Technical Report

Pattern Recognition and Image Processing Group
Institute of Computer Aided Automation
Vienna University of Technology
Favoritenstr. 9/1832
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801-18351
Fax: +43 (1) 58801-18392
E-mail: michael.rauter@chello.at
URL: http://www.prip.tuwien.ac.at/

PRIP-TR-116 7. Oktober 2007

Effizientes Object Tracking durch Programmierung von Mehrkernprozessoren und Grafikkarten

Michael Rauter

Abstract

In dieser Masterarbeit wird ein Object Tracking System realisiert, das bei Video-Material mit Auflösungen von bis zu 768 Œ 576 Bildpunkten bei 3 Farbkanälen Echtzeitfähigkeit erreichen soll (Def.: das Verarbeiten von 25 Bildern pro Sekunde - in 40 Millisekunden müssen alle Algorithmen fertig berechnet werden). Dabei werden state-of-the-art Algorithmen verwendet, die auf CPUs mit nur einem Rechenkern nur bei geringeren Video-Auflösungen Echtzeitfähigkeit erreichen. Ziel ist die Performance-Steigerung eines Object Tracking Systems. Die Motivation für eine Performance-Steigerung rührt daher, dass sich, je weniger Zeit für das Berechnen von Verarbeitungsschritten eines Object Tracking Systems benötigt wird, desto höhere Video-Auflösungen und -Frameraten handhaben lassen (resultiert in besseren Tracking-Ergebnissen), und es bleibt mehr Zeit für weitere Verarbeitungsschritte (z.B. höherentwickelte Data Association, Behaviour Detection). Die Berechnungsschritte, die im entwickelten System durchgeführt werden und als Gesamtpaket in Echtzeit laufen sollen, sind Datenerfassung, Bewegungserkennung samt Schatten- und Reflexionserkennung inklusive deren Entfernung, Connected Components Analysis, Verwalten der zu trackenden Objekte (Initialisierung, Tracking, Data Association, Löschen) und die grafische Ausgabe. Die Echtzeitfähigkeit wird dadurch erreicht, dass einerseits Algorithmen bzw. Algorithmusschritte, die parallelisierbar sind, auf die Grafikkarte ausgelagert werden und andererseits durch Multi-Threading Mehrkernprozessoren ausgereizt werden, indem verschiedene Phasen der Verarbeitungskette des Systems auf alle vorhandenen CPU-Kerne aufgeteilt und so Berechnungen simultan auf verschiedenen Rechenkernen ausgeführt werden (Datenerfassung, Bewegungserkennung, Object Tracking, Visualisierung). Mit diesen Konzepten erreicht das implementierte Object Tracking System eine Performancesteigerung um mehr als den Faktor 9 im Vergleich zu einer optimierten Single-Core CPU-Variante.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

Object Tracking[1] Algorithmen sind Gegenstand aktueller Forschung und gewinnen zunehmend an Bedeutung als Bestandteil von automatisierten Überwachungssystemen. Aufgrund der inhärenten Echtzeitanforderung in solchen Systemen stellen sie enorme Anforderungen an die Rechenleistung moderner Prozessoren. Viele der in der wissenschaftlichen Literatur vorgestellten Algorithmen erfüllen die Echtzeitanforderung nicht bzw. nur unter äußerst groben Einschränkungen (geringere Bildauflösungen oder Bildwiederholungsraten).
Ziel dieser Arbeit ist die Implementierung eines *Object Tracking* Systems, das modernste Algorithmen (siehe Abschnitt 6.3) und Echtzeitfähigkeit vereint. Unter Echtzeitfähigkeit ist in dieser Masterarbeit das Verarbeiten von PAL Video Material (768×576 bzw. 720×576 Bildpunkte á 3 Farbkanälen bei einer Wiedergabefrequenz von 25 Hertz) gemeint. Die Verarbeitungsschritte sind Datenerfassung, Bewegungserkennung samt Schatten- und Glanzlichterkennung inklusive deren Entfernung, Erkennen zusammenhängender Pixelregionen im Bild (*Connected Components Analysis*), *Tracking* der Regionen, *Data Association*, Verwalten der getrackten Objekte[2] inklusive Initialisierung neuer Kandidaten und Löschen verwaister Objekte und letztendlich auch die Ausgabe in grafischer Form.

Zum Erreichen dieser Anforderung werden zwei wesentliche Mechanismen verwendet: Erstens sollen zeitintensive Berechnungen auf die Grafikkarte ausgelagert werden, um ihre pa-rallelen Fähigkeiten für entsprechend geeignete Verarbeitungsschritte zu nutzen. Seit der Vertex- und Fragment-Prozessor von Grafikkarten durch Shaderprogramme frei

[1] Unter *Object Tracking* Algorithmen versteht man Algorithmen, die das Verfolgen von Objekten in Bildfolgen ermöglichen und so den Weg, den ein Objekt genommen hat, erfassen können.

[2] Als ein zu trackendes Objekt wird in dieser Arbeit eine rechteckige Pixelregion verstanden, die durch Bewegung eines realen Objekts (z.B. Person, Fahrzeug), aufgenommen durch die Videokamera, im Videodatenstrom segmentiert und extrahiert werden kann.

programmierbar ist, eignen sich diese auch dafür, andere Berechnungen als das Rendern grafischer Szenen zu übernehmen (siehe Abschnitt 2). Die Unterstützung der Programmierung der Grafikkarten und deren Weiterentwicklung gipfelt mit der Entwicklung von CUDA (*Compute Unified Device Architecture*), der neuen Architektur für Berechnungen auf *nVidia GeForce 8* Grafikkarten, in einem neuen Höhepunkt. Grafikkarten bieten sich insbesondere deshalb an, parallelisierbare Algorithmusschritte auszuführen, weil sie von ihren Entwicklern kontinuierlich mit mehr Shadereinheiten (das sind die Recheneinheiten) ausgestattet wurden, mit denen es möglich ist, Berechnungen parallel auf verschiedenen Speicherelementen durchführen zu lassen. Somit lässt sich ein deutlich höherer Berechnungsdurchsatz als auf Prozessoren mit einem Rechenkern erreichen (siehe dazu Abschnitt 8), sofern die zu erledigenden Aufgaben für eine parallelisierte Umsetzung auf der Grafikkarte geeignet sind. Auch die Preisfrage spielt hier eine Rolle: unter den eben genannten Bedingungen erhält man einen Mehrkern-Prozessor mit momentan maximal bis zu 32-mal mehr Rechenkernen (128 *Unified Shader* einer *nVidia GeForce 8800 Ultra* verglichen mit 4 Kernen eines *Quad-Core* Prozessors[3]) und das alles unter Berücksichtigung des Preises - eine *nVidia GeForce 7950 GT* mit 8 Vertex-Shader-Einheiten und 24 Fragment-Shader-Einheiten ist für unter 200 € verfügbar[4] bzw. wird in neuen Komplettsystemen ohnehin bereits in vielen Fällen eine entsprechend geeignete Grafikkarte eingebaut. Weiters kann davon ausgegangen werden, dass zukünftige Grafikkartengenerationen mit entsprechender Leistungssteigerung einhergehen und somit Berechnungen, die auf der Grafikkarte ausgeführt werden, schneller durchgeführt werden (siehe [OLG+05]). Auch zeigt die Erfahrung, dass mit jeder Grafikkartengeneration die Leistungsfähigkeit von Grafikkarten in Bezug auf Flexibilität und Einsatzbereich steigt, da Grafikkarten zunehmend leistungsfähiger bezüglich umsetzbarer Operationen und Funktionalitäten werden und Restriktionen, die noch bei Grafikkarten wie der *NVidia GeForce 7* gültig sind, fallen könnten (siehe [OLG+05] für genauere Informationen).
Einer der Schwerpunkte dieser Arbeit ist die Darlegung der Umsetzung eines Teils eines typischen *Tracking* Systems auf der Grafikkarte unter Berücksichtigung der damit einhergehenden Vor- und Nachteile bzw. Einschränkungen.
Zweitens soll mittels *Multi-Threading* ein weiterer Geschwindigkeitsschub erreicht werden. Durch *Multi-Threaded*-Programmierung kann man alle Kerne eines Mehrkernprozessors nutzen, um sie für die Berechnung der benötigten Algorithmen heranzuziehen [Sta05]. Aktuelle Systeme besitzen bereits häufig einen Mehrkernprozessor, und gerade die zukünftige Entwicklung bei Prozessoren deutet auf eine Erhöhung der Anzahl an Prozessorkernen hin,

[3]Bei diesem Vergleich sind die Einschränkungen von Shaderprogrammen auf Grafikhardware, die durch die Hardware-Architektur bedingt sind und auf Standard-Prozessoren nicht gegeben sind, zu beachten.
[4]Stand Juni 2007

da die Prozessorhersteller in letzter Zeit feststellen mussten, dass die Geschwindigkeitssteigerung bei Prozessoren durch reine Takterhöhung des Rechenkernes nicht dauerhaft fortführbar ist. Durch Erhöhung der Anzahl an vorhandenen Prozessorkernen lässt sich aber auch langfristig eine steigende Prozessorleistung erreichen [Bod06].

Das in dieser Arbeit vorgestellte *Tracking* System erfüllt auf der verwendeten Test-Hardware (siehe Abschnitt 8.4) die an das System gestellten Anforderungen, und es bleibt sogar noch Rechenkapazität ungenutzt, was insbesondere deshalb interessant ist, da so noch weitere Rechenzeit für andere Aufgaben zur Verfügung steht. Beispielsweise könnte der in dieser Arbeit implementierte *Data Association* Algorithmus erweitert werden, um Problemfälle, bei denen der implementierte *Tracker* versagt (z.B. Verdeckungsprobleme etc.), zu beheben oder um etwa zusätzlich einen *Behavior Detection* Algorithmus [JH95, Bux03] zur Überwachung etwaiger ungewöhnlicher Verhaltensmuster zu observierender Personen berechnen zu lassen.

Es folgt ein kurzer Überblick über die Kapitel dieser Masterarbeit:
Kapitel 1 stellt die Einleitung der Masterarbeit dar. In Kapitel 2 wird ein Überblick über wissenschaftliche Arbeiten zum Thema gegeben. In Kapitel 3 werden die Grundlagen der Grafikkarten-Programmierung sowohl hinsichtlich der Hardware als auch hinsichtlich der verfügbaren Programmierschnittstellen behandelt. In Kapitel 4 folgt eine Erläuterung der Konzepte der Grafik-karten-Programmierung, in welchem die notwendigen Voraussetzungen, Regeln und Probleme der Programmierung von Grafikkarten behandelt werden. In Kapitel 5 wird die *Multi-Threading*-Programmierung besprochen. In Kapitel 6 werden Definitionen für das bei dieser Masterarbeit entstandene *Object Tracking* System gegeben und Rahmenbedingungen festgelegt. Kapitel 7 behandelt Implementierungsdetails des *Tracking* Systems. Die Ergebnisse der Arbeit werden in Kapitel 8 zusammengefasst und präsentiert. Schließlich folgt in Kapitel 9 eine Schlussbetrachtung.

Kapitel 2

Verwandte wissenschaftliche Arbeiten

In diesem Kapitel werden wissenschaftliche Arbeiten vorgestellt, die sich mit der Auslagerung von Algorithmen auf die Grafikkarte sowie mit dem Thema *Multi-Threading* in Bezug auf Performancesteigerung beschäftigen.
Die ausgewählten Arbeiten zum Thema der GPGPU (*General Purpose Computation on Graphics Processing Units*) werden in verschiedene Themengebiete unterteilt, in welche sich die wissenschaftlichen Arbeiten einteilen lassen. Es existieren wissenschaftliche Arbeiten, welche die Verwendung von Grafikkarten zur Beschleunigung von rechenintensiven Verarbeitungsvorgängen zum Thema haben, in den Bereichen physikalische Simulationen, Signal- und Bildverarbeitung, *Computer Vision*, *Offline Rendering*, algorithmische Geometrie oder Datenbanken.

Nutzung von Grafikkarten zur Beschleunigung von Berechnungen

Die Idee, Grafikkarten zur Beschleunigung von rechenintensiven Algorithmen zu verwenden, kam bereits auf, bevor Grafikkarten durch Vertex- und Pixelshader individuell programmierbar waren (z.B. [RS01]).
Für die ersten wissenschaftlichen Abhandlungen wurden *Framebuffer-Blending*-Operationen mittels Grafik-API-Aufrufen verwendet. Später kamen sogenannte *Register Combiners* zum Einsatz, und schließlich wurde die Grafikhardware mit programmierbaren Shadereinheiten bis auf einige Einschränkungen (siehe Kapitel 3 und Kapitel 4) frei programmierbar.

Physikalische Simulationen

Hier sollen stellvertretend für alle Arbeiten, die physikalische Simulationen unter Zuhilfenahme von Grafikhardware zum Thema haben, einige wichtige Arbeiten vorgestellt werden. In [LWK03] wird mit Hilfe der *Lattice Boltzmann*-Methode die physikalische Bewegung von Flüssigkeitspartikeln simuliert und es werden Register Combiners dazu verwendet, um Addition, Subtraktion und Multiplikation auf der Grafikkarte zu bewerkstelligen. Divisionen und andere komplexe Operationen werden durch vorberechnete Lookup-Tables realisiert. Nach Darstellung dieser Autoren wird eine bis zu 50-mal schnellere Ausführungsgeschwindigkeit erreicht, als dies durch die Verwendung der CPU möglich ist; allerdings sei angemerkt, dass bei dieser Arbeit das Ergebnis lediglich auf der Grafikkarte gerendert wurde und somit ein Rücktransfer der Ergebnisdaten entfallen konnte. Eine neuere Arbeit um dieselbe Forschergruppe, die bereits Fragment-Shader zur Berechnung der Simulation verwendet, findet sich in [LFWK05]. Auch hier wird mittels der *Lattice Boltzmann*-Methode die Bewegung der Flüssigkeitspartikel simuliert. Es wird ein Performancegewinn gegenüber einer CPU-Variante um den Faktor acht bis fünfzehn angegeben. Die neuesten diesbezügliche Publikation der Forschergruppe zum Thema findet sich in [ZQFK07] und [YWC07], wo ein Performancegewinn von zehn bis elf zu aktuellen CPUs angegeben wird. In [HBSL03] wird eine interaktive Simulation zur Berechnung von Wolken vorgestellt. In dieser Simulation werden partielle Differentialgleichungen verwendet, um die Bewegung von Flüssigkeitsteilchen der Wolken zu beschreiben oder alle weiteren Einflüsse auf die Wolken modellieren zu können, etwa thermodynamische Prozesse. Zur Berechnung auf der Grafikkarte kommen Fragment-Shader zum Einsatz. Als Datenstruktur werden aus Gründen verbesserter Performance an Stelle von echten 3D-Texturen Schichten von 2D-Texturen verwendet.

Signal-, Bildverarbeitung und *Computer Vision*

Aus dem mittlerweile breiten Archiv wissenschaftlicher Arbeiten, die die Grafikkarten für die Beschleunigung von Signalverarbeitungs-, Bildverarbeitungs- oder *Computer Vision*-Aufgaben einsetzen, sollen in diesem Abschnitt einige wichtige Arbeiten stellvertretend ausgewählt und präsentiert werden. In [GLG05] wird der erreichbare Geschwindigkeitsgewinn durch Verwendung von Grafikkarten für Bildverarbeitungsaufgaben untersucht und mit einer SSE-optimierten CPU-Variante verglichen. Es werden mehrere Bildverarbeitungs-Operationen untersucht, so etwa Bild-invertierung, Helligkeits- und Kontrasterhöhung, Farbraumumrechnung vom RGB-Farbraum in den XYZ-Farbraum, Faltungsoperationen wie Laplace-Filterung, Anwendung eines 3x3 Mean-Filters, 3x3 Gauss-

Filterung, Median-Filter oder Gradientenfilter, Thresholding und die morphologischen Operationen Dilation und Erosion. Angeführt wird auch die Anzahl der in den verwendeten Shaderprogrammen benötigten Textur-Lookups und arithmetischen Operationen. Verglichen werden sowohl verschiedene Implementierungen für die Grafikkarte untereinander (sowohl DirectX als auch OpenGL unter verschiedenen Fragmentprofilen) als auch im Vergleich zur CPU-Variante. Es werden für alle untersuchten Bildverarbeitungs-Operationen der Geschwindigkeitsgewinn für ATI- bzw. nVidia-Karten angegeben. Je nach Operation fallen die Ergebnisse unterschiedlich aus, bei pixelweise arbeitenden Operationen fällt der Zugewinn höher aus als bei Faltungsoperationen, welche wiederum einen stärkeren Zugewinn als Vergleichsoperationen aufweisen. Allerdings sollte beachtet werden, dass die Ergebnisse nicht mehr dem aktuellen Stand der Dinge in Sachen Grafikkarten-Programmierung entsprechen, da beispielsweise die in der Arbeit genannten Ausführungszeiten für Vergleichsoperationen auf Grafikhardware ab der *nVidia GeForce 6* und Shadermodell 3.0 dank dynamischer Branching-Funktionalität nicht länger gültig sind. Es zeigt sich deutlich ein inhärentes Problem hinsichtlich Dauer der Gültigkeit von Aussagen bezüglich Ausführungsgeschwindigkeiten und Einschränkungen von Grafikkarten, da mit jeder neuen Grafikkartengeneration diese Aussagen ungültig werden können und dies auch mit gewisser Wahrscheinlichkeit tun, wie die Entwicklung über die letzten Jahre gezeigt hat. Nichtsdestotrotz bleibt die Kernaussage der Arbeit gültig, die besagt, dass durch die Verwendung von Grafikhardware viele Bildverarbeitungsoperationen beschleunigt berechnet werden können.

In [RS01] wird Konturesegmentierung durch Level Sets auf der Grafikkarte mit Hilfe von *Frame-buffer-Blending* realisiert. Dabei kompromittiert auch die geringere Rechengenauigkeit von Grafikkarten, die damals noch keine *floating-point*-Genauigkeit boten, die Ergebnisse nicht. Für ein 128 mal 128 Pixel großes Bild benötigt die vorgestellte Konturesegmentierung 2 Millisekunden auf einem *nVidia GeForce 2 Ultra* Grafikchip.

Es folgen Arbeiten, in denen Frameworks für die Bildverarbeitung auf der Grafikkarte entstanden. So wird etwa in [Jar04] ein einfaches Framework vorgestellt, das einen Gauss-Filter und einen Nachfilter zur Aufbereitung von Nachtszenen bereitstellt. Die Ergebnisse von Filter-Operationen können wiederum als Input für andere Filter-Operationen dienen. So wird repräsentativ ein scotopischer Filter implementiert, der sich durch Hintereinanderausführung der beiden anderen Filter erzeugen lässt. In [Fun05] wird das unter dem Namen OpenVIDIA GPU bekannte Framework vorgestellt. Darin werden unter anderem Korrektur radialer Linsenverzerrung, ein Canny Edge Detector, Handtracking, Bildregistrierung von Panaromateilbildern oder Feature-Vektor Berechnung, die für *Tracking* Algorithmen benötigt werden, implementiert.

In [PGB07] werden verschiedene variationelle Methoden auf der Grafikkarte umgesetzt. So wird Entrauschen von Bildern, Featureextraktion, Detektion und Segmentierung von Texturunregelmäßigkeiten und Bildsegmentierung in Regionen implementiert. Das System schafft die Verarbeitung von Videostreams in Echtzeit bei einer Auflösung von 640×480 Bildpunkten bei 30 Bildern pro Sekunde.
In [MA03] wird die Fast Fourier Transformation mittels Fragment-Shadern auf Grafikhardware umgesetzt. Unter Ausnutzung von Fragment-Shadern und *floating-point*-Texturen erreicht die Implementierung auf einer *nVidia GeForce FX 5800 Ultra* Grafikkarte bei einem 512 mal 512 großen Bild eine Durchsatzrate von 1.6 Bildern pro Sekunde. Die FFT ist damit allerdings auch ein Beispiel eines Algorithmus, der von der Verwendung der GPU nicht so stark profitiert wie dies zum Beispiel Faltungsoperationen tun, da, wie in der Arbeit beschrieben wird, eine FFT Bibliothek, die die FFT auf der CPU berechnet, vergleichbare Performance-Werte liefert. In einer neueren Publikation [SL05] kann aber ein Geschwindigkeitszuwachs gegenüber besagter CPU Bibliothek erreicht werden. In dieser Arbeit wird die FFT zur Rekonstruktion von in der Medizin gebräuchlichen Magnetresonanzbildern verwendet und, um bessere Geschwindigkeit erreichen zu können, auf der Grafikkarte berechnet. Dabei wird mit dem sogenannten *Decimation-in-Time Butterfly*-Algorithmus die FFT berechnet. Generell kommen Fragment-Shader in der Form zum Einsatz, dass mit einem Multipass-Ansatz verschiedene Fragment-Shader nacheinander ausgeführt werden. Um eine bessere Lastverteilung zu erreichen, werden aber auch der Vertex-Shader und der Rasterizer bemüht, um etwa Interpolation von Werten zu bewerkstelligen. Für verschiedene Bildgrößen, angefangen von 256×256 Pixeln bis zu 2048×1024 Pixeln, wurde unter Verwendung von *floating-point*-Genauigkeit ein Geschwindigkeitszuwachs zwischen Faktor 1.33 und 1.97 im Vergleich zu einer CPU-Implementierung erreicht.

Eine Arbeit, die sich mit einer Aufgabenstellung eines Teilproblems des in dieser Masterarbeit behandelten *Tracking* Systems befasst, ist [GRNG05], in welcher Vordergrund/ Hintergrund- Segmentierung in Bildsequenzen mittels Fragment-Shadern bewerkstelligt wird. Dabei werden anhand eines Hintergrundmodells, zusätzlichen Bayes-Schätzungen und eines Kollinearitätskriteriums lokale Pixelnachbarschaften der Größe von 3×3 Pixeln untersucht, um die erforderliche Segmentierung zu berechnen. Es werden die gemessenen Berechnungszeiten ohne Speichertransfers von und zur Grafikkarte angegeben, was aber den Vergleich mit einer CPU-Implementierung unmöglich macht.
Auch *Stereo Vision*-Programme können vom Gebrauch von Grafikhardware profitieren, wie in [WK04] gezeigt wird, indem die Grafikkarte dazu genutzt wird, um durch Verwendung von Fragment-Shadern Tiefenhypothesen für die einzelnen Bildpixel zu berechnen, zu bewerten und auszuwählen. Für eine Bildgröße von 320 mal 240 Bildpunkten läuft das vorgestellte System bei vier Kameras und 20 Tiefenhypothesen laut angegebenen Werten mit vierzehn Bildern pro Sekunde auf einer *nVidia GeForce FX 5600* Grafikkarte.
Sogar einige *Tracking* Algorithmen lassen sich auf der Grafikkarte umsetzen, etwa ein *KLT-Tracker* (siehe [SFPG06] - in dieser Arbeit werden verschiedene Schritte des *KLT-Tracking*-Algorithmus auf Fragment-Programme aufgeteilt). So werden Bildpyramiden für Helligkeitswerte und Gradientenwerte erzeugt, in welchen dann jeweils Features über einen Eckenerkennungs-Algorithmus bestimmt werden. Ausgehend von dieser sogenannten *Cornerness Map*, welche die Features definiert, wird versucht, diese im nächsten Bild der Bildsequenz wiederzufinden. Das Extrahieren der brauchbaren Features aus der *Cornerness Map* wird nach Rücktransfer der *Cornerness Map* in den Hauptspeicher auf der CPU durchgeführt. Das *KLT-Tracking* nach Rückübertragung der brauchbaren Features wieder auf der Grafikkarte, allerdings ist dieser Schritt auf 7 mal 7 Pixelregionen beschränkt. Da aber das *Tracking* auf den einzelnen Pyramidenebenen durchgeführt wird, können so ohnehin auch stärkere Bewegungen erkannt werden. Die Implementierung erreicht eine bis zu 20-mal höhere Ausführungsgeschwindigkeit als auf der CPU. So können bei einem 1024×768 Bild 1000 Features bei 30 Hertz getrackt werden. Als essentiell für korrekte Ergebnisse wird bei dieser Arbeit die Rechengenauigkeit angegeben. So reicht die interne Genauigkeit von 24-Bit bei ATI-Karten, die in der Arbeit fälschlicherweise als 32-Bit Genauigkeit interpretiert wurde, bei 16-Bit Texturformaten bereits für korrekte Ergebnisse aus, während auf nVidia-Karten 32-Bit Texturformate benötigt wurden. Es wird berichtet, dass bei Grafikkarten von ATI die Performance besser ausfiel, was sich durch die unterschiedlichen verwendeten Texturformate erklären lässt, da die Ausführungszeiten von Shaderprogrammen und Transferzeiten unmittelbar mit dem gewählten Texturformat zusammenhängen. Zusätzlich zum *KLT-Tracker* wird in dieser Arbeit auch *SIFT Feature Extraction* auf der

Grafikkarte umgesetzt. Auf einer *nVidia GeForce 7900 GTX* werden 1000 SIFT-Features bei einer Auflösung von 640 × 480 Bildpunkten mit 10 Hertz ermittelt.

Rendering

Aufgrund des Berechnungsaufwands von Rendering-Algorithmen wurde die Beschleunigung der Berechnungen von Rendering-Algorithmen durch Auslagerung auf Grafikhardware in mehreren Arbeiten untersucht.

In [CHH02] wird als Teil des *Ray Tracing* Algorithmus die Schnittberechnung von Strahlen mit Dreiecken mit Hilfe von Fragment-Shadern auf Grafikhardware umgesetzt. Die restlichen Aufgaben werden auf der CPU ausgeführt. Für den GPU-unterstützten *Ray Tracing* Algorithmus wird auf einer *nVidia GeForce 3* GPU, verglichen mit einer CPU-Variante, ein Performancezuwachs von 22% erreicht, laut [LC04] ist das implementierte System allerdings nicht schneller als eine CPU-Variante.

In [PBMH05] wird ebenfalls ein *Ray Tracing* Algorithmus auf der GPU implementiert. Der Prozess des *Ray Tracing* wird umformuliert in einen sogenannten *Streaming Ray Tracer*, der aufgrund seiner Parallelisierbarkeit für eine Umsetzung auf die GPU geeignet ist. Als *Acceleration Data Structure* kommt ein uniformes Grid zum Einsatz, das sich sehr effizient auf 3D-Texturen auf der Grafikkarte abbilden lässt. Vier Fragment-Programme sorgen für die Abarbeitung des *Ray Tracing* Algorithmus: Strahlerzeugung vom Augpunkt, Traversieren des Grids, Schnitt von Strahlen und Dreiecken sowie die Schattierungsberechnung. Die Performance des Systems wird nur theoretisch angegeben. Tatsächliche Messwerte werden nicht angeführt, jedoch wird in [LC04] erwähnt, dass das implementierte System nicht schneller als eine CPU-Variante ist.

In [PDC+03] wird *Photon Mapping* auf der GPU implementiert. Als Datenstruktur für die Speicherung der Photonen kommt ein uniformes Grid zur Anwendung. Für das Füllen dieses Grids werden zwei Algorithmen vorgestellt. Als erstes wird ein Verfahren verwendet, das dies über mehrere Durchläufe mittels Fragment-Programmen und *Bitonic Merge Sort* bzw. binärer Suche bewerkstelligt. Während die binäre Suche mit $\log n$ Texture-Lookups für jedes Photon noch effizient ist und vor allem nur einen Durchlauf benötigt, stellt sich die Berechnungskomplexität des Soriteralgorithmus mit $O(\log^2 n)$ Durchläufen als Problem dar, da jeder Durchlauf für die komplette Anzahl an Photonen durchgeführt werden muss. Deshalb wurde ein zweites Verfahren entwickelt, das unter Ausnutzung von Vertex-Shadern und Stencil-Buffer das Füllen des Grids in einem Durchlauf durchführt. Für das Rendering kommt ein stochastischer *Ray Tracer* zum Einsatz, mittels mehrerer Fragment-Programme werden Schnittpunkte berechnet, für welche dann direkte Beleuchtung, Reflexionen und Lichtbrechung berechnet werden und die Ergebnisse des *Photon Mapping* in die Berechnung miteinbezogen werden, um die *Global Illumination* der Szene erfassen zu können. Für Ergebnisse bezüglich Qualität und Performance wird auf die Arbeit verwiesen.

Als weitere Arbeit, die sich mit dem Thema *Photon Mapping* auseinandersetzt, sei [LC04] erwähnt. In dieser Arbeit wird der *Final-Gathering*-Schritt beim *Photon Mapping* auf die GPU ausgelagert. Kaustiken, Reflexionen und Schatten werden ebenfalls auf der GPU simuliert. Das *Photon Tracing* und alle weiteren Verarbeitungsschritte erfolgen auf der CPU. Ein 512×512 großes Bild wird auf einem *Intel Pentium 4 mit 2.4 Ghz* einer *nVidia GeForceFx 5950* mit mehr als 35 Bildern pro Sekunde gerendert. Für genauere Informationen bezüglich Polygonzahl und Photonenanzahl sei auf die Arbeit verwiesen.
Der „Parthenon"-Renderer aus [Hac05] simuliert *Global Illumination* durch wiederholtes Rasterisieren der Szene von verschiedenen Blickpunkten aus. Aus diesen Ansichten wird eine Schätzung der indirekten Beleuchtung akkumuliert. Direkte Beleuchtung wird in einem weiteren Durchlauf mit Rendertechniken aus der Echtzeitgrafik inklusive *Shadow Mapping* und *Stencil Shadows* berechnet. Der *Final Gathering* Schritt wird im Gegensatz zu CPU-Implementierungen für alle Pixel durchgeführt, nicht nur für eine möglichst repräsentative Teilmenge der Pixel samt anschließender Interpolation, wie dies auf der CPU üblich ist. Die Performance soll laut Autor schneller sein als bei vergleichbaren CPU-Varianten.
In [CH05] wird als Methode zur Lösung von *Radiosity*-Gleichungen *Progressiv Radiosity* eingesetzt, um iterativ über Beschreibung und Berechnung des Energietransfers mittels Formfaktoren einen Fixpunkt zu erreichen bzw. sich einem solchen anzunähern und so die *Global Illumination* einer Szene zu simulieren. Die *Radiosity*-Lösung kann für eine Cornell Box mit 10000 Elementen eine 90%-ige Konvergenz zur korrekten Lösung bei einer Durchsatzrate von 2 Bildern pro Sekunde berechnen.

Weitere Arbeiten

Abschließend soll noch ein kurzer Überblick über weitere Arbeiten aus Bereichen gegeben werden, die nicht zu den bisher genannten Gebieten zählen.
Probleme der algorithmischen Geometrie auf der Grafikkarte zu berechnen, wird in [KEHKL$^+$99] beschrieben. Dabei werden Voronoi Diagramme mittels Parallelprojektionen, linearer Interpolation durch den *Rasterizer* und Verwendung des *Z-Buffers* auf der GPU berechnet. Auf einem *Infinite Reality* Grafikchip wird interaktives Rendering bei einer Auflösung von 512×512 Pixeln und bis zu 10000 Regionsmittelpunkten bewerkstelligt.
In [GLW$^+$05] werden Datenbank-Operationen und Datenbank-Queries auf Grafikhardware umgesetzt. So werden etwa Konjunktion, Selektion, Aggregation, semi-lineare Queries und Joins durch Verwendung der Grafikkarte beschleunigt. Ein Vergleich zwischen einer Grafikkarten-Implementierung eines *nVidia GeForceFX 5900 Ultra* Grafikchips und einer optimierten CPU-Implementierung auf einem *Intel Xeon Dual Prozessor* mit 2.8 GHz zeigt die Performance für verschiedene Operationen. Mit Ausnahme von Akkumulation,

welche auf der GPU 20-mal mehr Zeit benötigt als auf der CPU, erreicht die GPU-Variante bessere Werte, semi-lineare Queries können bis um den Faktor 10 gesteigert werden, die anderen Operationen erreichen einen Performancezuwachs um den Faktor 2 bis 5.

Multi-Threading zur Performance-Steigerung

In [GG05] wird ein *Multi-Threaded* System implementiert, das sowohl den Prozessor als auch die Grafikkarte für Berechnungen nutzt und somit dem in dieser Masterarbeit verfolgten Ansatz entspricht. Ziel des Systems ist das Generieren virtueller Ansichten einer von mehreren Kameras aufgenommenen dynamischen Szene von einer neuen frei wählbaren Kameraansicht.
Zuerst wird eine Vordergrund/Hintergrund-Segmentierung durchgeführt, danach wird die radiale Linsenverzerrung der Kameras auf der Grafikkarte zurückgerechnet, um dann basierend darauf Tiefenwerte zu berechnen. Diese Tiefenwert-Berechnung erfolgt auch teilweise auf der Grafikkarte, oder besser gesagt, im zuständigen Grafikkarten-Thread. Der auf der Grafikkarte berechnete Schritt ist der *Plane-Sweeping*-Algorithmus, der als Ergebnis eine Tiefenmap und ein interpoliertes Farbwertbild zugehörig zu den Tiefenwerten bezüglich der Kameraposition, die zwischen den Kamerapositionen der Eingangsbilder liegt, liefert. Dieser Schritt benötigt für ein 640 × 480 Pixel großes Bild für 40 mögliche Tiefenwerte etwa 100 Millisekunden. Durch Histogrammberechnung aus den Tiefenwerten auf der CPU kann über den Durchschnitt und die Varianz dieses Histogrammes der Suchbereich für den 3D-Raum dynamisch angepasst werden. Die Daten werden durch einen Regularisierungsalgorithmus auf der CPU aufbereitet und in ein 3D-Mesh umgewandelt, welches anschließend mit Hilfe der Grafikkarte für viele verschiedene Kamerapositionen dargestellt werden kann.
Das *Multi-Threaded* System funktioniert so, dass es jeweils einen Thread für die zu erledigenden Aufgaben von CPU und Grafikkarte gibt. Die beiden Threads kommunizieren miteinander und synchronisieren sich über *Mutexe* und sorgen so für eine richtige Abarbeitungsreihenfolge neu eingehender Bilder.
Es werden keine Ergebnisse bezüglich Ausführungszeiten präsentiert, weiters wird erwähnt, dass Ausführungszeiten der einzelnen Arbeitsschritte beim Timing der Thread-Synchronisation herangezogen wurden. Allerdings fallen Ausführungszeiten natürlich für unterschiedliche Hardware verschieden aus, womit sie ein schlechtes Kriterium zur Thread-Steuerung darstellen. Denkbar wäre allerdings, dass so versucht wird, durch Schlafenlegen von Threads Parallelität von CPU- und Grafikkarten-Berechnungen zumindest teilweise zu erreichen. Aber dazu gibt es keine diesbezüglichen Informationen. Stattdessen wird behauptet, dass durch das *Multi-Threaded* System CPU und Grafikkarte komplett parallel arbeiten - diese Aussage kann aber nicht nachvollzogen werden, da eben keinerlei Performanceangaben gemacht werden und das verwendete Testsystem nicht angegeben wird. Somit kann diese Aussage nicht überprüft werden, und auch nach den in dieser

Masterarbeit gemachten Erfahrungen ist diese Aussage anzuzweifeln.

Kapitel 3

Grundlagen der Grafikkarten-Programmierung

Zunächst werden Informationen präsentiert, die zur Programmierung der Grafikkarte - auch GPU (*Graphics Processing Unit*) genannt - unerlässlich sind. Dabei werden alle betreffenden Bereiche vorgestellt, sowohl die hardware-technischen Aspekte als auch die Programmierschnittstellen für Grafikkarten.
Die Hardware muss speziellen Anforderungen Genüge tun, um für GPGPU (*General-Purpose Computation Using Graphics Hardware*) geeignet zu sein. Diese Anforderungen werden in Abschnitt 3.1 spezifiziert.
Bei den Programmierschnittstellen hat man die Möglichkeit, aus mehreren Alternativen zu wählen, die individuelle Eigenschaften und Bedingungen mit sich bringen (siehe Abschnitt 3.2).

3.1 Hardware-spezifische Eigenschaften

Aktuelle GPUs, sowohl von nVidia[1] als auch von ATI[2] (den beiden Marktführern bei Grafik-karten-Hardware), sind in der Lage, GPGPU Anwendungen zu ermöglichen. Je nach verwendetem Modell und Treiber muss man allerdings immer mit individuellen Eigenschaften und Problemen kämpfen. Für die Applikation, die in dieser Masterarbeit entstand, wurde deshalb nur für nVidia-Hardware entwickelt und getestet, jedoch wäre ein Umstellen auf ATI-Harware kein Problem, wenn man diverse Anpassungen vornehmen würde. Generell ist anzuraten, aktuelle Treiberversionen bzw. die letzte als funktionierend bekannte Treiberversion zu verwenden.

[1] Die nVidia Corporation entwickelt Grafikprozessoren und Chipsätze für PCs. (www.nvidia.com)
[2] ATI Technologies Ltd. stellt Grafikkarten und andere Hardwarekomponenten her. (www.ati.com)

Im Folgenden wird er grundlegenden Aufbau von Grafikhardware, definiert durch die sogenannten Grafikpipeline (*graphics pipeline*) beschrieben, die den Datenfluss und die Berechnungsabfolge in Grafikhardware definiert (in Abbildung 3.1 sieht man eine vereinfachte Darstellung der Grafikpipeline, welche die für GPGPU relevanten Aspekte abstrahiert). Die in der Grafikpipeline getroffene Aufteilung in Stages ermöglicht es, für bestimmte Stages hohe Ausführungsgeschwindigkeiten durch parallele Abarbeitung von Instruktionen auf verschiedenen Speicherelementen zu erreichen.

