kapitel 3 vorgestellte $O(1)$ -Mittelwertfilter als CUDA-Kernel implementiert werden kann.

4.2 Mittelwertfilter

Dieser Abschnitt demonstriert die Schritte einer effizienten CUDA-Implementation anhand des in 3.1 (Seite 23) vorgestellten $O(1)$ -Mittelwertfilters. Der Filter besteht aus zwei Teilen. In einem ersten Schritt wird entlang der Zeilen gefiltert, im zweiten Schritt entlang der Spalten. Aufgrund der Assoziativität der linearen Faltung können beide Schritte auch in vertauschter Reihenfolge ausgeführt werden. Für die CUDA-Implementation ist es vorteilhaft zunächst alle Spalten zu filtern. Der Grund dafür wird später offensichtlich. Der in Abschnitt 3.1 vorgestellte Mittelwertfilter ist sequentiell und eignet sich daher nicht direkt für eine CUDA-Implementierung, da ein einzelner CUDA-Core wesentlich langsamer ist als ein CPU-Kern. Der Algorithmus muss daher parallelisiert werden. Bei der Entwicklung von CUDA-Algorithmen versucht man immer eine möglichst hohe Parallelität zu erreichen. Nur wenn mehr Threads erstellt werden als aktuelle Grafikkarte parallel verarbeiten kann, wird der Algorithmus auf zukünftiger Hardware eine Geschwindigkeitssteigerung erzielen. Diese Aussage beruht auf dem in 4.1 vorgestellten Paradigma, welches an dieser Stelle zur besseren Übersichtlichkeit nochmals zusammengefasst werden soll. Die Anzahl der Cores steigt mit zukünftigen Hardwaregenerationen an, während die Rechenleistung eines einzelnen Core nur minimal wächst. Um eine möglichst hohe Parallelität zu erreichen sollte daher im Fall des Mittelwertfilters idealerweise für jeden Bildpunkt ein eigener Thread instanziiert werden. Dies ist jedoch nicht möglich, da der $O(1)$ -Algorithmus auf zuvor berechnete Ausgabewerte zugreift. Würde für jeden Pixel ein eigener Thread erstellt werden so wäre nicht sichergestellt, dass die benötigten Ausgabewerte bereits vorliegen. Bei genauer Betrachtung des Zugriffsmusters (siehe *Abb. 47* nächste Seite) ist zu erkennen, dass in die Berechnung des Mittelwertes einer konkreten Position nur der darüberliegende Ausgabewert derselben Spalte einfließt. Das Ergebnis ist also von den anderen Spalten



Beispiel: Filtern des Pixels (2, 2)

Der Mittelwert wird berechnet indem vom Ausgabewert (2, 1) der Eingabewert (2, 0) entfernt- und der Eingabewert (2, 3) hinzugefügt wird.

- ➔ Der Ausgabewert (2, 1) muss vor der Berechnung bekannt sein.
- ➔ Die Ausgabewerte (2, 1) und (2, 2) können nicht parallel berechnet werden.

Zu Filterung eines Pixels (x, y) wird der Ausgabewert (x, y - 1) benötigt.

- ➔ Der benötigte Ausgabewert liegt in derselben Spalte.
- ➔ Alle Spalten können parallel berechnet werden.

Abb. 47: Abhängigkeiten des vertikalen $O(1)$ -Mittelwertfilters

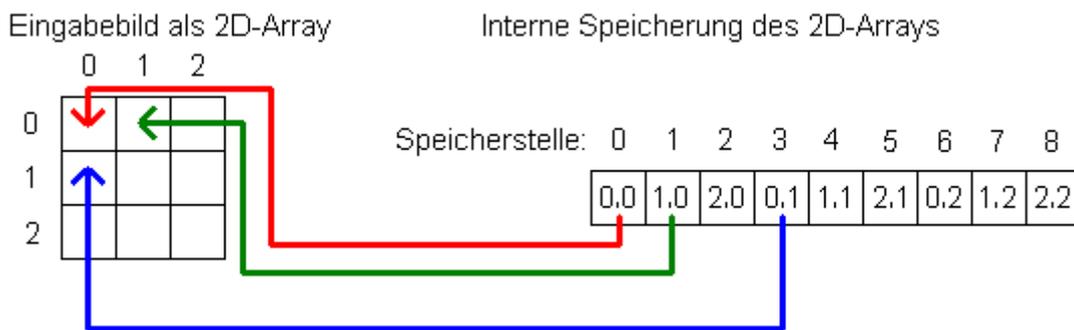


Abb. 48: Row-Major-Speicherung eines zweidimensionalen Arrays

unabhängig. Aus diesem Grund ist es möglich, für jede Spalte einen eigenen Thread zu erstellen, wobei jeder Thread in einer Schleife sequentiell alle Pixel der jeweiligen Spalte filtert.

Um einen CUDA-Kernel implementieren zu können, muss das Format der Eingabedaten bekannt sein und das gewünschte Ausgabeformat festgelegt werden. Im Fall des Mittelwertfilters werden wir ein 8-Bit Graustufenbild als zweidimensionales Array ablegen, wobei die Pixel in Row-Major-Order gespeichert sind (siehe Abb. 48). Das Ausgabeformat soll dem Eingabeformat entsprechen.

Es ist nun möglich eine erste Version des CUDA-Kernels zu implementieren (siehe Listing 1, nächste Seite). Dem Kernel werden zwei Zeiger `*I` und `*J` übergeben, die respektive auf das Eingabebild und Ausgabebild im Global-Memory der Grafikkarte zeigen. Der Parameter `fSize` definiert die Größe des Filterkerns, `width` und `height` Breite und Höhe des zu filternden Bildes. In Zeile 03 ermittelt jeder Thread anhand der Variable `blockIdx.x`

```
00: __global__ mittelwert_y(char *I, char *J,
01:                          int fSize, int width, int height  )
02: {
03:     int x = blockIdx.x;
04:     int sum = 0;
05:
06:     for (int y = 0; y <= fSize/2; y++)
07:         sum += I[y * width + x];
08:     J[x] = sum / fSize;
09:
10:     for (int y = 0; y + 1 < height; y++) {
11:         if (y - fSize/2 > 0)
12:             sum -= I[(y - fSize/2) * width + x];
13:         y++;
14:         if (y + fSize/2 < h)
15:             sum += I[(y + fSize/2) * width + x];
16:         J[y * width + x] = sum / fSize;
17:     }
18: }
```

Kernel-Aufruf:

```
19: mittelwert_y<<<width, 1>>>(I, J, fSize, width, height);
```

Listing 1: CUDA-Kernel des $O(1)$ -Mittelwertfilters

welche Spalte durch ihn gefiltert werden soll. In den Zeilen 06 – 08 wird der oberste Pixel der Spalte gefiltert. Dazu werden in einer Schleife (Zeile 06 – 07) die entsprechenden Pixel der ersten $fSize/2$ Zeilen aufsummiert und anschließend (Zeile 08) der Mittelwert durch Division mit `fSize` ermittelt. Für Positionen außerhalb des Bildbereichs wird der konstante Wert 0 angenommen. In der Hauptschleife (Zeile 10 – 16) wird die zuvor berechnete Summe schrittweise aktualisiert. In jedem Durchlauf wird ein Wert von der Summe subtrahiert (Zeile 11 – 12), die vertikale Filterposition aktualisiert (Zeile 13) und schließlich ein Wert zur Summe addiert (Zeile 14 – 16). Die beiden `if`-Abfragen (Zeile 11 und 14) dienen zur Randbehandlung. Sie stellen sicher, dass für außerhalb des Bildbereichs liegende Positionen der Wert 0 angenommen wird. Die Ausgabe des aktualisierten Mittelwertes erfolgt in Zeile 16. Die in *Listing 1* gezeigte Version verwendet zwar mehrere

Cores, pro Core wird jedoch nur ein Thread instanziiert (siehe `<<<width, 1>>>` in Zeile 19). Bereits in Abschnitt 4.1.3 wurde erwähnt, dass die CUDA-Hardware ihre Stärken nur voll ausspielen kann wenn mehrere Threads pro Core erstellt werden, wobei die Zahl der Threads möglichst ein Vielfaches von 32 sein sollte, da jeweils 32 Threads zu einem Warp zusammengefasst werden. Die optimale Threadanzahl kann oft nur in Versuchsreihen empirisch ermittelt werden. Im Fall des $O(1)$ -Mittelwertfilter wurden bei 128 Threads pro Core die besten Ergebnisse erzielt. *Listing 2* zeigt einen entsprechend modifizierten Mittelwertfilter. Im Vergleich zu *Listing 1* geänderte Zeilen wurden fett markiert.

```

00: __global__ mittelwert_y(    int *I, int *J,
01:                            int fSize, int width, int height    )
02: {
03:     int x = blockIdx.x * 128 + threadIdx;
04:     if (x >= w)
05:         return;
06:
07:     int sum = 0;
08:
09:     for (int y = 0; y <= fSize/2; y++)
10:         sum += I[y * width + x];
11:     J[x] = sum / fSize;
12:
13:     for (int y = 0; y + 1 < height; y++) {
14:         if (y - fSize/2 > 0)
15:             sum -= I[(y - fSize/2) * width + x];
16:         y++;
17:         if (y + fSize/2 < h)
18:             sum += I[(y + fSize/2) * width + x];
19:         J[y * width + x] = sum / fSize;
20:     }
21: }

```

Kernel-Aufruf:

```

22: mittelwert_y<<<ceil(width/128), 128>>>(I, J, fSize, width,
    height);

```

Listing 2: Verbesserter CUDA-Kernel des $O(1)$ -Mittelwertfilter

Der Kernelaufruf (Zeile 22) wurde so abgeändert, dass jeweils 128 Threads pro Block instanziiert werden. Die Anzahl der benötigten Blöcke reduziert sich dementsprechend von `width` auf `width/128`. Da die Spaltenanzahl nicht notwendigerweise ein Vielfaches von 128 ist, muss das Ergebnis aufgerundet werden, um sicherzustellen, dass alle Spalten am rechten Bildrand bei der Filterung ebenfalls berücksichtigt werden. Bei Bildbreiten, die kein Vielfaches von 128 sind, werden also mehr Threads als vorhandene Bildspalten instanziiert. In Zeile 03 ermittelt jeder Thread die ihm zugewiesene Spalte aus den Variablen `blockIdx.x` und `threadIdx`. Wie bereits in Abschnitt 4.1.3 beschrieben, ist es wichtig, dass alle Threads eines Warps auf dasselbe Segment im Global Memory zugreifen, da die Speicherzugriffe sonst serialisiert werden müssen. Da die GPU immer Threads mit aufeinanderfolgenden Nummern zu einem Warp zusammenfasst (hier Threads 0 – 31 im 1. Warp, Threads 32 – 63 im 2. Warp, Threads 64 – 95 im 3. Warp, Threads 96 – 127 im 4. Warp), wurde die Berechnung in Zeile 03 so gewählt, dass Threads mit benachbartem `threadIdx` auch benachbarte Spalten behandeln. Bei Zugriffen auf die erste Bildzeile ist somit sichergestellt, dass alle Threads eines Warps auf dasselbe Segment zugreifen. Ab der zweiten Bildzeile kann es jedoch vorkommen, dass die Threads eines Warps auf zwei verschiedene Segmente zugreifen, da einzelne Segmente jeweils im 128 Byte Abstand beginnen. Nur wenn die Spaltenanzahl des Bildes ein Vielfaches von 128 ist, erfolgen alle Speicherzugriffe des Kernels ohne Serialisierung. Dieses Problem ist jedoch vernachlässigbar, da pro Warp auf maximal zwei unterschiedliche Segmente zugegriffen wird und sich die Anzahl der nötigen Serialisierungen somit in Grenzen hält. Die Notwendigkeit der Serialisierung könnte jedoch umgangen werden, indem jede Bildzeile bei Übertragung in das Global-Memory auf ein Vielfaches von 128 Byte gepadded und der Kernel so angepasst wird, dass Spalten, die durch Padding hinzugekommen sind, nicht gefiltert werden. Zeile 04 und 05 stellen sicher, dass bei Bildern, deren Breite kein Vielfaches von 128 ist, kein Versuch unternommen wird nicht vorhandene Spalten zu filtern (diese Zeile ist wegen der `ceil`-Anweisung in Zeile 22 notwendig).

Der halbe Teil des Mittelwertfilters wurde nun in CUDA implementiert. Um die korrekten Mittelwerte zu berechnen, müssen jedoch nach Filterung der Spalten auch die Zeilen gefiltert werden. Der in Listing 2 gezeigte vertikale Mittelwertfilter kann entsprechend angepasst werden. Eine solche Version wäre jedoch sehr langsam. Es mag unverständlich sein weshalb der Filter in Zeilenrechnung deutlich langsamer sein könnte als die vertikale Variante. Die Antwort liegt in der Serialisierung der Speicherzugriffe eines Warps. Wird der in Listing 2 beschriebene vertikale Mittelwertfilter zum horizontalen Filter konvertiert, so greifen zwangsläufig *alle* Threads eines Warps auf unterschiedliche Segmente zu. Dies bedeutet, dass auf *alle* Speicherstellen seriell zugegriffen werden muss. Diese Behauptung soll nun

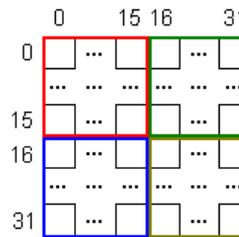
begründet werden: Aufgrund der Row-Major-Speicherung liegen horizontale benachbarte Pixel an benachbarten Speicherstellen (siehe *Abb. 48*, Seite 58) und damit mit hoher Wahrscheinlichkeit im selben 128-Byte Segment. Vertikale benachbarte Pixel liegen jedoch im Gegensatz dazu bei Bildbreiten größer 127 Pixel im Speicher weit auseinander und somit *niemals* im selben 128-Byte Segment (siehe *Abb. 48*, Seite 58). Aufgrund dieses Sachverhalts wäre der vertikale Mittelwertfilter etwa 32x schneller als die horizontale Variante. Da diese Situation nicht befriedigend ist bedient man sich bei der Implementierung des Mittelwertfilters eines Tricks: Zuerst wird das Bild wie gehabt mit einem vertikalen Mittelwertfilter gefiltert. Anschließend wird das Bild als zweidimensionale Matrix interpretiert und transponiert. Aufgrund der Transposition werden Zeilen zu Spalten und Spalten zu Zeilen. Es ist daher möglich, ein zweites Mal den vertikalen Mittelwertfilter aufzurufen. In einem letzten Schritt wird das Ergebnis noch einmal transponiert, um die ursprüngliche Anordnung wiederherzustellen. Dieser Trick wird in einigen Beispielfiltern des NVIDIA GPU Computing SDK verwendet. Das Problem der zusätzlichen zeilenweisen Filterung reduziert sich also auf die Programmierung eines effizienten CUDA-Kernels zur Matrizen-Transposition. Dies ist jedoch keine triviale Aufgabe. Jeder Pixel könnte transponiert werden indem seine x - und y -Koordinaten vertauscht werden, d.h. jeder Thread würde den Pixel an der Position (x, y) des Eingabebildes einlesen und anschließend im Ausgabebild an der Position (y, x) ausgeben. Diese Vorgehensweise ist jedoch sehr ungünstig, da alle Speicherzugriffe eines Warps auf dasselbe Speichersegment zugreifen sollten. Beim Lesen müssen dementsprechend benachbarte Threads auf benachbarte Speicherstellen zugreifen. Dies wäre weiter kein Problem, allerdings schreiben diese Threads danach zwangsläufig in unterschiedliche Segmente. Das folgende Beispiel soll das Problem verdeutlichen: Thread 0 liest Speicherstelle (x, y) , Thread 1 liest Speicherstelle $(x+1, y)$, Thread 2 Speicherstelle $(x+2, y)$. Alle Threads greifen auf benachbarte Speicherstellen zu. Bei der Ausgabe schreibt Thread 0 an Speicherstelle (y, x) , Thread 1 Speicherstelle $(y, x + 1)$, Thread 2 Speicherstelle $(y, x + 2)$. Aufgrund der Row-Major-Speicherung des Bildes greifen alle Threads auf unterschiedliche Segmente zu und die Schreibzugriffe müssen serialisiert werden. Dieses Problem kann gelöst werden indem die Daten blockweise über das Shared-Memory geroutet werden (siehe *Abb. 49*, nächste Seite). [27] beschäftigt sich ausführlich mit der effizienten Matrizen-Transposition in CUDA. Im Rahmen dieser Diplomarbeit wurde von einer Implementierung der CUDA-optimierten Matrizen-Transposition abgesehen, da eine entsprechende Funktion ohnehin Bestandteil des GPU Computing SDKs ist.

Die Laufzeit der im Rahmen dieser Diplomarbeit erstellten CUDA-Version des $O(1)$ -Mittelwertfilters beträgt 2.75ms auf einer GeForce GTX 260 bei einem 8-Bit Bild der

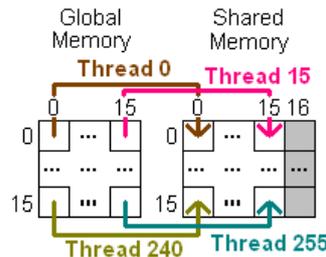
Effiziente Transposition einer 32x32 Matrix:

1 Unterteilung in 16x16 große Blöcke

Block 1 Block 2
Block 3 Block 4

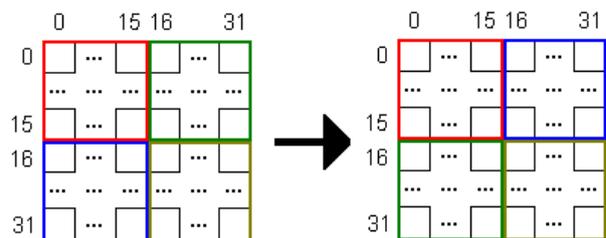
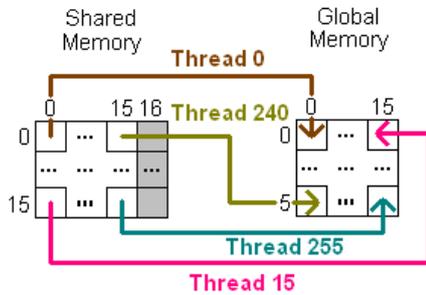


2 Jeder Core bearbeitet einen Block. Pro Core wird ein 272 (17x16) Byte großer Bereich im Shared-Memory allokiert. Die 17. Spalte ist nötig um Bankkonflikte (siehe 4.1.3) zu vermeiden. Pro Core werden 256 (16x16) Threads instanziiert. Jeder Thread kopiert einen Pixel des Blocks in das Shared Memory. Hinweis: Es werden jeweils 32 Threads zu einem Warp zusammengefasst, aber nur jeweils 16 Threads greifen auf aufeinanderfolgende Speicherstellen zu. Daher ist pro Warp eine einzige Serialisierung erforderlich.



Spalte 16 = Padding um Bankkonflikte im Shared-Memory zu vermeiden.

3 Die Threads bauen den transponierten 16x16 Block in der Ausgabematrix auf. Jeder Thread kopiert einen Pixel. Um die Transposition zu erreichen entsprechen die Lesepositionen des Shared-Memory den vertauschten Schreibpositionen aus Schritt 2. Beispiel: Der Thread 240 schrieb in Schritt 2 an Position (0, 15). Dementsprechend liest er in Schritt 3 die Position (15, 0).



Weiters muß jeder Block an die Position (y, x) geschrieben werden, wobei (x, y) den Koordinaten des linken, oberen Pixels des Blocks im Eingabebild entspricht.

