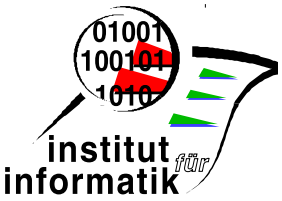
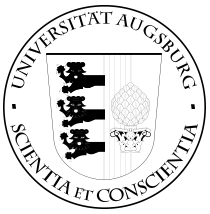
Institut für Informatik



Fachbereich für
Multimedia-Konzepte
und Anwendungen

UNIVERSITÄT AUGSBURG
D - 86135 Augsburg, Germany



D I P L O M A R B E I T

im Studiengang Angewandte Informatik

Analyse GPU-basierter Feature Tracking Methoden für den Einsatz in der Augmented Reality

Florian E. Mücke

Prüfer	:	Prof. Dr. E. André
Zweitprüfer	:	Prof. Dr. B. Möller
Betreuer	:	Dr. K. Dorfmüller-Ulhaas
Eingereicht am	:	20. Juni 2007

Florian Mücke
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.informatik.uni-augsburg.de
http://flo.mueckeimnetz.de

Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt zu haben. Alle Abbildungen und Grafiken wurden von mir eigenhändig erstellt oder in abweichenden Fällen deren Quelle angegeben. Die verwendete Literatur und sonstige Hilfsmittel sind vollständig angegeben.

(Florian Mücke)
Königsbrunn, 20. Juni 2007

Abstract

Feature tracking is one of the basic tasks of Computer Vision, which often provides the basic fundament for Augmented Reality applications. However, Augmented Reality applications require these tasks to be carried out in realtime. Recent computer systems fail to handle the enormous amount of image data to be processed though. By now the performance of modern graphics hardware is an order of magnitude higher than the processing power of CPUs. Therefore, and because of their programmable components, they have become interesting for other applications. This thesis proves that feature tracking methods can be efficiently implemented on modern GPUs (graphics processing units) and can be carried out in realtime. First of all common methods used in feature tracking are explained. Then the necessary background needed for the impementations on the GPUs is provided by the illustration of the graphics pipeline and the parallel programming model of current GPUs. In addition to this, the following GPU implementations which are of crucial importance to this topic are examined: the GPU based version of Sinha's KLT tracker and his SIFT (Scale Invariant Feature Transform) feature extractor [SFPG06], as well as Heymann's SIFT implementation [Hey05], the OpenVIDIA feature tracker [FM05] and an implementation of Harris' and Stephens' original corner detector [HS88]. A comparative evaluation of the implemented and original methods is made by specific benchchmarks.

Keywords: feature tracking, feature extraction, Augmented Reality, realtime, performance, GPU, GPGPU, implementation, SIFT, KLT, Difference of Gaussian, Harris corner detector, Canny edge detector, OpenVIDIA

Kurzfassung

Die Merkmalsverfolgung (engl. feature tracking) ist eine der grundlegenden Aufgaben des Maschinellen Sehens, die für Anwendungen der Erweiterten Realität (engl. Augmented Reality) oftmals den ersten Schritt darstellt. Für Anwendungen der Erweiterten Realität ist es jedoch erforderlich, diese Aufgaben möglichst in Echtzeit zu bewerkstelligen. Aufgrund der immensen Menge der zu verarbeitenden Bilddaten sind CPU-Systeme damit jedoch meist überfordert. Moderne Grafikkarten erreichen mittlerweile ein Vielfaches der Rechenleistung von CPUs und sind aufgrund dessen und wegen ihrer programmierbaren Komponenten auch für andere Anwendungen interessant geworden. Diese Arbeit zeigt, wie sich Verfahren zur Merkmalsverfolgung effizient auf modernen GPUs (Graphics Processing Units) implementieren lassen und dass sie in Echtzeit ausgeführt werden können. Dazu werden zunächst gängige Methoden zur Merkmalsverfolgung vorgestellt. Die nötigen Grundlagen für die GPU-Implementierungen liefern anschließend die Erklärung der Grafikpipeline und des parallelen Programmiermodells aktueller GPUs. Im Anschluss daran werden die folgenden GPU-Implementierungen zu diesem Thema untersucht: Sinhas GPU-basierter KLT-Trackers und sein SIFT (Scale Invariant Feature Transform)-Merkmalsextraktor [SFPG06], die SIFT Implementierung von Heymann [Hey05], der Feature-Tracker des OpenVIDIA Projektes [FM05] und eine GPU-Implementierung des ursprünglichen Eckendetektors von Harris und Stephens [HS88]. Eine vergleichende Bewertung der umgesetzten zu den ursprünglichen Verfahren wird durch Benchmarks ermöglicht.

Schlagworte: Merkmalsverfolgung, Merkmalsextraktion, Erweiterte Realität, Echtzeit, Leistung, GPU, GPGPU, GPU-Implementierung, SIFT, KLT, Difference of Gaussian, Harris-Eckendetektor, Canny-Kantendetektor, OpenVIDIA

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

Die Merkmalsverfolgung, das *Feature-Tracking*, ist eine der grundlegenden Aufgaben des Maschinellen Sehens, die für Anwendungen der Erweiterten Realität (engl. Augmented Reality) oftmals den ersten Schritt darstellt. Gerade bei Aufgaben der Erweiterten Realität, in der Objekte im Kontext des Benutzers hinzugefügt oder auch ersetzt werden, müssen Algorithmen für die Objekterkennung und -verfolgung, in Echtzeit ablaufen. Aufgrund der immensen Menge der zu verarbeitenden Bilddaten sind echtzeitfähige Systeme jedoch schwer zu realisieren. Normale CPUs (*central processing units*), wie sie in gängigen Computersystemen zu finden sind, sind damit meist überfordert. Darüber hinaus hat die wachsende Auflösung von Videokameras und der Wunsch nach ständig höherer Genauigkeit die Anforderungen an echtzeittaugliche Systeme weiter erhöht. Diese müssen fähig sein, Unmengen an Bilddaten schnell zu verarbeiten.

Da bei der Merkmalsverfolgung die Informationen der einzelnen Merkmale unabhängig voneinander berechnet werden, lässt sich dieser Vorgang gut in parallele Teilschritte zerlegen. Spezialisierte Hardware, die die parallele Verarbeitung der Daten übernimmt, ist jedoch teuer. Moderne GPUs (*graphics processing units*) erreichen mit ihren programmierbaren parallelen Verarbeitungseinheiten mittlerweile ein Vielfaches der Rechenleistung von CPUs. Sie stellen daher eine ernstzunehmende Alternative zu teurer Spezialhardware dar.

Auch bieten GPUs aufgrund der Ähnlichkeit beider Kernthematiken eine gute Ausgangslage für die Verarbeitung von visuellen Algorithmen: Während sich die Computergrafik mit dem Erzeugen von Bildern aus mathematisch beschriebenen Szenen befasst, geht es beim Maschinellen Sehen prinzipiell darum, aus Bildern eine mathematische Beschreibung der Szene zu bestimmen. Es liegt also nahe, die GPU, auch für diese Zwecke zu nutzen.

Aufgrund ihrer Leistungsfähigkeit sind sie inzwischen auch für andere Anwendungen interessant geworden: darunter unter anderem für Berechnungen der Linearen Algebra [KW03], Bildverarbeitung und -analyse [CBdR03], physikalische Simulationen [LFWK05] und Beschleunigungen von Sortieralgorithmen [GRHM05, GZ06]. Rund um das Thema GPGPU (*General-Purpose computation on GPUs*), dem Berechnen von allgemeinen Aufgaben auf dem Grafikprozessor, hat sich eine rege Entwicklergemeinschaft aus verschiedenen Wissenschaftlern und Forschern gebildet [GPGa]. Eine

ausführliche Übersicht der Entwicklungen in diesem Bereich ist in [OLG+07] zu finden.

Des Weiteren gibt es einige Veröffentlichungen, die sich mit der Beschleunigung von Verfahren des Maschinellen Sehens auf der GPU befassen: Eine echtzeitfähige Umsetzung der Bildregistrierung mittels Gradientenfluss ist beispielsweise in [SDR03] von Strzodka et al. beschrieben. In [SDR03] behandeln Strzodka und Garbe ein GPU-basiertes System, das eine Bewegungsabschätzung in Videosequenzen mittels Optischem Fluss ermöglicht. Klein und Murray stellen in [KM06] einen 3D-Kanten-Tracker vor, der auf GPU-Partikelfiltern basiert. Rao [RH05] implementierte ein System, mit dem es möglich ist, menschliche Gliedmaßen ohne Einsatz von weiteren Markern zu verfolgen. Heymann [Hey05] setzte den SIFT-Algorithmus, ein Verfahren zur Bestimmung von Merkmalspunkten, die invariant gegenüber Größenänderungen sind, auf der GPU um. Sinha beschrieb in [SFPG06] eine GPU-Implementierung des Trackingverfahrens von Kanade, Lucas und Tomasi (KLT) sowie eine weitere Umsetzung von SIFT auf der GPU. In [FM04,FM05] stellen Fung und Mann eine Software-Bibliothek (*OpenVIDIA*) vor, in der verschiedene Algorithmen des Maschinellen Sehens umgesetzt sind. Bei deren Entwicklung stand allerdings die gleichzeitige Ausführung auf mehreren GPUs im Vordergrund. Aber auch auf anderer Hardware, wie etwa FPGAs (*Field Programmable Gate Arrays*), wurden Verfahren zur Merkmalsverfolgung implementiert [NH05].

Da sich die vorliegende Arbeit auf die Verfolgung von natürlichen Merkmalen auf der GPU beschränken möchte, werden die Umsetzungen von Heymann und Sinha und der Feature-Tracker aus dem OpenVIDIA Projekt untersucht. Die erst kürzlich veröffentlichte Beschreibung von Ready und Taylor [RT07] konnte in dieser Arbeit nicht mehr berücksichtigt werden.