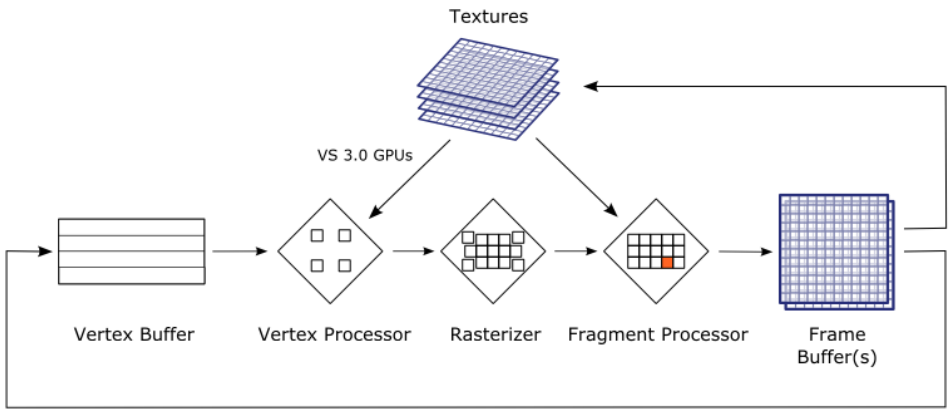


Abbildung 3.1: Die Grafikpipeline (vereinfacht) (nach [OLG+05]).

Das Zusammenspiel der für GPGPU relevanten Pipeline-Stages: In der traditionellen *fixed function pipeline* sind alle Verarbeitungsschritte der einzelnen Stages fest vorgegeben. Bei neuerer Hardware sind der Vertex-Prozessor und der Fragment-Prozessor frei programmierbar.

3.1.1 *Fixed function pipeline*

In der traditionellen *fixed function pipeline* ist jeder Verarbeitungsschritt (also jede Stage) fest in der Hardware verankert und somit vorgegeben. Die in der (vereinfachten) Schematisierung dargestellten Stages unterteilen sich in den *Vertex-Buffer*, Vertex-Prozessor, *Rasterizer* und Fragment-Prozessor (siehe Abbildung 3.1).
Im *Vertex-Buffer* werden über API-Aufrufe die Geometriedaten geladen, die als Vektoren in Objektkoordinaten vorliegen. Der Vertex-Prozessor wandelt die Objektkoordinaten über die Transformation mit der *Modelview-Matrix* und der *Projection-Matrix* in Bildkoordinaten um, definiert aus diesen Vertices Dreiecke und führt Beleuchtungsberechnungen für die Vertices aus [OLG+05].

Der *Rasterizer* hat die Aufgabe, Fragmente für alle Dreiecke zu erzeugen. Fragmente sind potentielle Pixel im fertigen Bild, die in nachfolgenden *Pipeline-Stages* allerdings wieder verworfen werden können, da sie z.B. von anderen Fragmenten überdeckt werden können oder nicht auf dem Bildschirm sichtbar sein werden. Für jedes Fragment werden in der *fixed function pipeline* sämtliche Parameterwerte, die durch die Vertices gegeben sind, interpoliert.
Schließlich werden im Fragment-Prozessor für alle Fragmente Farbwerte berechnet, die sich unter anderem durch die interpolierten Parameterwerte definieren und eventuelle Texturwerte zur Farbberechnung heranziehen.
Abschließende Tests sondern Fragmente, die keinen Beitrag zum fertigen Bild leisten, aus, und somit bleibt nur eine Teilmenge der Fragmente über, die als Pixel in den Framebuffer geschrieben werden.
Die Programmierung kann lediglich über die Befehle der Grafik-API zur Steuerung der einzelnen Stages erreicht werden, und man kann somit nur mit wenigen festgelegten Befehlen und Rechenoperationen Berechnungen durchführen.
Für die in Abbildung 3.1 dargestellte Schematisierung bedeutet dies, dass der Programmierer die Verarbeitungsschritte der einzelnen Stages nicht beeinflussen kann, so etwa bei normalem Rendern einer Szene das standardmäßige Beleuchtungsmodell von *per-vertex lighting* (*Gouraud-Shading*) auf *per-fragment lighting* (z.B: *Phong-Shading*) umzustellen. Dies ist erst mit dem erweiterten Shadermodell, welches programmierbare Vertex- und Fragment-Shader unterstützt, möglich. Die Programmierbarkeit ohne Shader ist stark eingeschränkt und eigentlich nur über Umwege und für sehr wenige Probleme sinnvoll einsetzbar, zum Beispiel über *Blending*-Operationen oder *Stencilbuffer* - all diese Ansätze müssen aber über die Standard-API Funktionen bewerkstelligt werden und schränken somit die Möglichkeiten der Programmierung ein.
Genauere Informationen finden sich in [SWND05].

3.1.2 *Programmable pipeline*

Um die Beschränkungen der *fixed function pipeline* aufzuheben, wurde mit der Einführung von programmierbaren Vertex- und Fragment-Shadern die sogenannte *programmable pipeline* eingeführt. In der Praxis sieht das so aus, dass die fixe Funktionalität des Vertex-Prozessors und Fragment-Prozessors gegen individuelle Vertex-Shader- und des Fragment-Shader Programme ausgetauscht werden können. Anfangs wurden diese Programme in Assembler geschrieben, bald gab es aber auch Hochsprachen, die in Abschnitt 3.2.2 beschrieben werden. Mit jedem neuen Shadermodell fallen Einschränkungen weg, allerdings ist die Programmierbarkeit von GPUs - verglichen mit der CPU - nach wie vor einge-

schränkt. Diese Einschränkungen ergeben sich aus der Anforderung an Parallelisierbarkeit von Verarbeitungsschritten. Genannte Einschränkungen sind Ressourcen-bezogene Limitierungen wie Programmlänge, Anzahl von Speicherzugriffen (*Texture-Lookups*) etc., aber auch Einschränkungen bei Funktionalitäten, wovon die fehlende Unterstützung für *Scatter*-Operationen in Fragment-Shadern wohl die gravierendste Einschränkung darstellt.
Dies liegt daran, dass Fragment-Shader *single-instruction, multiple-data* (SIMD) Parallel-Prozes-soren sind. Auf alle Elemente eines Streams (das ist die Menge der Input-Elemente) werden dieselben Instruktionen (dasselbe Shaderprogramm) angewendet, seit Shadermodell 3.0 aufgelockert durch verbesserte Loop- und Branching-Unterstützung. Im Gegensatz dazu sind Vertex-Shader *multiple-instruction, multiple-data* (MIMD) Architekturen, die zwar theoretisch flexiblere Programmierung erlauben und Branching (Verzweigungen) effizienter ausführen als Fragment-shader, allerdings nicht die Effizienz von Fragment-Shadern erreichen(siehe [LKO05]). Aufgrund der Tatsache, dass bei Problemen der Computergrafik deutlich mehr Fragmente als Vertices zu verarbeiten sind, findet man auf GPUs mehr Fragment- als Vertex-Shader (*nVidia GeForce 7950 GT*: 8 Vertex-Shader-Einheiten und 24 Fragment-Shader-Einheiten, *ATI Radeon X1950 XT*: 8 Vertex-Shader-Einheiten und 48 Fragment-Shader-Einheiten). Das ist der Grund, warum Fragment-shader bei GPGPU-Problemen die erste Wahl sind bzw. bisher waren, denn mit der kommenden Grafikkarten-Generation (die ersten *nVidia GeForce 8* Karten sind bereits erhältlich) verschwinden die hardwarebedingten Unterschiede zwischen den einzelnen Shadern, die mit den neuen Geometrie-Shadern auch einen neuen Shadertyp aufzuweisen haben. Durch die sogenannten *Unified Shader* kann nun die Grafikkarte sämtliche Shadereinheiten nach eigenem Ermessen als Vertex-, Fragement- oder Geometrie-Shader einsetzen.
Hier sollen aber nochmals kurz die bisher geltenden Einschränkungen erläutert werden, da sie zum Verständnis beitragen und speziell auch bei dieser Masterarbeit prägenden Charakter hatten.
Durch die SIMD-Architektur von Fragment-Shadern werden Fragmente effizient parallel bearbeitet, allerdings entscheidet der SIMD-Prozessor, in welcher Reihenfolge die Shader zur Abarbeitung kommen, was Algorithmen unmöglich macht, die eine bestimmte Ausführungsreihenfolge der Speicherelemente voraussetzen, wie z.B. die Berechnung von Integralbildern (deshalb werden in dieser Masterarbeit Integralbilder auf der CPU berechnet).
Weiters ist die Schreibposition für das Ergebnis des Shaderprogramms, nämlich die Position des gerade bearbeiteten Fragments, immer fix. Zwar können für die Berechnung des gerade in Arbeit befindlichen Fragments beliebige Texturwerte verwendet werden (wobei die Quelltextur nicht gleich der Zieltextur sein darf und die Anzahl der *Texture-Lookups* je nach verwendetem Shaderprofil beschränkt ist), was einer *Gather*-Operation (siehe auch

Abschnitt 4.3) entspricht, die Schreibposition ist aber immer fix, somit werden *Scatter*-Operationen (siehe Abschnitt 4.3) nicht unterstützt. Es benötigt aber die Mehrheit aller Algorithmen *Scatter*-Operationen, wodurch nicht alle Algorithmen für eine Umsetzung für die GPU geeignet sind.

Mit CUDA (siehe auch Abschnitt 3.2.3), welches für *nVidia GeForce 8* Grafikchips und folgende Modelle entwickelt wurde und auf nVidia-Karten beschränkt ist, fallen einige der in dieser Masterarbeit genannten Einschränkungen der GPU-Programmierung weg, speziell das erwähnte *Scatter*-Problem ist gelöst - *Scatter*-Operationen werden in CUDA explizit unterstützt.

3.1.3 Datentransfer CPU-GPU

Eine weitere wichtige hardware-technische Begebenheit sind die zeitintensiven Datentransfers von Hauptspeicher zu Grafikkartenspeicher und wieder zurück. Da AGP-Grafikkarten im Vergleich zu PCI-Express Grafikkarten hier hohe Transferzeiten verursachen - AGP 2x erreicht 532 MB/sec in eine Richtung [SD98], AGP 8x erreicht dementsprechend 2.1 GB/-sec in eine Richtung, PCI-Express 16x erreicht 4 GB/sec (theoretische 2.5 GBit/sec in beide Richtungen [BAS03] werden in der Praxis zu 1.6 GBit/sec, was 0.2 GB/sec entspricht - mal 16 ergibt 3.2 GB/sec [Buc05]) - ist die Verwendung von PCI-Express Grafikkarten bei zeitkritischen Aufgabenstellungen unerlässlich, speziell bei GPGPU Programmen, die auf den dauernden Transfer von Daten angewiesen sind, wie das beim Projekt dieser Masterarbeit der Fall ist. Bei Echtzeitsystemen, die ständig neue Daten auf die Grafikkarte holen und zurückschicken müssen, sind niedrige Transferzeiten essentiell - es sollten nur absolut notwendige Daten transferiert und schnelle Texturformate, also Standard-Texturformate (siehe Abschnitt 3.1.5), verwendet werden, wenn das möglich ist.
Selbst auf Grafikkarten, die den PCI-Express-Bus nutzen, sind die Transferzeiten der Flaschenhals schlechthin. Die Transferzeiten hängen direkt vom verwendeten Texturformat ab, je mehr Genauigkeit man braucht, desto längere Transferzeiten müssen einkalkuliert werden (siehe Tabelle 8.2).

3.1.4 Rechengenauigkeit bei floating-point Operationen auf der GPU

Es ist von fundamentaler Bedeutung, die Genauigkeit von *floating-point*-Berechnungen auf der Grafikkarte zu verstehen, um Fehler zu vermeiden bzw. Abweichungen von Berechnungen, die auf der CPU andere Ergebnisse liefern, erklären zu können. Dazu muss man sich die Eigenschaften von Gleitkommazahlen veranschaulichen.
Eine Gleitkommazahl wird durch folgende Beschreibung charakterisiert:

$$sign \times 1.mantissa \times 2^{(exponent-bias)}$$

Die verfügbare Bit-Anzahl wird auf die darzustellenden Teile für Vorzeichen, Mantisse und Exponent aufgeteilt. So werden beim IEEE 754 Standard etwa ein Bit für das Vorzeichen, 23 Bit für die Mantisse und 8 Bit für den Exponenten veranschlagt. Der Exponent wird um den festen Wert 127 verschoben, um mit positiven Werten auch negative Werte darstellen zu können - also auch sehr kleine Werte, die zu ihrer Darstellung einen negati-

ven Exponenten benötigen. Man spart sich so ein Bit und kann dies für andere Teile der Darstellung einsetzen, um deren Genauigkeit zu erhöhen. Weiters muss die führende 1 der Mantisse (bei Binärzahlen ist die erste relevante Ziffer immer 1) nicht extra gespeichert werden, und somit erhält man auch hier ein zusätzliches Bit für die Genauigkeit.

Die erreichbare Genauigkeit auf GPUs (siehe auch [Buc05]) durch die GPU *floating-point*-Formate sieht folgendermaßen aus:

- nVidia fp32: 23-Bit Mantisse, 8-Bit Exponent (1 Bit für das Vorzeichen)
- ATI fp24: 16-Bit Mantisse, 7-Bit Exponent (1 Bit für das Vorzeichen)
- nVidia fp16: 10-Bit Mantisse, 5-Bit Exponent (1 Bit für das Vorzeichen)

Mit dem IEEE 754 Standard und dem nVidia fp32-Format lassen sich in der Mantisse 2^{23} verschiedene Werte repräsentieren, und somit lassen sich Mantissenwerte bis zur sechsten Nachkommastelle exakt darstellen.
Beim fp24-Format von ATI kann man immerhin noch vier Nachkommastellen exakt repräsentieren, während beim fp16-Format von nVidia nur mehr 3 Nachkommastellen exakt darstellbar sind.
Die auf Grafikkarten maximal mögliche Genauigkeit beträgt 32-Bit. Es wird jedoch der von der CPU-Programmierung bekannte *floating-point*-Standard IEEE 754 nur teilweise von den Grafikkartenherstellern umgesetzt. Dieser Standard definiert Regeln, die zur Anwendung kommen, wenn eine Gleitkommazahl nicht exakt durch die vorhandene Bit-Anzahl repräsentiert werden kann und daher gerundet werden muss. Laut IEEE 754 wird auf die letzte darstellbare Nachkommastelle gerundet, also auf den nächsten darstellbaren Wert. Im Falle von zwei gleich weit entfernten nächsten Zahlen wird die Binärzahl mit geradem Bit gewählt. GPUs folgen diesen Rundungsregeln nicht zwingend und die Grafikkartenhersteller geben auch keine Informationen preis, wie ihre Hardware die Rundungsaufgaben erledigt. Allerdings finden sich in [HL04] Angaben zu Fehlerintervallen für verschiedene arithmetische Operationen mit *floating-point*-Operanden. Es werden die Unterschiede zwischen IEEE 754 und GPUs bei den begangenen Fehlern angeführt. Offensichtlich wird, dass für keine Operation die Fehler der CPU-Werte verglichen mit jenen der GPU-Werte gleich sind. Besonders eklatant ist der Unterschied bei der Division, wo etwa bei der getesteten nVidia-GPU der maximale Fehler mehr als doppelt so hoch ausfällt wie der maximale Fehler des IEEE 754 Standards, der immer 0.5 mal den mit der ersten nicht mehr exakt darstellbaren Nachkommastelle maximal darstellbaren Wert ausmacht. Auch die ATI-GPU liefert in diesem Test ähnliche Ergebnisse wie die nVidia-GPU (für genaue

Werte siehe [HL04]).

3.1.5 Texturformate und Größenbeschränkungen

Es existieren zu den in Abschnitt 3.1.4 und 3.1.3 genannten Beschränkungen weitere Einschränkungen, die die Grafikkartenprogrammierung erschweren. So sind die Größen von Texturen in jeder Dimension beschränkt, etwa darf auf *nVidia GeForce 7* Grafikkarten Texturen die Arraygröße von 4096 Elementen pro Dimension nicht überschritten werden, bei neueren *nVidia GeForce 8* Karten erhöht sich dieses Limit auf 8192 Einträge pro Dimension - somit lassen sich beispielsweise mit Hilfe von 2D Texturen auf einer *nVidia GeForce 7* Grafikkarte Datenströme der Größe von 16.777.216 Elementen verarbeiten. Bedenkt man aber nun, dass pro Texturelement dank der Vektordatentypen 4 Werte zur Verfügung stehen, erhöht sich diese Anzahl auf 67.108.864 Elemente. Hier kommt man bei *floating-point*-Texturen aber bereits an die Grenze des verfügbaren Grafikkartenspeichers (die beschriebene Konstellation benötigt bereits 256 MB Speicher). Benötigt man noch größere Datenströme, müssen diese entweder in eine 3D Textur verpackt oder in mehrere Teiltexturen aufgeteilt werden, allerdings hat man dann ohnehin mit der Problematik des unzureichenden Grafikkartenspeichers zu kämpfen.
Eine weitere Limitierung der Programmierung der GPU existiert durch die begrenzte Genauigkeit der Texturen - nicht zu verwechseln mit der Rechengenauigkeit innerhalb eines Shaders, die 16-Bit bzw. 32-Bit *floating-point*-Genauigkeit (mit den in Abschnitt 3.1.4 genannten Abweichungen vom IEEE 754-Standard) problemlos unterstützt. Je nach verwendetem Texturformat definiert sich nämlich die Genauigkeit der Texturen (die sozusagen den Datentyp für die Speicherung darstellen, während die Genauigkeit innerhalb eines Shaderprogrammes die Rechengenauigkeit bei den Rechenoperationen festlegt). Ab *nVidia GeForce 6* GPUs stehen *floating-point*-Texturformate zur Verfügung, die von 8 Bit bis 32 Bit Genauigkeit reichen. Zu beachten ist, dass der Datentransfer direkt proportional zur Speichergenauigkeit ausfällt, somit benötigt eine 32-Bit Textur etwa die vierfache Transferzeit einer 8-Bit Textur, um vom Hauptspeicher in den GPU-Speicher oder umgekehrt transferiert zu werden.
Die momentan maximal unterstützte Genauigkeit liegt bei 32-Bit, was für viele wissenschaftliche Anwendungen nicht ausreichend ist, da 64-Bit Genauigkeit benötigt wird. Durch Verwendung der Standard-Texturformate (das sind traditionell 8- oder 16-Bit Texturformate) erhält man schnellere Transferzeiten als bei Verwendung von *floating-point*-Texturformaten, allerdings haben diese Texturformate einerseits den Nachteil der geringeren Genauigkeit als 32-Bit Formate, andererseits werden bei diesen Standard-Texturformaten sämtliche Werte beim Schreiben in die Textur am Ende der Ausführung des Fragment-Shaders gemappt, und zwar bei z.B. 8-Bit Genauigkeit auf die Werte $\frac{0}{255}$ bis $\frac{255}{255}$ - dies kommt einem Clamping der Werte auf das Intervall [0..1] gleich. Verwendet

man diese Texturformate, hat der Programmierer dafür zu sorgen, dass sämtliche Ergebniswerte eines Shaders ins Intervall [0..1] fallen. Ist dies nicht der Fall, kann man sie noch mittels Scaling und Biasing in dieses Intervall verschieben und bei späteren Shaderaufrufen wieder zurücktransferieren, allerdings geht mit diesem Verfahren ein entsprechender Genauigkeitsverlust einher. Das Endergebnis ist dann aber wieder auf das Intervall [0..1] beschränkt.

Ein weiterer Aspekt, den man sich als Programmierer zunutze machen kann, sind die Modi zur Texturgrenzen-Behandlung (*texture border modes* oder auch *texture clamping modes*). Dabei handelt es sich um Regeln, wie die Grafikkarte bei Texturzugriffen auf Texel, die außerhalb der Textur liegen, vorzugehen hat. Solche Zugriffe passieren immer dann, wenn die berechneten Texturkoordinaten außerhalb der tatsächlichen Texturkoordinatengrenzen liegen. Es existieren verschiedene Texturgrenzen-Behandlungsmodi, z.B. kann das jeweils örtlich näheste Texel aus der entsprechenden Spalte/Zeile herangezogen werden, das sich wieder innerhalb der Textur befindet, oder man kann den Modus auf sich wiederholende Texturen einstellen. Man kann auch durch Definieren einer Randfarbe sämtliche Texelzugriffe auf Pixel, die außerhalb der Textur liegen, durch das Retournieren eines bestimmten Farbwertes setzen. Der Vorteil dieses Ansatzes ist, dass diese Grenzbehandlung automatisch passiert, wann immer man auf ein Texel zugreift, das außerhalb der Textur liegt, man muss sich also als Programmierer nur um das Einstellen des *texture border modes* kümmern, der für jede Textur eigens definiert wird. In der GPGPU-Praxis kann man sich diesen Vorteil bei einigen Algorithmen zunutze machen, so etwa bei in der Bildverarbeitung üblichen Faltungsoperationen. Muss man beim Implementieren von Faltungsoperationen auf der CPU entweder sich selbst um die Ausnahmefälle der Bildränder kümmern oder gleich das Bild bei jeder Faltung schrumpfen lassen, so werden bei einer GPU-Variante die *texture border modes* dazu eingesetzt, ganz automatisch diese Ausnahmesituationen zu behandeln. Das Shaderprogramm muss nicht angepasst werden. Wird durch einen Textur-LookUp-Befehl auf eine Textur im Shaderprogramm zugegriffen und sind dabei die spezifizierten Texturkoordinaten außerhalb der Textur, so kümmert sich die Grafikhardware ganz von allein darum, dass der entsprechend richtige Farbwert an das Shaderprogramm retourniert wird.
Auch die Einstellungen zur Texturinterpolation erlauben interessante Möglichkeiten. So wird je nach verwendeter Interpolationsregel bei Texturzugriffen auf Koordinatenpositionen, die zwischen mehreren Texeln liegen, z.B. linear interpoliert oder eine *Nearest Neighbour Interpolation* angewendet. Die lineare Interpolation ist gerade bei GPGPU-Problemen nicht immer erwünscht, da es nicht bei allen Problemstellungen korrekt ist, interpolierte Werte zu berechnen, weshalb von Algorithmus zu Algorithmus zu entscheiden ist und man gegebenenfalls besser eine *Nearest Neighbor Interpolation* verwendet. Hier sei erwähnt, dass die einzelnen Textur-Interpolationsmodi effizient in der Hardware umgesetzt sind, das heißt, dass, wenn es der umzusetzende Algorithmus zulässt bzw. wenn er davon profitiert, man selbstverständlich z.B. lineare Interpolation verwenden sollte.

3.2 Programmierschnittstellen

Es folgt ein Überblick über die vorhandenen Programmierschnittstellen zur Programmierung von Grafikkarten. Dabei muss zwischen Grafik-APIs und Shadersprachen unterschieden werden.
Die Grafik-API ist die eigentliche Schnittstelle zur Programmierung der GPU. Sie stellt die grundlegenden Befehle zur Verfügung, um die Grafikkarte anzusprechen und zu steuern. Hier haben sich in den letzten Jahren zwei Standards etabliert, einerseits DirectX von Microsoft und andererseits OpenGL. Während DirectX besonders für die Programmierung von Spielen breitere Verwendung findet, wird OpenGL vermehrt bei wissenschaftlichen Arbeiten eingesetzt, auch deshalb, da OpenGL Plattform-Unabhängigkeit bieten kann.
Shadersprachen erlauben das Schreiben von Programmen für die programmierbaren Shadereinheiten, das sind, wie bereits weiter oben erwähnt sind das Vertex-, Fragment- und seit neuestem Geometry-Shader. Im Laufe der letzten Jahre haben sich mehrere Shadersprachen entwickelt. nVidia stellt mit Cg eine plattformunabhängige Shadersprache zur Verfügung. HLSL, das Pendant zu Cg von Microsoft, verwendet dieselbe Syntax wie Cg. Mit GLSL existiert eine speziell für OpenGL entwickelte Shadersprache.

3.2.1 Übersicht der Grafik-APIs

Die Grafik-API hat - wie in Abbildung 3.2 zu sehen ist - die Aufgabe, als Schnittstelle zwischen Programm und Grafikhardware zu fungieren und Befehle zum Ansprechen der Grafikhardware durch das Programm entgegenzunehmen, an die Grafikkarte weiterzuleiten und gegebenenfalls fehlende bzw. nicht unterstützte Features zu bemängeln oder zu emulieren.

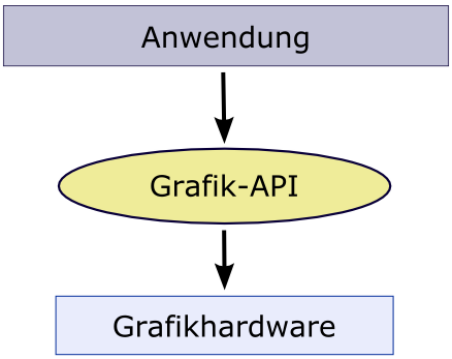


Abbildung 3.2: Die Grafik-API: Schnittstelle zwischen Anwendung und Grafikhardware

Sie abstrahiert die Funktionalitäten der Grafikhardware und stellt diese der Anwendung über Funktionen zur Verfügung.

Entsprechende Aufgaben sind das Erzeugen von Geometrie, Definieren von Lichtquellen, Parametrisieren des Beleuchtungsmodells und des Kameramodells, Hochladen von Texturdaten auf die Grafikkarte und anschließendes Texturieren von Objekten oder Operationen mit dem Framebuffer, um hier nur einige Aufgaben zu nennen. Die einzelnen Stages der Grafikpipeline werden dabei über eine Vielzahl von Funktionen gesteuert. Die Programmierung der Grafikkarte kann aber durch bloße Verwendung von Grafik-APIs nur bis zu einem gewissen Punkt betrieben werden (nur über das Aufrufen der API-Befehle). Beide APIs sind für das Schreiben von GPGPU Programmen verwendbar.

OpenGL

OpenGL ist eine Grafik-API und stellt somit die Schnittstelle zur Grafikhardware dar. Die erste Implementierung von OpenGL (*Open Graphics Library*) geht auf das Jahr 1993 zurück. Aus Iris GL von Silicon Graphics entstanden, wurde es zu einem offenen, hardware- und plattformunabhängigen Standard. Neue Versionen kommen zwar nicht im gleichen Tempo wie bei DirectX, aktuell ist man bei Version 2.1 angelangt, doch es steht bereits Version 3.0 in den Startboxen, welche wie DirectX 10 ebenfalls Geometry-Shader unterstützen wird. OpenGL ist abwärtskompatibel und emuliert von der Grafikkarte nicht unterstützte Features per Software. Die Entwicklung von OpenGL wird vom sogenannten OpenGL Architecture Review Board (ARB) überwacht, welches sich aus den Firmen 3Dlabs, Apple Computer, ATI, Dell, IBM, Intel, nVidia, SGI oder Sun Microsystems zusammensetzt. Ursprünglich war auch Microsoft mit an Bord, verließ aber im März 2003 das Konsortium.
Da in der Vergangenheit neue OpenGL-Versionen nicht in der gleichen Frequenz heraus-

kamen, wie dies bei DirectX der Fall war, musste ein anderes Konzept für die Implementierung neuer Hardware Features eingesetzt werden. Bei OpenGL passiert dies über sogenannte OpenGL-Extensions, das sind Erweiterungen, die von den Hardwareherstellern zur Verfügung gestellt werden. Neue Hardwarefunktionalität kann so in kürzester Zeit verfügbar gemacht werden. Für den Fall, dass die verwendete Hardware die benötigte Extension nicht unterstützt, ist es daher die Aufgabe des Programmierers, abzufragen, ob die Extension verfügbar ist. Vom OpenGL Architecture Review Board geprüfte Erweiterungen beginnen in ihrem Namen mit dem Präfix „ARB“, Hersteller-spezifische Erweiterungen tragen ein entsprechendes, dem Hersteller zuordenbares Präfix. Das „EXT“-Präfix ist für Erweiterungen reserviert, die von mehreren Herstellern unterstützt werden. Als weiterführende Literatur sei hier auf die Standardwerke [SWND05] und [Shr03] verwiesen.

DirectX (Direct3D oder DirectX Graphics)

Die DirectX-Bibliothek stellt eine Schnittstelle zur Programmierung sämtlicher Multimedia-Aufgaben unter Windows-Systemen zur Verfügung, darunter Direct3D, welches den Zugriff auf die Grafikhardware unterstützt und somit bei der GPU-Programmierung zum Einsatz kommt, DirectAudio zum Zugriff auf die Sound-Hardware oder DirectInput zur Verwendung von einer Vielzahl von Eingabegeräten.
Shaderprogrammierung wird seit Version 8 unterstützt, Programme wurden in dieser Version noch mit Assembler geschrieben, ab Version 9 war HLSL verfügbar. Mit Windows Vista ist mittlerweile bereits Version 10 erschienen, das als wichtiges neues Feature die Programmierung von Geometry-Shadern unterstützt.

3.2.2 Shadersprachen

Shadersprachen wurden eingeführt, um den Prozess der Shaderprogrammierung dadurch zu vereinfachen, dass nicht länger Assembler-Programme geschrieben werden mussten, sondern die Programmierung über eine Hochsprache vonstatten ging. Sie stellen alle notwendigen Funktionalitäten zur Programmierung von Shaderprogrammen zur Verfügung. Momentan hat man als Programmierer die Wahl zwischen Cg, HLSL und GLSL. Alle drei Sprachen sind für die Lösung von GPGPU-Problemen geeignet. Sie stellen dem Programmierer eine API zur Verfügung, mit deren Hilfe er die für die jeweilige Shadersprache geschriebenen Shaderprogramme laden, kompilieren, Variablen übergeben oder Textur-Einheiten parametrisieren kann. Mit Hilfe dieser API können also Shaderprogramme in der Grafikpipeline installiert werden, die den Vertex-Prozessor und den Fragment-Prozessor der Grafikpipeline mit individuellen Programmen bestücken. Shaderprogramme können nicht nur zur Kompilierzeit, sondern auch zur Laufzeit eines Programmes kompiliert werden. Um Hardware mit unterschiedlichen Funktionalitäten zu unterstützen, werden sogenannte Profile verwendet, die einen bestimmten Funktionssatz definieren.
Für welche Shadersprache man sich entscheidet, hängt meist von der Zielplattform und vom zu entwickelnden Programm ab. Cg und GLSL können durch Plattformunabhängigkeit glänzen, Cg bietet den Vorteil, nicht auf eine Grafik-API beschränkt zu sein, wie dies bei HLSL und GLSL der Fall ist. HLSL ist bei Spieleprogrammierung oft die erste Wahl, da es mit DirectX eng verknüpft ist und neue Features meist als Erstes unterstützt (siehe Geometry-Shader). Wissenschaftliche Arbeiten verwenden häufig Cg oder GLSL.

Cg

Cg (*C for Graphics*) ist eine Shadersprache zum Schreiben von Shaderprogrammen, wurde von nVidia 2002 vorgestellt und ist mittlerweile in Version 1.5 verfügbar. Cg ist plattformunabhängig, es kann sowohl mit DirectX als auch mit OpenGL verwendet werden. Die Syntax ist stark an C angelehnt, wie bereits der Name der Sprache vermuten lässt. Die Cg Runtime Library stellt die API dar, durch welche der Programmierer die Shaderprogramme in der Grafikpipeline installieren kann.
Es existieren elementare Datentypen wie float, half, bool etc., spezielle 2-, 3- und 4-dimensionale Vektorvarianten dieser elementaren Typen sowie viele Operatoren und mathematische und GPU-relevante Funktionen, die speziell auf das Rechnen mit diesen Vektordatentypen optimiert wurden.
Das Zusammenspiel von Cg, der gewählten Grafik API und der Grafikhardware ist in Abbildung 3.3 schematisiert.

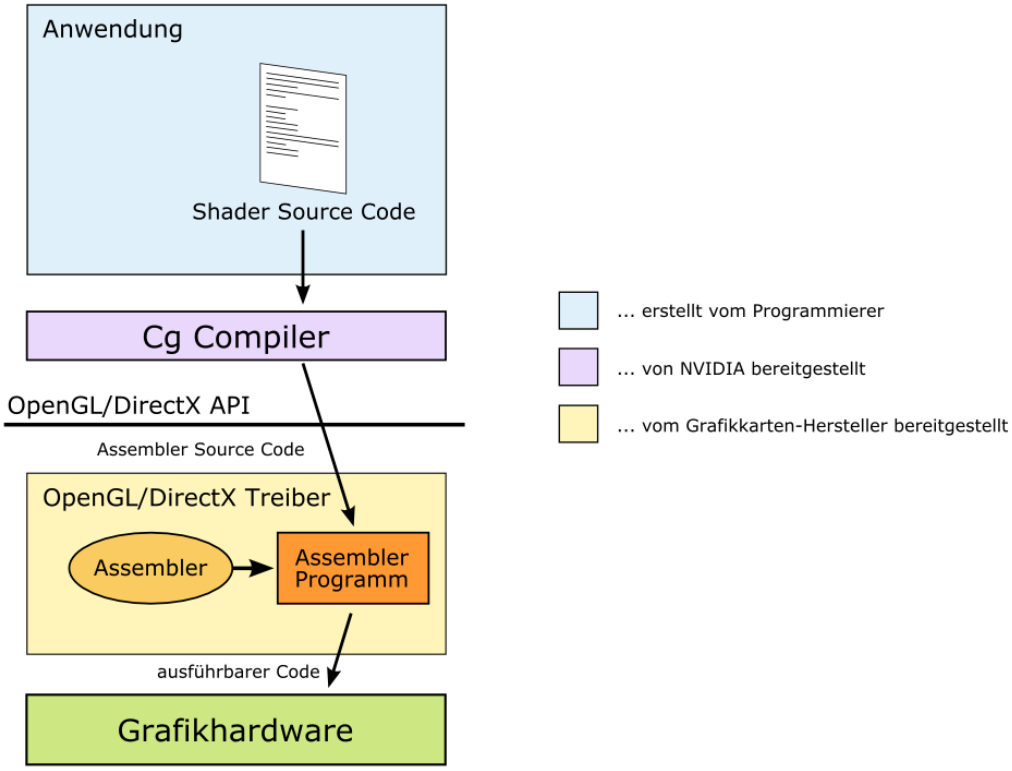


Abbildung 3.3: Die Cg Ausführungsumgebung (nach [Ros04])

Der Programmierer stellt den Shader Code bereit, der vom Cg Compiler in Assembler Code umgewandelt und der Grafik API übergeben wird, welche diesen mit ihrem Assembler in einen für die Grafikhardware ausführbaren Code verwandelt.

Weitere Informationen finden sich in [FK03].

HLSL

Die *High Level Shading Language* (kurz HLSL) ist eine Shadersprache zum Schreiben von Shaderprogrammen und wurde von Microsoft 2002 mit Erscheinen von DirectX 9.0 vorgestellt. HLSL ist auf die Verwendung mit DirectX beschränkt, was nur den Einsatz unter Windows-Systemen ermöglicht. Der HLSL-Compiler geht den Umweg über ein allgemein gehaltenes Assemblerprogramm, welches dann zur Laufzeit von einem speziell für die vorhandene Hardware angepassten Assembler übersetzt wird.
Da HLSL und Cg in Zusammenarbeit von Microsoft und nVidia entwickelt wurden, ähneln sich beide Sprachen sehr.
Fragment-Shader heißen in HLSL Pixelshader, sind aber in ihrer Funktionalität absolut equivalent. Neu hinzugekommen ist die Unterstützung der Programmierung von Geometry-Shadern (seit DirectX 10).
Die HLSL-Ausführungsumgebung wird in Abbildung 3.4 dargestellt und zeigt die Abarbeitungs-

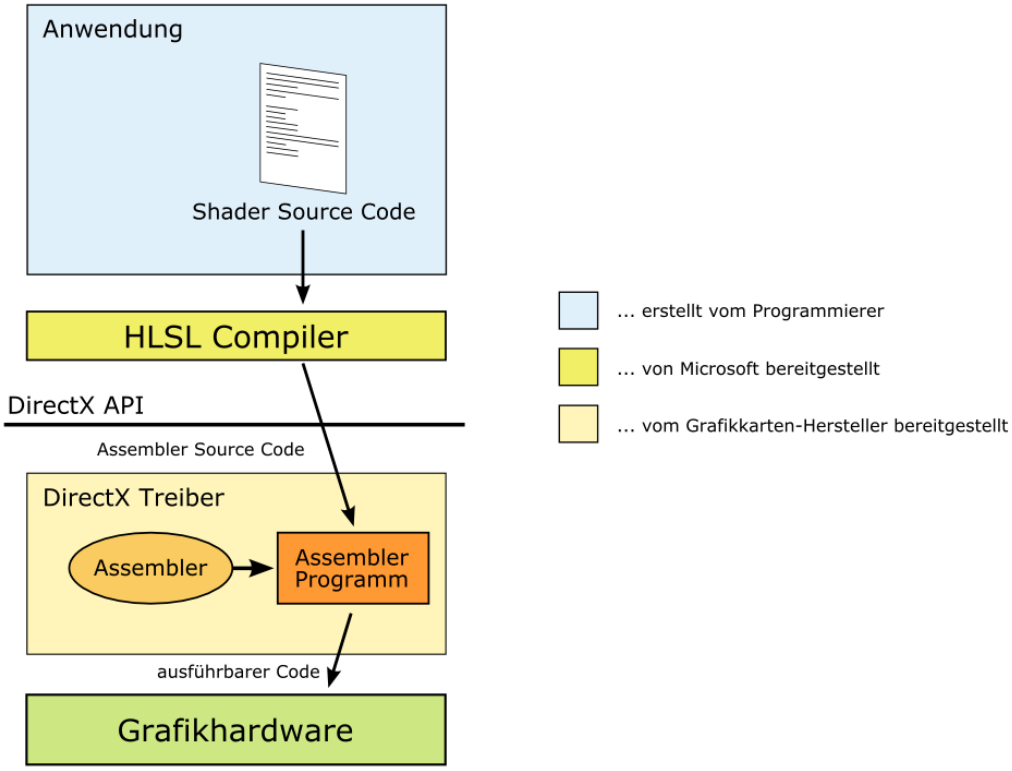


Abbildung 3.4: Die HLSL Ausführungsumgebung (nach [Ros04])

reihenfolge von Shadercode bis hin zur fertigen, für die Grafikkarte verständlichen Maschinensprache. Der vom Programmierer bereitgestellte Shader Code wird vom HLSL-Compiler in Assembler Code umgewandelt und an DirectX übergeben, welches diesen

dann mittels Assembler in ein für die Grafikhardware ausführbares Maschinenprogramm übersetzt.

