

Promon200

---

# Bachelor Thesis

Fachhochschule Nordwestschweiz - Studiengang Informatik

Kunde:	Prof. Dr. Christoph Stamm
Betreuer:	Martin Schindler
Experte:	Charles Zehnder
Studierende:	Claudio Wettstein Daniel Suter
Revision:	1.00
Datum:	18. August 2011

## Revisionen

VERSION	DATUM	AUTOR	KOMMENTAR
0.01	07.04.2011	Claudio Wettstein	Erstellen der Vorlage
0.02	07.04.2011	Daniel Suter	Kapitel Performance
0.03	28.04.2011	Claudio Wettstein	Kapitel Average Brightness
0.04	01.05.2011	Daniel Suter	Kapitel Velocity
0.05	13.06.2011	Claudio Wettstein	Kapitel Average Brightness erweitert
0.06	14.07.2011	Daniel Suter	Kapitel Pattern matching
0.07	19.07.2011	Claudio Wettstein	Kapitel Frequenzbestimmung
0.08	19.07.2011	Daniel Suter	Kapitel Waver Beschichtung
0.09	26.07.2011	Claudio Wettstein	Kapitel Deckeltest
0.10	26.07.2011	Daniel Suter	Kapitel Integration von Cuda in Promon
0.11	11.08.2011	Daniel Suter	Kapitel Entscheidungskomponente
0.12	11.08.2011	Claudio Wettstein	Kapitel Automatische Konfiguration und Cuda Integration
0.13	15.08.2011	Daniel Suter	Komplette Überarbeitung der Dokumentation
0.14	15.08.2011	Claudio Wettstein	Komplette Überarbeitung der Dokumentation
0.15	17.08.2011	Daniel Suter	Kapitel Framestep
0.16	17.08.2011	Claudio Wettstein	Kapitel Reflexion
0.17	18.08.2011	Claudio Wettstein	Abstract und Fazit
0.90	18.08.2011	C. W. und D. S.	Draft
1.00	18.08.2011	C. W. und D. S.	Finale Version

## Abstract

Diese Arbeit baut auf dem bestehenden Promon200 auf, welches verwendet wird, um in von Hochgeschwindigkeitskameras gefilmten, vollautomatisierten Produktionssystemen Fehler mittels Bildanalyse zu erkennen. Im Rahmen dieser Arbeit wurde untersucht, ob sich eine Auslagerung der Bildanalyse auf die Grafikkarte mittels Cuda lohnt. Dabei wurde erkannt, dass gut parallelisierbare und besonders rechenintensive Aufgaben einen Geschwindigkeitsvorteil aufweisen, wenn sie auf der Grafikkarte gerechnet werden. Zusätzlich wurden für zwei Produktionssysteme Filter zur Fehlererkennung entwickelt. Diese erkennen inkorrekte Zustände, ohne falsche Events bei korrekten Zuständen zu werfen. Um die Framerate der Hochgeschwindigkeitskamera erreichen zu können, wurden automatische und nicht automatische Massnahmen zur Laufzeitoptimierung der Filter analysiert und umgesetzt. Bei der Analyse der bestehenden Massnahmen wurde festgestellt, dass C#-Filter auf eine Pixelstep-Erhöhung mit bei weitem höheren Geschwindigkeitsvorteilen reagieren als ihre Cuda Konkurrenten.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>6</b>
<b>2</b>	<b>Promon200</b>	<b>7</b>
2.1	Architektur . . . . .	8
<b>3</b>	<b>Testsuite</b>	<b>11</b>
3.1	Zeitmessung . . . . .	11
3.1.1	Probleme . . . . .	12
3.2	Videoauswahl . . . . .	13
3.3	Gebrauch . . . . .	13
3.3.1	Konfiguration . . . . .	13
3.3.2	Ausgaben . . . . .	14
<b>4</b>	<b>Filter mittels Cuda parallelisieren</b>	<b>16</b>
4.1	NVIDIA Cuda Architektur . . . . .	16
4.2	Invert-Filter . . . . .	16
4.2.1	Allozieren von GRAM . . . . .	17
4.2.2	Memory Copy von Host nach Device und zurück . . . . .	18
4.2.3	Parallelisierung . . . . .	19
4.2.4	Schlussfolgerung . . . . .	19
4.3	Average Brightness . . . . .	20
4.3.1	Pro Spalte . . . . .	20
4.3.2	Pro Spalte mit Faktorisierung . . . . .	20
4.3.3	Prefix Scan . . . . .	21
4.3.4	Prefix Scan mit zweier Potenz . . . . .	24
4.3.5	Prefix Scan pro Zeile . . . . .	24
4.3.6	Vergleich . . . . .	24
4.4	Velocity-Filter . . . . .	26
4.4.1	Komplette Binarisierung . . . . .	26
4.4.2	Binarisierung erste Zeile . . . . .	26
4.4.3	Vergleich . . . . .	27
4.5	Pattern Matching . . . . .	28
4.5.1	Prefix Sum . . . . .	28
4.5.2	Thread pro Zeile . . . . .	29
4.5.3	Asynchrone Matchbestimmung . . . . .	30
4.5.4	Vergleich . . . . .	31
4.6	Integration von Cuda in Promon200 . . . . .	32
4.6.1	Generelle Änderungen . . . . .	32
4.6.2	Composite Filter . . . . .	33
4.7	Frames parallelisieren . . . . .	36

<b>5</b>	<b>Video Analyse</b>	<b>37</b>
5.1	Analysezeitpunkt bestimmen . . . . .	37
5.1.1	False-True-Trigger . . . . .	37
5.1.2	Frequenz messen . . . . .	38
5.2	Klassifizierung . . . . .	39
5.2.1	Waver Beschichtung . . . . .	39
5.2.2	Deckelmontage . . . . .	43
5.3	Automatische Konfiguration . . . . .	46
5.3.1	Konzept . . . . .	46
5.3.2	GUI . . . . .	47
5.3.3	Implementation . . . . .	48
5.3.4	Ausblick . . . . .	49
<b>6</b>	<b>Entscheidungskomponente</b>	<b>50</b>
6.1	Optimierungen ohne Benutzerinteraktion (Analyse) . . . . .	50
6.1.1	Pixel überspringen . . . . .	50
6.1.2	Frames überspringen . . . . .	52
6.2	Optimierungen mit Benutzeraktion . . . . .	54
6.2.1	GUI . . . . .	54
6.2.2	Konzept . . . . .	55
6.2.3	Implementation . . . . .	55
<b>7</b>	<b>Reflexion</b>	<b>57</b>
7.1	Zusammenarbeit . . . . .	57
7.2	Betreuung . . . . .	57
7.3	Zeitplan . . . . .	58
	<b>Bibliografie</b>	<b>59</b>
	<b>Listings</b>	<b>60</b>
	Abbildungsverzeichnis . . . . .	61
	Tabellenverzeichnis . . . . .	62
	Codelisting . . . . .	63
	<b>Anhang</b>	<b>64</b>
	Glossar . . . . .	64
	Ehrlichkeitserklärung . . . . .	65
	Originalauftrag . . . . .	66
	Erstellte und bearbeitete Dateien . . . . .	71

# 1 Einleitung

Dieser Bericht ist gedacht für die Betreuer und Experten dieser Bachelor Thesis, sowie zukünftige Studierende, welche an *Promon200* weiterarbeiten. Er enthält die Resultate und konzeptionellen Überlegungen die gemacht wurden, um *Promon200*<sup>1</sup> zu erweitern.

*Promon200* ist eine Software zur Überwachung von vollautomatisierten Produktionsanlagen, die durch eine Hochgeschwindigkeitskamera gefilmt werden. Es erkennt Produktionsfehler in Echtzeit, damit fehlerhafte Produkte aussortiert werden können. Dazu werden vom Benutzer Bereiche festgelegt, in welchen Filter mittels Bildanalyse fehlerhafte Abläufe erkennen. Dem Benutzer steht eine Breite Palette von Filtern und Funktionen zur Verfügung, damit er dieses Ziel erreichen kann.

Die Auslagerung von Berechnungen auf die Grafikkarte, welche diese hoch-parallel ausführen kann, liegt im Trend. Die bestehenden Filter werden auf der CPU ausgeführt. Multicore Prozessoren werden bereits genutzt. Es wird analysiert, ob eine Verarbeitung dieser Filter auf der Grafikkarte Geschwindigkeitsvorteile bringt. Dazu werden drei bestehende Filter, namentlich *Average Brightness*, *Velocity* und *Pattern Matching* mittels Cuda von NVIDIA umgesetzt. Das Ziel ist eine Verarbeitungszeit von maximal 5 ms pro Filter bei einer Grösse von 600x400 Pixeln. Die Geschwindigkeitsmessungen müssen unabhängig von *Promon200* und möglichst automatisiert durchgeführt werden.

Im zweiten Teil werden Videos von automatisierten Produktionsabläufen ausgewählt. Es werden Filter mittels Cuda umgesetzt, welche Fehler in den Abläufen erkennen. Zusätzlich wird eine Komponente entwickelt, um auf einem Video die Frequenz eines sich wiederholenden Vorgangs subframegenau zu bestimmen.

Im letzten Teil dieser Arbeit wird eine Entscheidungskomponente ausgearbeitet, welche hilft, die Framerate bei ungenügend performanter Hardware oder schlechter Konfiguration auf das gewünschte Niveau zu bringen. Es sind bereits zwei Massnahmen integriert. Einerseits ein Pixelstep, welcher jedes n-te Pixel überspringt und ein Framestep, welcher *Promon200* nur jedes n-te Frame verarbeiten lässt. Es werden weitere Massnahmen ausgearbeitet. Die bestehenden und neuen Massnahmen werden auf ihren Performancegewinn sowie auf mögliche Veränderungen der Resultate analysiert.

Um die Lesbarkeit zu erhöhen werden projektspezifische Namen wie Methoden- und Klassennamen kursiv geschrieben. Der Originalauftrag befindet sich im Anhang dieses Dokumentes.

---

<sup>1</sup><http://www.fhnw.ch/technik/imvs/forschung/projekte/promon200/Promon>

## 2 Promon200

Da diese Arbeit auf *Promon200*<sup>2</sup> basiert, folgt hier eine kurze Beschreibung der Software. Die Applikation ist in drei Teile gegliedert (siehe Abbildung 2.1). In der *Toolbox* unten sind alle zur Verfügung stehenden Filter und Kontrollelemente aufgelistet. Filter analysieren oder bearbeiten das aktuelle Frame und geben Resultate als Ausgabewerte zurück. Kontrollelemente dienen zur weiteren Verarbeitung von Ausgabewerten. So kann z.B. ein Ausgabewert eines Filters mit Hilfe eines Kontrollelements auf einen Grenzwert geprüft werden.

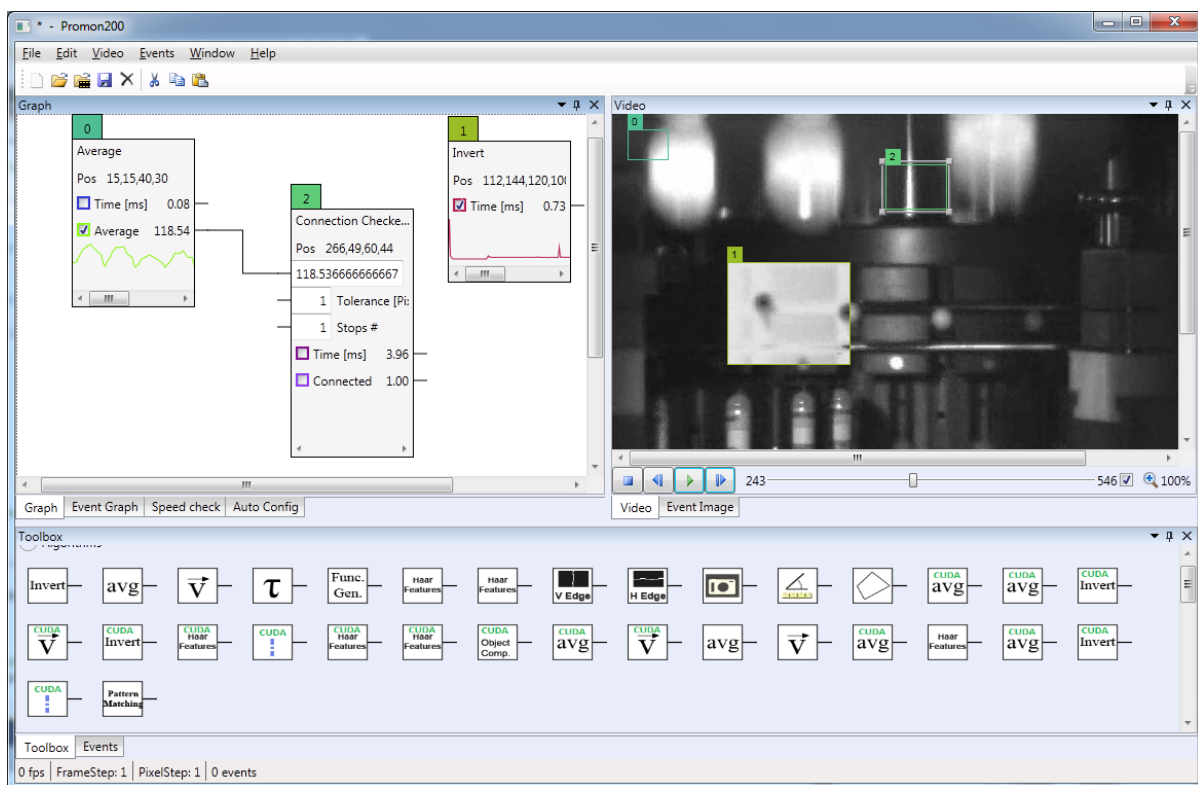


Abbildung 2.1: Das Promon200 mit platzierten Filtern

Filter und Kontrollelemente können per Drag & Drop in das *Graph* Tab oben links gezogen werden. Dort können Filterinformationen abgelesen, Eingabewerte verändert und Filterausgänge mit Filtereingängen verbunden werden. Durch Verbindungen können Ausgabewerte eines Filters als Eingabewerte eines anderen Filters verwendet werden. Es gibt auch Eingänge, welche ein Eingabebild erwarten. Diese können mit Hilfe des *FrameCapture* Control zur Verfügung gestellt werden. Dies erfordert eine Verbindung wie in Abbildung 2.2 zu sehen ist.

<sup>2</sup><http://www.fhnw.ch/technik/imvs/forschung/projekte/promon200/Promon>

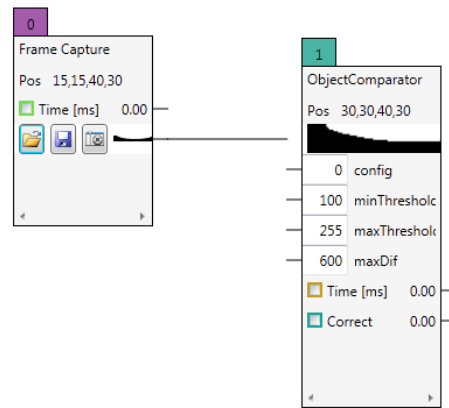


Abbildung 2.2: Verbindung um Ausgabebild von Control zu übernehmen

Auf der rechten Seite wird das aktuelle Frame angezeigt. Dieses kann direkt von einer Kamera oder von einem geladenen Video kommen. Wird ein Video geladen, kann mit Hilfe der Navigationsschaltflächen durch das Video navigiert werden. Zudem ist auf der rechten Seite auch ersichtlich, wo sich welcher Filter befindet. Grösse und Position können dort direkt angepasst werden.

Unten kann auf das Tab *Events* gewechselt werden. Dieses zeigt chronologisch alle aufgetretenen *Events* an und lässt den Benutzer zu diesen Frames springen um die Situation zu analysieren. *Events* werden verwendet um inkorrekte Produktionsabläufe anzuzeigen und werden vom *Trigger Control* ausgelöst.

## 2.1 Architektur

Nachfolgend wird der für diese Arbeit wesentliche Teil der Architektur des bestehenden *Promon200* aufgezeigt.

### Allgemein

Das *MainWindow* fungiert als Koordinator. Um ein Video zu öffnen wird über das *PromonDocument* auf den *VideoProcessor* zugegriffen (siehe Abbildung 2.3). Der *VideoProcessor* ist für das Auslesen von Videos und Streams zuständig. Für das Navigieren durch das Video verwendet das *VideoPane* auch den *VideoProcessor*. Hat der *VideoProcessor* das neue Frame geladen wird er auf dem *GraphPane* die Filterverarbeitung auslösen. Das *GraphPane* hält alle konfigurierten Filter in einer Liste. Neue Frames werden seriell gelesen, die Filter werden aber parallel auf dieses Frame aufgerufen.

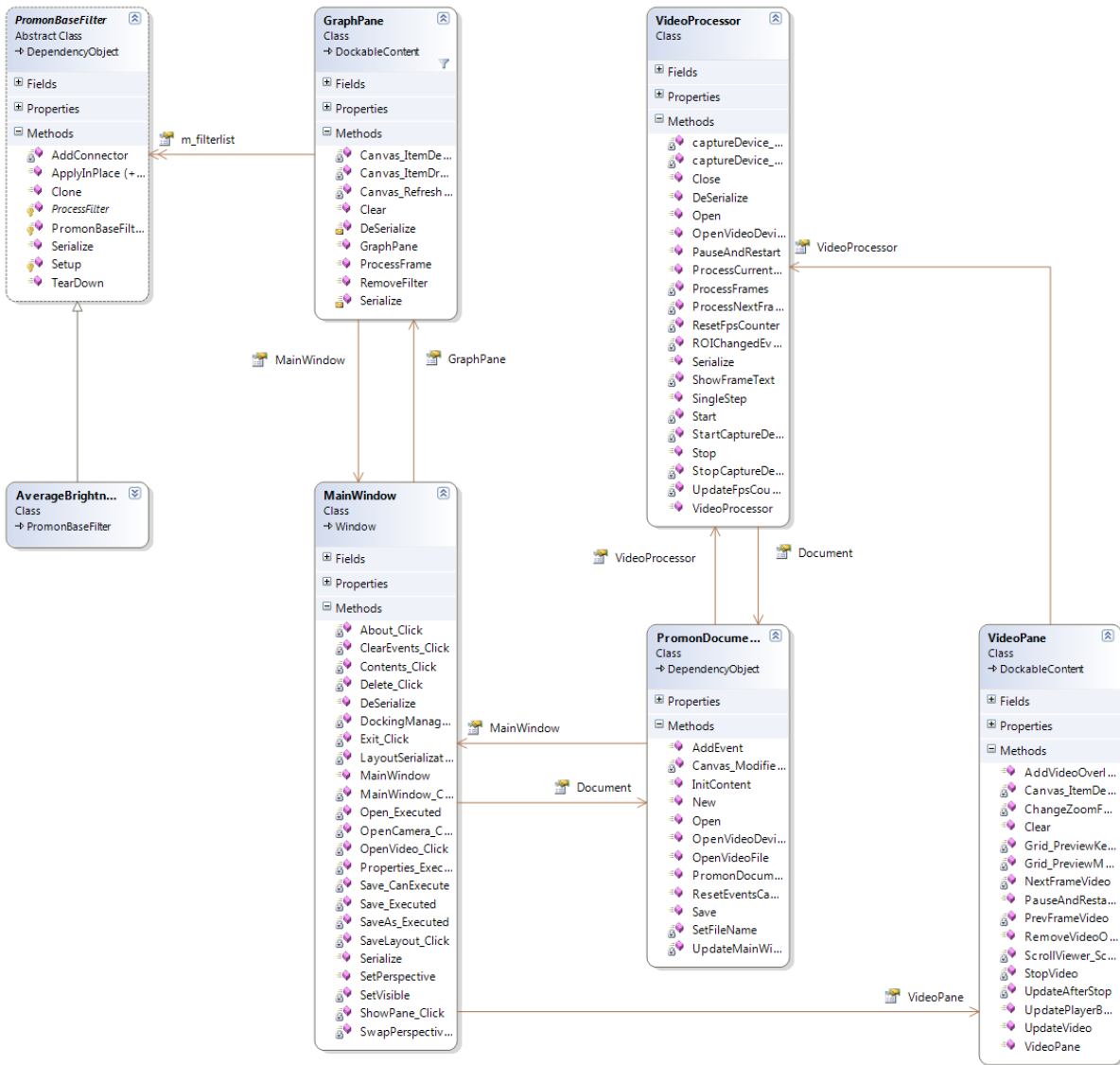


Abbildung 2.3: Ausschnitt aus der bestehenden Architektur

## Filter

Alle Filter erben von der Basisklasse *PromonBaseFilter*, welche bereits weitreichende Funktionalitäten zur Verfügung stellt. Filterketten (alle Filter inklusive Verbindungen, Eingabe- und Ausgabewerte) können gespeichert und geladen werden. Filterbereich und Grösse können im GUI angepasst und Verbindungen erstellt werden. All diese Funktionalitäten werden unterstützt, wenn von *PromonBaseFilter* geerbt wird. Notwendig ist in der eigenen Klasse nur die Definition von Ein- und Ausgängen, sowie das Implementieren der Filterfunktionalität in der Methode *ProcessFilter*.

## Verbindungen

Für Ein- und Ausgänge werden die *WPF*<sup>3</sup> *DependencyProperties*<sup>4</sup> verwendet, wodurch das GUI über aktuelle Werte informiert wird. Eingabe- und Ausgabewerte sind vom Typ *double*. Dadurch können beliebige Ein- und Ausgänge miteinander verbunden werden. Ausgänge mit einer Verbindung schreiben ihre Werte in eine *Queue*. Dies ermöglicht es Kontrollelementen abzuwarten, bis alle asynchron angekommenen Eingabewerte vom gleichen Frame vorhanden sind, bevor die Werte synchron weiterverarbeitet werden.

---

<sup>3</sup><http://msdn.microsoft.com/en-us/library/ms754130.aspx>

<sup>4</sup><http://msdn.microsoft.com/en-us/library/system.windows.dependencyproperty.aspx>

## 3 Testsuite

Um die Laufzeit der bestehenden wie auch der zu entwickelnden Filter messen zu können, wird eine Testsuite erstellt. Diese ermöglicht es Auswirkungen von Änderungen auf die Performance der Filter effizient feststellen und diese dokumentieren zu können. Es handelt sich um ein Konsolenprogramm, welches konfigurierte Filter auf ausgewählte Videos anwendet und die benötigte Zeit misst. Für die konfigurierten Filter werden die Messwerte angezeigt und abgespeichert. Die Testsuite misst nur die Laufzeit der benötigten Filter und gibt keine Auskunft über die korrekte Funktionsweise der einzelnen Filter.

Alle Zeitmessungen basieren auf folgendem Testsystem :

**Max Clock Speed:** 3068 MHz

**Number of cores:** 4

**Address width:** 64 Bit

**Graphic card:** NVIDIA GeForce GTX 480

**Resolution x Color depth:** 1680 x 1050 x 4'294'967'296 colors

**System:** x64-based PC

**Physical memory:** 5.99 GB

### 3.1 Zeitmessung

Da hier Filter erarbeitet werden, welche bei einer Filtergrösse von 600x400 Pixel eine maximale Laufzeit von 5 ms haben dürfen, ist eine genaue Zeitmessung besonders wichtig. Zur Zeitmessung stehen bereits verschiedene Tools zur Verfügung, welche analysiert wurden.

#### Stopwatch

Ein bekannter Vertreter für die Zeitmessung wird vom .NET Framework durch die *Stopwatch*<sup>5</sup> zur Verfügung gestellt. Die Genauigkeit kann mit Hilfe des Codes in Listing 3.1 analysiert werden.

```
1     long frequency = Stopwatch.Frequency;
2     Console.WriteLine("  Timer frequency in ticks per second =
      {0}", frequency);
3     long nanosecPerTick = (1000L*1000L*1000L) / frequency;
4     Console.WriteLine("  Timer is accurate within {0} nanoseconds
      ", nanosecPerTick);
```

Listing 3.1: Genauigkeit von Stopwatch

Auf dem Testsystem wurde eine Genauigkeit von 333 Nanosekunden ausgegeben.

<sup>5</sup><http://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch.aspx>

## Kernel32.dll

Ein weiterer bekannter Vertreter für Zeitmessungen ist die *QueryPerformanceCounter*<sup>6</sup> Methode der Library "Kernel32.dll". Sie liefert die verstrichenen Prozessor Ticks zurück.

Um die verstrichene Zeit zu erhalten werden die verstrichenen Ticks durch die Frequenz geteilt. Auch hier resultiert auf dem Testsystem eine Genauigkeit von 333 Nanosekunden.

Da beide Variante die gleiche Genauigkeit aufweisen, könnten beide Systeme für die Zeitmessung verwendet werden. Für diese Arbeit wurde die Stopwatch gewählt, da diese durch ihre Methoden das Messen vereinfacht und bereits in das .NET-Framework integriert ist.