Der blaue Block wird beispielsweise von (16, 0) nach (0, 16) verschoben.

Abb. 49: Effiziente Transposition einer Matrix

Auflösung 1280x720. Dieser Wert beinhaltet auch die benötigte Zeit, um das Eingabebild in das Global Memory zu kopieren und das gefilterte Bild zurück in den Hauptspeicher zu transferieren. Der Speed-Up gegenüber der in Kapitel 3 getesteten CPU-Version (22.48ms) beträgt 9.18. Die gebenchmarkte Version des CUDA-Mittelwertfilters enthält zusätzliche Optimierungen, die aus Platzgründen nicht in dieser Diplomarbeit beschrieben werden.

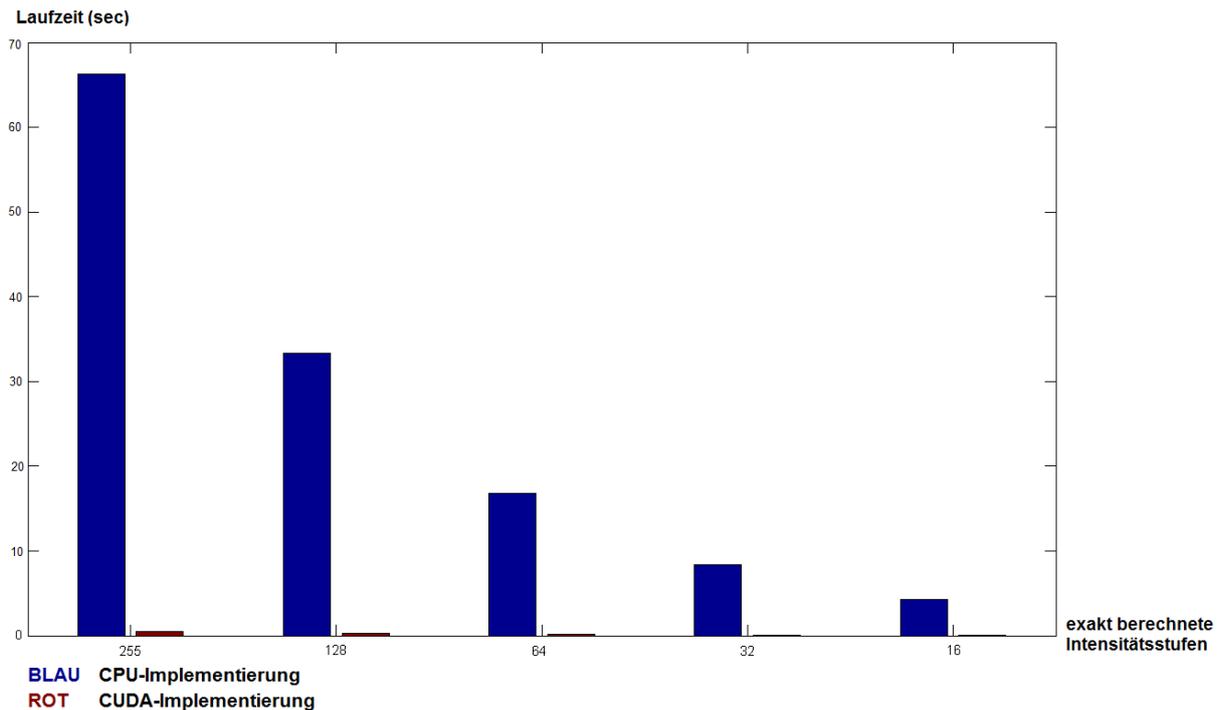


Abb. 50: CPU-Version von Yangs $O(1)$ Bilateral Filter (blau),
CUDA-Implementierung von Yangs $O(1)$ Bilateral Filter (rot)

4.3 Andere Filter

Auf die Besprechung der CUDA-Kerne weiterer Filter wird in dieser Diplomarbeit aus Platzgründen verzichtet. Die Algorithmen der in Kapitel 3 vorgestellten Glättungsfilter können jedoch allesamt leicht parallelisiert werden und die Gewährleistung eines günstigen Zugriffsmusters in das Global-Memory ist nicht schwieriger als beim vorgestellten Mittelwertfilter. Der im Rahmen dieser Diplomarbeit implementierte Guided Image Filter benötigte auf einer GeForce 260-Karte 8.79ms zur Filterung eines 8-Bit Bildes der Auflösung 1280x720. Dieser Wert beinhaltet auch die benötigte Zeit um Eingabebild und Guidance Image in das Global Memory zu kopieren, sowie das Zurückkopieren des gefilterten Bildes in den Hauptspeicher. Im Vergleich zur sequentiellen CPU-Version (87.89ms) entspricht dies einem Speedup von 10. Abb. 50 stellt CPU- und CUDA-Version des $O(1)$ Bilateral Filter von Yang et al. gegenüber. Die CPU-Version stammt von Website der Paper Autoren, die CUDA-Version wurde im Rahmen dieser Diplomarbeit erstellt. Die CUDA-Implementierung erreichte, je nach Approximation, Geschwindigkeiten zwischen 477.82ms (alle 255 Intensitätsstufen werden exakt berechnet) und 35.11ms (16 Intensitätsstufen werden exakt berechnet).

Interessante wissenschaftliche Publikationen zum Thema GPU-Programmierung wurden beispielsweise von [28], [29], [30], [31], [32], [33], [34], [35] und [36] verfasst. Das Paper „Debunking the 100X GPU vs. CPU Myth“ [37], welches von Intel nahestehenden

Wissenschaftlern verfasst wurde, kommt zum Ergebnis, dass eine vergleichsweise günstige NVIDIA GTX 280-Grafikkarte parallele Algorithmen im Schnitt „nur“ 2.55x schneller als ein deutlich teurerer Intel Core i7 960 ausführt (in einigen Marketingpublikationen führender Grafikkartenhersteller werden hingegen beeindruckende 100fache Geschwindigkeitssprünge versprochen). Dies wird unter anderem damit begründet, dass herkömmliche Prozessoren einen deutlich größeren Cache vorweisen können als aktuelle GPUs und somit viele Zugriffe in den langsamen Hauptspeicher durch Cache-Hits vermieden werden können. Hingegen erfolgen bei CUDA-Grafikkarten, aufgrund der kleinen Cache-Größe sowie der deutlich höheren Zahl paralleler Threads, im Allgemeinen viele gleichzeitige Zugriffsversuche in den Speicher, wodurch es zu Kollisionen und Serialisierungen kommt. Die Laufzeiten, der im Rahmen dieser Diplomarbeit implementierten Glättungsalgorithmen, wurden unter anderen Bedingungen gemessen, da bei den CPU-Implementierungen kein hoher Optimierungsaufwand betrieben wurde. Insbesondere wurde nur ein einzelner Core- und keine SIMD-Instruktionen (Single Instruction Multiple Data) eingesetzt. Auch die CPU-Implementation des $O(1)$ Bilateral Filter von Yang et al., welche der Homepage der Paper Autoren entnommen wurde, weist nur geringe Optimierungen auf.

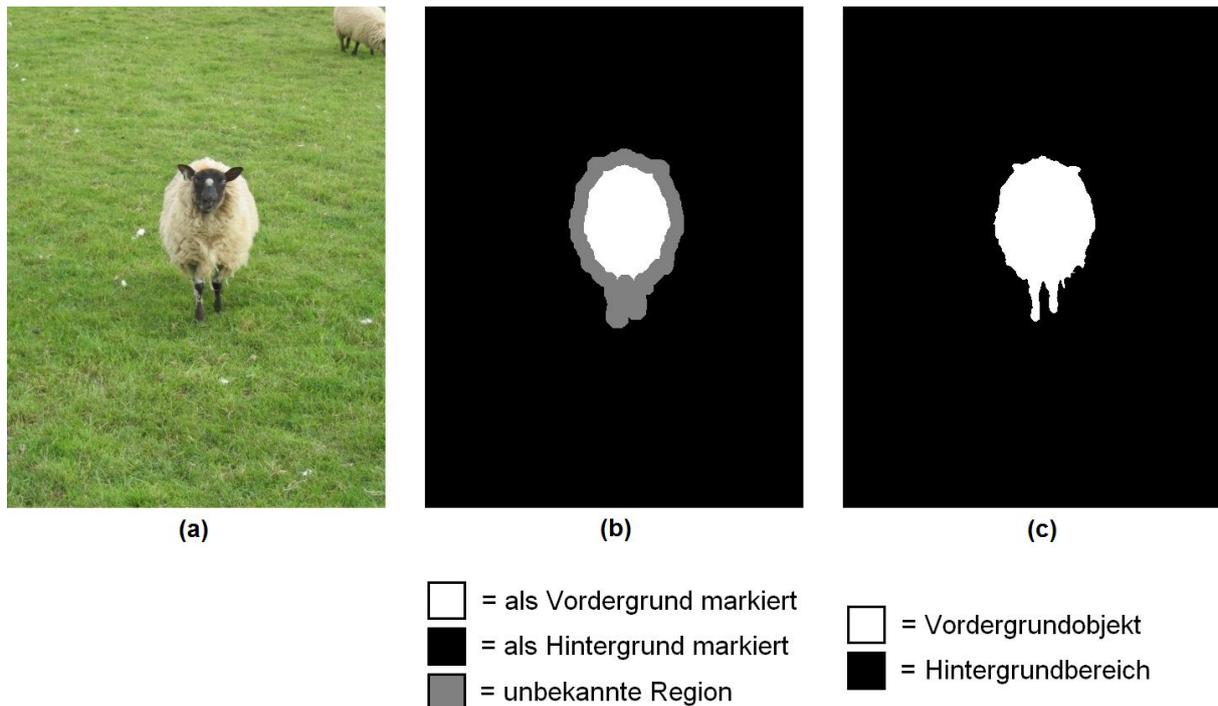


Abb. 51:

Interaktive Bildsegmentierung
 (a) zu segmentierendes Bild
 (b) vom Benutzer bereitgestellte Information
 (c) Ausgabe des Klassifizierungsalgorithmus

5. Interaktive Bildsegmentierung

5.1 Motivation

Anhand des Beispiels „Interaktive Bildsegmentierung“ wird demonstriert wie eine Problemstellung aus der Praxis mit Hilfe kantenerhaltender Glättungsfiler gelöst werden kann. Im Gegensatz zur automatischen Bildsegmentierung muss bei der interaktiven Bildsegmentierung Information durch den Benutzer bereitgestellt werden. Üblicherweise legt der Anwender für eine gewisse Anzahl von Bildpunkten fest, ob der jeweilige Punkt Teil des zu segmentierenden Objekts ist oder zum Hintergrund gehört (siehe *Abb.51(b)*). Anschließend trifft der Segmentierungsalgorithmus für jeden unbekanntes Pixel eine binäre Entscheidung zwischen Vordergrundobjekt und Hintergrundbereich, welche in einer Binärkarte festgehalten wird (siehe *Abb.51(c)*).

5.2 Algorithmus

Um die Vorteile einer neuen Lösung zu erkennen, müssen die Probleme des alten Ansatzes bekannt sein. In diesem Sinne wird in dieser Diplomarbeit zunächst ein einfacher

Segmentierungsalgorithmus vorgestellt und anschließend untersucht wie dessen Nachteile durch Integration kantenerhaltender Glättungsfiler beseitigt werden können.

Einfache Segmentierungsalgorithmen arbeiten häufig unter der Annahme, dass die Farbinformation eines Bildpunktes ausreicht, um zwischen Vordergrundobjekt und Hintergrundbereich zu unterscheiden. Ein naiver Ansatz könnte daher alle unbekanntes Bildpunkte, deren Farbe mit einem Punkt im vom Anwender festgelegten Vordergrundbereich übereinstimmt, als Vordergrund klassifizieren und analog mit dem Hintergrund verfahren. Diese Vorgehensweise hat jedoch einen offensichtlichen Nachteil. Aufgrund der hohen Farbanzahl natürlicher Bilder wird im Allgemeinen für die überwiegende Mehrheit der unbekanntes Pixel in keinem der vom Benutzer festgelegten Bereiche eine exakte farbliche Übereinstimmung registriert. Um dieses Problem zu minimieren, wird häufig in einem quantisierten Farbraum gearbeitet. Da durch die Quantisierung farblich ähnliche Pixel in einem Wert zusammengefasst werden, wird für eine hohe Zahl unbekanntes Pixel eine exakte farbliche Übereinstimmung mit zumindest einem Pixel aus dem vom Benutzer definierten Bereich gegeben sein. Im konkreten Fall des 8-Bit RGB-Farbraums könnte der $256 \times 256 \times 256$ Farben umfassende Würfel beispielsweise durch Quantisierung zu einem $32 \times 32 \times 32$ Würfel reduziert werden, wodurch sich die mögliche Farbanzahl von etwa 16.7 Millionen auf 32.768 verringert. Durch die Quantisierung wird jedoch ein weiteres Problem des naiven Ansatzes offensichtlich: In vielen Fällen besitzen Pixel des vom Benutzer gewählten Vordergrundbereichs nach der Quantisierung identische Farbwerte mit einem oder mehreren Pixel des gewählten Hintergrunds. Um dennoch eine Entscheidung treffen zu können, wird dieses Problem oft durch den Aufbau von Histogrammen umgangen. In einem Vordergrundhistogramm werden die quantisierten Farbwerte der Pixel im vom Benutzer festgelegten Vordergrundbereich festgehalten. Analog dazu wird ein Hintergrundhistogramm aufgebaut. Bei der anschließenden Klassifizierung entscheidet der Algorithmus dann auf Vordergrund, wenn das jeweilige Bin des Vordergrundhistogramms einen höheren Wert als das entsprechende Hintergrundhistogramm-Bin aufweist. Im umgekehrten Fall wird der Bildpunkt als Hintergrund klassifiziert.

Abb. 52(a) zeigt ein mit dem vorgestellten Algorithmus segmentiertes Vordergrundobjekt. Es ist zu erkennen, dass die Qualität des Ergebnisses nicht besonders hoch ist. In [38] wurde festgestellt, dass Histogramm-Bins nicht direkt zur Klassifikation herangezogen werden sollten. Stattdessen wird empfohlen, mit Hilfe der Histogramm-Bins, ein Cost-Volume aufzubauen und dieses anschließend durch kantenerhaltende Filterung, unter Verwendung des zu segmentierenden Bildes als Guidance Image, zu glätten. Das Cost-Volume, ein zweidimensionales Floating-Point Array, dessen Dimension dem zu segmentierenden Bild

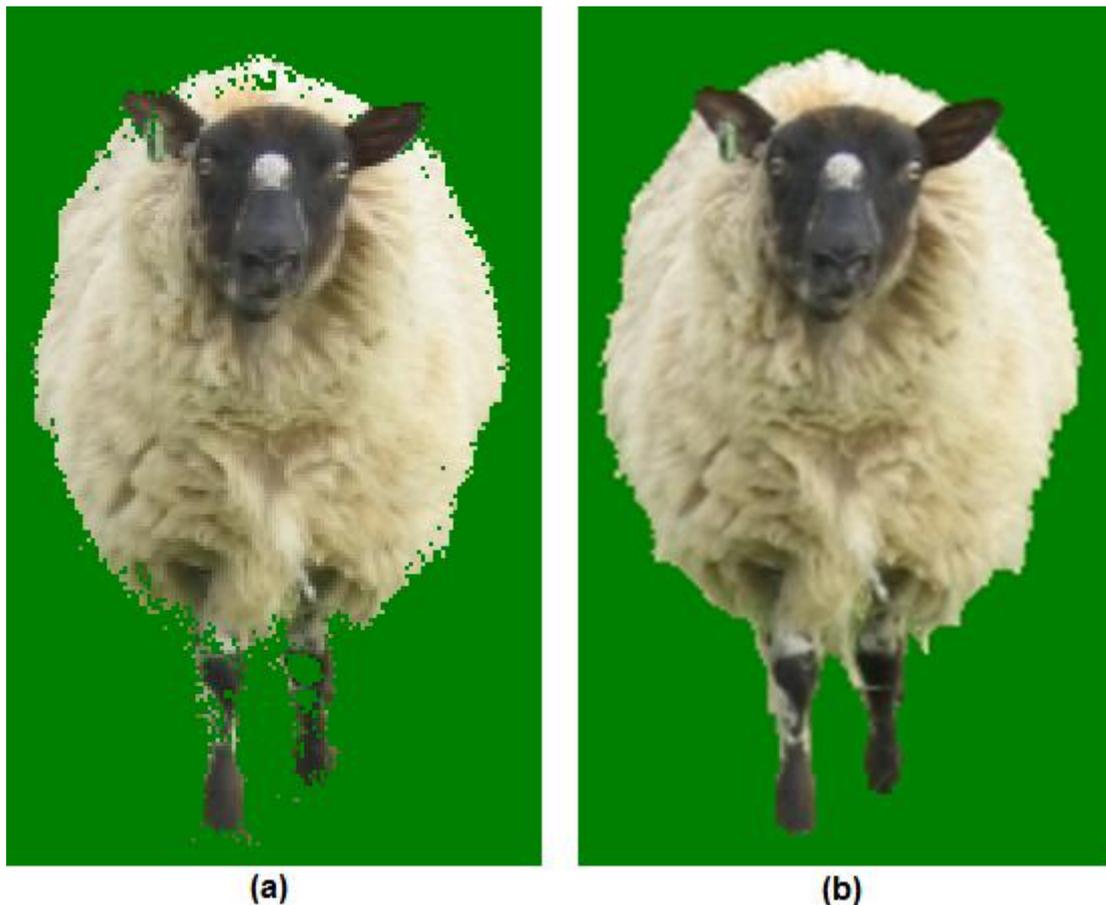


Abb. 52: (a) Segmentierung ohne Cost-Volume
(b) Segmentierung durch Glättung des Cost-Volumes mit Guided Image Filter

entspricht, zeigt für jeden Bildpunkt mit welcher Wahrscheinlichkeit er Teil des Vordergrundobjekts ist. Die Wahrscheinlichkeiten können direkt aus den beiden Histogrammen ermittelt werden:

$$CostVolume(x, y) = \frac{histFG[farbe]}{histFG[farbe] + histBG[farbe]} \quad (13)$$

x und y sind die Koordinaten des Bildpunktes dessen Vordergrundwahrscheinlichkeit berechnet werden soll, $histFG$ und $histBG$ respektive Vorder- und Hintergrundhistogramm, $farbe$ der quantisierte Farbwert des Bildpunktes (x, y) im zu segmentierenden Bild.

Beim Aufbau des Cost-Volumes ist zu beachten, dass die Wahrscheinlichkeiten jener Pixel, die im vom Benutzer festgelegten Vorder- bzw. Hintergrundbereich liegen, nicht mit Formel 13 berechnet werden dürfen. Die entsprechenden Wahrscheinlichkeiten werden stattdessen auf 1.0 (Pixel im Vordergrundbereich) bzw. 0.0 (Pixel im Hintergrundbereich) gesetzt.