1.1 Zielsetzung

Diese Diplomarbeit möchte zeigen, dass sich moderne Grafikkarten als effiziente Plattform zur Echtzeit-Verarbeitung von Bilddaten einsetzen lassen. Daher wird beschrieben, wie bekannte Verfahren auf der GPU implementiert werden können. Des Weiteren werden existierende GPU-Umsetzungen untersucht und deren Leistungsfähigkeit im Vergleich zu CPU-Implementierungen beurteilt. Hierfür müssen die verwendeten Algorithmen sowie die nötigen Grundlagen der GPU-Programmierung erläutert werden. Ziel dieser Arbeit ist es auch, die technischen Möglichkeiten zu ergründen, auf Einschränkungen hinzuweisen, Probleme aufzuzeigen und die gesammelten Erfahrungen weiterzugeben. Die quantitative

1.2 Übersicht

Kapitel 2 beschreibt, warum sich moderne Grafikkarten als leistungsstarke Koprozessoren eignen, Kapitel 3 definiert zunächst den Begriff der Augmented Reality und gibt einen Einblick in gängige Techniken des Feature-Trackings. In Kapitel 4 wird die Grafikpipeline aktueller GPUs und das GPU-Programmiermodell erläutert, während in Kapitel 5 die einzelnen Implementierungen untersucht und bewertet werden.

2 Die GPU – der heimische Supercomputer

2.1 Leistungsstark und erschwinglich

Die Rechenleistung von Grafikprozessoren hat, im Vergleich zu Desktopprozessoren, in den letzten Jahren überdurchschnittlich zugenommen. Laut Owens et al. [OLG+07] war dies pro Jahr ein Faktor von 1,7 bis 2,3. Diese Wachstumsrate übersteigt deutlich das oft zitierte Mooresche Gesetz, das bei üblichen Mikroprozessoren Anwendung findet; danach beträgt sie 1,4. Dieser Unterschied liegt unter anderem an der Multimilliarden-Dollar-Spieleindustrie, die die Entwicklung im Grafikbereich kontinuierlich vorantreibt. Abbildung 2.1 zeigt die Leistungsentwicklung von GPUs und CPUs in den letzten fünf Jahren[1].

Aktuelle Grafikarchitekturen bieten eine enorme Speicherbandbreite und Rechenleistung: eine Radeon HD 2900 XT schafft nahezu 475 GFLOPs, eine GeForce 8800 Ultra einen theoretischen Spitzenwert von 581 GFLOPs. Dies hätte 1993 für einen Platz in der TOP500[2] der schnellsten Computer gereicht. Ein aktueller Core2Duo von Intel schafft nur um die 50 GFLOP pro Sekunde. Zum Vergleich: die *Cell Broadband Engine* (Cell BE) von Sony, Toshiba und IBM, wie sie unter anderem in Sonys Playstation 3 verbaut ist, erreicht mit ihren neun Kernen (8 Synergistic Processing Elements + 1 Power Processing Element) einen theoretischen Spitzenwert für einfache Genauigkeit von 256 GFLOP pro Sekunde [Ste06]. Aktuelle Grafikkarten haben eine Speicherbandbreite von etwa 100 GB/s, während ein aktueller Core2Duo etwa 8 GB/s erreicht.

Die Leistungsfähigkeit der Halbleiter in den Prozessoren ist für beide Sparten gleich, da sie denselben Nutzen aus den Fortschritten im Fertigungsprozess ziehen können. CPUs sind jedoch für hohe Leistung auf sequentiellem Code optimiert, wohingegen GPUs für die Verarbeitung von parallelen Daten entwickelt wurden. Bei CPUs geht daher ein großer Teil ihrer Transistorfläche für Caches und zum Gewinnen von Parallelität auf Ausführungsebene durch Sprungvorhersage und *out-of-order*-Ausführung

[1] Die GFLOP-Werte der GPUs wurden aus der Anzahl der pro Takt und Pixelprozessor ausführbaren parallelen Rechenoperationen eigenhändig berechnet und mit bekannten Werten abgeglichen [Ste06, ath06, And07, Ber07, OLG+07].

[2] http://www.top500.org

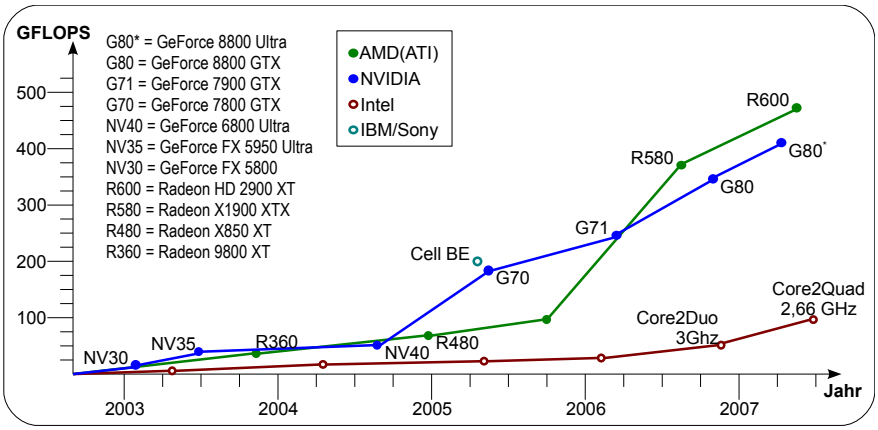


Abb. 2.1: Gleitkommaoperationen pro Sekunde
Die Anzahl Gleitkommaoperationen pro Sekunde (gemessen an den Multiplizier-Addier-Einheiten, wobei ein MADD als zwei Gleitkommaoperationen zählt) hat bei den GPUs, im Gegensatz zu den CPUs, in den letzten Jahren drastisch zugenommen.

von Befehlen verloren. GPUs erreichen dagegen durch ihre hohe Datenparallelität eine deutlich höhere arithmetische Dichte bei gleicher Transistorzahl, da ihre Transistoren direkt zur Berechnung verwendet werden. Wenn man bedenkt, dass aktuelle GPU-Spitzenmodelle aus ungefähr 700 Millionen und Core2Duo Prozessoren aus knapp 300 Millionen Transistoren bestehen, wird die immense Leistung, die in den GPUs steckt schnell offensichtlich.

Aktuelle GPUs, wie sie heute in Heimrechnern zu finden sind, bieten eine Rechenleistung, für die man vor ein paar Jahren noch ein Rechenzentrum aufsuchen musste. Eine topaktuelle Grafikkarte, wie die Radeon HD 2900 XT von AMD, kostet knapp 380 Euro, ein Core2Duo E6700 mit 2,66 GHz von Intel etwa 290 Euro. Dem nicht einmal 100 Euro großen Preisunterschied steht allerdings etwa die zehnfache Leistung auf der GPU Seite gegenüber. Implementierungen von Sinha [SFPG06] und Heymann [Hey05] sowie eigene Berechnungen bestätigen, dass selbst mit älteren Modellen eine Leistungssteigerung um den Faktor 6–12 in Anwendungen durchaus realistisch ist.

2.2 Flexibel und programmierbar

Nicht nur die rohe Leistung, sondern auch die Flexibilität und einfache Programmierbarkeit heutiger Grafikhardware hat dazu beigetragen, dass ein immer breiter werdendes Interesse besteht, sie für andere Anwendungen zu nutzen. Während frühe Grafikarchitekturen lediglich eine feste Ausführungspipeline und maximal 8 Bit Farbinformation pro Kanal boten, halten heutige GPUs voll programmierbare Prozessoreinheiten bereit und unterstützen vektorbasierte Gleitkommaoperationen mit Werten in einfa-

cher Genauigkeit nach IEEE 754 (siehe Unterabschnitt 4.1.4). Vor ein paar Jahren mussten diese Einheiten noch mittels Maschinensprache programmiert werden. Heute gibt es hierfür richtige Hochsprachen (siehe Abschnitt 4.2). Zudem kommen mit jeder neuen GPU-Generation neue Funktionen hinzu, die die Programmierbarkeit weiter verbessern (siehe Abschnitt 4.2).

Für die Attraktivität der GPU als Programmierplattform spielt also nicht nur die rohe Leistung, sondern auch die steigende Genauigkeit und die schnell wachsende Programmierfähigkeit eine entscheidende Rolle.

2.3 Einschränkungen und Schwierigkeiten

Auch die GPU ist laut Owens et al. [OLG+07] kaum ein Allheilmittel für alle möglichen Rechenaufgaben, denn ihre Leistung resultiert aus einer hochspezialisierten Architektur, die über Jahre hinweg entwickelt und abgestimmt wurde, um maximale Leistung in den hochparallelen Arbeitsschritten der Computergrafik zu erzielen. Erst die gesteigerte Flexibilität der GPUs und ihre raffinierte Ausnutzung durch GPGPU-Entwickler ermöglichte den Einsatz in anderen Gebieten, die ursprünglich nicht für die GPU gedacht waren [OLG+07]. Trotzdem gibt es noch genügend Anwendungen, die sich nicht für die GPU eignen und wohl auch nicht eignen werden, wie beispielsweise die zeigerlastige Textverabeitung: hier dominiert die Speicherkommunikation, weswegen die Anwendung schwer zu parallelisieren ist. Auch verschiedene rechenintensive Aufgaben, wie die Kryptographie, erschließen sich aufgrund fehlender Ganzzahlberechnung und ihren verknüpften logischen Operationen (bitweises und logisches `AND`, `OR`, ..., Shifts) momentan noch nicht für die GPU. Daneben mangelt es heutigen GPUs außerdem an einigen grundlegenden programmiertechnischen Konstrukten, wie zum Beispiel einer effizienten *scatter*-Speicheroperation (indexbasierte Schreiboperation in Feldern, siehe Abbildung 4.1.1). Die Steigerung der Genauigkeit auf 32 Bit hat viele Anwendungen im Bereich der *General Purpose Computation on GPUs* ermöglicht, auf die in [OLG+07] näher eingegangen wird. Das Fehlen von doppelter Genauigkeit (64 Bit) verschließt allerdings immer noch die Pforten für viele größere Berechnungsprobleme in der Wissenschaft.

Durch das ungewöhnliche Programmiermodell der GPUs bleibt es schwierig, nichtgrafische Aufgaben auf die Grafikhardware abzubilden. Stattdessen ist ein Ummünzen der Berechnungen in grafische Ausdrücke nötig, für das sehr fundiertes Wissen über sowohl die Hardware als auch über die umzusetzenden Algorithmen erforderlich ist. Allerdings sind, wie auch Owens et al. in [OLG+07] schreiben, trotz der Herausforderungen bei der Programmierung die möglichen Vorteile schwer zu ignorieren. Diese sind eine enorme Rechenleistung mit einer steileren Wachstumskurve als bei traditionellen

CPUs.