### 3.1.1 Probleme

Bei beiden Vertretern können viele Faktoren die Zeitmessung beeinflussen. Die Meisten sind maschinenabhängig und einige beeinflussen die Zeitmessung nicht so stark, wenn ein grosser Programmteil auf die Grafikkarte ausgelagert wird. Wichtige Faktoren werden nachfolgend beschrieben.<sup>[5]</sup>

#### Multicore Prozessoren

Da die Arbeit auf mehrere Prozessorkerne aufgeteilt werden kann, hängt das Messresultat von der Aufteilung ab und kann sich je nach Durchlauf stark unterscheiden. Ausserdem kann es vorkommen, dass ein Prozess auf einem Prozessorkern gestartet, dann auf Eis gelegt und später auf einem anderen Prozessorkern wieder aufgenommen wird. Im Test unterschieden sich Messresultate des Invert-Filters um bis zu Faktor 2.

Um das Problem zu lösen bietet das .NET Framework die Funktion *Thread.BeginThreadAffinity*<sup>7</sup>, welche sicherstellen sollte, dass der umgebene Codeblock auf einem Prozessorkern läuft. Dies führte aber nicht zur gewünschten Stabilisierung, weil diese Funktion nicht mit Hyper-Kernen<sup>8</sup> zurecht kommt.<sup>[6]</sup> Um auch mit diesen Umgehen zu können, wird die native Funktion *SetThreadAffinityMask*<sup>9</sup> verwendet. Nach der Anpassung war die Laufzeit zwar um einiges langsamer (Faktor 2), dafür resultierten konstante Testwerte.

#### Frequenzänderung

Neue Prozessoren haben die Fähigkeit, falls nötig, zu übertakten. Da bei der Zeitberechnung die verstrichenen Ticks durch die ausgelesene Frequenz geteilt werden, resultiert bei einer Frequenzänderung ein falsches Resultat. Anstatt die verstrichene Zeit zu messen, könnten auch die verstrichenen Ticks miteinander verglichen werden. Dies ist auch oft sinnvoller, wenn auf mehreren Rechnern mit unterschiedlichen Ressourcen gemessen wird. Denn bessere Hardware bringt zwar eine schnellere Verarbeitungszeit, aber die benötigten Prozessor Ticks bleiben ungefähr gleich. Da in diesem Fall aber grosse Teile auf der Grafikkarte und nicht auf dem Prozessor gerechnet werden, sind die Prozessor Ticks nicht aussagekräftig. Der Prozessor schickt die Aufgabe nur an die Grafikkarte und wartet auf das Resultat. Ein schneller Prozessor wird mehr Ticks warten als ein langsamer. Deshalb werden nicht die benötigten Prozessor Ticks, sondern die benötigte Zeit gemessen.

<sup>6</sup><http://msdn.microsoft.com/en-us/library/ms644904%28v=vs.85%29.aspx>

<sup>7</sup><http://msdn.microsoft.com/en-us/library/system.threading.thread.beginthreadaffinity.aspx>

<sup>8</sup><http://www.intel.com/cd/corporate/techtrends/emea/deu/platform-technology/hyper-threading/311658.htm>

<sup>9</sup><http://msdn.microsoft.com/en-us/library/ms686247%28v=vs.85%29.aspx>

Das Problem mit der Frequenzänderung wurde gelöst, indem das Über- und Untertakten im BIOS ausgeschaltet wurde.

### Messung benötigt Zeit

Um die Zeitmessung durchzuführen wird auch Prozessorzeit benötigt, welche in die Zeitmessung mit einfließt.

Um diese Zeit abzuziehen, kann eine Zeitmessung ohne Inhalt erfolgen und die dafür benötigte Zeit von den Messresultaten abgezogen werden. Da die benötigte Zeit für die Zeitmessung im Vergleich zu den Messzeiten sehr klein und konstant ist, wurde dies hier vernachlässigt werden.

### Initialisierung

Bei der ersten Berechnung können starke Schwankungen auftreten, besonders beim Aufsetzen des Cuda Kontextes wird viel Zeit benötigt.[7]

Um dies zu verhindern kann der Aufruf mehrmals wiederholt werden. Optimalerweise wird der erste Aufruf ignoriert, da bei ihm die grössten Schwankungen auftreten und über den Rest der Durchschnitt gebildet. Da bei Cuda auch sonst Ausreisser vorhanden sein können, welche ein Vielfaches der durchschnittlichen Zeit benötigen, wurde hier der Median über alle Frames eines Videos gebildet.

## 3.2 Videoauswahl

Grundsätzlich kann jeder Filter auf alle Videos angewendet werden, jedoch wird nicht immer eine sinnvolle Ausgabe resultieren. Da die Ausgabe für die Zeitmessung allerdings nicht von Bedeutung ist, wird immer mit dem Video *Videos/Medicine Capsules.avi* gemessen, welches eine Filtergrösse von 600x400 Pixeln unterstützt.

## 3.3 Gebrauch

Für das Programm sind keine Parameter erforderlich. Die Konfiguration wird direkt im Code vorgenommen.

### 3.3.1 Konfiguration

Im Konstruktor der Klasse *Testsuite.Configuration* können Filter von *Promon200* direkt zur Filterliste hinzugefügt werden. Die hinzugefügten Filter werden dadurch auf jedes Frame aller Videos angewendet. Die Videos können in einer weiteren Liste definiert werden, wie im Listing 3.2 zu sehen ist. Zu den Videos wird auch ein frei wählbarer Name und der Filterbereich gespeichert. Der Filterbereich gehört zum Video, da die Filter mit den selben Filtergrössen gegeneinander verglichen werden sollen.

```
1 public Configuration()
2 {
3     // Filters
```

```

4     Filters = new LinkedList<PromonBaseFilter>();
5     Filters.AddLast(new PatternMatchingHaar());
6     Filters.AddLast(new InvertCuda());
7
8     // Videos
9     Videos = new LinkedList<Video>();
10    Videos.AddLast(new Video("Medicine Capsules",
11                             @"..\..\Videos\Medicine Capsules.avi",
12                             new System.Drawing.Rectangle(30, 30, 25, 25)));
13 }

```

Listing 3.2: Configuration

### 3.3.2 Ausgaben

Pro Video werden, die in der Tabelle 3.1 beschriebenen, Information auf der Konsole ausgegeben.

INFORMATION	BESCHREIBUNG
Video name	Definierter Videoname aus der Konfiguration
Video resolution	Grösse der Videoframes in Pixel (Auflösung)
Video length	Länge des Videos in Frames
Filter size	Grösse und Position des Filters in Pixel

Tabelle 3.1: Ausgaben der Testsuite pro Video

Pro Video werden für jeden Filter, die in der Tabelle 3.2 aufgelisteten, Informationen ausgegeben.

INFORMATION	BESCHREIBUNG
Filter name	Name des Filters wie er im Filter-Attribut <i>Name</i> definiert wurde
Average	Durchschnittlich benötigte Zeit des Filters über alle Frames des Videos
Median	Median der benötigten Zeit des Filters über alle Frames des Videos

Tabelle 3.2: Ausgaben der Testsuite pro Filter

### Auswertung

Zusätzlich zu den Konsolenausgaben wird am Ausführungsort eine csv-Datei mit dem Namen *statistics.csv* erstellt. Bei jeder Ausführung der Testsuite werden die neuen Resultate an das Ende angehängt. In der Tabelle 3.3 sind alle Daten zu sehen, die pro Video und Filter abgespeichert werden.

INFORMATION	BESCHREIBUNG
Video name	Definierter Videoname aus der Konfiguration
Date	Datum und Uhrzeit wann der Test ausgeführt wurde auf Sekunden genau
Filter name	Name des Filters wie er im Filter-Attribut <i>Name</i> definiert wurde
Average	Durchschnittlich benötigte Zeit des Filters über alle Frames des Videos
Median	Median der benötigten Zeit des Filters über alle Frames des Videos

Tabelle 3.3: Informationen die bei jeder Ausführung der Testsuite abgespeichert werden

Dies ermöglicht eine Auswertung des Verlaufs der Performanz eines Filters über einen bestimmten Zeitraum.

Ausserdem wird eine csv-Datei mit dem Namen *statistics.csv* und angehängtem Datum erstellt, welche es erlaubt Filter direkt gegeneinander zu vergleichen. Die Daten sind in einer Form, welche es erlaubt ein Diagramm zu erstellen, welches die konfigurierten Filter bei allen getesteten Filtergrössen gegenüberstellt.

## 4 Filter mittels Cuda parallelisieren

Im ersten Teil dieser Arbeit werden einige der bestehenden Filter (*Average Brightness*, *Velocity* und *Pattern Matching*) mittels Cuda umgesetzt. Das Ziel ist abzuklären, ob die Cuda-Implementierungen eine tiefere Verarbeitungszeit aufweisen, als die vorhandenen C# Implementationen. Die Zeitmessungen werden mit der Testsuite durchgeführt. Die Korrektheit der Filter wird in *Promon200* überprüft. Dazu werden die Filter übereinander gelegt und müssen bei jedem Frame dieselben Ausgabewerte zurückgeben. Für alle drei Filter wurde ein Video bestimmt, um die Korrektheit zu testen. Das Graustufen-Video *Videos/Medicine Capsules.avi* eignet sich für alle beschriebenen Filter. Es zeigt eine Anlage die Medikalkapseln auf einem Band transportiert. Beim *Velocity*-Filter kann die Geschwindigkeit einer Kapsel bestimmt werden, welche heller als die Umgebung ist. Für den *Average-Brightness*-Filter kann der durchschnittliche Grauwert einer beliebigen Framestelle bestimmt werden. Hier ist kein Farbvideo notwendig, da die vorhandene Implementation nur den ersten Kanal eines Pixels beachtet. Der *Pattern-Matching*-Filter kann eine herunterfallende Kapsel matchen.

### 4.1 NVIDIA Cuda Architektur

Die *Compute Unified Device Architecture* (Cuda) ist eine abstrahierende Schicht, um Berechnungen auf die Grafikkarte auszulagern. Grafikkarten verfügen über sehr viele Cores, verglichen mit den CPUs, sind dafür aber wesentlich tiefer getaktet. Man verspricht sich dadurch, dass parallele Berechnungen performanter auf der Grafikkarte berechnet werden können, als auf der CPU. Analog sollten serielle Berechnungen auf der Grafikkarte vermieden werden. Für diese Arbeit wurde die Version 3.2 verwendet.

**Device** In diesem Fall die Grafikkarte

**Host** In diesem Fall der Computer in dem sich die Grafikkarte befindet

**GRAM** RAM der Grafikkarte

### 4.2 Invert-Filter

Um die Grundlagen für die Umsetzung der bestehenden Filter in Cuda zu schaffen wurde zuerst ein Invert-Filter in Cuda implementiert. Dieser Filter invertiert alle Pixel in einem gegebenen Bereich (255-Pixelwert). Er wurde ausgewählt, da er einfach zu parallelisieren ist und man die Ausgabe direkt im Frame sieht. Es wurde Schritt für Schritt versucht, diesen Filter zu optimieren, damit die Verarbeitungszeit kleiner 5 ms ist. Zusätzlich wird der Filter gegen den bestehenden Invert-Filter verglichen, der in C# umgesetzt wurde.

Im *Invert Cuda Basis*-Filter wird bei jedem Filteraufruf zuerst GRAM alloziert und anschließend das komplette Frame auf das Device kopiert. Danach wird der Filterbereich mittels eines Threads pro Pixel invertiert. Das Frame mit invertiertem Filterbereich wird zurück ins Host RAM kopiert, damit die Änderungen sichtbar werden. Schlussendlich wird das GRAM wieder freigegeben.

Die erste Version des Invert-Filters, *Basis* genannt, ist bei allen getesteten Filtergrößen langsamer als die C#-Implementation (siehe Abbildung 4.2). Weiter wird bei einer Filtergröße von 600x400 Pixel die Rahmenbedingung von einer Laufzeit  $< 5$  ms verletzt. Nachfolgend wird dieser Filter analysiert und auf mögliches Optimierungspotenzial untersucht.

### 4.2.1 Allozieren von GRAM

Bei jedem Filteraufruf wird GRAM alloziert und nach dem invertieren des Filterbereichs wieder freigegeben. Solange die Filtergröße gleich gross bleibt, kann das allozierte GRAM für jedes Frame wiederverwendet und überschrieben werden. Bei einer Größenänderungen, muss der Speicherbereich freigegeben und neu alloziert werden, da mehr bzw. weniger Daten in das GRAM kopiert werden. Am Schluss des Streams muss der reservierte Speicherbereich wieder freigegeben werden. Durch diese Anpassung fällt das Allozieren und Freigeben von GRAM pro Frame weg. Um diese Optimierung umsetzen zu können, muss die Basisklasse *PromonBaseFilter* um zwei Methoden ergänzt werden (siehe Listing 4.1). Eine *Setup* Methode, welche vor dem ersten Aufruf der *applyFilter* Methode ausgeführt wird und eine *TearDown* Methode um Aufräumarbeiten durchzuführen (in diesem Fall das Deallozieren). Die Default-Implementation sind zwei leere überschreibbare Methoden. Dadurch müssen die bestehenden C#-Filter nicht geändert werden. Diese Optimierung erspart auf dem Testsystem ungefähr 0.4 ms pro Frame.

```

1  /// <summary>
2  /// Needs to be called once before using the filter.
3  /// </summary>
4  /// <param name="roi">filter area</param>
5  /// <param name="pixelSize">pixel size in Bytes</param>
6  /// <param name="stride">stride of the frame</param>
7  protected virtual void Setup(Rectangle roi, int pixelSize, int
    stride) {}
8
9  /// <summary>
10 /// Needs to be called after using the filter and will before
    resizing the filter area.
11 /// </summary>
12 /// <param name="final">final Tear down, object will be destroyed
    afterwards</param>
13 public virtual void TearDown(bool final) {}

```

Listing 4.1: Neue Methoden in der Klasse PromonBaseFilter.cs

Die Parameter in der *Setup* Methode sind nötig, damit innerhalb des Filters das benötigte GRAM berechnet werden kann (z.B. *Filterhoehe \* stride* wenn der gesamte Filterbereich kopiert werden soll). Weiter erben alle Cuda-Filter von *CudaBaseFilter* (siehe Abbildung 4.1), welche die *Setup*- und *TearDown* Methode *abstract* definiert, um das Memory Management für alle Cuda-Filter zu erzwingen. Die Basisklasse bietet zudem Methoden für das Allozieren und Freigeben von Speicher, welche in allen Filtern verwendet werden können. Zusätzlich dient *CudaBaseFilter* zur Unterscheidung von klassischen C#- und Cuda-Filtern. Dies wurde in der Version *Komplettes Frame* umgesetzt.

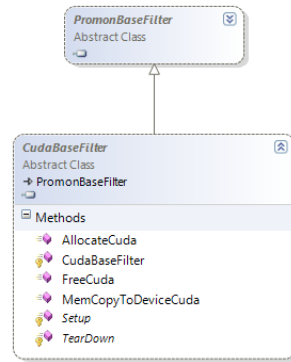


Abbildung 4.1: Klassendiagramm vom CudaBaseFilter, von welchem alle Cuda-Filter erben

#### 4.2.2 Memory Copy von Host nach Device und zurück

Im Filter *Komplettes Frame* wurde immer das ganze Frame auf das Device kopiert, obwohl nur der Filterbereich innerhalb des Frames benötigt wird. Um die zu kopierende Datenmenge zu verringern, wurden zwei verschiedene Ansätze verfolgt.

Beim ersten Ansatz wird nur der Filterbereich auf das Device übertragen. Dies erfordert, dass vor der Übertragung der Filterbereich auf dem Host in ein neues Array kopiert wird. Das Umkopieren ist notwendig da der Speicherbereich der Filterregion nicht zusammenhängend ist. Durch diese Änderung wird die zu übertragende Datenmenge minimal. Jedoch ist das Umkopieren eine teure Operation. Dieses Verfahren wurde im Filter *Umkopieren* umgesetzt. Bis zu einer Filtergrösse von 150x100 Pixel ist diese Version performanter als das Kopieren des ganzen Frames (*Komplettes Frame*). Bei sehr grossen Filtergrössen wird der Aufwand des Umkopierens zu gross und der Filter ist langsamer als alle anderen Implementationen.

Beim zweiten Ansatz werden nur die vom Filter betroffenen Zeilen kopiert. Somit fällt das Umkopieren in ein neues Array weg, da der Speicherbereich zusammenhängend ist. Dies kann relativ einfach durch ein Offset realisiert werden:

$$p' = p + filterY * frameStride + filterX * pixelSize$$

**p** Pointer auf die erste Speicherstelle des Bildes

**filterY** Filter Y-Position in Pixel

**filterX** Filter X-Position in Pixel

**frameStride** Stride des Frames in Bytes

**pixelSize** Anzahl Bytes pro Pixel

Auch hier werden nicht benötigte Daten kopiert, jedoch einiges weniger. Dieses Verfahren wurde im Filter *Zeilen kopieren* umgesetzt. Wie in der Abbildung 4.2 zu sehen ist, liefert es die beste Performance für sämtliche getesteten Filtergrössen mit Cuda. Ein Nachteil ist hier, dass durch das Zurückkopieren der Bildbereich ausserhalb des Filters überschrieben wird. Somit ist es nicht möglich, zwei Invert-Filter in den gleichen Zeilen zu benutzen. In der Praxis ist dies kein Problem, da die anderen Filter das Originalframe nicht verändern.

### 4.2.3 Parallelisierung

Laut Entwicklerdokumentation sollte der gesamte Rechenaufwand in möglichst kleine Teilstücke gebrochen werden[1]. Es macht somit Sinn, dass wenn möglich ein Thread nur ein einzelnes Pixel bearbeitet anstatt eine ganze Zeile eines Bildes. Dadurch erhöht sich die Skalierbarkeit der Filter.

### 4.2.4 Schlussfolgerung

Die vorher durchgeführten Tests und Optimierungen haben gezeigt, dass folgende Ziele verfolgt werden sollen:

- Das Allokieren von Speicher auf dem Device ist je nach Grafikkarte eine sehr teure Operation. Deshalb ist es besser den Speicher nur einmal zu allozieren und selber zu verwalten.
- Daten, die auf das Device kopiert werden müssen, sollten minimiert werden. Dies darf jedoch nicht um jeden Preis geschehen, wie das Beispiel mit der *Umkopieren*-Filterversion gezeigt hat.
- Um die Performanz zu maximieren, muss das Problem in möglichst kleine Teilprobleme gebrochen werden. Dadurch wird eine hohe Parallelität bei der Verarbeitung garantiert. Weiter skalieren die Filter auch bei anderen leistungsstärkeren Grafikkarten besser.

Bei dem einfachen Invert-Filter zeigt sich, das Cuda ungeeignet für kleine Aufgaben ist. Dies liegt am Overhead des Daten Kopierens von Host nach Device und zurück. Da dieser flacher ansteigt, als die Verarbeitungszeit der CPU, wird die Cuda-Implementierung ab grossen Filterbereichen performanter als das sequentiell verarbeitete C# Gegenstück.

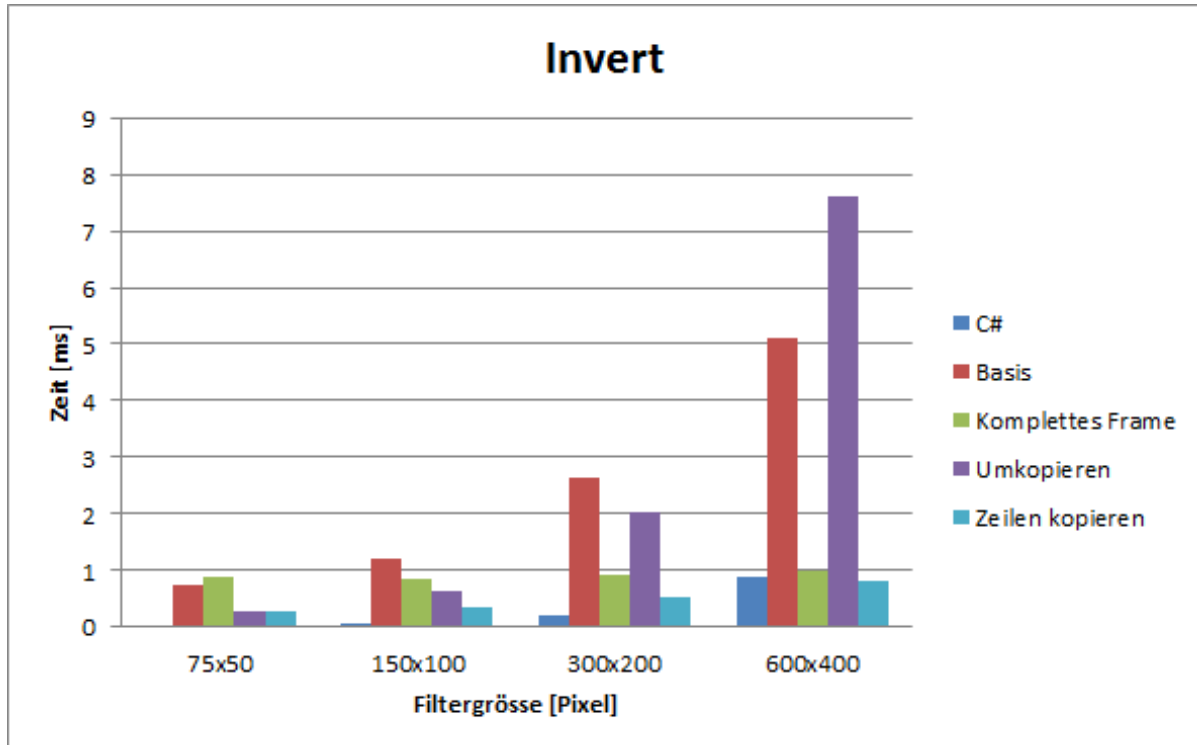


Abbildung 4.2: Geschwindigkeitsvergleich der verschiedenen Invert Filter

## 4.3 Average Brightness

Die C#-Version des Average Brightness Filter, berechnet den durchschnittlichen Grauwert des Filterbereichs. Dazu berechnet er die Summe aller Werte des 1. Farbkanals jedes Pixels und dividiert diese durch die Anzahl der Pixel. Die Cuda Implementationen beachten jedoch alle Farbkanäle. Dadurch wird auch bei Farbbildern ein sinnvoller Wert zurückgegeben. Allerdings ist der Rechenaufwand natürlich grösser. Um die Cuda Version trotzdem direkt gegen die C#-Version vergleichen zu können, wurde die Version *Pro Spalte* auch mit nur einem Farbkanal implementiert.

### 4.3.1 Pro Spalte

In einer ersten Cuda Version des Average Brightness Filters wurde der Durchschnitt jeder Spalte berechnet und in einem Float Array gespeichert. Der Durchschnitt einer Spalte kann jeweils von einem Cuda Thread berechnet werden. Dadurch wird eine mässige Parallelität erreicht. Die Datenmenge wurde nun auf eine Zeile reduziert. Das Berechnen des Durchschnitts einer Zeile kann nicht ohne weiteres parallelisiert werden (dazu mehr in Kapitel 4.3.2), daher wird die Zeile zurückgegeben und der Durchschnitt seriell auf der leistungsstarken CPU berechnet. Von diesem Filter wurden zusätzliche Messungen gemacht, in welchem nur der erste Farbkanal beachtet wird (*Pro Spalte x1*). Dadurch kann er direkt mit der C#-Variante verglichen werden. Die Messresultate vom Filter, bei welchen alle Farbkanäle berücksichtigt werden, sind mit *Pro Spalte x3* gekennzeichnet. Im Vergleich ist ersichtlich, dass diese Filterversion bei einer Filtergrösse von 600x400 ein wesentlicher Geschwindigkeitsgewinn gegenüber der C#-Implementation aufweist (siehe Abbildung 4.6). Auch ist ersichtlich, dass die Zeitunterschiede zwischen der Ein-Farbkanal und der Drei-Farbkanal Version minimal sind. Dies ist damit zu erklären, dass die meiste Zeit für das Memory kopieren benötigt wird und die drei Mal mehr Cuda-Threads, welche gestartet werden, nicht so stark ins Gewicht fallen.

### 4.3.2 Pro Spalte mit Faktorisierung

In dieser Version wird der Ansatz verfolgt, die Menge an Werten, welche zurückgegeben und seriell auf der CPU verrechnet werden, zu reduzieren. Denn bei einer Filtergrösse von 600x400 Pixeln sind dies 1800 Werte (Es werden alle 3 Farbkanäle beachtet). Daher wird versucht die erhaltenen Werte wieder in eine Matrix Form zu bringen und auf dieser den gleichen Algorithmus nochmals anzuwenden. Um zu wissen, ob dies möglich ist, müsste die Länge faktorisiert werden, was einen exponentiellen Aufwand bedeutet. Daher wird nur nach den Faktoren 2 und 3 gesucht. Sind 2 und 3 keine Faktoren, wird der Algorithmus abgebrochen und der Durchschnitt der zurückbleibenden Werte muss seriell auf der CPU verrechnet werden. Die beiden Faktoren 2 oder 3 sind in etwa 66% aller Zahlen vorhanden. Würde zusätzlich noch der Faktor 5 berücksichtigt, würden ca. 73% der Zahlen abgedeckt werden.