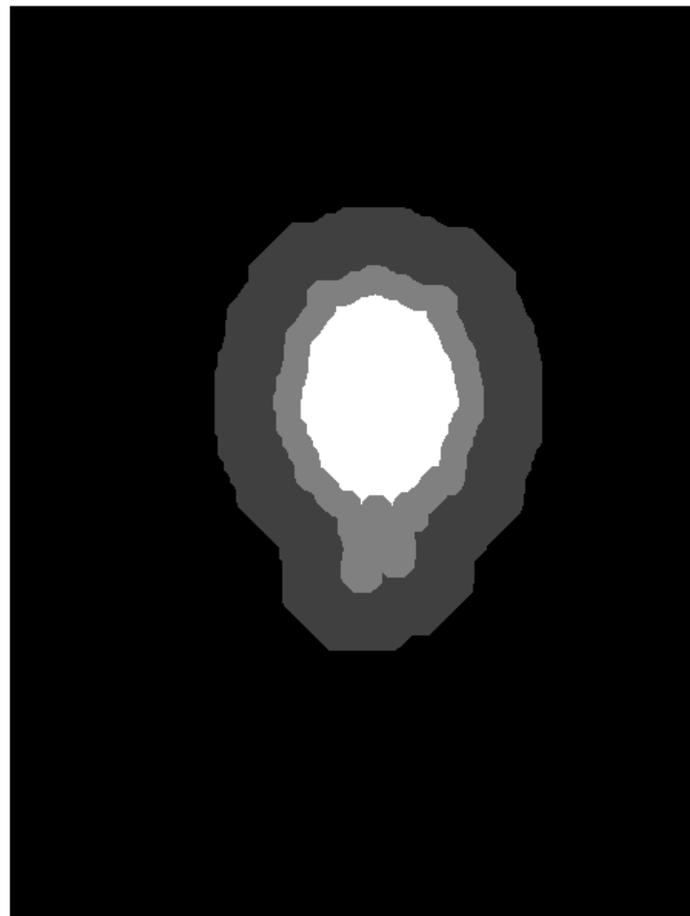
Anschließend wird das Cost-Volume gefiltert und alle Positionen, deren Wahrscheinlichkeiten im gefilterten Cost-Volume über 0.5 liegen, werden dem Vordergrundobjekt zugeordnet (siehe *Abb. 52(b)*, Seite 68). Der Aufbau des Cost-Volumes führt nicht direkt zu einer Steigerung der Segmentierungsqualität. Erst durch die anschließende Filterung wird ein qualitativ hochwertiges Ergebnis erreicht, da die kantenerhaltende Glättung zu einer Reduktion des Rauschanteils führt und die Segmentierung somit robuster wird. Im Rahmen der Evaluierung (siehe Abschnitt 5.3) wurde empirisch ermittelt, welcher Filter für die Glättung des Cost-Volumes am Besten geeignet ist.

In der Praxis werden beim Aufbau des Hintergrundhistogramms oft nicht alle vom Benutzer als Hintergrund klassifizierten Pixel herangezogen, sondern nur jene die sich in räumlicher Nähe zum unbekanntem Bereich befinden. Dies beruht auf der Annahme, dass Hintergrundinformationen im unbekanntem Bereich farblich sehr stark dem naheliegenden, vom Benutzer als Hintergrund markierten, Pixel gleichen. Weiters kann der in diesem Abschnitt vorgestellte Algorithmus mehrmalig iteriert werden, indem die ausgegebene Binärkarte einer Iteration als Eingabe der folgenden Iterationen verwendet wird (anstatt der vom Benutzer festgelegten Wahrscheinlichkeiten wird ab der zweiten Iteration die Binärkarte der vorhergehenden Iteration eingesetzt). Auf diese Weise kann das Ergebnis in vielen Fällen Schritt für Schritt verbessert werden. Der beschriebene Algorithmus beruht auf den Ausführungen in [38].

5.3 Evaluierung

Ziel der folgenden Evaluierung ist es festzustellen, ob bei Segmentierung mit dem in Abschnitt 5.2 vorgestellten Algorithmus signifikante Qualitätsunterschiede zwischen den kantenerhaltenden Filtern (Joint Bilateral Filter, Joint Bilateral Median Filter, Guided Image Filter) bestehen. Zusätzlich wurden auch die weniger geeigneten Filter ohne Guidance Image (Mittelwertfilter, Gauß-Filter, Median-Filter, Bilateral-Filter) evaluiert, um deren Qualitätsverlust zu belegen. Insbesondere soll gezeigt werden, dass eine kantenerhaltende Funktion ohne Guidance Image, wie sie der Bilateral-Filter bietet, nicht ausreichend ist. Um einen Bezug zu derzeitigen State-Of-The-Art Algorithmen herzustellen wurde auch der Grab-Cut Algorithmus [39] in die Evaluierung einbezogen. In diesem Abschnitt wird daher auch festgestellt, welche qualitativen Unterschiede zwischen den vorgestellten kantenerhaltenden Filtern mit Guidance Image und Grab-Cut bestehen.

Jeder der acht getesteten Segmentierungsalgorithmen wird auf 50 Bilder der Middlebury-Datenbank [40][56] angewandt. Zusätzlich zu den eigentlichen Fotos stehen jedem

**Lasso-Trimap**

	= Vordergrundpixel		= Hintergrundpixel, der zum Aufbau des Hintergrundmodells verwendet wird
	= Unbekannter Bereich		= Hintergrundpixel, der nicht zum Aufbau des Hintergrundmodells verwendet wird

Abb. 53: Lasso Trimap aus der Datenbank

Algorithmus die entsprechenden Lasso-Trimaps der Datenbank zu Verfügung (siehe *Abb. 53*), welche die für die interaktive Segmentierung benötigte a-priori Information enthalten (obwohl die Karten vier verschiedene Werte enthalten werden sie oft Trimaps genannt).

Die Anzahl der Iterationen wurde bei allen getesteten Algorithmen, abgesehen von Grab-Cut, auf 5 fixiert. Bei Grab-Cut wurden lediglich drei Iterationen durchgeführt, da in diesem Fall genauere Ergebnisse erzielt wurden. Für jede errechnete Segmentierung wurde die Fehlerrate (error rate) zu den Ground-Truth Daten der Datenbank ermittelt:

$$errorRate(img) = \frac{\#_Anzahl_unterschiedliche_Pixel}{\#_Pixel_mit_Intensität_128_in_Lasso_Trimap} \cdot 100 \quad (14)$$

Im Zähler wird die Anzahl der Pixel ermittelt, die sich in errechneter Segmentierung und Ground Truth genau um den Wert 1 unterscheiden, wobei zuvor beide Bilder in den Intensitätsbereichbereich 0...1 skaliert werden müssen. Da die Ground-Truth Bilder der Datenbank auch unbekannte Werte enthalten können (welche in der Ground-Truth einen Wert von 0.5 haben), werden alle Intensitätsunterschiede <1 ignoriert. Im Nenner wird die Anzahl der Pixel ermittelt, die in der gegebenen Lasso-Trimap als unbekannt definiert sind. Somit gibt die Fehlerrate den prozentuellen Anteil der missklassifizierten Pixel in der unbekanntem Trimap Region wieder. In dieser Diplomarbeit wird die durchschnittliche Fehlerrate über alle 50 Bilder zur Bestimmung der Qualität eines Algorithmus herangezogen.

Bei auf Filterung basierenden Algorithmen hängt die Qualität der Segmentierung stark von den gewählten Filterparametern ab. In der folgenden Diagrammen (siehe *Abb. 54 – 56* auf den folgenden Seiten) werden jeweils die Fehlerraten für jenes Parameterset gelistet, bei dem die durchschnittliche Fehlerrate über alle 50 Bilder am Geringsten war. Grab-Cut und Guided Image Filter lieferten die genauesten Segmentierungen. Im gewählten Testset beträgt die durchschnittliche Fehlerrate des Grab-Cut Algorithmus 0.0554. Der Guided Image Filter weist hingegen eine Fehlerrate von 0.0568 auf. Es kann also festgehalten werden, dass im Durchschnittsfall keine signifikanten Qualitätsunterschiede zwischen den beiden Algorithmen erkennbar waren. Bei einzelnen Bildern des Testsets kam es jedoch vor, dass einer der beiden Algorithmen deutlich überlegen war. Beispielsweise besitzt Grab-Cut bei „person5.bmp“ (siehe *Abb. 61*, Seite 79) eine recht hohe Fehlerrate von 0.1563. Bei diesem Bild ist das Ergebnis des Guided Image Filters (Fehlerrate: 0.0176) deutlich näher an der Ground-Truth. Umgekehrt war bei „Memorial.bmp“ (*Abb. 62*, Seite 80) die Fehlerrate des Guided Image Filter (0.1035) viel höher als die des Grab-Cut Algorithmus (0.0741). Joint Bilateral Filter (durchschnittliche Fehlerrate: 0.0673) und Joint Bilateral Median Filter (0.0690) lieferten ebenfalls gute Ergebnisse. Im Vergleich zu Grab-Cut und Guided Image Filter waren die Fehlerraten jedoch im Durchschnitt signifikant höher. Der bilaterale Filter ohne Guidance Image besitzt eine durchschnittliche Fehlerrate von 0.0710 und ist daher nicht besonders für die Segmentierungsaufgabe geeignet. Mittelwert-Filter, Gauß-Filter und Median-Filter besitzen respektive Fehlerraten von 0.0747, 0.0733 und 0.0746. Wie zu erwarten war, sind diese Filter ebenfalls nicht zur Segmentierung geeignet. Zusammenfassend kann gesagt werden, dass der Guided Image Filter, im evaluierten Testset, die Qualität des Grab-Cut Algorithmus erreichen konnte, wobei der Guided Image Filter den Grab-Cut Algorithmus hinsichtlich der Verarbeitungsgeschwindigkeit selbstverständlich deutlich übertrifft. Weitere beispielhafte Resultate finden sich in *Abb. 57 – 62*.

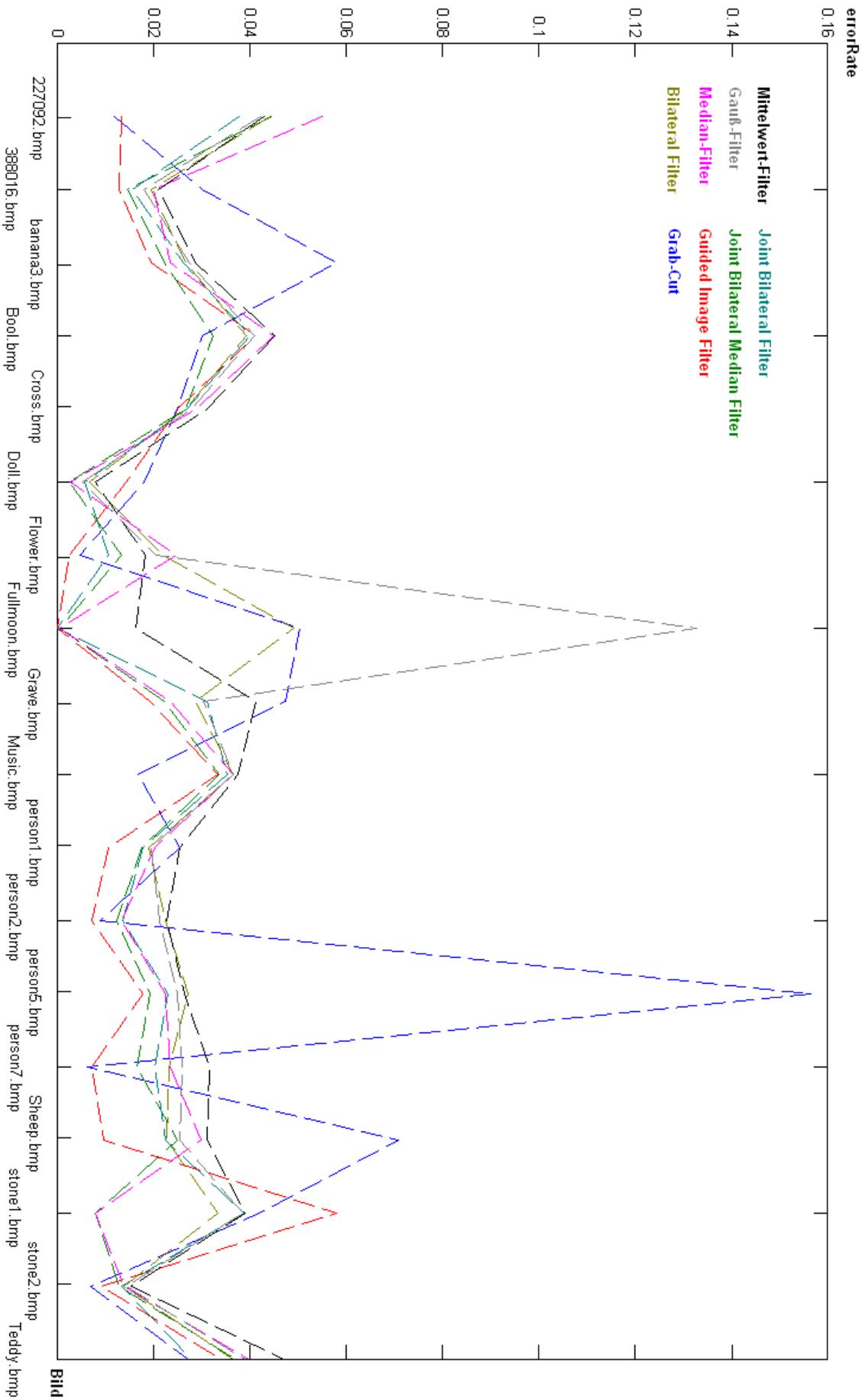


Abb. 54: Segmentierung mit Cost-Volume, Bilder 1 - 18

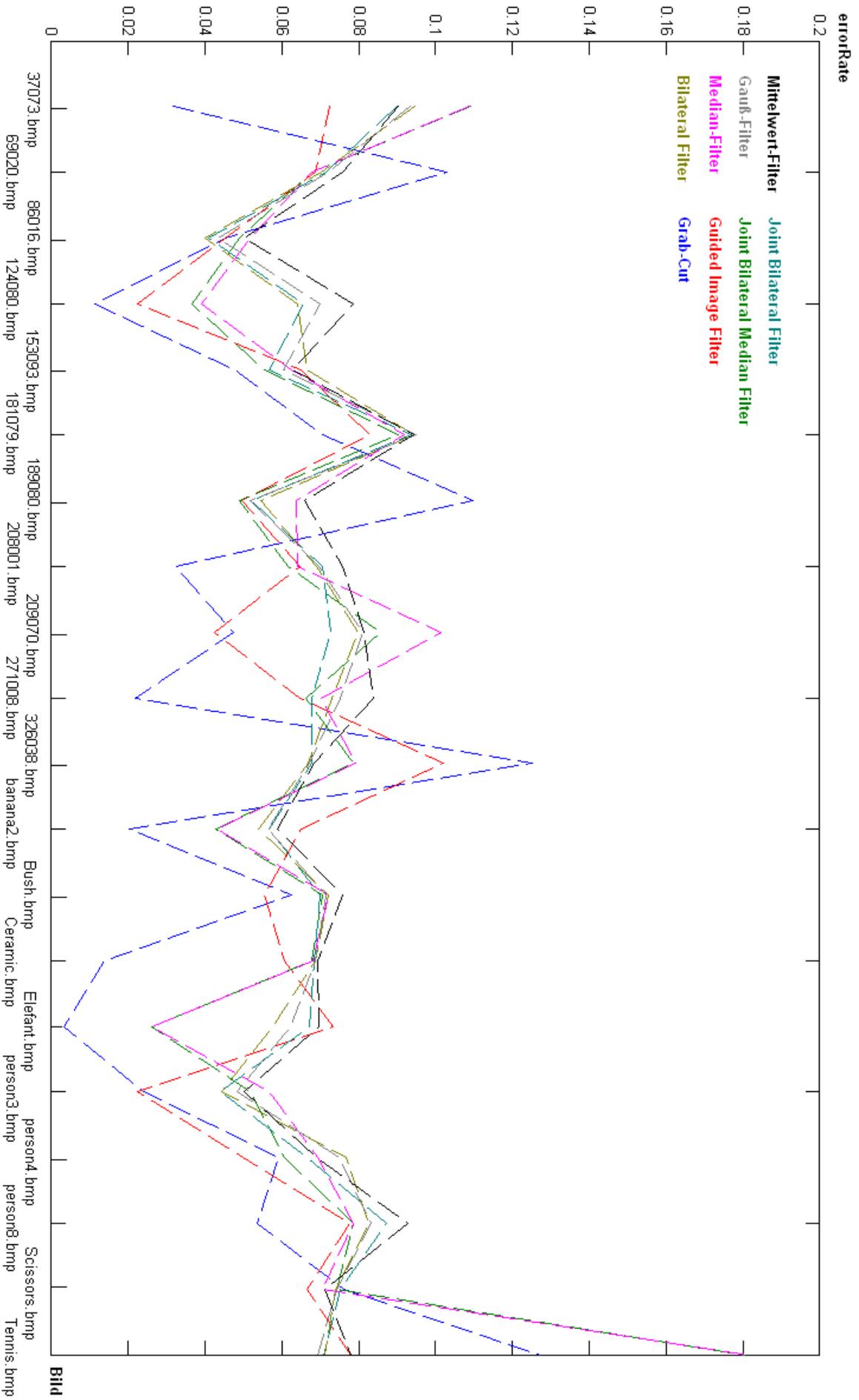


Abb. 55: Segmentierung mit Cost-Volume, Bilder 19 - 38

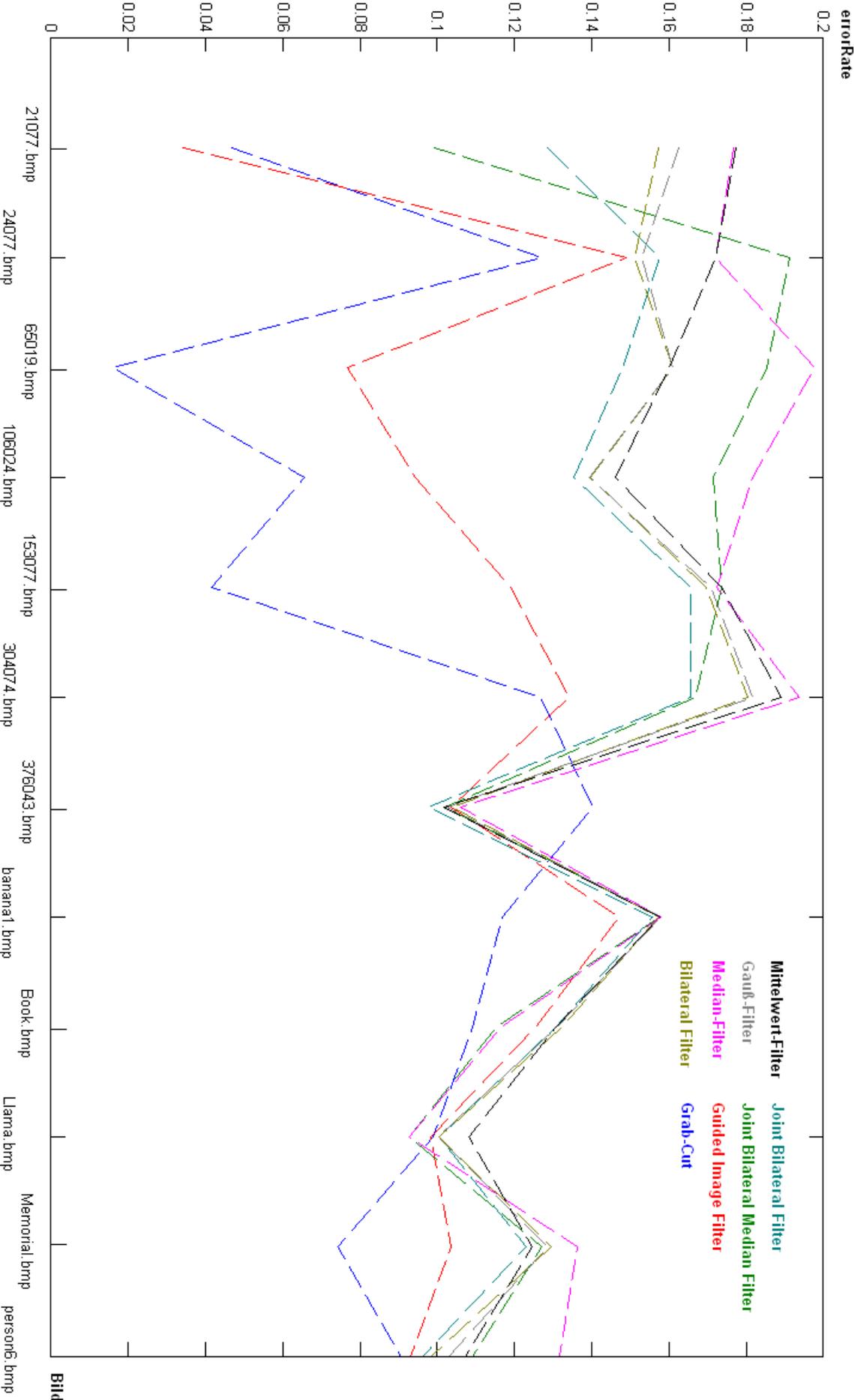


Abb. 56: Segmentierung mit Cost-Volume, Bilder 39 - 50



Abb. 57: Segmentierung von „Flower“
 (a) Zu segmentierendes Bild
 (b) Lasso-Trimap
 (c) Joint Bilateral Filter
 (d) Joint Bilateral Median Filter
 (e) Guided Image Filter
 (f) Grab-Cut