GLSL

GLSL (*(Open) Graphics Library Shading Language*) stellt eine Shadersprache zum Entwickeln von Shaderprogrammen dar und wurde mit der OpenGL Version 2.0 im Jahr 2004 eingeführt, nachdem sie zuvor schon ab Version 1.5 über Extensions verfügbar war. GLSL ist ebenfalls an die Programmiersprache C angelehnt. Sie kann nur in Kombination mit OpenGL verwendet werden und ist mit ihr eng verwoben. GLSL unterstützt ähnliche Datentypen und Funktionen wie die beiden anderen Shadersprachen, allerdings fehlt ein half- sowie double-precision *floating-point*-Format. Es existieren spezielle Anbindungen an OpenGL, so ist es in einigen Shaderprofilen möglich, im Shaderprogramm auf Variablen des OpenGL-States direkt zuzugreifen.

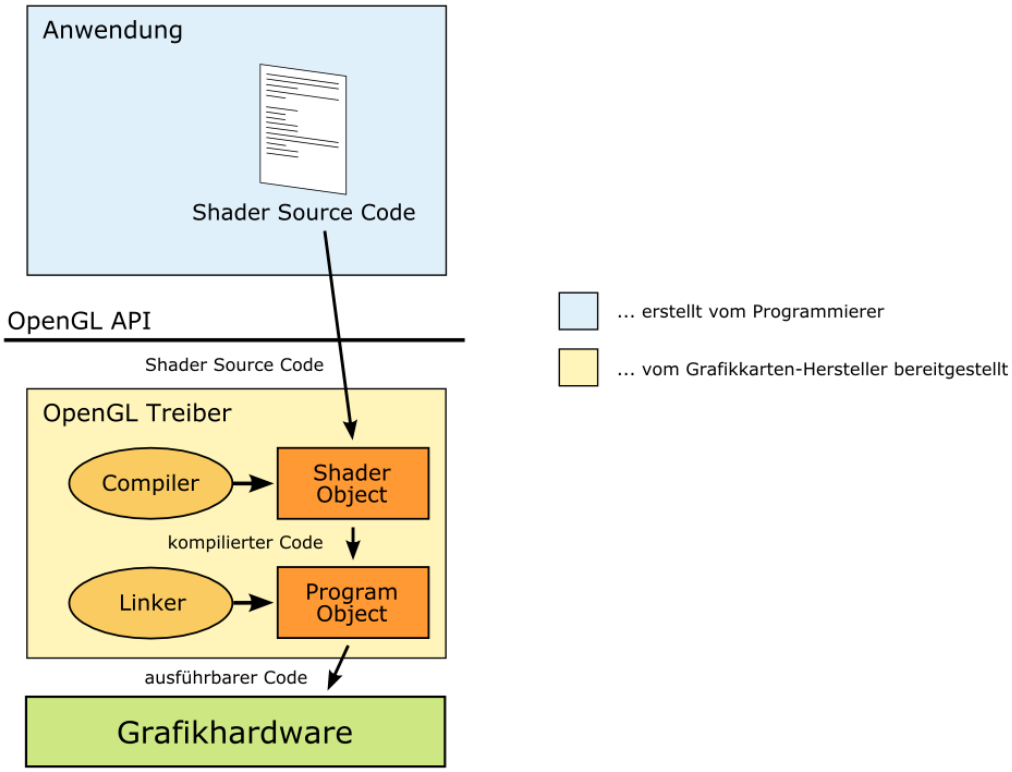


Abbildung 3.5: Das GLGL Ausführungsmodell (nach [Ros04])

Im Gegensatz zu Cg und HLSL ist bei GLSL der Compiler Teil des OpenGL-Treibers. Dies ermöglicht es Hardware-Herstellern, einen für ihre Hardware optimierten Compiler herauszubringen. Das Kompilieren von Shaderprogrammen erfolgt dynamisch zur Laufzeit.
In Abbildung 3.5 sieht man das Zusammenspiel von Shaderprogramm und OpenGL API im *execution model* für GLSL. Der Programmierer stellt den Shader Code bereit, der an die OpenGL API übergeben wird, welche diesen zuerst in ein Shader Object kompiliert und ihn per Linker schließlich in ein ausführbares Program Object verwandelt, welches von der Grafikhardware ausgeführt werden kann.
Zusätzliche Eigenschaften und Details findet man in [Ros04].

3.2.3 Frameworks zur GPU-Programmierung

Neben der Standardvariante (Verwendung einer Grafik-API in Verbindung mit einer Shadersprache) gibt es auch den Ansatz, GPU-Programmierung über sogenannte Frameworks zu realisieren, das sind libraries, die meist vollständig oder zumindest bis zu einem gewissen Teil von der Grafikhardware abstrahieren und dem Programmierer ein eigenes Programmiermodell und Interface zur Verfügung stellen, mit dessen Hilfe er GPGPU-Programme entwickeln kann.

BrookGPU

BrookGPU wurde von der Stanford University[3] entwickelt, ist an die Programmiersprache C angelehnt und spezialisiert sich mit Hilfe eines Datenstrommodells ausschließlich auf GPGPU-Aufgaben (siehe [12] für weiterführende Informationen).

Sh

Sh (*Serious Hack Inc. at the time*) wurde von der University of Waterloo[4] vorgestellt und orientiert sich an C++. Es ist sowohl für grafische Aufgaben als auch für GPGPU geeignet. Mehr Informationen finden sich unter [8].

[3]Die Stanford University (www.stanford.edu) ist eine der weltweit führenden Forschungs- und Lehrinstitutionen und unterhält unter anderem das Stanford Computer Graphics Laboratory (http://graphics.stanford.edu), das für die Entwicklung von BrookGPU verantwortlich ist.
[4]University of Waterloo: http://www.uwaterloo.ca/

CUDA

CUDA (*Compute Unified Device Architecture*) - momentan noch im Beta-Status - wird von nVidia speziell für G80-Grafikkarten und höher entwickelt und läutet eine neue Ära bei der GPGPU-Programmierung ein. CUDA ist ein Programmier-Interface für die Entwicklung komplexer, berechnungsintensiver Problemstellungen, die mit Hilfe von Standard C gelöst werden können. Einige traditionelle Probleme der GPU-Programmierung (siehe Abschnitt 4) sind in CUDA nicht mehr vorhanden, als Beispiel sei hier das Wegfallen des *Scatter*-Problems genannt. Weitere Informationen findet man unter [10].

CTM

CTM (*Close To (the) Metal*) ist ein Hardware-Interface für AMD-Parallelprozessoren und ATI-Grafikkarten, das Stream Processing beschleunigen soll. CTM ermöglicht Entwicklern uneingeschränkten Zugriff auf Instruktionen und Speicher von AMD-Parallelprozessoren, die für *Stream Processing* ausgelegt sind, sowie ATI-Grafikkarten. CTM verfolgt also im Gegensatz zu CUDA einen low-level Ansatz und ermöglicht hardwarenahen Zugriff auf die Parallelprozessoren.
Detaillierte Informationen finden sich unter [7].

Kapitel 4

Konzepte der GPGPU-Programmierung

In diesem Kapitel werden grundlegende Konzepte der GPGPU-Programmierung vorgestellt. Grafikkarten wurden zum Berechnen und Anzeigen von Computergrafiken entworfen, und dem-entsprechend müssen Eigenschaften und Regeln beachtet werden, wenn man versucht, die GPU auch für andere Aufgabenstellungen zu verwenden. So muss jedes GPGPU-Problem so formuliert werden, dass die Grafikkarte durch das Berechnen der vermeintlichen Grafik, die tatsächlich aber der zu berechnende Algorithmus ist, die Aufgabenstellung löst. Das heißt, es muss jeder umzusetzende Algorithmus in Form von geometrischen Primitiven und darauf definierten Shading-Operationen definiert werden. Um dies zu bewerkstelligen, wurden in [Har05] grundlegende Konzepte eingeführt, die im Folgenden beschrieben werden. Gelingt es mit diesen Konzepten und Regeln, einen Algorithmus auf der GPU umzusetzen, erhält man in vielen Fällen (immer dann, wenn der Algorithmus für eine Umsetzung der Grafikkarte geeignet ist - siehe dazu Abschnitt 4.3) bei einer Datenmenge, die die Shadereinheiten der Grafikkarte auch auslastet (mehrere hundert Datenelemente), durch die parallele Ausführung der Shaderprogramme auf eben diesen Daten einen im Vergleich zur CPU enormen Leistungsschub. Es sind bei weitem nicht alle Algorithmen, die auf CPUs umgesetzt werden können, auf der GPU umsetzbar. Oft lassen sich aber zumindest Teilprobleme auf die Grafikkarte auslagern. Allerdings resultiert aus der GPU-Programmierung ein zusätzlicher Programmieraufwand verglichen mit der CPU-Programmierung und muss deshalb beim Softwaredesign einkalkuliert werden. Der Mehraufwand muss sich durch die erreichbaren Geschwindigkeitsgewinne rechnen.
In [Har05] findet sich ein Maß, mit dessen Hilfe man die Eignung eines Algorithmus für eine etwaige GPU-Umsetzung abschätzen kann. Dieses Maß ist die arithmetische Intensität. Je höher die arithmetische Intensität ist, desto höher ist der zu erwartende Geschwindig-

keitszuwachs durch eine Umsetzung auf der GPU. Die arithmetische Intensität berechnet sich wie folgt:

$$arithmetic\ intensity = \frac{operations}{words\ transferred\ (\text{bandwidth})}$$

Sie sagt aus, dass sich ein Algorithmus desto besser eignet, je mehr Berechnungen man auf den vom Systemspeicher in den Grafikkartenspeicher übertragenen Daten durchführt. Als Beispiel für einen diesbezüglich geeigneten Algorithmus wird auf [KW05] verwiesen. In dieser Abhandlung wird demonstriert, wie sich partielle Differentialgleichungen auf der GPU lösen lassen.
Die Eigenschaft des Parallelismus von Rechenschritten impliziert auch Unabhängigkeit der Berechnungsausführung, das bedeutet, dass für alle Datenelemente eines Datenstroms dieselben Operationen ausgeführt werden und dass die Berechnung eines Elements nicht von der Berechnung anderer abhängig sein darf. Weiters ist die Berechnungsreihenfolge der einzelnen Elemente nicht fest vorgegeben, sondern wird von der Hardware festgelegt, und somit sind Algorithmen, die eine bestimmte Abarbeitungsreihenfolge der Speicherelemente benötigen und nicht in einen Algorithmus mit mehreren Durchläufen umformuliert werden können, für eine Umsetzung auf der GPU nicht geeignet.

4.1 CPU-GPU Analogien

Bekannte Konzepte der CPU-Programmierung lassen sich auf jeweilige Entsprechungen bei der GPU-Programmierung abbilden - diese Analogien werden im Folgenden behandelt. Die folgenden Konzepte wurden von Mark Harris in [Har05] vorgestellt. Es folgt eine kurze Beschreibung der einzelnen Analogien.

GPU Textures = CPU Arrays

Die Datenstrukturen, die die GPU verarbeiten kann, sind Texturen (Texturen sind rechtecktige Arrays von Daten [SWND05] wie etwa Farbwerte. Die einzelnen Arrayelemente werden *Texel* genannt). Sämtliche Algorithmen müssen deshalb so programmierbar sein, dass sie mit ihren Daten als Input- und Output-Arrays arbeiten.

Kernel: GPU Fragment Programs = CPU „Inner Loops“

Arbeitet man auf der CPU ein Array ab, so wird dies dadurch erreicht, dass eine Schleife nach und nach alle Elemente des Arrays berechnet. Der Berechnungsschritt (Schleifen-

rumpf) wird dann für alle Elemente des Arrays durchgeführt. Auf der GPU stellen die Fragment-Programme genau diesen Berechnungsschritt, also den Schleifenrumpf, dar. Um den Schleifenkopf braucht man sich nicht zu kümmern, da die Abarbeitung der einzelnen Arrayelemente automatisch parallelisiert passiert. Für alle zu erzeugenden Fragmente wird der Fragment-Shader ausgeführt. Man spricht in diesem Zusammenhang auch von Kernels [Har05]. Man hat aber dafür zu sorgen, dass für jedes Element des Input-Arrays ein Fragment erzeugt wird, das genau mit der richtigen Position im Output-Array verknüpft ist.

Render-to-Texture = Feedback

Um Ergebnisse von Berechnungen der Shaderprogramme weiterverarbeiten zu können, werden die Ergebnisdaten wieder in eine Textur und somit in den Grafikspeicher geschrieben, da sie sonst nach Beendigung des Shaderprogramms verloren gehen, weil sie während der Ausführung des Shaderprogramms nur in lokal verfügbaren Variablen gehalten werden.
Wurden in den Anfängen der GPGPU die Ergebnisse noch in den Framebuffer gerendert und dann von dort in eine Textur kopiert, wird bei aktuelleren Arbeiten optimalerweise direkt in eine Zieltextur gerendert. Dieses Verfahren ermöglicht eine bessere Geschwindigkeit als der Umweg über den Framebuffer, wird als Render-to-texture bezeichnet und kann mittels *pbuffers* oder der neueren *Frame Buffer Objects* realisiert werden (siehe Abbildung 4.1). Hat man die Ergebnisdaten erst einmal in einer Textur im Grafikspeicher, kann diese dann als Input für weitere Shaderprogramme dienen oder auch zurück in den Systemspeicher ausgelesen werden.
Anzumerken ist, dass die Zieltextur innerhalb eines Shaderdurchlaufes nicht als Quelltextur verwendet werden darf! Diese Einschränkung ist direkte Folge der Parallelisierbarkeit von Shaderprogrammen. Man beachte den Unterschied zur CPU-Programmierung, bei der das Lesen und Schreiben in ein und dasselbe Array kein Problem darstellt.
Berechnet sich ein Algorithmus über mehrere Zwischenschritte, so lässt man jeden Zwischenzustand von einem eigenen Shaderprogramm berechnen. Allerdings muss ein Berechnungsschritt komplett für den Datenstrom abgeschlossen werden, bevor die Berechnung des nächsten Schrittes beginnen kann. Man verwendet den zuletzt berechneten Ausgabestrom (also die Output-Textur) als Input-Textur für das nächste Shaderprogramm. Die gerade eben für den Input verwendete Textur wird zur Ausgabetextur. So wiederholt sich das Spiel für alle weiteren etwaigen nachfolgenden Shaderprogramme. Man spricht bei dieser Technik auch von *texture ping-ponging* oder *ping-pong buffering* [Mic05].

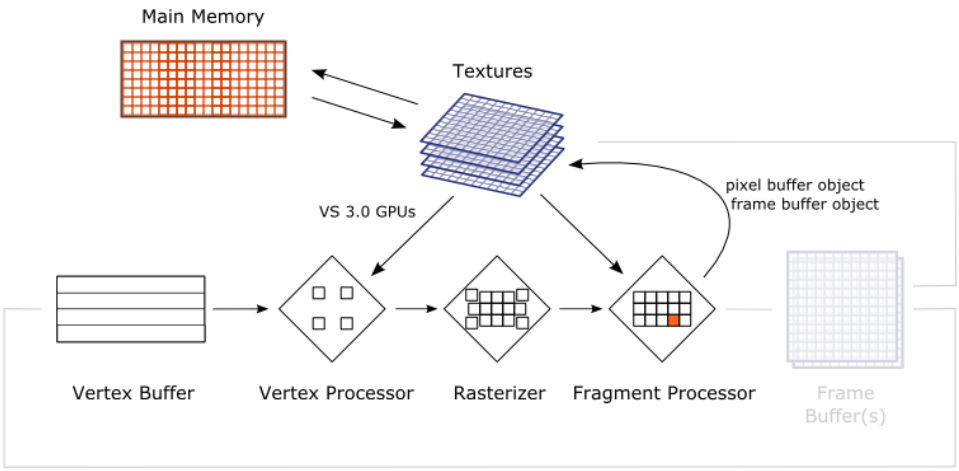


Abbildung 4.1: Die Grafikpipeline bei GPGPU Anwendungen (angelehnt an [OLG+05])

Geometry Rasterization = Computational Invocation

Um Berechnungen mit dem Fragment-Shader durchzuführen, müssen Fragmente erzeugt werden. Dies ist Aufgabe des Rasterizers, der aus zuvor erzeugten Geometriedaten entsprechende Fragmente erzeugt. Für jedes dieser Fragmente wird dann das Fragment-Programm ausgeführt. Der Programmierer muss sicherstellen, dass die gewünschte Anzahl von Datenelementen, also Fragmenten, erzeugt wird. Gewährleistet werden kann dies dadurch, dass man ein Viereck als Geometrie verwendet, welches dann mittels einer Orthogonalprojektion (keine perspektivische Verzerrung) gerendert wird. Dabei wählt man den Viewport mit denselben Dimensionen, die auch der Eingabedatenstrom besitzt. Somit stimmen die Inputdatenmenge und die erzeugte Fragmentmenge, die genau einer Position im Output-Array enspricht, überein, und man hat eine Eins-zu-eins-Abbildung von Inputelement auf Fragment und von Fragment auf Outputelement realisiert (man spricht auch von *One-to-one Pixel to Texel Mapping*).

Texture Coordinates = Computational Domain

Um auf Texturwerte lesend zugreifen zu können, verwendet man Texturkoordinaten, die als Array-Index fungieren. Beim Lesen aus Texturen, deren Größe ungleich der aktuellen *Output Range* ist, passiert entweder eine *data amplification/magnification* (die Textur wird sozusagen hochgerechnet) oder eine *data minification* (die Textur wird heruntergerechnet). Dies geschieht durch Anwenden einer zuvor beim Erzeugen der Texturen fest-

gelegten Vorschrift, standardmäßig wird linear interpoliert, oft ist in der GPGPU aber eine *Nearest Neighbor Interpolation* vorzuziehen. Zu beachten ist, die Größe der Ergebnistextur (die *Output Range*) hat nichts mit der *Computational Domain* zu tun, wird durch die Anzahl der erzeugten Fragmente durch den Rasterizer definiert und hängt somit von mehreren Faktoren wie definierter Geometrie, Kameraparameter, Viewport etc. ab.

Vertex Coordinates = Computational Range

In der GPGPU wird die Geometrie erzeugt (zb: vier Vertices, die zusammen ein Viereck bilden), welche zusammen mit dem Vertex-Prozessor festlegt, wie viele Outputfragmente später vom Rasterizer erzeugt werden. Die Vertex-Koordinaten werden durch den Vertex-Prozessor unverändert weitergegeben, und es werden keine weiteren Berechnungen durchgeführt. Die *Output Range* ist also direkt von den Vertexkoordinaten abhängig, das heißt, die Ausgabemenge von Fragmenten hängt direkt von der erzeugten Geometrie ab.

4.2 Datenströme

Der Begriff des Datenstroms beschreibt eine beliebig große Menge an Elementen gleichen Datentyps, auf denen Operationen durchgeführt werden [Buc05]. Die Gleichheit des zugrunde liegenden Datentyps ist notwendige Bedingung für die Parallelisierbarkeit von Verarbeitungsschritten auf der GPU. Die Operationen selbst werden in eine gekapselte Einheit zusammengefasst, die auch als Kernel oder Kerneloperation bezeichnet wird [Buc05, Har05]. Sie stellt den Schleifenrumpf dar, der für jedes Element des Datenstroms genau einmal angewendet wird. Die einzelnen Kernel-Berechnungen sind voneinander unabhängig, daraus folgt auch der Umstand, dass Eingabedatenströme nicht gleich dem Ausgabedatenstrom sein dürfen, da sonst Inkonsistenzen entstehen können. Weiters lassen sich nur durch diese Unabhängigkeit die Kernel-Berechnungen effizient auf mehreren Shadereinheiten zur selben Zeit parallel ausführen. Um einen großen Nutzen aus der parallelen Verarbeitung des Datenstroms ziehen zu können, empfiehlt es sich, einen großen Datenstrom zu verwenden, um etwaigen Overhead aus Verwaltungsaufwand zu minimieren.

4.3 Realisierbare Operationen auf der Grafikhardware

Nun werden verschiedene Operationen, die sich auf der GPU implementieren lassen, vorgestellt. Lokal arbeitende Algorithmen, die nicht auf Ergebnisse anderer Berechnungen angewiesen sind, eignen sich besonders gut für eine Umsetzung auf der GPU, während sich

globale Algorithmen schlecht auf die GPU portieren lassen. Im Kontext der Grafikkarten-Programmierung versteht man unter lokalen Algorithmen solche, die zur Berechnung eines Datenelements keine oder nur wenige Informationen aus der lokalen Nachbarschaft, die aus Gründen der Cache-Effizienz möglichst in unmittelbarer Domainnähe sein sollten, benötigen. Globale Algorithmen sind dann entsprechend Algorithmen, die auf eine Vielzahl von Informationen und Abhängigkeiten angewiesen sind und somit der einfachen Parallelisierbarkeit entgegenwirken. Als Beispiel für lokale Algorithmen seien hier stellvertretend aus dem Bereich der Bildverarbeitung Farbraumumrechnungen oder Faltungsoperationen erwähnt. Nicht alle lokal operierenden Bildverarbeitungsalgorithmen sind jedoch für eine Umsetzung auf der GPU geeignet: So lassen sich beispielsweise Histogramme nur mit Mühe über den Umweg bzw. die Kosten eines zusätzlichen Verarbeitungsdurchlaufs für jeden Histogrammbin auf der GPU realisieren, da für eine effiziente Histogramm-Berechnung die durch Fragment-Shader nicht unterstützte *Scatter*-Operation benötigt wird. Doch selbst Algorithmen, die ohne Scattering auskommen, können für eine Umsetzung auf der GPU völlig ungeeignet sein, wie das Beispiel der Berechnung von Integralbildern zeigt. Da Integralbilder in einer festen Reihenfolge berechnet werden (z.B. von der linken oberen Bildecke ausgehend zeilen- bzw. spaltenweise) und diese Reihenfolge zwingend eingehalten werden muss, um ein korrektes Ergebnis zu erhalten, sind sie auf der GPU nicht sinnvoll berechenbar, da Kerneloperationen in beliebiger Reihenfolge auf dem Datenstrom angewendet werden. Die einzige funktionierende Möglichkeit wäre wieder eine Implementierung mittels mehrerer Durchläufe, genauer gesagt benötigte man die Anzahl der im Bild vorkommenden Pixel an Durchläufen, und in jedem Durchlauf selbst würde die Kerneloperation für alle Pixel ausgeführt werden. Wie man sieht, ist dieser Ansatz ineffizient und daher ungeeignet. Schließlich seien noch globale Algorithmen als Algorithmen spezifiziert, die eine beliebige Anzahl beliebiger Elemente des Datenstroms zu beliebiger Zeit während der Ergebnisberechnung benötigen. Viele aus der CPU-Programmierung bekannte Algorithmen fallen in diese Klasse, beispielsweise Suchalgorithmen.

Das Speichermodell der GPU unterscheidet sich grundlegend von dem der CPU [LKO05]. Daraus resultieren einige wichtige Einschränkungen bei der GPU-Programmierung. Zwar kann für die Berechnung des gerade in Arbeit befindlichen Fragments auf beliebige Texturwerte zugegriffen werden (wobei die Textur nicht die gerade in Arbeit befindliche Textur sein darf, da diese Konstellation nicht definiert ist), was einer *Gather*-Operation (siehe Abbildung 4.2) gleichkommt, die Schreibposition ist aber immer fix, was *Scatter*-Operationen (siehe Abbildung 4.3) unmöglich macht. Wie bereits erwähnt, benötigen aber viele Algorithmen *Scatter*-Operationen, wodurch sie für eine Umsetzung für die GPU schlecht

geeignet sind.

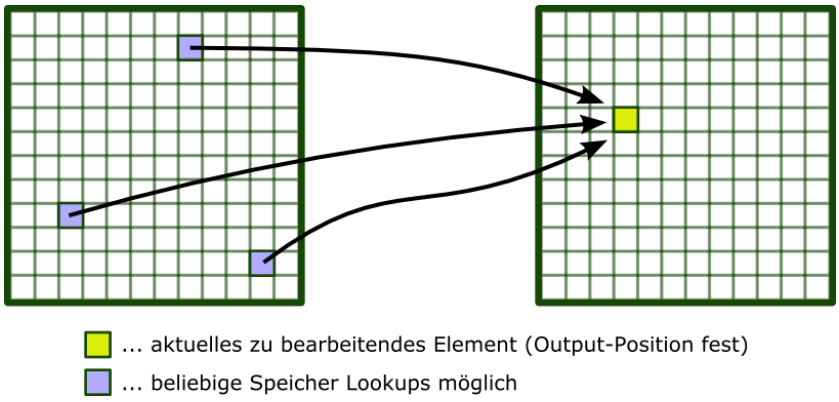


Abbildung 4.2: Die Funktionsweise der *Gather*-Operation

Das Ergebnis berechnet sich aus verschiedenen Arraywerten, die Ouput-Position ist fix.

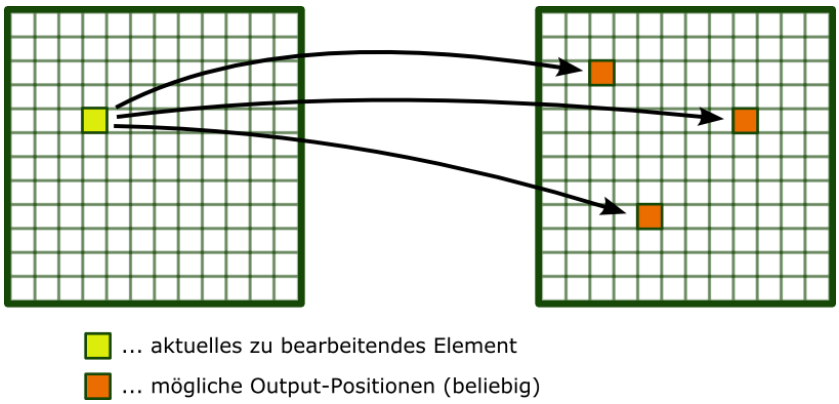


Abbildung 4.3: Die Funktionsweise der *Scatter*-Operation

Ausgehend vom gerade zu bearbeitenden Array-Index kann auf beliebige Positionen geschrieben werden, die Pfeile sollen hier lediglich die Möglichkeit aufzeigen, auf beliebige Speicherpositionen schreiben zu können.

Zur besseren Verdeutlichung: Eine *Gather*-Operation ist folgende Zuweisung:

```
int i = array_2D[index_x,index_y];
```

Eine *Scatter*-Operation ist:

```
array_2D[index_x,index_y] = i;
```

In [Buc05] gibt es einige Ansätze, die das Problem der fehlenden *Scatter*-Operation in Fragment-shadern zu umgehen versuchen. So gelingt es für fixe *Scatter*-Addressen, *Scatter*-Operationen in *Gather*-Operationen zu konvertieren auf Kosten eines zusätzlichen Durchlaufes (in weiterer Folge auch *pass* genannt). Eine weitere vorgestellte Technik ist *address sorting*, das ebenfalls *Scatter*-Operationen in *Gather*-Operationen konvertiert, jedoch einen weiteren Durchlauf benötigt. Mit dieser Technik ist es allerdings möglich, auch Probleme mit *Scatter*-Addressen, die nicht fix sind, zu realisieren. Schließlich wird auch noch die Möglichkeit in Betracht gezogen, Vertex-Shader für *Scatter*-Operationen zu verwenden. Für genauere Details und Einschränkungen der Algorithmen sei aber auf die oben genannte Arbeit verwiesen.

Die Schreibposition im Outputdatenstrom ist immer fest und durch die Position des gerade durch den Fragment-Shader in Arbeit befindlichen Fragments definiert. Als Standard-Operation ergibt sich somit eine Abbildungsfunktion, die ein Element des Eingabedatenstroms auf ein Element des Ausgabedatenstroms abbildet. Alle bisherigen Überlegungen ensprechen dieser Art von Operation.

Bei einigen Algorithmen ist es aber notwendig, aus einem großen Eingabestrom einige wenige Werte oder einen einzelnen Wert zu berechnen, z.B. Minimum, Maximum oder die Summe eines Datenstroms. Für solche Anforderungen existiert in der GPU-Programmierung die *Reduce*-Operation. Sie funktioniert so, dass durch Zusammenfassen von lokal benachbarten Datenstromelementen zu einem neuen Wert ein kleinerer Datenstrom erzeugt wird, der dann in weiteren Verarbeitungsschritten erneut verkleinert wird, bis man schließlich den ursprünglichen Datenstrom mit Hilfe dieser wiederholt angewendeten Reduktionsoperation in einen Ergebniswert umgewandelt hat.

In der Praxis bieten sich Datenstromdimensionen an, die einem Vielfachen von 2 entsprechen, da dann durch Reduzierung um den Faktor 2 schließlich genau ein Wert übrig bleibt. Hat eine Dimension des Datenstroms etwa einen ungeraden Wert, stellt sich die Reduktion als schwieriger heraus und muss um die Behandlung von Spezialfällen (wenn z.B. am Arrayrand nicht die beabsichtigte Anzahl von Eingabewerten für die Reduktion vorhanden ist) erweitert werden. In Abbildung 4.4 sieht man die Funktionsweise der *Reduce*-Operation skizziert.

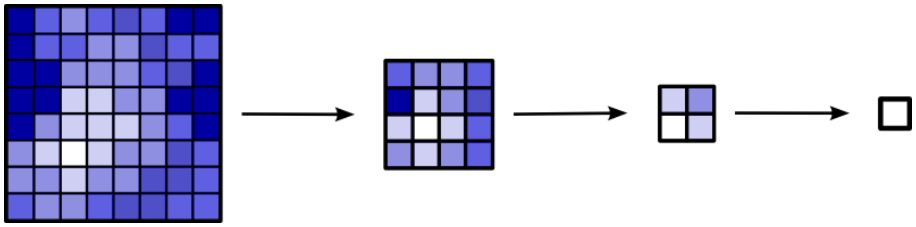


Abbildung 4.4: Die Funktionsweise der Reduktionsoperation

Aus einem Datenstrom mit vielen Werten berechnet sich z.B. wie hier das Maximum (helle Farben symbolisieren hier höhere Werte).

4.4 Texturen als GPU-Datenstruktur

Die auf der Grafikhardware vorhandenen Datenstrukturen, die die Datenströme enthalten, sind die Texturen. Durch Grafik-API-Befehle lassen sich Texturen erzeugen, parametrisieren, kopieren und löschen, durch die Shaderanwendung werden die Texturen dann mit Ergebnissen der durchgeführten Operationen beschrieben. Es existieren spezielle Grafik-API-Befehle, mit deren Hilfe Texturen vom Hauptspeicher in den Grafikkartenspeicher und retour geschrieben werden können.
Es sind ein-, zwei- und dreidimensionale Texturen definiert. Die maximal möglichen Texturgrößen sind in Abschnitt 3.1.5 erwähnt. Aufgabe des Programmierers ist es, die verwendeten Datenstrukturen geeignet abzubilden, sei es der einfache Fall, ein 2D Array auf eine 2D Textur abzubilden oder ein ein- oder mehrdimensionales Array auf eine 2D oder 3D Textur umzuverteilen. Weiters sollte man sich die Eigenschaft zunutze machen, dass pro Texel vier Werte gespeichert werden können. Somit bietet es sich an, einen Datenstrom gegebenenfalls in vier Teile zu zerlegen und auf die vier „Texturebenen“ aufzuteilen. Nutzt man dieses Feature nicht, verliert man den Vorteil einer um den Faktor 4 erhöhten Ausführungsgeschwindigkeit. Zu beachten ist aber, dass etwaige Array-Lookups nun ebenfalls auf diese spezielle Aufteilung auf die Texturebenen Rücksicht nehmen müssen, um sicherzustellen, dass tatsächlich auf die richtigen Arrayelemente zu den Berechnungen zugegriffen wird.
Für das Zugreifen auf beliebige Arraypositionen der Textur kommen sgn. Texturkoordinaten zum Einsatz. Bei der Definition des zu rendernden Vierecks müssen auch gleich Texturkoordinaten definiert werden, die später angeben, wie die entsprechenden Texturen sozusagen auf das Viereck „geklebt“ werden (man spricht auch von *Mapping*). Der Vertex-Prozessor gibt die Texturkoordinaten dann bei GPGPU-Programmen unverändert weiter. Der Rasterizer interpoliert schließlich bei der Erzeugung der Fragmente die Tex-

turkoordinaten für jedes einzelne Fragment. Übergibt man später dem Shaderprogramm diese Texturkoordinaten - was sich durch entprechende Aufrufe von API-Funktionen der Shadersprache bewerkstelligen lässt - so kann man mit Hilfe dieser Texturkoordinaten innerhalb des Shaderprogramms Arrayelemente über diese Texturkoordinaten, die sozusagen den Arrayindex darstellen, ansprechen. Für jede Textur-einheit der Grafikkarte lassen sich so eigene Texturkoordinaten übergeben, oft reicht aber für GPGPU-Probleme bereits ein Satz von Texturkoordinaten aus.

4.5 Umsetzung von Algorithmen als GPGPU-Programm

Aus den vorgestellten Mechanismen und Konzepten der GPGPU-Programmierung resultieren folgende nötigen Schritte zur Umsetzung eines Programms als GPGPU-Programm auf der GPU:

1. auf die Grafikkarte auslagerbare Algorithmusteile ermitteln
2. GPU-Datenstrukturen (Texturen) erzeugen und die Eingabedatenströme in diese laden
3. den Fragment-Shader mittels Funktionsaufrufen der API der Shadersprache in der Grafik-pipeline installieren
4. texturiertes Viereck definieren und mit der exakt der Elementmenge des Datenstroms entsprechenden Größe mit orthogonaler Kamera, die genau auf dieses Viereck ausgerichtet sein muss, mittels Shaderprogrammen, die den Algorithmus umsetzen, rendern - zusätzlich muss der Viewport exakt die Größe des Vierecks abdecken
5. für weitere Berechnungsdurchläufe Schritt 3 und 4 wiederholen, dabei die Ergebnistextur des vorangegangenen Schrittes als Eingabedatenstrom verwenden - für mehrere Durchläufe bietet sich das *ping-ponging*-Verfahren an (Abschnitt 4.1) - zu lange Shaderprogramme gegebenenfalls auf mehrere Shaderprogramme aufteilen
6. Auslesen der Ergebnistextur in den Hauptspeicher über einen Aufruf einer entsprechenden Grafik-API-Funktion

Kapitel 5

Grundlagen und Konzepte der *Multi-Threaded*-Programmierung

In diesem Kapitel werden Grundlagen und Konzepte der *Multi-Threaded*-Programmierung sowie bekannte Probleme und verfügbare Lösungsansätze vorgestellt.
Serielle Programme arbeiten sequentiell einem bestimmten Programmpfad folgend, der sich aus Variablenzuständen und entsprechenden Abfragen und Kontrollstrukturen ergibt, die Anweisungen ab. Sie arbeiten nach und nach die einzelnen Aufgabenstellungen ab, können aber nicht mehrere Dinge gleichzeitig berechnen. Vielmehr liegen gerade in Mehrkernprozessoren sämtliche Prozessorkerne bis auf einen, der die gesamte Arbeit tut, brach. Will man alle Prozessorkerne in die Durchführung der Berechnungen miteinbeziehen, so muss man eine andere als die serielle Programmiertechnik verwenden. Die benötigte Programmiertechnik heißt in diesem Fall *Multi-Threading*.
Der Begriff *Multi-Threading* deutet bereits die Funktionsweise des Konzeptes an. Es existieren mehrere Programmfäden (auch Threads [TS02] genannt), die gleichzeitig verschiedene Operationen und Berechnungen durchführen können. Mit gleichzeitig ist gemeint, dass die gesamte zur Verfügung stehende Rechenleistung auf die einzelnen Threads aufgeteilt wird, sei es nun die Rechenleistung eines Prozessors oder mehrerer Prozessorkerne.
Dabei ist es Aufgabe des Betriebssystems, dafür Sorge zu tragen, dass alle Threads in absehbarer Zeit einen Prozessor zugeteilt bekommen, um ihre Berechnungen ausführen zu können. Das Betriebssystem ist auch wieder die Instanz, die den einzelnen Threads den Prozessor nach der ihnen zugeteilten Zeit (man spricht auch von Zeitscheiben) wieder wegnimmt, um ihn anderen Threads zur Verfügung zu stellen. Dieser Vorgang des Zuteilens und Entziehens der Prozessorzeit muss aber für die Threads völlig transparent bleiben, ein weiterer Punkt, den das Betriebssystem gewährleisten können muss.
Sämtliche im Thread gültigen Speicherbereiche müssen vor und nach dem Entziehen des

Prozessors wieder im selben Zustand sein, damit eine konsistente Programmausführung gesichert bleibt. Dabei kann nicht gesagt werden, in welcher Reihenfolge die einzelnen Threads an die Reihe kommen und wann im Speziellen genau ein bestimmter Thread zum Zug kommt. Damit gehen aber auch einige der typischen Probleme einher, die bei der Programmierung von *Multi-Threaded* Systemen gang und gäbe sind und für die es mittlerweile einige standardisierte Konzepte gibt, mit deren Hilfe sich diese Probleme umgehen oder vermeiden lassen.