Da die Faktorensuche und ein zusätzlicher Kernelaufruf Zeit benötigen, muss zuerst überprüft werden, ob sich dieser Aufwand lohnt. Es wurde eine Mindestanzahl der verbleibenden Werte eingeführt, welche überschritten werden muss, damit überhaupt nach Faktoren gesucht wird. Ist die Anzahl Elemente zu klein, werden sie zur Berechnung an die CPU zurückgegeben. Zusätzlich kann die Mindesthöhe der neuen Matrix angegeben werden, welche durch die Faktorensuche gefunden werden muss. Ist diese nicht erreicht lohnt sich ein weiterer Kernelaufruf nicht. Trotz diesen Einstellungen ist es nicht gelungen, einen Geschwindigkeitsvorteil gegenüber der *Pro Spalte* Version zu erreichen (siehe Abbildung 4.6). Dies ist damit zu erklären, dass die C#-Variante bis zu einer Filtergrösse von 300x200 Pixeln immernoch einiges schneller ist als alle

Cuda Konkurrenten. Es ist also performanter 60'000 Werte auf der CPU aufzusummieren, als dies auf die Grafikkarte auszulagern. Die Annahme, dass das Wegfallen der Zeit des Memory Kopierens bei einem zweiten Durchgang des Algorithmus darauf einen grossen Einfluss hat, erwies sich als falsch.

### 4.3.3 Prefix Scan

Diese Filterversion stellt die weitere Parallelisierung in den Vordergrund. Dazu wurde das *Nvidia Prefix Scan Sum Sample*<sup>10</sup> und das Paper *Parallel Prefix Sum (Scan) with CUDA*[3] als Grundlage verwendet. Dieser Algorithmus berechnet für jede Stelle eines Arrays die Summe aller vorhergehenden Werte und speichert diese an der entsprechenden Position ab (siehe Abbildung 4.3). In diesem Fall wird nur die Endsumme benötigt, damit aus dieser der Durchschnitt berechnet werden kann.

<b>Input</b>	[3 1 7 0 3 2 6 3]
<b>Resultat</b>	[0 3 4 11 11 14 16 22]

Abbildung 4.3: Exklusiver Prefix Scan mit Addition

Es werden halb so viele Threads gestartet wie Farbwerte (Pixel x Farbkanäle) vorhanden sind. Jeder Thread zählt zwei Werte zusammen. Danach berechnet die Hälfte der Threads die Summe von zwei so eben berechneten Zwischensummen (siehe Abbildung 4.4), die andere Hälfte der Threads läuft auch weiter, berechnet aber in dieser Phase nichts mehr. Trotzdem müssen sie aufgrund der nächsten Phase immer mit den berechnenden Threads synchron sein. Diese Phase (nachfolgend Reduktionsphase genannt) läuft solange, bis durch die Halbierung der Anzahl der Threads kein berechnender Thread mehr läuft.

<sup>10</sup><http://developer.download.nvidia.com/compute/cuda/1.1/Projects/scanLargeArray.zip>

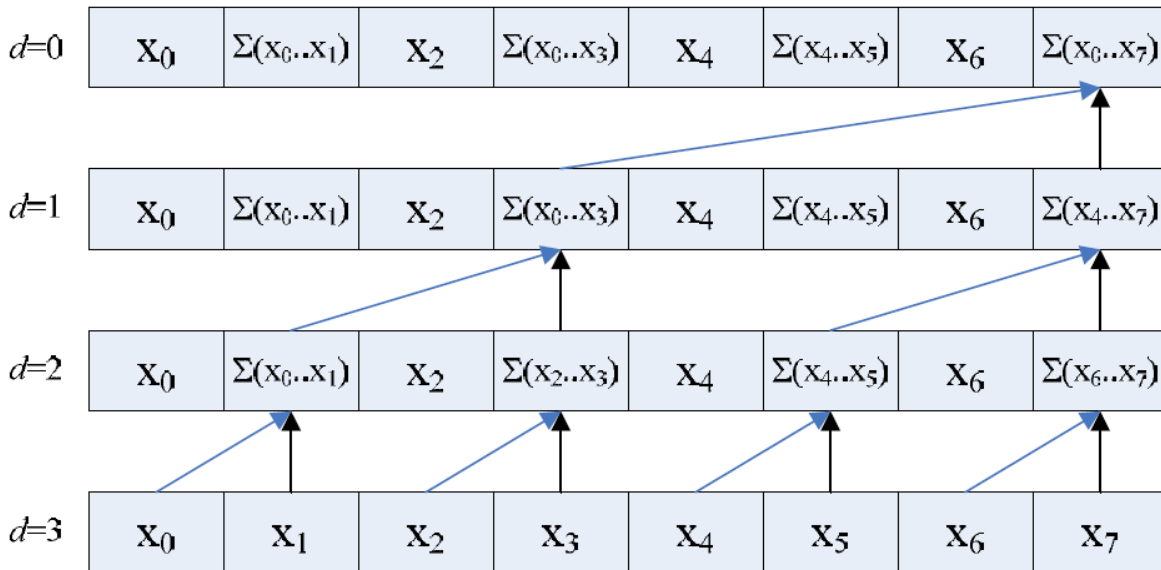


Abbildung 4.4: Reduktionsphase des Prefix Scans [3]

Wenn ein Array mit einer Größe eine zweier Potenz übergeben wird, steht im letzten Feld bereits nach der Reduktionsphase die gewünschte Gesamtsumme, andernfalls muss auch die zweite Phase des Algorithmus ausgeführt werden.

In der zweiten Phase des Algorithmus wird eine Null ans Ende des Arrays gesetzt. Danach jeder berechnende Thread den Wert mit dem eigenen Index und denjenigen mit dem Index/2 (siehe Abbildung 4.5) und schreibt das Resultat an die Stelle mit dem eigenen Index. Die Anzahl der berechnenden Threads wird nun wieder laufend verdoppelt, bis alle Zwischenwerte berechnet wurden.

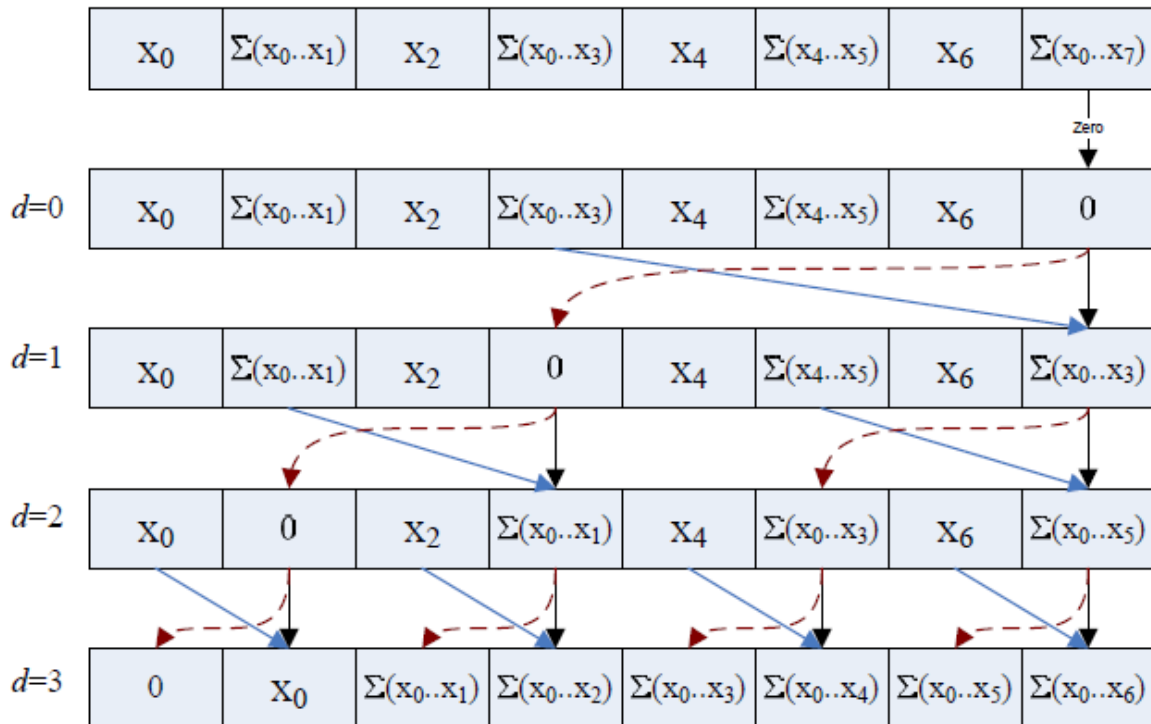


Abbildung 4.5: Zweiter Phase des Prefix Scans [3]

Zusätzlich muss darauf geachtet werden, dass die Synchronisation nur innerhalb eines GPU Cores geschehen kann. Dieses Problem wurde im NVIDIA Beispiel mit Hilfe von Rekursion gelöst. Der Prefix Sum Algorithmus wird auf kleine Bereiche angewendet. Die Endsummen dieser Bereiche werden zwischengespeichert und schlussendlich auf die entsprechenden Werte addiert. Das Ganze ist rekursiv aufgebaut, so dass jegliche Grössen verarbeitet werden können.

In dieser Filterversion wird davon ausgegangen dass die Grösse des Arrays keine zweier Potenz ist. Daher muss der ganze Algorithmus ausgeführt werden. Da der Input in diesem Fall kein zusammenhängendes Float-Array, sondern ein nicht zusammenhängendes Byte-Array ist, musste die *loadSharedChunkFromMem* Methode angepasst werden. Sie ist dafür verantwortlich die Daten ins *SharedMemory*<sup>11</sup> der Grafikkarte zu kopieren. Bei dieser Gelegenheit werden die nicht zusammenhängenden Byte-Werte auch gleich in ein zusammenhängendes Float-Array gemappt. Der Prefix Scan Algorithmus wird auf den ganzen Filterbereich angewendet. Wie die Abbildung 4.6 im Vergleich zeigt, ist diese Variante langsamer als die erste Variante aus Kapitel 4.3.2. Dies ist damit zu erklären, dass einer der  $filter\_bytes/2$  gestarteten Threads  $\log_2(filter\_bytes/2)$  Additionen berechnet und vor jeder Addition durch eine Thread Synchronisation gezwungen wird, auf die anderen zu warten. In der ersten Filterversion hingegen ist jede Spalte unabhängig von den Anderen. In jedem Thread werden  $filter\_height$  Additionen und eine Division durchgeführt.

**filter\_bytes** Grösse des Filters in Bytes

**filter\_height** Höhe des Filters in Pixel

<sup>11</sup>[http://developer.download.nvidia.com/compute/cuda/2.3/toolkit/docs/NVIDIA\\_CUDA\\_Programming\\_Guide\\_2.3.pdf](http://developer.download.nvidia.com/compute/cuda/2.3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf), Kapitel 3.2.2

#### 4.3.4 Prefix Scan mit zweier Potenz

In dieser Filterversion wird die übergebene Matrix um so viele Nullen erweitert, dass die Länge eine zweier Potenz ist. Dadurch ist nur die Reduktionsphase des Prefix Scan Sum Algorithmus notwendig, allerdings werden im schlimmsten Fall  $filter\_bytes/2 - 1$  überflüssige Elemente hinzugefügt, wodurch die Berechnung verlangsamt wird. Im Vergleich ist diese Version daher bei gewissen Filtergrößen schneller als die *Prefix Scan* version, bei anderen wiederum langsamer. So ist in Abbildung 4.6 ersichtlich, dass die *Prefix Scan* Variante bei einer Filtergröße von 150x100 Pixeln schneller ist, als die *Prefix Scan mit zweier Potenz* Version. Dies ist damit zu erklären, dass  $\log_2(100 * 100 * 3)$  ungefähr 14.87 ergibt, was heisst, dass nur  $2^{15} - 2^{14.87} \approx 2'824 \approx 9\%$  Nullen aufgefüllt werden müssen. Bei der Filtergröße von 600x400 Pixel, wo der Logarithmus dualis den Wert 19.46 ergibt, ist die *Prefix Scan* Variante schneller, da bei der *Prefix Scan mit zweier Potenz* Variante  $2^{20} - 2^{19.46} \approx 327'396 \approx 45\%$  Nullen aufgefüllt werden müssen.

**filter\_bytes** Grösse des Filters in Bytes

#### 4.3.5 Prefix Scan pro Zeile

Als nächstes wird versucht die Anzahl Berechnungen zu reduzieren. Deshalb wird der Prefix Sum nicht auf die ganze Matrix, sondern auf die einzelnen Zeilen ausgeführt. Dadurch werden  $filter\_height$  Kernelaufufe mit je  $\log_2(filter\_stride/2)$  Threads notwendig. Die Synchronisation muss nur innerhalb der Zeilen gewährleistet sein. Nachdem alle Zeilen berechnet wurden, werden die Endsummen der Zeilen in die erste Zeile transponiert und auf dieser erneut der Prefix Sum Algorithmus angewendet. Als Alternative könnten auch alle Zeilensummen zurückgegeben und diese seriell auf der CPU zusammengezählt werden.

Allerdings hat der Vergleich gezeigt, dass die *Prefix Scan pro Zeile* Variante bei weitem nicht konkurrenzfähig ist. Dies ist auf die übermässig vielen Cuda Aufrufe und dem damit verbundenen Overhead zurückzuführen.

**filter\_height** Höhe des Filters in Pixel

**filter\_stride** Breite des Filters in Bytes

#### 4.3.6 Vergleich

Im Vergleich ist zu berücksichtigen, dass die C#-Implementierung, wie auch die *Pro Spalte x1* Version nur den ersten Farbkanal berücksichtigen. Die restlichen Cuda Versionen berücksichtigen alle Farbkanäle und haben dadurch drei Mal so viele Werte zu verarbeiten. In Abbildung 4.6 ist ersichtlich, dass die C#-Implementierung bis zu einer Filtergröße von 300x200 Pixel wesentlich schneller ist als die *Pro Spalte* Version. Erst bei der Filtergröße von 600x400 Pixel sind einige Cuda Implementierungen schneller als die C#-Version.

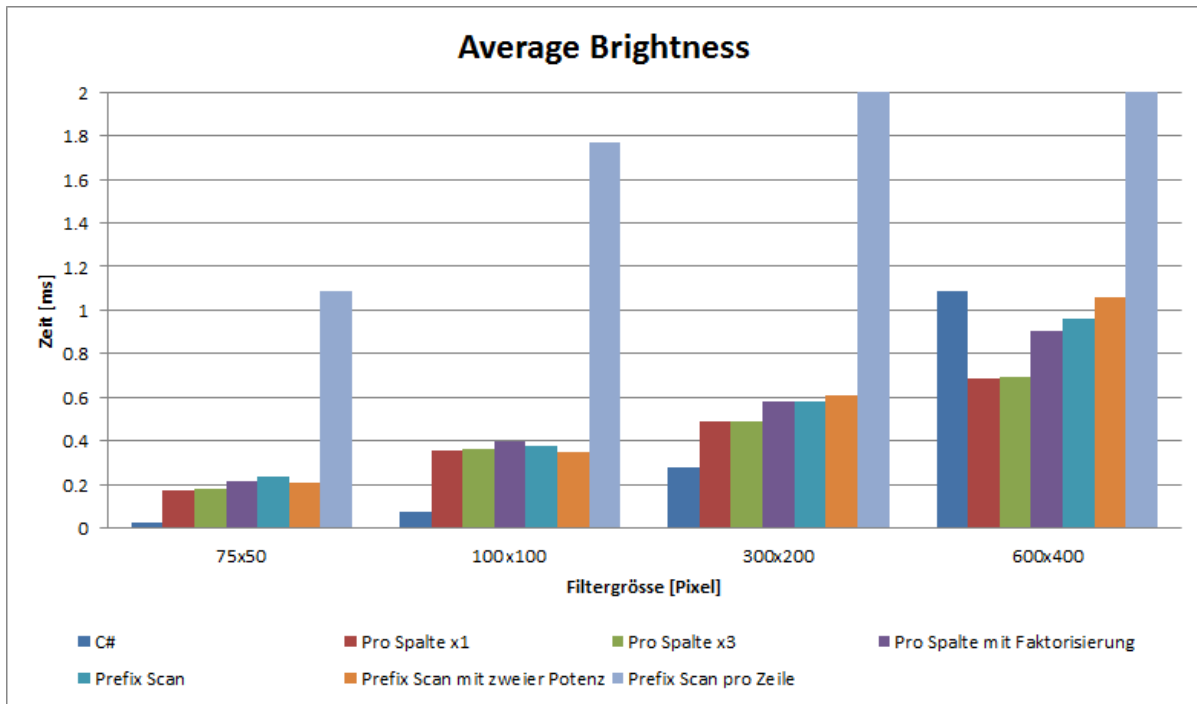


Abbildung 4.6: Geschwindigkeitsvergleich der verschiedenen Average Brightness Filter

## 4.4 Velocity-Filter

Der Velocity-Filter misst die Verschiebung eines Objektes in Pixeln pro Frame. Dazu wird das Frame zuerst anhand eines zuvor festgelegten Grenzwertes binarisiert. Danach wird das erste Auftreten eines weissen Pixels von links nach Rechts gesucht und zwischengespeichert. Beim nächsten Frame wird die Verschiebung des ersten weissen Pixels ausgegeben. Dies setzt voraus, dass das gemessene Objekt heller als seine Umgebung ist oder der Filter eine Option anbietet, um das erste schwarze Pixel zu suchen. Alternativ kann auch ein Invert-Filter verwendet werden, um dies zu erreichen.

Probleme bei diesem Filter sind einerseits negative Geschwindigkeiten, wenn das Objekt aus dem Filterbereich verschwindet und das nächste Objekt erscheint. Andererseits wird nur das Erste von links gemessen, wenn sich mehrere Objekte sich gleichzeitig im Filterbereich befinden. Auf diese Probleme wird nicht weiter eingegangen, da das Ziel die Geschwindigkeitssteigerung des vorhandenen Filters ist.

### 4.4.1 Komplette Binarisierung

In der vorhandenen C#-Variante wird Pixel für Pixel binarisiert. Falls es unter dem Grenzwert ist, wird es weiss und geprüft, ob es weiter links als ein bisheriges weisses Pixel liegt. Dies funktioniert parallel nicht, weil sonst das Risiko besteht, dass sich die einzelnen Threads gegenseitig überschreiben.

In der Cuda-Variante werden zuerst alle Zeilen, die den Filterbereich enthalten, auf das Device kopiert. Anschliessend wird der Filterbereich binarisiert und zurückkopiert. Auf dem Host wird nun seriell nach dem Auftreten des ersten weissen Pixels gesucht. Dazu wird Spalte für Spalte nach einem weissen Pixel gesucht und abgebrochen, sobald eines gefunden wurde. Es kann abgebrochen werden, da nur das Pixel am weitesten Links interessant ist für die Berechnung der Verschiebung.

Diese Variante ist sehr inperformant wie man in der Abbildung 4.7 sehen kann. Sie ist auf keiner Grösse konkurrenzfähig mit der C#-Version. Zwar kann die Binarisierung parallel durchgeführt werden, jedoch muss der gesamte Filterbereich (und der Speicherbereich dazwischen) vom Device wieder zurück auf den Host kopiert werden, um nachher noch eine serielle Suche nach dem ersten weissen Pixel durchzuführen. Dies kann bei der seriellen C# Variante gleichzeitig durchgeführt werden (Binarisierung und Suche nach dem weissen Pixel), da keine Gefahr durch gegenseitiges Überschreiben besteht.

### 4.4.2 Binarisierung erste Zeile

Bei dieser Version wird auf dem Device das Pixel nicht mehr direkt binarisiert, sondern das Pixel der ersten Zeile in der gleichen Spalte. Auch hier besteht das Problem, dass sich Threads gegenseitig überschreiben könnten, jedoch spielt es keine Rolle, wenn ein weisses Pixel nochmals auf weiss gesetzt wird und schwarze Pixel nicht gesetzt werden.

Dadurch muss nur noch die erste Zeile vom Device auf den Host zurückkopiert und auf weisse Pixel geprüft werden. In der Abbildung 4.7 ist klar zu sehen, dass diese Variante (*Binarisierung erste Zeile*) ab einer Filtergrösse von 150x100 Pixeln performanter ist als die erste Version (*Komplette Binarisierung*).

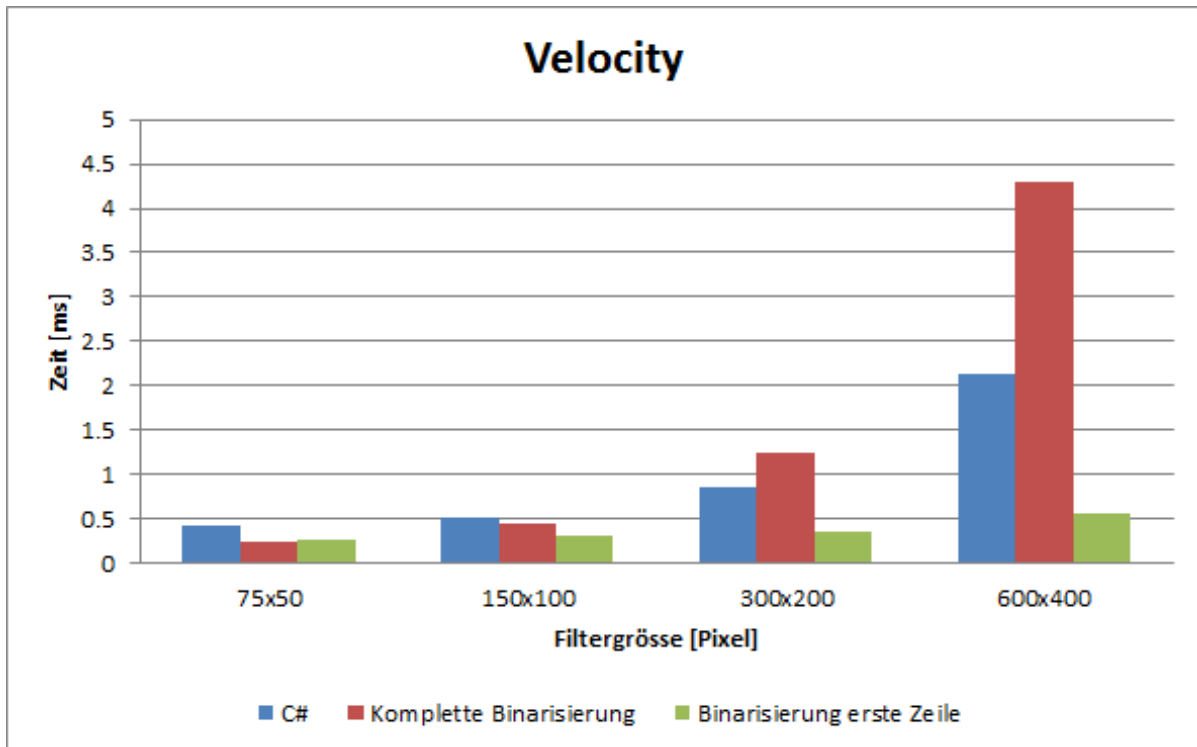


Abbildung 4.7: Geschwindigkeitsvergleich der verschiedenen Velocity Filter

#### 4.4.3 Vergleich

Die Version *Komplette Binarisierung erste Zeile* wird bei grossen Filtern (600x400 Pixel) sehr performant verglichen mit der C#-Variante. Dies ist vor allem auf den verschwindend klein werdenden Overhead der Cuda-Architektur zurückzuführen, bei grossem Rechenaufwand.

## 4.5 Pattern Matching

Die bestehende C#-Implementation des Pattern Matchings in *Promon200* basiert auf dem Paper *Fast pattern matching using orthogonal Haar transform*[4]. Es wird nur der erste Farbkanal verwendet, somit können nur Graustufen-Bilder erkannt werden. Dabei wird zuerst das Integral Bild<sup>12</sup> des Templates berechnet. Danach wird der Haar Vektor bestehend aus 16 Komponenten bestimmt. Dieser wird aus 25, über das Integral Bild, gleichmässig verteilten Punkten berechnet.

Anschliessend wird im Filterbereich (ROI) des Frames das Integral Bild berechnet. Nun wird für jede mögliche Position des Templates innerhalb der ROI der Haar Vektor bestimmt und gegen denjenigen des Templates verglichen. Für den Vergleich wird die Summe der absoluten Differenzen beider Vektoren verwendet. Liegt diese Summe unter einem selbst berechneten Grenzwert wurde ein möglicher Match gefunden. Dieser wird gegen den derzeitig besten Match verglichen und falls kleiner, überschrieben.

Die rechenintensiven Punkte dieser Version des Pattern Matchings ist die Berechnung des Haar Vektors für jede mögliche Template Position wie in der Abbildung 4.8 zu sehen ist. Die parallele Variante führt das Matching einiges schneller durch, birgt jedoch das Risiko, dass die einzelnen Threads sich gegenseitig überschreiben mit Matches, die die Grenzwertbedingung erfüllen.