Fehlerrate: 0.0104
 Fehlerrate: 0.0132
 Fehlerrate: 0.0023
 Fehlerrate: 0.0045

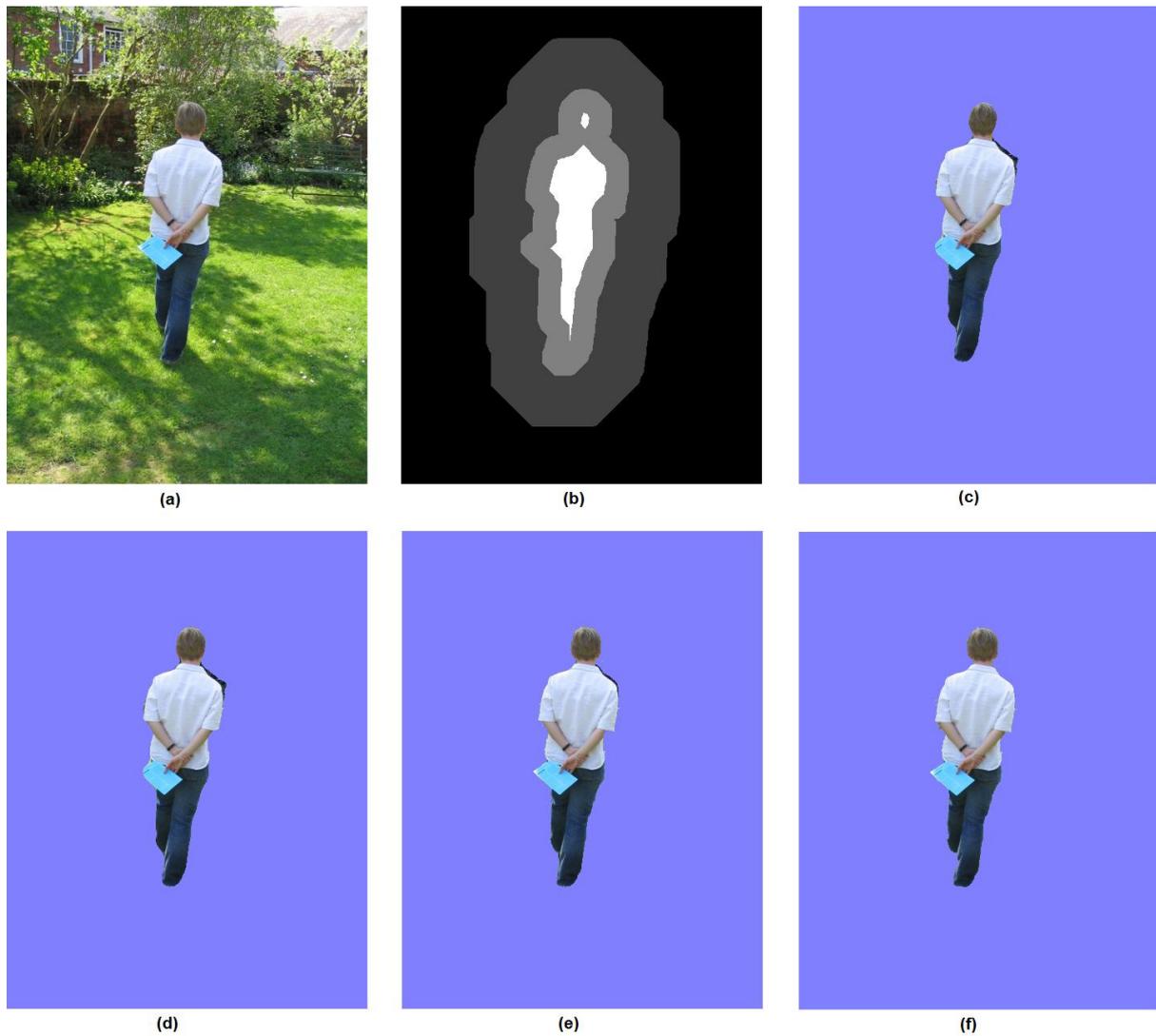
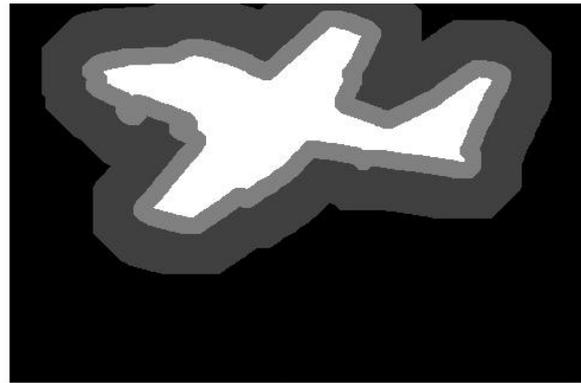


Abb. 58: Segmentierung von „person7“
 (a) Zu segmentierendes Bild
 (b) Lasso-Trimap
 (c) Joint Bilateral Filter
 (d) Joint Bilateral Median Filter
 (e) Guided Image Filter
 (f) Grab-Cut

Fehlerrate: 0.0202
 Fehlerrate: 0.0163
 Fehlerrate: 0.0071
 Fehlerrate: 0.0061



(a)



(b)



(c)



(d)



(e)



(f)

Abb. 59: Segmentierung von „37073“
 (a) Zu segmentierendes Bild
 (b) Lasso-Trimap
 (c) Joint Bilateral Filter
 (d) Joint Bilateral Median Filter
 (e) Guided Image Filter
 (f) Grab-Cut

Fehlerrate: 0.0901
 Fehlerrate: 0.1092
 Fehlerrate: 0.0723
 Fehlerrate: 0.0317

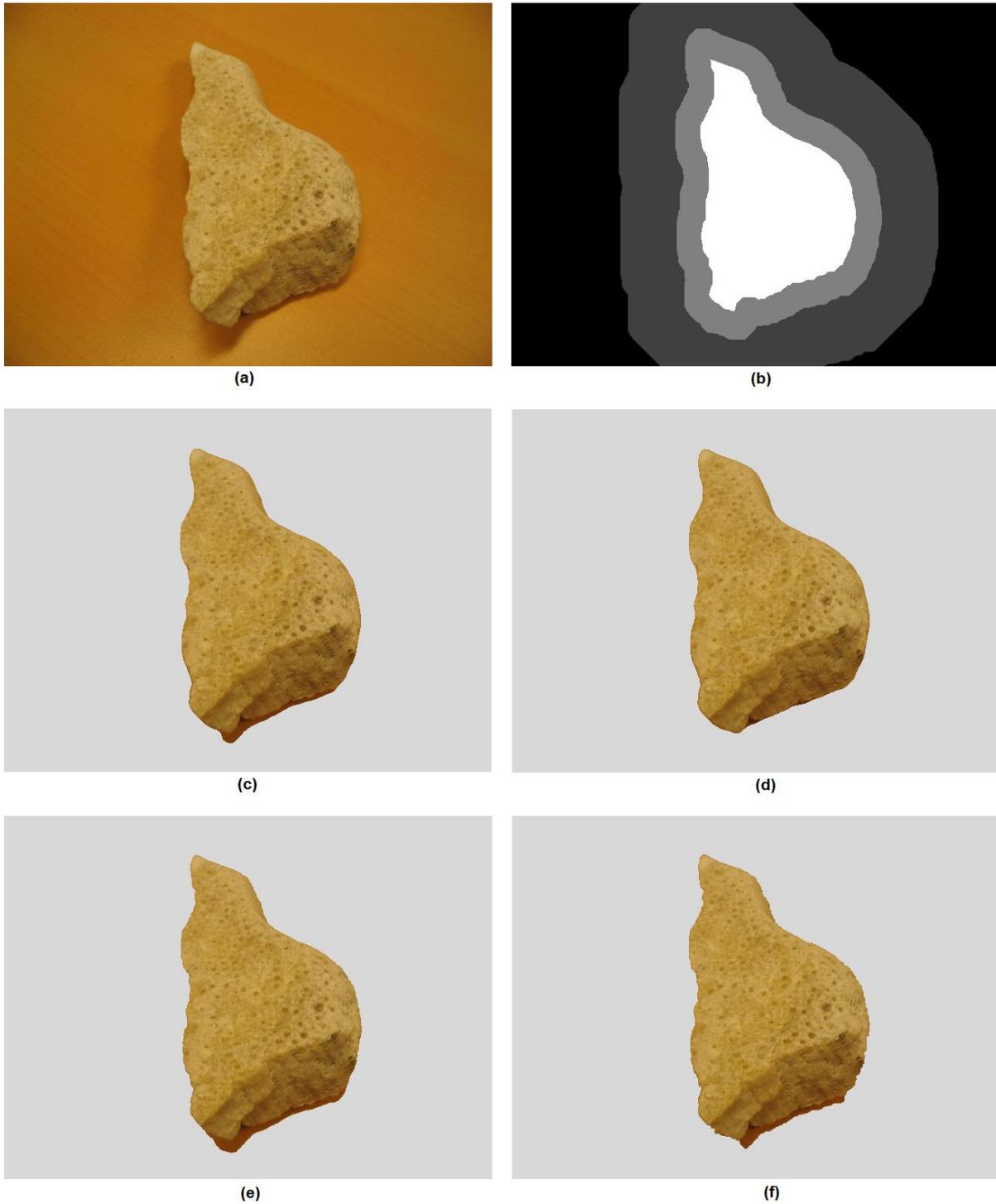


Abb. 60: Segmentierung von „stone1“
 (a) Zu segmentierendes Bild
 (b) Lasso-Trimap
 (c) Joint Bilateral Filter
 (d) Joint Bilateral Median Filter
 (e) Guided Image Filter
 (f) Grab-Cut

Fehlerrate: 0.0387
 Fehlerrate: 0.0079
 Fehlerrate: 0.0578
 Fehlerrate: 0.0418



Abb. 61: Segmentierung von „person5“
 (a) Zu segmentierendes Bild
 (b) Lasso-Trimap
 (c) Joint Bilateral Filter
 (d) Joint Bilateral Median Filter
 (e) Guided Image Filter
 (f) Grab-Cut

Fehlerrate: 0.0229
 Fehlerrate: 0.0192
 Fehlerrate: 0.0176
 Fehlerrate: 0.1563

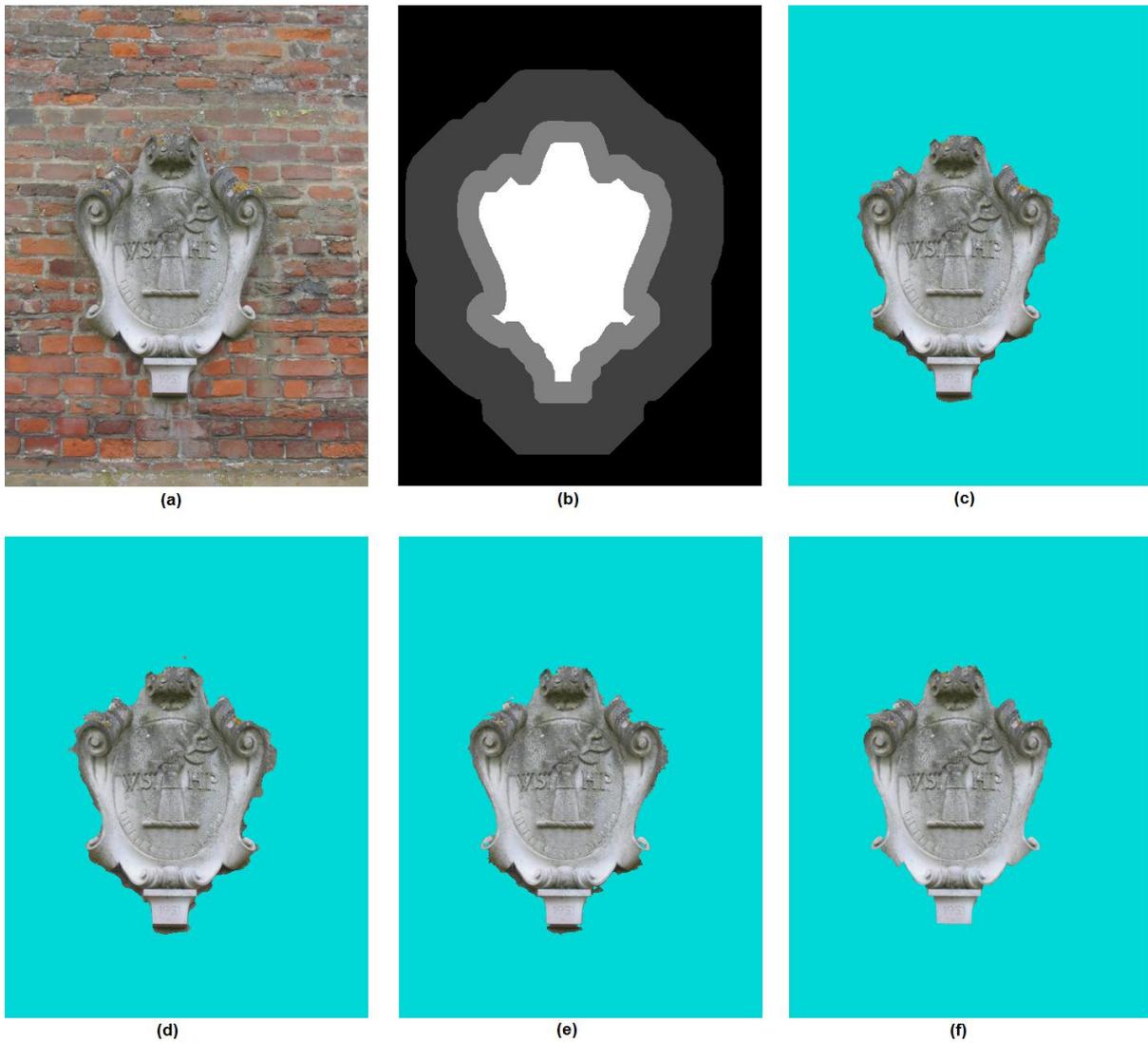


Abb. 62: Segmentierung von „memorial“
 (a) Zu segmentierendes Bild
 (b) Lasso-Trimap
 (c) Joint Bilateral Filter
 (d) Joint Bilateral Median Filter
 (e) Guided Image Filter
 (f) Grab-Cut

Fehlerrate: 0.1230
 Fehlerrate: 0.1270
 Fehlerrate: 0.1035
 Fehlerrate: 0.0741

5.4 Paint Selection

Bisher wurde davon ausgegangen, dass der Benutzer die gesamte benötigte a-priori Information vor Aufruf des Algorithmus definiert. Wenn die gegebene Information nicht für eine Segmentierung in gewünschter Qualität ausreicht, muss die a-priori Information vom Benutzer überarbeitet und anschließend der gesamte Algorithmus erneut ausgeführt werden. Aus Sicht des Anwenders wäre es jedoch wünschenswert, bei jeder Veränderung der a-priori Information sofort Feedback in Form eines segmentierten Bildes zu erhalten. Aufgrund der hohen Geschwindigkeit des Guided Image Filters ist dies prinzipiell möglich. Bei Verwendung des in Abschnitt 5.2 beschriebenen Algorithmus treten jedoch unerwünschte Effekte auf:

- Zu Beginn sind alle Pixel als unbekannt gekennzeichnet. Beginnt der Benutzer mit der Kennzeichnung einer Vordergrundregion führt der anschließende Aufruf der Segmentierungsroutine dazu, dass alle Pixel des Bildes als Vordergrund klassifiziert werden. Umgekehrt wird, wenn der Benutzer zuerst den Hintergrundbereich kennzeichnet, zunächst das gesamte Bild als Hintergrund klassifiziert.
- Bei der Veränderung der a-priori Information in einem bestimmten Bereich kann es vorkommen, dass weit entfernte Pixel ihre Klassifizierung verändern. Im Allgemeinen erwarten Anwender jedoch eine gewisse Lokalität der getätigten Eingaben.

Paint-Selection [41] ist ein auf ein Graph-Cut basierender Algorithmus, der die genannten Probleme minimiert. In diesem Abschnitt soll gezeigt werden, dass kantenerhaltende Glättungsfiler oft einfach in bestehende Algorithmen integriert werden können. Im Fall des Paint-Selection Algorithmus kann Graph-Cut einfach durch einen Guided Image Filter ersetzt werden:

Bei Paint-Selection beginnt der Benutzer immer mit der Markierung der Vordergrundbereiche, durch Zeichnen des Foreground-Scribbles. Während des Zeichenvorgangs wird in periodischen Abständen (bspw. alle 15ms) eine Segmentierungsroutine aufgerufen, welche mit zwei Parametern *new_fg* und *seg* arbeitet. *new_fg* enthält alle Pixelpositionen, die seit dem letzten Aufruf der Routine dem Foreground-Scribble hinzugefügt wurden. *seg* ist eine Binärkarte, welche die im letzten Aufruf der Routine errechneten Pixelklassifikationen enthält, wobei beim ersten Aufruf alle Bildpositionen in *seg* als Hintergrund gekennzeichnet sind. Bei



Erklärung: Da der mittlere Teil der Blume in der letzten Iteration nicht als Vordergrund markiert wurde (siehe 'seg'), zeichnet der Benutzer ein **Foreground Scribble** in diesem Bereich. Die Grafik demonstriert den anschließenden Aufbau der beiden Histogramme.

Die gezeigte Segmentierung ('seg') wurde händisch erstellt um die Arbeitsweise des Algorithmus möglichst übersichtlich beschreiben zu können. Tatsächlich mit Paint-Selection berechnete Segmentierungen werden in Abb. 64 gezeigt.