GPUs können einige Probleme viel schneller als CPUs lösen, da sie hochgradig spezialisiert sind, etliche Operationen parallel ausführen können und einen äußerst schnellen parallelen Speicherzugriff haben. Die Fähigkeiten und die damit verbundenen Möglichkeiten haben aber ihren Preis: GPUs tauschen Flexibilität gegen Geschwindigkeit. Nach Owens et al. gibt es viele Probleme, die sie derzeit nicht angehen können und es ist unbedingt notwendig, die eigenen Anwendungen nach dem Wissen um ihre Stärken und Schwächen zu planen [OLG+07].

3 Feature Tracking in der Augmented Reality

Dieses Kapitel gibt einen Einblick in gängige Techniken des *Feature-Trackings* und stellt den Zusammenhang zur *Erweiterten Realität* her.

Dazu wird zunächst der Begriff der *Erweiterten Realität* erklärt und Beispielanwendungen genannt. Im darauffolgenden Abschnitt wird der Vorgang des *Feature-Trackings* erläutert und anschließend Techniken zum Finden und Verfolgen von Merkmalen in Bildern vorgestellt.

3.1 Die Erweiterte Realität

Unter *Erweiterter Realität (augmented reality, AR)* versteht man die computergestützte Erweiterung der Realitätswahrnehmung. Diese Informationen können generell alle menschlichen Sinne ansprechen, häufig ist mit Erweiterter Realität jedoch nur die visuelle Darstellung von Informationen gemeint.

Milgram u.a. beschreiben *Erweiterte Realität* und *Erweiterte Virtualität (augmented virtuality, AV)* im „Realitäts-Virtualitäts-Kontinuum“ als Teile der *Gemischten Realität (mixed reality, MR)* [MTUK94].

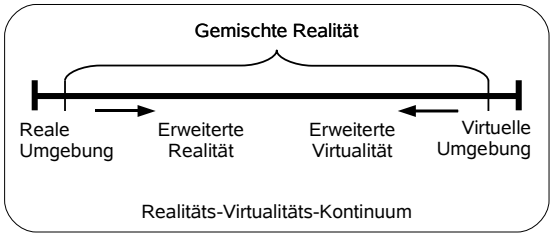


Abb. 3.1: Vereinfachte Darstellung des RV-Kontinuums nach Milgram [MTUK94]

Im Gegensatz zur Erweiterten Virtualität, bei der eine vollständig virtuelle Umgebung durch reale Informationen ergänzt wird, wird in der Erweiterten Realität die Wahrnehmung des realen Umfeldes durch virtuelle Informationen erweitert. Azuma stellt in seiner Definition [Azu95] folgende Anforderungen an ein AR-System:

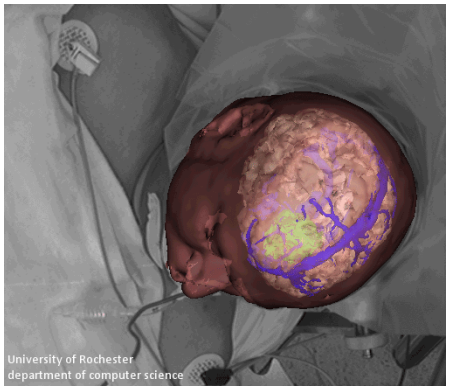
1. Verbindung zwischen Realität und virtueller Welt
2. Interaktives System mit Ablauf in Echtzeit
3. Dreidimensionaler Bezug zwischen realen und virtuellen Objekten

Erweiterte Realität ist prinzipiell nicht auf die Nutzung von visuellen Ausgabemedien wie etwa *head mounted displays (HMD)* beschränkt, sondern kann verschiedenste Bereiche der menschlichen Sinneswahrnehmung ansprechen.

Funktionsweise Um Informationen über das Umfeld zu erfassen, gibt es viele Möglichkeiten: optisch, akustisch, mittels GPS, über Radar, etc. In den meisten Fällen wird jedoch eine Kombination aus Kamera und einem optischen Ausgabemedium, wie Bildschirmen (Handydisplays) und VR- oder halbdurchsichtigen Brillen, gewählt, da diese Lösungen kostengünstig, mobil einsetzbar und nicht zu komplex sind. Azuma nennt aber auch ausgefallenere Techniken, wie beispielsweise eine direkte Projektion des erzeugten Bildes auf der Netzhaut des Auges (Retinaprojektion) [ABB+01]. Bei der Ausgabe wird dann das Eingangsbild mit der generierten Szene überlagert. Diese wird nach der Auswertung der Bild- und Videoinformationen des Umfeldes erzeugt. Um die Anforderung nach der Echtzeitfähigkeit erfüllen zu können, ist eine schnelle Analyse und anschließende Visualisierung der Informationen notwendig. Schon ein kleiner zeitlicher Versatz (Latenz) zwischen Ausgangs- und Endbild kann die Informationen unpassend werden lassen sowie die Bedienbarkeit des Systems beeinträchtigen.

Anwendungen Der Einsatz von *Augmented Reality Anwendungen* ist in vielen Bereichen des Alltags denkbar. Schenkt man den Visionen von Mark Weiser [Wei91] über die Rechnerallgegenwart, dem *ubiquitous computing*, Beachtung, so werden Computer in Zukunft untrennbar und transparent mit unserem Alltag verbunden sein. AR-Systeme werden sicherlich auch ihren Teil dazu beitragen und uns allgegenwärtig mit zusätzlichen Informationen versorgen, egal ob bei der Arbeit, einem Spaziergang oder zuhause.

Beispiele hierfür wären die Darstellung von Planungsergebnissen in realen Fertigungsanlagen zur Überprüfung in der Automobilindustrie, Ergänzung des Kamerabildes um virtuelle Objekte in der Fernsehindustrie, Einblendung von nicht sichtbaren Organen als Hilfestellung bei Operationen oder zur Ausbildung, Anzeige von Gelände- oder Freund/Feind-Informationen für Militär oder Katastrophenschutz, Einblenden von Zusatzinformationen für Techniker bei der Wartung von Geräten, Head-Up-Displays in Flug- oder Fahrzeugen, virtuelle Objekte in Museen oder um virtuelle Komponenten erweiterte Spieleanwendungen.

(a) 

(b) 

Abb. 3.2: Beispiele für AR-Anwendungen
(a) Medizinische AR-Anwendung (Universität Rochester), (b) Roivis(R) Fabrikplanungs- und Produktvisualisierungsanwendung (metaio GmbH), benutzt mit Erlaubnis

Die meisten Anwendungen der Erweiterten Realität werden an diversen Universitäten und Einrichtungen erforscht und sind nach wie vor in einem relativ frühen Entwicklungsstadium. Dies liegt einerseits an den Voraussetzungen, die ein solches System erfüllen soll (unauffällig, am besten tragbar und trotzdem leistungsstark und echtzeitfähig). Andererseits sind sowohl die recht hohen Anforderungen an die Hardware (Prozessoren, Energieversorgung, etc.), als auch an die Software der Informationsverarbeitung und -präsentation eine nicht unerhebliche Hürde. Das wohl größte Problem für Systeme der Erweiterten Realität, ist eine korrekte Beziehung zum jeweiligen Umfeld herzustellen und sich darauf abzustimmen. Dazu gehört sowohl eine exakte kontextbezogene Positionsbestimmung, als auch die Registrierung von umliegenden Objekten und Interessenquellen. Um eine virtuelle Szene möglichst überzeugend in eine reale einbetten zu können, sind Daten notwendig, welche die Geometrie der Umgebung beschreiben. Diese Daten sind aber nicht immer verfügbar und lassen sich aufgrund ihrer Komplexität kaum in Echtzeit erzeugen. Die Anforderungen an ein AR-System sind aufgrund der erforderten höheren Genauigkeit laut Azuma wesentlich höher als für Systeme der virtuellen Realität [Azu95].

3.2 Feature Tracking

Unter *Tracking* wird generell die Lokalisierung sowie Verfolgung der Pose eines beliebigen Objekts verstanden. Dabei wird grundsätzlich zwischen sensorbasiertem und bildbasiertem Tracking unterschieden.

Beim sensorbasierten Tracking werden ein oder mehrere Sensoren, das können magnetische, funkbasierte, akustische oder inertiale Sensoren sein, verwendet, um die Pose

Markerbasierte Posenbestimmung

1. Kanten finden
   - Bild binarisieren, Schwellwert anwenden und schwarz/weiß-Kante ablaufen
   - oder Gradientenmaxima ablaufen
2. Konturextraktion: Bestimmung der Projektionen von Quadraten
3. Bestimmen der vier Eckpunkte dieser Quadrate
4. Zuordnung bekannter Marker zu den im Bild gefundenen Quadraten
5. Berechnung der Homographie
6. Erstellen einer Liste aus 2D/3D Korrespondenzen der korrekt erkannten Marker
7. Berechnung der Kamera-Pose aus den 2D/3D Korrespondenzen

Abb. 3.3: Ablauf einer markerbasierten Posenbestimmung

eines Objekts zu bestimmen oder zu verfolgen.

Das für diese Arbeit relevante bildbasierte Tracking nutzt Kameras, um Informationen über die Umgebung zu erhalten. Man unterscheidet hierbei zwischen markerbasiertem und markerlosem Tracking.

Beim markerbasierten Tracking werden künstliche Markierungen, die Marker, in einer Umgebung platziert und mit deren Hilfe schließlich die Pose eines Objekts berechnet. Eine verbreitete quelloffene Softwarebibliothek für markerbasiertes Tracking ist das *ARToolKit*[1]. Abbildung 3.3 zeigt den Ablauf eines markerbasierten Systems zur Posenbestimmung.

Im Gegensatz zu den markerbasierten Trackingsystemen, in denen die Kamerapose aus den 2D/3D-Korrespondenzen berechnet wird, die durch Auswertung der Markercodes bestimmt wurden, muss diese beim markerlosen Tracking aufgrund von natürlichen Merkmalen bestimmt werden. Hier können bekannte Informationen über die Umgebung, wie etwa eine geschätzte Pose oder Informationen über die Beschaffenheit der natürlichen Marker, das Tracking-Verfahren deutlich vereinfachen.

Bei den markerlosen Verfahren unterscheidet man des Weiteren zwischen *modellbasierten* und *merkmalsbasierten* Verfahren. Die modellbasierten Verfahren projizieren bekannte 3D Informationen, beispielsweise Linien aus einem CAD-Modell, ins Videobild und verfolgen diese projizierten Informationen.