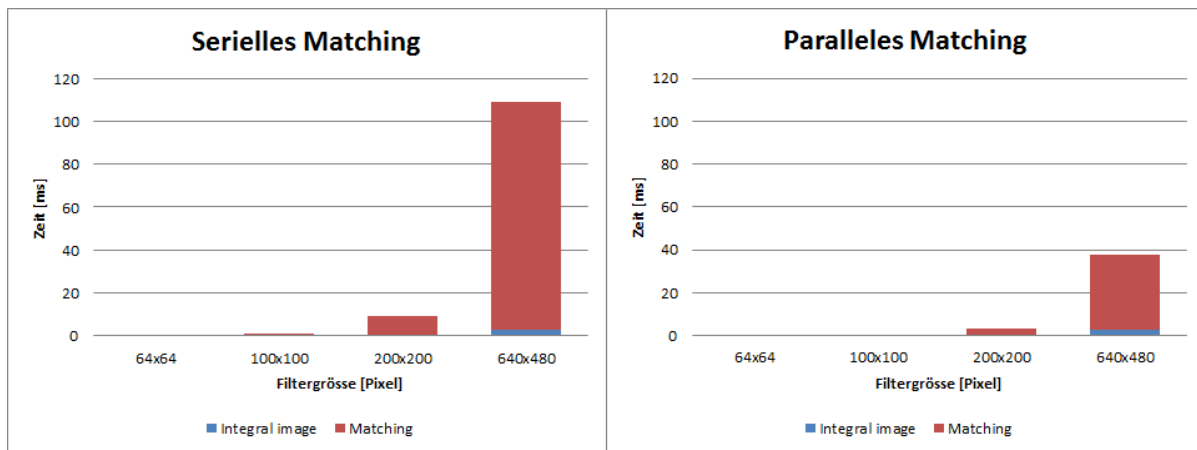


Abbildung 4.8: Operationen der seriellen und parallelen C#-Version des Pattern Matchings

### 4.5.1 Prefix Sum

Das Integral Bild und der Haar Vektor des Templates müssen nur einmal berechnet werden und können somit im Speicher auf dem Device gehalten werden. Dazu wird bei der Instanziierung des Filters direkt Speicher auf dem Device alloziert und der Haarvektor berechnet.

Das Integral Bild des Frames ändert bei jedem Frame und muss deshalb immer neu berechnet werden. Die Berechnung wird mittels des im Kapitel 4.3.3 vorgestellten Prefix Scan Algorithmus berechnet. Der Algorithmus wird für jede Zeile aufgerufen, danach das Bild transponiert und nochmals auf die transponierten Zeilen der Matrix angewendet. Nun liegt ein transponiertes Integral Bild im Speicher. Da die gleiche Prozedur für die Berechnung des Integral Bildes beim Template angewendet wurde, kann der Vergleich direkt geschehen, ohne dass das Bild nochmals transponiert wird. Jedoch muss beachtet werden, dass beim Resultat die x- und y-Koordinate vertauscht sind.

<sup>12</sup><http://de.wikipedia.org/wiki/Integralbild>

Anschliessend wird für alle möglichen Template Positionen in der ROI  $(fWidth - tWidth + 1)(fHeight - tHeight + 1)$  Mal der Haar Vektor bestimmt. Dies wird parallel auf dem Device durchgeführt, jedoch kann die aktuelle Differenz der Vektoren nicht direkt mit dem derzeitigen Minimum verglichen werden und muss somit zwischengespeichert werden. Der sofortige Vergleich ist nicht möglich, da die einzelnen Threads sich gegenseitig überschreiben könnten.

**fWidth** Filterbreite in Pixeln

**fHeight** Filterhöhe in Pixeln

**tWidth** Templatebreite in Pixeln

**tHeight** Templatehöhe in Pixeln

Als letztes wird das Minimum gesucht, welches den besten Match repräsentiert. Dieser Algorithmus wird seriell auf dem Device durchgeführt, damit nur 3 Werte zurückkopiert werden müssen (Minimum, x-Position, y-Position). Es muss nun beachtet werden, dass der x -und y-Wert vertauscht sind, da die Berechnungen auf transponierten Integralbildern geschah.

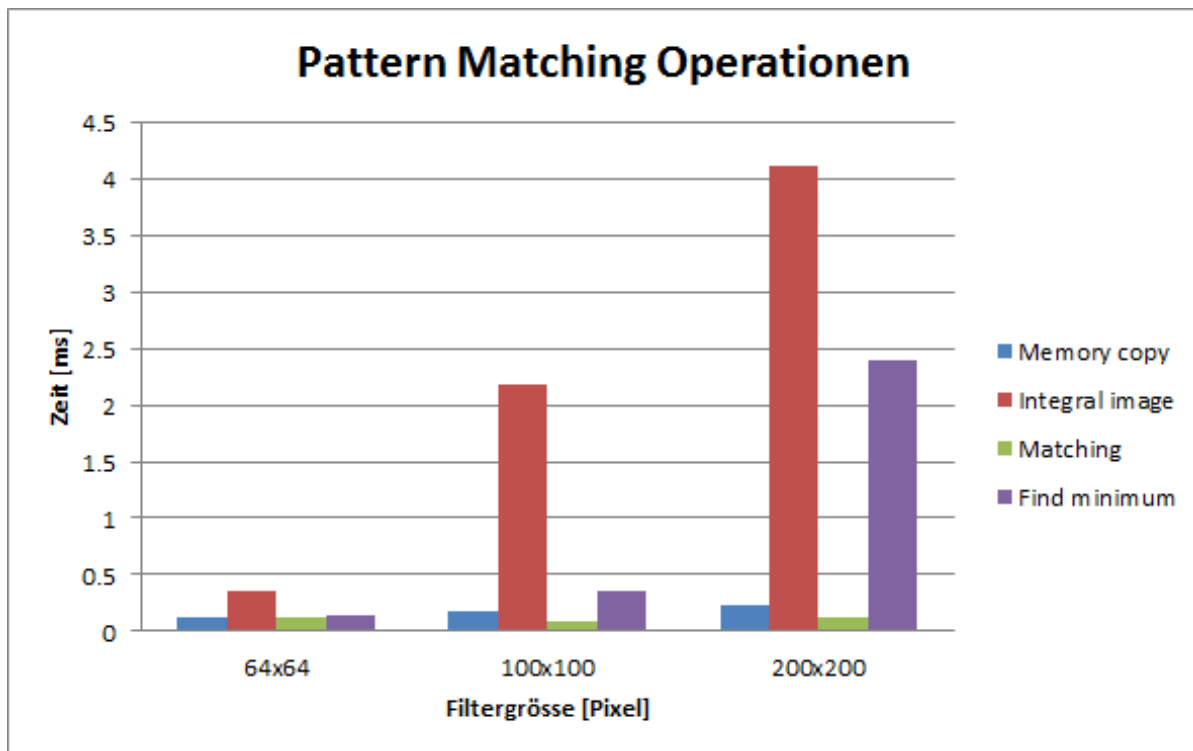


Abbildung 4.9: Die verschiedenen Operationen des Cuda Pattern Matchings

#### 4.5.2 Thread pro Zeile

Wie in der Abbildung 4.9 zu sehen ist, beansprucht die Integralbildberechnung der ROI am Meisten Zeit. Die Berechnung mittels Prefix Scans wird in der Version *Thread pro Zeile* ausgetauscht durch eine simple Variante, welche pro Thread eine Zeile des Bildes aufsummiert. In einem weiteren Schritt werden die Spalten aufsummiert, wodurch man das Integral Bild erhält. Das Transponieren des Frames fällt hier weg, da einmal auf Zeilen und einmal auf Spalten gearbeitet wird. Diese Art der Berechnung des Integral Bildes ist performanter, wie in der Abbildung 4.10 zu sehen ist. Dadurch ist diese Filterversion bei allen Grössen schneller als die

vorhergehende Version. Der Prefix Scan Algorithmus erlaubt eine höhere Parallelität, jedoch zerstört der Overhead der vielen Kernelcalls und die Synchronisierung der einzelnen Threads (siehe Kapitel 4.3.3) diesen Vorteil.

### 4.5.3 Asynchrone Matchbestimmung

Das Minimum wird auf der GPU seriell gesucht, was bei einer grossen Menge von Werten zeitintensiv ist (ein Drittel der gesamten Ausführungszeit). Der Grenzwert eines guten Matches ist vor Ausführung bekannt, deshalb müssen nur Resultate beachtet werden, die tiefer als dieser Grenzwert sind. In dieser Version wird ein Resultat an die erste Speicherstelle des allozierten GRAMs geschrieben, falls der Match genügend gut ist.

Hier besteht die Gefahr, dass zwei oder mehr Threads gleichzeitig einen Match finden und sich überschreiben. Dieses Problem besteht bei der Version *C# parallel* auch, ist aber weniger wahrscheinlich aufgrund der geringeren Anzahl an Cores auf der CPU. Deshalb muss der Benutzer sich entscheiden, ob er eine performantere Variante möchte (*Asynchrone Matchbestimmung* oder *C# parallel*) oder eine langsamere, dafür Exakte (*Thread pro Zeile* oder *C# seriell*).

Eine weitere Möglichkeit ist, das Minimum parallel zu suchen. Dies ist mittels einer Abwandlung des Prefix Scan Algorithmus umsetzbar. Aufgrund der schlechten Performanz, die die letzten Tests mit Prefix Sum Algorithmus ergaben, wurde dieser Ansatz nicht weiter verfolgt. Auch müsste für jede Differenz die x- und y-Position zwischengespeichert werden, da sich die Werte verschieben würden.

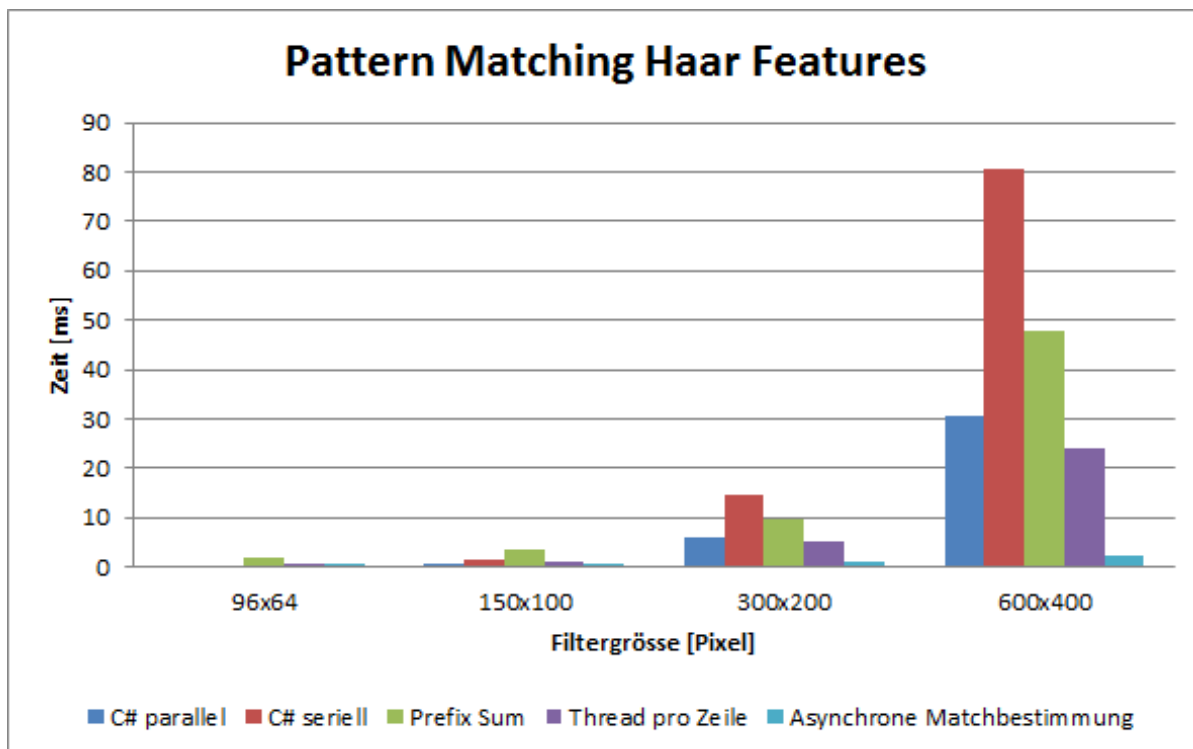


Abbildung 4.10: Geschwindigkeitsvergleich der verschiedenen Pattern Matching Filter

#### 4.5.4 Vergleich

Die Geschwindigkeitstests wurden mit einer Templategrösse von 64x64 Pixel durchgeführt. Wie bereits bei anderen Cuda Filtern zu sehen ist, werden die Cuda Varianten bei einer grossen ROI verglichen mit den C# Versionen performanter. Die Version *Asynchrone Matchbestimmung* benötigt auch bei einer Filtergrösse von 640x480 Pixeln weniger als 5 ms, während die C#-Version ein Vielfaches davon benötigt.

## 4.6 Integration von Cuda in Promon200

Da die entwickelten Filter alle von einer cuda-fähigen Grafikkarte abhängig sind, müssen grundsätzliche Überlegungen gemacht werden, was passiert, wenn keine cuda-fähige Grafikkarte vorhanden ist. Auch sonst haben Cuda-Filter Besonderheiten, welche in *Promon200* berücksichtigt werden müssen.

Dazu wurde eine Methode in *Promon200* integriert, welche prüft, ob Cuda unterstützt wird. Zuerst wird versucht die cuda-fähigen Grafikkarten auszugeben. Falls die benötigte dll nicht vorhanden ist, wird eine Exception abgefangen und die Methode gibt *false* zurück. Weiter wird überprüft, ob die zurückgegebenen Grafikkarten nicht emuliert sind.

### 4.6.1 Generelle Änderungen

Falls die Grafikkarte Cuda nicht unterstützt, sollen Cuda-Filter nicht angezeigt werden. Cuda-Filter erben von der Klasse *CudaBaseFilter*. Dadurch kann beim Start überprüft werden, ob es sich um einen Filter dieses Typs handelt und der Filter nur hinzugefügt werden, wenn die Grafikkarte cuda-fähig ist.

Die C#-Filter werden pro Frame parallel ausgeführt, um die vorhandenen Cores möglichst gut auszulasten. Dies ist bei Cuda-Filtern komplizierter. In Cuda 3.2 bekommt jeder Host-Thread automatisch einen eigenen Cuda-Kontext. In diesem Kontext kann Device-Memory alloziert, kopiert und damit gearbeitet werden. Allerdings ist es nicht möglich auf das Device-Memory eines anderen Kontextes zuzugreifen. Aus Optimierungsgründen, wie in Kapitel 4.2 beschrieben, wird das Device-Memory nur einmal alloziert und dann bei gleichbleibender Filtergrösse immer wiederverwendet. Dies spart auf dem Testsystem ca. 0.4 ms pro Filteraufruf. Werden die Cuda-Filter nun aber parallel ausgeführt, kriegt jeder einen neuen Thread und damit einen eigenen Cuda-Kontext. Dadurch kann nicht mehr auf das bereits allozierte Device-Memory zugegriffen werden.<sup>[2]</sup>

Dieses Problem wurde gelöst, indem Cuda-Filter in einer separaten Liste verwaltet werden, welche seriell ausgeführt wird. Dies erlaubt, dass die C#-Implementationen weiterhin parallel aufgerufen werden können. Dazu wurde die Klasse *GraphPane* angepasst. Ausserdem wird ein letztes *TearDown* der einzelnen Filter nötig. Dieses kann nicht im Destruktor aufgerufen werden, da der Cuda-Kontext zu diesem Zeitpunkt nicht mehr existiert. Daher wird es in der *MainWindow\_Closing* Methode der *MainWindow* Klasse ausgeführt.

### Ausblick

Das Problem lässt sich auch eleganter lösen. Cuda stellt in der Driver API Befehle zur Verfügung, um einen anderen Kontext zu übernehmen.<sup>13</sup> Diese Befehle sind in der Runtime API weg abstrahiert worden. Es ist seit Cuda 3.1 aber möglich, Driver API Aufrufe mit Runtime API Aufrufen zu mischen.<sup>[8]</sup> Mit den Befehlen *cuCtxPushCurrent* und *cuCtxPopCurrent* können die Cuda-Kontexte gewechselt werden.

In dem nun erschienen Cuda 4 wurde das Verhalten mit mehreren Host-Threads verbessert. Es ist nun möglich auf das Device-Memory eines anderen Host-Threads zuzugreifen.<sup>[9]</sup> Bei einem Wechsel auf Cuda 4 dürften daher einige Probleme wegfallen und die Cuda-Filter könnten parallel ausgeführt werden. Die Verwaltung in einer eigenen Liste würde wegfallen. Dadurch könnte sich auch die Performance bei mehreren verwendeten Cuda-Filtern weiter verbessern. Ausserdem unterstützen neuere Grafikkarte die parallele Verarbeitung von mehreren Kernelaufrufen,

<sup>13</sup>[http://developer.download.nvidia.com/compute/cuda/3.1/toolkit/docs/online/group\\_\\_CUCTX.html](http://developer.download.nvidia.com/compute/cuda/3.1/toolkit/docs/online/group__CUCTX.html)

bei Älteren werden diese in eine *Queue* gefüllt und seriell abgearbeitet. Ob die parallele Verarbeitung unterstützt wird, kann mit *cudaGetDeviceProperties* abgefragt werden. Die Anzahl Kernel, welche parallel abgearbeitet werden können steht im Property *concurrentKernels*.<sup>14</sup>

## 4.6.2 Composite Filter

Um die Übersichtlichkeit zu bewahren sollen Filter, von welchen sowohl eine C#- als auch eine Cuda-Version vorhanden ist, vereint werden. Dazu wurde die abstrakte Basisklasse *CompositeFilter* erstellt, welche sich als Filter in *Promon200* anzeigen lässt und dem Benutzer die Möglichkeit gibt, die verwendete Implementation per Checkbox zu wechseln. Dazu wurde eine Checkbox als Eingabeparameter in *Promon200* integriert.

Damit die Basisfunktionalitäten nicht mehrmals implementiert werden müssen, wurden sie bereits in der Basisklasse eingefügt. Die konkreten Composite-Filter müssen lediglich die Implementierung der Methoden *GetNewCSharpFilter* und *GetNewCudaFilter* bereitstellen. *CompositeFilter* erbt von *PromonBaseFilter* damit es als Kompositum<sup>15</sup> für die vereinten Filter fungieren kann (siehe Abbildung 4.11).

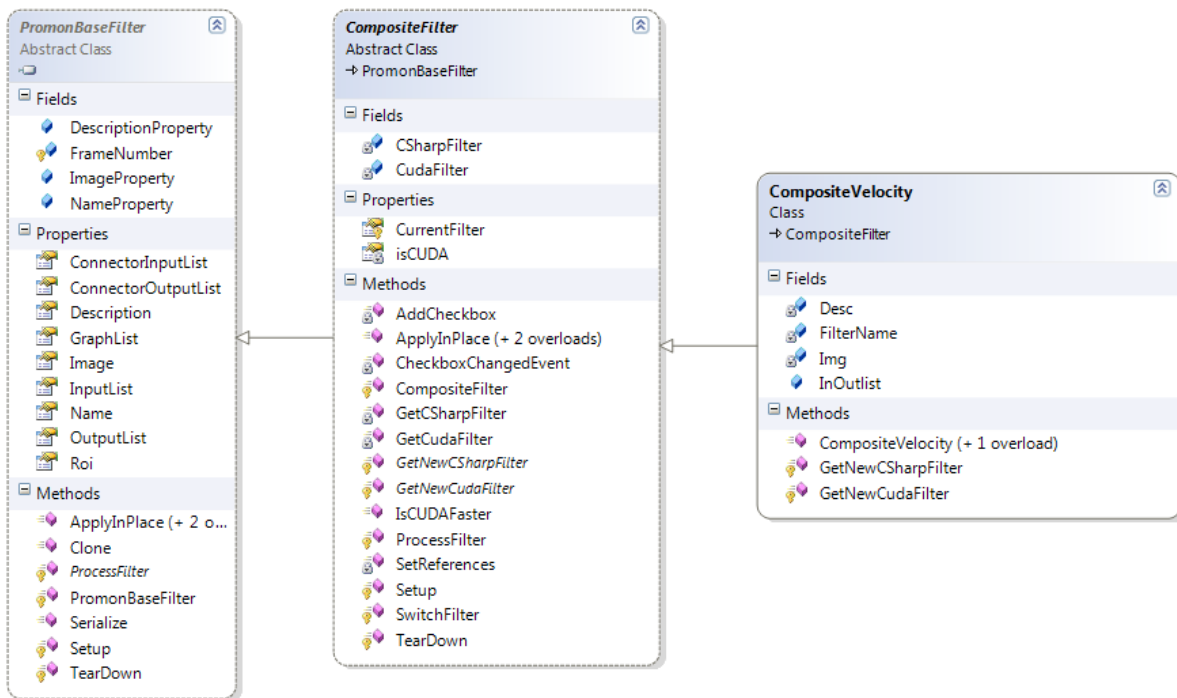


Abbildung 4.11: UML-Diagramm vom *CompositeVelocity* und seinen Basisklassen

Aufgrund dieser Erweiterung werden folgende Begriffe eingeführt:

**Kompositum / Composite-Klasse** Klasse, welche die verschiedenen Versionen der Filter vereint.

**Variante / Version** Konkrete Implementierung eines im Kompositum integrierten Filters.

<sup>14</sup><http://developer.download.nvidia.com/compute/cuda/>

[3.0/toolkit/docs/online/group\\_\\_CUDART\\_\\_DEVICE\\_g5aa4f47938af8276f08074d09b7d520c.html](http://developer.download.nvidia.com/compute/cuda/3.0/toolkit/docs/online/group__CUDART__DEVICE_g5aa4f47938af8276f08074d09b7d520c.html)

<sup>15</sup>[http://de.wikipedia.org/wiki/Kompositum\\_\(Entwurfsmuster\)](http://de.wikipedia.org/wiki/Kompositum_(Entwurfsmuster))

Die *CompositeFilter*-Klasse fügt automatisch eine Checkbox zu den Eingängen hinzu (siehe Abbildung 4.12), welche in der Eingangsliste (Liste von allen Filtereingängen) am Schluss angefügt wird. Da dieser Eingang am Schluss angehängt wird, werden die einzelnen Filter nicht beeinträchtigt, da innerhalb des Filters nur mit Indizes gearbeitet wird. Bei der Instanziierung der konkreten Implementation wandelt *Promon200* diese in, für den Benutzer, sichtbare Eingänge des Controls um. Bei der Instanziierung der zugehörigen Filter (z.B. die Cuda-Implementation) wird dessen Ein- und Ausgangsliste durch diejenige der Komposite-Klasse ersetzt. Dadurch wird gewährleistet, dass alle Ein- und Ausgänge des Kompositums korrekt weitergeleitet werden. Zusätzlich gehen die Daten bei einem Wechsel der Variante (*SwitchFilter*-Methode) durch den Benutzer nicht verloren, da immer auf der gleichen Referenz gearbeitet wird. Die Checkbox wird nur hinzugefügt, falls die Grafikkarte cuda-fähig ist.

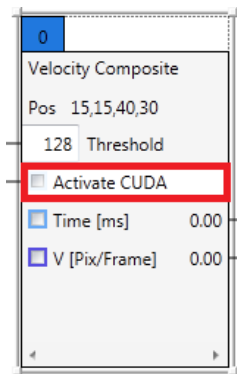


Abbildung 4.12: Composite Velocity Filter mit Checkbox zum Wechseln der Version

Laut Auftraggeber müssen die einzelnen Filter immer noch separat instanzierbar sein, da noch nicht klar ist, wie diese in der Zukunft verwendet werden. Dadurch gibt es Redundanzen. Der Name, die Beschreibung, die Ein- und Ausgänge sind im Kompositum und in den dekorierten Klassen vorhanden. Um diese Redundanz zu vermeiden, müssten die integrierten Klassen diese Informationen von der Composite-Klasse übernehmen (z.B. über eine statische Variable). Dadurch würde auch das Neusetzen der Referenz der Eingangsliste wegfallen.

## Speichern und Laden

Wenn Filterketten gespeichert werden, werden diese in eine XML Datei geschrieben. Dieses XML enthält alle Eingabewerte und kann im Konstruktor von *PromonBaseFilter* übergeben werden, um einen Filter wiederherzustellen.

Da bei den Composite-Filtern die Checkbox für das Aktivieren von Cuda nur vorhanden ist, wenn Cuda unterstützt wird, kann beim Speichern der Filterketten unterschiedliches XML entstehen. Die Checkbox wird in das XML eingefügt, wenn Cuda unterstützt wird (siehe Abbildung 4.13), ansonsten nicht. Dies kann zu Problemen führen. Wird z.B. die Filterketten ohne Cuda Unterstützung gespeichert, ist die Checkbox für das Aktivieren von Cuda nicht im XML vorhanden. Wird diese Filterkette nun aber mit Cuda Unterstützung geladen werden, wird die Checkbox im XML erwartet. Dies führt zu einem Fehler. Auch im umgekehrten Fall treten Probleme auf.

```
<Filter>
  <Type>Filter.CompositePatternMatching</Type>
  <Input Type="System.Int32" Value="0" />
  <Output Type="System.Double" Value="0" />
  <Output Type="System.Int32" Value="0" />
</Filter>
```

Abbildung 4.13: XML eines gespeicherten Composite Filters

Das Speichern und Laden von Filtern ist in der Basisklasse *PromonBaseFilter* geregelt. Es wäre aber unschön dort einzugreifen um einen Spezialfall des *CompositeFilters* zu handhaben. Deshalb wird das XML vor dem Laden in der Methode *CompositeFilter.CheckboxInXml* auf diese Fälle überprüft und falls nötig angepasst. Dies führt dazu, dass der *PromonBaseFilter* korrektes XML erhält und damit keine Probleme entstehen.