Abb. 63: Histogrammaufbau im modifizierten Paint-Selection Algorithmus

jedem Aufruf der Segmentierungsroutine werden mit Hilfe der beiden Parameter die bereits aus Abschnitt 5.2 bekannten Vorder- und Hintergrundhistogramme aufgebaut. Zunächst werden die Farben aller Pixel, deren Position in *new_fg* enthalten ist, dem Vordergrundhistogramm hinzugefügt. Danach wird die Bounding-Box der Pixelpositionen in *new_fg* berechnet und die errechnete Box in alle vier Richtungen um einen konstanten Wert vergrößert. Anschließend wird die Farbe jedes Pixels, der in *seg* als Vordergrund markiert ist, dem Vordergrundhistogramm hinzugefügt, wenn seine Position in der erweiterten Bounding Box enthalten ist. Um eine mögliche doppelte Zählung auszuschließen werden jene Pixel, die auch in *new_fg* enthalten sind, nicht nochmals berücksichtigt. Für das Hintergrund-Histogramm werden die Farben 1200 zufällig ermittelter Pixelpositionen verwendet, die in *seg* als Hintergrund klassifiziert- und nicht in der vergrößerten Box enthalten sind. Durch diese Vorgehensweise ist bereits beim ersten Aufruf der Klassifikationsroutine ein befülltes Hintergrundhistogramm vorhanden. Es wird somit vermieden, dass bei nicht vorhandenem Background-Scribble alle Pixel zwangsweise als Vordergrund klassifiziert werden. Die beschriebene Vorgehensweise wird in Abb. 63 grafisch veranschaulicht.

Nach Aufbau der Histogramme wird das bereits aus Abschnitt 5.2 bekannte Cost-Volume berechnet (siehe Formel 13, Seite 68). Es ist zu beachten, dass allen Positionen unter dem Foreground-Scribble der Wert 1.0 zugewiesen werden muss. Das Cost-Volume wird mittels Guided Image Filter geglättet und das Ergebnis unter Verwendung eines Thresholds von 0.5

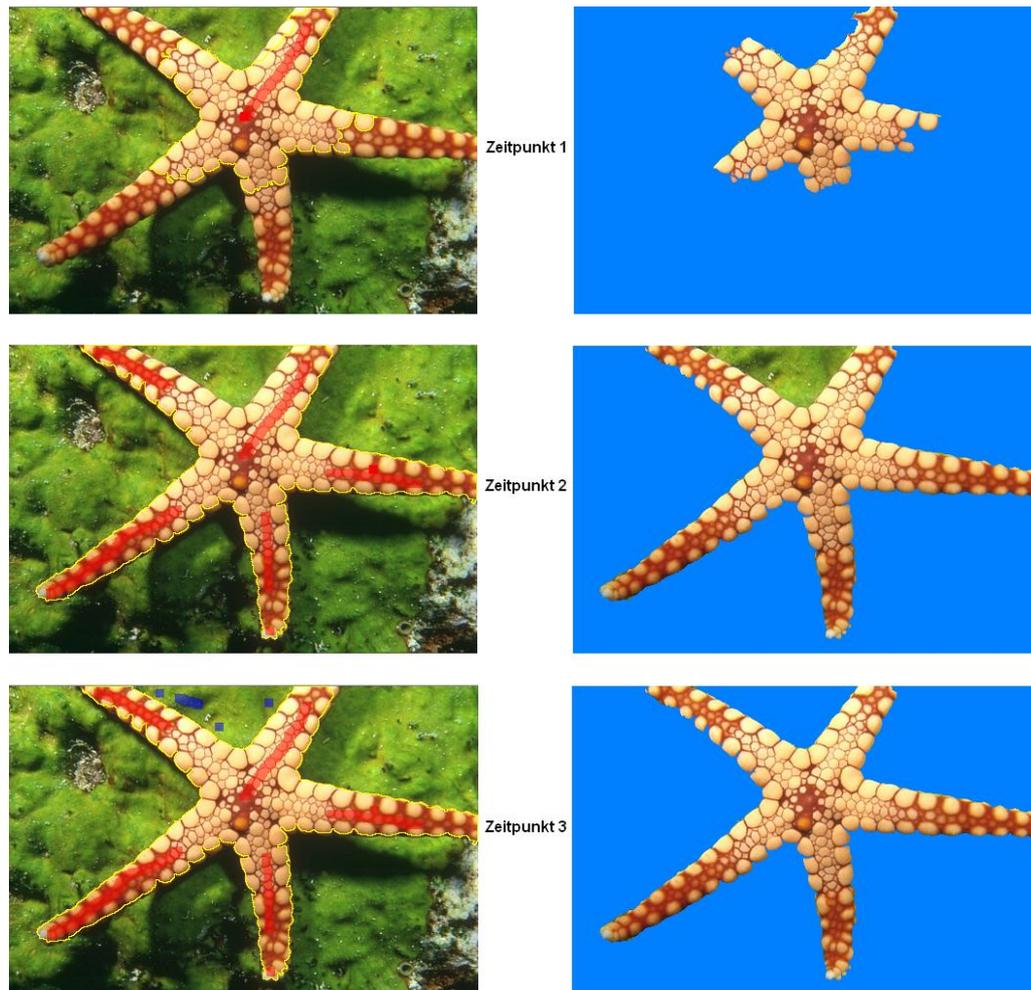


Abb. 64: Paint-Selection mit Guided Image Filter

binarisiert. In einem letzten Schritt werden alle Pixelpositionen, die im binarisierten Cost-Volume eine zusammenhängende Region mit einer Position aus new_fg bilden, in seg als Vordergrund klassifiziert. Alle anderen Klassifikationen in seg werden nicht verändert. Durch die Constraint der zusammenhängenden Region wird eine gewisse Lokalität des Algorithmus gewährleistet, da in vielen Fällen nur Pixel in der Nähe von new_fg in der Region enthalten sind.

Aufgrund der Arbeitsweise des Paint-Selection Algorithmus sind im Allgemeinen nur wenig bis gar keine Background-Scribbles nötig um eine hochwertige Segmentierung zu erzielen. In dieser Diplomarbeit wird daher auf eine Beschreibung der Background-Scribble Logik verzichtet. Die entsprechenden Ausführungen des Paint-Selection Papers [41] können jedoch einfach an den hier beschriebenen modifizierten Algorithmus angepasst werden.

Abb. 64 zeigt einige Screenshots des im Rahmen dieser Diplomarbeit implementierten modifizierten Paint-Selection Algorithmus. In der linken Spalte ist das zu segmentierende

Objekt inklusive vom Benutzer gezeichneter Foreground-Scribbles (rot) und Background-Scribbles (blau) zu sehen. Die rechte Spalte zeigt das segmentierte Vordergrundobjekt.

5.5 Alpha-Matting

Die in Abschnitt 5.2 und 5.4 vorgestellten Segmentierungsalgorithmen sind binäre Klassifikatoren, da nur zwischen den Klassen Vordergrundobjekt und Hintergrundbereich unterschieden wird. Solche Algorithmen werden auch Hard-Segmentation Verfahren genannt. Im Gegensatz dazu berechnen auf Alpha-Matting basierende Verfahren keine direkte Klassenzugehörigkeit. Stattdessen wird für jeden Bildpunkt ein Wert α (alpha) ermittelt, der angibt, wie viel seiner Farbinformation Teil des Vordergrundobjekts ist. Das Ergebnis eines Alpha-Mattings zeigt daher in welchem Verhältnis ein gegebener ursprünglicher Pixel mit einem neuen Hintergrund überblendet werden muss, um das Vordergrundobjekt auf den neuen Hintergrund zu übertragen. Da Alpha-Matting das Vordergrundobjekt sozusagen weich aus dem Hintergrund herauslöst wird oft der Begriff Soft Segmentation als Synonym für Alpha-Matting verwendet. Auf Alpha-Matting basierende Verfahren sind bei Pixeln, die sowohl Vorder- als Hintergrundinformation enthalten, binären Klassifikatoren überlegen. Solche Pixel finden sich beispielsweise häufig an den Rändern des Vordergrundobjekts. Semitransparente Objekte, wie beispielsweise eine farbige Glasflasche, enthalten auch im inneren Bereich Pixel mit Vorder- und Hintergrundinformation. Die Autoren des Guided Image Filters zeigen in ihrer Arbeit [3] wie das Ergebnis eines binären Klassifikators in eine Alpha-Matte konvertiert werden kann. Es reicht aus, das Ergebnis des binären Klassifikators mit dem Guided Image Filter unter Verwendung des ursprünglichen, zu segmentierenden Bildes, als Guidance Image zu filtern. Diese Anwendung des Guided Image Filters wird von den Autoren als Guided Feathering bezeichnet, da insbesondere an den Rändern gefedert oder behaarter Tiere beeindruckende Ergebnisse erzielt werden können (siehe *Abb. 65*, nächste Seite).

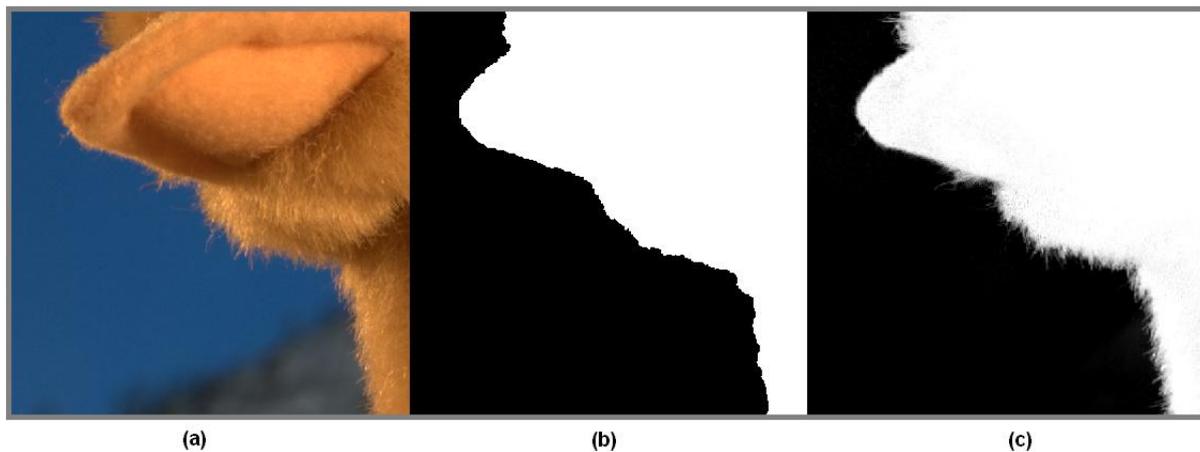


Abb. 65: Alpha-Matting mit Guided Image Filter
(a) Zu segmentierendes Bild
(b) Hard-Segmentation
(c) Alpha-Matting (Soft-Segmentation)
Berechnet durch den Referenzcode der Guided Image Filter [3] Autoren

6. Optimierung von Tiefenkarten in Disparitätsvideos

6.1 Motivation

Tiefenkarten stereoskopischer Videos weisen oft eine hohe Anzahl von Fehlern auf. In diesem Abschnitt soll untersucht werden, ob fehlerhafte Tiefenkarten mit Hilfe kantenerhaltender Glättungsfilter optimiert werden können. Es ist anzunehmen, dass durch kantenerhaltende Glättung der Tiefenkarte unter Zuhilfenahme des zugehörigen Kamerabildes als Guidance Image, die Zahl der fehlerhaften Punkte signifikant reduziert werden kann. Diese Hypothese soll in Abschnitt 6.2 überprüft werden. Im Rahmen dieser Diplomarbeit wurde ein Programm entwickelt, das ein Set von Tiefenkarten mit den vorgestellten kantenerhaltenden Filtern (Joint Bilateral Filter, Joint Bilateral Median Filter, Guided Image Filter) glättet und die Abweichung der Ergebnisse zu den Ground-Truth Daten ermittelt.

Da Stereo-Matcher einen hohen Komplexitätsgrad aufweisen, können qualitativ hochwertige Tiefenkarten auf derzeit verbreiteter Hardware nur bei kleineren Bilddimensionen in Echtzeit berechnet werden. Um dennoch eine Echtzeitverarbeitung hochauflösender Bilder zu erreichen, wird der Stereo-Matcher oft auf einem verkleinerten Kamerabildpaar ausgeführt und die berechnete Tiefenkarte in einem Post-Processing Schritt auf die Größe des ursprünglichen Kamerabildes skaliert. Aus der Bildverarbeitung sind viele performante Skalierungsalgorithmen bekannt, wie beispielsweise Nearest-Neighbour Sampling, Bilinear Sampling oder Bicubic Sampling. Ein entscheidender Nachteil dieser Vorgehensweise ist jedoch der Verlust der exakten Kantenpositionen beim Downsampling der Kamerabilder. Weiters können die oben genannten traditionellen Skalierungsfiler die exakten Kantenpositionen im Upsampling-Schritt nicht wiederherstellen. Folglich werden in den Kantenbereichen viele falsche Disparitäten ausgegeben. In [42] demonstrierten Kopf et al., dass dieses Problem durch Berücksichtigung des ursprünglichen Kamerabildes im Upsampling Prozess minimiert werden kann, da das ursprüngliche Kamerabild die beim Downsampling verloren gegangenen exakten Kantenpositionen enthält. Kopf et al. empfehlen einen modifizierten Joint Bilateral Filter im Upsampling Schritt einzusetzen. Es ist zu vermuten, dass Joint Bilateral Median Filter und Guided Image Filter ebenfalls gut für diese Aufgabe geeignet sind. In Abschnitt 6.3 soll festgestellt werden welche qualitativen Unterschiede zwischen den drei genannten kantenerhaltenden Filtern, bei der beschriebenen Upsampling-Problemstellung, bestehen. Im Rahmen dieser Diplomarbeit wurde ein Programm entwickelt, das ein Set von verkleinerten Tiefenkarten, sowohl mit traditionellen Algorithmen (Nearest-Neighbour Upsampling, Bicubic Upsampling) als auch mit angepassten kantenerhaltenden Filtern (Joint Bilateral Filter, Joint Bilateral Median Filter,

Guided Image Filter), hochskaliert und die Abweichung der Ergebnisse zu den Ground-Truth Daten ermittelt.

6.2 Kantenerhaltende Glättung von Tiefenkarten

Ziel der folgenden Evaluierung ist festzustellen, ob die Qualität von Tiefenkarten durch kantenerhaltende Filterung mit Guidance Image verbessert werden kann. Weiters soll ermittelt werden, ob ein bestimmter Filter besonders für diese Aufgabe geeignet ist.

Jeder der drei evaluierten Filter (Joint Bilateral Filter, Joint Bilateral Median Filter, Guided Image Filter) wird auf 30 Tiefenkarten angewandt, welche mit dem in [43] vorgestellten Stereo-Matcher berechnet und von den Autoren des Papers freundlicherweise zu Verfügung gestellt wurden. Da die kantenerhaltende Filterung nicht nur falsche Disparitäten korrigiert, sondern auch zu einem gewissen Grad korrekte Information zerstört, werden die Filter in dieser Evaluierung nicht auf den gesamten Bildbereich angewendet. Davon ausgehend, dass ein Stereo Matcher üblicherweise in nicht verdeckten Regionen nahezu fehlerfrei arbeitet, werden diese Regionen bei der Glättung nicht angetastet. Die Disparitäten der restlichen Bildbereiche (der Occluded Regions) sind hingegen im Allgemeinen deutlich stärker fehlerbehaftet. In diesen Bereichen erscheint eine Glättung sinnvoll. Um die Evaluierung durchführen zu können werden also, zusätzlich zu den Tiefenkarten Bilder benötigt, die zeigen welche Bereiche verdeckt bzw. unverdeckt sind. Diese Bilder, oft Occlusion Maps genannt, werden von jedem Stereo-Matcher mit Occlusion Handling berechnet [44]: Stereo Matcher mit Occlusion Handling ermitteln zunächst aus dem gegebenen Kamerabildpaar die beiden zugehörigen Tiefenkarten. Anschließend werden die verdeckten Regionen der linken Tiefenkarte ermittelt, indem die rechte Tiefenkarte gewarpt wird. Dazu werden die Disparitäten der rechten Tiefenkarte horizontal entsprechend ihrer Disparität verschoben und in der gewarpten Tiefenkarte abgelegt. Befindet sich beispielsweise in der rechten Tiefenkarte an der Position $(x=10, y=3)$ ein Disparitätswert von 4, so wird dieser Wert in der gewarpten rechten Tiefenkarte an Position $(x=14, y=3)$ abgelegt. Werden mehrere Disparitäten auf dieselbe Position verschoben, so wird lediglich die höchste dieser Disparitäten gespeichert, da höhere Disparitäten näher beim Betrachter liegen und somit alle dahinterliegenden Disparitäten abdecken. Das gewarpte rechte Tiefenbild entspricht im Idealfall exakt der zuvor berechneten linken Tiefenkarte. Jene Positionen, in denen die Unterschiede einen gewissen Schwellwert überschreiten, stellen verdeckte Positionen da. Zu diesem Zeitpunkt kann die Occlusion Map also durch einen einfachen Vergleich ermittelt werden. Der Stereo Matcher verwendet diese Information üblicherweise zur nachträglichen Korrektur verdeckter Regionen (beispielsweise durch Inpainting-

Algorithmen). Im Rahmen dieser Evaluierung wird die Information verwendet, um zu entscheiden, welche Positionen geglättet werden. Occluded Regions befinden sich oft in jenen Bereichen in denen nur eine der beiden Kameras einen bestimmten Teil des Objekts erfassen konnte.

Die Qualität eines Filters hängt stark von den gewählten Parametern (Filterradius, etc.) ab. Im Rahmen dieser Evaluierung wurde versucht, für jeden getesteten Filter ein Parameterset zu finden, welches für jede der 30 Tiefenkarten ein gutes Ergebnis liefert. Dazu wurden pro Filter mehrere 100 mögliche Parametersets automatisiert getestet. Um die Qualität eines konkreten Sets zu ermitteln, wurde für jede der 30 geglätteten Tiefenkarten der Mean Absolute Error (MAE) zur Ground Truth berechnet. Jenes Parameterset, bei dem der Durchschnitt der 30 berechneten MAE-Werte am Geringsten ist, stellt das Optimum da. In der folgenden Tabelle (siehe nächste Seite) sind die einzelnen Filter jeweils mit dem ermittelten optimalen Parameterset vertreten.

Für 43,33% (13/30) der Tiefenkarten errechnete der Joint Bilateral Median Filter das genaueste Ergebnis. Joint Bilateral Filter und Guided Image Filter konnten respektive bei 33% (10/30) und 20% (6/30) der Tiefenkarten die beste Qualität erzielen. Lediglich 3,33% (1/30) der Tiefenkarten wurden durch alle getesteten Glättungsalgorithmen verschlechtert. Im Durchschnitt lieferte der Joint Bilateral Median Filter die höchste Qualität. Um besser entscheiden zu können, ob die Tiefenkarten durch Glättung signifikant verbessert werden konnten, werden nun ausgewählte Ergebnisse in Bildform präsentiert. Bei allen gezeigten Tiefenkarten wurden die Disparitäten auf den maximalen Intensitätsbereich gespreizt.