Im Folgenden werden Probleme der *Multi-Threaded*-Programmierung vorgestellt und deren Lösung aufgezeigt. Hier werden grundlegende Standard-Konzepte erörtert, die in dieser Arbeit dazu verwendet wurden, um die Synchronisation innerhalb des *Multi-Threaded* System zu gewährleisten.

5.1 Potentielle Probleme bei *Multi-Threaded* Programmen

Probleme entstehen in *Multi-Threaded* Programmen durch die beliebige Ausführungsreihenfolge der einzelnen Threads. Hier sollen nun diese Probleme näher erörtert und die ensprechenden Lösungen demonstriert werden.

5.1.1 Überschreiben von gemeinsam genutzten Speicherbereichen

Ein Problem ist das Überschreiben von gemeinsam genutzten Speicherbereichen. Versuchen zwei oder mehr Threads auf einen gemeinsamen Speicherbereich zu schreiben, dann werden sie sich je nach Ausführungsreihenfolge diesen Speicherbereich gegenseitig überschreiben. In [Sta05] findet sich eine ausführliche Beschreibung des Problems. Versuchen etwa zwei Threads, jeweils den Wert aus dem Speicher zu lesen, zu verarbeiten und anschließend das Ergebnis wieder in den Speicher zu schreiben, kann es vorkommen, dass die beiden Leseoperationen als Erstes passieren, da das Betriebssystem direkt nach der Leseoperation des ersten Threads den Prozessor dem zweiten Thread zugeteilt hat, der dann seinerseits den Wert ausliest und damit rechnet und ihn schließlich in den Speicher transferiert. Teilt das Betriebssystem nun den Prozessor wieder dem ersten Thread zu, so arbeitet dieser noch mit dem alten Wert, obwohl mittlerweile schon ein aktuellerer Wert im Speicher steht und mit diesem weitere Berechnungen angestellt werden müssten. Vernachlässigt man das Problem, indem man sich nicht um die explizite Synchronisation der Threads durch Zugriffsregelung auf die Ressourcen kümmert, führt dies zu Endlosschleifen bzw. Programmabstürzen. Oft machen sich diese Probleme nicht unmittelbar bemerkbar, da sie nicht zwingend bei jeder Programmausführung auftreten müssen, da das *Scheduling* (die Ablaufplanung für die zur Ausführung kommenden Threads) von Ausführung zu Ausführung bzw. von System zu System variieren kann. Reproduzierbarkeit von Problemen und das Debugging ist daher bei *Multi-Threaded* Programmen kein triviales Unterfangen.

5.1.2 Speicherbereich-Zugriffsregelung mittels Semaphoren

Um das Problem des gegenseitigen Überschreibens von gemeinsam genutzten Speicherbereichen zu lösen bzw., allgemeiner definiert, um sicherzustellen, dass nur eine dafür vorgesehene Anzahl von Threads gleichzeitig kritischen Code zur Ausführung bringen darf, existiert der Mechanismus des Semaphor [Sta05]. Er regelt den Zugriff gemeinsam genutzter Daten nach der erlaubten Anzahl von Threads. Die jeweiligen Codeabschnitte der Threads, die für diese Zugriffe verantwortlich sind, nennt man auch *Critical Sections* [Dij65, Sta05]. Während *Counting Semaphors* [Sta05] einer bestimmten Anzahl von Threads gleichzeitig gestatten, eine gemeinsame Ressource zu nutzen, erlaubt der *Binary Semaphore* (auch *Mutex* [TS02, Sta05] genannt) die exklusive Nutzung einer Ressource durch genau einen Thread. Der exklusive Zugriff auf Ressourcen wird durch Versperren und Entsperren des *Mutex* erzielt. Somit lassen sich die Probleme des gegenseitigen Überschreibens von Speicherbereichen (auch *Race Conditions* [NBF96, Sta05] genannt) vermeiden. Zu beachten ist allerdings, dass Semaphoren nicht vor sogenannten *Deadlocks* [Sfi89, Sta05] schützen, das sind Zustände, in denen sich Threads gegenseitig blockieren, indem sie Ressourcen gesperrt halten und auf jeweils von anderen Threads gegenseitig gesperrte Ressourcen zusätzlich zugreifen wollen und somit wartend verharren, um weiterzuarbeiten und die von ihnen gesperrte Ressource wieder freizugeben. Keiner der Threads kommt dann je zum Abschluss seiner Berechnungen und das Programm hängt somit fest.

5.1.3 *Spinlocks*: *Busy Waiting* als Rechenzeitfresser

In vielen Programmen ist es notwendig, dass man Threads auf eine Zustandsänderung warten lässt, damit diese korrekt weiterarbeiten können. Man spricht in diesem Zusammenhang auch von einer Sperre oder Barriere, an welcher der Thread sozusagen wartet, bis sie aufgehoben ist. Prüft man etwa mittels einer *while*-Schleife den Zustand einer gemeinsam genutzten Ressource, welche sich also jederzeit ändern könnte, so verbringt der Thread die komplette Zeit seiner Zeitscheibe damit, diese Variable zu prüfen. Freilich ließe sich die Prozessorzeit besser nutzen, indem andere Threads zur Ausführung kämen, die ihre Aufgaben erledigen könnten. Speziell in Systemen mit nur einem Prozessorkern ist das wiederholte Warten auf eine Zustandsänderung der Variablen völlig sinnlos, da während der Ausführung des Threads ja kein anderer Thread die Variable ändern könnte. In Mehrkernprozessor-Systemen erhält man immerhin die kürzest mögliche Reaktionszeit, aber dennoch ist das in den meisten Fällen überflüssige Überprüfen der Schleifenbedingung unerwünscht. Zur Lösung dieses Problems existiert das Konzept des *Monitors* [Sta05], das im Folgenden beschrieben werden soll.

5.1.4 *Monitors* zur Lösung des *Spinlock*-Problems

Mit Hilfe des Konzeptes des *Monitors* [Sta05] lässt sich das Problem des *Busy Waitings* beheben. Dabei wird nach Sperren des für die zu prüfende Variable zugehörigen *Mutex* diese geprüft, und erfüllt sie nicht die Bedingung zum Überspringen der Barriere, wird ein sogenanntes *Conditional Wait* [TS02] (auch *Conditional Variables* [NBF96]) vollzogen, was einem Warten des Threads auf ein Verständigungssignal gleichkommt. Dabei wird der *Mutex* automatisch wieder entsperrt. In den anderen Threads, die nach dem Sperren des *Mutex* diese Variable ändern, wird nach dem Ändern ein Signal emittiert und der *Mutex* wieder freigegeben. Dieses Signal weckt den wartenden Thread auf, der nun nach dem automatischen Sperren des *Mutex* seine Bedingung erneut prüft. Erlaubt der Variablenwert das Überspringen der Barriere, muss noch der *Mutex* wieder entsperrt werden und der Weiterverarbeitung der ausständigen Aufgaben innerhalb des Threads steht nichts mehr im Wege, andernfalls wartet der Thread erneut. Über ein spezielles Broadcast-Signal lassen sich auch Schranken für mehrere Threads implementieren, die dann alle gleichzeitig durch den Broadcast des Signals aufgeweckt werden.

5.1.5 Deadlock: Blockierende Threads

Ein Problem, das die ganze Programmausführung lahmlegen kann, ist das sogenannte *Deadlock*. Ein *Deadlock* ist ein permanenter Zustand des Blockierens einer Reihe von Threads, die um den Zugriff auf gemeinsame Ressourcen konkurrieren [Sta05]. Die Threads befinden sich in einem *Deadlock*-Zustand, wenn jeder Thread auf ein Ereignis wartet, bei dem eine angeforderte Ressource frei wird, das Ereignis aber von einem anderen Thread ausgelöst werden muss, der selbst blockiert ist (der selbst auf eine andere blockierte Ressource wartet). Da keines der Ereignisse, die zur Lösung der Patt-Stellung nötig sind, ausgelöst werden kann (da die entsprechenden Threads blockiert sind), ist dieser Zustand des Blockierens permanent.

Notwendige (1-3) und hinreichende (1-4) Bedingungen für eine *Deadlock*-Situation sind [Sta05] (Bedingung 4 ist eine potentielle Konsequenz aus Bedingung 1-3):

1. Kollidierende Interessen mehrerer Threads für mindestens eine über Synchronisationsmechanismen (z.B. *Mutual Exclusion* durch Semaphoren) gesperrte Ressource
2. Auf eine Ressource wartende Threads halten selbst gesperrte Ressourcen gesperrt.
3. Das Erzwingen der Freigabe von gesperrten Ressourcen ist von außen nicht möglich.

4. Zyklisches Warten auf gesperrte Ressourcen: Es existiert eine geschlossene Kette von Threads, sodass jeder Thread zumindest eine Ressource gesperrt hält, die vom nächsten Thread in der Kette benötigt wird.

Bei dem in dieser Masterarbeit entstandenen System werden *Deadlock*-Situationen verhindert, indem sichergestellt wird, dass die vierte (hinreichende) Bedingung für eine *Deadlock*-Situation nicht erfüllt ist. Die von mehreren Threads gemeinsam genutzten Ressourcen sind Queues, die jeweils nur von einem Thread gefüllt werden dürfen und von denen jeweils nur ein (anderer) Thread Elemente entnehmen darf. Die Kommunikation zwischen verschiedenen Threads wird über verschiedene Queues (siehe Abschnitt 7.3 für die Beschreibung der Queue-Datenstruktur) bewerkstelligt und dabei existiert keine zyklische Kette von Threads, die Ressourcen benötigen, aber gesperrt halten (Bedingung 4).

5.1.6 Starvation: Verhungernde Threads

Das Problem der *Starvation* tritt auf, wenn ein lauffähiger Thread vom *Scheduler* auf unbestimmte Zeit übersehen wird [Sta05] (der Thread also nie zur Ausführung kommt, obwohl er seine Ausführung fortsetzen könnte). Diese Situation passiert dann, wenn der Scheduler dem Thread aufgrund seiner Priorität andere Threads vorzieht.
So ist ein mögliches Szenario, dass drei Threads periodischen Zugriff auf eine Ressource benötigen. Angenommen, der erste Thread erhält Zugriff auf die Ressource und der zweite und dritte Thread sind somit blockiert. Entsperrt der erste Thread die Ressource, könnten beide wartenden Threads zur Ausführung kommen. Angenommen, der Scheduler ermöglicht nun dem zweiten Thread Zugriff auf die Ressource und somit das Sperren der Ressource, blockiert der dritte Thread weiterhin. In der Zwischenzeit benötigt der erste Thread erneut die Ressource, dann wäre es beim nächsten Entsperren der Ressource durch den zweiten Thread denkbar, dass der Scheduler erneut dem dritten Thread den Zugriff versagt, indem er dem ersten Thread gestattet, auf die Ressource zuzugreifen. Wiederholt sich das Spiel anschließend fortlaufend, indem immer abwechselnd der erste und dritte Thread Zugriff erhalten, so leidet der zweite Thread unter *Starvation*.

Bei dem in dieser Masterarbeit entstandenen System existiert das Problem der *Starvation* nicht, da immer nur zwei Threads um eine gemeinsam genutzte Ressource konkurrieren und die Threads über den Mechanismus des *Monitors* (als Teil der implementierten Queue-Datenstruktur - siehe Abschnitt 7.3) miteinander synchronisiert sind und so über das Emittieren eines Signals die Threads abwechselnd zur Ausführung kommen.

5.2 Formen der Arbeitsteilung in *Multi-Threaded* Programmen

Es existieren in *Multi-Threaded* Systemen mehrere Formen der Arbeitsteilung, deren Aufbau und Funktionsweise im Folgenden kurz beschrieben werden sollen.

5.2.1 Das *Boss/Worker Thread-Model*

Beim *Boss/Worker Thread-Model* [NBF96] (auch *Delegation Thread-Model* [HH03]) delegiert und koordiniert ein Manager-Thread die Arbeit mehrerer Arbeiter-*Threads* (Abbildung 5.1). Der Manager-Thread stellt die Kontrollinstanz dar. Er nimmt sämtlichen Input entgegen und verteilt dementsprechend anstehende Arbeit auf die Arbeiter-Threads.

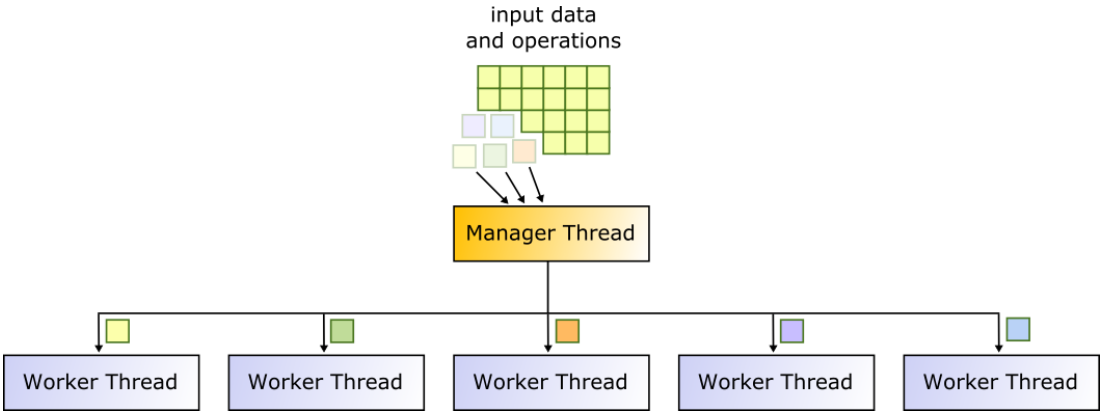


Abbildung 5.1: Das *Boss/Worker Thread-Model* (auch *Delegation Thread-Model*)

5.2.2 Das *Peer Thread-Model*

Das *Peer Thread-Model* [NBF96] (auch *Peer-to-Peer Thread-Model* [HH03]) basiert auf gleichberechtigten Threads, die sich selbstständig koordinieren (siehe Abbildung 5.2). Es muss sich lediglich zu Beginn des Arbeitsprozesses ein Thread um die Erzeugung der anderen Threads kümmern, sobald die Initialisierungsphase abgeschlossen ist, reiht er sich in die Liste der Threads ein, um dann selber als gleichberechtigter Arbeiter am Arbeitsprozess teilzunehmen.

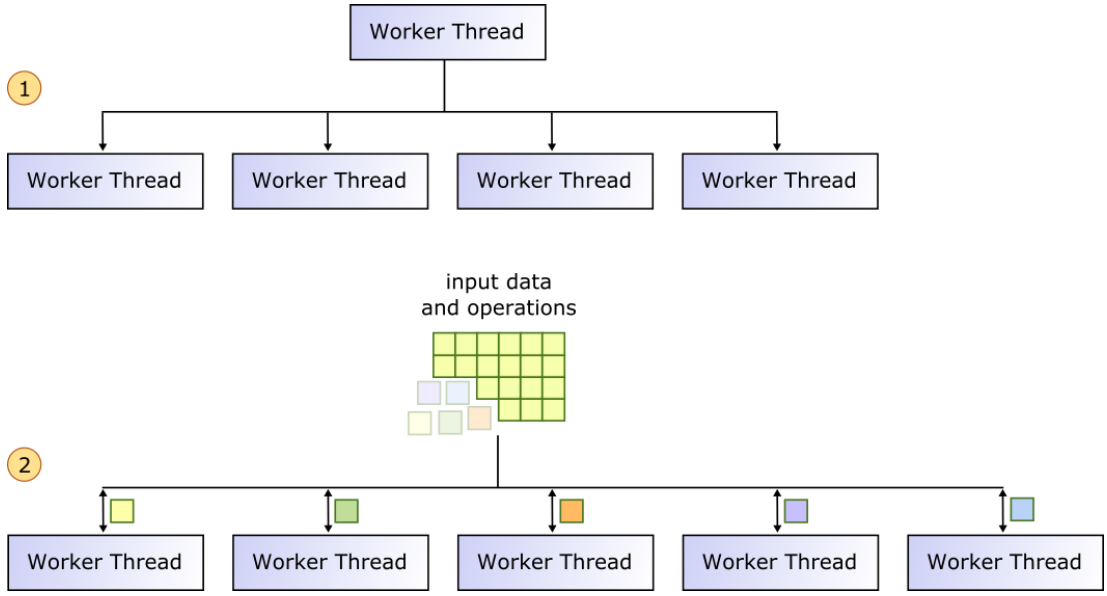


Abbildung 5.2: Das *Peer Thread-Model* (auch *Peer-to-Peer Thread-Model*)

5.2.3 Das *Pipeline Thread-Model*

Beim *Pipeline Thread-Model* [NBF96, HH03] werden die zu erledigenden Aufgaben auf die einzelnen Threads aufgeteilt, die dann im Fließbandverfahren die einzelnen Verarbeitungsschritte nacheinander ausführen (siehe Abbildung 5.3). Zwecks Erhöhung der Leistung sollten alle Arbeiter immer genug Arbeit haben, um auch wirklich gleichzeitig arbeiten zu können, was durch Vorkehrungen und sorgfältige Programmplanung erreicht werden kann. Zusätzliche Informationen finden sich in [But97].

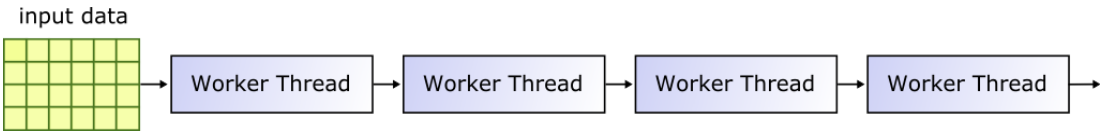


Abbildung 5.3: Das *Pipeline Thread-Model*

5.2.4 Das *Producer-Consumer Thread-Model*

Im *Producer-Consumer Thread-Model* [HH03] (auch *Client/Server Thread-Model* [But97]) fordert ein Client vom Server an, eine bestimmte Operation durchzuführen. Der Server führt diese Operation unabhängig vom Client durch, der Client kann also auf den Server warten oder auch zwischenzeitlich andere Aufgaben durchführen, um zu einem späteren

Zeitpunkt das Ergebnis abzufragen (siehe Abbildung 5.4). Dieses Verfahren wird eingesetzt, um gemeinsame Ressourcen verschiedenen Clients zur Verfügung zu stellen und den Zugriff auf diese Ressource zu synchronisieren.

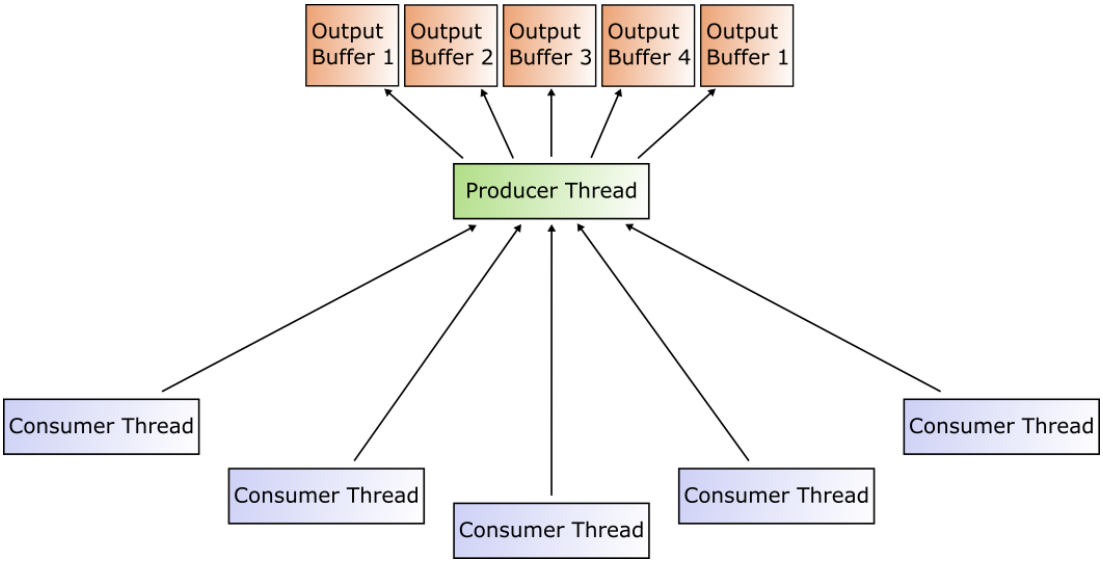


Abbildung 5.4: Das *Producer-Consumer Thread-Model* (*Client/Server Thread-Model*)

Kapitel 6

Definitionen und Rahmenbedingungen für das *Object Tracking* System

In diesem Kapitel werden Definitionen und Algorithmen, auf denen das *Object Tracking* System basiert, vorgestellt. Zuerst wird die Aufgabenstellung definiert, dann werden die einzelnen Komponenten eines *Tracking-Systems* definiert und erläutert. Danach folgt eine Beschreibung der verwendeten Algorithmen, die in den verschiedenen Systemkomponenten zum Einsatz kommen.

6.1 Definition der Aufgabenstellung

Zunächst wird definiert, welche Teile des *Object Tracking* Systems in welcher Detailtiefe umgesetzt werden sollen und welche zusätzlichen Anforderungen an das System gestellt werden.
Als *Tracking* System sei ein automatisiertes System zu verstehen, das ausgehend von den Einzelbildern eines Videodatenstroms, aufgenommen durch eine statische Kamera, über mehrere Verarbeitungsschritte bis zur Ausgabe der im Bild sichtbaren beweglichen Objekte[1] als getrackte Bildregionen gelangt. Etwaige darauf basierende Verarbeitungsschritte (als Beispiel sei Verhaltensanalyse von getrackten Personen erwähnt) sind zwar ein logischer weiterer Schritt von Überwachungssystemen, sollen hier aber explizit nicht als Teil des *Tracking* Systems verstanden werden.
In dieser Masterarbeit sollen alle Komponenten eines solchen *Tracking* Systems umgesetzt werden, wobei der Schwerpunkt auf den low-level Komponenten des Systems liegen soll,

[1] als ein zu trackendes Objekt wird in dieser Arbeit eine rechteckige Pixelregion verstanden, die durch Bewegung eines realen Objekts (wie etwa eine Person oder ein Fahrzeug), aufgenommen durch die Videokamera, im Videodatenstrom segmentiert und extrahiert werden kann.

das sind speziell die Bewegungserkennung (sie stellt fest, in welchen Bereichen - genauer gesagt, bei welchen Pixeln - im Bild Bewegung stattfindet, sich also Objekte bewegen) und die Bewegungsverfolgung (das *Objekt Tracking*).
Es existieren in einem *Tracking* System aber noch weitere Verarbeitungsschritte, wenn man von *Tracking* Systemen spricht, die in wissenschaftlichen Arbeiten wenig behandelt werden, die aber mindestens genauso wichtig sind.
Die Datenerfassung ist speziell bei Anwendungen, die im Realbetrieb laufen müssen, kein triviales Unterfangen, da im Falle von Fehlfunktionen oder Ausfällen des Aufnahmesystems die Ergebnisse verfälscht werden und in einem im Realbetrieb laufenden System somit das regelmäßige fehlerfreie Auslesen und Übertragen von Kamerabildern gewährleistet sein muss.
In dieser Masterarbeit soll dieser Aspekt allerdings nicht vertiefend behandelt werden, stattdessen kommen stellvertretend bereits aufgenommene Videosequenzen zum Einsatz.

Zwischen der Bewegungserkennung und der Bewegungsverfolgung liegt ein weiterer Verarbeitungsschritt. Es müssen nämlich die zu trackenden Bildregionen erst einmal ermittelt werden. Die Bewegungserkennung sagt für jedes Pixel nur aus, ob für dieses Pixel Bewegung detektiert wurde. Nun muss aber festgestellt werden, welche Pixelansammlungen Regionen definieren, die den sich bewegenden Objekten entsprechen. Diese Regionen spezifizieren die beweglichen Objekte im Bild. Allerdings muss beachtet werden, dass im laufenden Prozess des *Trackings* das *Tracking* einiger dieser Regionen bereits in vorangegangenen untersuchten Bildern gestartet wurde und sie somit nicht mehr als neue Objekte aufgefasst werden dürfen. Nur die noch nicht getrackten Objekte dürfen als neu hinzukommende Objekte erkannt werden.
Schließlich müssen auch noch verlorene Objekte (das sind Objekte, die über mehrere Bilder hinweg beim *Tracking* nicht mehr wiedergefunden werden konnten) aus der Liste der zu trackenden Objekte entfernt werden.
Diese Punkte sind Teil der in dieser Masterarbeit als *High Level Data Association* bezeichneten Komponente, die Zusammenhänge und Korrespondenzen von Bildregionen in verschiedenen Bildern herstellt. Zwar stellt auch die *Tracking*-Komponente Objektzusammenhänge dar, allerdings stellt diese Art der in dieser Arbeit als *(Low Level) Frame To Frame Data Association* bezeichneten Komponente nur einfache Zusammenhänge dar. Im Gegensatz dazu wird bei der *High Level Data Association* über Heuristiken und spezielle Regeln festgestellt, wo sich Regionen aus vorangegangenen Bildern im aktuellen Bild befinden könnten, und sie verbessern die Ergebnisse des *Tracking* Algorithmus durch zusätzliche Schätzungen und Korrespondenzberechnungen. So lassen sich etwa Überde-

ckungsprobleme, die in den aufgenommenen Kamerabildern eines *Tracking* Systems häufig vorkommen, in einigen Fällen lösen - in [CRM03] kommen etwa Kalman-Filter zum Einsatz. Für detaillierte Information bezüglich *Data Association* wird auf [BSF88, RH01] verwiesen.
Im vorgestellten System wird die *High Level Data Association* nur bis zu einem gewissen Punkt betrieben (Regionsvergleich anhand von Ähnlichkeiten und örtlichem Bezug). *Prediction Filter* und ähnliche Verfahren werden nicht implementiert, da der Schwerpunkt der Arbeit auf der Performancesteigerung des Systems liegt. Eine Erweiterung um diese Funktionalitäten wäre aber ohne weiteres möglich (Kalman-Filter würden die Echtzeitfähigkeit des *Tracking* Systems nicht gefährden).
Als Track-Management werden im Folgenden das Initialisieren von neu zu trackenden Objekten, Suchen bereits getrackter Objekte und Löschen von verwaisten Objekten verstanden. Auch die *High Level Data Association* hängt mit diesen Aufgaben zusammen und wird deshalb als Teil des Track-Managements verstanden.

Das System soll echtzeitfähig sein, was im konkreten Fall so definiert sei, dass das Verarbeiten von PAL Video Material (768×576 bzw. 720×576 Bildpunkte á 3 Farbkanälen bei einer Wiedergabefrequenz von 25 Hertz) in Echtzeit möglich sei.
Das Verarbeiten von 25 Bildern pro Sekunde ist deshalb nicht irrelevant, da bewegte Objekte in kürzerer Zeit kürzere Distanzen zurücklegen und sich ihre Position in aufeinanderfolgenden Bildern somit auch örtlich weniger stark ändert, was einerseits die nötigen Iterationsschritte bis zur Konvergenz von *Tracking* Algorithmen wie *Mean Shift* verringern kann und andererseits das Auffinden der Objekte leichter und in einigen Fällen sogar robuster machen kann (etwa beim *Mean Shift* Algorithmus bei sich schnell bewegenden Objekten: Je näher die aktuelle Position des Objekts der Position aus dem letzten Bild ist, desto wahrscheinlicher fällt sie in das Kernelfenster und wird somit wiedergefunden).
Die inkludierten Verarbeitungsschritte sind Datenerfassung (Laden des Videos inklusive Decodierung), Bewegungserkennung samt Schatten- und Glanzlichterkennung inklusive deren Entfernung, Erkennen zusammenhängender Regionen im Bild (*Connected Components Analysis*), *Tracking* der Regionen, einfache *Data Association* (Verwalten der getrackten Objekte inklusive Initialisierung neuer Kandidaten und Löschen verwaister Objekte) und letztendlich auch die Ausgabe in grafischer Form über ein GUI (*Graphical User Interface*).

6.2 Komponenten des *Tracking* Systems

Das *Tracking* System, das in dieser Arbeit entwickelt wird, besteht aus mehreren Komponenten, die jeweils für die Erledigung einer ganz speziellen Aufgabe konzipiert werden. Dabei fungiert jede Komponente als Instanz, die nach Vollendung eines Arbeitsschrittes die von ihr vollzogenen Berechnungen an die nächste Komponente weiterreicht, die dann wiederum ihrerseits darauf aufbauende Arbeiten leistet.
Es handelt sich also um ein Pipeline-Konzept (Arbeit nach Vorbild von Fließbandarbeit), und es bietet sich an bzw. es ist eine naheliegende Überlegung, dieses Pipeline-Konzept auch im Kontext der *Multi-Threaded*-Programmierung anzuwenden (genau dieses Konzept wird nämlich auch für das *Multi-Threading* in dieser Arbeit verwendet, worauf später in der Arbeit genauer eingegangen wird).
In einem *Tracking* System, das als Input Bildfolgen verwendet, ist nicht bereits zum Start des *Tracking*-Programms die gesamte Datenmenge verfügbar, sondern diese wird für das System erst nach und nach verfügbar gemacht, etwa durch die aufgenommenen Einzelbilder einer Überwachungskamera. Somit ist es Aufgabe des Systems, während der gesamten Programmausführung alle notwendigen Informationen im Speicher resident zu behalten,

um neue Daten auch im Kontext der bereits gesammelten Informationen bewerten zu können.
Bei neu verfügbaren Daten startet die erste Komponente, sobald sie mit allen vorherigen Berechnungsaufgaben fertig ist, damit, diese neuen Daten zu verarbeiten, und reicht nach Abschluss dieser Berechnungen die Daten über eine passende Schnittstelle an die nächste Instanz weiter, die nun ihrerseits ihre Aufgaben startet. Dieses Spiel wiederholt sich bis zur letzten Komponente, die schließlich mit dem fertigen Ergebnis aufwarten kann, das dann visualisiert oder an eine Kontroll- bzw. Überwachungsinstanz weitergegeben werden kann.

Es folgt eine Beschreibung der verschiedenen Komponenten. Zum besseren Verständnis des Zusammenspiels der einzelnen Komponenten ist in Abbildung 6.1 der Verarbeitungsablauf des in dieser Masterarbeit implementierten Systems skizziert.

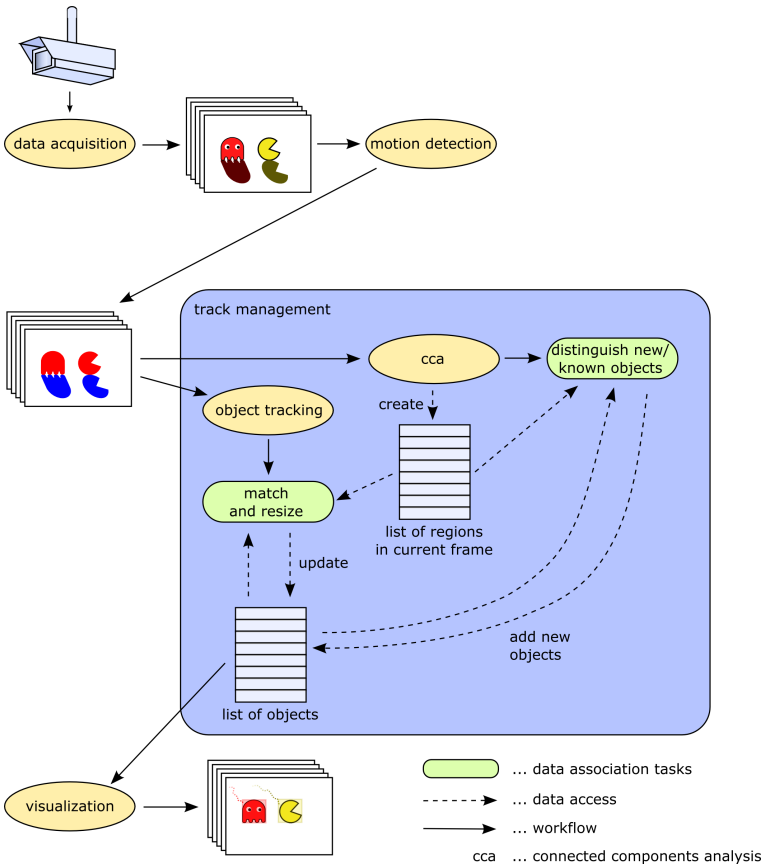


Abbildung 6.1: Die Zusammenspiel der Systemkomponenten

Ausgehend von einer aufgenommenen Bildfolge werden Bewegungserkennung, *Object Tracking* und Visualisierung der Ergebnisse berechnet. Das *Object Tracking* wird durch die zusätzlichen Verarbeitungsschritte von *Data Association* und *Connected Components Analysis* unterstützt. Über eine Liste von Regionen im aktuellen Bild und bereits getrackten Objekten in vorhergehenden Bildern wird ein Abgleich durchgeführt, um neue Objekte zu registrieren und das *Tracking* bekannter Objekte weiter zu verbessern.

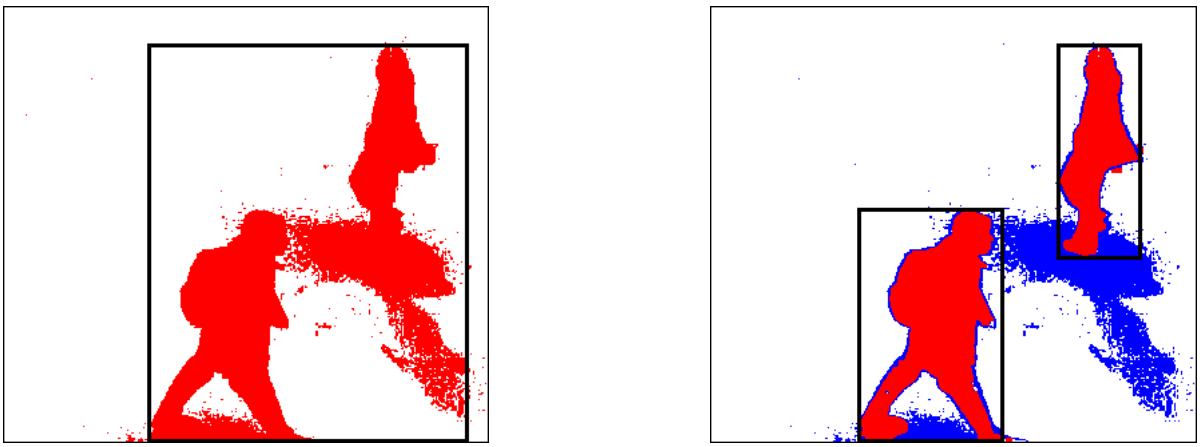
Datenerfassungskomponente

Um neu verfügbare Daten weiterverarbeiten zu können, müssen diese zuerst von der Datenerfassungskomponente (z.B. einer Überwachungskamera) aufgenommen werden. Diese Komponente regelt den Zugriff auf die Datenquelle und muss bei möglichen Störungen und Ausfällen der Datenübertragung Konsistenz wahren und gegebenenfalls andere Komponenten in Kenntnis setzen. Eine Störung bzw. der Ausfall der Datenübertragung resultiert unweigerlich in beeinträchtigten oder falschen *Tracking*-Ergebnissen, und nichts wäre fataler, als vom Begehen von Fehlern nicht einmal etwas zu wissen. Im günstigsten Fall verliert man getrackte Objekte temporär oder auch langfristig, aber man weiß zumindest, dass man den Daten des Systems nicht mehr vertrauen darf, im schlechtesten Fall bleiben aber die Fehler verborgen und führen so zu Missverständnissen und falschen Schlussfolgerungen. Daher müssen im Falle von Übertragungsfehlern und Datenausfällen geeignete Gegenmaßnahmen eingeleitet werden (z.B. wäre, sobald wieder gültige Daten eintreffen, als Recovery-Maßnahme ein spezieller *Data Association* Algorithmus denkbar, der versucht, die zuletzt bekannten Positionen der getrackten Objekte auf die neuen Regionen des Bildes abzubilden und im Falle des Scheiterns die Objekte löscht).