## 4.7 Frames parallelisieren

Um Frames parallel im *Promon200* verarbeiten zu können müssen einige Bedingungen erfüllt sein. Erstens müssen die Ausgabewerte der einzelnen Filter zwischen gespeichert werden, damit diese synchron weiterverarbeitet werden können. Dies ist vor allem für Controls mit mehreren Eingängen wichtig. Denn wenn dort die Eingabewerte von verschiedenen Frames vorhanden sind resultiert ein falsches Resultat. Dies wurde von Herr Schindler am 26.07.2011 mit Hilfe von *Queues* in den Verbindungen implementiert. Weiter muss zu allen Resultaten die Framenummer gespeichert werden, damit aus der *Queue* die zusammengehörenden Eingabewerte gelesen werden können. Weiter wurde mit der Einführung der *Queues* auch eine Framenummer eingebaut, welche bei jedem Ausgabewert mitgegeben wird. Allerdings wird diese bisher noch von keinem Control beachtet, da das parallele Ausführen noch nicht möglich ist.

Wenn Cuda-Filter mehrere Frames parallel verarbeiten, entsteht das Problem, dass sie sich gegenseitig den Speicher überschreiben. Denn die Cuda-Filter müssen den Filterbereich zuerst auf die Grafikkarte kopieren, bevor sie dieses verarbeiten können. Dafür wird derselbe Speicher immer wiederverwendet. Möchten mehrere Frames parallel verarbeitet werden, müssten die Cuda-Filter für jedes Frame diesen Speicher allozieren. Möglich wäre eine feste Anzahl an parallel zu verarbeitenden Threads zu definieren. Dadurch könnte mit Indizes, welche den Filtern beim Aufruf mitgegeben wird, auf verschiedene Speicherbereiche zugegriffen werden. C#-Filter haben dieses Problem nicht, da sie den Speicher nicht kopieren müssen.

Da in Cuda 3.2 aber immer nur auf Speicherbereiche des eigenen Cuda-Kontextes zugegriffen werden kann und jeder Thread einen eigenen Kontext bekommt, müsste noch ein Kontext Wechsel implementiert oder Cuda 4 verwendet werden. Dies ist in Kapitel 4.6.1 näher beschrieben.

## 5 Video Analyse

Zur Klassifizierung (gut/schlecht) eines Produktionsvorganges werden Konzepte ausgearbeitet und umgesetzt. Dadurch können fehlerhafte Produkte ausgesondert werden. Zusätzlich wird ein Control zur subframegenauen Frequenzbestimmung eines sich wiederholenden Vorgangs entwickelt.

### 5.1 Analysezeitpunkt bestimmen

Typischerweise wiederholt sich ein Produktionsvorgang (z.B. Deckel auf Flasche schrauben) immer wieder. Zuerst muss daher bestimmt werden, zu welchem Zeitpunkt des Vorganges erkannt werden kann, ob ein Fehler vorliegt (z.B. immer wenn der Deckel an einer gewissen Position ist). Dieser Zeitpunkt soll möglichst schnell mit Hilfe eines Filters gefunden werden. Eine Variante ist, eine regelmässige Farbänderung (oder Spiegelung) zu beobachten. Die Farbänderung kann z.B. mit Hilfe des *Average Brightness Filters* aufgezeigt werden. Der durchschnittliche Farbwert kann mit dem *Compare Control* auf einen Grenzwert getestet werden, wird dieser über- oder unterschritten ist das Objekt in Position (siehe Abbildung 5.1). Nun könnte der Zustand des Produktionszustandes mittels eines Klassifizierungsfilters (siehe Kapitel 5.2) auf Fehler geprüft werden.

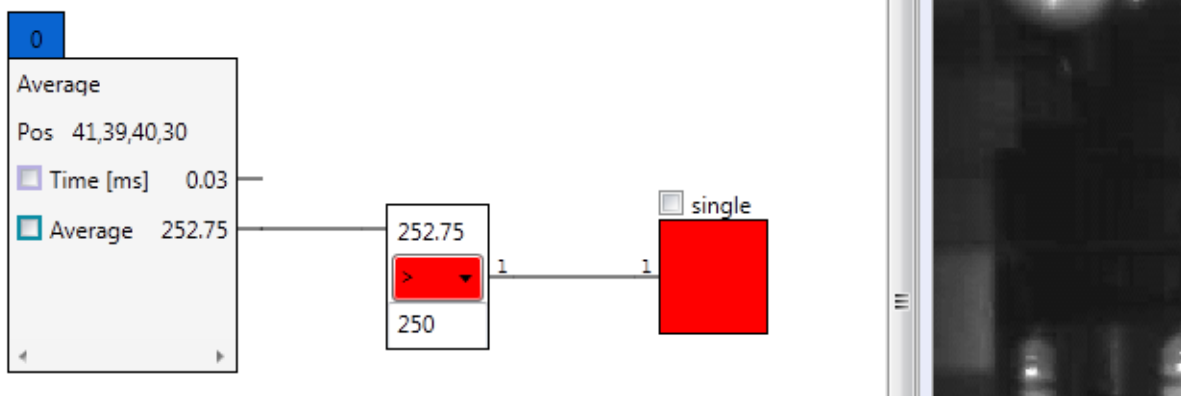


Abbildung 5.1: Analysezeitpunkt bestimmen mittels Grenzwert

#### 5.1.1 False-True-Trigger

Wird die Position eines Objektes während eines Produktionsvorganges wie in Kapitel 5.1 gefunden, werden möglicherweise mehrere Frames als korrekte Position erkannt werden, da der Grenz-

wert für mehrere Frames über- oder unterschritten wird. Dies ist aber nicht immer erwünscht. Manchmal soll die Bildanalyse genau einmal durchgeführt werden, sobald der Grenzwert über- oder unterschritten wird. Danach soll auf die nächste Periode (nächster Deckel) gewartet werden. Dies ist relativ einfach zu implementieren, in dem nur beim Wechsel von *false* auf *true* eine Aktion ausgeführt wird. Allerdings kann ein Problem entstehen, wenn der Grenzwert während einer Periode mehrmals überquert wird. Dies kann durch Ungenauigkeiten in den Frames geschehen und macht sich durch Zacken im Ausgabegraph bemerkbar (siehe Abbildung 5.2).

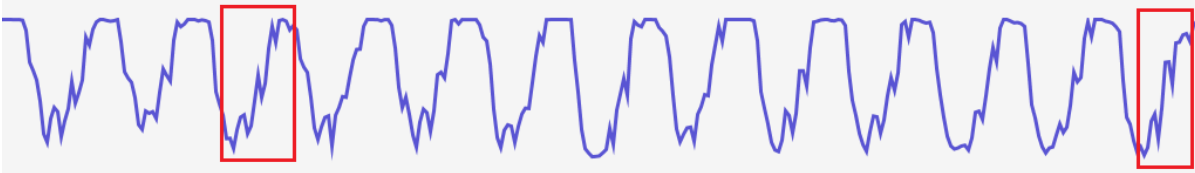


Abbildung 5.2: Unschöne Kantenübergänge in den Filterausgabewerten

Dafür wurde der *False-True-Trigger* erstellt. Wenn dieser *true* erhält, wechselt der Ausgabewert für genau ein Frame auf *true*, danach wechselt der Ausgabewert wieder auf *false*. Bevor dieser wieder auf *true* wechseln kann, muss eine konfigurierbare Anzahl an Frames vergangen sein. Dadurch können Zacken gehandhabt werden (siehe Abbildung 5.3).

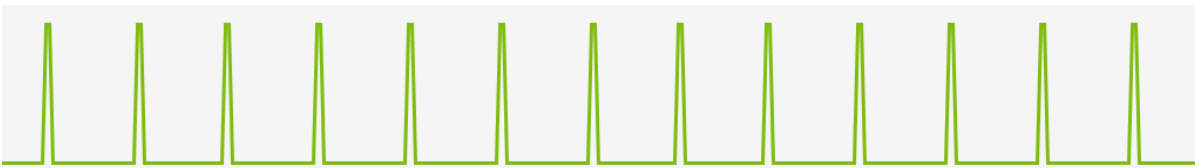


Abbildung 5.3: Output von False-True-Trigger

### 5.1.2 Frequenz messen

Mit Hilfe des *Frequenz Controls* kann die Frequenz gemessen werden. Es erwartet einen *bool* als Eingabewert und dividiert die Anzahl *true*s durch die Anzahl Frames. Im Textfeld oben, kann eingestellt werden, wie viele der letzten Frames beachtet werden sollen (siehe Abbildung 5.4). Im unteren Bereich ist die Frequenz ersichtlich.

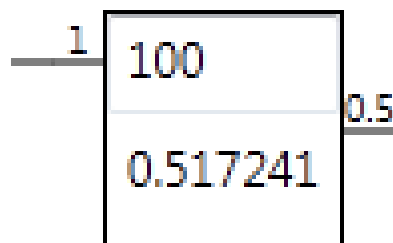


Abbildung 5.4: Frequenz Control, Anzahl verwendeter Frames ist konfigurierbar

Um das Alter der Eingabewerte nachträglich bestimmen zu können, wird eine Framenummer geführt. Alle *true*s werden mit der Framenummer in eine *Queue*<sup>16</sup> gespeichert. Neue Einträge haben eine höhere Framenummer und werden ans Ende der *Queue* angehängt. Dies ermöglicht es, die zu alten Einträge zu löschen, in dem die Framenummern der Einträge am Anfang der *Queue* überprüft und gegebenenfalls gelöscht werden. Sobald der erste Eintrag gefunden wird, welcher nicht gelöscht werden muss, kann aufgrund der gegebenen Reihenfolge in der *Queue* abgebrochen werden.

Die Anzahl beachteter Frames kann im laufenden Betrieb angepasst werden. Wenn diese verkleinert wird, werden die nicht mehr benötigten Einträge aus der *Queue* gelöscht. Wenn die Anzahl beachteter Frames vergrößert wird, wird die Anzahl gespeicherter Einträge kontinuierlich erhöht, bis die gewünschte Anzahl erreicht wird. In der Zwischenzeit wird nur durch die gespeicherte Anzahl Frames und nicht durch die Anzahl gewünschter Frames geteilt.

Um die Frequenz eines Arbeitsablaufes bestimmen zu können, wird idealerweise ein *False-True-Trigger* (siehe Kapitel 5.1.1) vorgeschaltet. Durch dieses wird nur ein Wechsel von *false* auf *true* als *true* weitergegeben und es können zackige Übergänge der Ausgabewerte gehandhabt werden.

## 5.2 Klassifizierung

Es wurden zwei Videos ausgewählt, welche auf fehlerhafte Produktionszustände untersucht werden. Dazu wird im ersten Teil das Video einer Waverbeschichtungsanlage analysiert und im zweiten Teil, eine Anlage, welche Deckel auf Flaschen montiert.

**Fehlerevent** Ein Fehlerevent in *Promon200* stellt einen Fehler beim derzeitige Frame dar.

### 5.2.1 Waver Beschichtung

Im ersten Video (zu finden unter *Videos/AR31.AVI*) geht es darum, den Fehlerfall einer Lackbeschichtungsanlage von Wavern zu erkennen. Es wird zuerst ein möglicher Ansatz beschrieben, danach die zugehörnde Implementierung mittels Cuda.

#### Fehlerdefinition

Die Beschichtung verläuft inkorrekt, wenn der Strahl unterbrochen wird oder sich Tröpfchen bilden (vgl. Abbildung 5.5).

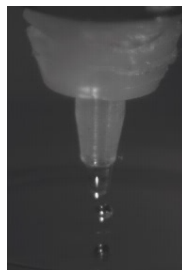


Abbildung 5.5: Fehler bei der Waverbeschichtungsanlage durch Tröpfchenbildung

<sup>16</sup><http://msdn.microsoft.com/en-us/library/7977ey2c.aspx>

## Konzept

Die Anlage kann sich in verschiedenen Zuständen befinden, was die Sache erschwert:

**Abgeschaltet** Strahl unterbrochen, da die Anlage nicht im Betrieb ist. Dies sollte nicht zwingend einen Fehlerevent auslösen.

**Gestartet** Der Strahl ist unterbrochen, da der Lackstrahl den Waver noch nicht erreicht hat. Dies sollte nicht zwingend ein Fehlerevent auslösen.

**In Betrieb - OK** Strahl ist durchgehend. Dies ist der Idealfall.

**In Betrieb - Tröpfchen** Strahl ist nicht durchgehend und es bilden sich Tröpfchen. Hier muss ein Fehlerevent ausgelöst werden.

Der Strahl ist vom menschlichen Auge vor allem durch die Lichtreflexion des Lackstrahls sichtbar. Die Reflexion ist jedoch nicht durchgehend, womit sich ein Ansatz, welcher auf eine kontinuierliche Reflexion prüft, nicht sinnvoll ist.

Ein weiterer Ansatz beruht auf der Farbänderung, die der Lackstrahl im Vergleich zum Hintergrund verursacht. Diese Farbänderung ist einiges schwächer als die Reflexion und von Auge schwer erkennbar, reicht aber um den Strahl maschinell zu erkennen. Dazu wird der Filterbereich zeilenweise durchgegangen und die Nachbarpixel auf eine Grauwertänderung überprüft. Liegt die mathematisch absolute Änderung unter einem konfigurierbarem Grenzwert, ist auf dieser Zeile ein Unterbruch vorhanden.

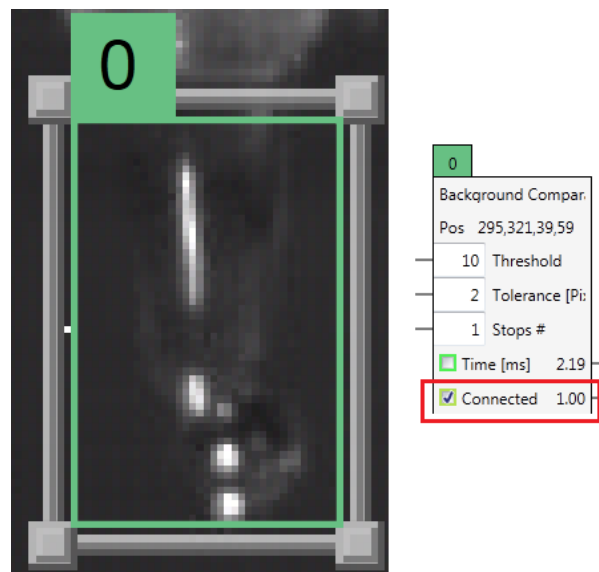


Abbildung 5.6: Anlage in Betrieb - Strahl OK

Durch mögliche Unschärfen oder Verpixelung bei den Frames werden teilweise Unterbrüche erkannt wo keine sind. Deshalb erlaubt der Filter die Konfiguration einer Toleranz, welche die Mindestanzahl an aufeinanderfolgenden Zeilen definiert, die einen Unterbruch darstellen. Dies ist in der Abbildung 5.6 zu sehen. Es wurde ein Unterbruch erkannt (markiert durch ein weisses Pixel links), welcher aber nur eine Zeile gross ist und aufgrund der Toleranz von zwei zu keinem Fehler führt.

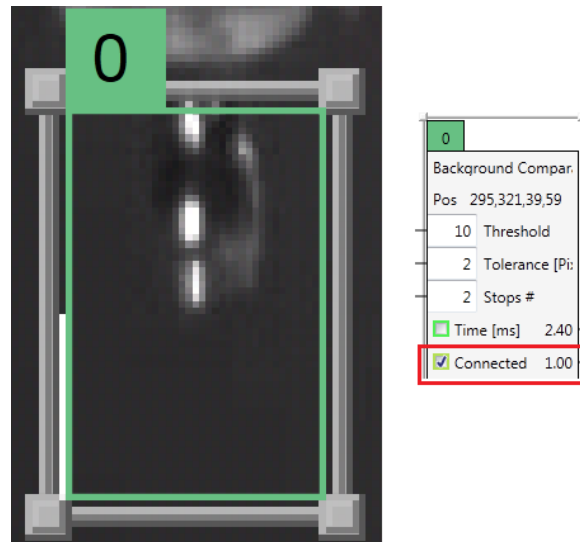


Abbildung 5.7: Anlage gestartet - Unterbruch aber kein Fehler

Mit den bisherigen Konfigurationsmöglichkeiten (Grenzwert und Toleranz) werden die Zustände *Abgeschaltet* und *Gestartet* auch als Fehler erkannt. Dies kann gewollt sein, es sollte aber die Möglichkeit bestehen, diese zu tolerieren. Deshalb wird eine weitere Konfigurationsmöglichkeit namens *Stopp*s eingeführt. Diese definiert die Mindestanzahl an Unterbrüchen die gefunden werden muss, damit ein Fehlerevent ausgelöst wird. Bei den Zuständen *Abgeschaltet* und *Gestartet* entsteht nur ein Unterbruch, beim Zustand *In Betrieb - Tröpfchen* mindestens Zwei (vor und nach dem Tröpfchen). Somit kann die minimale Anzahl Stopps auf Zwei gesetzt werden, damit das gewünschte Verhalten entsteht. In der Abbildung 5.7 ist der Startzustand zu sehen. Der Filter erkennt korrekterweise einen Unterbruch ab der Hälfte des Filterbereiches. Dieser stellt jedoch keinen Fehlerzustand dar, da es sich nur um einen Unterbruch handelt. Bei einer Tröpfchenbildung wird ein Fehlerzustand signalisiert, wie in der Abbildung 5.8 zu sehen ist.

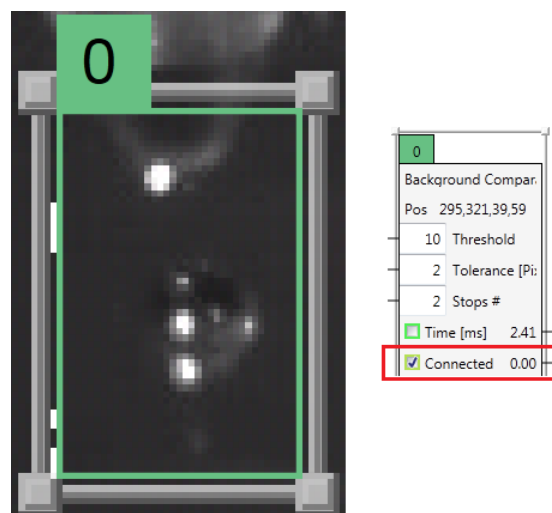


Abbildung 5.8: Anlage in Betrieb - Fehler wegen den Tröpfchen

### Implementation

Zusätzlich zum benötigten Speicher für das Frame werden  $n$  Bytes alloziert (nachfolgend Result Array genannt), wobei  $n$  der Filterhöhe entspricht. Diese  $n$  Bytes repräsentieren je eine Zeile des Filterbereichs. Zu Beginn werden sie mit 0 initialisiert, was für einen Unterbruch steht.

In einem ersten Ansatz wurde zeilenweise parallelisiert (*Pro Zeile*). Dies bedeutet, dass ein Thread alle Pixel einer Zeile durchläuft und die absolute Differenz zweier Pixel gegen den Grenzwert überprüft. Ist die Differenz grösser, wird im Result Array das entsprechende Byte auf 1 gesetzt (vgl. Abbildung 5.9). Anschliessend muss nur das Result Array zurückkopiert werden und auf die, im Ansatz beschriebenen, Bedingungen getestet werden.

0	1	1	0
12	15	15	20
12	22	22	20
12	22	22	20
12	15	15	20

Abbildung 5.9: Speicherverwaltung bei einem Grenzwert von 8

In der zweiten Version wurde der Filter weiter parallelisiert. Ein Thread vergleicht nur noch zwei Pixel (*Connection Checker Pro Pixel*). Dies ist möglich, da nur ein erkannter Strahl mit einem 1 im Result Array markiert wird und das gegenseitige Überschreiben mit 1 keine Rolle spielt. Diese Version ist performanter, wie in der Abbildung 5.10 zu sehen ist.

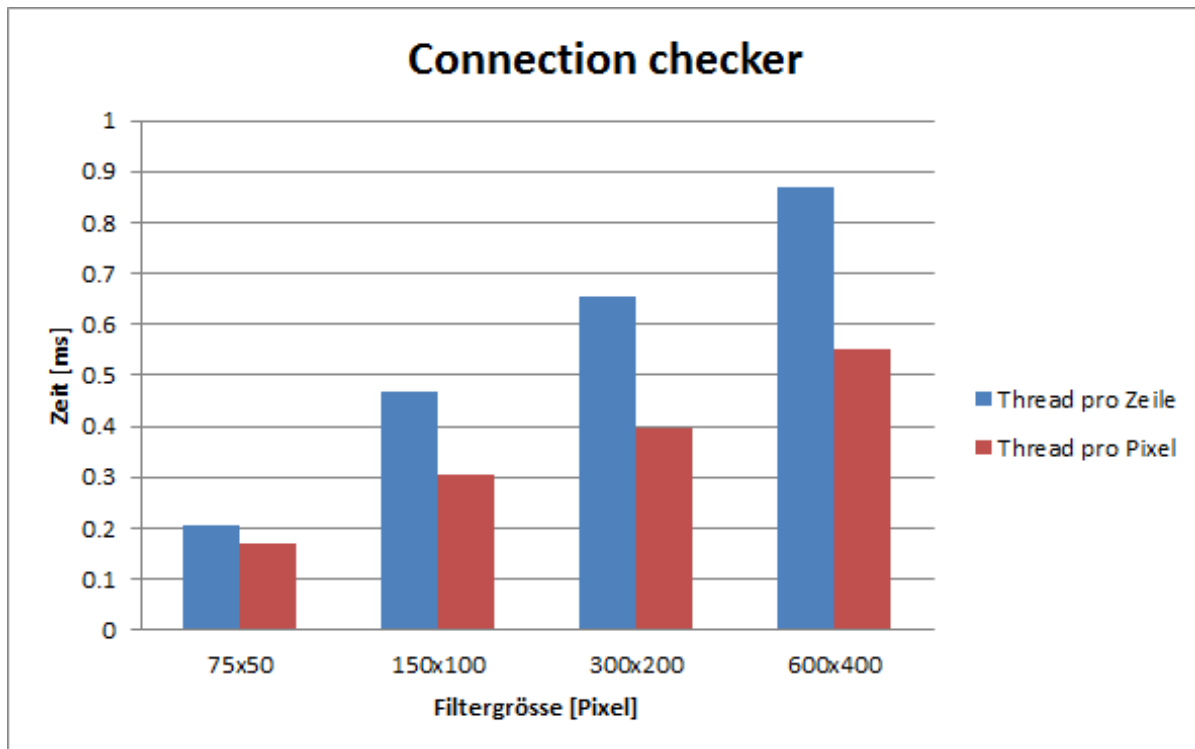


Abbildung 5.10: Performanzvergleich der Waverbeschichtungsfilter

## Anwendungsszenario

Dieser Filter wurde mit dem Video *Videos/AR31.AVI* getestet. Die dazugehörige Konfiguration befindet sich unter *Videos/waver.PM2*. Die Tröpfchenbildung wurde korrekt als Fehler erkannt. Es ergeben sich bei höheren Frames noch zwei False-Positives,. Durch die mangelhafte Auflösung ist der Strahl auch vom Auge her nicht sichtbar. Die Filterkette läuft auf dem Testsystem mit 400 FPS und erfüllt somit die Rahmenbedingung bezüglich Geschwindigkeit.

### 5.2.2 Deckelmontage

Als zweites Klassifizierungvideo wurde das Video *Videos/Stoerung3\_fehler.avi* gewählt. Es zeigt wie Deckel auf Flaschen montiert werden. Diese Deckel können defekt sein und sollen mit Hilfe eines Filters erkannt werden.

#### Fehlerdefinition

Defekte Deckel haben keine schöne Kante. Es können Dellen vorhanden sein, wie dies in Abbildung 5.11 zu sehen ist.

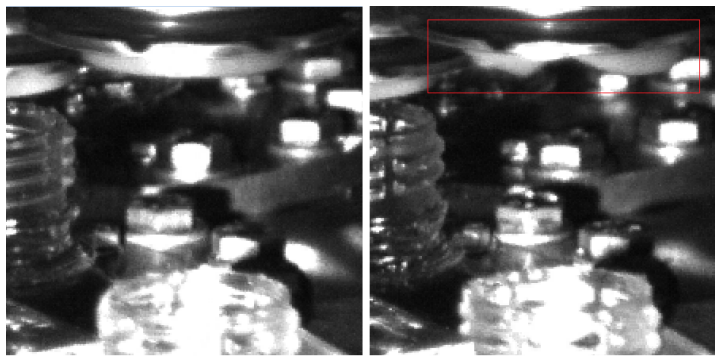


Abbildung 5.11: Deckel links korrekt, Deckel rechts fehlerhaft

#### Konzept

Im Video unterscheidet sich der Grauton des Deckels von der Umgebung. Dies kann genutzt werden, um die Deckelform zu erkennen. Mit Hilfe von zwei Grenzwerten, einem Minimum und einem Maximum, kann der Grauton des Objekts (hier der Deckel) eingegrenzt werden. Das Frame kann dann binarisiert werden, wobei alle Grauwerte innerhalb der Grenzen auf Weiss und alle anderen auf Schwarz geändert werden. In dem nun erhaltenen Bild kann die Kante des Objekts gut erkannt werden. Sie liegt beim Übergang von Weiss nach Schwarz.

Steht zusätzlich zum Frame noch ein Template von einem guten Deckel zur Verfügung, kann die Kante des Templates gegen diejenige des Frames verglichen und die Differenz gegen eine maximale Differenz getestet werden. Wird diese überschritten, ist der Deckel defekt.