„Tsukuba“ (Abb. 66, Seite 90): Im oberen Teil der Lampe (dunkelgelber Bereich) wurden durch den Stereo-Matcher einige Punkte fälschlicherweise dem Vordergrund zugeordnet. Der Joint Bilateral Median Filter konnte diese Stelle nahezu perfekt korrigieren. Joint Bilateral Filter und Guided Image Filter lieferten ebenfalls gute Ergebnisse, konnten in diesem Fall jedoch die Qualität des Joint Bilateral Median Filters nicht ganz erreichen, da durch die Mittelwertbildung das Ergebnis zu stark von inkorrekten Disparitäten beeinflusst wurde. Eine ähnliche Situation besteht im linken, unteren Teil der Lampe (grüner Bereich). Auch der hellblau gekennzeichnete Lampenbereich wurde durch den Stereo-Matcher fälschlicherweise als Vordergrund klassifiziert. Dieser Fehler konnte jedoch durch keinen Filter korrigiert werden, da er in einem unverdeckten Bereich fällt. Zu einem interessanten Ergebnis kam es im linken, unteren Teil des Kopfes (roter Bereich). Alle drei Filter konnten die falschen Disparitäten weitgehend korrigieren, jedoch blieb in allen Fällen ein kleiner isolierter Bereich falscher Disparitäten unverändert. Auch hier ist der Grund für das fehlerhafte Verhalten, dass

der isolierte Bereich unverdeckt und daher von der Filterung nicht betroffen ist. Zusammenfassend kann festgestellt werden, dass alle drei Filter die „Tsukuba“-Tiefenkarte signifikant verbessern konnten. Durch die Mittelwertbildung waren Joint Bilateral Filter und Guided Image Filter dem Joint Bilateral Median Filter jedoch unterlegen. Diese visuelle Analyse wird durch die berechneten MAE-Werte bestätigt. Nach Filterung mit Joint Bilateral Median betrug der MAE 0.0630, während Joint Bilateral Filter und Guided Image Filter respektive MAE-Werte von 0.0771 und 0.0830 erreichten.

	Ohne Glättung	Joint Bilateral	Joint Bilateral Median	Guided Image
		$kSize = 19$ $\sigma_s = 9$ $\sigma_l = 0.25$	$kSize = 19$ $\sigma_s = 9$ $\sigma_l = 0.4$	$kSize = 19$ $\varepsilon = 0.00000005$
Bild	Mean Absolute Error			
Aloe	0.0563	0.0475	0.0441	0.0462
Art	0.0866	0.0678	0.0632	0.0686
baby1	0.0331	0.0304	0.0321	0.0314
baby2	0.0840	0.0790	0.0806	0.0774
baby3	0.0360	0.0326	0.0288	0.0321
books	0.1492	0.1451	0.1472	0.1456
bowling1	0.1320	0.1289	0.1292	0.1300
bowling2	0.0433	0.0398	0.0400	0.0400
cloth1	0.0182	0.0181	0.0182	0.0183
cloth2	0.0435	0.0422	0.0424	0.0428
cloth3	0.0469	0.0452	0.0459	0.0454
cloth4	0.0684	0.0676	0.0605	0.0685
cones	0.0619	0.0536	0.0527	0.0522
dolls	0.0454	0.0366	0.0361	0.0365
flowerpots	0.0307	0.0283	0.0282	0.0289
lampshade1	0.0618	0.0553	0.0566	0.0544
lampshade2	0.0563	0.0512	0.0557	0.0514
laundry	0.0897	0.0809	0.0641	0.0842
midd1	0.1628	0.1527	0.1614	0.1518
moebius	0.0786	0.0683	0.0673	0.0669
monopoly	0.0537	0.0476	0.0529	0.0485
plastic	0.0446	0.0403	0.0413	0.0412
reindeer	0.0326	0.0250	0.0242	0.0258
rocks1	0.0177	0.0157	0.0155	0.0157
rocks2	0.0225	0.0216	0.0225	0.0214
Teddy	0.0373	0.0354	0.0347	0.0360
tsukuba	0.1047	0.0771	0.0630	0.0830
venus	0.0237	0.0192	0.0178	0.0211
wood1	0.0800	0.0816	0.0803	0.0821
wood2	0.0139	0.0137	0.0115	0.0143
Durchschnitt	0.0605	0.0549	0.0539	0.0554

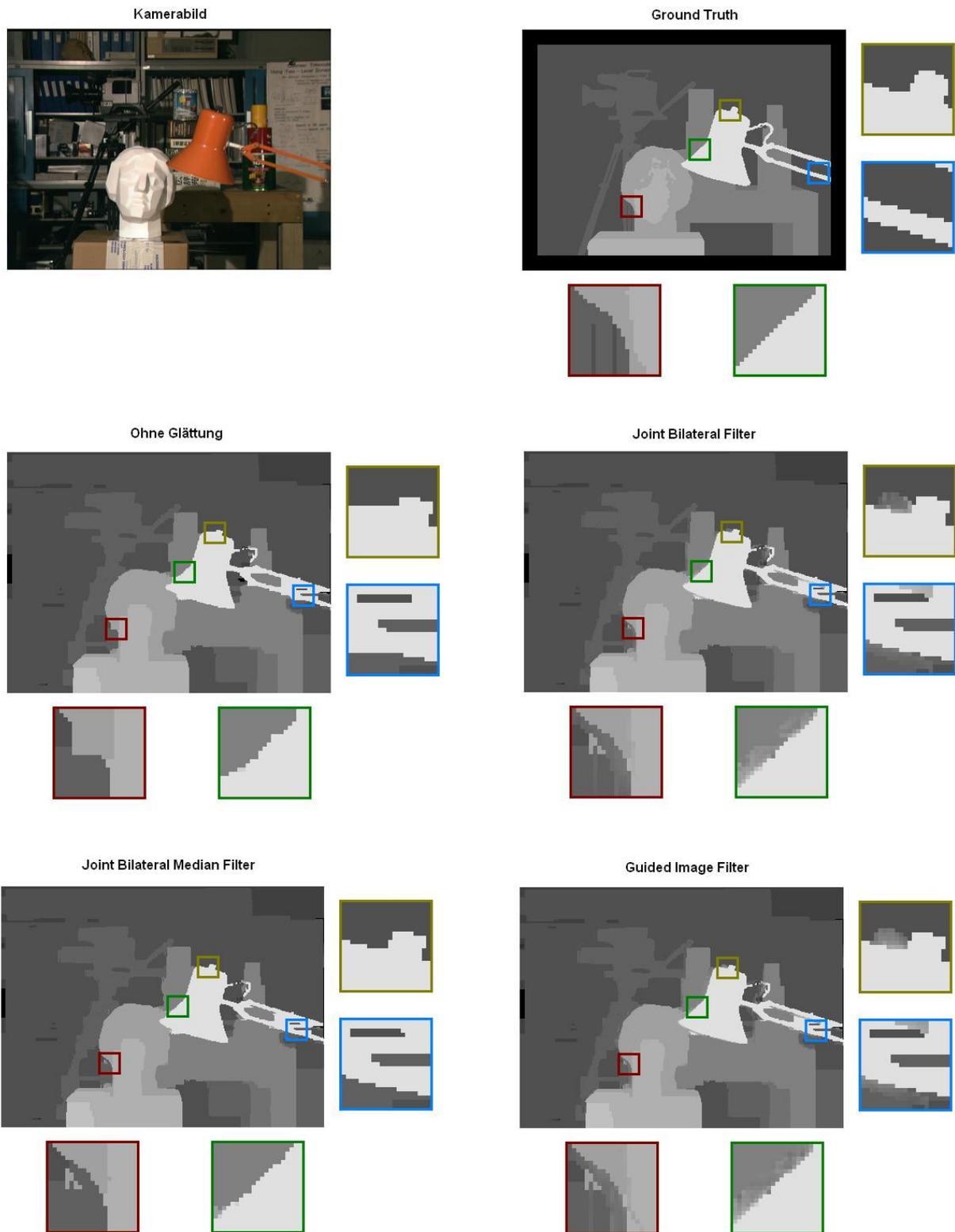


Abb. 66: Glättung der „Tsukuba“-Tiefenkarte

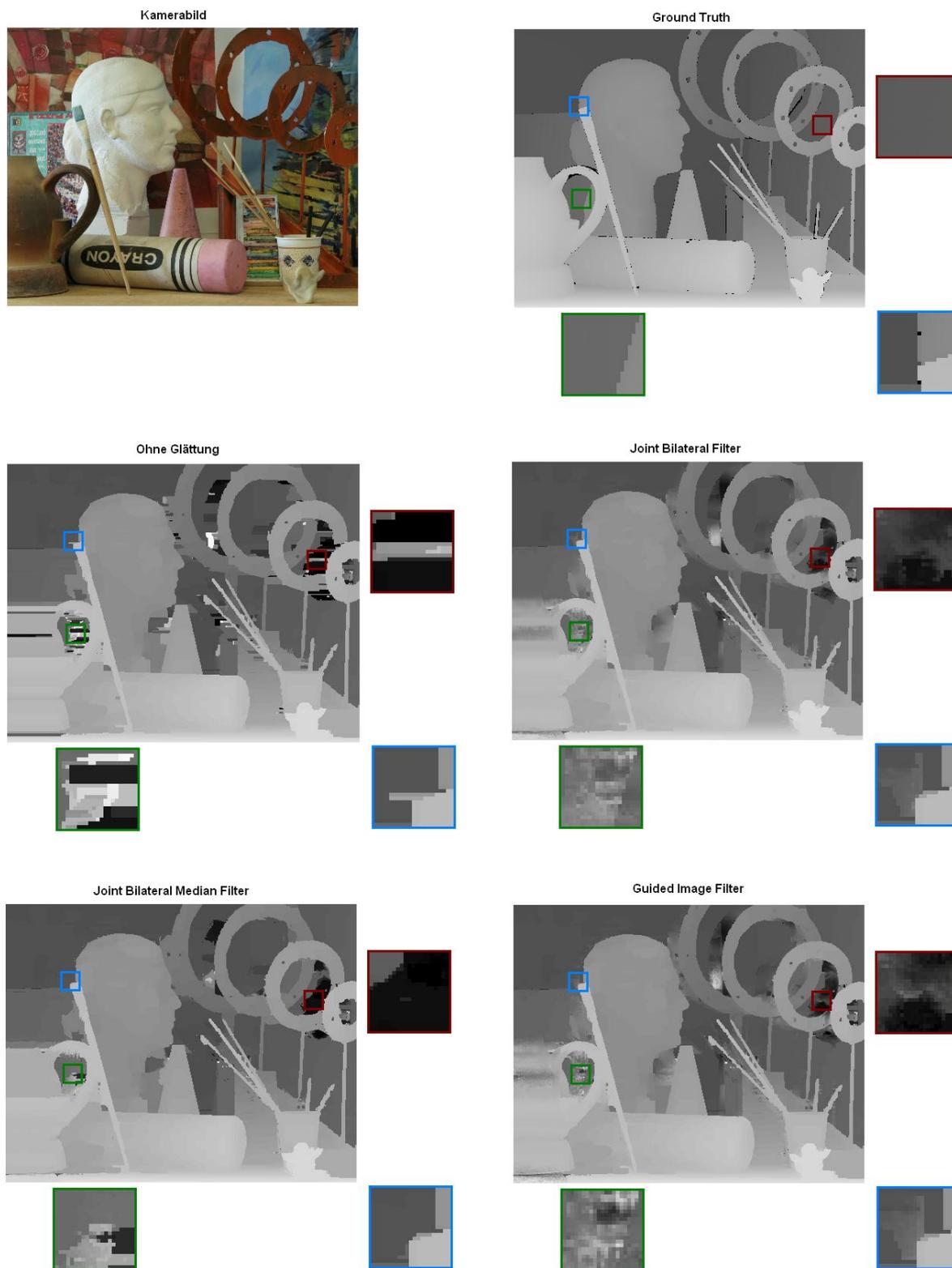


Abb. 67: Glättung der „Art“-Tiefenkarte

„Art“ (Abb. 67): Die ungeglättete Tiefenkarte weist starke Artefaktbildung in den verdeckten Regionen auf, beispielsweise im Hintergrundbereich zwischen Gefäß und Henkel (siehe grüner Bereich). In der vergrößerten Darstellung ist zu erkennen, dass die Artefakte in diesem Bereich von allen getesteten Filtern stark reduziert werden konnten. Weiters ist in der Ground-Truth Karte zu erkennen, dass der genannte Bereich von einer Kante durchkreuzt werden sollte. Keiner der Filter konnte diese Kante sichtbar werden lassen. Der rote Bereich verdeutlicht die unterschiedliche Arbeitsweise von Median- und Mittelwertbildung. Der Medianfilter entscheidet sich für eine in der Umgebung vorhandene Disparität, wodurch keine neuen Disparitäten entstehen und in der vergrößerten Darstellung zwei voneinander getrennte Bereiche zu erkennen sind. Die auf Mittelwert basierenden Joint Bilateral Filter und Guided Image Filter erzeugen hingegen durch die Mittelung viele neue, aber ähnliche, Disparitäten. Am Hinterkopf (hellblauer Bereich) kommt es zu einer interessanten Situation. Laut Ground-Truth Daten sollte die Spitze des Pinsels direkt am Rand des Kopfs liegen. In der Tiefenkarte des Stereo-Matchers scheint der Pinsel jedoch weiter links zu sein. Eine genauere Betrachtung ergibt jedoch, dass die Position des Pinsels korrekt erkannt wurde. Vielmehr wurde der Randbereich des Hinterkopfes fälschlicherweise auf die Hintergrunddisparität gesetzt. Keiner der drei getesteten Filter konnte den Fehler korrigieren, da die Stelle in einem unverdeckten Bereich lag. Auch in dieser Tiefenkarte war der Joint Bilateral Median Filter mit einem MAE von 0.0632 den beiden Konkurrenten (Joint Bilateral Filter: 0.0678, Guided Image Filter: 0.0686) überlegen. Alle geglätteten Tiefenkarten waren jedoch eine signifikante Verbesserung zur ungeglätteten Tiefenkarte (MAE: 0.0866). Bei den zwei bisher untersuchten Tiefenkarten kam es mehrmals zu der Situation, dass fehlerhafte Disparitäten nicht korrigiert wurden, weil sie im unverdeckten Bereich liegen. Es stellt sich also die Frage, ob es möglicherweise vorteilhaft wäre, den gesamten Bildbereich in die Glättung einzubeziehen. Abb. 68 (nächste Seite) zeigt das Ergebnis einer Glättung über den gesamten Bildbereich. In diesem Fall konnte der Guided Image Filter den fehlenden Teil des Hinterkopfes zwar teilweise wiederherstellen (grüner Bereich), jedoch werden weite Teile des Bildbereichs durch Artefakte gestört, die bei exklusiver Filterung des verdeckten Bereichs nicht auftreten (siehe roter Bereich). Es war nicht möglich diese Artefakte durch Anpassen der Filterparameter vollständig zu eliminieren. Der MAE des Guided Image Filters, bei Glättung über alle Bereiche, betrug 0.0741. Bei ausschließlicher Filterung der verdeckten Bereiche wurde hingegen der bereits oben genannte MAE von 0.0686 erreicht. Im Rahmen dieser Diplomarbeit wurden alle drei Glättungsfiler versuchsweise auf den gesamten Bereich der 30 Tiefenkarten angewendet. Im Vergleich zur empfohlenen Vorgehensweise, war bei jedem Filter ein signifikanter Qualitätsabfall feststellbar.

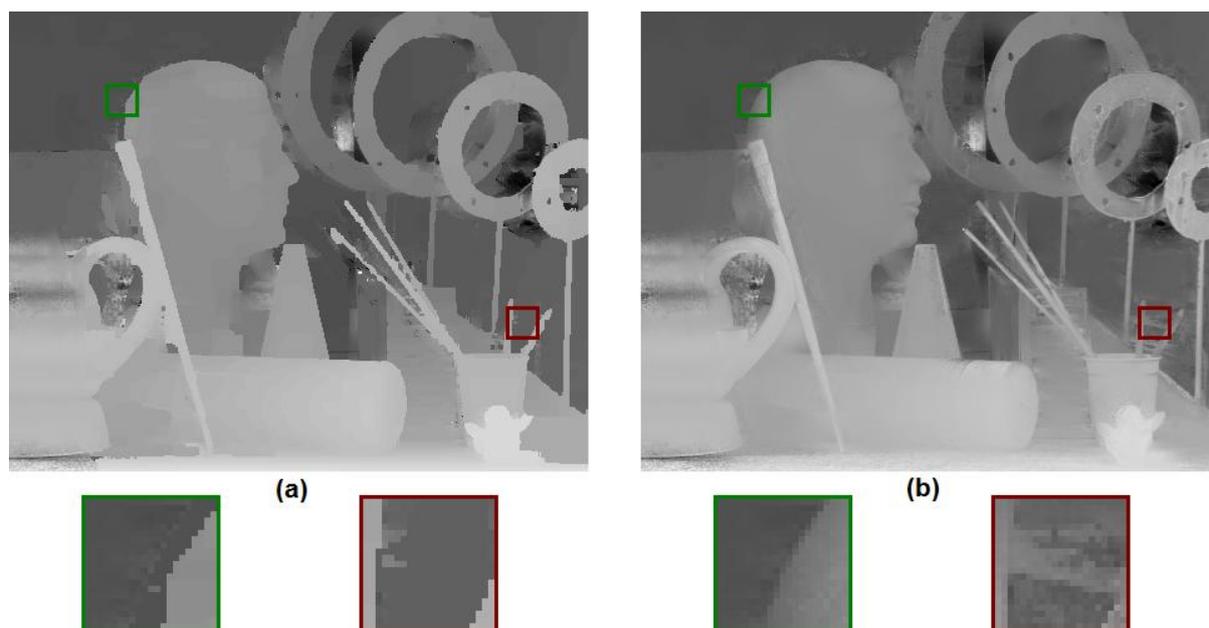


Abb. 68: (a) Filterung der verdeckten Bereiche
(b) Filterung aller Bereiche

„Wood1“ (Abb. 69, nächste Seite): Diese Tiefenkarte konnte, als einzige des Testsets, nicht durch kantenerhaltende Filterung optimiert werden. Bei der ungefilterten Tiefenkarte betrug der MAE zur Ground Truth 0.08. Bei Anwendung des Joint Bilateral Median Filters blieb die Qualität nahezu unverändert (MAE: 0.0803). Joint Bilateral Filter und Guided Image Filter berechneten respektive Tiefenkarten mit einem MAE von 0.0816 bzw. 0.0821. Der verdeckte Bereich des Bildes befindet sich zwischen den mit Pfeil markierten Holzstücken. Da die dort liegenden Disparitätswerte bereits durch den Stereo-Matcher weitgehend korrekt gesetzt wurden, konnte der Joint Bilateral Median Filter keine Verbesserung erzielen. Die beiden auf Mittelwert basierenden Glättungsfilter verschlechterten die Tiefenkarte sogar um eine Spur, da sie Artefakte im verdeckten Bereich erzeugten. Dieses Ergebnis bestätigt auch die zu Beginn des Abschnitts aufgestellte Behauptung, dass kantenerhaltende Filter zu einem gewissen Grad korrekte Information zerstören und daher möglichst nicht in Bereichen eingesetzt werden sollten, in denen kaum falsche Disparitäten vorhanden sind.

Zusammenfassend kann festgehalten werden, dass die Ergebnisse der getesteten Filter im Durchschnitt den ungeglätteten Tiefenkarten signifikant überlegen sind. Der Joint Bilateral Median Filter scheint besonders gut für die Aufgabe geeignet zu sein. Zwischen Joint Bilateral Filter und Guided Image Filter konnte hingegen kein signifikanter Unterschied festgestellt werden. Bei performancekritischen Anwendungen kann ein Einsatz des Guided Image Filters sinnvoll sein, da er im Gegensatz zum Joint Bilateral Median Filter sehr effizient implementiert werden kann (siehe Kapitel 3).