Bewegungserkennungskomponente

Die Bewegungserkennungskomponente oder *Motion Detection* Komponente bestimmt die Pixel im aktuell bearbeiteten Videobild, welche zu potentiell sich bewegenden Objekten gehören. Es gibt mehrere Verfahren, die enstanden sind, um dieses Problem zu lösen.
Frame Differencing-Algorithmen sind die simplesten Algorithmen dieser Art (die Funktionsweise wird in [TKBM99] beschrieben).
Höherentwickelte Verfahren basieren auf Hintergrundmodellen. Hintergrundmodelle sagen das Auftreten von Helligkeits- bzw. Farbwerten von Pixeln vorher und lassen so die Klassifikation von Pixeln als Vorder- bzw. Hintergrundpixel zu.
Die Mehrzahl der Verfahren versucht mittels statistischer Verteilungen die Pixelwerte zu modellieren, wobei die Vereinfachung getroffen wird, dass die einzelnen Pixel unabhängigen Verteilungen folgen. Beispielsweise wird mit Gauss-Verteilungen das Auftreten von Helligkeits- bzw. Farbwerten vorhergesagt, und durch Vergleich mit den Parametern der Verteilung (Mittelwert und Varianz) lassen sich Aussagen über Pixel insofern treffen, als man den zu untersuchenden Pixel als Vorder- oder Hintergrund klassifizieren kann [FM99, JDWR00, WC01, CBT01, LBC02, WFSM02]. Durch Erweiterung der Algorithmen kann festgestellt werden, ob Pixel, bei denen Bewegung detektiert wurde, zusätzlich darauf überprüft werden sollen, ob diese festgestellte Bewegung aus eventuellem

Schattenwurf oder Reflexionen resultieren könnte [HHD99, EHD00, CGP+01, WBHK06]. Das Detektieren von Schatten und Reflexionen ist deshalb wichtig, da sonst Objektschatten und -Reflexionen als zum Objekt gehörig interpretiert werden und so eine deutlich größere Region bei der *Connected Components Analysis* erkannt wird als die Region, die durch das tatsächliche Objekt definiert ist. In ungünstigen Fällen können so auch zwei Objekte als ein großes Objekt detektiert werden (siehe Abbildung 6.2).



(ohne Schatten-Detektion) (mit Schatten-Detektion)

Abbildung 6.2: Schatten- und Reflexionsdetektion verbessern die Regionserkennung

Es gibt auch Verfahren, die mehr als eine Gauss-Verteilung verwenden, um einen Pixel zu beschreiben [SG99]. In [HHD00] wird eine bimodale Verteilung durch Ermittlung von Minimum, Maximum von Helligkeitswerten und Differenz von Helligkeitswerten aufeinanderfolgender Bilder konstruiert und für die Klassifikationstests herangezogen (für Details wird auf die Arbeit verwiesen). Fällt einer dieser Tests positiv aus, so kann der Pixel wieder nachträglich als Hintergrund markiert werden. Ergebnis dieses Verarbeitungsschrittes ist ein Bild, in dem nur jene Bereiche (genauer gesagt Pixel) im Bild Farbwerte ungleich schwarz erhalten, bei denen der Test auf Bewegung positiv ausgefallen ist. Das Hintergrundmodell wird laufend nachgeführt, um eine aktuelle Schätzung gewährleisten zu können.
Die Hintergrundmodellierung kann auch als Filterproblem aufgefasst werden, etwa durch *Wiener Filtering* [TKBM99] oder *Kalman Filtering* [RMK95].
Ein Verfahren, das die *Bayes Decision Rule* verwendet, um die Klassifikation zu ermitteln, wird in [LHGT03] beschrieben.

Tracking-Komponente

Die *Tracking*-Komponente trackt alle bisher bekannten Objekte, das heißt, sie sucht ausgehend von der zuletzt bekannten Position der Objekte diese im neuen Bild.
Hier sollen *Tracking* Algorithmen, die markante Features tracken, und solche, die ganze Bildregionen tracken, unterschieden werden:
Feature Tracker (z.B. [TK91]) versuchen, markante Punkte im Bild (z.B: Ecken) in aufeinanderfolgenden Bildern wiederzufinden. Die Zuordnung von Features zu Objekten ist nicht Teil des *Trackers* und muss durch zusätzliche Algorithmen gewährleistet werden.
Region Tracker (*Blob Tracking*) [CRM03, Bra98] versuchen, eine Region eines Bildes in darauffolgenden Bildern wiederzufinden.
Diese Art der *Frame To Frame Data Association* ist als *Low Level* Prozess alleine nicht in der Lage, Verdeckungsprobleme zu lösen, was den Einsatz höher entwickelter *High Level* Prozesse zur *Data Association* erforderlich macht.
In dieser Arbeit wird ein *Region Tracker* implementiert (siehe Abschnitt 6.3.2), der auf den Ergebnissen der Bewegungserkennung und anschließender Regionsbestimmung mittels *Connected Components Analysis* aufsetzt und die einzelnen Regionen trackt.
Der letztendlich im entstandenen *Tracking* System verwendete *Region Tracking* Algorithmus ist austauschbar, solange er die an ihn gestellten Anforderungen (Auffinden von Regionen in verschiedenen Bildern in einer Pixelumgebung einer festgelegten Größe) erfüllen kann. Dies wird durch Implementierung von zwei verschiedenen *Region Trackern* überprüft, die gegeneinander ausgetauscht werden können (siehe Abschnitt 6.3.2 ab Seite 71 für die Algorithmus-Beschreibung und Abschnitt 7.5 für Implementierungsdetails für den *Mean Shift Tracker*, basierend auf Farbhistogrammen, sowie den *Mean Shift Tracker*, basierend auf Integralbildern der Differenzbilder).

Track-Manager-Komponente

Als *Track-Manager*-Komponente wird in dieser Masterarbeit die Komponente verstanden, die die Verbindung zwischen der Bewegungserkennungs- und der *Tracking*-Komponente herstellt. Diese Komponente kümmert sich um die *High Level Data Association*. Sie ist verantwortlich dafür, mittels *Connected Components Analysis* Zusammenhänge von Pixeln herzustellen und zusammenhängende Pixel (eine Menge von benachbarten Pixeln) zu Regionen zusammenzufassen. Jede dieser gefundenen Regionen entspricht einem Objekt oder mehreren, sich teilweise überdeckenden oder zumindest unmittelbar aneinander angrenzenden Objekten - bei fehlerhaften Ergebnissen des vorangegangenen Bewegungserkennungsschrittes (je mehr Pixel falsch klassifiziert wurden, desto schlechter ist das

Ergebnis) können aber auch Fehldetektionen als Regionen interpretiert werden.
Ob neue Regionen nur in sogenannten Portalen oder im gesamten Bild gesucht werden, hängt von der Implementierung ab. Die ermittelten Regionen entsprechen den zu trackenden Objekten.
Die Track-Manager-Komponente verwaltet die Liste von zu trackenden Objekten und erzeugt neue zu trackende Objekte, falls diese nicht bereits in der Liste der zu trackenden Objekte vorhanden sind. Sie ist auch dafür verantwortlich, dass Objekte, die in einer festgelegten Zeit nicht mehr erfolgreich wiedergefunden werden konnten, aus der Liste der zu trackenden Objekte gelöscht werden.
Dies wäre auch ein passender Zeitpunkt, höher entwickelte *data-association* Algorithmen in die Verarbeitungskette einzubinden.
Der Track-Manager ist also die Komponente, die alle wichtigen Informationen über die gerade getrackten Objekte besitzt.

6.3 Auswahl der Algorithmen

Nun werden die verwendeten Algorithmen, die in dem hier vorgestellten *Tracking* System zum Einsatz kommen, vorgestellt. Für jede Aufgabe kommen bestimmte Algorithmen zum Einsatz, die auf Echtzeitfähigkeit untersucht und miteinander verglichen wurden.

6.3.1 Algorithmus zur Bewegungs-, Schatten- und Reflexionenerkennung

Für die Bewegungserkennung und die Bestimmung von Schattenwurf und Reflexionen kommt der Algorithmus aus [WBHK06] zum Einsatz. Dieser Algorithmus arbeitet im IHLS-Farbraum (*Improved Hue, Luminance and Saturation Space*), das ist eine Erweiterung des bekannten HLS-Farbraumes, der statt durch Rot-, Grün- und Blauinformationen die Pixelwerte in Form von Helligkeit, Sättigung und Farbton festlegt. Der IHLS-Farbraum stellt insofern eine Erweiterung des HLS-Farbraumes dar, als er die Farbtoninformation zusätzlich durch die Sättigungsinformation gewichtet, was den positiven Effekt bedingt, dass Farbtonwerten nur dann größere Bedeutung beigemessen wird, wenn sie eine stärkere Sättigung aufweisen. Auf diese Art und Weise kann Fehlern beim Farbton, die durch Rauschen verursacht wurden, besser entgegengewirkt werden. Nimmt man nämlich nur die Farbtoninformation zur Hand, so erhält man auch für Pixel, die z.B. im Schatten liegen und kaum Sättigung aufweisen, einen Farbtonwert, der dann in weiteren Berechnungen in gleichem Ausmaß in die Berechnungen einfließt wie Pixel, in deren Farbtonwert

man wesentlich mehr Vertrauen hat in Bezug auf die Korrektheit des Messwertes. Die Umrechnung von RGB-Farbwerten in IHLS-Farbwerte erfolgt folgendermaßen:

$$s = max\,(R,G,B) - min\,(R,G,B) \tag{6.1}$$

beliebige Helligkeitsumrechnung, z.B: $$y = 0.2125R + 0.715G + 0.0721B \tag{6.2}$$

$$cr_x = R - \frac{G+B}{2}\ ,\ cr_y = \frac{\sqrt{3}}{2}\,(B-G) \tag{6.3}$$

$$cr = \sqrt{cr_x^2 + cr_y^2} \tag{6.4}$$

$$\theta^H = \begin{cases} undefined & if\ cr = 0 \\ \arccos(\frac{cr_x}{cr}) & elseif\ cr_y <= 0 \\ 360 \circ - \arccos(\frac{cr_x}{cr}) & else \end{cases} \tag{6.5}$$

mit cr_x, cr_y als Farbton-Koordinaten und $cr \in [0,1]$ als Chrominanz, sowie $s \in [0,1]$

Da bei jeder beliebigen Farbwert-Beobachtung $\cos(\theta^H) = \frac{cr_x}{cr}$ und $\sin(\theta^H) = -\frac{cr_y}{cr}$ gilt, sind keine kostspieligen trigonometrischen Funktionen zur Berechnung nötig.

Der vorgestellte Algorithmus funktioniert unter Zuhilfenahme eines Hintergrundmodells, wie es in vielen *Motion Detection*-Algorithmen, die für statische Kameraaufnahmen entwickelt wurden, zur Anwendung kommt (siehe Abschnitt 6.2). Dabei werden an jeder Pixelposition des Kamerabildes statistische Kenngrößen wie Mittelwert, Standardabweichung, Varianz oder ähnliche Größen gespeichert und - sobald neue Information verfügbar ist - nachgeführt, um so durch Vergleich mit neu ankommenden Pixelwerten aus aktuellen Kamerabildern durch Verwendung festgelegter Grenzwerte Änderungen in der Farbinformation eines Pixels erkennen zu können. So erkannte Änderungen werden als Bewegung definiert. Mit weiteren Vergleichstests von bestimmten Pixelattributen bezüglich bestimmter Grenzwerte lassen sich auch Schatten- und Reflexionsregeln finden, die brauchbare Ergebnisse liefern. Die in [WBHK06] definierten Berechnungsregeln lauten (alle folgenden Berechnungen werden pixelweise durchgeführt und es seien zur Initialisierung des

Hintergrundmodells n Bilder bekannt):

$$C_s = \sum_{i=1}^{n} s_i * \cos\theta_i^H,\ S_s = \sum_{i=1}^{n} s_i * \sin\theta_i^H \quad \text{hier wird also mit der Sättigung gewichtet} \tag{6.6}$$

$$\bar{R_n} = \frac{\sqrt{C_s^2 + S_s^2}}{n} \quad \bar{R_n} \text{ ist die Durchschnittslänge des Chrominanz-Vektors} \tag{6.7}$$

$$\bar{c_n} = \left(\frac{C_s}{n}, \frac{S_s}{n}\right) \quad \bar{c_n} \text{ ist der Durchschnitts-Chrominanz-Vektor} \tag{6.8}$$

Die Durchschnittshelligkeitswerte μ_y und die Standardabweichung σ_y der Helligkeitswerte werden aus den n Bildern berechnet, weiters wird $\bar{c_n}$ berechnet. Zusätzlich wird die durschnittliche euklidische Distanz σ_D zwischen $\bar{c_n}$ und den beobachteten Chrominanz-Vektoren berechnet. Es müssen also die n Bilder ein zweites Mal durchlaufen werden, um diese Berechnungen ausführen zu können, da $\bar{c_n}$ ja bereits bei der Berechnung von σ_D bekannt sein muss.

Nun kann mit folgenden beiden Regeln Bewegung detektiert werden (y_o sei die aktuell beobachtete Helligkeit, s_o die Sättigung und h_o der aktuell beobachtete Chrominanz-Vektor):

$$|(y_o - \mu_y)| > \alpha * \sigma_y \quad \vee \quad \|\bar{c_n} - s_o * h_o\| > \alpha * \sigma_D \tag{6.9}$$

Schatten wird detektiert, wenn eine der folgenden drei Bedingungen erfüllt ist:

$$y_o < \mu_y \quad \wedge \quad |y_o - \mu_y| < \beta * \mu_y \tag{6.10}$$

$$s_o - \bar{R_n} < \tau_{ds} \tag{6.11}$$

$$\left\|h_o \bar{R_n} - \bar{c_n}\right\| < \tau_h \tag{6.12}$$

α, β, τ_{ds} und τ_h sind Parameter, gebräuchliche Werte finden sich in [WBHK06].

Schatten werden sowohl durch Testen der Helligkeit, Untersuchen der Sättigung als auch durch Untersuchen des Chrominanz-Vektors ermittelt. Highlights können analog durch Anpassung der Regeln 6.10, 6.11 und 6.12 für dementsprechend erhöhte Werte bestimmt werden.
Das Hintergrundmodell wird schließlich mit dem zuletzt bearbeiteten Bild upgedated, wobei Pixel, die als Bewegung erkannt werden, schwächer in das Modell integriert werden als Pixel, die als Hintergrund erkannt wurden.

6.3.2 Algorithmus zum Tracken von Objekten

Der verwendete Algorithmus, der für das *Tracking* von Objekten verwendet wird, ist *Mean Shift*. Bei *Mean Shift* handelt es sich um ein nicht parametrisiertes Verfahren zur Bestimmung lokaler Maxima in Verteilungen. Der Algorithmus wurde erstmals von Fukunaga und Hostetler in [FH75] vorgestellt, und in [Che95] wurden von Cheng die Eigenschaften des Verfahrens weiter untersucht. [CM99]. Comanciciu zeigte schließlich in [CRM03], wie mittels *Mean Shift Object Tracking* realisiert werden kann. *Mean Shift* kann für eine Reihe von Bildverarbeitungsaufgaben eingesetzt werden, neben *Tracking* kann auch Filterung und Segmentierung mit Hilfe von *Mean Shift* berechnet werden.
Für das in dieser Masterarbeit entstandene *Tracking* System wurden zwei verschiedene *Tracking* Algorithmen implementiert und getestet. Einerseits wird der *Tracker* von Comanciciu, der auf Farbhistogrammen basiert, mit adaptiver Kernelgröße implementiert, andererseits kommt ein besonders effizienter *Mean Shift Tracker* zum Einsatz (siehe [BFB04]), der nur auf die Helligkeitsverteilung im Differenzbild angewendet wird und die *Mean Shift* Berechnungen effizient über zuvor berechnete Integralbilder in zehn arithmetischen Operationen und zwölf Array-Zugriffen ausführt [BFB05].

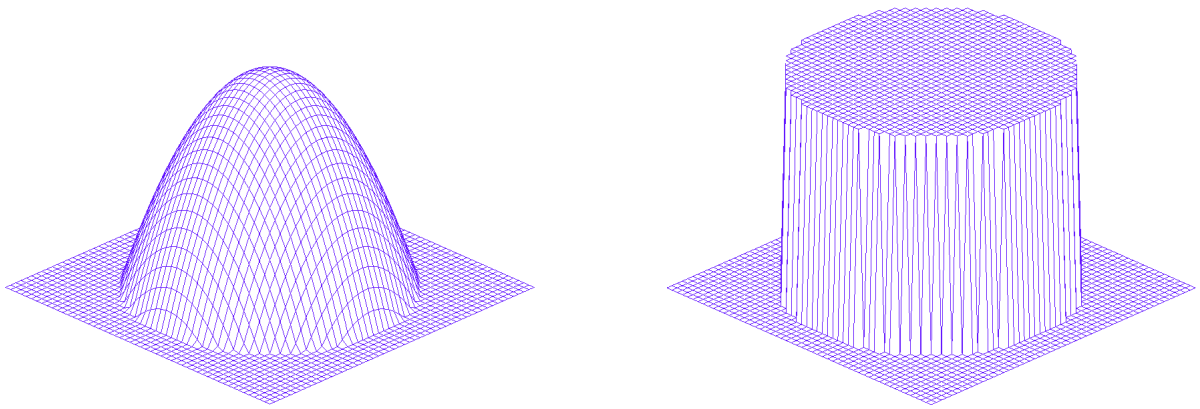
Funktionsweise des *Mean Shift* Algorithmus

Ausgehend von einer Startposition wandert *Mean Shift* entlang des Gradienten bis hin zu einem lokalen Maximum der Verteilung. Die Verteilung ist definiert durch den jeweils gewählten Featureraum und kann im Prinzip anhand beliebiger Features definiert sein.
Das Finden lokaler Maxima wird dadurch erreicht, dass lokal benachbarte Werte in einem Fenster bestimmter Größe auf der aktuellen Position zentriert herangezogen werden und dann das Mittel berechnet wird. Die Position des Mittels definiert zusammen mit der aktuellen Ausgangsposition den sogenannten *Mean Shift*-Vektor.
Lokal benachbarte Sample-Werte werden mit Hilfe eines Kernels verschieden stark gewichtet und erlauben so die Schätzung der Verteilung. So fließen sie durch die Gewichtung in

das sogenannte Zielmodell ein, mit welchem bei jedem Iterationsschritt des *Mean Shift* Algorithmus das Kandidatenmodell verglichen wird, welches durch Kernelanwendung auf der aktuellen Iterationsposition auf die Featurewerte bestimmt wird. Wie viele Werte dabei einfließen hängt von der Größe des Kernels (also von der Fenstergröße) ab. Man wählt einen monoton fallenden Kernel, an dem periphär gelegene Werte weniger stark in das Ergebnis einfließen als zentral gelegene (also nahe dem Fenstermittelpunkt liegende) Werte. In der Praxis hat sich der Epanechnikovkernel (6.13), für dessen Erzeugung man das Epanechnikovprofil[2] verwendet, bewährt, da er die Eigenschaft besitzt, dass seine Ableitung konstant ist und den Einheitskernel (6.14) - auch Flatkernel genannt - definiert. Abbildung 6.3 zeigt Epanechnikovkernel und Einheitskernel.

$$K_E(\mathbf{x}) = \begin{cases} 1 - \|\mathbf{x}\|^2 & if \ \ \|\mathbf{x}\| <= 1 \\ 0 & else \end{cases} \tag{6.13}$$

$$K_F(\mathbf{x}) = \begin{cases} 1 & if \ \ \|\mathbf{x}\| <= 1 \\ 0 & else \end{cases} \tag{6.14}$$



(a) (b)

Abbildung 6.3: (a) Epanechnikovkernel und (b) Einheitskernel

Man spricht in diesem Zusammenhang auch davon, dass der Epanechnikovkernel Shadow-Kernel des Einheitskernels ist [Che95]. Der *Mean Shift*-Vektor geht in dieselbe Richtung

[2]Das Profil eines Kernels ist definiert als Funktion $k : [0, \infty] \rightarrow R$ mit $K(\mathbf{x}) = k(\|\mathbf{x}\|^2)$

wie der Gradient auf der Verteilungsdichte, die man durch Anwendung des Shadowkernels schätzen kann.

Da man zur Berechnung des *Mean Shift*-Vektors den Kernel, der durch das abgeleitete Kernelprofil definiert ist[3], benötigt, vereinfachen sich bei Verwendung des Epanechnikovkernels die Formeln zur Berechnung des *Mean Shift*-Vektors.

Zur Erzeugung des Targetmodells und der Kandidatenmodelle wird also der Epanechnikovkernel verwendet, für die Berechnung des *Mean Shift*-Vektors dessen Ableitung.
Die Verwendung des Epanechnikovkernels zur Erzeugung des Target- bzw. Kandidatenmodells wird auf Seite 77 für den *Mean Shift Tracker*, basierend auf Farbhistogrammen, beschrieben.
Die Berechnung des *Mean Shift*-Vektors erfolgt über Formel (6.15).
Da g im Falle des Epanechnikovkernels konstant ist, reduziert sich die Berechnung auf ein gewichtetes Mittel (6.17).
Zur Berechnung des *Mean Shift*-Vektors verwendet man ein Ähnlichkeitsmaß, gebräuchlich ist etwa der Bhattacharyya Koeffizient (6.18). Dieses Maß definiert die eigentliche Verteilung, das heißt, Ähnlichkeiten bzw. das Modell erklärende Pixelwerte liefern einen hohen Wert, dem Modell widersprechende Pixelwerte liefern einen geringen Wert als Maß der Ähnlichkeit. Das Mittel gibt dementsprechend in Bezug auf die Ausgangsposition die Richtung an, in welche ein Maximum der Verteilung in Form von dominanten Ähnlichkeitswerten vorhanden ist. Zu beachten ist, dass, wenn überhaupt keine Features des Objekts in den Fensterbereich fallen, der Algorithmus freilich das Maximum und damit die Aufenthaltsposition des Objekts nicht findet.

$$m_h(x) = \frac{\sum_{i=1}^{n_h} x_i w_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)}{\sum_{i=1}^{n_h} w_i g\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} - x \tag{6.15}$$

wobei h der Fensterradius sei, n_h die Anzahl der Datenpunkte, die x_i somit die Elemente im Merkmalsraum, w_i die Gewichte definiert durch ein vorher festgelegtes Ähnlichkeitsmaß (6.18) für den Vergleich von Ziel- und Kandidatenmodell, g der Kernel ist, der durch Ableitung aus dem Shadowkernel hervorgeht.

[3] $k(x)$ sei für alle $x \in [0, \infty]$ differenzierbar, dann ist $g(x) = -k'(x)$

Man kann aber auch direkt die neue Position $\hat{x}$ berechnen:

$$\hat{x}\left(x\right) = \frac{\sum_{i=1}^{n_h} x_i w_i g\left(\|\frac{x-x_i}{h}\|^2\right)}{\sum_{i=1}^{n_h} w_i g\left(\|\frac{x-x_i}{h}\|^2\right)} \tag{6.16}$$

Verwendet man den Epanechnikovkernel so ist der abgeleitete Kernel konstant (der Einheitskernel) und man erhält die vereinfachte Gleichung (6.17):

$$\hat{x}(x) = \frac{\sum_{i=1}^{n_h} x_i w_i}{\sum_{i=1}^{n_h} w_i} \tag{6.17}$$

Die diskrete Version des Bhattacharyya-Koeffizienten:

$$\hat{\rho}(x) \equiv \rho[\hat{p}(x), \hat{q}] = \sum_{i=1}^{m} \sqrt{\hat{p}_u(x)\,\hat{q}_u} \tag{6.18}$$

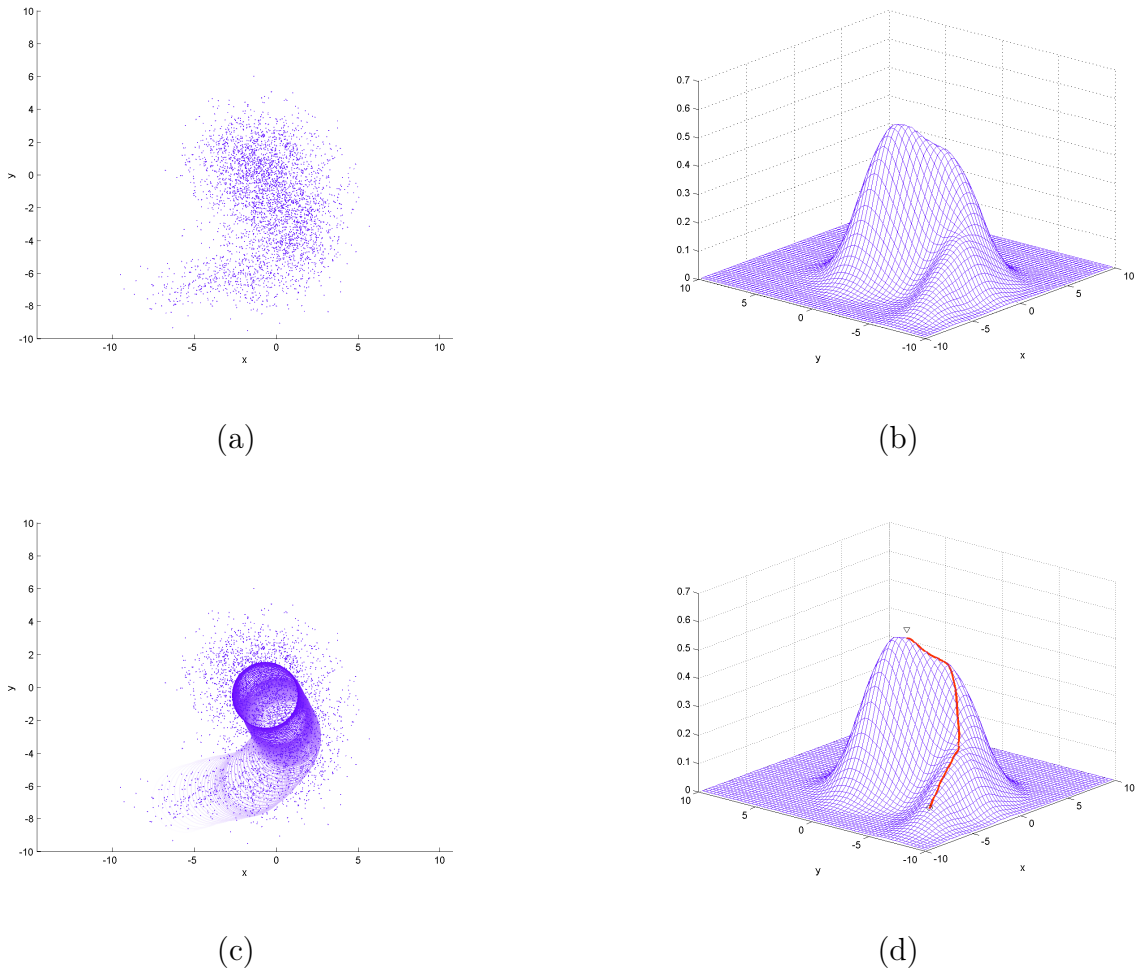
$$d(\hat{p}(x), \hat{q}) = \sqrt{1 - \rho[\hat{p}(x), \hat{q}]} \tag{6.19}$$

$\hat{p}(x)$ sei das aktuell beobachtete Kandidatenmodell, $\hat{q}$ das Zielmodell

Wiederholt man die Berechnung des *Mean Shift*-Vektors iterativ nachdem man die Ausgangsposition zur Berechnung immer anhand des Vektors verschiebt, gelangt man nach einer endlichen Anzahl von Schritten zu einem Maximum der Verteilung. Man bricht die Iteration ab, wenn der Abstand zwischen der ermittelten *Mean Shift* Position und der Ausgangsposition des aktuellen Schrittes kleiner als ein Pixel ist. In der Praxis wird weiters eine Schranke für die Anzahl an Iterationen verwendet, um das Terminieren des Algorithmus sicherzustellen. Das Ähnlichkeitsmaß kann auch dazu verwendet werden, die Fenstergröße nach jedem Iterationsschritt oder nachträglich nach einer Iterationsfolge um einen gewissen Prozentsatz zu variieren (z.B. +/- 10%) und darauf zu testen, ob mit dieser Fenstergröße das Zielmodell besser erklärt wird.

In Abbildung 6.4 sieht man die Funktionsweise des Algorithmus:
Abbildung 6.4 (a) zeigt die Sample-Verteilung, die z.B. als Werte des Ähnlichkeitsmaß durch Gewichtung mit dem Kernel berechnet werden können.
Mittels Parzen Windows [DHS00] kann die durch die Samples definierte Verteilungsdichte der Features (Abbildung 6.4 (b)) unter Verwendung eines Kernels (z.B: Epanechnikovkernel) geschätzt werden. Dieses Verfahren ist parameterfrei und kann somit beliebige Verteilungen approximieren.
Lokale Maxima der Verteilung können mit Hilfe des *Mean Shift* Algorithmus durch iterative Berechnung des *Mean Shift*-Vektors und Verschiebung des Kernels um den *Mean Shift*-Vektor gesucht werden, indem die Samples für die *Mean Shift* Berechnung herangezogen werden (Abbildung 6.4 (c)).
Abbildung 6.4 (d) zeigt den Iterationspfad definiert durch die Iterationsschritte des *Mean*

Shift Algorithmus auf der geschätzten Dichtefunktion. Der schwarze Kreis markiert die Anfangsposition des Algorithmus, das schwarze Dreieck das gefundene Maximum. Zu beachten ist, dass beim *Mean Shift*-Algorithmus nicht die gesamte Verteilung geschätzt werden muss. Es genügt, die Verteilung lokal an der aktuellen Position des Kernels mit Hilfe des Kernels zu schätzen und den *Mean Shift*-Vektor zu berechnen.



(a) (b)

(c) (d)

Abbildung 6.4: Kernel Dichte-Schätzung und *Mean Shift*-Algorithmus

(a) Sample-Verteilung (definiert durch Ähnlichkeitsmaß, z.B: Bhattacharyya-Koeffizient)
(b) Dichteschätzung mittels Epanechnikovkernel
(c) *Mean Shift*-Algorithmus
(d) *Mean Shift*-Iterationspfad auf der geschätzten Dichtefunktion

Mean Shift-Tracking mittels Farbhistogrammen

Der in [CRM03] von Comanciciu et. al. vorgestellte *Tracking* Algorithmus verwendet als Feature-raum Farbhistogramme. Dabei wird für jedes zu trackende Objekt ein Zielmodell definiert, mit welchem dann Kandidatenmodelle verglichen werden. Kandidatenmodelle werden mittels Gleichung (6.20) für alle u Merkmale des Merkmalsraumes (also die Histogrammbins) ermittelt. Dabei wird jedem Pixel im Kernelfenster ein Histogrammbin zugeordnet und unter Zuhilfenahme der Kernelgewichte im Histogramm eingetragen. Das Zielmodell wird zuvor analog bestimmt. Ziel- und Kandidatenmodell werden unter Verwendung des Bhattacharyya Koeffizienten in spezieller Form von Gleichung (6.21) verglichen und fließen als ein Teil der *Mean Shift*-Vektor Berechnung in Form von Gewichten ein.

$$\hat{p}_u(x) = \frac{1}{\sum_{i=1}^{n_h} k\left(\left\|\frac{x-x_i}{h}\right\|^2\right)} \sum_{i=1}^{n_h} k\left(\left\|\frac{x-x_i}{h}\right\|^2\right) \delta\left[b(x_i) - u\right] \tag{6.20}$$

wobei k der Kernel, δ das Kronecker-Delta sei (das lediglich die Funktion hat, sicherzustellen, dass nur Gewichte zu dem entsprechend richtigen Bin einen Beitrag leisten) und durch $\delta(k_d) = \begin{cases} 1 & , \quad k_d = 0 \\ 0 & , \quad sonst \end{cases}$ definiert ist,
und $b : R^2 \rightarrow \{1 \ldots m\}$ sei eine Funktion, die einem Pixel x_i gemäß seinem Farbwert einen entsprechenden Historgrammbin zuordnet.

$$w_i = \sum_{u=1}^{m} \sqrt{\frac{\hat{q}_u}{\hat{p}_u(x)}} \delta\left[b(x_i) - u\right] \tag{6.21}$$

Die Berechnungen erfolgen pixelweise, mit wachsender Fenstergröße steigt der Aufwand quadratisch an. Lediglich durch eine geringere Wahl der Binauflösung lässt sich etwas Performance gewinnen, bei wirklich großen Fenstergrößen (z.B: 80*80 Pixeln) bricht die Performance aber dennoch ein. Bereits bei einem Dutzend Objekten ist die Echtzeitfähigkeit des Algorithmus nicht mehr gegeben.

Performantes *Mean Shift Tracking* mit Hilfe von Integralbildern

Um dem Performance-Problem der großen Fenstergrößen Herr zu werden, wird der in [BFB04] verwendete *Mean Shift Tracker* verwendet, der unter Verwendung von Integralbildern (auch als *Summed Area Table* bekannt) des Differenzbildes der Bewegungserkennungskomponente anhand der bestimmten Helligkeitsbereiche im Bild Objekte zu tracken

vermag.

Als Integralbild sei das Bild definiert, das an Position (x,y) die kumulative Summe aller links oder oberhalb befindlichen Pixel enthält, mathematisch lässt sich das Integralbild beschreiben als

$$I_{int}(x,y) = \sum_{x'<=x,\ y'<=y} I(x',y') \tag{6.22}$$

Berechnen kann man das Integralbild I_{int} eines Bildes I iterativ mit folgender Berechnungsvorschrift (entnommen aus [AYJC05] - auch die Berechnung der Momentbilder (Formel 6.26) kann im gleichen Berechnungsdurchlauf ermittelt werden):

$$r(x,y) = r(x,y-1) + I(x,y) \tag{6.23}$$

$$I_{int}(x,y) = I_{int}(x-1,y) + r(x,y) \tag{6.24}$$

Der Vorteil von Integralbildern besteht darin, dass die Summe der Helligkeitswerte jedes beliebig großen rechteckigen Bereiches des Bildes mit einigen wenigen Berechnungen durchgeführt werden kann:

$$\begin{aligned} S_{area} = {} & I_{int}(x-1,y-1) + I_{int}(x+W-1,y+H-1) \\ & -I_{int}(x-1,y+H-1) - I_{int}(x+W-1,y-1) \end{aligned} \tag{6.25}$$

wobei W die Breite und H die Höhe des rechteckigen Bereiches ist.

Zur Berechnung des *Mean Shift*-Vektors, basierend auf Integralbildern des Differenzbildes, benötigt man auch noch die Integralbilder der ersten Momente des Bildes. Diese berechnen sich wie folgt:

$$I^x_{int}(x,y) = \sum_{x'<=x,\ y'<=y} x' I(x',y') \tag{6.26}$$

$$I^y_{int}(x,y) = \sum_{x'<=x,\ y'<=y} y' I(x',y') \tag{6.27}$$

Ausgehend von der Formel zur Berechnung des *Mean Shift*-Vektors für das Differenz-

bild:

$$m_x(x,y) = \frac{\sum_{i=1}^{n_h} x_i I(x_i, y_i)}{\sum_{i=1}^{n_h} I(x_i, y_i)} - x \tag{6.28}$$

$$m_y(x,y) = \frac{\sum_{i=1}^{n_h} y_i I(x_i, y_i)}{\sum_{i=1}^{n_h} I(x_i, y_i)} - y \tag{6.29}$$

wobei $I(x,y)$ das Differenzbild sei, ergeben sich für Integralbilder folgende Berechnungsregeln:

$$m_x(x,y) = \frac{S_{area}(I^x_{int})}{S_{area}(I_{int})} - x \tag{6.30}$$

$$m_y(x,y) = \frac{S_{area}(I^y_{int})}{S_{area}(I_{int})} - y \tag{6.31}$$

6.3.3 Algorithmus zur Bestimmung zusammengehöriger Pixelregionen

Zur Bestimmung der zusammengehörigen Pixelregionen des Ergebnisbildes der Bewegungserkennung kommt ein Component-Labeling Algorithmus aus [CCL04] zur Anwendung, der in linearer Zeit *Connected Components Analysis* für das gesamte Bild durchführt. Der Algorithmus wurde für Binärbilder vorgestellt, kann aber auch für Grauwertbilder oder Farbbilder verwendet werden, solange man eine Regel definiert, die eindeutig Vordergrund und Hintergrund klassifiziert und differenziert. Der Algorithmus arbeitet das Bild vom Pixel in der linken oberen Ecke bis hin zum letzten Pixel in der rechten unteren Ecke ab. Dabei werden vier wesentliche Fälle unterschieden (siehe auch Abbildung 6.5):

1. Wird ein äußerer Konturpunkt (Fall (a) in Abbildung 6.5) zum ersten Mal erreicht (wurde er noch nicht markiert - weshalb und wie die Markierung durchgeführt wird, wird im nächsten Absatz beschrieben), so wird die Kontur verfolgt (siehe Konturverfolgung auf Seite 81), bis wieder der Konturpunkt erreicht ist; allen Punkten der Kontur wird ein Label zugewiesen.

2. Gelangt man zu einem bereits markierten äußeren Konturpunkt, verfolgt man die aktuelle Scanline, solange man immer auf der Position eines zur Region gehörenden Pixels ist (Fall (b) in Abbildung 6.5). All diese Pixel werden ebenfalls mit dem Label des Konturpunktes versehen.

3. Kommt man zu einem internen Konturpunkt, so erhält dieser auch das Label des äußeren Konturpunktes und man startet eine Konturverfolgung für diese innere Kontur, alle Punkte der inneren Kontur erhalten ebenfalls das Label derselben Komponente (Fall (c) in Abbildung 6.5).