#### Implementation

Zuerst wird der Framebereich mit Hilfe von zwei einstellbaren Grenzwerten binarisiert. Dies kann parallel pro Pixel geschehen, da jedes Pixel unabhängig von allen anderen Pixeln ist.

Danach wird der binarisierte Framebereich mit dem binarisierten Template verglichen. Sind diese unterschiedlich gross, wird nur der kleinere Bereich verglichen. Für den Vergleich wird ein Thread pro Spalte verwendet. Der Thread geht Pixel für Pixel durch die Spalte und sucht im Framebereich, wie auch im Template, nach dem ersten schwarzen Pixel. Dieses symbolisiert die Kante. Hier ist es wichtig, dass die Kante (besonders auf dem Template) sauber verläuft, da ein schwarzes Pixel in der ersten Zeile bereits als Kante erkannt wird (siehe Abbildung 5.12). Als Konfigurationshilfe kann die Checkbox *Config* angewählt werden, dadurch erhält der Benutzer eine Rückmeldung mittels Einfärbungen der Kantenunterschiede.

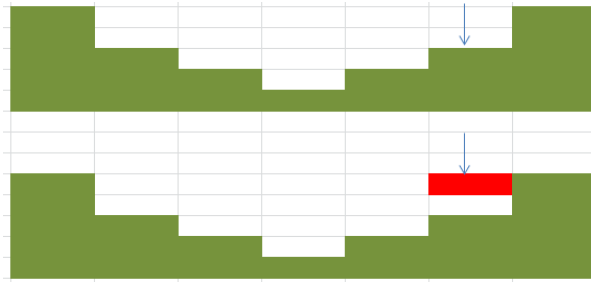


Abbildung 5.12: Spaltenweise Suche nach der Kante (erstes schwarzes Pixel)

Die Kantenverschiebung berechnet sich aus der Differenz der Zeilenindizes, auf welchen das erste schwarze Pixel gefunden wurde. Diese Kantenverschiebung ist in Abbildung 5.13 durch die Einfärbung sichtbar, welche im Konfigurationsmodus angezeigt wird. Weisse und schwarze Pixel befinden sich ausserhalb der Kantenverschiebung. Grüne Pixel kennzeichnen die Kante des Deckels im Template, während blaue Pixel die Kante des Deckels im Frame zeigen. Bei einem perfekten Match ist das Bild schwarz-weiss.



Abbildung 5.13: Links: binarisiertes Template; Rechts: Kantenvergleich

Die Differenzen pro Spalte werden nun quadriert und aufsummiert um eine Gesamtdifferenz zwischen Template und Framebereich zu erhalten. Durch die Quadrierung resultieren immer positive Werte, wodurch sich positive und negative Unterschiede nicht ausgleichen können. Ausserdem hilft sie kleine Verschiebungen um wenige Pixel weniger zu gewichten, als grosse Kantenänderungen. Defekte Deckel haben an gewissen Orten besonders grosse Kantenverschiebungen wie in Abbildung 5.14 zu sehen ist. Dadurch können diese Deckel gut erkannt werden. Die Gesamtdifferenz wird am Schluss durch die Anzahl Spalten dividiert, dadurch resultieren auch bei einem höheren Pixelstep ähnliche Werte.

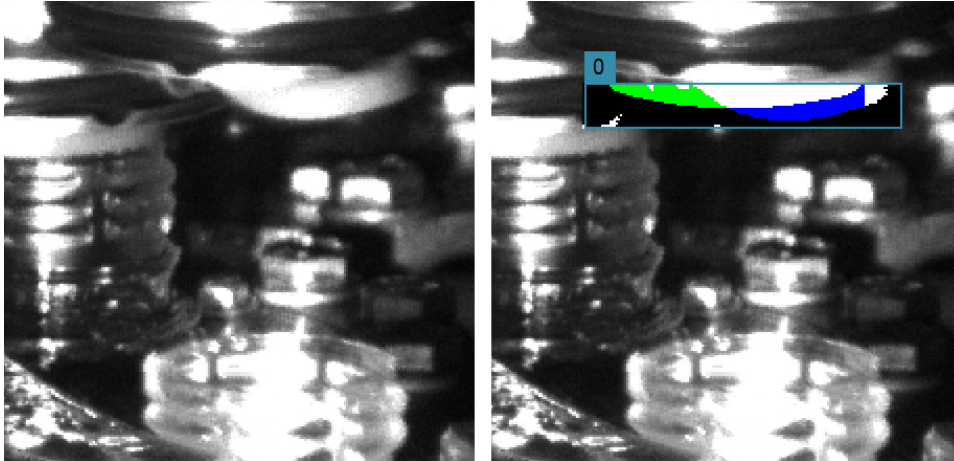


Abbildung 5.14: Links: Originalbild von defektem Deckel; Rechts: Kantenvergleich

Für die Analyse benötigt der Filter (*Object Comparator*) auch bei einer Filtergröße von 600x400 Pixel weniger als eine Millisekunde (siehe Abbildung 5.15).

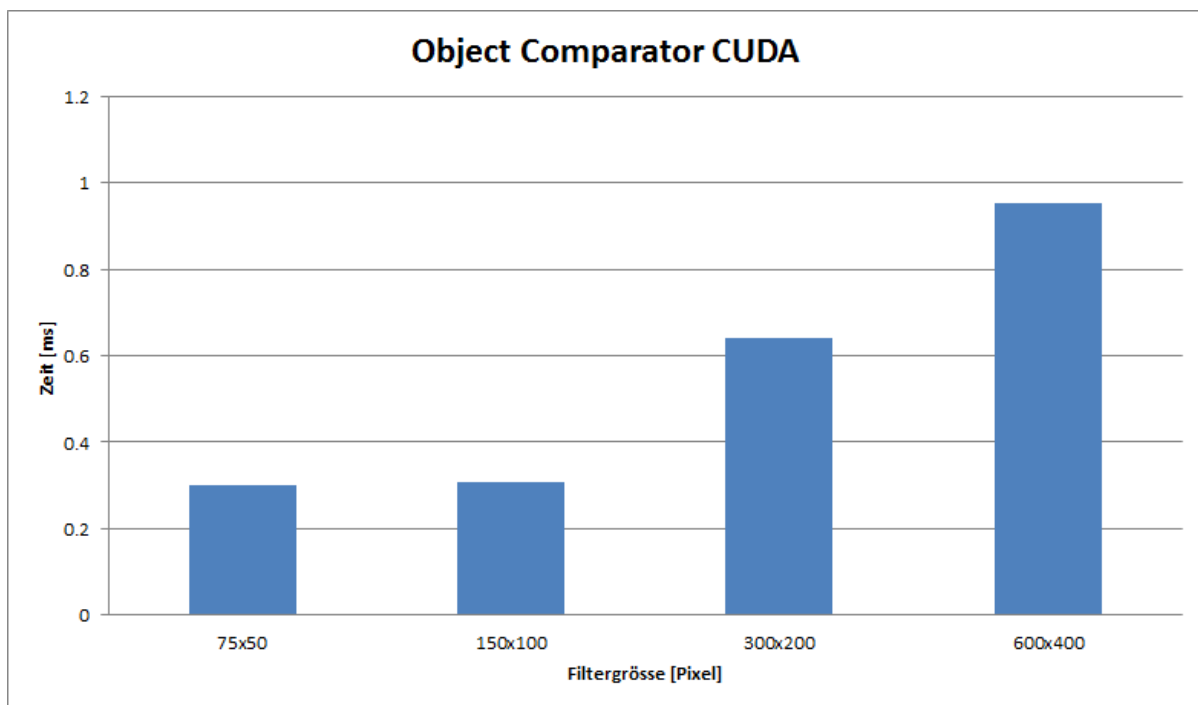


Abbildung 5.15: Geschwindigkeitsvergleich bei verschiedenen Filtergrößen

### Anwendungsszenario

Um den *Object Comparator* zu testen wurde er auf das Video *Videos/Stoerung3\_fehler.avi* angewendet. Es konnten 450 Frames pro Sekunde überprüft werden. Alle defekten Deckel wurden erkannt und es wurde kein Fehlalarm ausgelöst.

Um die Tests nachzuvollziehen kann die Filterkette aus der Datei *Videos/objectComparator*

*\_stoerung3.PM2* geladen werden. Da die Filter Eingabebilder enthalten, welche separat gespeichert werden, müssen die Bilder, welche sich Ordner *Videos/objectComparator\_stoerung3* befinden, in den Ausführungspfad kopiert werden.

## 5.3 Automatische Konfiguration

Im Gespräch mit dem Kunden kristallisierte sich heraus, dass eine einfachere Konfiguration der Klassifizierungsfiler (siehe Kapitel 5.2) wünschenswert wäre. Der *Object Comparator* hat z.B. 3 Eingabewerte, von welchen das Resultat abhängt. Die Werte der ersten beiden Eingänge müssen zwischen 0 und 255 liegen, da sie einen Grauwert darstellen. Der letzte Eingabewert ist nur durch den Wertebereich von *double* beschränkt, liegt aber bei den zur Verfügung stehenden Videos zwischen 0 und 100. Durch die Wertebereiche der Eingabeparameter sind hier also  $256 * 256 * 101 \approx 6.6$  Millionen Konfiguration möglich.

Da aber alle Eingabewerte wichtig sind und daher nicht reduziert werden können, wurde ein anderer Ansatz gewählt. Die Filter sollen mit Hilfe eines Algorithmus konfiguriert werden.

### 5.3.1 Konzept

Die Klassifizierungsfiler entscheiden, ob der Zustand im Filterbereich korrekt ist. Allerdings kann ohne weitere Informationen nicht festgestellt werden, ob die Entscheidung korrekt war. Daher werden für die automatische Konfiguration Frames benötigt (nachfolgend Lernframes genannt), welche korrekte sowie auch nicht korrekte Zustände darstellen. Zu diesen Frames muss das erwartete Ergebnis gespeichert werden. Nun kann der Klassifizierungsfiler mit einer beliebigen Konfiguration auf die Lernframes angewendet werden. Entspricht das Ergebnis des Filters der gespeicherten Erwartung, ist die Konfiguration für dieses Frame korrekt.

#### Bewertung

Es ist einfach eine Konfiguration zu finden, welche alle korrekten Zustände als korrekt erkennt. Denn es gibt Konfigurationen, welche nahezu unabhängig vom Frame *true* zurückgeben. So wird zum Beispiel der *ObjectComparator* immer eine identische Kante wie das Template finden, wenn aufgrund der Grenzwerte Frame und Template schwarz werden. Das Gleiche gilt für Konfigurationen zum Erkennen von inkorrekten Zuständen. Eine sinnvolle Konfiguration wird aber bei korrekten Zuständen keinen Fehlalarm auslösen und bei allen nicht korrekten Zuständen immer einen *Fehlerevent* werfen. Beides ist gleichermassen wichtig. Dies fließt auch in die Bewertung der Konfiguration ein. Eine perfekte Konfiguration erhält eine Bewertung von 100%, sie gibt bei allen Lernframes das erwartete Resultat zurück. Für das Erkennen von korrekten Zuständen als korrekt kann der Filter maximal 50% erreichen. Die restlichen 50% werden durch das Erkennen von nicht korrekten Zuständen als nicht korrekt vergeben. Durch diese Aufteilung wird sichergestellt, dass beide Situationen gleich gewichtet werden, auch wenn z.B. von fehlerhaften Zuständen weniger Lernframes zur Verfügung stehen. Es wäre auch möglich das Erkennen von fehlerhaften Zuständen stärker zu gewichten.

$$\text{Bewertung} = 100 * \left( \frac{\text{ok}}{\text{anzOk}} * 0.5 + \frac{\text{notOk}}{\text{anzNotOk}} * 0.5 \right)$$

**ok** Anzahl korrekter Frames, welche als korrekt erkannt wurden

**anzOk** Anzahl korrekter Frames

**notOk** Anzahl nicht korrekter Frames, welche als nicht korrekt erkannt wurden

**anzNotOk** Anzahl nicht korrekter Frames

## Algorithmus

Ziel ist das Finden einer Konfiguration, welche bei allen Lernframes das erwartete Resultat zurückgibt. Eine Konfiguration, bei welcher nicht 100% der Resultate korrekt waren, wird im Betrieb falsche *Events* werfen. Gibt es keine perfekte Konfiguration, soll diejenige gewählt werden, welche bei den meisten Lernframes das erwartete Resultat zurückgibt. 1% Unterschied ist bereits entscheidend, da so die Anzahl falscher *Events* verringert werden kann. Die Konfiguration eines Filter ist keine tägliche Arbeit. Er wird nur neu konfiguriert, wenn falsche *Events* geworfen wurden.

Aufgrund dieser Umstände wird ersichtlich, dass die benötigte Zeit, um eine Konfiguration zu finden, nicht im Vordergrund steht. Es ist aber besonders wichtig, die beste Konfiguration zu finden. Dies wurde bei der Wahl des Algorithmus berücksichtigt.

## Genetischer Algorithmus

Mit einem genetischen Algorithmus kann eine perfekte Konfiguration möglicherweise schnell gefunden werden, insbesondere wenn viele perfekte Konfigurationen zur Verfügung stehen. Grundsätzlich ist er aber geeigneter um gute Konfigurationen zu finden. Ein weiteres Problem ist, dass, falls keine perfekte Konfiguration gefunden wird, nicht festgestellt werden kann, ob die gefundene Konfiguration die Bestmögliche ist. Da es in diesem Fall sehr wichtig ist, dass die beste Konfiguration gefunden wird, ist dieser Ansatz hier nicht geeignet.

## Brute Force

Durch das Testen aller möglichen Konfigurationen kann sichergestellt werden, dass die beste Konfiguration gefunden wird. Wird eine perfekte Konfiguration gefunden kann der Algorithmus abgebrochen werden. Die Laufzeit dieses Algorithmus ist

$$time \times frames \prod_{i=1}^n ((max_i - min_i) / step_i + 1).$$

**time** benötigte Zeit für einen Filteraufruf

**frames** Anzahl Lernframes, welche zur Verfügung stehen

**n** Anzahl konfigurierbarer Eingabeparameter

**max.i** maximaler Wert des i-ten Eingabeparameters

**min.i** minimaler Wert des i-ten Eingabeparameters

**step.i** Schritt zwischen 2 Werten des i-ten Eingabeparameters

Aufgrund der Anforderungen passt dieser Ansatz perfekt und wurde gewählt.

## 5.3.2 GUI

Das Werkzeug zur automatischen Konfiguration wurde, wie in Abbildung 5.16 zu sehen ist, implementiert. Alle konfigurierbaren Filter, welche zur Zeit verwendet werden, werden oben in der Dropdown-Liste angezeigt. In der Tabelle sind alle automatisch konfigurierbaren Eingänge, des in der Dropdown-Liste gewählten Filters, ersichtlich. Da die Laufzeit für die Konfigurationssuche mit der Anzahl Konfigurationsmöglichkeiten steigt, können diese begrenzt werden. Durch das Angeben eines Minimums und eines Maximums wird der Wertebereich des Eingabeparameters eingeschränkt. Zusätzlich kann der Schritt zum nächsten Wert angegeben werden. Durch diese Anpassungen kann die Laufzeit für das Suchen der besten Konfiguration verkürzt werden. Die maximale Laufzeit ist unten Links ersichtlich. Mit den Schaltflächen kann das aktuelle Frame zu den Lernframes hinzugefügt werden. Die Anzahl vorhandener Lernframes

wird in Klammern angegeben. Wird die Suche über die zur Verfügung stehende Schaltfläche gestartet, kann in den anderen Tabs normal weitergearbeitet werden, es kann allerdings zu Geschwindigkeitseinbußen kommen.

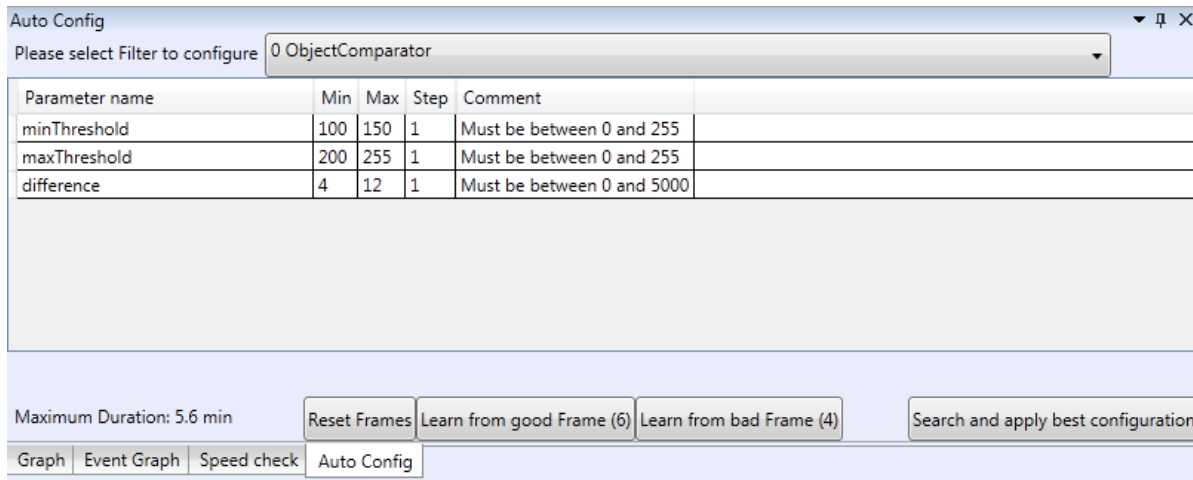


Abbildung 5.16: Werkzeug zum automatischen Konfigurieren eines Filters

### 5.3.3 Implementation

Filter welche automatisch konfiguriert werden können, implementieren das *IAutoConfigFilter* Interface implementieren.

```

1 /// <summary>
2 /// Get minimum and maximum of the parameters. Gives always the
3 /// same reference, so that the configuration can be changed.
4 /// </summary>
5 /// <returns>list of the properties of the input parameters</
  returns>
6 ParamProperties [] GetBorders ();
7
8 /// <summary>
9 /// Get list with references to the input parameters.
10 /// </summary>
11 /// <returns>list of references to the input parameters</returns>
12 FilterInputList GetInputParameterList ();

```

Listing 5.1: Methoden des *IAutoConfigFilter* Interfaces

Mit der *GetInputParameterList* Methode wird eine Liste von allen automatisch konfigurierbaren Eingängen zurückgegeben. Dies ermöglicht das automatische Konfigurieren von Filtern, welche Bilder als Eingabewerte aufweisen. Da die Eingabebilder nicht automatisch generiert werden können und sollen, wird dasjenige verwendet, welches im GUI für den Filter gewählt wurde.

Die *GetBorders* Methode liefert die Wertebereiche der einzelnen Parameter zurück. Dies ist notwendig, um alle möglichen Konfigurationen zu testen. Diese Wertebereiche können durch den Benutzer weiter eingeschränkt werden.

Zum automatischen Konfigurieren von Filtern wird ein *AutoConfig* Objekt erstellt, welchem der zu konfigurierende Filter übergeben wird. Der Filter wird geklont und weiter wird nur noch mit

dem Klon gearbeitet. Dadurch wird die originale Verarbeitungskette nicht beeinflusst. Insbesondere werden bei den Filteraufrufen die Rückgabewerte nicht in die *ConnectionQueue* (Queue in welche die eigenen Ausgabewerte geschrieben werden und von welcher der Verbindungspartner die Eingabewerte liest) gefüllt. Zusätzlich wird dadurch ermöglicht, dass der Benutzer normal mit den restlichen GUI Komponenten weiterarbeiten kann.

#### 5.3.4 Ausblick

Sinnvollerweise sollten die Frames, mit welchen ein Filter konfiguriert wurde, abgespeichert werden. Denn wenn der Filter im Betrieb bei einem Frame das falsche Resultat zurückliefert, muss dieser neu konfiguriert werden. Damit die Konfiguration verbessert werden kann, ist es sinnvoll, die alten Lernframes wieder zu verwenden, ansonsten gehen möglicherweise Informationen verloren. Zusätzlich muss das Frame, bei welchem der Filter das falsche Resultat zurücklieferte, zu den Lernframes hinzugefügt werden.

Ausserdem wäre es hilfreich, wenn die verwendeten Lernframes angeschaut und einzeln wieder gelöscht werden könnten. Zur Zeit ist nur ein kompletter Reset möglich. Dies war nicht Bestandteil des Auftrages und soll als Idee für die zukünftige Entwicklung aufgefasst werden.

## 6 Entscheidungskomponente

*Promon200* bietet verschiedene Massnahmen zur Optimierung der Verarbeitungsgeschwindigkeit, falls die Hardware zu schwach oder die Filterkette zu schlecht ist. Die bisherigen Massnahmen sind ohne Benutzerinteraktion und werden folglich automatisch angewendet, falls die gewünschte Framerate im laufenden Betrieb nicht erreicht wird. Zuerst werden die bisherigen Massnahmen analysiert und ausgewertet. Anschliessend werden Massnahmen mit Benutzerinteraktion ausgearbeitet und umgesetzt.

### 6.1 Optimierungen ohne Benutzerinteraktion (Analyse)

Die vorhandenen Optimierungsmassnahmen sind einerseits das Überspringen von Pixeln und andererseits das Überspringen von Frames. Die beiden Massnahmen werden zuerst beschrieben und danach auf ihren Performanzgewinn untersucht. Es werden auch die Unterschiede zwischen C#- und Cuda-Implementationen analysiert. Anhand einer Filterkette werden die Auswirkungen auf das Gesamtergebnis untersucht und die Änderungen der Ausgabewerte eines einzelnen Filters ausgewertet.

#### 6.1.1 Pixel überspringen

Bei dieser Methode wird jeweils nur jedes  $n$ . Pixel in x- und y-Richtung beachtet. Dies bedeutet, dass sich der Aufwand auf  $\frac{1}{n^2}$  verringert. Es wurde nun analysiert, wie sich diese Optimierungsmassnahme auf die Verarbeitungszeit und den Ausgabewert auswirkt.

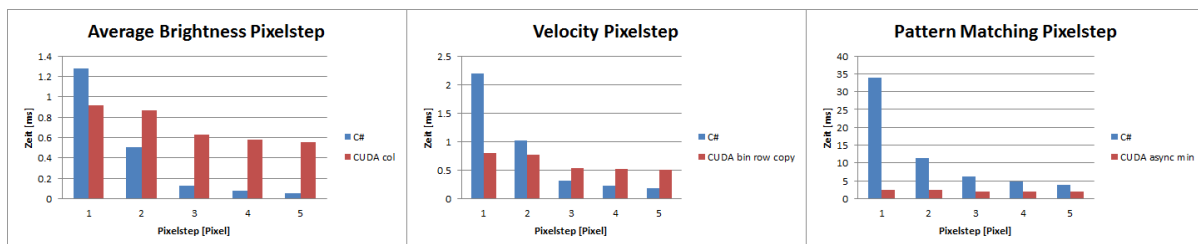


Abbildung 6.1: Geschwindigkeitsvergleich zwischen C#- und Cuda-Filtern mit Pixelstep

Für die Zeitmessung wurden die Filter in der Testsuite mit einer Filtergrösse von 600x400 Pixeln isoliert getestet. Es wurde nicht die gesamte Filterkette getestet, damit es keine Beeinflussung der verschiedenen Filter untereinander gibt. Wie in der Abbildung 6.6 zu sehen ist, gibt es deutliche Unterschiede zwischen den C#- und den Cuda-Versionen der verschiedenen Filter. Allgemein kann gezeigt werden, dass der Pixelstep grössere Auswirkungen auf C#-Filter hat, als auf Cuda-Filter. Während die Einsparung bei den Ersteren bei über 50% liegt, ist sie bei Cuda-Filtern nur zwischen 5% und 10% pro Schritt. Dies kann mit dem bereits gezeigten Overhead der Cuda-Architektur begründet werden. Cuda-Filter sind besser bei grossen Verarbeitungsmengen, da sie besser skalieren und der Overhead prozentual weniger ausmacht, wogegen C#-Filter in

der Verarbeitungszeit linear ansteigen. Der Pixelstep kann mit dem Verkleinern eines Filters verglichen werden, wobei dies nicht für jeden Filter zutrifft (z.B. beim Pattern Matching muss dennoch das ganze Integralbild berechnet werden). Auch wird beim Cuda-Filtern immer noch der ganze Filterbereich kopiert, da das Umkopieren jedes  $n$ . Pixels zu kostenintensiv wäre.