Diese Arbeit befasst sich vorrangig mit merkmalsbasierten Verfahren, in denen geometrische Primitive wie Linien, Punkte oder Konturen im Bild erkannt werden und versucht wird deren Bewegung im Bildstrom zu verfolgen. Man bezeichnet solche Verfahren auch als „on-line“-Tracker, da sie lediglich die Informationen des aktuellen Bildstroms zur Erkennung und Verfolgung der Merkmale verwenden.

Wie im vorhergehenden Abschnitt erwähnt, ist es für Systeme der Erweiterten Reali-

[1] ARToolKit, http://www.hitl.washington.edu/artoolkit/

tät essentiell, ihre Umgebung zu erfassen, um einen korrekten Bezug zu ihr herstellen zu können. Dafür wird in den meisten Fällen die Umgebung mittels Kameras als Strom von Videobildern erfasst. Im Folgenden werden die Grundlagen und Ansätze für Techniken zur Auswertung der Umgebungsinformationen dieser videobasierten Systeme erläutert, bevor diese im darauffolgenden Kapitel genauer beschrieben werden.

Erkennung von Merkmalen

Die Erkennung von Merkmalen und Merkmalspunkten, die *feature detection*, ist häufig der Ausgangspunkt für weitere Algorithmen, wie beispielsweise Posenbestimmung oder Objekterkennung. Das Ergebnis der auf die Merkmalserkennung folgenden Operationen hängt also direkt mit der Güte der gewählten Merkmale zusammen. Sind diese schlecht gewählt, weil sie einen geringen Wiedererkennungswert haben, so werden mit den darauf folgenden Algorithmen auch keine optimalen Ergebnisse erzielbar sein. Daher wurden verschiedene Verfahren entwickelt, die zum Teil auf unterschiedliche Merkmale setzen. Man unterscheidet prinzipiell vier Typen primärer Merkmale, von denen jedoch nur die ersten drei für diese Arbeit relevant sind:

- **Kanten:** Kanten sind Punkte, die zwei Bereiche eines Bildes trennen. Sie haben betragsmäßig große Gradienten und sind lokal gesehen von eindimensionaler Struktur, was einer starken Änderung der Intensitätswerte in einer Richtung entspricht. Beispiele für Kantendetektoren sind der Prewitt- und Sobeloperator (siehe Unterabschnitt 3.3.1) oder der Canny-Kantendetektor (siehe Unterabschnitt 3.3.3).

- **Ecken und Interessenspunkte:** Ecken sind Punkte zweidimensionaler Struktur, deren Gradienten einen hohen Grad an Eckigkeit aufweisen, was einer plötzlichen Kontraständerung in beiden Richtungen entspricht. Diese Punkte müssen nicht zwangsläufig echte Ecken sein. Das populärste Beispiel für einen Eckendetektor ist das von Harris und Stephens [HS88] entwickelte Verfahren, das in Unterabschnitt 3.3.4 behandelt wird.

- **Blobs:** Unter *Blobs* versteht man Bereiche und Interessenspunkte, die auch in Bereichen erkannt werden können, die zu glatt für Eckendetektoren sind. Meistens wird dazu ein besonders bevorzugter Punkt herangezogen, beispielsweise ein lokales Maximum. Als Beispiele seien an dieser Stelle der *Laplacian of Gaussian* (LoG, siehe Unterabschnitt 3.3.2) und der *determinant of the Hessian* (DoH) Operator genannt.

- **Ridges:** Unter *ridge* versteht man eine Gruppe aus Punkten, die in $n-1$ ihrer n Dimensionen lokale Maxima sind. Bei lokalen Minima spricht man von Tä-

lern (*valleys*). Solche Punkte findet man beispielsweise in den Innenbereichen langgezogener Objekte wie etwa den Fingern einer Hand.

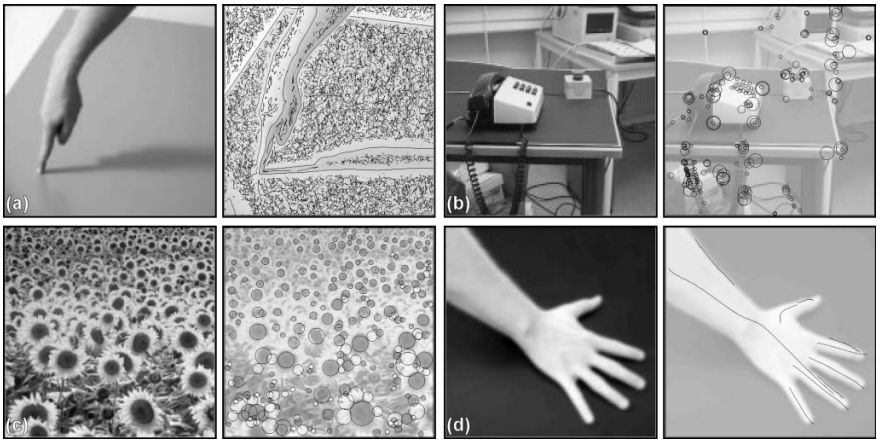


Abb. 3.4: Merkmalstypen: Kanten, Ecken, Blobs und Ridges; entnommen aus [Lin96]

Es ist aber auch möglich eine Entscheidung über die Besonderheit eines Bildteils aufgrund anderer Merkmale wie etwa einer bestimmten Farbgebung, wie bei der Erkennung von menschlichen Gliedmaßen oder der Veränderung der Bildinhalte, die durch die Bewegung von Objekten entsteht, zu treffen.

Merkmalsextraktion

Unter dem Herauslösen von Merkmalen, der Merkmalsextraktion (*feature extraction*), versteht man die Gewinnung von Merkmalsbeschreibungen, die eine möglichst eindeutige Identifizierung der einzelnen Merkmale ermöglichen. Dabei können Invarianzeigenschaften dieser Beschreibungen, wie etwa Invarianz gegenüber Rotation, Streckung und Änderung in der Bildhelligkeit, die Chance einer korrekten Zuordnung erheblich verbessern. Um Invarianzen zu erreichen, wird oft eine Kombination aus Gradientenvektor und Abschätzungen der Krümmung (zweite Ableitung der Bildfunktion) des jeweiligen Interessenspunkts in einer lokalen Umgebung um diesen betrachtet. Auch durch einen Vergleich mit den Nachbarpunkten in einem Skalenraum können Invarianzeigenschaften erzielt werden. Dies wird beispielsweise bei der *Scale Invariant Feature Transform* (SIFT) in Unterabschnitt 3.3.6 genutzt, um Invarianz gegenüber Größenänderungen zu erreichen.

Der Übergang zwischen Erkennen der Merkmale und ihrem Herauslösen ist fließend, eine klare Trennung gibt es in der Praxis meist nicht. Man kann die Merkmalsextraktion daher auch als zweite Stufe des Erkennungsprozesses betrachten.

Verfolgung von Merkmalen (Tracking)

Wie eingangs erwähnt, bezeichnet *Tracking* das Wiederfinden markanter Merkmale über eine Folge von Bildern (Video). Durch eindeutige Zuordnung der Merkmale kann man Rückschlüsse über die Bewegung der Kamera und der aufgenommenen Objekte ziehen. Diese Informationen lassen sich zudem als weitere Hilfsmittel zur Erfassung der räumlichen Strukturen der Umgebung nutzen. Es gibt dabei verschiedene Ansätze, dies zu lösen. Zum einen besteht die Möglichkeit, die besonderen Merkmale nur aus dem ersten Bild zu extrahieren und diese dann in den Folgebildern lediglich in einer lokalen Umgebung zu suchen. Ein Verfechter dieses dynamischen Schemas ist der KLT-Tracker (siehe Unterabschnitt 3.3.5). Ein anderer Ansatz gewinnt erst die Merkmale aus jedem Bild und findet dann die korrespondierenden in der Bilderserie. Ein Nachteil dieser Technik ist, dass Fehler bei den Übereinstimmungen dazu neigen, sehr groß zu werden. Um nicht mit einer falschen Zuordnung hängen zu bleiben, ist es notwendig, die Referenzmerkmale in bestimmten Abständen zu erneuern.

Beim dynamischen Ansatz kann im Gegensatz zum statischen das zeitliche Modell als Entscheidungshilfe genutzt werden, um zu bestimmen, wo und wie im Folgebild nach Merkmalen gesucht werden soll. Durch eine relative Bestimmung der Folgemerkmale kann sich jedoch über die gesamte Sequenz ein Versatz *(drift)* ergeben. Ein Problem, dem sich Shi und Tomasi in [ST94] angenommen haben, ist eine plötzliche gravierende Änderung im Folgebild, wie beispielsweise eine Rotation oder eine Verschiebung. Diese können aufgrund eines beschränkten Zielfensters, in dem nach passenden Merkmalen gesucht wird, nicht erfasst werden.

Zur Bestimmung der Korrespondenzen können beispielsweise iterative Verfahren wie *RANSAC* (siehe Unterabschnitt 3.3.7) zum Einsatz kommen.

3.3 Methoden und Techniken zur Verfolgung von Merkmalen

In diesem Abschnitt werden stellvertretend einige gängige Verfahren betrachtet, die im Bereich der Merkmalsverfolgung häufig Anwendung finden. GPU-Umsetzungen dieser Verfahren werden später in Kapitel 5 genauer betrachtet. Als Erstes wird der Sobelfilter vorgestellt, der exemplarisch für die Gruppe der gradientenbasierten Kantendetektoren steht. Diese sind Grundlage vieler weiterer Techniken, wie auch für den Eckendetektor von Harris und Stephens in Unterabschnitt 3.3.4. Dieser wiederum wird verwendet, um markante Stellen eines Bildes, die Interessenspunkte, herauszufiltern. Solche Punkte können als Referenzwerte für weitere Verfahren, wie etwa dem KLT-Tracker (behandelt in Unterabschnitt 3.3.5) dienen.

Die skalierungsinvariante Merkmalstransformation (SIFT) in Unterabschnitt 3.3.6

macht sich mehrere Verfahren, wie beispielsweise die Laplace- bzw. DoG-Filter (Unterabschnitt 3.3.2), zu eigen, um damit bestimmte Invarianzeigenschaften für die Merkmale zu erreichen. Dort wird außerdem ein Deskriptor für die Merkmale definiert, der ermöglicht, diese zweifelsfrei wiederzuerkennen. Dies wird im Gegensatz zu markerbasierten Verfahren, mit denen in erster Linie die Kamerapose bestimmt wird, zur Erkennung von beliebigen Objekten, wie etwa Gesichtern, Gegenständen oder Gebäuden genutzt.