4. Wird ein bereits markierter innerer Konturpunkt erreicht, verfolgt man wieder die Scanline, solange man Punkte der Komponente vorfindet und markiert sie mit dem Label dieses Konturpunktes (Fall (d) in Abbildung 6.5).

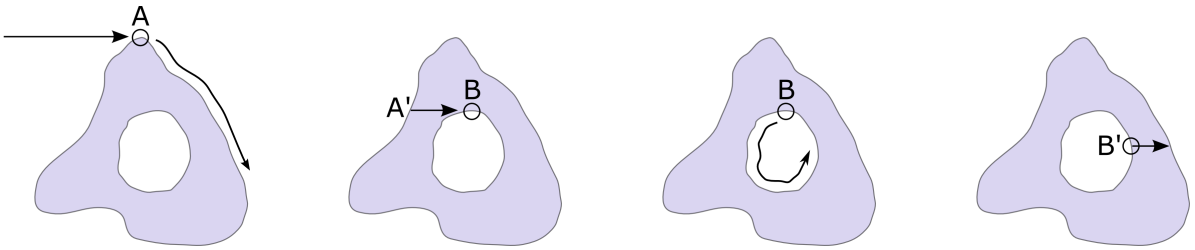


Abbildung 6.5: Die Fälle des Konturverfolgungs und -labeling Algorithmus aus [CCL04])

Um die Korrektheit des Algorithmus sicherzustellen, muss die erste Zeile des Bildes leer von Regionen sein, dies erreicht man entweder durch Löschen der gesetzten Werte in dieser Zeile oder durch Vergrößern des Bildes. Für das Markieren der Pixel benötigt man neben dem tatsächlichen Bild ein weiteres Array, das dieselben Dimensionen wie das Bildes hat. Ein Index wird erstellt, um die aktuelle Labelnummerierung konsistent zu halten.
Die vier oben genannten Fälle können zu drei Schritten zusammengefasst werden. Der erste Schritt behandelt neue äußere Konturpunkte, Schritt zwei neue innere Konturpunkte und der dritte Schritt alle Pixel der Komponente, die nicht von den ersten beiden Schritten bearbeitet wurden.

1. Wenn der Punkt unmarkiert ist und sich über ihm kein gesetzter Pixel befindet, so muss der aktuelle Punkt Teil einer noch nicht behandelten externen Kontur sein. Der mitgeführte Index wird dem Pixel zugewiesen (also im Markierungsarray gespeichert), dann wird die Kontur verfolgt (siehe unten) und allen Pixeln der Kontur wird auch der aktuelle Index zugewiesen. Schließlich wird der Index inkrementiert.

2. Ist unter dem aktuell bearbeiteten (und daher markierten) Pixel kein gesetzter Pixel und ist dieser unmarkiert (wann und warum auch nicht gesetzte Pixel markiert werden, wird in Kürze behandelt), folgt daraus, dass der Pixel ein innerer Konturpunkt

sein muss. Ist der Pixel unter dem aktuellen Punkt markiert, so handelt es sich um einen externen Konturpunkt. Ist er nicht markiert, handelt es sich, wie gesagt, um einen inneren Konturpunkt und der vorherige Pixel auf der aktuellen Scanline muss markiert werden. Weiters wird auch der aktuelle Punkt mit derselben Markierung gekennzeichnet. Schließlich wird wieder die (innere) Kontur verfolgt und alle Pixel auf dieser erhalten ebenfalls dieselbe Markierung.

3. Ist der aktuelle Punkt kein Konturpunkt (wenn er durch die beiden vorhergehenden Schritte nicht behandelt wurde), so muss der linke Nachbar bereits markiert sein und es wird diese Markierung zum Kennzeichnen des aktuellen Punktes verwendet.

Um die Konturverfolgung bei äußeren Konturpixeln zu verhindern, werden die Kontur umgebende Pixel mit negativen Werten markiert. Somit sind diese nicht gesetzten, die Kontur umgebenden Pixel nicht mehr unmarkiert und werden somit im zweiten Schritt richtig erkannt und die Konturverfolgung wird unterdrückt. Außerdem werden so innere Konturen nur einmal verfolgt.

Die Konturverfolgung arbeitet mit Hilfe einer *Tracer*-Funktion (siehe nächster Absatz). Wenn die *Tracer*-Funktion den Startpunkt zur Konturverfolgung als isolierten Pixel identifiziert, endet die Konturverfolgung bereits gleich zu Beginn, sonst liefert die *Tracer*-Funktion den Konturpunkt zurück, der auf den aktuellen Punkt folgt. Die Konturverfolgung terminiert, wenn der von der *Tracer*-Funktion zurückgelieferte Punkt der ursprüngliche Anfangspunkt der Konturverfolgung ist und der darauf folgende zurückgelieferte Konturpunkt wieder der Punkt ist, der beim ersten Aufruf der *Tracer*-Funktion ermittelt wurde.

Die *Tracer*-Funktion liefert ausgehend von einem Pixel den im Uhrzeigersinn nächsten Konturpunkt von den acht Pixeln, die den aktuellen Pixel umgeben. Findet sich kein Punkt, handelt es sich um einen isolierten Pixel. Wird eine Verfolgung einer äußeren Kontur gestartet, so beginnt die *Tracer*-Funktion idealerweise mit dem rechten oberen Nachbarpixel, da ja bereits der obere Pixel als nicht gesetzt identifiziert wurde und damit die Konturverfolgung gestartet wurde. Analog beginnt man bei inneren Konturen mit dem linken unteren Nachbarpixel. Während der Konturverfolgung lässt man die *Tracer* Funktion immer dort beginnen, wo unter Berücksichtigung des letzten Konturpunktes der darauffolgende Konturpunkt als Nächstes liegen könnte. War der letzte Punkt zum Beispiel links unterhalb des aktuellen Punktes, so kann der nächste Punkt nicht links, sondern nur links oberhalb des aktuellen Punktes sein, und somit startet man den aktuellen *Tracer* genau dort. Man nummeriert die acht Nachbarpixel aufsteigend, und so kann für jeden Schritt die initiale Suchposition der Konturverfolgung (ausgenommen natürlich die Startposition) mittels $d + 2 \pmod 8$ einfach berechnet werden. Während man

in der *Tracer*-Funktion den nächsten Konturpunkt sucht, kann man auch gleich umgebende nicht gesetzte Pixel mit negativen Werten versehen. So kann man diese Aufgabe elegant nebenbei in der *Tracer*-Funktion abarbeiten.

Kapitel 7

Implementierung des *Object Tracking* Systems

In diesem Abschnitt wird das entstandene *Object Tracking* System beschrieben, wie es konzipiert ist und wie es implementiert wurde. Zuerst werden die verwendeten Programmierwerkzeuge, die zur Entwicklung des *Tracking* Systems eingesetzt wurden, vorgestellt und die Hardware beschrieben, die das Zielsystem darstellt, danach wird der für das System verwendete *Multi-Threading*-Ansatz erläutert, schließlich folgen Implementierungsdetails zur Bewegungserkennung auf der GPU sowie zum *Object Tracking* Algorithmus.

7.1 Verwendete Programmierschnittstellen und -bibliotheken

Die zur Entwicklung des *Tracking* Systems eingesetzten Werkzeuge sind die verwendeten Programmiersprachen, die Programmierschnittstellen und Bibliotheken.
Als Programmiersprachen zur Realisierung des Projektes wird *C* bzw. *C++* verwendet. Die Grafikkartenprogrammierung erfolgte mittels *OpenGL* als Grafik-API (siehe Abschnitt 3.2.1), *Cg* als Shadersprache (siehe Abschnitt 3.2.2), sowie der Programmierbibliotheken *Glut* [3] zum Erstellen des OpenGL-Kontextes und *Glew* (*The OpenGL Extension Wrangler Library* [2]) zum Initialisieren von OpenGL-Erweiterungen. Als Fragmentprofil kommt „fp40“ zum Einsatz.
Die *Computer Vision*-Library *OpenCV* [9] wird für grundlegende Bildverarbeitungsfunktionalitäten eingesetzt. Diese stellt sowohl performante SSE-Implementierungen (*Streaming SIMD Extensions*) für grundlegende Bildverarbeitungsfunktionalitäten wie das Erstellen und Kopieren von Bildern oder das Laden von Videos zur Verfügung als auch verschiedenste Algorithmen aus der *Computer Vision*. Beispielsweise wird für die Berechnung der Integralbilder, auf denen eine der beiden implementierten Versionen des *Mean*

Shift Tracker basiert, die entsprechende *OpenCV* Funktion verwendet, da diese unter Ausnutzung der speziellen SSE-Funktionalitäten von Prozessoren weniger Zeit brauchen als eine Standard-Implementation des Algorithmus. Für das Projekt wurden aber größtenteils nur die grundlegenden *OpenCV* Funktionen benötigt.
Für die *Multi-Threaded*-Programmierung wird die *Boost* library [1] und als Alternative die *pthreads* library [6] eingesetzt. Das Visualisierung der Ergebnisse und das Steuern der Applikation wird über ein GUI bewerkstelligt, welches mit *Gtk+* [5] implementiert wurde. Zwar stünde unter *OpenCV* ebenfalls ein simples GUI zur Verfügung, doch ist dieses erstens nicht besonders leistungsfähig und zweitens vor allem nicht *thread-safe*, was den Einsatz in einer *Multi-Threaded* Applikation ausschließt.
Alle verwendeten Bibliotheken sind für verschiedenste Betriebssysteme verfügbar. Somit ließe sich das unter Windows entwickelte System auch leicht auf Linux portieren.

7.2 Verwendete Hardware

Das Test- bzw. Zielsystem für das *Object Tracking* System ist ein Rechner mit *Intel Core2 Duo 6300* CPU (2*1.86 Ghz), 2 GB RAM, *nVidia GeForce 7950 GT* GPU (PCI Express) mit Windows XP, das Zweitsystem - um den Unterschied zu einer Single-Core CPU sehen zu können - ist ein *AMD Athlon64 3700+* (2.38 Ghz), 3 GB RAM, *nVidia GeForce 6800 GS* (PCI Express) mit Windows XP. Die verwendete Grafikkarten-Treiberversion war *nVidia ForceWare* 91.47.
Der benötigte Speicher hängt von der Größe des zu verarbeitenden Videodatenstroms ab, bei einem PAL Video beläuft sich der gesamte benötigte RAM-Speicher der Applikation auf weniger als 200 MB. In diesem Fall wird auch eine Grafikkarte mit 128 MB benötigt, da ein Bild etwas mehr als 7 MB Grafikkartenspeicher belegt - $768 \times 576 \times 4 \times 4$ Byte - Breite mal Höhe mal Anzahl der Farbkanäle (inklusive Alpha-Kanal, der für einige Berechnungen zweckentfremdet wird) mal Anzahl der benötigten Bytes für die Rechengenauigkeit - und für die Berechnungen der Zwischenbilder, Bilder des Hintergrundmodelles (Durchschnittsbild, Standardabweichungsbild, etc.) jeweils zwei Texturen für das Texture Ping Ponging und so insgesamt schließlich mehr als ein Dutzend Texturen benötigt werden. Bei 128 MB lassen sich bei einer Bildgröße, wie sie durch ein PAL Video definiert ist, 18 Texturen der benötigten Größe hinterlegen. Da aber ohnehin eine schnelle Grafikkarte benötigt wird (*nVidia GeForce 6800 GS* oder besser), steht somit diese Grafikkartenspeichergröße zur Verfügung.

7.3 Die *Multi-Threaded* Implementierung des *Tracking* Systems

Um die Leistungsfähigkeiten von Mehrkern-Rechner-Architekturen ausschöpfen zu können, muss man *Multi-Threaded* programmieren, wie dies bereits in Abschnitt 5 erläutert wurde. Bei dem in dieser Masterarbeit entstandenen *Object Tracking* System wird deshalb die Arbeit in mehrere Threads aufgeteilt. Die Aufteilung kann aber nicht beliebig erfolgen. Da der gesamte Verarbeitungsprozess vom Eingangsbild ausgehend bis hin zu den letztendlichen *Tracking*-Ergebnissen eine aufeinander aufbauende Verarbeitungsreihenfolge bedingt, kann man nicht gleichzeitig für dasselbe Bild Bewegungserkennung und *Object Tracking* durchführen. Möglich ist jedoch, wenn man die Bewegungserkennung für das aktuelle Bild durchführt, das *Object Tracking* auf bereits vorangegangenen Bildern, für die bereits die Bewegungserkennung berechnet wurde, durchzuführen. Man arbeitet also in einer ineinander verzahnten Reihenfolge die Bilder ab. Während der Bewegungserkennungs-Thread schon die Bewegung im neuesten Bild ermittelt, werden gleichzeitig in einem weiteren Thread die Objekte im vorangegangenen Bild getrackt. Einzelne Bilder werden so in einer festen Reihenfolge von den einzelnen Threads bearbeitet, die Threads selber führen parallel auf mehreren Bildern Berechnungen durch. Da diese feste Verarbeitungsreihenfolge für die Bilder gilt und die Threads, einer nach dem anderen, auf einem Bild arbeiten, bietet sich der Begriff der Fließbandarbeit (auch *pipeline* genannt) an.
In Abbildung 7.2 sieht man die schematische Darstellung des *Multi-Threading*-Ansatzes innerhalb des *Object Tracking* Systems. Zu beachten ist, dass sich bei diesem Ansatz die Verarbeitungszeit eines einzelnen Bildes nicht ändert (sie steigt sogar etwas aufgrund des Verwaltungsaufwandes für *Multi-Threading*), aber durch die gleichzeitige Bearbeitung von verschiedenen Einzelbildern steigt der Durchsatz, wie ein Vergleich der gewöhnlichen, seriellen Abarbeitung (siehe Abbildung 7.1) mit der parallelen Abarbeitung (siehe Abbildung 7.2) zeigt.

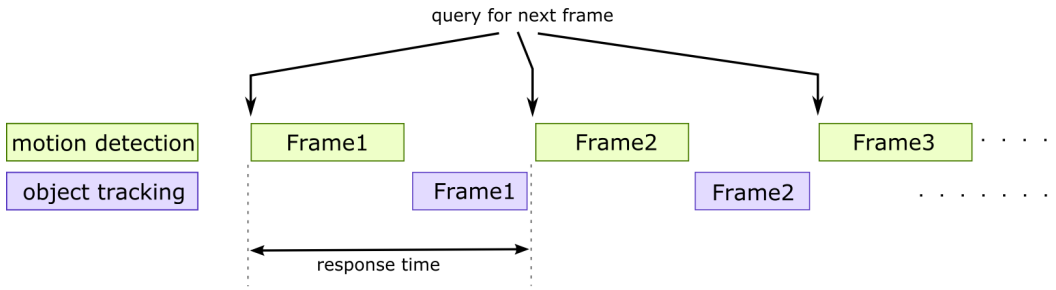


Abbildung 7.1: Herkömmliche, serielle Abarbeitungsreihenfolge von Verarbeitungsschritten innerhalb des seriellen *Tracking* Systems

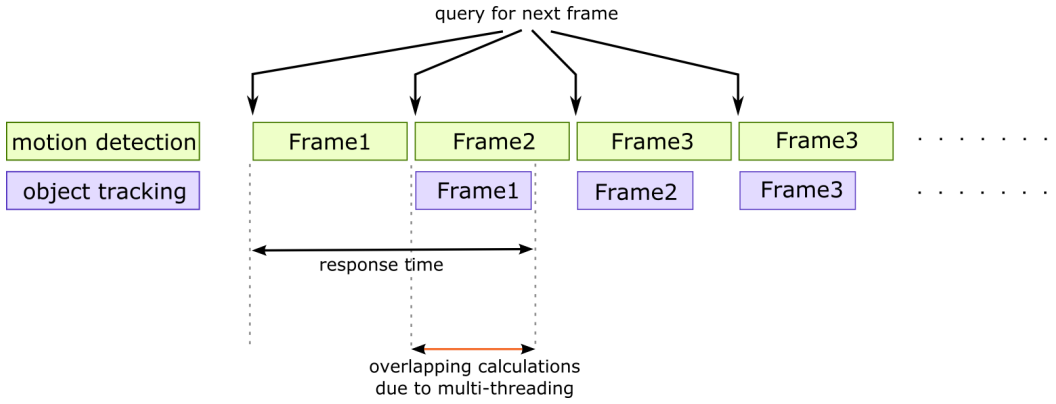


Abbildung 7.2: Parallele Abarbeitungsreihenfolge von Verarbeitungsschritten innerhalb des *Multi-Threaded Tracking* Systems

In den beiden Abbildungen wurde die Berechnungszeit der beiden Verarbeitungschritte zur Bewegungserkennung und zum *Object Tracking* als unterschiedlich lang dargestellt, wie es auch im implementierten System der Fall ist. Zu Abbildung 7.2 sei gesagt, dass die rot eingezeichnete Zeitspanne die ist, die man durch das *Multi-Threading* Konzept einsparen kann. Es ist gut zu erkennen, dass die Reaktionszeit, also die Zeit, die das System benötigt, um ein neues Bild komplett abzuarbeiten, durch Verwendung von *Multi-Threading* nicht geringer wird.
Die beiden Abbildungen vereinfachen auch, da im System noch zusätzliche Threads für die Bild-erfassung und für die Ausgabe verwendet werden. Das GUI benötigt einen weiteren Thread. Am grundlegenden Konzept ändert sich aber nichts. Jeder Thread, der die Ergebnisse eines anderen benötigt, reiht sich in die Pipeline ein und übergibt seine Ergebnisse an einen eventuellen Nachfolge-Thread. Lediglich der GUI-Thread ist komplett losgekoppelt und kümmert sich in regelmäßigen Zeitabständen darum, dass bereits vorliegende Ergebnisse angezeigt werden bzw. dass das GUI reaktiv bleibt und die Steuerelemente verwendet werden können. Auch sei erwähnt, dass sich zwar der Thread für die Bewegungserkennung nur um diese Aufgabe kümmert (also die Grafikkarte steuert), der Thread, der das *Object Tracking* durchführt, aber auch gleich das Track-Management und alle weiteren Algorithmusschritte, die auf der CPU zu berechnen sind, übernimmt. Der Input-Thread holt Bilder von der Videoquelle und stellt sie für den Bewegungserkennungs-Thread bereit. Dieser führt die Bewegungserkennung durch und stellt die Ergebnisse dem *Object Tracking* Thread zur Verfügung. Die einzelnen Threads benachrichtigen den Output-Thread, sobald sie Zwischenbilder berechnet haben, die visualisiert werden sollen. Der Output-Thread sammelt diese Informationen und signalisiert dem GUI schließlich, dass neue Ergebnisse vorliegen, welches sich dann um das Visualisieren der Bilder kümmert. In Abbildung 7.3 ist dieses Zusammenspiel der einzelnen Threads visualisiert.

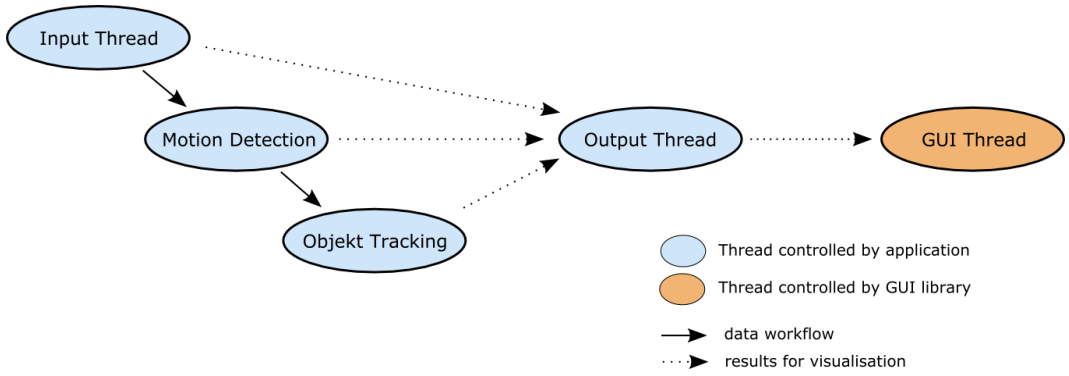


Abbildung 7.3: Das Zusammenspiel der Threads im System

Als Datenstruktur, die als Schnittstelle und Synchronisationsmechanismus zwischen den Threads fungiert, wird eine Queue, die nach dem FIFO-Prinzip (*first in, first out*) arbeitet, verwendet, die allerdings nur eine bestimmte Anzahl an Elementen (also Bildern) fassen kann (auch *Bounded-Buffer* [Sta05]). Versucht ein Thread über den *push*-Befehl eine bereits volle Queue weiter zu füllen, versetzt die implementierte Queue-Klasse den Thread mittels eines *Monitors* effizient in den Wartezustand, bis sie wieder neue Elemente aufnehmen kann (bis ein anderer Thread sich ein Bild aus der Queue geholt und es somit aus der Queue entfernt hat). Umgekehrt verhält sie sich äquivalent, sobald ein Thread mittels des *pop*-Befehls versucht, ein Bild aus der Queue zu lesen, wenn diese noch leer ist. Der Thread wird so lange schlafen gelegt, bis ein anderer Thread die Queue wieder gefüllt hat. In der Praxis reicht bereits eine geringe Anzahl für die Queue-Elemente aus (z.B: 10 Elemente). Es existiert für jeden Zwischenschritt eine eigene Queue, also eine, die die Inputbilder hält, eine, die die Bewegungserkennungs-Ergebnisse speichert, eine für *Tracking*-Ergebnisse und eine mit den zu visualisierenden Bildern. Diese spezielle Art von Queue ist deshalb wichtig, da es sein kann bzw. vorkommt, dass ein Thread deutlich länger braucht als alle anderen Threads und somit - würde man die Daten nicht puffern - Daten verlorengingen bzw. so lange neuer Speicher angefordert werden würde, bis der komplette verfügbare Arbeitsspeicher vollgeschrieben wäre und schließlich kein weiterer freier Speicher mehr zur Verfügung stünde.
Generell warten schnellere Threads auf die langsamen Threads, was aber im konkreten Fall deshalb nicht tragisch ist, da man mittels *Monitors Spinlocks* verhindern kann (*Spinlocks* wurden in Abschnitt 5.1.3 erklärt).
Verwendete OpenGL-Kontexte dürfen nur innerhalb eines Threads verwendet werden, somit wird der OpenGL-Kontext innerhalb des Bewegungserkennungs-Threads erzeugt und vor dem Thread-Ende deinitialisiert. Sämtliche GPU-basierende Funktionen werden innerhalb dieses Threads ausgeführt.

7.4 Implementierungsdetails zur Motion Detection auf der Grafikkarte

Zur Implementierung der Bewegungserkennung auf der Grafikkarte seien hier einige Vorgänge genauer beschrieben. Für die Initialisierung des Hintergrundmodells für den Motion Detection Algorithmus (genaue Beschreibung des Algorithmus siehe Abschnitt 6.3.1) werden zwei Durchläufe benötigt. Im ersten Durchlauf lassen sich das Durchschnittsbild für die Helligkeit und das Standardabweichungsbild für die Helligkeit sowie der durchschnittliche Chrominanzvektor berechnen. Der zweite Durchlauf ist nötig, um die durchschnittliche

euklidische Distanz zwischen dem durchschnittlichen Chrominanzvektor (der also bereits berechnet worden sein muss) und den einzelnen Chrominanzvektoren zu ermitteln. All diese Größen werden für jeden Pixel berechnet und in entsprechenden Texturen auf der GPU resident gehalten.
Da aber der Grafikkartenspeicher begrenzt ist (siehe dazu auch Abschnitt 7.2), können die Durchschnittsbilder nur durch eine Näherung mittels *Running Average*-Methoden berechnet werden, da größere Mengen an Bildern für die Initialisierung des Hintergrundmodells herangezogen werden. Wegen des großen Speicherbedarfes, den bereits ein Bild hat, kann man nicht alle Bilder gleichzeitig in den Speicher der GPU laden, um dann den Durchschnitt aus allen Bildern auf einmal zu berechnen, sondern muss immer mit den einzelnen Bildern nach und nach das Durchschnittsbild nachführen.
Ein weiterer wichtiger Punkt ist die notwendige Rechengenauigkeit für das Hintergrundmodell. Auf der Grafikkarte stehen verschiedene Texturformate zur Verfügung (siehe Abschnitt 3.1.5), die alle spezielle Eigenschaften besitzen und unterschiedliche Genauigkeit liefern. Das gewählte Texturformat hängt auch stark mit der erreichbaren Ausführungsgeschwindigkeit auf der GPU zusammen. Zwar reicht für die meisten Algorithmusschritte der implementierten Bewegungserkennung ein Texturformat mit 8-Bit oder 16-Bit Genauigkeit aus, was auch mit einer rasanten Berechnungsgeschwindigkeit einhergeht (siehe auch Tabelle 8.2), allerdings wird zum sinnvollen Updaten des Hintergrundmodells eine höhere Rechengenauigkeit benötigt.
Das zuletzt verarbeitete Bild wird so in das Hintergrundmodell integriert, dass - den Ergebnissen der Bewegungserkennung Folge leistend - die Pixel, die als Vordergrund detektiert wurden, schwächer integriert werden als entsprechende Hintergrundpixel. Da die Werte, mit denen ein neues Bild einfließt, sehr gering sind (z.B: 0.5×10^{-3}, man also für die binäre Darstellung der Mantisse der berechneten Werte zumindest 10 Bit benötigt) und nicht innerhalb der Genauigkeit liegen, die mit gemappten 8-Bit bzw. 16-Bit Texturen darstellbar ist, benötigt man eine höhere Rechengenauigkeit und ungemappte Werte. Deshalb ist es ratsam, ein 32-Bit Texturformat zu wählen, um eine korrekte Integration neuer Bilder in die Bilder des Hintergrundmodells zu gewährleisten.
Neben der Berechnungsgeschwindigkeit (also der Shader-Ausführungsgeschwindigkeit) wird noch ein wichtigerer Aspekt durch das gewählte Texturformat bedingt. Es hängen nämlich auch die Transferzeiten vom Hauptspeicher in den Grafikkartenspeicher und umgekehrt direkt vom gewählten Texturformat ab, da höhere Genauigkeit auch mehr Speicherbedarf impliziert.
So benötigt der Transfer einer Textur in einem 32-Bit Texturformat bis zu viermal länger als eine Textur mit 8-Bit bzw. 16-Bit Genauigkeit.

Da aber die vierfache Transferzeit die Echtzeitfähigkeit des Systems verhindert, muss ein anderer Weg beschritten werden. Deshalb wird für den Transfer eine Textur in einem 8-Bit Texturformat verwendet und anschließend durch einen zusätzlichen Shaderpass eine Konvertierung in ein 32-Bit Texturformat durchgeführt. Danach werden sämtliche Berechnungen mit 32-Bit Genauigkeit ausgeführt, und vor dem Rücktransfer erfolgt eine erneute Konvertierung, diesmal zurück in eine Textur mit einem 8-Bit bzw. 16-Bit Texturformat. So lassen sich die Transferzeiten in einem vernünftigen Rahmen halten auf Kosten einer zusätzlichen Shaderausführung, die allerdings aufgrund der geringen Ausführungszeit im Millisekundenbereich verschmerzbar ist.
Neue Bilder werden nach eventuellem Postprocessing (z.B: Weichzeichnungsfilter) zuerst in den IHLS-Farbraum konvertiert, indem die verfügbaren vier Farbkanäle der Textur so ausgenutzt werden, dass die Helligkeit im ersten Kanal, die Sättigung im zweiten und die beiden Komponenten des Chrominanzvektors im dritten und vierten Kanal gespeichert werden.
Für das Hintergrundmodell kann dann ein Durchschnittsbild berechnet werden, das den Durchschnitt für die Helligkeit, Sättigung und auch gleich den Durchschnitts-Chrominanz-Vektor beinhaltet.
Die Berechnung des Standardabweichungsbildes und der durchschnittlichen euklidischen Distanz neu eingehender Chrominanz-Vektoren zum Durchschnitts-Chrominanz-Vektor sind weitere Berechnungssschritte.
Nachdem das Hintergrundmodell initialisiert wurde, kann es zur Bestimmung von Bewegungen eingesetzt werden. Wieder werden neue Bilder nach eventuellem Postprocessing zuerst in den IHLS-Farbraum konvertiert, um dann pixelweise mit den Regeln aus Abschnitt 6.3.1 überprüft zu werden.
Danach kann mittels morphologischer Operationen (Dilation, Erosion) Opening und Closing durchgeführt werden, um Pixelrauschen zu eliminieren bzw. Löcher in Objektregionen zu stopfen.
Schließlich werden auch gleich noch die Momentbilder, die für den *Tracking* Algorithmus benötigt werden, ermittelt (sie werden also nicht im selben Schleifendurchlauf wie die Integralbilder berechnet, sondern auf der Grafikkarte, da dies noch schneller ist).
Das Ergebnis wird schließlich wieder in den Hauptspeicher transferiert, und auch die sogenannte *Motion Mask* (das maskierte Bild der Bewegungserkennung, welches die Segmentierung in Vorder- und Hintergrundpixel enthält) kann übertragen werden, um eine aussagekräftigere Visualisierung zu ermöglichen.
Es sei angemerkt, dass die Konvertierung der 8-Bit Textur in die 32-Bit Textur erst bei der IHLS-Konvertierung durchgeführt wird, da erst zu diesem Zeitpunkt die Genauigkeit

benötigt wird. Verzichtet man auf die 32-Bit Genauigkeit, funktioniert das System bis auf die Integration der neuen Bilder in das Hintergrundmodell ebenfalls fehlerfrei, allerdings hat man dann eben aufgrund der fehlenden Rechengenauigkeit das Problem, dass Objekte zu schnell oder zu langsam integriert werden, da man nicht diese Feinabstufung der Werte zur Verfügung hat, die man benötigen würde, um ein korrektes Ergebnis gewährleisten zu können.
Auch bei Verwendung von 32-Bit Genauigkeit muss man sich einiger GPU-bedingter Eigenschaften bewusst sein. In Abschnitt 3.1.4 wird die Thematik der Rechengenauigkeit von Grafikkarten bei *floating-point*-Berechnungen erörtert und es werden Abweichungen vom IEEE 754 Standard erläutert.
Daraus resultiert nämlich, dass beim verwendeten Bewegungserkennungs-Algorithmus die berechneten Ergebnisbilder bei der CPU- bzw. GPU-Variante unterschiedlich ausfallen. Dies äußert sich durch die Notwendigkeit, Thresholdwerte anpassen zu müssen, um möglichst ähnliche Ergebnisse zu erhalten.
Bereits die übertragenen Eingabebilder werden nämlich durch leicht voneinander abweichende Werte auf GPU bzw. CPU dargestellt (die Rohdaten, die als unsigned byte Werte vorliegen, werden aus Performancegründen erst auf der GPU in *floating-point*-Werte umgerechnet).
Diese Abweichung ist zwar erst in der fünften bzw. oft erst in der sechsten Nachkommastelle vorhanden, doch aufgrund von Fehlerfortpflanzung und Einführung neuer Fehler[1] nach jeder arithmetischen Operation durch die dann angewendeten Rundungsregeln, welche von denen des IEEE 754 Standards auf GPUs abweichen, ergeben sich nach und nach deutlichere Abweichungen, verglichen mit der CPU-Variante.
Gerade das Nachführen des Hintergrundmodelles erfordert mehrere arithmetische Operationen und wird für jedes zu verarbeitende Bild durchgeführt, sodass eine Vielzahl an Fehlerquellen zum Ergebnis beiträgt. Bemerkbar macht sich dieses Phänomen durch Detektion von Rauschen als Bewegung, wenn man die Thresholdwerte nicht vergrößert. Passt man die Werte an, erhält man äquivalente Ergebnisse mit der CPU-Variante.
Ein Phänomen, das während dieser Masterarbeit beobachtet und untersucht wurde, ist die Tatsache, dass die GPU den für die API-Befehle zuständigen CPU-Kern immer dann blockiert, wenn Datentransfers durchgeführt werden.
Versuche, dieses Problem in den Griff zu bekommen, unter anderem durch Kontaktaufnahme mit nVidia-Entwicklern im Forum der GPGPU Website [4], führten zu keiner Lösung. Vorgeschlagen wird, DMA-Transfers zu initiieren und den zuständigen Thread nach In-

[1] Der Begriff Fehler definiert hier die Abweichung der berechneten Werte vom IEEE 754 Standard. (Es erfahren auch Werte, die nach IEEE 754 Standard berechnet werden, Rundungsfehler.)

itiierung des Transfers für die erwartete Zeitspanne (die man zuvor durch Messungen ermittelt) schlafen zu legen. DMA-Transfers können mittels sogenannter PBO-Transfers (*Pixel Buffer Objects*) realisiert werden (siehe dazu ebenfalls das GPGPU Forum [4]), allerdings erhöhten sich die Transferzeiten bei Verwendung von PBO-Transfers um mehr als das Doppelte (unter Beachtung der als für PBO-Transfers geeignet bzw. als funktionierend bekannten Texturformate und Kombinationen derselben), was den Vorteil eventuellen asynchronen Verhaltens hinfällig werden ließ.
Für die Ausführung der Shaderprogramme wird empfohlen, diese samt der zu rendernden Geometrie zu laden, danach die Befehle gegebenenfalls zu flushen und den Thread ebenfalls schlafen zu legen, um so anderen Threads die CPU-Zeit zur Verfügung zu stellen.
Tatsächlich konnte so ein zusätzlicher Performancegewinn erzielt werden, allerdings muss die Wartezeit für jedes System individuell angepasst werden.

7.5 Implementierungsdetails zum *Tracking* Algorithmus

Für das Tracken der Objekte werden zwei *Tracking* Algorithmen getestet und verglichen. Als Erstes kommt der Algorithmus aus [CRM03] basierend auf Farbhistogrammen mit adaptiven Kernelgrößen zum Einsatz (siehe Abschnitt 6.3.2).
Der Algorithmus leidet speziell bei großen Videoauflösungen an dem Problem drastischer Performanceeinbrüche bei Objekten, die sich über mehrere hundert Pixel erstrecken. Lässt sich der Algorithmus für wenige Objekte moderater Größe noch brauchbar berechnen, so bricht die Performance bei mehr als einem halben Dutzend Objekten bereits spürbar ein, speziell wenn einige davon große Ausmaße annehmen.
Die ursprüngliche Implementation, die dynamisch Speicher für neue Kernelgrößen (drei passende Kernelgrößen - einen Kernel in Größe des betrachteten Fensters, ein um 10% kleinerer und ein um 10% größerer Kernel) allokierte und freigibt, sobald der Algorithmus die Fenstergröße anpasst, wird geändert. Die Änderung besteht darin, dass sämtliche Fenstergrößen auf eine statische Kernelgröße skaliert und interpoliert werden.
Zusätzlich werden für besonders große Objekte in regelmäßigen Abständen symmetrisch um den Objektmittelpunkt Pixel des Objektes bei der Berechnung ausgelassen, um Rechenzeit zu sparen. Je größer die Objekte sind, desto mehr Pixel werden so vernachlässigt.

Der Ansatz funktioniert, die Rechenzeit ist nicht mehr von der Objektgröße abhängig, aber dennoch lässt sich damit nur eine begrenzte Anzahl von Objekten in Echtzeit tracken (je nach Objektgröße war am Testsystem bei 8-12 Objekten die Echtzeitfähigkeit nicht

mehr gegeben).

Als Alternative wird der Algorithmus aus [BFB04] implementiert (siehe dazu Abschnitt 6.3.2), der *Mean Shift Tracking* basierend auf den Helligkeitswerten des Differenzbildes der Bewegungserkennung durchführt. Die benötigten Momentbilder werden bereits auf der Grafikkarte auf den Ergebnisbildern der Bewegungserkennung berechnet und anschließend als Ergebnis in den Hauptspeicher zurückgelesen. Der CPU Thread kümmert sich schließlich um die Berechnung der eigentlichen Integralbilder und verwendet dazu die optimierten OpenCV-Funktionen. Danach wird mit Hilfe dieser Integralbilder der *Mean Shift*-Vektor iterativ berechnet und die aktuelle Position verschoben, bis der Algorithmus konvertiert.
Der Track-Manager-Teil der Applikation arbeitet für beide *Tracking* Algorithmen ähnlich. Nach Ermittlung der zusammenhängenden Pixelregionen wird die euklidische Distanz vom Regionsmittelpunkt zum Objektmittelpunkt aller bisher in der Liste der bereits getrackten Objekte befindlichen Objekte ermittelt. Für das örtlich nächste Objekt wird jeweils die Pixelausdehnung[2] der entsprechenden Objekte in x- und y-Richtung untersucht. Übersteigt die jeweilige Pixelausdehnung des Objekts addiert zur entsprechenden Pixelausdehnung der Region (plus ein kleines Korrekturintervall) den Abstand von Regions- zu Objektmittelpunkt in x- bzw. y-Richtung, wird diese Region als bereits getrackt eingestuft, ansonsten wird ein neues zu trackendes Objekt erzeugt und in die Liste aufgenommen.
Beide *Mean Shift*-Algorithmen arbeiten mit adaptiver Kernelgröße, die aufgrund eines Ähnlichkeitsmaßes mit etwas größeren bzw. kleineren Kernelgrößen verglichen und durch diese ersetzt wird, falls eine bessere Übereinstimmung, also ein höherer Ähnlichkeitswert, bei den anderen Kernelgrößen vorliegt. Das Ähnlichkeitsmaß beim Farbhistogramm-*Tracker* definiert sich aus der prozentuellen Übereinstimmung von Belegungen der Histogrammbins, das Ähnlichkeitsmaß beim *Mean Shift Tracker* basierend auf Integralbildern wird durch Helligkeitsanteile pro Pixel aller Pixel des betrachteten Kernelfensters berechnet. Bei beiden Algorithmen tendieren die Kernelgrößen mit fortlaufender Zeit dazu, kleiner zu werden als die tatsächliche Größe der zu trackenden Pixelregionen. Diesem Phänomen wird entgegengewirkt, indem beim Test, der feststellt, ob die in der *Connected Components Analysis* ermittelten Regionen bereits als getrackte Objekte behandelt werden, bei einem positiven Testergebnis (also wenn die Region als bereits getrackt erkannt wird) gleichzeitig die Größe des getrackten Objektes vergrößert wird, falls die Region größer ist als die durch die Kernelgröße spezifizierte Objektgröße.