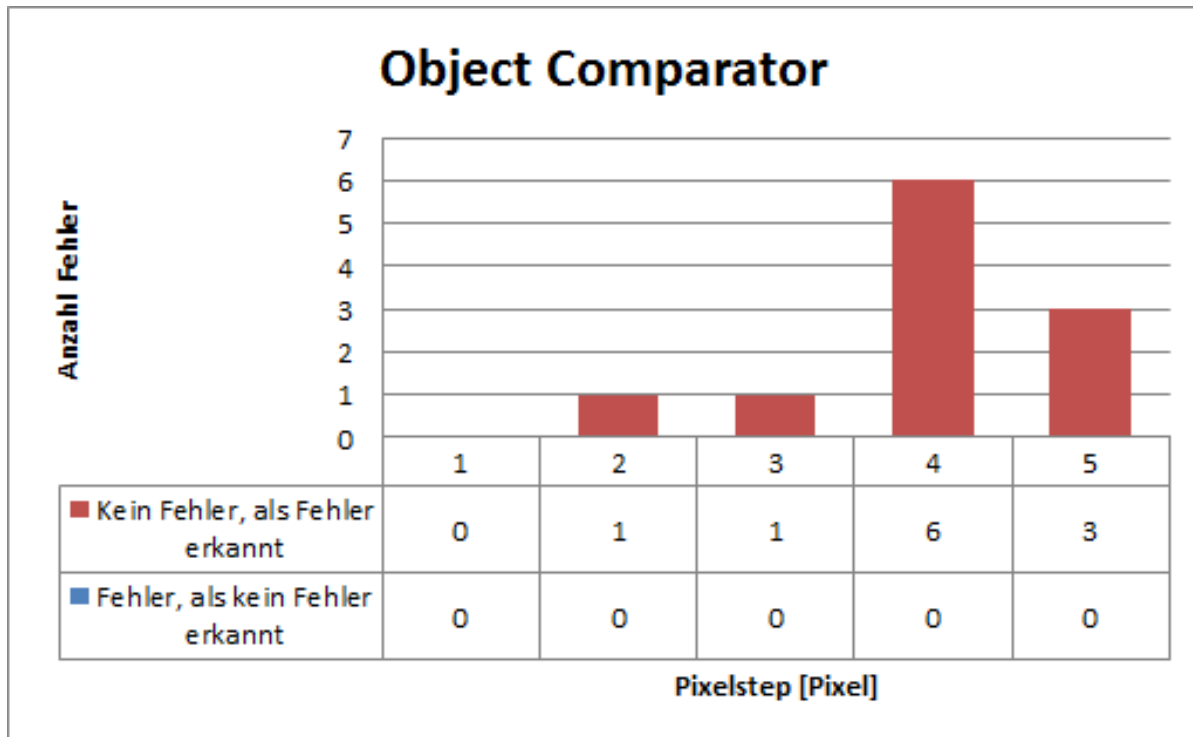


Abbildung 6.2: Pixelstep Geschwindigkeitsvergleich zwischen C#- und Cuda-Filtern

Zur Analyse der Veränderungen der Ausgabewerte wurden die Filter *Object Comparator* und *Average Brightness* ausgewählt. Im einen Fall wird eine Filterkette, im anderen ein einzelner Filter getestet. Beim *Object Comparator* wurde darauf geachtet, wie sich die ausgelösten Events bei verschiedenen Pixelsteps verhalten. Es ist wichtig, dass Fehler immer noch erkannt werden und weniger schlimm, falls es ein False-Positiv<sup>17</sup> gibt (ein falsches Produkt beim Kunden richtet mehr Schaden wegen möglicher Klagen an, als ein aussortiertes Korrektes). Beim Testvideo *Videos/Stoerung3\_fehler.avi* gibt es nur einen inkorrekten Deckel. Bei jedem Pixelstep wurde richtigerweise ein Event für diesen Deckel ausgelöst. Jedoch stieg die Anzahl an False-Positives massiv an wie in der Abbildung 6.2 zu sehen ist. Man kann die Anzahl an False-Positives mit ansteigendem Pixelstep nicht abschätzen, da diese abhängig von der Konfiguration ist. Es muss jedoch angemerkt werden, dass die Filterkonfiguration für jeden Pixelstep angepasst werden könnte, um das Resultat zu verbessern. Dazu müssten die Filter erweitert werden, damit sie mit mehreren Konfigurationen, abhängig vom Pixelstep, arbeiten könnten. Zusätzlich müsste die Komponente zur automatischen Konfigurationen, mit verschiedenen Pixelsteps simulieren, um die jeweils beste Konfiguration herauszufinden. Dies wurde im Rahmen dieser Arbeit nicht umgesetzt.

Beim Average Brightness Filter wurde die mathematisch absolute prozentuale Abweichung zum Originalwert gemessen. Hier sieht man relativ klar, dass die Abweichung grösser wird, je höher der Pixelstep (vgl. Abbildung 6.3) ist. Man von Ausreißern nicht gefeit, wie man in den ersten 20 Frames mit Pixelstep 4 sieht.

<sup>17</sup>[http://de.wikipedia.org/wiki/False\\_positive#Wahrheitsmatrix:\\_Richtige\\_und\\_falsche\\_Klassifikationen](http://de.wikipedia.org/wiki/False_positive#Wahrheitsmatrix:_Richtige_und_falsche_Klassifikationen)

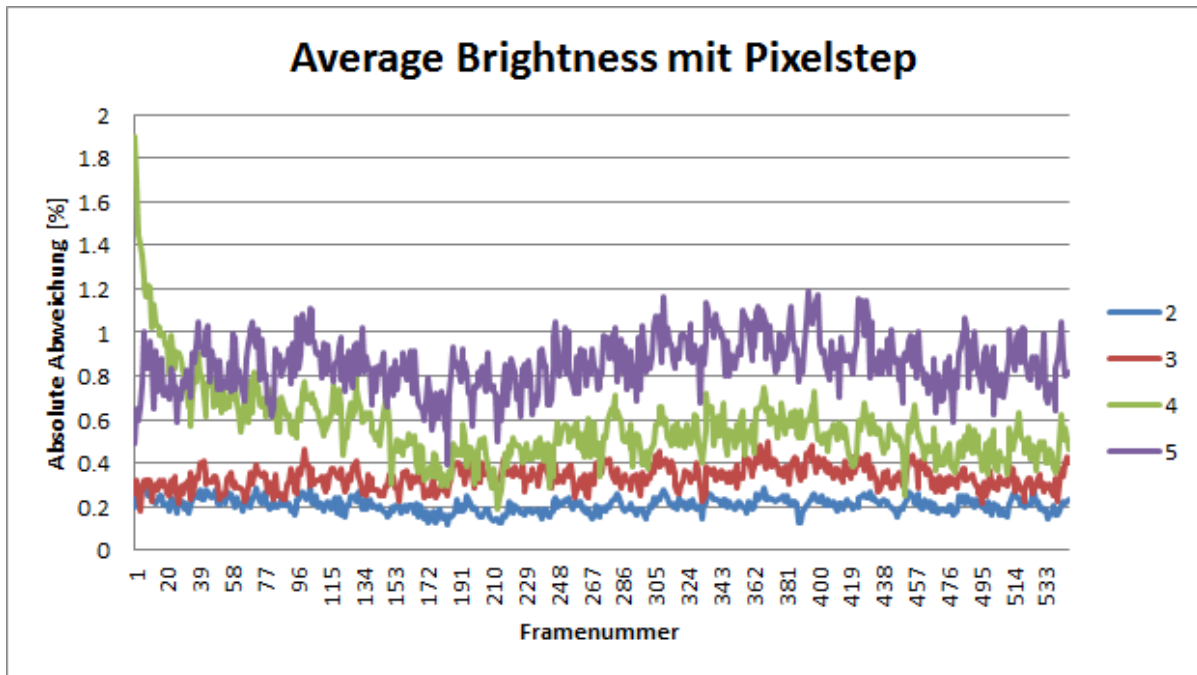


Abbildung 6.3: Ausgabewerte des Average-Brightness Filters bei verschiedenen Pixelsteps und einer Grösse von 600x400 Pixeln

### 6.1.2 Frames überspringen

Beim Framestep wird jeweils nur jedes  $n$ . Frame verarbeitet. Auch wenn ein Frame nicht verarbeitet wird, wird es von *Promon200* geladen. Zuerst wurde die Geschwindigkeit bei der Filterkette zur Erkennung defekter Deckel gemessen. Diese Filterkette ist unter Videos/objectComparator\_stoerung3.PM2 zu finden. Die FPS Rate erhöht sich wie erwartet mit zunehmenden Framestep (siehe Abbildung 6.4). Da die Frames dennoch geladen werden, verdoppelt sich die Geschwindigkeit bei einer Verdopplung des Framesteps nicht. Weiter ist zu sehen, dass der Geschwindigkeitsgewinn auf dem Testsystem ab einem Framestep von 4 insignifikant wird und man praktisch nur noch das Laden von Frames misst.

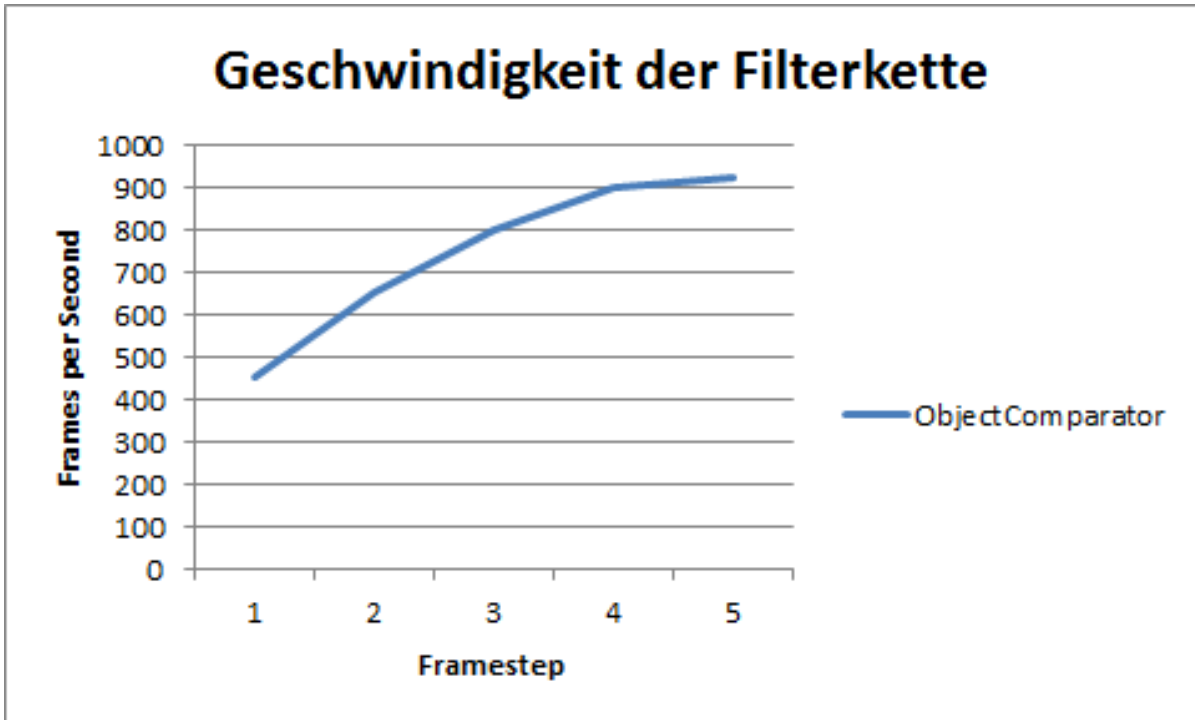


Abbildung 6.4: Framerate der Filterkette zur Erkennung falscher Deckel bei verschiedenen Framesteps

Für die Auswertung der Ausgabewerte wurde wiederum die Filterkette zur Erkennung defekter Deckel verwendet. Man sieht in der Abbildung 6.5, dass der falsche Deckel schon ab Framestep 3 nicht mehr erkannt wird. Dies hat zwei Ursachen: Erstens wird das fehlerhafte Frame übersprungen. Zweitens wurde der False-True-Trigger mit Framestep 1 konfiguriert. Dadurch werden jetzt Framestep Mal mehr Frames abgewartet bis die Ausgabe ändert. Teilt man die Anzahl abzuwartender Frames durch den Framestep wird der fehlerhafte Deckel zwar erkannt, jedoch gibt es auch viele False-Positives.

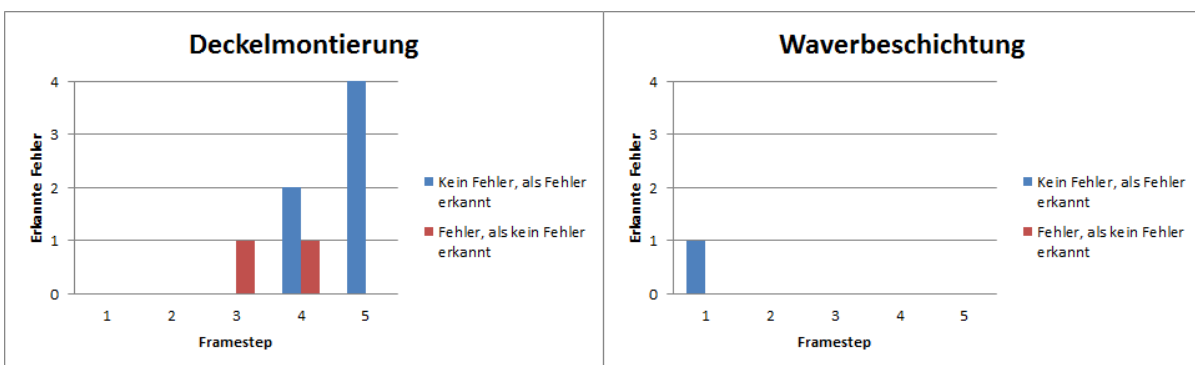


Abbildung 6.5: Auswertung der ausgelösten Fehlererevents bei verschiedenen Framesteps

Als weiterer Test wurde die Filterkette zur Erkennung einer korrekten Waverbeschichtung verwendet (zu finden unter Videos/waver.PM2). Wie im Kapitel 5.2.1 beschrieben, liefert diese Filterkette bereits bei Framestep 1 ein False-Positive. Wie in der Abbildung 6.5 zu sehen ist,

wird die Tröpfchenbildung bei jedem Framestep korrekt erkannt. Das False-Positive verschwindet ab Framestep 2, da das als inkorrekt erkannter Strahl Frame übersprungen wird. Dass sogar bei hohem Framestep der Fehler noch richtig erkannt wird, liegt an der grossen Anzahl an Frames (ca. 10), die den Fehler enthalten.

## 6.2 Optimierungen mit Benutzeraktion

Der Benutzer kann seine Filterkette auch manuell optimieren. Da er dazu aber tiefes Wissen über die *Promon200*-Architektur und Bildanalyse braucht, muss dies vereinfacht werden. Das Ziel ist dem Benutzer Tipps zu geben, wie er seine Filterkette performanter gestalten kann.

### 6.2.1 GUI

Der Benutzer hat grundsätzlich zwei Möglichkeiten die gesamte Verarbeitungszeit zu optimieren. Erstens kann er die Filtergrössen verkleinern, was zu einer kleineren Verarbeitungszeit der einzelnen Filter führt. Zweitens kann er bei Composite-Filtern die performantere Version (Cuda oder C#) wählen.

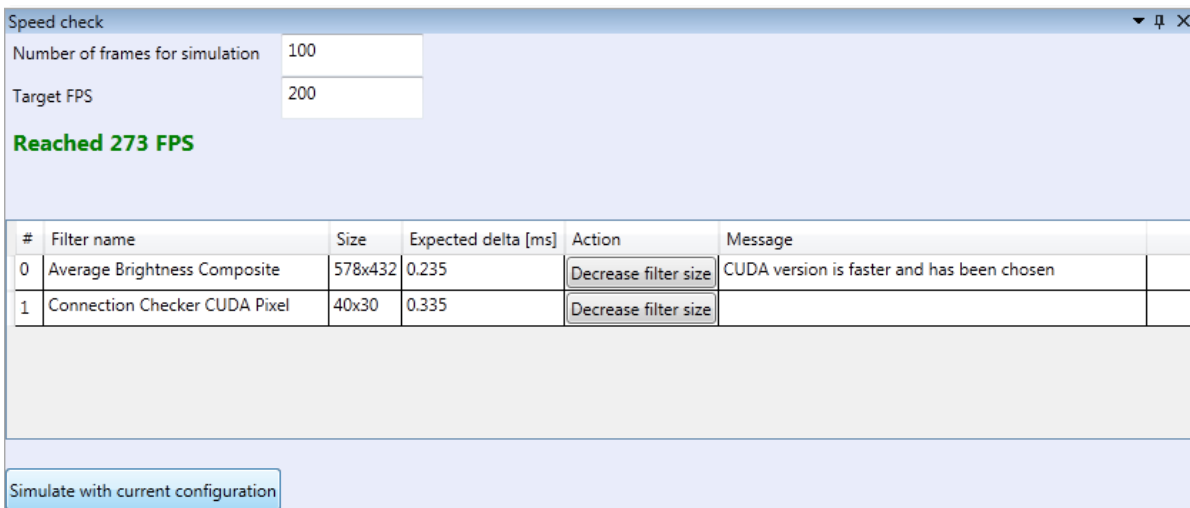


Abbildung 6.6: Speed Check Tab, welches zur Optimierung der Filterkette dient

Zur Geschwindigkeitsoptimierung wurde ein zusätzliches Tab im linken Bereich zu *Promon200* hinzugefügt, welches dem Benutzer Tipps zur Optimierung gibt (Abbildung 6.6). Das Video mit den Filterbereichen auf der rechten Seite ist immer noch sichtbar, wo der Benutzer die Möglichkeit hat, die Grösse und Position von Filtern direkt anzupassen. Die angestrebte Framerate kann definiert werden, wie auch die Anzahl Frames über welche die mittlere Geschwindigkeit eruiert wird.

Die Geschwindigkeitsanalyse muss explizit über die Schaltfläche "Simulate with current configuration" ausgelöst werden. Dadurch kann der Benutzer mehrere Änderungen an der Konfiguration vornehmen, ohne das gleich neu gemessen wird. Sobald die Simulation abgeschlossen ist wird die erreichte Framerate angezeigt. Zusätzlich wird jeder Filter aufgelistet mit folgenden Informationen:

- #: ID des Filters

- **Filter name:** Name des Filters
- **Size:** Derzeitige Grösse des Filterbereichs
- **Expected Delta [ms]:** Dieser Wert gibt den zu erwartenden Geschwindigkeitsgewinn in Milisekunden an, falls der Filter mittels der "Decrease filter size" Schaltfläche verkleinert wird
- **Action:** Die Schaltfläche "Decrease filter size" verkleinert den Filter um 20%
- **Message:** Hier werden Änderungen angezeigt, die automatisch vorgenommen wurden

Bei Composite-Filtern wird automatisch die, für die angegebene Filterkette, performantere Version gewählt.

### 6.2.2 Konzept

Für die Simulation mussten mehrere Entscheidungen getroffen werden betreffend der Verarbeitung. Die Filter könnten, analog zur Testsuite, isoliert ausgeführt werden oder in der gesamten Filterkette. Hier wurde die gesamte Filterkette gewählt, da sich die Filter gegenseitig beeinflussen können. Dies liegt an der parallelen Verarbeitung der C#-Filter. Zusätzlich können Controls auf einen Ausgabewert eines Filters warten. Somit können nicht die einzelnen Verarbeitungszeiten der Filter addiert werden, um die Gesamtzeit zu erhalten. Für den Benutzer ist die Gesamtzeit seiner Filterkette relevanter, als die Zeit, die ein einzelner Filter benötigt.

Weiter gibt es verschiedene Varianten, wie das Frame verarbeitet werden kann. Es können die Methoden *VideoProcessor.ProcessCurrentFrame* oder *GraphPane.ProcessFrame* verwendet werden. Der grobe Unterschied der beiden Methoden ist, dass die Erstere das Frame zuerst noch aus dem Video lädt und dadurch mehr Zeit benötigt. Bei der Verarbeitung eines Streams oder Videos muss das Frame auch zuerst geladen werden, dadurch liegt diese Methode näher an der realen Verarbeitungszeit und wurde gewählt.

### 6.2.3 Implementation

Bei einer Simulation wird zuerst für alle Composite-Filter die Gesamtzeit über die definierte Anzahl Frames gemessen, welche jede Version (C# und Cuda) innerhalb der gesamten Filterkette benötigt. Diejenige, die eine tiefere Gesamtzeit aufweist wird automatisch ausgewählt.

Bei einer Simulation wird zuerst für jeden Composite-Filter die Zeit der gesamten Filterkette über die konfigurierte Anzahl an Frames gemessen. Diese Simulation wird immer mit beiden Varianten C# und Cuda durchgeführt. Danach wird die Gesamtzeit beider Versionen für jeden Composite-Filter verglichen und die Version aktiviert, welche eine kürzere Gesamtverarbeitungszeit aufweist.

Im zweiten Schritt wird die angepasste Filterkette nochmals auf die gewünschte Anzahl Frames angewendet. Jeder Filter wird zuerst mit seiner Originalgrösse ausgeführt, danach um einen konstanten Faktor verkleinert, nochmals ausgeführt und wieder vergrössert. Für beide Grössen wird die Gesamtzeit der ganzen Filterkette zwischengespeichert. Durch die Simulation mit verschiedenen Filtergrössen kann dem Benutzer aufgezeigt werden, was eine Reduktion um 20% eines Filters an Gesamtzeit erspart. Für diesen Wert wird bewusst der Durchschnitt über alle Frames berechnet und nicht der Median, da Ausreisser die Framerate drastisch drücken können durch die serielle Verarbeitung der Frames in *Promon200*.

## Fazit

In dieser Arbeit wurde aufgezeigt, dass mit Hilfe von Cuda bei rechenintensiven Filtern, wie z.B. dem Pattern Matching, ein Geschwindigkeitsvorteil gegenüber der C#-Variante erzielt werden kann. Dies besonders bei grossen Filterbereichen, da dort der Overhead eines Cuda Aufrufes und die Zeit für das Memory kopieren im Vergleich zur Rechenzeit kleiner ist. Es muss daher auch gesagt werden, dass für kleine Filterbereiche keine Geschwindigkeitsvorteile erzielt werden konnten.

Es wurden zwei Klassifizierungfilter mit Cuda entwickelt, welche inkorrekte Zustände in voll-automatisierten Produktionssystemen erkennen können. Da die Konfiguration dieser Filter schwierig und zeitaufwändig ist, wurde zusätzlich ein Werkzeug entwickelt, welches die Filter automatisch anhand von Lernframes (Beispielframes von korrekten und inkorrekten Produktionszuständen) konfiguriert.

Weiter wurde eine Komponente zur Erreichung einer gewünschten Framerate entwickelt. Diese gibt Auskunft über die aktuell erreichbare Framerate und gibt, falls diese unter der gewünschten Framerate liegt, Auskunft über Massnahmen, um die Framerate zu verbessern. Einige Entscheidungen können von der Komponente selbst gefällt werden, andere müssen dem Benutzer überlassen werden.

Bei der Analyse der bestehenden Massnahmen wurde herausgefunden, dass der Pixelstep bei den C#-Filtern die Verarbeitungszeit nahezu auf den theoretischen Wert von  $\frac{1}{n^2}$  reduziert wird. Während die Zeitersparnis bei Cuda-Filtern eher klein ist und bei ca. 15% liegt.

Zukünftig sähen wir Potential in einer Weiterentwicklung des *Promon200*, bei welcher auf der Grafikkarte bereits die Rohdaten der Hochgeschwindigkeitskamera verarbeitet werden. Dadurch wären die Frames schon auf der Grafikkarte und könnten von den Cuda-Filtern direkt verwendet werden ohne Memory kopieren zu müssen. So könnten die Cuda-Filter auch bei kleinen Filtergrössen konkurrenzfähig werden. Auch in der Cuda Version 4 liegt Potential, da sie bessere Unterstützung für Multithreading auf Host Seite bietet. Dies könnte besonders für neuere Grafikkarten interessant sein, da sie das gleichzeitige Verarbeiten von mehreren Kernelaufrufen erlauben. Dadurch könnten Cuda-Filter in *Promon200* weiter an Geschwindigkeit gewinnen.

## 7 Reflexion

Die Reflexion enthält unsere persönlichen Erfahrungen, welche wir während dieser Arbeit gemacht haben. Sie enthält eigene Meinungen und persönliche Wertungen.

### 7.1 Zusammenarbeit

Wir arbeiteten fast immer zusammen an der FH. Dadurch wussten wir immer, was der Andere macht und konnten uns gegenseitig helfen. Die Zusammenarbeit lebte von konstruktiver Kritik und gegenseitigem Ansporn. Trotzdem war die Atmosphäre harmonisch. Dass wir beide noch nie mit Cuda gearbeitet und auch mit Bildverarbeitung nicht vertraut waren, gab uns zusätzliche Motivation. Das Aufgabengebiet wirkte zusätzlich interessant, da wir in der parallelen Ausführung die Zukunft sehen. Auch geschätzt haben wir, dass wir Einblick in mehrere Gebiete erhielten und nicht einen Filter bis zum Optimum ausreizen mussten.

Cuda in Visual Studio 2010 einzurichten gab einige Probleme. Das Problem teilten wir mit vielen und so fanden wir viele Anleitungen im Internet, allerdings funktionierte keine bei uns. Eine Kombination aus einigen Anleitungen brachte schliesslich die Lösung und wurde in einer eigenen Anleitung dokumentiert.

Zusätzlich erschwerend waren grössere Änderungen im Promon200 während unserer Bachelorarbeit. Hier waren wir froh, mit Herrn Schindler, einen hilfsbereiten und direkten Ansprechpartner zu haben.

Da wir beide über keine Bildverarbeitungsgrundlagen verfügen, sind einige Ansätze wohl anders gewählt worden als dies Personen mit breitem Bildverarbeitungswissen getan hätten. Davon liessen wir uns aber nicht abschrecken, sondern arbeiteten umso motivierter weiter.