3.3.1 Der Sobel-Kantendetektor

Der Sobel-Operator führt auf einem Bild eine lokale Messung der Gradienten in x und y Richtung aus, welche diejenigen Bereiche besonders hervorhebt, die eine hohe Kontraständerung haben, was einer Kante im Bild entspricht. Dies wird oft benutzt, um den größten Gradienten für jeden Punkt eines Graustufenbildes zu finden.

Die Idee hinter gradientenbasierten Verfahren, wie etwa Ecken- und Kantendetektoren, ist die Annahme, dass das Bild als kontinuierliche 2D-Funktion vorlag und in den Bildpunkten diskretisiert wurde. Diese kontinuierliche zweidimensionale Funktion lässt sich durch Auswertung der Intensitätswerte der Bildpunkte näherungsweise bestimmen und ihre Ableitungen untersuchen.

Der Sobelfilter besteht aus zwei um 90° versetzten 3×3 Faltungsmasken:

-1	0	$+1$
-2	0	$+2$
-1	0	$+1$

G_x

$+1$	$+2$	$+1$
0	0	0
-1	-2	-1

G_y

Dabei ist G_x der vertikale Filter (horizontale Gradienten) und G_y der horizontale Filter (vertikale Gradienten). Diese Masken reagieren maximal auf Kanten, die direkt horizontal oder vertikal im Bild laufen. Sie können unabhängig voneinander angewandt werden, um für jede Richtung eigene Messwerte der Gradientengrößen zu liefern. Die Stärke des Gradienten ist dabei gegeben durch $|G| = \sqrt{{G_x}^2 + {G_y}^2}$, was meist durch die einfacher zu berechnende Form $|G| = |G_x| + |G_y|$ angenähert wird. Die Gradientenorientierung θ erhält man mittels $arctan(G_y/G_x)$, falls $G_x \neq 0$. Ein Wert von 0 bedeutet dabei einen Kontrastwechsel von dunkel nach hell und von links nach rechts. Alle anderen Winkel werden von dieser Richtung gegen den Uhrzeigersinn gemessen.

Durch die Glättungsfunktion der Faltungsmasken ist der Sobel-Operator weniger anfällig gegenüber Bildrauschen. Eine Betrachtung der einzelnen Farbkanäle eines Bildes bringt gegenüber der Untersuchung der Intensitätswerte (Graustufen) zusätzlichen Nutzen.

3.3.2 Der LoG/DoG Kanten/Blob-Detektor

Im Gegensatz zum Sobelfilter in Unterabschnitt 3.3.1, der die lokalen Gradienten annähert, wird ein Laplacefilter dazu verwendet, um die zweite Ableitung anzunähern. Dieser Filter gehört zu den *zero-crossing*-Verfahren, bei denen der Nullübergang untersucht wird. Der Laplacewert $L(x, y)$ an der Stelle (x, y) eines Bildes, mit den Intensitätswerten des Bildpunktes $I(x, y)$, ist gegeben durch:

$$L(x, y) = \nabla^2(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Dies kann mit Hilfe einer Faltungsmaske berechnet werden. 3×3 Masken, die die zweite Ableitung im Sinne des Laplacefilters annähern, sind beispielsweise folgende:

0	−1	0
−1	4	−1
0	−1	0

−1	−1	−1
−1	8	−1
−1	−1	−1

Da diese Filter die zweite Ableitung der Bildfunktion annähern, reagieren sie ziemlich sensibel auf Rauschen (quadratischer Term). Dies sieht man leicht im direkten Vergleich mit dem Bild des Sobelfilters (S. 18). Daher wird meistens das Bild vor der Anwendung des Laplacefilters durch einen Gaußfilter geglättet. Diese kombinierte Filteroperation wird als *Laplacian of Gaussian* bezeichnet.

Das Resultat eines skalierungsnormierten LoG-Filters kann auch durch die Differenz zweier verschieden stark geglätteter Gaußbilder angenähert werden. Dieser *Difference of Gaussian* Filter, kurz DoG, kommt beispielsweise in SIFT (siehe Unterabschnitt 3.3.6) zum Einsatz. Dort können die bereits vorhandenen Gaußbilder des Skalenraums kostengünstig durch einfache Differenzbildung als Ersatz für σ^2-normierte *Laplacians of Gaussians* hergenommen werden. Lowe leitet in [Low04] folgenden Zusammenhang zwischen beiden Funktionen her:

$$DoG(x, y, \sigma_1, \sigma_2) \approx \left(\frac{\sigma_2}{\sigma_1} - 1\right) \sigma_1^2 \, LoG(x, y, \sigma_1)$$

Dabei ist $DoG(x, y, \sigma_1, \sigma_2)$ die Differenz zweier Gaußbilder mit σ_2 und σ_1 an der Stelle (x, y) sowie $LoG(x, y, \sigma_1)$ die Laplacefunktion $L(x, y)$ eines Gaußbildes mit Glättungsfaktor σ_1. Der Normierungsfaktor σ_1^2 wird dabei zum Erhalt der Skalierungsinvarianz benötigt.

LoG- und DoG-Filter reagieren generell besonders stark auf Kanten und Ecken, sind jedoch stabiler als die gängigen Gradientenverfahren. Da ihre Maxima und Minima invariant gegenüber Größenänderungen sind, werden sie auch häufig als Detektoren für andere Interessenspunkte eingesetzt. Dazu werden sie beispielsweise auf verschiedene

Skalierungen des Eingangsbildes angewandt und miteinander verglichen (siehe SIFT in Unterabschnitt 3.3.6).

3.3.3 Der Canny-Kantendetektor

Der Canny Operator [FPWW03] wurde von Canny mit dem Ziel entwickelt ein optimaler Kantendetektor zu sein. Er arbeitet auf einem Graustufenbild als Eingabe und kennzeichnet darin die Stellen der aufgespührten Helligkeitsänderungen. Der Canny Operator besteht aus einem mehrstufigen Prozess. Dieser ist in vier Schritte unterteilt:

Schritt 1 - Rauschkompensation Das Bild wird mit einem Gaußfilter geglättet, um dem Bildrauschen entgegenzuwirken.

Schritt 2 - Gradienten erstellen Regionen im Bild, deren erste partielle Ableitung hoch ist, werden durch einen Operator ähnlich dem Sobel-Operator hervorgehoben. Aus diesen Gradientendaten bestimmt man deren Betrag und Orientierung.

Schritt 3 - Unterdrückung von Nichtmaxima Alle Punkte, die kein lokales Maximum in wenigstens einer Richtung (senkrecht zur Kante) sind, werden unterdrückt.

Schritt 4 - Unterdrücken von Kanten Die Kanten werden nun basierend auf den zuvor bestimmten Beträgen der Gradienten und Kantenrichtungen abgefahren und einer hystereseartigen Schwellwert-Prüfung unterzogen. Diese wird durch zwei Schwellwerte, einen oberen und einen unteren Grenzwert, bestimmt. Liegt ein Punkt über dem oberen Wert, wird er sofort übernommen, liegt er unter dem unteren, sofort verworfen. Liegt er zwischen beiden so wird er übernommen, vorausgesetzt er ist mit Punkten verbunden, die über dem oberen Grenzwert liegen. Dadurch werden zu schwache Kanten unterdrückt und Probleme an Ecken gelöst.

Abbildung 3.5f (S. 18) zeigt ein Ergebnis dieses Filters. Dabei wurde für den unteren Schwellwert 0,05 und für den oberen 0,5 gewählt.

3.3.4 Der Harris-Eckendetektor

Einer der bekanntesten Kanten und Eckendetektoren wurde von Harris und Stephens vorgestellt [HS88]. Die Idee dahinter ist, dass man einen Punkt leicht erkennen kann, wenn man ihn durch einen kleinen Ausschnitt (Fenster) betrachtet. Verschiebt man dieses Fenster in irgendeine Richtung, so sollte sich dies in einer großen Änderung der Intensität niederschlagen. Harris und Stephens beschreiben daher ein Maß für die Stärke einer Ecke, das über das Verhältnis der Eigenwerte des Strukturtensors berechnet wird. Dieser beschreibt die Verteilung der Gradienten um den Ursprung $\nabla I = (0, 0)$.