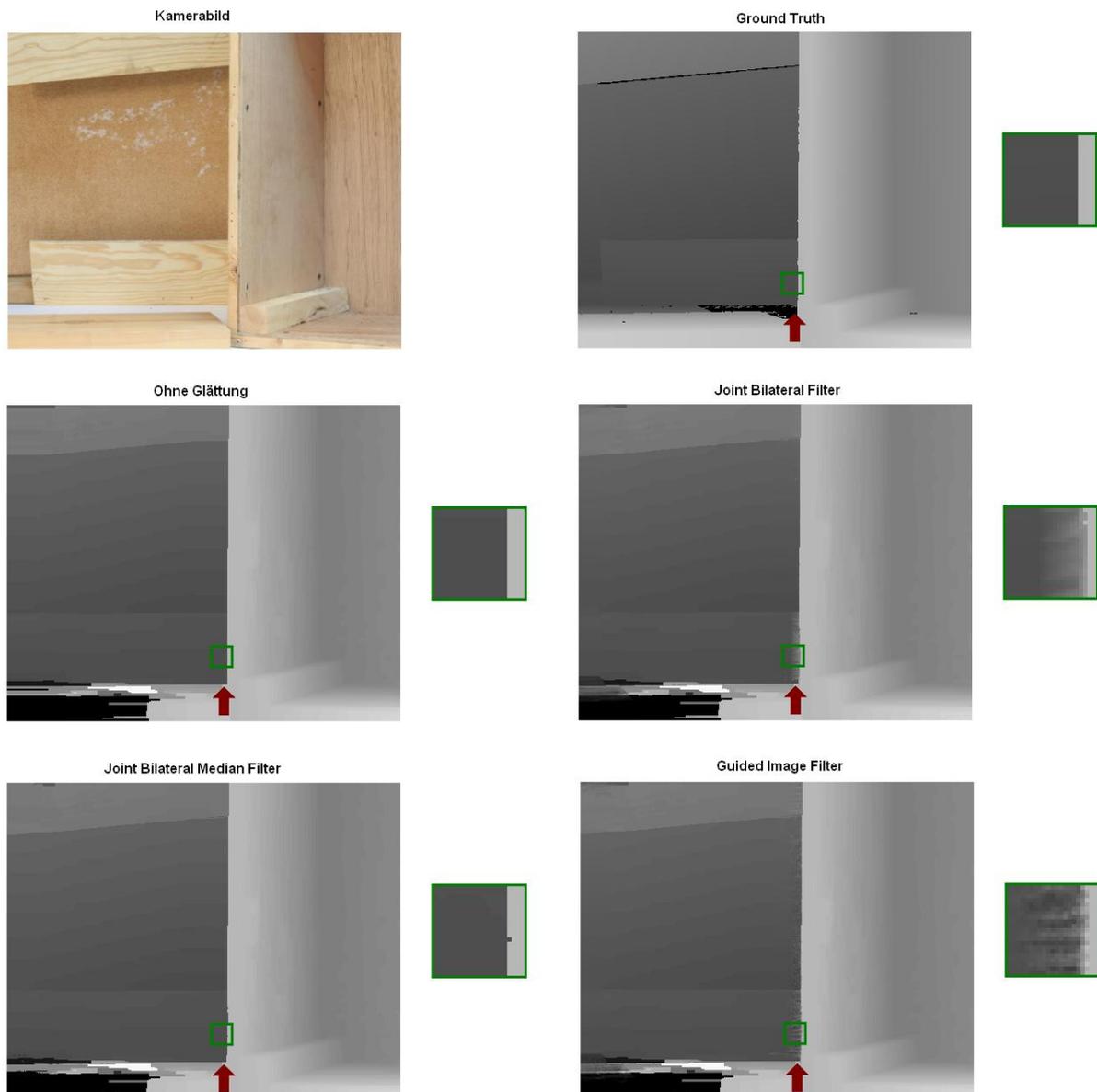


Abb. 69: Glättung der „Wood1“-Tiefenkarte

6.3 Upsampling von Tiefenkarten

Im Rahmen dieser Evaluierung soll festgestellt werden, welche qualitativen Vorteile durch den Einsatz kantenerhaltender Filter im Upsamplingprozess erreicht werden können. Die Evaluierung wurde wie folgt durchgeführt: Ein Kamerabildpaar der Middlebury-Datenbank [40][56] wurde per Nearest-Neighbour Downsampling um einen bestimmten Faktor verkleinert. Anschließend wurde der in [38] vorgestellte Stereo-Matcher auf die verkleinerten Bilder angewendet, um die Tiefenkarte des linken Kamerabildes zu berechnen. Der MATLAB-Code des Stereo Matchers wurde freundlicherweise von den Autoren des Papers zu Verfügung gestellt. In einem letzten Schritt wurden die berechneten Tiefenkarten auf die

Größe des unskalierten Kamerabildes gesampelt. Dabei wurden folgende Upsampling-Verfahren evaluiert: Nearest-Neighbour Upsampling, Bicubic Upsampling, Nearest-Neighbour Upsampling mit anschließender Glättung durch Joint Bilateral Filter, Joint Bilateral Median Filter und Guided Image Filter. Alle Glättungsfiler nutzten das unskalierte linke Kamerabild als Guidance Image. Die Qualität der Ergebnisse wurde anhand des Mean Square Errors (MSE) zur Ground-Truth ermittelt, wobei die Ground-Truth Daten dem Middlebury-Set entnommen wurden.

Wie bereits in Abschnitt 5.1 erwähnt, stammt die Idee, Tiefenkarten unter Verwendung eines Guidance Image zu skalieren, von Kopf et al., welche in ihrer Arbeit „Joint Bilateral Upsampling“ [42] einen modifizierten Joint Bilateral Filter mit integrierter Upsamplinglogik vorstellen. Es soll nun gezeigt werden, dass die in dieser Diplomarbeit verwendete sequentielle Methode (Nearest-Neighbour Upsampling mit anschließender Anwendung des Joint Bilateral Filters) nicht exakt dem Filter von Kopf et al. entspricht. Weiters wird begründet, warum die sequentielle Methode dem Algorithmus von Kopf et al. vorgezogen wurde.

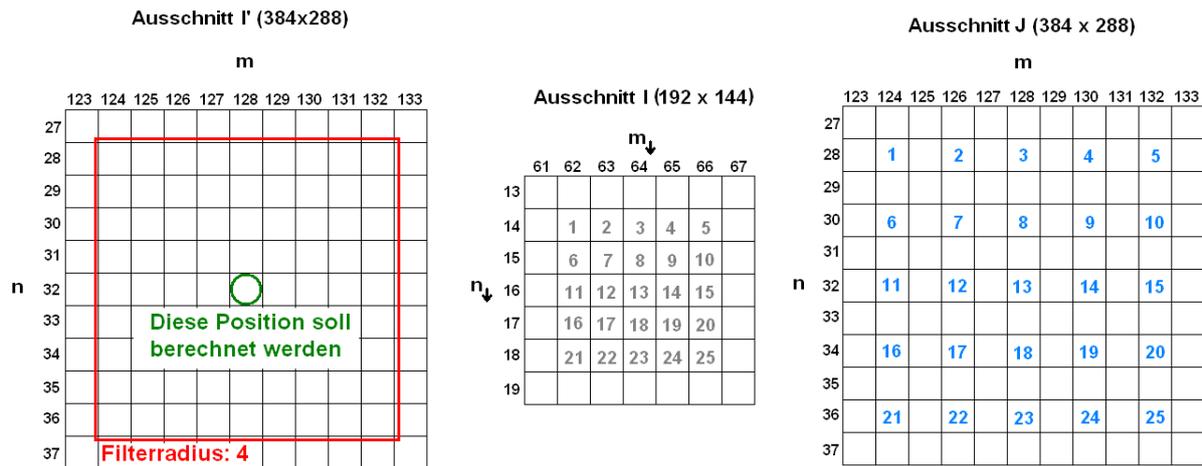
Der Algorithmus von Kopf et al. skaliert ein Bild I mit Hilfe eines modifizierten Joint Bilateral Filter, unter Verwendung eines Guidance Image J , zu einem vergrößerten Bild I' . Die Dimension von I' entspricht dem Guidance Image J . Beispielsweise kann eine 96x72 Pixel große Tiefenkarte I unter Verwendung eines 384x288 großen Kamerabildes J auf eine Größe von 384x288 skaliert werden. Formel 16 definiert den modifizierten Joint Bilateral Filter:

$$I'(p) = \frac{1}{W} \sum_{q_{\downarrow} \in R_p} G_{\sigma_s}(\|p_{\downarrow} - q_{\downarrow}\|) \cdot G_{\sigma_l}(\|J_p - J_q\|) \cdot I_q \quad (16)$$

$$W = \sum_{q_{\downarrow} \in R_p} G_{\sigma_s}(\|p_{\downarrow} - q_{\downarrow}\|) \cdot G_{\sigma_l}(\|J_p - J_q\|)$$

Die gezeigte Formel entspricht nahezu dem bereits in Formel 5 definierten Joint Bilateral Filter. Der Unterschied ist die Einführung der Koordinaten p_{\downarrow} und q_{\downarrow} , welche die zu p und q korrespondierenden Positionen im niedrig aufgelösten Bild I kennzeichnen. Formel 17 zeigt den Zusammenhang von p und p_{\downarrow} bzw. q und q_{\downarrow} :

$$p_{\downarrow} = \text{floor}\left(\frac{p}{\text{fact}_x}\right) \quad q_{\downarrow} = \text{floor}\left(\frac{q}{\text{fact}_y}\right) \quad (17)$$



$$I'[128,32] = \frac{1}{W} \left(G_{\sigma_s} \left(\frac{\| [64,16] - [62,14] \|}{13} \right) \cdot G_{\sigma_1} \left(\frac{| J[128,32] - J[124,28] |}{13} \right) \cdot \frac{1}{1} \cdot [62,14] + \dots + G_{\sigma_s} \left(\frac{\| [64,16] - [66,18] \|}{13} \right) \cdot G_{\sigma_1} \left(\frac{| J[128,32] - J[132,36] |}{13} \right) \cdot \frac{1}{25} \cdot [66,18] \right)$$

Abb. 70: Berechnung der Ausgabewerte bei 2x Joint Bilateral Upsampling mit $r = 4$. Das Guidance Image J wird nicht vollständig gesampelt.

$fact_x$ und $fact_y$ definieren horizontalen und vertikalen Upsampling-Faktor. Diese Faktoren können direkt aus den Auflösungen der Bilder I und J abgeleitet werden. Bei Verwendung der zuvor definierten Auflösungen gilt beispielsweise $fact_x = 384/96 = 4$, sowie $fact_y = 288/72 = 4$. Abb. 70 zeigt eine grafische Darstellung des modifizierten Joint Bilateral Filters. Die wesentliche Eigenschaft des modifizierten Joint Bilateral Filters ist, dass bei der Summenbildung über die Koordinaten q_{\downarrow} iteriert wird. Da q_{\downarrow} aufgrund des floor -Operators in Formel 17 nur ganzzahlige Werte annehmen kann, wird das Guidance Image J nicht vollständig gesampelt. Hierin liegt auch der wesentliche Unterschied zu der in dieser Diplomarbeit eingesetzten sequentiellen Methode, in welcher das Guidance Image J stets vollständig gesampelt wird.

Die beschriebene sequentielle Methode wurde in einigen Kursen ausländischer Universitäten zur Approximation des Joint Bilateral Upsampling Algorithmus eingesetzt [46]. Der Vorteil liegt darin, dass kein neuer Filter programmiert werden muss, da das Bild einfach per Nearest-Neighbour Upsampling auf die gewünschte Größe skaliert- und anschließend mit einem herkömmlichen Joint Bilateral Filter gefiltert werden kann. Die Methode kann daher auch mit bestehenden Joint Bilateral Median- und Guided Image Filtern durchgeführt werden. Wäre in dieser Diplomarbeit hingegen das Joint Bilateral Upsampling implementiert worden, so hätten ebenfalls entsprechende „Joint Bilateral Median Upsampler“ und „Guided Image

„Upsampler“ entworfen werden müssen (die Begriffe wurden unter Anführungszeichen gesetzt, da sie vom Autor dieser Diplomarbeit stammen und in anderen wissenschaftlichen Publikationen nicht verwendet werden), um Ergebnisse vergleichen zu können. Eine Analyse des Guided Image Filters ergab, dass eine Konvertierung zum „Guided Image Upsampler“ höchstwahrscheinlich möglich ist. Im Rahmen dieser Diplomarbeit wurde jedoch von der Ausarbeitung eines solchen Filters abgesehen. Kopfs Algorithmus besitzt potentiell einen Performancevorteil gegenüber der beschriebenen sequentiellen Methode, da bei gleichem Filterradius weniger Positionen in der Berechnung berücksichtigt werden. Bei Verwendung eines Joint Bilateral Filter mit konstanter Laufzeit bezüglich der Kernelgröße (siehe Kapitel 3) scheint dieser Vorteil jedoch nicht relevant zu sein.

Die Evaluierung wurde auf einem Subset des Middlebury-Sets durchgeführt. Es wurden die Bilder „cones“, „teddy“, „tsukuba“ und „venus“ gewählt, da diese Bilder auch im MATLAB-Referenzcode des Stereo-Matchers eingesetzt werden um dessen Leistung zu demonstrieren. Der MATLAB-Quellcode konnte somit ohne Modifikation übernommen werden. Alle vier Kamerabildpaare wurden vor Anwendung des Stereo Matchers um die Faktoren 2x, 4x, 6x und 8x reduziert. Das anschließende Upsampling der Tiefenkarten erfolgt durch ein C++ Programm, welches die Ausgaben des Stereo Matchers per Nearest-Neighbour Interpolation skaliert und auf das Ergebnis, die im Rahmen dieser Diplomarbeit entwickelten kantenerhaltenden Filter anwendet. Wie bereits in Abschnitt 6.2 wurden pro Filter mehrere 100 Parametersets automatisiert getestet, um möglichst hochwertige Ergebnisse zu erreichen. Aufgrund der geringen Größe des Testsets sind die in diesem Abschnitt ermittelten Ergebnisse nicht zur Generalisierung geeignet. Sie sollen vielmehr zeigen welche qualitativen Vorteile bei den genannten Testbildern feststellbar waren.

Abb. 71 (nächste Seite) zeigt das im Rahmen dieser Evaluierung berechnete MSE-Profil. Es kann festgestellt werden, dass durch die Anwendung eines kantenerhaltenden Glättungsfilters mit Guidance Image die Ergebnisse bei allen getesteten Upsampling Faktoren signifikant verbessert werden konnten. Bei den im Rahmen dieser Evaluierung getesteten Tiefenkarten konnte der Guided Image Filter signifikant bessere Ergebnisse als Joint Bilateral Filter und Joint Bilateral Medianfilter erreichen. Bei den traditionellen Upsamplingverfahren ohne kantenerhaltender Filterung war das Bicubic Upsampling dem Nearest Neighbour Upsampling signifikant überlegen. Abb. 73 (Seite 99) zeigt mit welchen Filterparametersets die besten Ergebnisse erzielt werden konnten.

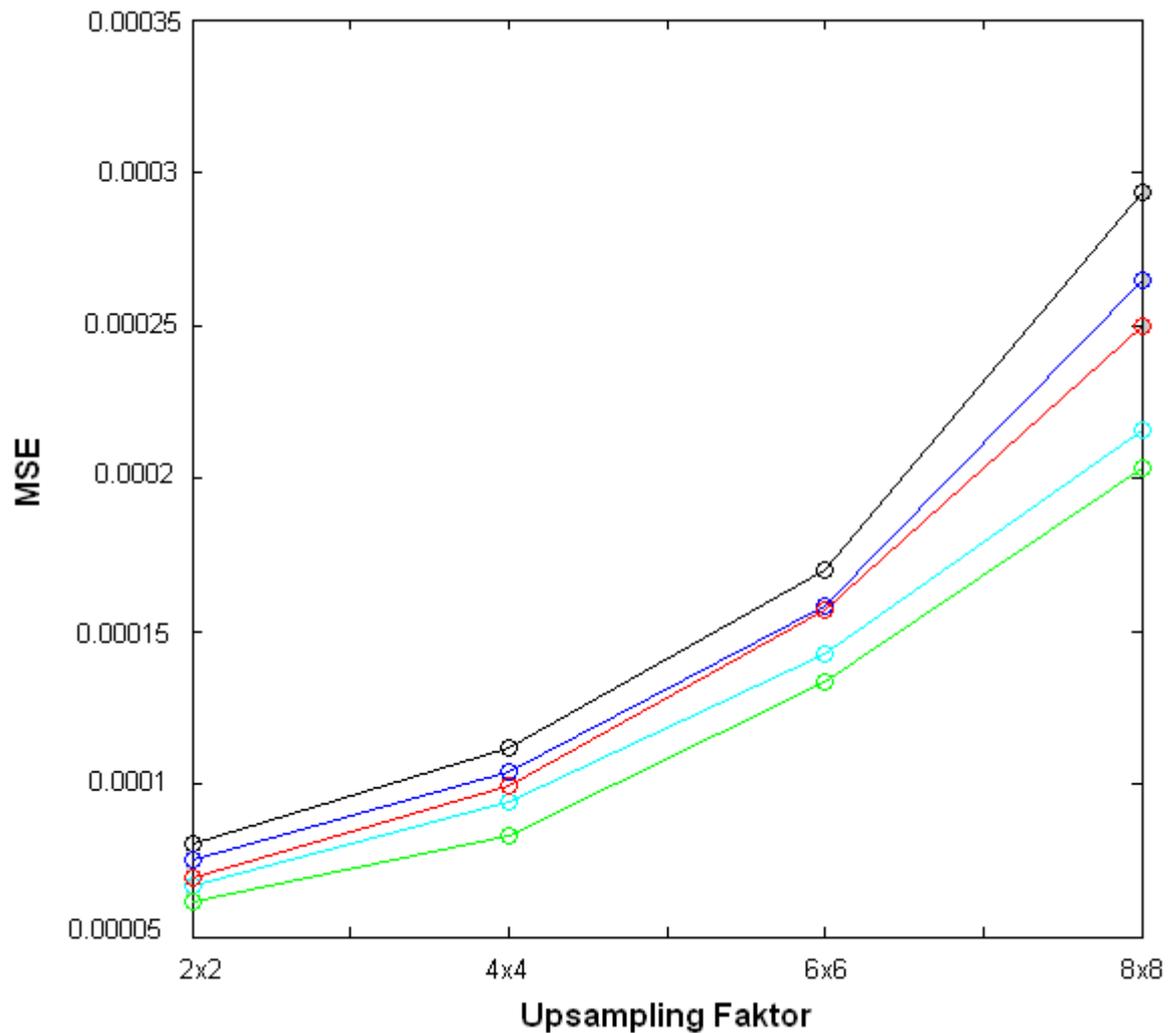


Abb. 71: MSE Profil bei verschiedenen Upsampling Faktoren
schwarz: Nearest Neighbour Upsampling
blau: Bicubic Upsampling
cyan: Nearest Neighbour Upsampling + Joint Bilateral Filter
rot: Nearest Neighbour Upsampling + Joint Bilateral Median Filter
grün: Nearest Neighbour Upsampling + Guided Image Filter

Abb. 72 (nächste Seite) stellt die Ergebnisse der getesteten Verfahren visuell gegenüber. Bei Nearest Neighbour Upsampling ohne anschließende Glättung ist starke Blockbildung in den Kantenbereichen zu erkennen. Durch bikubisches Upsampling kann die Blockbildung zwar deutlich reduziert werden, im Gegenzug verlieren jedoch alle Kanten signifikant an Schärfe. Durch Nearest Neighbour Upsampling mit anschließender Glättung durch einen Joint Bilateral Median Filter konnte die Blockbildung entfernt werden ohne die Kantenschärfe zu beeinträchtigen. Joint Bilateral Filter und Guided Image Filter reduzierten hingegen in vielen Regionen die Kantenschärfe. Im Gegensatz zum Joint Bilateral Median Filter konnten sie jedoch Lampenschirm und –stangen besser erhalten.