### 7.2 Betreuung

Mit unserem Betreuer, Martin Schindler, haben wir uns während der Unterrichtszeit alle zwei Wochen, in der unterrichtsfreien Zeit häufiger getroffen. Wir zeigten ihm was wir gemacht haben, wo wir Probleme hatten und wie wir diese gelöst haben oder zu lösen gedenken. Bei diesen Treffen wurde auch immer überprüft, ob wir uns im Zeitplan befinden und bei einer Abweichung nach den Gründen gesucht und Massnahmen definiert. Er war sehr interessiert, gab wichtige Inputs und passte das Promon200 bei Problemen schnell an.

Unser Kunde, Prof. Dr. Christoph Stamm, nahm an einigen unserer Treffen mit Herr Schindler teil und überprüfte den Fortschritt. Er sah den grösseren Zusammenhang und zeigte auf, was wie stark zu gewichten war. Auch wurden neue Ideen präsentiert, von welchen ein Teil umgesetzt werden konnte.

Den Experten, Charles Zehnder, haben wir am Anfang der Arbeit getroffen. Er hat sich dafür eingesetzt, klare Rahmenbedingungen zu schaffen.

## 7.3 Zeitplan

Der Zeitplan konnte ziemlich gut eingehalten werden. Die Implementation der bestehenden Filter in Cuda benötigte mehr Zeit als angenommen. Dies liegt vor allem an den Schwierigkeiten mit dem Prefix Sum Algorithmus. Dieser konnte für den Average Brightness Filter verwendet werden, erreichte aber keine genügende Performance. Da der Prefix Sum Algorithmus auch im Pattern Matching eingesetzt werden sollte, wurde lange versucht, diesen zu verbessern - leider ohne Erfolg. Hier wurde uns zu spät bewusst, dass wir diesen Ansatz aufgeben mussten, was in doppelt so vielen Arbeitsstunden als geplant für diese beiden Filter resultierte (siehe Abbildung 7.1). Der Zeitverzug konnte durch erhöhten Einsatz in der unterrichtsfreien Zeit schnell aufgeholt werden. Auch wurde der Dokumentationsaufwand unterschätzt.

Task#	Beschreibung	Soll Zeit	Ist Zeit	%
1	Anforderungen definieren	8	8	100.00%
2	Zeitplanung	6	6	100.00%
3	Einarbeitung Promon200	8	6	75.00%
4	CUDA einrichten und Wissen erarbeiten	30	39	130.00%
5	Bildverarbeitungswissen erarbeiten	30	11.5	38.33%
6	Videos auswählen gemäss Kriterien	4	4	100.00%
7	Testsuite Konzept	6	4	66.67%
9	Testsuite erstellen	50	47.5	95.00%
11	Average-Brightness CUDA	20	46.5	232.50%
12	Velocity CUDA	45	32	71.11%
13	Pattern-Matching CUDA	45	79	175.56%
14	Analyse der umgesetzten Filter	20	27.5	137.50%
16	Videoauswahl treffen	4	0	0.00%
17	Klassifizierungskriterien ausarbeiten	20	11.5	57.50%
19	Kriterien in Filter umsetzen	100	118.5	118.50%
20	Frequenzbestimmung eines repetitiven Vorgangs	30	29	96.67%
21	Massnahmen definieren	20	36.5	182.50%
22	Konzept erstellen	24	26.5	110.42%
24	Konzept umsetzen	60	60	100.00%
25	Analyse der Massnahmen	20	22.5	112.50%
27	Dokumentation Vorgehensweise	20	27.5	137.50%
28	Dokumentation Konzept	22	39.5	179.55%
29	Dokumentation Implementierung	20	19.5	97.50%
30	Dokumentation Testresultate	20	21	105.00%
31	Dokumentation Kontrolle Zeitplan	8	9.5	118.75%
32	Dokumentation Reflexion	30	10.5	35.00%
33	Präsentation vorbereiten	10	0	0.00%
34	Präsentation halten	2	0	0.00%
35	Projektmanagement	40	21	52.50%

Abbildung 7.1: Auswertung der einzelnen Jobs

# Literaturverzeichnis

- [1] [http://developer.download.nvidia.com/compute/cuda/3\\_1/toolkit/docs/NVIDIA\\_CUDA\\_C\\_ProgrammingGuide\\_3.1.pdf](http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf), Kapitel 5.1
- [2] [http://developer.download.nvidia.com/compute/cuda/3\\_1/toolkit/docs/NVIDIA\\_CUDA\\_C\\_ProgrammingGuide\\_3.1.pdf](http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf), Kapitel 3.3.1
- [3] [http://developer.download.nvidia.com/compute/cuda/1\\_1/Website/projects/scan/doc/scan.pdf](http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf) (12.08.2011)
- [4] [http://www.ee.cuhk.edu.hk/~wlouyang/Papers/Manu\\_OHT.pdf](http://www.ee.cuhk.edu.hk/~wlouyang/Papers/Manu_OHT.pdf) (12.08.2011)
- [5] [http://www.strchr.com/performance\\_measurements\\_with\\_rdtsc](http://www.strchr.com/performance_measurements_with_rdtsc) (12.08.2011)
- [6] <http://khason.net/blog/parallel-programming-well-it%E2%80%99s-all-about-cpu-affinity-or-how-to-set-processor-affinity-in-wpf/> (14.08.2011)
- [7] <http://forums.nvidia.com/index.php?showtopic=158779> (13.08.2011)
- [8] <http://developer.nvidia.com/cuda-toolkit-31-downloads> (18.08.2011)
- [9] <http://developer.nvidia.com/cuda-toolkit-40> (12.08.2011)
- [10] <http://de.wikipedia.org/wiki/CUDA> (18.08.2011)
- [11] [http://de.wikipedia.org/wiki/Thread\\_%28Informatik%29](http://de.wikipedia.org/wiki/Thread_%28Informatik%29) (18.08.2011)
- [12] <http://de.wikipedia.org/wiki/CPU> (18.08.2011)

# Abbildungsverzeichnis

2.1	Das Promon200 mit platzierten Filtern . . . . .	7
2.2	Verbindung um Ausgabebild von Control zu übernehmen . . . . .	8
2.3	Ausschnitt aus der bestehenden Architektur . . . . .	9
4.1	Klassendiagramm vom CudaBaseFilter, von welchem alle Cuda-Filter erben . .	18
4.2	Geschwindigkeitsvergleich der verschiedenen Invert Filter . . . . .	19
4.3	Exklusiver Prefix Scan mit Addition . . . . .	21
4.4	Reduktionsphase des Prefix Scans [3] . . . . .	22
4.5	Zweiter Phase des Prefix Scans [3] . . . . .	23
4.6	Geschwindigkeitsvergleich der verschiedenen Average Brightness Filter . . . .	25
4.7	Geschwindigkeitsvergleich der verschiedenen Velocity Filter . . . . .	27
4.8	Operationen der seriellen und parallelen C#-Version des Pattern Matchings . .	28
4.9	Die verschiedenen Operationen des Cuda Pattern Matchings . . . . .	29
4.10	Geschwindigkeitsvergleich der verschiedenen Pattern Matching Filter . . . . .	30
4.11	UML-Diagramm vom <i>CompositeVelocity</i> und seinen Basisklassen . . . . .	33
4.12	Composite Velocity Filter mit Checkbox zum Wechseln der Version . . . . .	34
4.13	XML eines gespeicherten Composite Filters . . . . .	35
5.1	Analysezeitpunkt bestimmen mittels Grenzwert . . . . .	37
5.2	Unschöne Kantenübergänge in den Filterausgabewerten . . . . .	38
5.3	Output von False-True-Trigger . . . . .	38
5.4	Frequenz Control, Anzahl verwendeter Frames ist konfigurierbar . . . . .	38
5.5	Fehler bei der Waverbeschichtungsanlage durch Tröpfchenbildung . . . . .	39
5.6	Anlage in Betrieb - Strahl OK . . . . .	40
5.7	Anlage gestartet - Unterbruch aber kein Fehler . . . . .	41
5.8	Anlage in Betrieb - Fehler wegen den Tröpfchen . . . . .	41
5.9	Speicherverwaltung bei einem Grenzwert von 8 . . . . .	42
5.10	Performanzvergleich der Waverbeschichtungsfilter . . . . .	42
5.11	Deckel links korrekt, Deckel rechts fehlerhaft . . . . .	43
5.12	Spaltenweise Suche nach der Kante (erstes schwarzes Pixel) . . . . .	44
5.13	Links: binarisiertes Template; Rechts: Kantenvergleich . . . . .	44
5.14	Links: Originalbild von defektem Deckel; Rechts: Kantenvergleich . . . . .	45
5.15	Geschwindigkeitsvergleich bei verschiedenen Filtergrößen . . . . .	45
5.16	Werkzeug zum automatischen Konfigurieren eines Filters . . . . .	48
6.1	Geschwindigkeitsvergleich zwischen C#- und Cuda-Filtern mit Pixelstep . . . .	50
6.2	Pixelstep Geschwindigkeitsvergleich zwischen C#- und Cuda-Filtern . . . . .	51
6.3	Ausgabewerte des Average-Brightness Filters bei verschiedenen Pixelsteps und einer Größe von 600x400 Pixeln . . . . .	52
6.4	Framerate der Filterkette zur Erkennung falscher Deckel bei verschiedenen Framesteps . . . . .	53
6.5	Auswertung der ausgelösten Fehlerereignisse bei verschiedenen Framesteps . . . .	53
6.6	Speed Check Tab, welches zur Optimierung der Filterkette dient . . . . .	54

7.1 Auswertung der einzelnen Jobs . . . . . 58

# Tabellenverzeichnis

3.1	Ausgaben der Testsuite pro Video . . . . .	14
3.2	Ausgaben der Testsuite pro Filter . . . . .	14
3.3	Informationen die bei jeder Ausführung der Testsuite abgespeichert werden . .	15
7.1	Dateien und Projekte die im Rahmen dieser Arbeit erstellt und bearbeitet wurden	71

# Listings

3.1	Genauigkeit von Stopwatch . . . . .	11
3.2	Configuration . . . . .	13
4.1	Neue Methoden in der Klasse PromonBaseFilter.cs . . . . .	17
5.1	Methoden des IAutoConfigFilter Interfaces . . . . .	48

# Anhang

## Glossar

BEGRIFF	BESCHREIBUNG
CUDA	Technologie von NVIDIA, welche erlaubt Programmteile zu entwickeln, die durch den Grafikprozessor (GPU) auf der Grafikkarte abgearbeitet werden.[10]
Device	Wird hier verwendet als Synonym für Grafikkarte.
Host	Wird hier verwendet als für PC in welchem die Grafikkarte steckt.
RAM	Schneller, flüchtiger Zwischenspeicher.
GRAM	RAM auf der Grafikkarte.
Thread	Sequentieller Abarbeitungslauf eines Prozesses.[11]
Prozessor	Zentrale Verarbeitungseinheit (ZVE) eines Computers.[12]
CPU	Synonym für Prozessor.
GPU	Prozessor auf der Grafikkarte, typischerweise mit vielen tief getaktete Kernen.
Prozessor Tick	Kleinste Zeiteinheit, welche vom Prozessor erkannt wird. Ein 3 GHz Prozessor macht 3 Milliarden Ticks pro Sekunde.
Hyper Kern	Technologie von Intel zur effizienteren Nutzung, bei welcher pro physikalischem Prozessorkern mehrere logische Prozessorkerne vorgegaukelt werden.
Promon200	Software zum Erkennen von Fehlern in vollautomatisierten Produktionsabläufen.
Filter	Komponente in Promon200 zur Bildanalyse oder Bildverarbeitung.
Control	Komponente in Promon200 zur Weiterverarbeitung von Ausgabewerten von Filtern oder Controls.
Filterkonfiguration	Eingabewerte eines Promon200 Filters.
Filterkette	Ein oder mehrere Filter mit Positionen, Verbindungen und Filterkonfigurationen.
Frame	Aktuelles Bild einer Kamera oder eines Videos.
stride	Anzahl Bytes die eine Zeile eines Frames benötigt.
Eingabebild	Bild, welches von einem Filter als Eingabewert benötigt wird.

## Ehrlichkeitserklärung

### Promon200

#### Bachelorarbeit von Daniel Suter und Claudio Wettstein Frühlingssemester 2011

Hiermit erklären wir, dass diese Arbeit selbständig erarbeitet wurde und keine anderen Quellen oder Hilfsmittel verwendet wurden als jene, die angegeben sind.

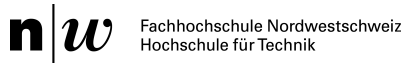
Ort: Datum: Daniel Suter:

---

Ort: Datum: Claudio Wettstein:

---

## Originalauftrag



Brugg-Windisch, 17. Februar 2011

# Informatik Projektarbeit P6

## Promon200

Auftraggeber: IMVS

Ausgangslage: Hochgeschwindigkeitskameras werden oft zur Überwachung sehr schneller, vollautomatisierter, industrieller Abläufe eingesetzt. Im Zuge der Marktanforderung, dass die einmal aufgenommenen Prozesse immer schneller zu analysieren sind, soll das aufgenommene Bildmaterial nicht nur zur Dokumentation des Ablaufes dienen, sondern soll vielmehr für Echtzeit-Bildverarbeitung direkt verwendet werden. Dabei werden die Bilddaten an einen Rechner übertragen, auf diesem in Echtzeit (z.B. 200 Bilder pro Sekunde) mit neu zu entwickelnden, parallelen, adaptiven Algorithmen analysiert und bei Bedarf Ereignisse ausgelöst, welche den Automationsprozess steuern.

Ziel der Aufgabe: Das Ziel dieser Aufgabe ist die generelle Einarbeitung in CUDA und die Auseinandersetzung mit der parallelen Bildverarbeitung auf der Grafikkarte. Die bestehende SW (Promon200) soll in Form eigenständig entwickelter Komponenten weiterentwickelt werden.

Problemstellung: Die bestehende SW nutzt die parallele Verarbeitung bereits teilweise. Weitere Komponenten sollen für eine parallele Verarbeitung umgebaut werden.

1) CUDA: Parallele Arbeiten können auch direkt auf der GPU (Grafikkarte) ausgeführt werden. Bestehende und neue Bildverarbeitungsschritte sollen mittels CUDA auf der GPU ausgeführt werden.

2) Entscheidungskomponente: Die SW soll in der Lage sein, die vorgegebenen Verarbeitungsschritte auch dann in Echtzeit durchzuführen, wenn die HW, den Anforderungen nicht ganz gerecht wird. In einem solchen Fall müssen automatisch Entscheidungen getroffen werden, welche Verarbeitungsschritte vereinfacht und damit beschleunigt werden.

Technologien/Fachliche Schwerpunkte/Referenzen

- GPU Programmierung (CUDA)
- C# und WPF
- Bild- und Videoverarbeitung

Betreuer: Martin Schindler, Tel.: 056 462 48 88, [martin.schindler@fhnw.ch](mailto:martin.schindler@fhnw.ch)

Studierende: Daniel Suter, [daniel.suter@students.fhnw.ch](mailto:daniel.suter@students.fhnw.ch)  
Claudio Wettstein, [claudio.wettstein@students.fhnw.ch](mailto:claudio.wettstein@students.fhnw.ch)

Projektdauer: 21.2.2011 bis 20.8.2011

# Aufgabenstellung

## 1 Einarbeitung

### 1.1 Vorbereitung

Fixieren Sie für die Unterrichtszeit alle zwei Wochen und für die unterrichtsfreie Zeit einen wöchentlichen Besprechungstermin mit Ihrem Betreuer. Nutzen Sie diese Besprechungen, um Ideen, Vorschläge und Arbeiten zu präsentieren und zu diskutieren.

Bevor Sie mit der Arbeit richtig loslegen, erstellen Sie einen möglichst detaillierten Zeitplan für die Dauer dieser Projektarbeit. Definieren Sie die wichtigsten Meilensteine.

### 1.2 Infrastruktur und Programmierumgebung

Richten Sie sich eine Arbeitsstation zur Software-Entwicklung ein. Verwenden Sie Visual Studio 2010. Als Programmiersprache wird C# mit WPF verwendet. Für eine produktive Arbeitsweise unter Windows ist ein direkter Zugang zu der MSDN-Library unerlässlich.

### 1.3 Literatur- und Softwarestudium

Als Einstieg in CUDA empfehlen wir Ihnen die entsprechenden Seiten von Nvidia [CUDA]. Suchen Sie sich weitere Literatur (Foren, Fachbücher, englischsprachige Original-Papers, Webseiten usw.) und bibliographieren Sie alle wichtigen Quellen.

Als praktischen Einstieg in die Thematik empfehlen wir Ihnen das Nachvollziehen vorhandener CUDA-Beispiele, (z.B. unser Performancevergleich Projekt [Perf]) und die Entwicklung eigener einfacher Programme.

### 1.4 Testset

Damit Sie die Güte Ihrer laufenden Arbeit ständig selber beurteilen können, benötigen Sie Algorithmen und Datensätze. Wählen Sie dazu ein paar geeignete Testvideos aus und definieren Sie die Testumgebung. Implementieren Sie Algorithmen, die sich für den Performance-Vergleich zwischen CPU und GPU gut eignen. Besprechen Sie Ihr Testset (auf Papier) mit Ihrem Betreuer.

## 2 Fragestellungen

Gehen Sie den folgenden Fragestellungen detailliert nach. Arbeiten Sie diese theoretisch aus und setzen Sie sie nach Absprache mit dem Betreuer praktisch um.

### 2.1 Parallelisieren mit CUDA

Wie gut lassen sich die bereits auf der CPU implementierten Promon200 Bildverarbeitungsalgorithmen (Average-Brightness, Velocity und Pattern-Matching) auf der GPU beschleunigen? Wie viele Frames lassen sich parallel verarbeiten?

#### 2.1.1 Ziele

Sie haben die Algorithmen auf der GPU implementiert und die Effizienz derselben auf Ihrem Testset gemessen und analysiert. Unterscheiden Sie zwischen sequentiellen und parallelen Algorithmen und zwischen CPU und GPU Verarbeitung.

## 2.2 Video-Analyse

Anhand vorgegebener Videos sollen Ideen zur Klassifizierung des gefilmten, repetitiven Vorgangs (gut/schlecht) ausgearbeitet und umgesetzt werden. Zudem soll die Frequenz eines sich wiederholenden Vorgangs Subframegenau bestimmt werden.

### 2.2.1 Ziele

Sie haben ein Konzept ausgearbeitet und Teile davon nach Absprache mit dem Betreuer (auf der GPU) umgesetzt.

## 2.3 Entscheidungskomponente

Studieren Sie das Grundkonzept der Entscheidungskomponente in Promon200. Welche Massnahmen zur Einhaltung der Framerate lassen sich ohne oder mit Benutzerinteraktion umsetzen und wie soll die Benutzerinteraktion aussehen? Erstellen Sie ein konkretes Konzept für eine Entscheidungskomponente und setzen Sie dieses nach Absprache mit Ihrem Betreuer um.

### 2.3.1 Ziele

Analysieren Sie Ihre Entscheidungskomponente im praktischen Ablauf und zeigen Sie die Auswirkungen der einzelnen Optimierungsmöglichkeiten in Bezug auf die Verarbeitungszeit aber auch auf das Resultat der ganzen Bildverarbeitungskette.

## 3 Dokumentation

### 3.1 Schriftliche Dokumentation

Dokumentieren Sie schriftlich und elektronisch Ihre Vorgehensweise, Ihre Konzepte, die wichtigsten Punkte der Implementierungen und Ihre Testresultate. Überprüfen Sie auch den geplanten mit dem tatsächlichen Zeitplan und reflektieren Sie Ihre persönlichen Erlebnisse und Erfahrungen während dieser Arbeit.

Achten Sie unbedingt darauf, dass Sie persönliche Kommentare von Fakten strikte trennen. **Ein erster Teil der Dokumentation ist vollständig faktenbasiert.** Das bedeutet, dass keine Sätze der Art „Dann hatten wir das Problem x und versuchten es mit y zu lösen.“ auftreten dürfen. Falls ein solches Problem x aber wirklich existiert und nicht nur Sie damit nicht gleich zu Rande kamen, dann sollen Sie schreiben: „Tests z haben klar gezeigt, dass ein Problem x besteht. Mögliche Ansätze, um das Problem x zu lösen, sind a, b und c. Wir haben uns aus den Gründen e und f für Variante c entschieden.“ Erst in einem zweiten Teil sollen Sie Ihre persönlichen Eindrücke, Erlebnisse, Probleme und dergleichen formulieren.

Wichtig ist auch, dass eine gute Dokumentation auch noch nach vielen Jahren gelesen werden können muss und dass sie dem Leser ein gut abgerundetes Bild vermittelt, auch dann wenn er nicht direkt an der Arbeit beteiligt war. Bitte legen Sie auch grossen Wert auf sprachliche Qualität.

Das Zielpublikum dieser Dokumentation sind die Betreuer und zukünftige Studierende, welche in diesem Bereich weiterarbeiten wollen.

### 3.2 Präsentation

In Absprache mit Ihrem Betreuer und dem Auftragsteller der Arbeit wird von Ihnen eine Präsentation Ihrer Arbeit erwartet. Die Präsentation soll einerseits einen groben Überblick verschaffen und andererseits ein oder zwei wichtige und interessante Details hervorheben. Bei den Zuhörern dürfen Sie von einem technisch versierten Fachpublikum ausgehen.

Zusätzlich zur Präsentation wird eine prägnante Demonstration der Benutzung Ihrer Software erwartet.

## **4 Rechtliche Hinweise**

### **4.1 Vereinbarung über Urheberrechte an Computerprogrammen**

#### **4.1.1 Gegenstand dieser Vereinbarung**

Regelung der Urheberrechte an Computerprogrammen, die im Rahmen der Ausbildung aufgrund einer Aufgabenstellung seitens der FHNW oder eines externen Aufgabenstellers durch Studierende ohne Bezahlung geschrieben werden, auf der Basis des Urheberrechtsgesetzes vom 9. Oktober 1992.

#### **4.1.2 Verwendungs- und Änderungsbefugnisse**

Neu und von den Studierenden erstellte Computerprogramme aus der oben genannten Arbeit dürfen nach deren Abschluss sowohl seitens der Aufgabensteller wie auch von den beteiligten und persönlich genannten Studierenden beliebig und vergütungsfrei verwendet werden; sie dürfen auch verändert werden.

#### **4.1.3 Autorenhinweise**

Werden grössere Programmmodule dieser Arbeit weitgehend unverändert verwendet, so soll auf die Studierenden sowie auf die Aufgabensteller hingewiesen werden.

#### **4.1.4 Urheberrechte an Texten**

Die Urheberrechte an den zugehörigen und von den Studierenden verfassten Berichten und Dokumentationen stehen den Studierenden zu; die Aufgabensteller können diese für eigene Zwecke und in Bibliotheken vergütungsfrei verwenden.

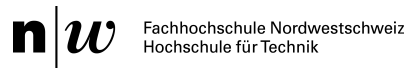
### **4.2 Schlussbestimmung**

Die Unterzeichneten anerkennen, den Text gelesen und verstanden zu haben und verpflichten sich mit Ihrer Unterschrift die aufgeführten Punkte und die allgemeine Sorgfaltspflicht einzuhalten.

## **5 Literaturhinweise**

[CUDA] GPU Programmierung mit CUDA. <http://developer.nvidia.com/object/gpucomputing.html>

[Perf] Performancevergleich c#, c++, Cilk, OpenMP, OpenCL, CUDA.



**Brugg-Windisch, den** .....

**FHNW, Informatik**

Martin Schindler .....

**Studierende**

Daniel Suter .....

Claudio Wettstein .....

## Erstellte und bearbeitete Dateien

Im Rahmen dieser Arbeit wurden Änderungen an den in der Tabelle 7.1 aufgelisteten Projekten und Dateien gemacht.

PROJEKT	DATEI	ÄNDERUNG
CudaFilter	*	Ganzes Projekt erstellt
Filter	*	Ganzes Projekt erstellt
TestSuite	*	Ganzes Projekt erstellt
Promon200WPF	ConfigPane	Erstellt
Promon200WPF	FalseTrueTriggerControl	Erstellt
Promon200WPF	FilterInputControlCheckbox	Erstellt
Promon200WPF	FrequenceControl	Erstellt
Promon200WPF	GraphPane.xaml.cs	Integration von Cuda-Filtern in einer separaten Liste zur seriellen Verarbeitung
Promon200WPF	ToolBoxPane	Integration neuer Controls
Promon200WPF	TrainingPane	View und ViewModel erstellt
Promon200WPF	CudaHelper	Neu erstellt
Promon200WPF	MainWindow	Integration neuer Tabs (SpeedCheck, AutoConfig) und letzter <i>TearDown</i> Aufruf
Promon200WPF	AutoConfig	Neu erstellt
Promon200WPF	CompositeFilter	Neu erstellt
Promon200WPF	CudaBaseFilter	Neu erstellt
Promon200WPF	PromonBaseFilter	Erweiterung um <i>Setup</i> und <i>TearDown</i> Methoden

Tabelle 7.1: Dateien und Projekte die im Rahmen dieser Arbeit erstellt und bearbeitet wurden