[2]Die Pixelausdehnung eines Objekts wird als die halbe Objektgröße in Pixeln, gemessen vom Mittelpunkt, definiert, je nachdem, ob die x- oder y-Ausdehnung betrachtet wird und enspricht der Ausdehnung der *Bounding Box*.

Eine weitere Regel verhindert, dass, wenn eine Region mehreren bereits getrackten, örtlich nahegelegenen Objekten zugeordnet werden könnte, kein Objekt eine Größenanpassung erfährt. So kann in einigen Verdeckungsfällen ein korrektes Ergebnis erreicht werden, in den meisten Fällen führen aber Verdeckungen dennoch zu Fehldetektionen.

Genauigkeit des *Trackers*

Bis auf Verdeckungen und zu einer zusammenhängenden Pixelregion verschmelzende Objekte trackt der implementierte *Tracking* Algorithmus Objekte fehlerfrei, solange die Schrittweite der Bewegungen nicht die Kernelgröße des *Mean Shift Trackers* übersteigt und die Ergebnisse der Bewegungserkennung korrekt sind. Als korrektes Ergebnis der Bewegungserkennung wird die sogenannte *ground truth* verstanden, die die Menge an Pixeln angibt, die tatsächlich durch bewegliche Objekte und nur durch solche, definiert ist. Inkorrekte Ergebnisse des Bewegungserkennungs-Algorithmus können folglich auch zu Fehldetektionen des *Trackers* führen.
Da der Schwerpunkt der Masterarbeit aber auf der performanten Umsetzung der Algorithmen liegt, wird das Problem der Verdeckungen nicht weiter studiert und könnte in etwaigen zukünftigen Weiterentwicklungen des Systems etwa durch Implementierung von Prädikationsfiltern behandelt werden.
Speziell kann jeder *Tracking* Algorithmus, der auf der Vordergrund-/Hintergrundsegmentierung, ermittelt durch einen Bewegungserkennungs-Algorithmus, basiert, den in dieser Arbeit verwendeten Algorithmus ersetzen und so gegebenenfalls bessere *Tracking*-Ergebnisse ermöglichen. Die Echtzeitfähigkeit ist so lange gegeben, als der neue *Tracking* Algorithmus nicht mehr Zeit benötigt als der in dieser Arbeit verwendete Algorithmus, was etwa 15 Millisekunden entspricht, da die restliche verfügbare Rechenzeit der Prozessorkerne für das GUI, das Dekodieren des Eingabevideos, die Bewegungserkennung sowie die Kontext-Switches beim *Multi-Threading* benötigt wird.
Festzuhalten ist somit, dass der in dieser Arbeit verwendete *Tracking* Algorithmus austauschbar ist und lediglich stellvertretend für eine Vielzahl von Algorithmen gesehen werden kann.

Kapitel 8

Ergebnisse der Arbeit

In diesem Kapitel werden die Ergebnisse der Masterarbeit zusammengefasst und präsentiert. Zuerst wird das enstandene Programm vorgestellt, danach werden Ergebnisse von Performance- und Zeitmessungen der einzelnen Verarbeitungsschritte sowie des Gesamtsystems dargelegt.
Sämtliche Zeitmessungen erfolgten über den Performance Counter, wobei die Zeitmessung von Verarbeitungsschritten auf der GPU so gemessen wurde, dass bei der ersten Zeitnehmung zuerst alle Befehle geflusht wurden und vor der zweiten Zeitnehmung ein *glFinish()* zum Einsatz kam. Dies ist deshalb notwendig, da die Grafikkarte Befehle buffert und erst dann abarbeitet, wenn sie es durch ihre Scheduling-policy für richtig befindet. Will man Ausführungszeiten messen, ist dieses Verhalten der Grafikkarte unerwünscht und kann durch entsprechende Flush- und Finish-Befehle verhindert werden. Dieser Mechanismus sollte nur zur Zeitmessung benutzt werden, da ansonsten die Ausführungseffizienz von Verarbeitungsschritten auf der GPU beeinträchtigt wird. Bei jedem Datentransfer zu und vom GPU-Speicher erfolgt ebenfalls implizit ein *glFinish()*, sodass speziell GPGPU-Programme bei sämtlichen Transfers blockieren (siehe nächster Absatz). Um eine robustere Messung zu erhalten, wird für 1000 Schleifendurchläufe (Bildverarbeitungen) die Zeit gemessen und danach gemittelt und dieser Wert als Messergebnis betrachtet.

8.1 Das entwickelte Programm

Es folgen Screenshots der in dieser Masterarbeit entstandenen Applikation. Zu sehen sind das GUI sowie die Visualisierung des Eingabe-Videos (Abbildung 8.1) bzw. die Ergebnisse der Bewegungserkennung (Abbildung 8.2) und des *Object Tracking* (Abbildung 8.3), welche auf das Video angewendet wurden. Bei der Bewegungserkennung sind Pixel, die als zu einem bewegten Objekt gehörig klassifiziert werden, in rot visualisiert. Pixel, bei denen

Schattenwurf detektiert wurde, werden in blau visualisiert, und bei grün gekennzeichneten Pixeln handelt es sich um Reflexionen, welche detektiert wurden. Die Ergebnisse des *Object Tracking* Algorithmus - also die getrackten Bildregionen - werden durch Überlagerung verschiedenfarbiger Rechtecke visualisiert, und die Trajektorien, die den zurückgelegten Weg, den ein Objekt genommen hat, repräsentieren, sind als entsprechend farbige Schlangenlinie realisiert. Das Video, welches auf den Abbildungen zu sehen ist, stammt vom PETS (*Performance Evaluation of Tracking and Surveillance*) Projekt [11] aus dem Jahr 2003. Für sämtliche Berechnungen und Zeitmessungen wird für diese Arbeit auf PETS-Videos zurückgegriffen.

8.2 Zeitmessung der GPU-Berechnungen verglichen mit der CPU-Varianten

Die Ausführungsgeschwindigkeiten der auf der GPU implementierten Arbeitsschritte für die beiden Testsysteme finden sich in Tabelle 8.1. Die Transferzeiten und die Berechnungszeiten werden separat angeführt. In dieser Tabelle sind Transferzeiten und Berechnungszeiten der auf der GPU durchgeführten Arbeitsschritte für die beiden Grafikkarten der beiden Testsysteme aufgeführt.

Es wurden jeweils die Zeiten für die *Single-Threaded* bzw. *Multi-Threaded* Applikation gemessen. Die höheren Zeiten in der *Multi-Threaded* Applikation erklären sich durch die Kontext-Switches beim Thread-Management. Es darf aber nicht vergessen werden, dass diese gering höher ausfallenden Zeiten durch die Ausnutzung mehrerer Rechenkerne für die gesamte Applikation mehr als kompensiert werden können. Es sei angemerkt, dass die gemessenen Zeiten nur die auf der GPU ausgeführten Arbeitsschritte des GPU-Thread beinhalten, in der Praxis wird aber auch noch Zeit für das Lesen und Schreiben aus der Queue benötigt. Das Transferieren der *Motion Mask* wird nur für Visualisierungszwecke benötigt. Es wird daher die Transferzeit jeweils mit bzw. ohne Transfer der Mask angegeben, da dieser Schritt eingespart werden kann.

In Tabelle 8.2 finden sich die Transfer- und Berechnungszeiten für das Testsystem mit der *nVidia GeForce 7950 GT* für verschiedene Texturformate. Man erkennt, dass die Zeiten sich proportional zur Rechengenauigkeit der verwendeten Texturformate erhöhen.

Um die Performancesteigerung, die bei Berechnungen durch die Auslagerung von Algorithmusschritten auf die GPU erreicht werden kann, messen zu können, wird eine reine

Abbildung 8.1: Visualisierung des Eingabevideos in der Applikation

Die in dieser Masterarbeit entstandene Applikation: Das GUI (Graphical User Interface) besteht aus dem Window-Manager, der Menu-Leiste, über die die Applikation steuerbar ist, dem Ausgabebereich zur Visualisierung der Ergebnisse, einem Button zum Pausieren des Systems und im unteren Fensterbereich Ausgaben bezüglich der gemessenen Ausführungszeiten. Im Ausgabebereich kann über ein *Tab Widget* (*Notebook Widget*) zwischen Eingabe-Video, Bewegungserkennung und *Object Tracking* gewählt werden. In diesem Screenshot sieht man die Visualisierung des Eingabe-Videos.

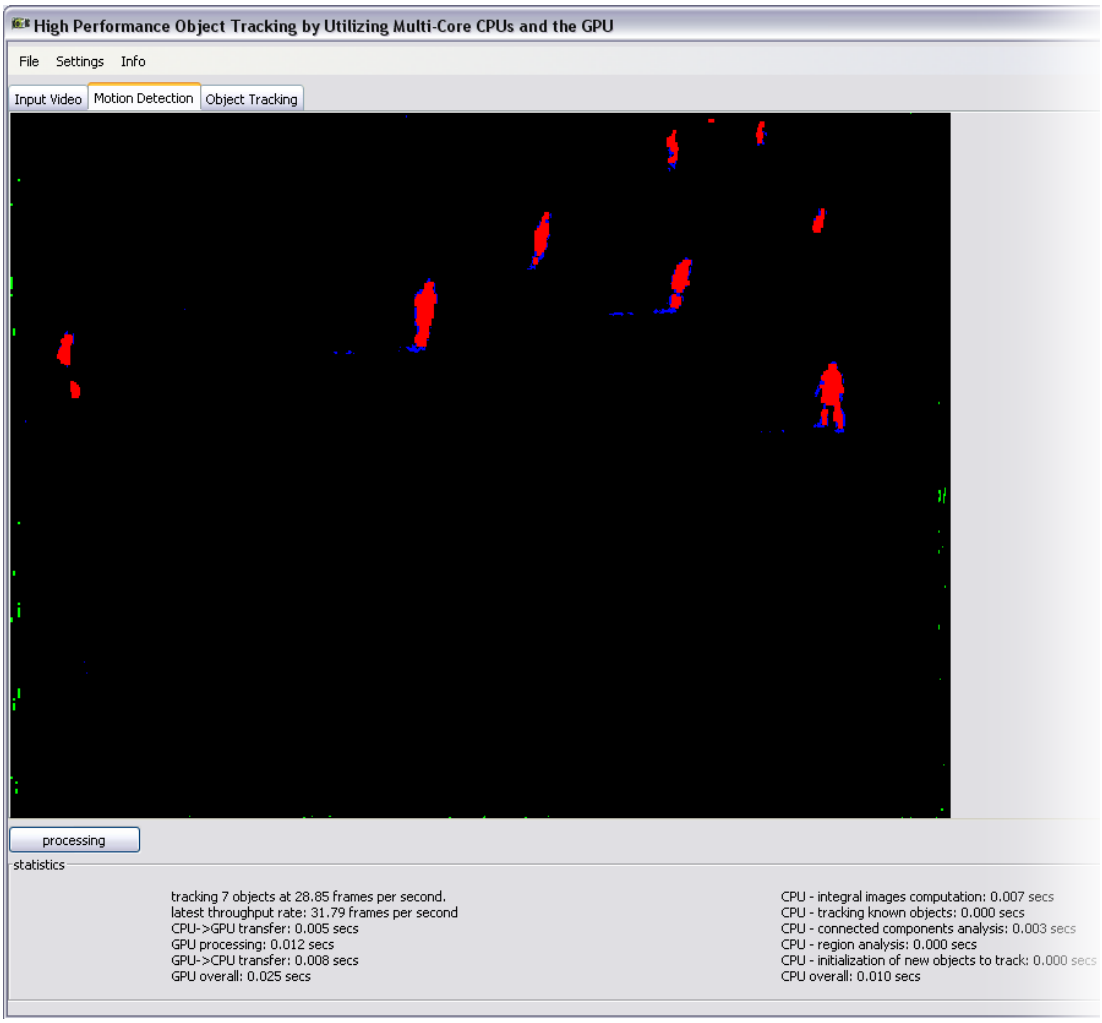


Abbildung 8.2: Visualisierte Ergebnisse der Bewegungserkennung in der Applikation

In diesem Screenshot sieht man die Ergebnisse der Bewegungserkennung visualisiert. Bei der Bewegungserkennung sind Pixel, die als zu einem bewegten Objekt gehörig klassifiziert werden, in rot visualisiert. Pixel, bei denen Schattenwurf detektiert wurde, werden in blau visualisiert, und bei grün gekennzeichneten Pixeln handelt es sich um Reflexionen, welche detektiert wurden.

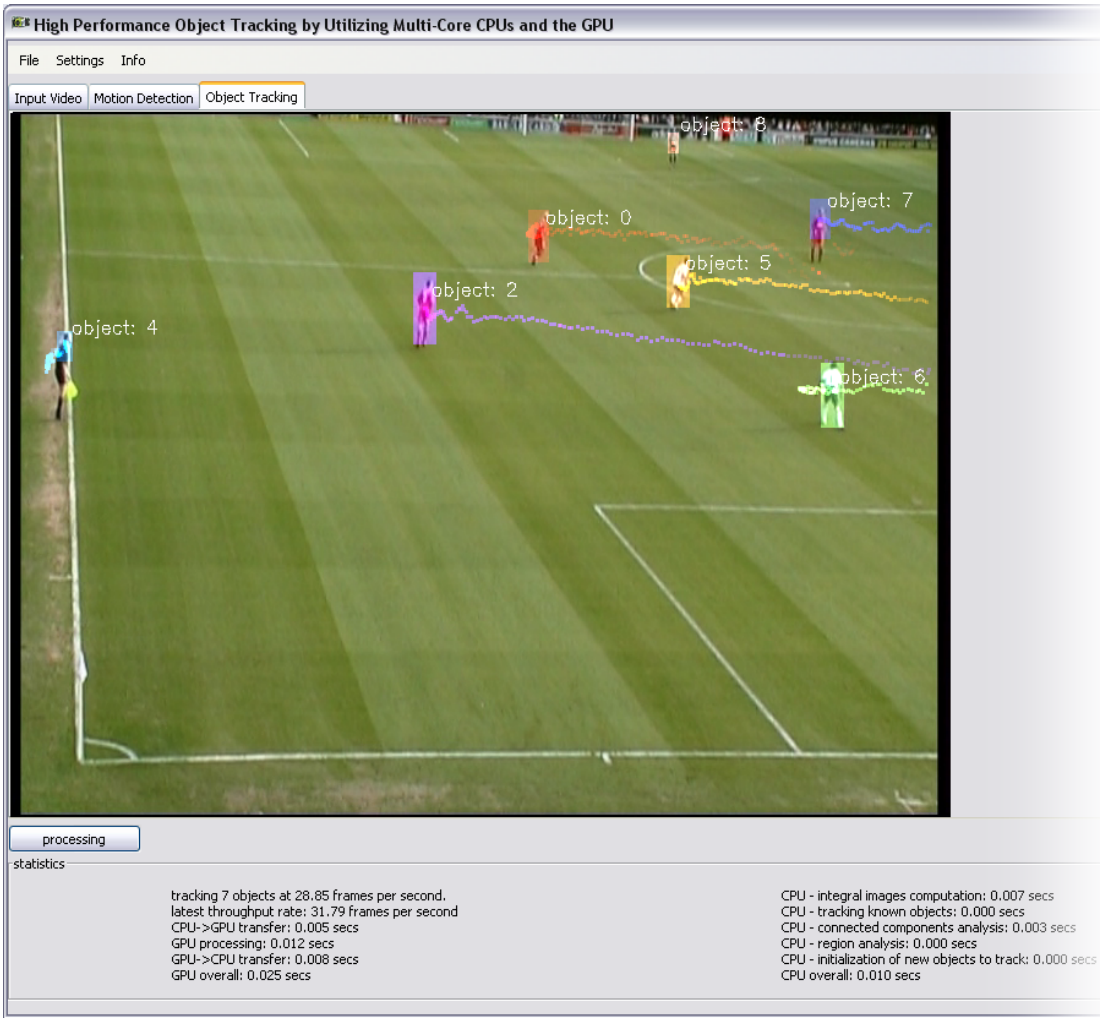


Abbildung 8.3: Visualisierung der Ergebnisse des *Object Tracking* in der Applikation

In diesem Screenshot sieht man die Ergebnisse des *Object Tracking* visualisiert. Die Ergebnisse des *Object Tracking* Algorithmus - also die getrackten Bildregionen - werden durch Überlagerung verschiedenfarbiger Rechtecke visualisiert, und die Trajektorien, die den zurückgelegten Weg, den ein Objekt genommen hat, repräsentieren, sind als entsprechend farbige Schlangenlinie realisiert.

GPU Transfer- und Ausführungszeiten

	Anwendungsschritt	Ausführungszeit
nVidia Geforce 6800 GS (*Single-Threaded* System)	CPU>GPU Transfer	~ 0.004194 sec
	Berechnungen	~ 0.021983 sec
	GPU>CPU Transfer (exkl. Motion Mask)	~ 0.003516 sec
	GPU>CPU Transfer (inkl. Motion Mask)	~ 0.007093 sec
nVidia Geforce 6800 GS (*Multi-Threaded* System)	CPU>GPU Transfer	~ 0.006874 sec
	Berechnungen	~ 0.022242 sec
	GPU>CPU Transfer (exkl. Motion Mask)	~ 0.003539 sec
	GPU>CPU Transfer (inkl. Motion Mask)	~ 0.007196 sec
nVidia Geforce 7950 GT (*Single-Threaded* System)	CPU>GPU Transfer	~ 0.003331 sec
	Berechnungen	~ 0.010687 sec
	GPU>CPU Transfer (exkl. Motion Mask)	~ 0.003142 sec
	GPU>CPU Transfer (inkl. Motion Mask)	~ 0.006051 sec
nVidia Geforce 7950 GT (*Multi-Threaded* System)	CPU>GPU Transfer	~ 0.007620 sec
	Berechnungen	~ 0.013543 sec
	GPU>CPU Transfer (exkl. Motion Mask)	~ 0.003119 sec
	GPU>CPU Transfer (inkl. Motion Mask)	~ 0.006262 sec

Tabelle 8.1: GPU Ausführungszeiten für die GPUs der beiden Testsysteme sowohl in der *Single-Threaded*- als auch der *Multi-Threaded*-Applikation.

Die gemessen Zeiten beziehen sich auf Videomaterial mit einer Auslösung von 768 × 576 Pixeln á 3 Farbkanälen.

Transfer- und Ausführungszeiten für verschiedene GPU-Formate (Geforce 7950 GT)		
Transfer-Format: RGBA8 Berechnungs-Format: RGBA8	Anwendungsschritt	Ausführungszeit
	CPU>GPU Transfer	~ 0.004299 sec
	Berechnungen	~ 0.004786 sec
	GPU>CPU Transfer	~ 0.003619 sec
Transfer-Format: RGBA8 Berechnungs-Format: RGBA16	Anwendungsschritt	Ausführungszeit
	CPU>GPU Transfer	~ 0.003300 sec
	Berechnungen	~ 0.004926 sec
	GPU>CPU Transfer	~ 0.003775 sec
Transfer-Format: RGBA16 Berechnungs-Format: RGBA16	Anwendungsschritt	Ausführungszeit
	CPU>GPU Transfer	~ 0.003332 sec
	Berechnungen	~ 0.004893 sec
	GPU>CPU Transfer	~ 0.003761 sec
Transfer-Format: RGBA8 Berechnungs-Format: RGBA32F	Anwendungsschritt	Ausführungszeit
	CPU>GPU Transfer	~ 0.003325 sec
	Berechnungen	~ 0.011130 sec
	GPU>CPU Transfer	~ 0.003724 sec
Transfer-Format: RGBA16 Berechnungs-Format: RGBA32F	Anwendungsschritt	Ausführungszeit
	CPU>GPU Transfer	~ 0.003324 sec
	Berechnungen	~ 0.011140 sec
	GPU>CPU Transfer	~ 0.003717 sec
Transfer-Format: RGBA32F Berechnungs-Format: RGBA32F	Anwendungsschritt	Ausführungszeit
	CPU>GPU Transfer	~ 0.008318 sec
	Berechnungen	~ 0.015849 sec
	GPU>CPU Transfer	~ 0.015718 sec
Transfer-Format: RGBA8 Berechnungs-Format: FLOAT_RGBA32_NV	Anwendungsschritt	Ausführungszeit
	CPU>GPU Transfer	~ 0.003407 sec
	Berechnungen	~ 0.011128 sec
	GPU>CPU Transfer	~ 0.003719 sec
Transfer-Format: RGBA16 Berechnungs-Format: FLOAT_RGBA32_NV	Anwendungsschritt	Ausführungszeit
	CPU>GPU Transfer	~ 0.003425 sec
	Berechnungen	~ 0.011300 sec
	GPU>CPU Transfer	~ 0.003713 sec
Transfer-Format: FLOAT_RGBA32_NV Berechnungs-Format: FLOAT_RGBA32_NV	Anwendungsschritt	Ausführungszeit
	CPU>GPU Transfer	~ 0.008342 sec
	Berechnungen	~ 0.015850 sec
	GPU>CPU Transfer	~ 0.015688 sec

Tabelle 8.2: Transfer- und Berechnungszeiten der auf die GPU ausgelagerten Verarbeitungsschritte für unterschiedliche Texturformate (Geforce 7950 GT)

Die Messungen beziehen sich auf Videomaterial mit einer
Auflösung von 768 × 576 Pixeln bei 3 Farbkanälen (Rücktransfer ohne Motion Mask).

CPU-Variante der Bewegungserkennung implementiert. Es wird ein Verarbeitungsschritt stellvertretend für alle weiteren genauer betrachtet (die Farbraumkonvertierung von RGB nach IHLS, die als erster Schritt im Bewegungserkennungs-Algorithmus durchgeführt werden muss) - genauer in dem Sinne, dass eine Standard CPU Implementierung des Algorithmusschrittes implementiert wird, die über gewöhnliche Schleifendurchläufe und Speicherzugriffe realisiert wird.
Weiters wird eine CPU-Variante programmiert, die durch Verwendung von OpenCV-Funktionen von CPU-Befehlssatzerweiterungen wie MMX (*Multi Media Extension*) bzw. SSE (*Streaming SIMD (Single Instruction Multiple Data) Extensions*) profitiert, womit eine erhöhte Ausführungsgeschwindigkeit von Berechnungsschritten erreicht werden kann.
Diese beiden CPU-Varianten wurden mit der GPU-Variante bezüglich Ausführungszeiten verglichen. Die Source-Codes für die beiden CPU-Varianten der Farbraumkonvertierung sowie die GPU-Variante als Shadercode finden sich im Appendix auf Seite 116. Die Ergebnisse sind in Tabelle 8.3 enthalten.
Zu beachten ist, dass die Zeit für die Berechnung auf der GPU sowohl ohne als auch mit Transferzeiten angeführt wird. Der Transfer muss allerdings nur einmal vor allen Berechnungen und einmal nach allen Berechnungen für das zu bearbeitende Bild vollzogen werden, daher wird mit jedem weiteren Berechnungsschritt diesem Nachteil sozusagen entgegengewirkt.
Man sieht, dass die Performancesteigerung durch die Verwendung der GPU unter Berücksichtigung der Transferzeit immer noch um den Faktor 11.5 bzw. 7.5 (für die beiden Testsysteme) schneller ist als die SSE-optimierte CPU-Variante.
Bedenkt man nun, dass die Transferzeiten nur einmal für alle Berechnungsschritte anfallen und hier nur die Farbraumkonvertierung betrachtet wurde, wird dieser Faktor mit jeder weiteren Berechnung größer statt kleiner!
Zur Standard-CPU-Implementierung fällt der Unterschied noch gravierender aus mit Geschwindigkeitszuwächsen um den Faktor 16.5 bzw. 21.8 für beide Testsysteme.

Die OpenCV-Variante profitiert - wie man den Ergebnissen aus Tabelle 8.3 entnehmen kann - stärker vom Intel Prozessor als vom AMD Athlon Prozessor, obwohl letzterer höher getaktet ist (man beachte, dass beim Intel Prozessor nur ein Kern zur Berechnung herangezogen wird). Die höhere Taktrate des AMD Athlon64 Prozessors schlägt sich in der schnelleren Ausführungsgeschwindigkeit bei der Standard-CPU-Variante nieder. Ebenso erkennt man die höhere Rechenleistung der *nVidia GeForce 7950 GT* GPU im Vergleich zur *nVidia GeForce 6800 GS* GPU - die Rechenleistung der GPU hängt nur unwesentlich von dem im System befindlichen Prozessor ab und sollte bei einem eventuellen Tausch auf anderen Prozessoren ähnliche Ergebnisse liefern.

Berechnungszeit für die Farbraumkonvertierung RGB nach IHLS		
AMD Athlon64 3700+ (2.38 Ghz)		Ausführungszeit
	CPU Impl. (Standard)	$\sim$ 0.144459 sec
	CPU Impl. (SSE,MMX)	$\sim$ 0.100974 sec
	GPU Impl. (Geforce 6800 GS)	$\sim$ 0.001036 sec
	GPU Impl. (inkl. Transfer)	$\sim$ 0.008746 sec
Intel Core2 Duo 6300 (1.86 Ghz)		Ausführungszeit
	CPU Impl. (Standard)	$\sim$ 0.160352 sec
	CPU Impl. (SSE,MMX)	$\sim$ 0.055105 sec
	GPU Impl. (Geforce 7950 GT)	$\sim$ 0.000859 sec
	GPU Impl. (inkl. Transfer)	$\sim$ 0.007332 sec

Tabelle 8.3: Berechnungszeit für die Farbraumkonvertierung RGB nach IHLS auf den CPUs bzw. GPUs der Testsysteme

(Videomaterial mit einer Auslösung von 768 $\times$ 576 Pixeln bei 3 Farbkanälen, durchschnittliche Zeit bei 1000 Iterationen)

Sämtliche Verarbeitungsschritte, die auf die GPU ausgelagert werden, und ihre entsprechenden Ausführungszeiten sind in Tabelle 8.4 angeführt. Verglichen wurden die Ausführungszeiten für die beiden CPUs sowie die beiden GPUs der beiden Testsysteme. Die CPU-Implementierung ist mittels der OpenCV-Funktionen optimiert, um MMX- bzw. SSE-Funktionalitäten der Prozessoren auszunutzen. Mitangeführt sind auch hier wieder Transferzeiten (ohne *Motion Mask*) für die GPU-Variante bzw. Datentypkonvertierungszeiten für die CPU-Variante. Die einzelnen Verarbeitungsschritte, die auf die GPU ausgelagert werden, sind die eigentliche *Motion Detection*, das Nachführen des Hintergrundmodells mit unterschiedlichen Updateraten für Vorder- bzw. Hintergrundpixel, morphologisches Closing und Opening, um Löcher zu stopfen und isolierte Pixel zu eliminieren, die

Anwendung der *Motion Mask* auf das Eingangsbild (Nullsetzen von in der *Motion Mask* nicht gesetzten Pixeln im Eingabebild) und schließlich die Berechnung der Momentbilder bezüglich x- und y- Koordinate, die vom *Mean Shift Tracker*, basierend auf Integralbildern, später benötigt werden.

Berechnungszeiten der GPU-Berechnungen im Vergleich zur CPU-Variante

AMD Athlon64 3700+ (2.38 Ghz)

Verarbeitungsschritt	Ausführungszeit
Hin-Transfer bzw. Konvertierung	~ 0.000003 sec
Bewegungserkennung	~ 0.198261 sec
Nachführen des Hintergrundmodells	~ 0.130180 sec
morphologisches Closing	~ 0.033783 sec
morphologisches Opening	~ 0.033774 sec
Anwendung der Motion Mask	~ 0.002026 sec
Berechnung der Momentbilder	~ 0.017669 sec
Rück-Transfer bzw. Konvertierung	~ 0.008136 sec
gesamt	~ 0.423832 sec

Intel Core2 Duo 6300 (1.86 Ghz)

Verarbeitungsschritt	Ausführungszeit
Hin-Transfer bzw. Konvertierung	~ 0.000001 sec
Bewegungserkennung	~ 0.135886 sec
Nachführen des Hintergrundmodells	~ 0.079205 sec
morphologisches Closing	~ 0.021774 sec
morphologisches Opening	~ 0.021622 sec
Anwendung der Motion Mask	~ 0.000976 sec
Berechnung der Momentbilder	~ 0.012964 sec
Rück-Transfer bzw. Konvertierung	~ 0.007923 sec
gesamt	~ 0.280351 sec

nVidia Geforce 6800 GS

Verarbeitungsschritt	Ausführungszeit
Hin-Transfer bzw. Konvertierung	~ 0.004347 sec
Bewegungserkennung	~ 0.006382 sec
Nachführen des Hintergrundmodells	~ 0.010972 sec
morphologisches Closing	~ 0.002301 sec
morphologisches Opening	~ 0.002257 sec
Anwendung der Motion Mask	~ 0.000266 sec
Berechnung der Momentbilder	~ 0.000309 sec
Rück-Transfer bzw. Konvertierung	~ 0.003646 sec
gesamt	~ 0.030480 sec

nVidia Geforce 7950 GT

Verarbeitungsschritt	Ausführungszeit
Hin-Transfer bzw. Konvertierung	~ 0.003343 sec
Bewegungserkennung	~ 0.002563 sec
Nachführen des Hintergrundmodells	~ 0.003928 sec
morphologisches Closing	~ 0.002133 sec
morphologisches Opening	~ 0.002032 sec
Anwendung der Motion Mask	~ 0.000403 sec
Berechnung der Momentbilder	~ 0.000376 sec
Rück-Transfer bzw. Konvertierung	~ 0.003163 sec
gesamt	~ 0.017941 sec

Tabelle 8.4: Vergleich der Berechnungszeiten: GPU- bzw. CPU-Variante

Zur grafischen Veranschaulichung werden in Abbildung 8.4 die für die Umsetzung auf der GPU geeigneten Berechnungen für die CPUs und GPUs der Testsysteme gegenübergestellt. Man sieht die deutliche Performancesteigerung der GPU-Variante und speziell den Einbruch der CPU-Variante bei größer werdenden Bildgrößen. Für kleine Bildgrößen ist der Vorteil der GPU-Implementierung noch nicht so stark, bei großen Bildgrößen sieht man aber die Stärke des Ansatzes. Besonders bei großen Bildgrößen wird der Vorteil der stark parallel verarbeiteten GPU-Implementierung deutlich, die dann ihre Vorzüge deutlich auszuspielen vermag und wesentlich weniger Zeit benötigt als die CPU-Implementierung.

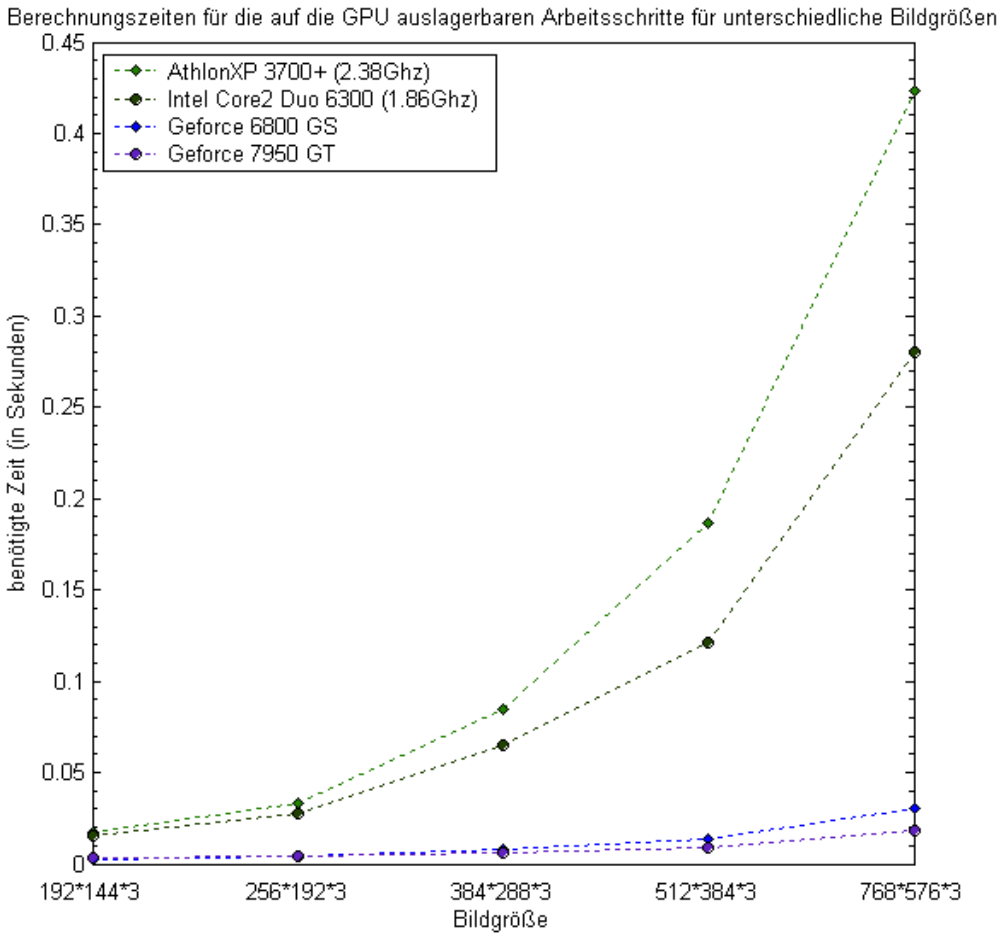


Abbildung 8.4: Berechnungszeiten für die auf die GPU auslagerbaren Verarbeitungsschritte für unterschiedliche Bildgrößen für die CPUs bzw. GPUs der beiden Testsysteme

8.3 Zeitmessungen des *Object Tracking* Algorithmus auf der CPU

Die Zeiten für die Berechnung der Algorithmusschritte des *Mean Shift Trackers*, basierend auf Integralbildern, die auf der CPU berechnet werden, sind in Tabelle 8.5 angeführt. Die Integralbild-Berechnung nimmt dabei zusammen mit der *Connected Components Analysis* die meiste Zeit in Anspruch, ist jedoch für gegebene Bildgrößen konstant bzw. annähernd konstant [CCL04]. Das Tracken bekannter Objekte ist mit deutlich weniger als einer Millisekunde kein limitierender Faktor. Für eine unterschiedliche Anzahl der zu trackenden Objekte ändert sich das Ergebnis praktisch nicht - pro 100 Objekten in der vierten Nachkommastelle (Millisekundenbereich). Auch die Regionsanalyse (*Data Association*) und das Initialisieren neuer Objekte benötigt nahezu keine Zeit.

Ausführungszeiten für die CPU-Arbeitsschritte auf den beiden Testsystemen

	Anwendungsschritt	Ausführungszeit
AMD Athlon64 3700+ (2.38 Ghz)	Integralbild-Berechnung	~ 0.007103 sec
	Tracking bereits bekannter Objekte	~ 0.000110 sec
	Connected Components Analysis	~ 0.006463 sec
	Regionsanalyse	~ 0.000012 sec
	Initialisierung neuer Objekte	~ 0.000017 sec
Intel Core2 Duo 6300 (1.86 Ghz)	Integralbild-Berechnung	~ 0.007657 sec
	Tracking bereits bekannter Objekte	~ 0.000030 sec
	Connected Components Analysis	~ 0.006456 sec
	Regionsanalyse	~ 0.000014 sec
	Initialisierung neuer Objekte	~ 0.000001 sec

Tabelle 8.5: CPU-Ausführungszeiten für die beiden Testsysteme

(gemessen mit dem Performance Counter im Multi-Threaded System bei 1000 Schleifendurchläufen)

8.4 Performance des Gesamtsystems

Die gemessene durchschnittliche Anzahl an Bildern, die pro Sekunde verarbeitet werden kann, also die Durchsatzrate, findet sich in Tabelle 8.6. Verglichen werden für die beiden Testsysteme die *Single-Threaded*- und *Multi-Threaded*-Variante der Applikation jeweils mit GPU-Unterstützung oder die reine CPU-Implementierung. Die CPU-Auslastung zeigt an, wieviel Prozent der verfügbaren Rechenleistung des Prozessors, die durch alle verfügbaren Kerne gebildet wird, durch die Applikation ausgelastet werden.