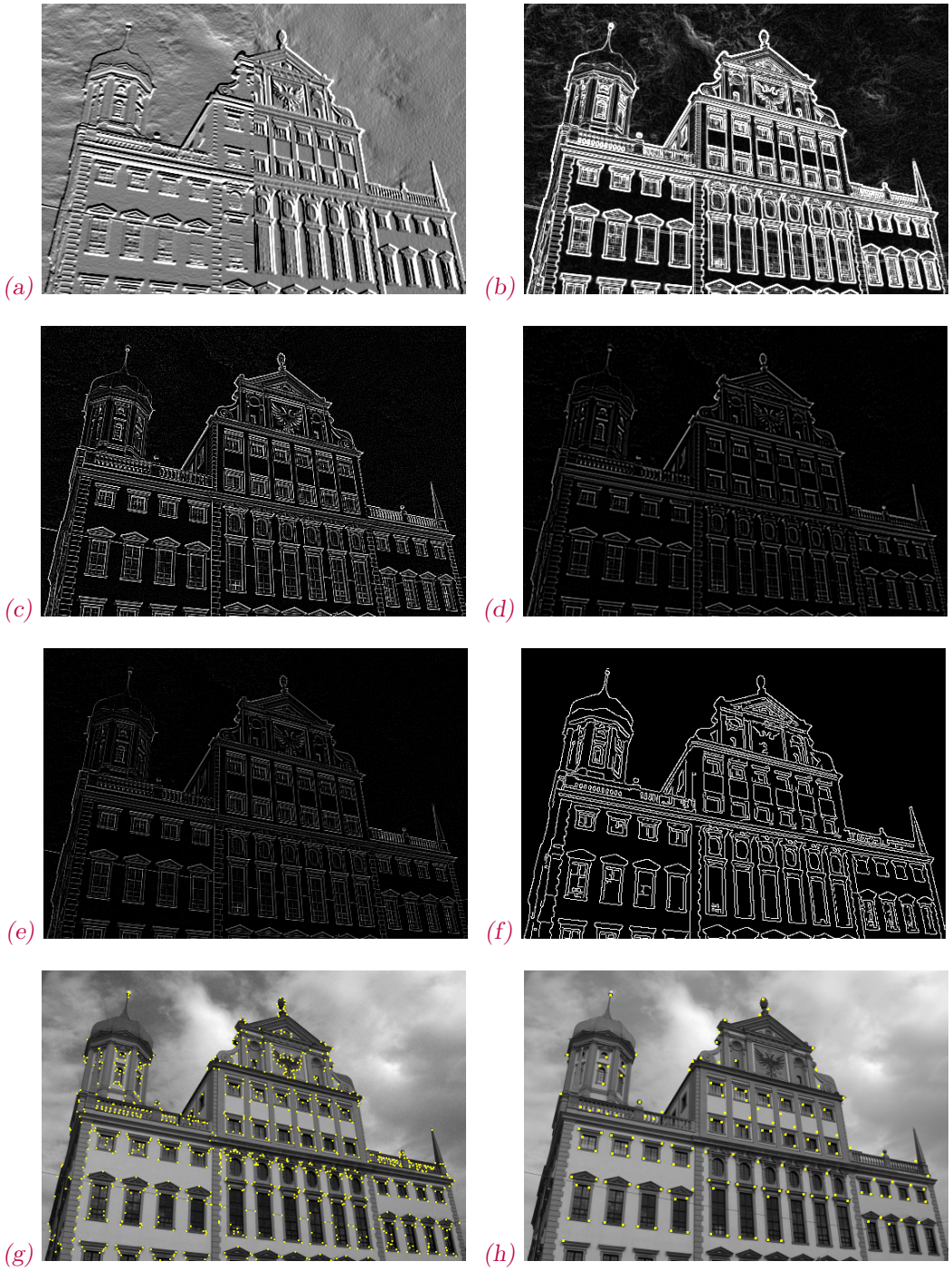


Abb. 3.5: Gegenüberstellung von Kanten- und Eckendedetektoren
(a) horizontaler/vertikaler Sobelfilter, (b) kombinierter Sobelfilter, (c) Laplacefilter, (d) Laplacian of Gaussian mit $\sigma = 1{,}4$, *(e) Difference of Gaussian mit* $\sigma_1 = 1{,}4$ *und* $\sigma_2 = 3{,}0$; *(f) Canny Kantendetektor mit* $\sigma = 1$, $TH_1 = 0{,}2$ *und* $TH_2 = 0{,}05$, *(g) Harris Eckendetektor mit Schwellwert* $0{,}02$, *(h) KLT Eckendetektor*

Der Strukturtensor ist eine symmetrische Matrix aus dem Produkt des Gradientenvektors

$$S = \sum_W \nabla I^T \nabla I = \sum_W \begin{bmatrix} \frac{\partial I}{\partial x}\frac{\partial I}{\partial x} & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & \frac{\partial I}{\partial y}\frac{\partial I}{\partial y} \end{bmatrix}, \tag{3.1}$$

wobei W die Nachbarschaft des Bildpunktes ist, dessen Tensor berechnet wird. Da immer nur kleine Verschiebungen dieses Nachbarschaftsfensters betrachtet werden, lässt sich dies auch durch eine Gaußmaske approximieren. Die Gradienten in x- und y-Richtung können beispielsweise durch die entsprechenden Sobelfilter ermittelt werden.

Da die Matrix S symmetrisch ist, lässt sie sich diagonalisieren und ihre Eigenwerte λ_1, λ_2 sind reell, die zugehörigen Eigenvektoren orthogonal. Es gilt:

1. Sind λ_1 und λ_2 groß, handelt es sich um eine Ecke.
2. Ist nur einer der beiden Eigenwerte groß, gibt es nur eine Änderung in einer Richtung. Dies entspricht einer Kante.
3. Sind λ_1 und λ_2 fast 0, so gibt es an dieser Stelle keine signifikante Richtungsänderung (homogene Fläche).

Um die Stärke der Kante bewerten zu können, schlagen Harris und Stephens folgendes Maß vor:

$$M_c = \lambda_1\lambda_2 - \kappa\,(\lambda_1 + \lambda_2)^2$$

Hierbei ist κ ein regelbarer Parameter, der angibt, wie stark der Algorithmus auf Kanten reagieren soll. In der Praxis hat sich ein Wert in der Größenordnung von 0,05 bewährt. Sie schlagen zudem eine andere Berechnungsvariante vor, bei der auf die Eigenwertzerlegung von S verzichtet werden kann:

$$M_c = \det(S) - \kappa\ \mathrm{Spur}^2(S) \tag{3.2}$$

Mittels Schwellwertanwendung und Unterdrückung von lokal nichtmaximalen Eckpunkten können so signifikante Eckpunkte herausgefiltert werden. Der Antwortwert M_c des Harris-Detektors ist invariant gegenüber Rotation, da die Größe der Gradienten und damit ihr Verhältnis gleich bleibt. Er ist auch teilweise invariant gegenüber Helligkeitsänderungen, da die Extrema dieselben bleiben. Das Endresultat kann jedoch aufgrund des veränderten Rauschanteils im Bild und dem gewählten Schwellwert bei der Auswertung abweichen. Die Merkmale des Harris/Stephens-Detektors sind jedoch nicht invariant gegenüber Größenänderungen.

Der Eckendetektor von Harris und Stephens ist Grundlage vieler anderer Ecken und Merkmalsdetektoren, wie der im Folgenden behandelte KLT-Tracker oder SIFT

in Unterabschnitt 3.3.6, bei denen eine leicht abgewandelte Version des Maßes für die Eckigkeit eines Punktes verwendet wird.

3.3.5 KLT - der Kanade-Lucas-Tomasi Tracker

Shi und Tomasi haben in [ST94] einen Algorithmus vorgestellt, der für das Tracking optimale Merkmale auswählt und diese im Auge behalten kann, den KLT (Kanade-Lucas-Tomasi) Tracker. Dieser wird nach über 20 Jahren immer noch in verschiedenen Bereichen des Maschinellen Sehens eingesetzt und ist zudem als freie, effiziente Implementierung verfügbar [Bir]. Der Grundgedanke bei KLT ist, dass nur solche Merkmale als gut angesehen werden, die sich auch gut verfolgen lassen. Daher sollte nach Shi und Tomasi das Tracking nicht von der Auswahl der Merkmale getrennt werden.

Ein Punkt ist ein gutes Merkmal, wenn an seiner Position starke Kontraste in sowohl der x als auch in der y-Richtung zu finden sind. Dies trifft beispielsweise bei einer Ecke zu. Die Auswahl der zu verfolgenden Merkmale geschieht genau wie beim Harris-Eckendetektor in Unterabschnitt 3.3.4 durch Betrachten der Eigenwerte des 2×2 Strukturtensors S (siehe Gleichung 3.1 auf S. 19), der die Intensitätsänderung der jeweiligen Bildregion beschreibt. Im Unterschied zur Methode von Harris und Stephens wird jedoch anstatt des Verhältnisses beider Eigenwerte nur der kleinste Eigenwert als Auswahlkriterium verwendet.

$$\lambda_{min} = \min \operatorname{eig}(S) = \min \operatorname{eig} \left(\sum_W \begin{bmatrix} \frac{\partial I}{\partial x}\frac{\partial I}{\partial x} & \frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \frac{\partial I}{\partial y}\frac{\partial I}{\partial x} & \frac{\partial I}{\partial y}\frac{\partial I}{\partial y} \end{bmatrix} \right)$$

Ein 25×25 Bildpunkte großes Fenster W wird dabei als Kandidat für ein Merkmal akzeptiert, wenn in dessen Mittelpunkt λ_{min} einen bestimmten Schwellwert überschreitet. Die Mittelpunkte benachbarter Kandidaten müssen zudem einen bestimmten Mindestabstand einhalten, was durch eine Unterdrückung nichtmaximaler Punkte (*non-maximal suppression*) erreicht wird. Im ersten Bild des Videostroms werden alle Merkmalskandidaten absteigend nach ihrem kleinsten Eigenwert in einer Liste L sortiert und die N größten ausgewählt.

Das KLT-Verfahren definiert ein Maß für die Unähnlichkeit (*dissimilarity*) zweier Merkmale, das den Grad der Änderung des Auftretens eines Merkmals zwischen dem ersten und dem aktuellen Videobild bestimmt, wobei affine Bildänderungen berücksichtigt werden können.

Gleichzeitig verwendet man ein Bewegungsmodell ähnlich dem Optischen Fluss, um die ausgewählten Merkmale über die Bildsequenz hinweg im Auge zu behalten. Um eine zuverlässige und schnelle Bearbeitung zu ermöglichen, wird der maximale Bilderabstand beschränkt. Trotzdem kann er im Vergleich zu konventionellen Ansätzen des

Erstellen der Merkmalsliste

1. Berechne Gradienten für jeden Bildpunkt p
2. Berechne für jeden p den kleinsten Eigenwert λ_{min} des Strukturtensors S in einem Fenter W um p
3. Ist $\lambda_{min} > \lambda_{thres}$ speichere p in einer absteigend sortierten Liste L
4. Entferne von oben nach unten alle Punkte aus L, die in der Umgebung des aktuellen p liegen

Tracking

0. Speichere Merkmale des ersten Bildes in einer Referenzliste L_0
1. Berechne für jedes p in L_0 die Koeffizienten von A und b aus Gleichung 3.3
2. Löse $A\,d = b$
3. Werte d aus und aktualisiere Punktverfolgung
4. Wiederhole Schritte 1-3 für jedes Bild; Erneuere alle x Bilder die Referenzwerte

Abb. 3.6: Ablauf des KLT-Algorithmus
Links das Herausfiltern und Finden der Merkmale, rechts der Trackingvorgang; nach [SFPG06]

Optischen Flusses nach Shi und Tomasi oft größer ausfallen [ST94].

In Abbildung 3.6 ist der komplette Ablauf des KLT-Tracking Algorithmus nochmals übersichtlich dargestellt.

Der eigentliche Teil des Algorithmus, der später dann auch in der GPU-Implementierung von Sinha in Abschnitt 5.2 für die Verfolgung der Merkmale zuständig ist, berechnet die Verschiebung der Merkmale in aufeinanderfolgenden Videobildern, unter der Annahme, dass die Beleuchtung konstant und die Bewegung im Bild relativ klein ist [SFPG06]. Die Verschiebung des Merkmals wird innerhalb eines „Tracking"-Fensters um die Position des Merkmales herum in zwei Bildern berechnet. Hierbei wird das Newton-Verfahren verwendet, um die Summe der Differenzenquadrate (SSD) zu minimieren.