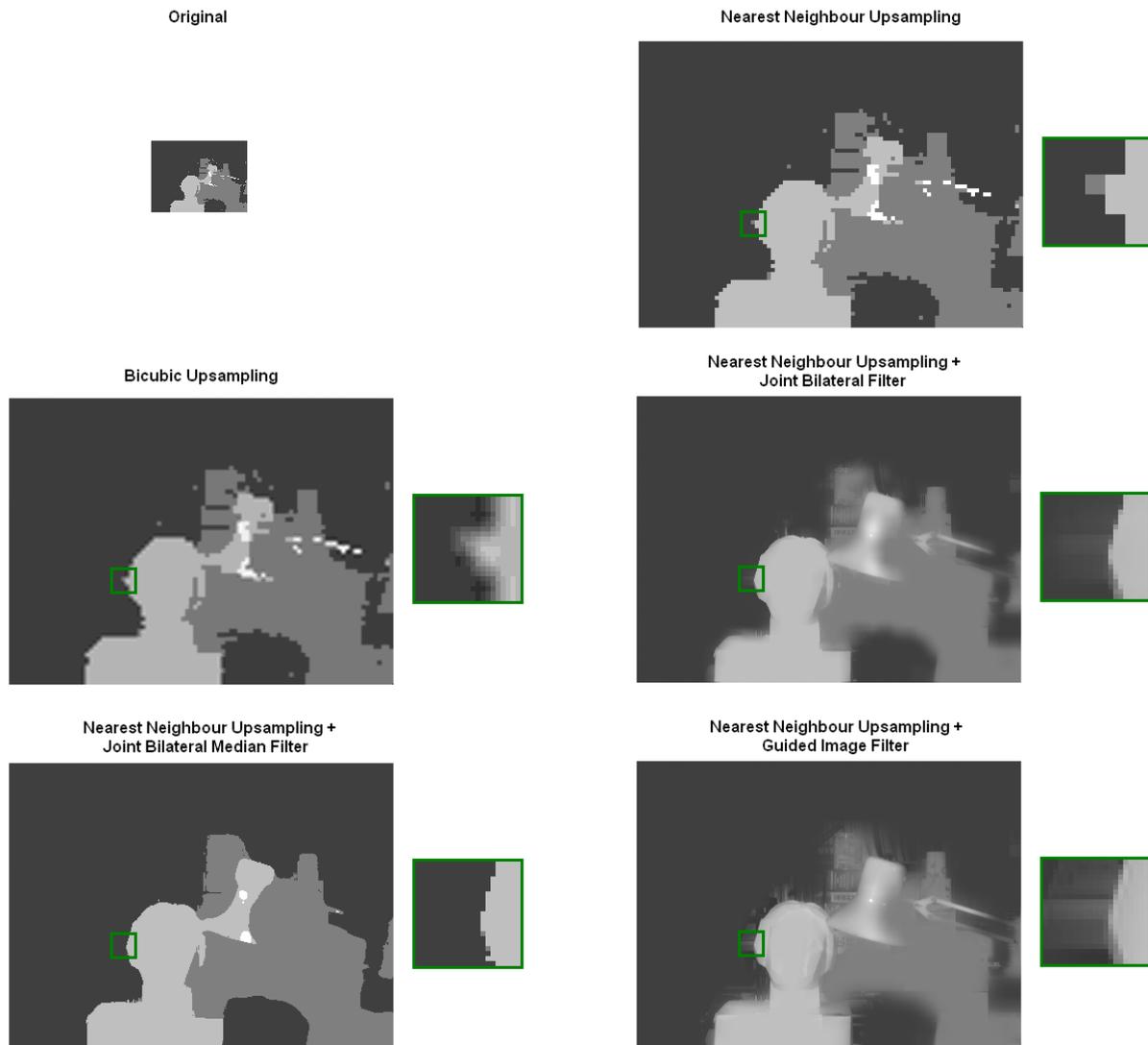


Abb. 72: 4x4 Upsampling von „Tsukuba“

	2x2 Upsampling	4x4 Upsampling	6x6 Upsampling	8x8 Upsampling
Nearest Neighbour + Joint Bilateral	$r = 9$ $\sigma_s = 6$ $\sigma_l = 0.12$	$r = 9$ $\sigma_s = 7$ $\sigma_l = 0.22$	$r = 9$ $\sigma_s = 7$ $\sigma_l = 0.23$	$r = 9$ $\sigma_s = 7$ $\sigma_l = 0.24$
Nearest Neighbour + Joint Bilateral Med.	$r = 7$ $\sigma_s = 6$ $\sigma_l = 0.15$	$r = 9$ $\sigma_s = 7$ $\sigma_l = 0.21$	$r = 9$ $\sigma_s = 7$ $\sigma_l = 0.24$	$r = 9$ $\sigma_s = 7$ $\sigma_l = 0.23$
Nearest Neighbour + Guided Image Filter	$r = 9$ $\varepsilon = 0.0004$	$r = 9$ $\varepsilon = 0.0032$	$r = 9$ $\varepsilon = 0.018$	$r = 9$ $\varepsilon = 0.02$

Abb. 73: Optimale Filterparameter

7. Zusammenfassung

Diese Diplomarbeit beschäftigte sich mit kantenerhaltenden Filtern und deren Applikation für Computer Vision-Aufgaben.

Nach einer kurzen Einführung (Kapitel 1) wurden in Kapitel 2 zunächst drei traditionelle Glättungsfilter (Mittelwertfilter, Gauß-Filter, Median-Filter) vorgestellt. Es wurde gezeigt, dass diese Filter die Bildschärfe beeinträchtigen, da Intensitätsabstände der Pixel nicht berücksichtigt werden. Aufbauend darauf wurden Bilateral Filter, Bilateral Median Filter und Guided Image Filter eingeführt, sowie deren kantenerhaltende Funktion demonstriert. Es wurde erwähnt, dass der Guided Image Filter ein eigenständiger Filter und nicht etwa nur eine effiziente Optimierung des Bilateral Filters ist.

Kapitel 3 beschäftigte sich mit der Implementierung der Filter. Für jeden Filter, abgesehen vom Guided Image Filter, wurde zunächst ein naiver Algorithmus gezeigt, der schrittweise optimiert wurde. Bei allen Filtern, abgesehen vom Bilateral Median Filter, wurde ein Algorithmus mit konstanter Laufzeit bezüglich der Kernelgröße präsentiert. Weiters wurde anhand des Bilateral Filters gezeigt, dass Approximationen oft erhebliche Performancegewinne bringen können ohne die Qualität des Ergebnisses signifikant zu beeinträchtigen.

Kapitel 4 führte den Optimierungsgedanken weiter und demonstrierte Methoden zur Parallelisierung der vorgestellten Algorithmen. Zunächst wurde die CUDA-Architektur moderner NVIDIA-Grafikkarten vorgestellt und anschließend gezeigt, wie der $O(1)$ -Mittelwertfilter parallel auf dieser Plattform implementiert werden kann.

In Kapitel 5 wurde ein interaktiver Segmentierungsalgorithmus vorgestellt, welcher kantenerhaltende Filter zur Glättung des Cost Volumes einsetzt. Desweiteren wurde ein modifizierter auf dem Guided Image Filter basierender Paint-Selection Algorithmus präsentiert.

Kapitel 6 zeigte, dass Tiefenkarten stereoskopischer Videos durch kantenerhaltende Filterung optimiert werden können. Außerdem wurde gezeigt, dass durch den Einsatz kantenerhaltender Filter im Upsampling-Prozess die Qualität der Skalierung signifikant verbessert werden kann.

Die in Kapitel 5 und 6 präsentierten Anwendungen konnten selbstverständlich nur einen kleinen Ausschnitt der möglichen Einsatzgebiete kantenerhaltender Filterung abdecken. Abschließend sollen einige weitere interessante Anwendungen genannt werden:

Rekursive Filterung: Bei der rekursiven Filterung wird der kantenerhaltende Filter mehrmals hintereinander aufgerufen, wobei ab dem zweiten Aufruf das Ergebnis des vorhergehenden Aufrufs gefiltert wird. Durch die mehrmalige Filterung wird der Kontrast in Bereichen mit bereits niedrigem Kontrast schrittweise reduziert, während hohe Kontraste (Kanten) erhalten bleiben. Natürlich tritt dieser Effekt auch bei einfacher (nicht rekursiver) Anwendung des kantenerhaltenden Filters zu einem gewissen Grad auf. Durch wiederholte Anwendung kann er jedoch wesentlich verstärkt werden. Winnemöller et al. demonstrieren in ihrer Arbeit [47], dass durch rekursive bilaterale Filterung abstrahierte- und cartoonähnliche Bilder erzeugt werden können. Es ist anzunehmen, dass Bilateral Median Filter und Guided Image Filter ebenfalls für diese Aufgabe geeignet sind.

Joint Upsampling: Die Nutzung kantenerhaltender Filter im Upsampling-Prozess ist keineswegs auf den in Kapitel 6 vorgestellten Anwendungsfall (Upsampling von Tiefenkarten) beschränkt. Eine weitere interessante Anwendung ist beispielsweise die Colorisierung von Graustufenbildern [42]. Hierzu wird zunächst ein beliebiger Colorisierungsalgorithmus, beispielsweise [52] auf das verkleinerte Graustufenbild angewandt. Anschließend wird das colorierte Bild unter Einsatz des unskalierten Graustufenbildes als Guidance Image auf die gewünschte Größe skaliert. Es wird empfohlen die Colorisierung im YIQ-Farbraum durchzuführen, da in diesem Fall nur die beiden Chrominanzkanäle (I und Q) berechnet werden müssen. Der Luminanzkanal (Y) kann hingegen direkt aus dem Graustufenbild übernommen werden. Auch im Zusammenhang mit Graph-Cut basierender Bildsegmentierung kann Joint Upsampling effektiv eingesetzt werden [42].

Filterung anderer Strukturen: Kantenerhaltende Filterung beschränkt sich nicht ausschließlich auf die Glättung zweidimensionaler Bilddaten. Bereits in Kapitel 5 (Cost Volume) und Kapitel 6 (Tiefenkarte) wurde gezeigt, dass auch andere zweidimensionale Strukturen geglättet werden können. [49] zeigte, dass auch durch einen 3D-Scanner ermittelte Mesh-Daten bilateral gefiltert werden können. Selbst Bewegungsdaten von Menschen werden bilateral gefiltert [50].

Sonstiges: In [55] und [48] wird empfohlen modifizierte Bilateral Filter zur Entfernung von Bildrauschen einzusetzen. [51] zeigte, dass bilaterale Filter zur Entfernung von Kompressionsartefakten genutzt werden können. In [54] wurde ein auf dem Bilateral Filter

basierender Algorithmus zur Entfernung von „Specular Highlights“ demonstriert. Weiters zeigte [53], dass der bilaterale Filter auch zur Dynamikkompression von HDR (High Dynamic Range)-Bildern eingesetzt werden kann.

Literaturverzeichnis

- [1]: Tomasi, C., Manduchi, R.: Bilateral Filtering for Gray and Color Images. ICCV (1998)
- [2]: Francis, J.J, De Jager, G.: The Bilateral Median Filter. Proceedings of the 14th Symposium of the Pattern Recognition Association of South Africa (2003)
- [3]: He, K., Sun, J., Tang, X.: Guided Image Filtering. ECCV (2010)
- [4]: NVIDIA CUDA Architecture: Introduction and Overview. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf (2009) Aufgerufen am: 30.05.2011
- [5]: Wikipedia: Glättung. <http://de.wikipedia.org/wiki/Glättung> Aufgerufen am: 30.05.2011
- [6]: Burger, W., Burge, M.J.: Digitale Bildverarbeitung: Eine Einführung mit Java und ImageJ. Springer-Verlag (2005)
- [7]: Petschnigg, G., Agrawala, M., Hoppe, H., Szeliski, R., Cohen, M., Toyama, K.: Digital Photography with Flash and No-Flash Image Pairs. SIGGRAPH (2004)
- [8]: McClellan, J.H., Schafer, R., Yoder, M.: Signal Processing First. Pearson (2003)
- [9]: Deriche, R.: Recursively implementing the Gaussian and its derivatives. Proceedings of the 2nd International Conference on Image Processing (1992)
- [10]: Vliet, L., Young, I., Verbeek, P.: Recursive Gaussian derivative filters. Proceedings of the International Conference on Pattern Recognition (1998)
- [11]: Huang, T.S.: Two-Dimensional Signal Processing II: Transforms and Median Filters. Springer-Verlag (1981)
- [12]: Weiss, B.: Fast Median and Bilateral Filtering. ACM Transactions on Graphics (2006)
- [13]: Tibshirani, R.J.: Fast Computation of the Median by Successive Binning. <http://stat.stanford.edu/~ryantibs/median/> (2008). Aufgerufen am: 30.05.2011
- [14]: Kopp, M., Purgathofer, W.: Efficient 3x3 Median Filter Computations. Machine Graphics & Vision. (Vol.4, No. 1/2, 1995)
- [15]: Perreault, S., Hebert, P.: Median Filtering in Constant Time. IEEE Transactions on Image Processing. (2007)
- [16]: Paris, S., Durand, F.: A fast approximation of the bilateral filter using a signal processing approach. ECCV (2006)
- [17]: Porikli, F.: Constant time $O(1)$ bilateral filtering. CVPR (2008)
- [18]: Pham, T.Q, Vliet, L.J.: Separable Bilateral Filtering For Fast Video Processing. IEEE International Conference on Multimedia & Expo (2005)
- [19]: Fast $O(1)$ Bilateral Filtering using trigonometric range kernels. IEEE Transactions on Image processing (2011)

- [20]: Yu, W., Franchetti F., Hoe, J.C., Chang, Y.-J., Chen, T.: Fast Bilateral Filtering by adapting block size. ICIP (2010)
- [21]: Yang, Q., Tan, K.H., Ahuja, N.: Real-time O(1) bilateral filtering. CVPR (2009)
- [22]: Kirk, D.B., Hwu W.-M.W., Programming Massively Parallel Processors. Morgan Kaufmann (2010)
- [23]: NVIDIA CUDA C Programming Guide.
http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf (2010). Aufgerufen am: 30.05.2011
- [24]: ATI Stream Computing OpenCL Programming Guide.
http://developer.amd.com/gpu_assets/ATI_Stream_SDK_OpenCL_Programming_Guide.pdf (2010). Aufgerufen am: 30.05.2011
- [25]: The OpenCL Specification, Version 1.1.
<http://www.khronos.org/registry/cl/specs/openc1-1.1.pdf> (2010). Aufgerufen am: 30.05.2011
- [26]: Sanders, J., Kandrot, E.: CUDA By Example – An Introduction to General-Purpose GPU Programming. Addison-Wesley (2011)
- [27]: Ruetsch, G., Micikevicius, P.: Optimizing Matrix Transpose in CUDA.
<http://www.cs.colostate.edu/~cs675/MatrixTranspose.pdf> (2009). Aufgerufen am: 30.05.2011
- [28]: Vineet, V., Narayanan, P.: Cuda cuts: Fast graph cuts on the GPU. IEEE Conference on Computer Vision and Pattern Recognition (2008)
- [29]: Shams, R., Kennedy, R.A.: Efficient Histogram Algorithms for NVIDIA CUDA Compatible Devices. Proceedings of the International Conference on Signal Processing and Communications Systems (2007)
- [30]: Satish, N., Harris, M., Garland, M.: Designing Efficient Sorting Algorithms for Manycore GPUs. Proceedings of IPDPS (2009)
- [31]: Pham, V., Vo, P., Hung, V.T., Bac, L.H.: GPU implementation of Extended Gaussian Mixture Model For Background Subtraction. RIVF (2010)
- [32]: Harish, P., Narayanan, P.J.: Accelerating large graph algorithms on the GPU using CUDA. IEEE High Performance Computing (2007)
- [33]: Manavski, S.A.: CUDA Compatible GPU as an efficient hardware accelerator for AES cryptography. IEEE International Conference on Signal Processing and Communications (2007)
- [34]: Zlatuska, M., Havran, V.: Ray Tracing on a GPU with CUDA -- *Comparative Study of Three Algorithms*. Proceedings of WSCG (2010)
- [35]: Lahabar, S., Agrawal, P., Narayanan, P.J.: High performance pattern recognition on GPU. Proceedings of NCVPRIPG (2008)

- [36]: Patel, P., Wong, J., Tatikonda, M., Marczewski, J.: JPEG Compression algorithm using CUDA. http://www.eecg.toronto.edu/~moshovos/CUDA08/arx/JPEG_report.pdf. Aufgerufen am: 30.05.2011
- [37]: Lee, V.W., Kim, C., Chhugani, J. Deisher, M., Kim, Daehyun, Nguyen, D., Satish, N., Smelyanskiy, M., Chennupati, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. International Symposium of Computer Architecture (2010)
- [38]: Rhemann, C., Hosni, A., Bleyer, M., Rother, C., Gelautz, M.: Fast Cost-Volume Filtering for Visual Correspondence and Beyond. CVPR (2011)
- [39]: Rother, C., Kolmogorov, V., Blake, A.: Grab-Cut – Interactive Foreground Extraction using Iterated Graph Cuts. ACM Transactions on Graphics (2004)
- [40]: Middlebury Stereo Vision Page. <http://vision.middlebury.edu/stereo/> Aufgerufen am: 30.05.2011
- [41]: Liu, J., Sun, J., Shum, H.-Y.: Paint-Selection. SIGGRAPH (2009)
- [42]: Kopf, J., Cohen, M., Lischinski, D., Uyttendaele, M.: Joint bilateral upsampling. SIGGRAPH (2007)
- [43]: Bleyer, M., Gelautz, M.: Simple but effective tree structures for dynamic programming-based stereo matching. VISAPP (2008)
- [44]: Wang, L., Jin, H., Yang, R., Gong, M.: Stereoscopic inpainting: Joint Color and Depth Completion from stereo images. CVPR (2008)
- [45]: Hu, H., De Haan, G., Trained bilateral filters and applications to coding artifacts reduction. Proceedings of ICIP (2007)
- [46]: Assignment #10: Joint Bilateral Upsampling. http://740-2.cs.nthu.edu.tw/~htchen/aip10/P10/P10_9862547/ Aufgerufen am: 30.05.2011
- [47]: Winnemöller, H., Olsen, S.C., Gooch, B.: Real-Time Video Abstraction. SIGGRAPH (2006)
- [48]: Viyaya, G., Vasudevan, V.: Bilateral Filtering using Modified Fuzzy Clustering for Image Denoising. International Journal on Computer Science and Engineering (2011)
- [49]: Fleishman, S., Drori, I., Cohen-Or, D.: Bilateral Mesh Denoising. ACM Transactions on Graphics (2003)
- [50]: Courty, N.: Bilateral Human Motion Filtering. 16th European Signal Processing Conference, Lausanne, Switzerland. (2008)
- [51]: Wan, S., Mrak, M., Ramzan, N., Izquierdo, E.: Perceptually Adaptive Joint Deringing-Deblocking Filtering for Scalable Video Transmission over Wireless Networks. SP:IC (2007)
- [52]: Levin, A., Lischinski, D., Weiss, Y.: Colorization using optimization. SIGGRAPH (2004)

- [53]: Durand, F., Dorsey, J.: Fast bilateral filtering for the display of High-Dynamic-Range images. SIGGRAPH (2002)
- [54]: Yang, Q., Wang, S., Ahuja, N.: Real-Time Specular Highlight Removal using Bilateral Filtering. ECCV (2010)
- [55]: Vijaya, G., Vasudevan, V.: A novel Noise Reduction Method using Double Bilateral Filtering. European Journal of Scientific Research (2010)
- [56]: Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. IJCV (2002)