Durchsatzraten des gesamten *Single-* bzw. *Multi-Threaded* Systems			
AMD Athlon64 3700+ (2.38 Ghz) nVidia Geforce 6800 GS *Single-Threaded*	Berechnungen auf	Durchsatzrate	CPU-Last
	nur CPU	2.212613 fps	100%
	CPU und GPU	15.780183 fps	100%
Intel Core2 Duo 6300 (1.86 Ghz) nVidia Geforce 7950 GT *Single-Threaded*	Berechnungen auf	Durchsatzrate	CPU-Last
	nur CPU	3.166214 fps	51%
	CPU und GPU	22.573725 fps	52%
AMD Athlon64 3700+ (2.38 Ghz) nVidia Geforce 6800 GS *Multi-Threaded*	Berechnungen auf	Durchsatzrate	CPU-Last
	nur CPU	2.246152 fps	100%
	CPU und GPU	15.041123 fps	100%
Intel Core2 Duo 6300 (1.86 Ghz) nVidia Geforce 7950 GT *Multi-Threaded*	Berechnungen auf	Durchsatzrate	CPU-Last
	nur CPU	3.246782 fps	56%
	CPU und GPU	28.668072 fps	85%

Tabelle 8.6: Durchsatzraten der *Single-* bzw. *Multi-Threaded* Variante in Kombination mit GPU-unterstützter Variante bzw. mit reiner CPU-Implementierung der Applikation auf den beiden Testsystemen im Vergleich

Messungen durchgeführt für Videomaterial mit einer Auslösung von 768 × 576 Pixeln bei 3 Farbkanälen.

Beim *AMD Athlon64* Testsystem, dessen Prozessor nur einen Kern besitzt, beträgt die CPU-Auslastung freilich in sämtlichen Szenarien 100%. Man sieht zwar, dass durch Verwendung der GPU auch hier ein deutlicher Geschwindigkeitsschub realisiert wurde, allerdings profitiert bei diesem System die reine CPU-Variante nicht vom *Multi-Threaded*-Ansatz - das System erreicht sogar etwas weniger Durchsatz als die *Single-Threaded*-Variante - und die GPU-Variante kann gar keinen Nutzen daraus ziehen, da, wie in Abschnitt 7.4 beschrieben wurde, CPU-GPU Parallelismus bei Datentransfers aufgrund der

dort genannten Einschränkungen nicht erreicht werden konnte.
Beim *Intel Core2 Duo* Prozessor des anderen Testsystems stellt sich die Situation folgendermaßen dar: Die CPU-Auslastung beträgt sowohl für die reine CPU-Implementierung als auch die GPU-unterstützte Variante der *Single-Threaded* Applikation 51% bzw. 52%, was so viel heißt, dass nur ein Prozessorkern zur Berechnung der Algorithmen herangezogen wird. Die zusätzlichen 1% bzw. 2% Auslastung kommen von der *Gtk+ library*, die einen eigenen GUI Thread verwendet.

Die *Multi-Threaded* Applikation erreicht eine Steigerung zur *Single-Threaded*-Variante, da nun der zweite Kern ebenfalls für die Berechnung der Algorithmen genutzt wird. Bei der reinen CPU-Variante fällt diese Geschwindigkeitssteigerung allerdings nur sehr gering aus, dies deshalb, weil das *Multi-Threaded* System ja so konzipiert wurde, dass sowohl die Bewegungserkennung als auch das *Tracking* jeweils als eigener Thread realisiert wurden. Da aber - wie in Tabelle 8.4 zu sehen ist, die Bewegungserkennung auf der CPU um ein Vielfaches länger dauert als das *Tracking* (siehe dazu Tabelle 8.5), wartet hier der *Tracking*-Thread immer auf den Bewegungserkennungs-Thread und dessen unverhältnismäßig lange Berechnungszeit bremst das restliche System aus. Man müsste also die Bewegungserkennung in mehrere einzelne Threads unterteilen, die sich dann die Arbeit aufteilen könnten. Dies würde zu einer höheren CPU-Auslastung und somit einer gesteigerten Durchsatzrate führen, die sich so bis zu circa 6 Bildern pro Sekunde erhöhen ließe.
Die GPU-unterstützte *Multi-Threaded* Applikation erreicht am *Intel Core2 Duo* System dank des 2-Kern-Prozessors die beste Durchsatzrate bei einer CPU-Auslastung von 85% (die volle Auslastung wird auch hier deshalb nicht erreicht, da der *Tracking*-Thread auf den Bewegungserkennungs-Thread warten muss), das heißt, der zweite Prozessorkern ist nur zu 72% ausgelastet und hätte noch ungenutzte Leistung für weitere Berechnungen.
Zu beachten ist, dass die Auslastung der CPU auch zu einem Teil durch das Dekomprimieren des Videos entsteht, das, um Speicherplatz zu sparen, mit dem *DivX*-Codec kodiert wird, und das Dekodieren von *DivX*-Videos auch Rechenzeit kostet. Das Visualisieren der Ergebnisse im GUI erfordert weitere Rechenleistung. Zusätzliche Zeit könnte eingespart werden, wenn die *Motion Mask* nicht übertragen wird. Dennoch werden diese Punkte als Teil des Systems betrachtet, und so ist es umso erfreulicher, dass dennoch Echtzeitfähigkeit erreicht werden kann.
Vergleicht man im GPU-unterstützten *Multi-Threaded* System die Durchsatzraten der beiden Systeme, so sind dies bei PAL Video-Material (768×576 Bildpunkte bei jeweils 3 Farbkanälen) im *Intel Core2 Duo* System 28.668072 Bilder pro Sekunde und im *AMD Athlon64* System 15.041123 Bilder pro Sekunde. Der gravierende Unterschied entsteht durch

die Tatsache, dass das *Multi-Threaded* System vom Mehrkernprozessor deutlich profitiert und im Gegensatz dazu auf einem Einkernprozessor Arbeitsschritte nicht effizient parallel berechnet werden können. Bei den CPU-Verarbeitungsschritten ist sogar das *AMD Athlon64* Testsystem bei naiver Implementierung durch die höhere Taktzahl leicht im Vorteil, die SSE-optimierte Variante erreicht auf dem Intel Prozessor bessere Werte (siehe Tabelle 8.3).
Der Umstand von parallel arbeitenden Kernen in einem Multikernsystem führt aber zu einem um einige Größenordnungen gesteigerten Geschwindigkeitsschub für das Gesamtsystem. Auch ist am *AMD Athlon64* System nur die *nVidia GeForce 6800 GS* eingebaut, während im *Intel Core2 Duo* System die *nVidia GeForce 7950 GT* ihre Arbeit verrichtet.

Zwar benötigt die *nVidia GeForce7950 GT* mit etwa 25 Millisekunden um etwa 8 Millisekunden weniger Zeit als die *nVidia GeForce 6800 GS* für dieselben GPU-Verarbeitungsschritte im *Multi-Threaded* System (siehe Tabelle 8.1 für genaue Zeitmessungen), allerdings ist ein weiterer Grund für die mehr als doppelt so lange Ausführungszeit auf dem *AMD Athlon64* Testsystem das Fehlen eines zweiten Prozessorkerns für das parallele Verarbeiten von Berechnungen, wie dies im *Intel Core2 Duo* System geschieht.
Würde der GPU Thread nicht während Datentransfers blockieren (siehe Seite 91 in Abschnitt 7.4 zur Behandlung der Problematik), so könnte man auch auf dem *AMD Athlon64* Testsystem bessere Durchsatzraten erreichen und auf dem *Intel Core2 Duo* System einen Prozessorkern entlasten und somit andere Aufgaben darauf ausführen.
Verwendet man gar nur geringere Videoauflösungen, lassen sich noch deutlich bessere Durchsatzraten erreichen. Eine Gegenüberstellung für verschiedene Bildgrößen sieht man in Abbildung 8.5. Man sieht, dass sich die Ausnutzung der GPU für die zur Auslagerung gewählten Berechnungsschritte bei beiden Testsystemen deutlich bemerkbar macht. Weiters kann man erkennen, dass auch der *Multi-Threading*-Ansatz auf dem Mehrkernsystem zu deutlichen Leistungssteigerungen führt.

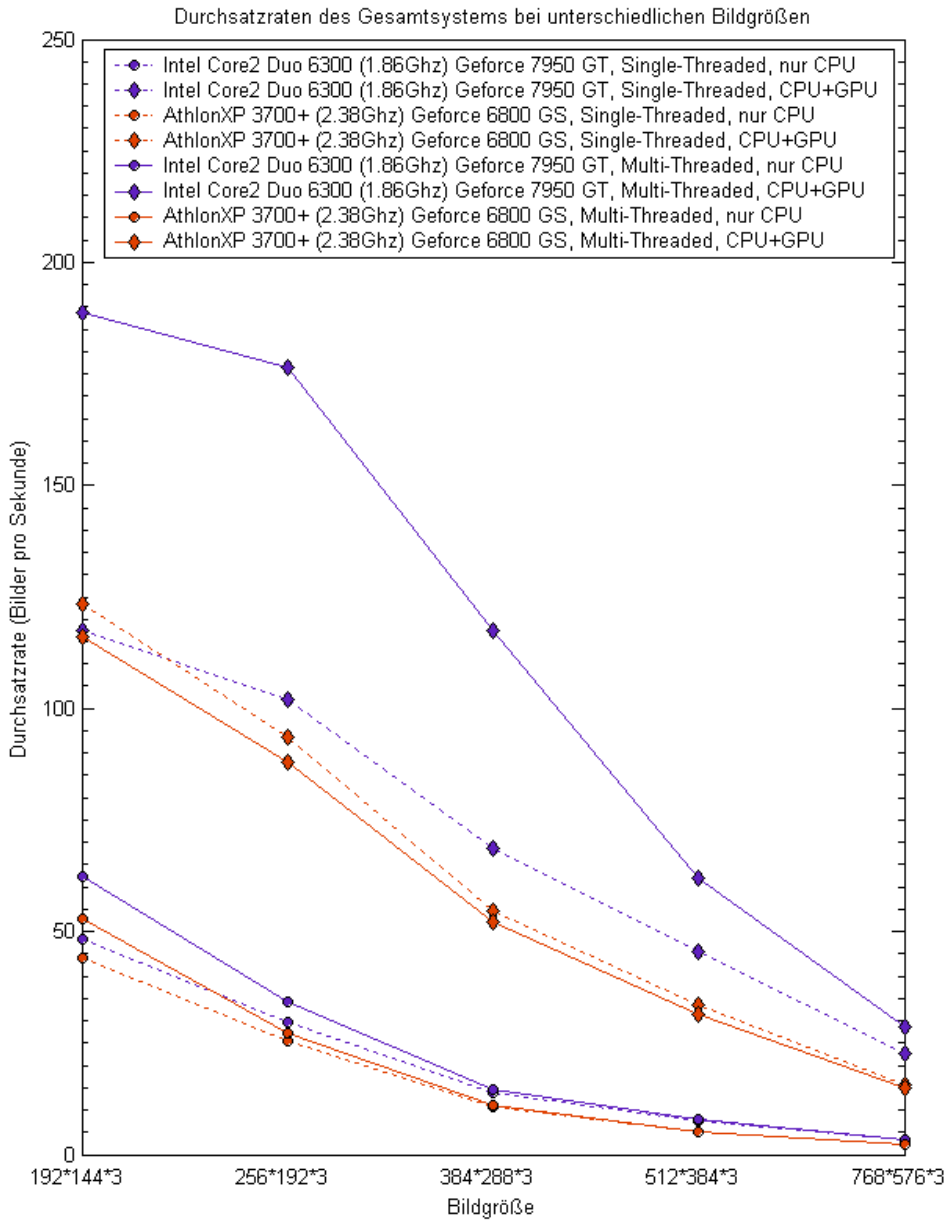


Abbildung 8.5: Durchsatzraten des Gesamtsystems für unterschiedliche Bildgrößen für sämtliche Kombinationen aus *Single-Threaded* bzw. *Multi-Threaded* und reiner CPU- bzw. GPU-unterstützter Implementierung für die beiden Testsysteme

Kapitel 9

Zusammenfassung und Schlussbetrachtung

In dieser Masterarbeit wurde ein voll automatisiertes *Object Tracking* System, das auf der Verwendung von Videomaterial basiert, welches durch eine statische Kamera aufgenommen wurde, mit dem Schwerpunkt Echtzeitanforderung implementiert und evaluiert.
Zur Erreichung der gesteckten Ziele wurden zwei wesentliche Mechanismen verwendet: einerseits wurden rechenintensive Algorithmen auf die Grafikkarte ausgelagert, andererseits wurden mittels *Multi-Threaded*-Programmierung Mehrkernprozessoren ausgenutzt.

Es konnte gezeigt werden, dass mit diesen Mechanismen das Tracken von mehreren hundert Objekten in Echtzeit für Videomaterial in einer Auflösung von 768×576 Pixeln bei 3 Farbkanälen möglich ist. Dabei werden neue Objekte nicht nur in Portalen sondern im gesamten Bild gesucht. Gegenüber einer optimierten CPU-Implementierung des Systems wird eine Geschwindigkeitssteigerung um mehr als den Faktor 9 erreicht. Dieser Faktor entspricht auch der Leistungssteigerung, wie sie mit dem System aus [BFB04]) erreicht wird. Die verwendeten Algorithmen sind rechenintensiv, dennoch können sie mit Hilfe der vorgestellten Techniken in Echtzeit berechnet werden, und das, ohne Abstriche bei der Bildauflösung machen zu müssen oder auf die Farbinformation der Videodaten verzichten zu müssen.
Die Systemanforderungen sind durch eine aktuelle Multi-Core CPU und eine aktuelle Grafikkarte definiert und sind somit entsprechend kostengünstig.
Blickt man in die Zukunft, so kann davon ausgegangen werden, dass sich die Rechenleistung von Grafikkarten weiter erhöhen wird. Gleichzeitig werden nach und nach weitere Restriktionen fallen, die momentan bei der Programmierung von GPGPU-Programmen noch Schwierigkeiten und Probleme bereiten können.

Die Entwicklung bei Prozessoren zeigt einen klaren Trend in Richtung Erhöhung der Anzahl der Rechenkerne, da der Geschwindigkeitssteigerung durch bloße Erhöhung der Taktfrequenz thermisch bedingte Grenzen gesetzt sind.
Beide Entwicklungen sind daher positiv für die vorgestellten Techniken zur Performancesteigerung zu sehen. Es kann davon ausgegangen werden, dass das vorgestellte System durch zukünftige Entwicklungen stark profitieren wird.

Als Verbesserungen für das vorgestellte System sollen hier nun einige mögliche Weiterentwicklungen skizziert werden.
Einen zusätzlichen Geschwindigkeitsschub könnte man erreichen, wenn man das in Abschnitt 7.4 angesprochene Problem der doppelt so langsamen asynchronen Datentransferzeiten - verglichen mit synchronen Datentransferzeiten - lösen würde, um so zu erreichen, dass CPU und GPU komplett parallel Berechnungen durchführen und während eines Datentransfers der zuständige CPU-Kern nicht mehr blockiert.
Die Bewegungserkennung könnte durch Integration eines *Mixture of Gaussians Models* robuster gemacht werden. Verdeckungsprobleme können mittels Prädikationsfiltern und verbesserter *Data Association* gelöst werden, indem verschmelzende Regionen als Gruppe von Objekten eine separate Behandlung erfahren könnten und bei erneuter Aufteilung in mehrere Objekte diese durch Vergleich mit den geschätzten Positionen des Prädikationsfilters wiedergefunden werden könnten. Eine weitere Möglichkeit wäre, Regionsmerkmale in die Berechnungen miteinzubeziehen, um Objekte nach der erneuten Aufteilung wiederzuerkennen.

Appendix

Codebeispiele

Es folgen Codebeispiele für die Farbraumkonvertierung vom RGB-Farbraum in den IHLS-Farbraum.

Das verwendete Cg-Shaderprogramm für die Farbraumumrechnung:

```
half4 rgb2ihls(half2 coords : TEX0,
               uniform samplerRECT texture): COLOR {
  half4 t = texRECT(texture, coords);
  half4 result = {0.0,0.0,0.0,0.0};
    // calculate luminance
  result.x = 0.2125*t.x + 0.7154*t.y + 0.0721*t.z;
    // calculate saturation
  result.y = max(t.x,max(t.y,t.z))-min(t.x,min(t.y,t.z));
  half2 cr;
    // x component for hue vector computation
  cr.x = t.x-(t.y+t.z)*0.5;
    // y component for hue vector computation
  cr.y = 0.8660254*(t.z-t.y);
    // length computation of hue vector for normalization
  half cr_dist = length(cr.xy);
  half div_cr;
    // use reciprocal to avoid 2 divisions by doing
    // one division here and later only 2 multiplications
  if (cr_dist>0.0) div_cr = 1.0/cr_dist; else div_cr=1.0;
    // weight hue vector components with saturation
  result.zw = (cr.xy*div_cr)*result.y;
  return(result);
}
```

Listing 9.1: Farbraumumrechnung RGB zu IHLS als Cg-Shadercode

Das Bild muss zuvor als Textur in den Grafikspeicher geladen werden und dient als

Input (*samplerRECT*). Der Datentyp *samplerRECT* definiert eine 2D Textur. Im Gegensatz zu *sampler2D* werden Texturkoordinaten als ganzzahlige Indexwerte verwendet und müssen somit nicht normalisiert werden.

Das Standard-CPU-Variante für die Farbraumumrechnung in C/C++:

```
1  void md_cpu::rgb2ihls() {
2    cvConvertScale(input_img,img,1.0/255.0,0);
3    int Step = ihls_img_luminance->widthStep/ihls_img_luminance->width;
4    float* data = (float*)img->imageData;
5    float* target1 = (float*)ihls_img_luminance->imageData;
6    float* target2 = (float*)ihls_img_saturation->imageData;
7    float* target3 = (float*)ihls_img_chrominance_x->imageData;
8    float* target4 = (float*)ihls_img_chrominance_y->imageData;
9    for (int y=0; y<img->height; y++) {
10     for (int x=0; x<img->width; x++) {
11         // luminance computation
12       target1[ihls_img_luminance->width*y+x] =
13         0.0721*data[(img->width*y+x)*3] +
14         0.7150*data[(img->width*y+x)*3+1] +
15         0.2125*data[(img->width*y+x)*3+2];
16
17         // saturation computation
18       float M = max(data[(img->width*y+x)*3],
19                     data[(img->width*y+x)*3+1]);
20       M = max(M,data[(img->width*y+x)*3+2]);
21       float m = min(data[(img->width*y+x)*3],
22                     data[(img->width*y+x)*3+1]);
23       m = min(m,data[(img->width*y+x)*3+2]);
24       target2[ihls_img_luminance->width*y+x] = M-m;
25
26         // hue vector computation
27       target3[ihls_img_luminance->width*y+x] =
28         data[(img->width*y+x)*3+2]-
29         (data[(img->width*y+x)*3+1]+data[(img->width*y+x)*3])*0.5;
30       target4[ihls_img_luminance->width*y+x] = 0.86602540*
31         (data[(img->width*y+x)*3]-data[(img->width*y+x)*3+1]);
32         // compute length of hue vector for normalization
33       float cr = sqrt(  target3[ihls_img_luminance->width*y+x]*
34                         target3[ihls_img_luminance->width*y+x]+
35                         target4[ihls_img_luminance->width*y+x]*
36                         target4[ihls_img_luminance->width*y+x]);
37         // use reciprocal avoid 2 divisions by doing
38         // one division here and later only 2 multiplications
39       float div = 1.0/cr;
40         // weight hue vector components with saturation
41       target3[ihls_img_luminance->width*y+x] =
42         target3[ihls_img_luminance->width*y+x]*div*
```

```
43            target2[ihls_img_luminance->width*y+x];
44          target4[ihls_img_luminance->width*y+x] =
45            target4[ihls_img_luminance->width*y+x]*div*
46            target2[ihls_img_luminance->width*y+x];
47        }
48      }
49  }
```

Listing 9.2: Farbraumumrechnung RGB zu IHLS als Standard C/C++ Code

Die OpenCV-(SSE-)optimierte CPU-Variante für die Farbraumumrechnung:

```
1  void md_opencv_cpu::rgb2ihls() {
2    cvConvertScale(input_img,img,1.0/255.0,0);
3    cvSplit(img, img_b, img_g, img_r,NULL);
4
5      // luminance computation
6    cvConvertScale(img_r,tmp_img_r,0.2125,0);
7    cvConvertScale(img_g,tmp_img_g,0.715,0);
8    cvConvertScale(img_b,tmp_img_b,0.0721,0);
9    cvAdd(tmp_img_r,tmp_img_g,ihls_img_luminance,NULL);
10   cvAdd(tmp_img_b,ihls_img_luminance,ihls_img_luminance,NULL);
11
12     // saturation computation
13   cvMax(img_r,img_g,tmp_img_r);
14   cvMax(img_b,tmp_img_r, tmp_img_r);
15   cvMin(img_r,img_g,tmp_img_g);
16   cvMin(img_b,tmp_img_g, tmp_img_g);
17   cvSub(tmp_img_r,tmp_img_g,ihls_img_saturation,NULL);
18
19     // hue vector computation
20   cvAdd(img_g,img_b,ihls_img_chrominance_x,NULL);
21   cvConvertScale( ihls_img_chrominance_x,
22                   ihls_img_chrominance_x,0.5,0);
23   cvSub(img_r,ihls_img_chrominance_x,ihls_img_chrominance_x,NULL);
24
25   cvSub(img_b,img_g,ihls_img_chrominance_y,NULL);
26   cvConvertScale( ihls_img_chrominance_y,
27                   ihls_img_chrominance_y,0.86602540,0);
28
29     // compute length of hue vector for normalization
30   cvMul(ihls_img_chrominance_x,ihls_img_chrominance_x,tmp_img_r);
31   cvMul(ihls_img_chrominance_y,ihls_img_chrominance_y,tmp_img_g);
32   cvAdd(tmp_img_r,tmp_img_g,tmp_img_r);
33   cvPow(tmp_img_r,tmp_img_r,0.5);
34
35     // use reciprocal avoid 2 divisions by doing
36     // one division here and later only 2 multiplications
37   cvDiv(NULL,tmp_img_r,tmp_img_g,1.0);
38     // weight hue vector components with saturation
39   cvMul(ihls_img_chrominance_x,tmp_img_g,ihls_img_chrominance_x);
40   cvMul(ihls_img_chrominance_x,
41         ihls_img_saturation,
42         ihls_img_chrominance_x);
```

```
43      cvMul(ihls_img_chrominance_y,tmp_img_g,ihls_img_chrominance_y);
44      cvMul(ihls_img_chrominance_y,
45              ihls_img_saturation,
46              ihls_img_chrominance_y);
47  }
```

Listing 9.3: Farbraumumrechnung RGB zu IHLS als OpenCV(SSE) C/C++ Code

Weblinks

[1] Boost C++ Libraries.
http://www.boost.org/, letzter Besuch: 20. Mai 2007.

[2] GLEW - The OpenGL Extension Wrangler Library.
http://glew.sourceforge.net/, letzter Besuch: 20. Mai 2007.

[3] GLUT. http://www.xmission.com/~nate/glut.html, letzter Besuch: 20. Mai 2007.

[4] GPGPU - General-Purpose Computation Using Graphics Hardware.
http://www.gpgpu.org/, letzter Besuch: 20. Mai 2007.

[5] GTK+. http://www.gtk.org/, letzter Besuch: 20. Mai 2007.

[6] pthreads - POSIX Threads for Win32.
http://sourceware.org/pthreads-win32/, letzter Besuch: 20. Mai 2007.

[7] AMD. CTM - Close To (the) Metal.
http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf,
letzter Besuch: 25. Juni 2007.

[8] RapidMind Inc. SH. http://libsh.org/, letzter Besuch: 20. Mai 2007.

[9] Intel. Open Source Computer Vision Library.
http://www.intel.com/technology/computing/opencv/index.htm, letzter Besuch: 20. Mai 2007.

[10] NVIDIA. CUDA. http://developer.nvidia.com/object/cuda.html, letzter Besuch: 20. Mai 2007.

[11] Reading University. PETS: Performance Evaluation of Tracking and Surveillance.
http://www.cvg.rdg.ac.uk/slides/pets.html, letzter Besuch: 20. Mai 2007.

[12] Stanford University. BrookGPU. http://graphics.stanford.edu/projects/brookgpu/,
letzter Besuch: 20. Mai 2007.

Literaturverzeichnis

[AYJC05] Kwang Ho An, Dong Hyun Yoo, Sung Uk Jung, and Myung Jin Chung. Real-Time Multi-View Face Tracking for Human-Robot Interaction. In *Development and Learning, 2005. Proceedings. The 4th International Conference*, pages 135–140, July 2005.

[BAS03] Ravi Budruk, Don Anderson, and Ed Solari. *PCI Express System Architecture*. Pearson Education, 2003.

[BFB04] Csaba Beleznai, Bernhard Frühstück, and Horst Bischof. Human Detection in Groups using a fast Mean Shift Procedure. In *International Conference on Image Processing*, pages 349–352, 2004.

[BFB05] Csaba Beleznai, Bernhard Frühstück, and Horst Bischof. Human Tracking By Mode Seeking. In *Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis*, pages 1–6, September 2005.

[Bod06] Arndt Bode. Multicore-Architekturen. *Informatik Spektrum*, 29(5): pages 349–352, 2006.

[Bra98] Gary R. Bradski. Real Time Face and Object Tracking as a Component of a Perceptual User Interface. In *WACV '98: Proceedings of the 4th IEEE Workshop on Applications of Computer Vision (WACV'98)*, page 214. IEEE Computer Society, Washington, DC, USA, 1998.

[BSF88] Yaakov Bar-Shalom and Thomas E. Fortman. *Tracking and data association*. Academic Press, Inc., Orlando, FL, USA, 1988.

[Buc05] Ian Buck. Taking The Plunge into GPU Computing. In Matt Pharr, editor, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 509–519. Addison-Wesley, March 2005.

[But97] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[Bux03] Hilary Buxton. Learning and understanding dynamic scene activity: a review. *Image and Vision Computing*, 21(1): pages 125–136, 2003.

[CBT01] Frédéric Cupillard, Francois Brémond, and Monique Thonnat. Tracking Groups of People for Video Surveillance. In *Proceedings of the 2nd European Workshop on Advanced Video-Based Surveillance System*, pages 88–100, University of Kingston (London), September 2001.

[CCL04] Fu Chang, Chun-Jen Chen, and Chi-Jen Lu. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision and Image Understanding*, 93(2): pages 206–220, 2004.

[CGP+01] Rita Cucchiara, Costantino Grana, Massimo Piccardi, Andrea Prati, and Stefano Sirotti. Improving Shadow Suppression in Moving Object Detection with HSV Color Information. In *Proceedings of the Fourth International IEEE Conference on Intelligent Transportation Systems*, pages 334–339, Oaklan, CA, USA, August 2001.

[CH05] Greg Coomba and Mark Harris. Global illumination using progressive refinement radiosity. In Matt Pharr, editor, *GPU Gems 2*, chapter 39, pages 635–647. Addison Wesley, March 2005.

[Che95] Yizong Cheng. Mean shift, mode seeking, and clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(8): pages 790–799, 1995.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2002.

[CM99] Dorin Comaniciu and Peter Meer. Mean shift analysis and applications. In *ICCV '99: Proceedings of the International Conference on Computer Vision-Volume 2*, page 1197. IEEE Computer Society, Washington, DC, USA, 1999.

[CRM03] Dorin Comaniciu, Visvanathan Ramesh, and Peter Meer. Kernel-based object tracking. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(5): pages 564–575, 2003.

[DHS00] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[Dij65] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communication of the ACM (Association for Computing Machinery)*, 8(9): page 569, 1965.

[EHD00] Ahmed M. Elgammal, David Harwood, and Larry S. Davis. Non-parametric Model for Background Subtraction. In *ECCV '00: Proceedings of the 6th European Conference on Computer Vision-Part II*, pages 751–767. Springer-Verlag, London, UK, 2000.

[FH75] Keinosuke Fukunaga and Larry D. Hostetler. The Estimation of the Gradient of a Density Function, with Applications in Pattern Recognition. *IEEE Transactions on Information Theory*, 21(1): pages 32–40, January 1975.

[FK03] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FM99] Alexandre R.J. Francois and Gerard G. Medioni. Adaptive Color Background Modeling for Real-Time Segmentation of Video Streams. In *Proceedings of the International Conference on Imaging Science, Systems, and Technology*, pages 227–232, Las Vegas, NA, 1999.

[Fun05] James Fung. Computer Vision on the GPU. In Matt Pharr, editor, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 40, pages 649–665. Addison Wesley, March 2005.

[GG05] Indra Geys and Luc Van Gool. Extended view interpolation by parallel use of the GPU and the CPU. In *Proceedings of IS&T/SPIE's (Society for Image Science and Technologie, Society of Photographic Instrumentation Engineers) 17th annual symposium on electronic imaging - videometrics VIII*, pages 96–107, January 2005.

[GLG05] Minglun Gong, Aaron Langille, and Mingwei Gong. Real-time image processing using graphics hardware: A performance study. In *International Conference on Image Analysis and Recognition*, pages 1217–1225, 2005.

[GLW+05] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 206. ACM Press, New York, NY, USA, 2005.

[GRNG05] Andreas Griesser, Stefaan De Roeck, Alexander Neubeck, and Luc Van Gool. GPU-Based Foreground-Background Segmentation using an Extended Colinearity Criterion. In *Vision, Modeling, and Visualization*, pages 319–326, 2005.

[Hac05] Toshiya Hachisuka. High-quality global illumination rendering using rasterization. In Matt Pharr, editor, *GPU Gems 2*, chapter 38, pages 615–633. Addison Wesley, March 2005.

[Har05] Mark Harris. Mapping Computational Concepts to GPUs. In Matt Pharr, editor, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 493–508. Addison-Wesley, March 2005.

[HBSL03] Mark Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101. Eurographics Association, 2003.

[HH03] C. Hughes and T. Hughes. *Parallel and Distributed Programming Using C++*. Addison-Wesley, Boston, MA, USA, 2003.

[HHD99] Thanarat Horprasert, David Harwood, and Larry S. Davis. A Statistical Approach for Real-time Robust Background Subtraction and Shadow Detection. In *IEEE International Conference on Computer Vision (ICCV'99) Frame-Rate Workshop*, pages 1–19, Corfu, Greece, September 1999.

[HHD00] Ismail Haritaoglu, Davis Harwood, and Larry S. David. W4: Real-Time Surveillance of People and Their Activities. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8): pages 809–830, 2000.

[HL04] Karl Hillesland and Anselmo Lastra. GPU floating-point paranoia. In *Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors, Los Angeles, USA*, pages C–8, 2004.

[Jar04] Frank Jargstorff. A Framework for Image Processing. In Randima Fernando, editor, *GPU Gems - Programming Techniques, Tips, and Tricks for Real-Time Graphics*, chapter 27, pages 445–467. Addison Wesley, 2004.

[JDWR00] Sumer Jabri, Zoran Duric, Harry Wechsler, and Azriel Rosenfeld. Detection and location of people in video images using adaptive fusion of color and edge information. In *Proceedings of the International Conference on Pattern Recognition*, volume 4, pages 627–630. IEEE Computer Society, Washington, DC, USA, September 2000.

[JH95] Neil Johnson and David Hogg. Learning the distribution of object trajectories for event recognition. In *BMVC '95: Proceedings of the 6th British conference on Machine vision (Vol. 2)*, pages 583–592. BMVA Press, Surrey, UK, UK, 1995.

[KEHKL+99] III Kenneth E. Hoff, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver. Fast computation of generalized Voronoi diagrams using graphics hardware. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 277–286. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.

[KW05] Jens Krüger and Rüdiger Westermann. A GPU Framework for Solving Systems of Linear Equations. In Matt Pharr, editor, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 703–718. Addison-Wesley, March 2005.

[LBC02] Na Li, Jiajun Bu, and Chun Chen. Real-Time Video Object Segmentation Using HSV Space. In *International Conference On Image Processing*, pages 85–88, 2002.

[LC04] Bent D. Larsen and Niels J. Christensen. Simulating photon mapping for real-time applications. In Alexander Keller and Henrik Wann Jensen, editors, *Rendering Techniques*, pages 123–132. Eurographics Association, 2004.

[LFWK05] Wei Li, Zhe Fan, Xiaoming Wei, and Arie Kaufman. Flow simulation with complex boundaries. In Matt Pharr, editor, *GPU Gems 2 - Programming*

Techniques for High-Performance Graphics and General-Purpose Computation, chapter 47, pages 747–764. Addison Wesley, March 2005.

[LHGT03] Liyuan Li, Weimin Huang, Irene Y. H. Gu, and Qi Tian. Foreground object detection from videos containing complex background. In *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, pages 2–10. ACM Press, New York, NY, USA, 2003.

[LKO05] Aaron Lefohn, Joe M. Kniss, and John D. Owens. Implementing Efficient Parallel Data Structures on GPUs. In Matt Pharr, editor, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 521–545. Addison-Wesley, March 2005.

[LWK03] Wei Li, Xiaoming Wei, and Arie E. Kaufman. Implementing lattice boltzmann computation on graphics hardware. *The Visual Computer*, 19(7-8): pages 444–456, 2003.

[MA03] Kenneth Moreland and Edward Angel. The FFT on a GPU. In *HWWS '03: Proceedings of the ACM Siggraph/Eurographics conference on Graphics hardware*, pages 112–119. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003.

[Mic05] Pauilus Micikevicius. GPU computing for protein structure prediction. In Matt Pharr, editor, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 43, pages 695–702. Addison Wesley, March 2005.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[OLG+05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Tim Purcell. A Survey of General-Purpose Computation on Graphics Hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, September 2005.

[PBMH05] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 268. ACM Press, New York, NY, USA, 2005.

[PDC+03] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[PGB07] Thomas Pock, Markus Grabner, and Horst Bischof. Real-time computation of variational methods on graphics hardware. In *Computer Vision Winter Workshop 2007*, pages 67–74. Institute for Computer Graphics and Vision, Graz University of Technology, Februar 2007.

[RH01] Christopher Rasmussen and Gregory D. Hager. Probabilistic data association methods for tracking complex visual objects. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(6): pages 560–576, 2001.

[RMK95] Christof Ridder, Olaf Munkelt, and Harald Kirchner. Adaptive Background Estimation and Foreground Detection using Kalman-Filtering. In *International Conference on Recent Advances in Mechatronics (ICRAM)*, pages 193–199, Istanbul, 1995.

[Ros04] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[RS01] Martin Rumpf and Robert Strzodka. Level Set Segmentation in Graphics Hardware. In *Proceedings of IEEE International Conference on Image Processing (ICIP'01)*, volume 3, pages 1103–1106, 2001.

[SD98] Tom Shanley and Dave Dzatko. *AGP System Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[Sfi89] Mukesh Sfinghal. Deadlock detection in distributed systems. *Computer*, 22(11): pages 37–48, November 1989.

[SFPG06] Sudipta N. Sinha, Jan-Michael Frahm, Marc Pollefeys, and Yakup Genc. GPU-based Video Feature Tracking and Matching. Technical report, University of North Carolina, Chapel Hill, May 2006.

[SG99] Chris Stauffer and W. Eric. L. Grimson. Adaptive background mixture models for real-time tracking. In *IEEE Conference on Computer Vision and Pattern Recognition*, volume 2, pages 246–252, 1999.

[Shr03] Dave Shreiner. *OpenGL Programming Guide: The Official Reference Document to OpenGL, Version 1.4*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[SL05] Thilaka Sumanaweera and Donald Liu. Medical image reconstruction with the FFT. In Matt Pharr, editor, *GPU Gems 2 - Programming Techniques for High-Performance Graphics and General-Purpose Computation*, chapter 48, pages 765–784. Addison Wesley, March 2005.

[Sta05] William Stallings. *Operating Systems (5th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[SWND05] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[TK91] Carlo Tomasi and Takeo Kanade. Detection and Tracking of Point Features. Technical Report CMU-CS-91-132, Carnegie Mellon University, April 1991.

[TKBM99] Kentaro Toyama, John Krumm, Barry Brumitt, and Brian Meyers. Wallflower: Principles and practice of background maintenance. In *IEEE Seventh International Conference on Computer Vision*, volume 1, pages 255–261, 1999.

[TS02] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[WBHK06] Horst Wildenauer, Philipp Blauensteiner, Allan Hanbury, and Martin Kampel. Motion Detection using an Improved Colour Model. In *Proceedings of the 2nd International Symposium on Visual Computing (ISVC06)*, pages 607–616. Springer, Lake Tahoe, NV, USA, November 2006.

[WC01] Zhengping Wu and Chun Chen. A New Foreground Extraction Scheme for Video Streams. In *MULTIMEDIA '01: Proceedings of the ninth ACM international conference on Multimedia*, pages 552–554. ACM Press, New York, NY, USA, 2001.

[WFSM02] Dongsheng Wang, Tao Feng, Heung-Yeung Shum, and Songde Ma. A Novel Probability Model for Background Maintenance and Subtraction. In *The 15th International Conference on Vision Interface*, page 109, 2002.

[WK04] Jan Woetzel and Reinhard Koch. Real-time multi-stereo depth estimation on GPU with approximative discontinuity handling. In *Visual Media Production, 2004. Conference on Visual Media Production 2004. First European Conference*, pages 245–254, 2004.

[YWC07] Juekuan Yang, Yujuan Wang, and Yunfei Chen. GPU accelerated molecular dynamics simulation of thermal conductivities. *Journal of Computational Physics*, 221(2): pages 799–804, 2007.

[ZQFK07] Ye Zhao, Feng Qiu, Zhe Fan, and Arie Kaufman. Flow simulation with locally-refined LBM. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 181–188. ACM Press, New York, NY, USA, 2007.