Sei $I(*, *, t)$ die Repräsentation des Videobildes zum Zeitpunkt t. Sei ferner die Verschiebung eines Bildpunktes (x, y) zwischen der Zeit t und $t + \Delta t$, bezeichnet durch $(\Delta x, \Delta y)$, klein, so gilt laut Sinha [SFPG06] unter Annahme gleichbleibender Szenenbeleuchtung:

$$I(x, y, t + \Delta t) = I(x + \Delta x, y + \Delta y, t).$$

Der Einfachheit halber sei $p = (x, y)^T$ und $d = (\Delta x, \Delta y)^T$, dann berechnet der KLT Algorithmus bei vorhandenem Bildrauschen r mit

$$I(p, t + \Delta t) = I(p + d, t) + r,$$

den Verschiebungsvektor d, der den folgenden Fehler

$$r = \sum_W (I(p + d, t) - I(p, t + \Delta t))^2$$

über einem kleinen Bildausschnitt W minimiert. Nähert man $I(p+d,t)$ durch die Taylorentwicklung an, so erhält man ein lineares System zur Bestimmung der Unbekannten d:

$$\underbrace{(\sum_{W} G^T G)}_{\mathbf{A}}(d) = \underbrace{\sum_{W} G^T \Delta I(p, \Delta t)}_{\mathbf{b}} \tag{3.3}$$

Dabei ist $G = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]$ der Gradientenvektor des Bildes im Punkt p. Sinha verwendet für seine Implementierung in Abschnitt 5.2 eine Variation der ursprünglichen KLT Gleichung, die später von Tomasi veröffentlicht wurde und beide Bilder symmetrisch betrachtet [SFPG06]. Sie unterscheidet sich von Gleichung 3.3 nur in der Definition von G:

$$G = \left[\frac{\partial(I(*,t) + I(*,t+\Delta t))}{\partial x}, \frac{\partial(I(*,t) + I(*,t+\Delta t))}{\partial y}\right].$$

Da die angenommene Linearität nur für kleines d gilt, verwendet man in der Praxis nach [SFPG06] oft einen KLT-Tracker, der auf mehreren Auflösungsstufen agiert, um größere Bewegungen besser handhaben zu können. Hierbei wird erst in grober Auflösung „getrackt“ und dann das Ergebnis in feineren Auflösungen verbessert. Mehrfaches Iterieren über jede Auflösungsstufe kann die Genauigkeit zusätzlich noch erhöhen. Da sich durch Kamerabewegung und Verdeckungen einige Merkmalspunkte verlieren können, müssen von Zeit zu Zeit neue ausgewählt werden, um eine konstante Anzahl an Merkmalspunkten zu gewährleisten [ST94].

Der KLT-Algorithmus kann, aufgrund des speziellen Maßes für die Unähnlichkeit zweier Merkmale, diese selbst unter affinen Transformationen korrekt erkennen.

3.3.6 SIFT - Scale Invariant Feature Transform

Im Folgenden soll nur ein kurzer Überblick über das *SIFT*-Verfahren gegeben werden. Detaillierte Informationen sind in der ausführlichen Veröffentlichung von Lowe [Low04] sowie in Beschreibungen von anderen Implementierungen [Hey05] vorhanden.

Die skalierungsinvariante Merkmalstransformation, *Scale Invariant Feature Transform* (SIFT), ist ein gängiger Algorithmus um gegenüber Verschiebung, Rotation, Größenänderung und Änderungen der Beleuchtungshelligkeit invariante Interessenspunkte aus Bildern zu gewinnen. SIFT ist dabei eine Kombination aus mehreren klassischen Ansätzen.

Ablauf Wie in Abbildung 3.7 illustriert, wird zuerst eine gaußsche Skalenraumpyramide aus dem Eingangsbild erzeugt und die Gradienten sowie DoG-Bilder (*Difference of Gaussian*, siehe Unterabschnitt 3.3.2) der verschiedenen Skalierungen berechnet. Diese sind in einer bestimmten Anzahl von Oktaven angeordnet, innerhalb derer sich

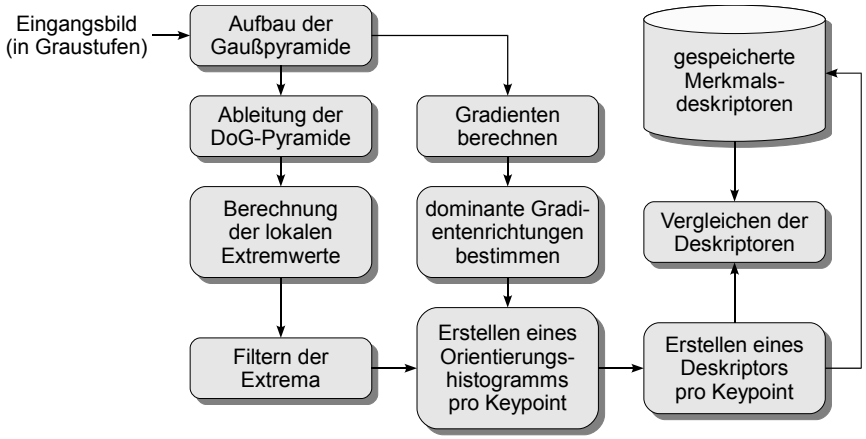


Abb. 3.7: SIFT - Modell des Ablaufs

der Skalierungsfaktor σ verdoppelt. Das Bild, das mit 2σ geglättet wurde, wird dann als Ausgangsbild für die nächste Oktave hergenommen. Abbildung 3.8 zeigt einen Ausschnitt dieser Pyramide. Die Interessenspunkte (Schlüsselpunkte) werden dann über die lokalen Extrema innerhalb des DoG-Skalenraums durch Vergleichen mit den jeweils neun Punkten der benachbarten sowie den acht direkt angrenzenden Punkten bestimmt. Da der DoG-Filter jedoch auch auf Kanten anspricht, werden in einem weiteren Nachbearbeitungsschritt die Eigenwerte der Hessematrix des Bildes untersucht, ähnlich wie beim Eckendetektor von Harris und Stephens in Unterabschnitt 3.3.4. Ist ihr Unterschied zu groß, wird der Punkt als zu kantenartig verworfen. Auch Punkte mit einem zu schwachen Intensitätswert werden aussortiert. Anschließend werden für jeden dieser Schlüsselpunkte die dominanten Gradientenrichtungen in einem von σ abhängigen Radius um den Punkt mittels Histogrammen in 10° Schritten (36 Histogrammbehälter) bestimmt. Dadurch können die Schlüsselpunkte später besser wiedererkannt werden.

Relativ zu jeder dieser dominanten Gradientenrichtungen werden nun in einem 4×4 großen Bereich um den Schlüsselpunkt, dessen Feldgröße wieder abhängig von σ ist, Orientierungshistogramme erstellt, die jeweils acht Hauptrichtungen erfassen. Diese werden dann rotationsinvariant in Form eines 128-elementigen Merkmalsdeskriptors gespeichert.

Tracking Mittels SIFT können ganze Objekte in Videoströmen verfolgt werden. Dazu müssen lediglich die charakteristischen Deskriptoren für dieses Objekt bestimmt werden und dann mit den Deskriptoren des jeweiligen Bildes verglichen werden. Laut Lowe [Low04] sind je nach Objektgröße und Auffälligkeit der Umgebung im Bild drei bis zehn Merkmale für eine zuverlässige Zuordnung nötig.

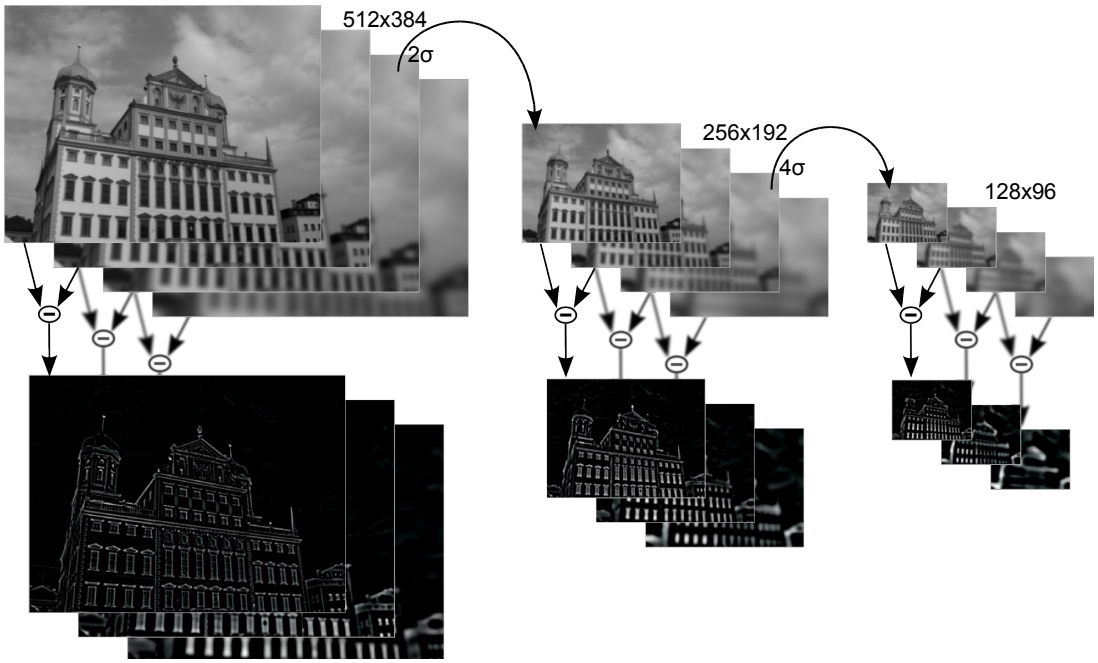


Abb. 3.8: Ausschnitt der gaußschen Skalenraumpyramide des SIFT Algorithmus
Die DoG-Bilder werden durch Differenzbildung aus den Gaußbildern gewonnen. Jeweils das Bild, das mit dem zweifachen σ geglättet wurde, dient als Ausgangsbild für die nächste Oktave.

Abb. 3.9: SIFT-Merkmalspunkte
Die SIFT-Merkmalspunkte wurden mit dem Referenzprogramm von Lowe erzeugt [Low04].

Die beste Übereinstimmung erzielt man laut Lowe mit einer Nachbarschaftssuche in einer Datenbank mit Referenzmerkmalspunkten. Ein Nachbar ist dabei ein Merkmalspunkt, dessen Deskriptorvektor den geringsten euklidischen Abstand hat. Da keine Algorithmen bekannt sind, die den exakten nächsten Nachbarn von Punkten in höherdimensionalen Räumen bestimmen können und effizienter sind als eine vollständige Suche, verwendet Lowe den approximativen *Best-Bin-First* (BBF) Algoritmus [BL97].

Dieser liefert den Nachbarn, der mit hoher Wahrscheinlichkeit der nächstgelegene ist.

Posenbestimmung Eine Bestimmung der Kamerapose kann hier mittels Initialisierung über ein kalibriertes Referenzbild erfolgen, indem die aktuell extrahierten SIFT-Merkmale mit den Referenzwerten abgeglichen werden und daraus die Pose berechnet wird.

3.3.7 RANSAC

Die Abkürzung RANSAC steht für *Random Sample Consensus* [FB81] und bezeichnet ein statistisches Verfahren, das Parameter eines mathematischen Modells aus beobachteten Daten bestimmt. Der Algorithmus geht dabei davon aus, dass die Daten sogenannte *Inlier*, das sind Punkte, die sich durch Parameter des Modells bestimmen lassen, und Ausreißer, welche nicht zum Modell passen, beinhalten.

Im Maschinellen Sehen wird der RANSAC-Algorithmus häufig zur Bestimmung der Korrespondenzen mehrerer Bilder eingesetzt. Auch einige der in dieser Arbeit besprochenen Methoden (siehe Abschnitt 5.4 und Abschnitt 5.3) verwenden dieses Verfahren. Der Algorithmus soll hier kurz am Beispiel der Posenbestimmung dargestellt werden:

1. Wähle eine minimale, zufällige Menge an Korrespondenzen und berechne die Pose
2. Überprüfe alle Korrespondenzen mit dieser Pose und bestimme die Anzahl der *Inlier*
3. Wiederhole dies solange, bis ein bestimmtes Kriterium k erfüllt ist; k kann dabei beispielsweise ein gewünschtes Verhältnis der *Inlier* zu den Ausreißern sein.

Ein Vorteil dieses Verfahrens ist, dass sich damit Parameter robust bestimmen lassen, selbst wenn in den Daten Ausreißer vorhanden sind. Als Nachteil kann das Fehlen einer zeitlichen oberen Schranke für die Berechnung der Parameter gesehen werden. Wird dennoch eine Zeitschranke benutzt, kann ein optimales Ergebnis nicht garantiert werden. In der Praxis nimmt man jedoch eine Ungenauigkeit in einem gewissen Rahmen in Kauf.

3.3.8 Weitere Verfahren

Neben den vorgestellten Vertretern der Merkmalsdetektoren gibt es natürlich auch noch einige andere bekannte Verfahren. Diese spielen allerdings für dir GPU-Umsetzungen in Kapitel 5 keine Rolle und werden deswegen hier nur der Vollständigkeit halber genannt und nicht weiter erläutert.

Der Merkmalsdetektor *FAST (Features from Accelerated Segment Test)* von Rosten und Drummond [RD05, RD06], der Bildpunkte in einem *Bresenham Kreis*[2] um den Kandidatenpunkt herum betrachtet und die Helligkeitswerte der umliegenden Punkte in diesem Kreis auswertet. Sind alle um mindestens x heller oder dunkler als der Kandidat, so ist dieser ein Merkmalspunkt. Diese gefundenen Merkmale sind zudem ziemlich robust [RD06]. Durch Verwenden eines Entscheidungsbaumes (ID3 Algorithmus) lassen sie sich zudem äußerst effizient bestimmen.

Der *SUSAN (Smallest Univalue Segment Assimilating Nucleus)* Merkmalsdetektor von Smith und Brady [SB95] verwendet, ähnlich wie bei FAST, eine kreisförmige Maske, innerhalb der die Helligkeit der Bildpunkte untersucht wird.

Beim *SURF*-Verfahren *(Speeded Up Robust Features)* setzen Bay, Tuytelaars und Van Gool [BTG06] auf bekannte Verfahren und reduzieren diese sinnvoll auf das Wesentlichste, wie beispielsweise ganzzahlige Faltungsmasken, um dadurch einen performanten Interessenspunktdetektor zu erhalten, der gegenüber Größenänderungen und Rotation invariant ist.

[2] Ein *Bresenham Kreis* wird durch eine Variante des Breseham Linienalgorithmus bestimmt. Dieser wird verwendet, um zu bestimmen, welche Punkte bei der Darstellung in einem Raster gezeichnet werden sollen. Ein darauf aufbauendes Verfahren ist beispielsweise auch der *Midpoint line*-Algorithmus [BB03, S. 57], der in [Möl05] besprochen wurde.

4 Rechnen auf der Grafikkarte - GPGPU

GPGPU steht für *General Purpose computation on Graphics Processing Units* und bezeichnet das Ausnutzen des Grafikprozessors zur Berechnung von nicht-grafischen Daten. In Kapitel 2 wurde erklärt, warum GPUs im Vergleich zu CPUs ein deutlich größeres Leistungspotential bieten. Beim GPGPU wird versucht dieses Potential für allgemeine Anwendungen zu nutzen.

Anwendungen für die Merkmalsverfolgung müssen Unmengen an Bilddaten verarbeiten können. Oft wird dabei eine einzelne Operation auf das gesamte Bild angewandt. Daher wäre ein Prozessor sinnvoll, der diese Operation auf allen Bildpunkten gleichzeitig ausführen könnte. Streamprozessoren verarbeiten Datenströme aus vielen gleichartigen Elementen und wenden auf diese bestimmte Kernoperationen parallel an. In Abschnitt 4.3 wird gezeigt, dass sich moderne GPUs ideal auf das Modell eines Streamprozessors abbilden lassen. Das dort vorgestellte Stream-Programmiermodell ist eine Abstraktion, die es erlaubt, die Programmierung auf der GPU ohne ihren grafischen Bezug zu betrachten.

Zunächst wird jedoch die Architektur heutiger GPUs erläutert, um die nötigen Hintergründe zu vermitteln. Dabei werden die einzelnen Stufen der Grafikpipeline ausführlich erklärt und es wird auf die programmierbaren Teile der GPU gesondert eingegangen. Des Weiteren wird untersucht, mit welcher Genauigkeit sich auf heutigen GPUs rechnen lässt, und welche Einschränkungen damit verbunden sind.

Im weiteren Verlauf dieses Kapitels wird dann auf Aspekte der Softwareseite einer GPU-Implementierung eingegangen. Dabei werden Shadersprachen vorgestellt, der grundlegende Programmablauf beschrieben und besondere Techniken der Grafikschnittstelle erläutert.

4.1 GPU-Architektur

In der Computergrafik beschreibt die Grafikpipeline, auch Rendering-Pipeline genannt, den Ablauf von der vektoriellen, mathematischen Beschreibung einer Szene zum gerasterten Bild auf dem Monitor. Die Grafikpipeline führt dabei verschiedene Aufgaben, wie beispielsweise die Koordinatentransformation, die Texturierung der Szenenobjekte oder ihre Beleuchtung durch.

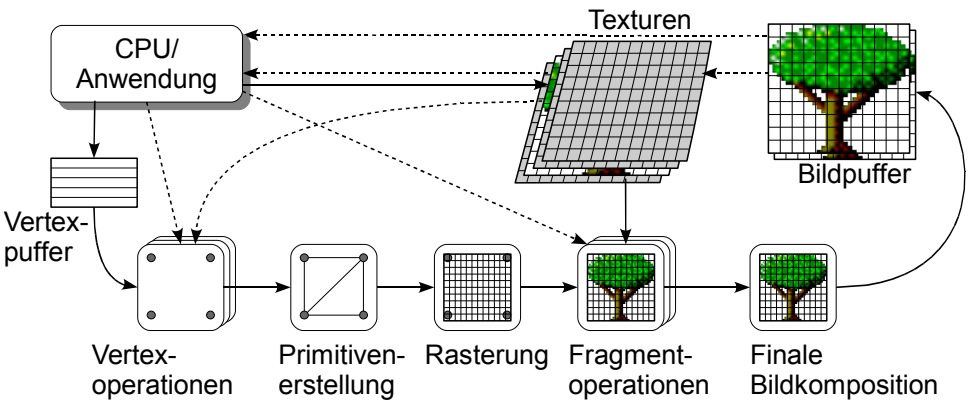


Abb. 4.1: Vereinfachte Darstellung der Grafikpipeline
Die durchgängigen Linien stellen den Verlauf der ursprünglichen Fixed-Function-Pipeline dar. Die möglichen Wege der modernen programmierbaren Pipeline sind durch die gestrichelten Linien illustriert. In der Vertex-Stufe können statt der üblichen Transformations- und Beleuchtungsberechnungen benutzerdefinierte Programme ausgeführt werden. Auch die Schattierung und Texturierung der Fragmente kann heute vom Benutzer kontrolliert werden.

Abbildung 4.1 zeigt den schematischen Aufbau der Grafikpipeline, nach dem heutige GPUs funktionieren. Diese ist so konzipiert, dass eine hohe Berechnungsrate durch parallele Ausführung erreicht werden kann. Die dargestellte Unterteilung der Pipelinestufen muss sich jedoch nicht hundertprozentig mit der Realisierung in Hardware decken. Die einzelnen Stufen der Pipeline werden von allen geometrischen Primitiven, wie Punkte, Linien oder Dreiecke, durchlaufen:

1. **Vertexoperationen:** *Vertices* sind Geometriepunkte, die die Primitive im Raum beschreiben. Sie sind charakterisiert durch 2D-/3D-Koordinaten, Farb- und Materialwerte, Normalenvektor und Texturkoordinaten.

   In der ursprünglichen Vertex-Transformations-Stufe wird die Transformation der Vertexposition, die Beleuchtungsberechnung und die Generierung und Transformierung von Texturkoordinaten vorgenommen. In heutigen GPUs lassen sich diese Bearbeitungsschritte durch benutzerdefinierte Programme festlegen, die dann für jeden Vertex auf dem Vertexprozessor (Abbildung 4.2) ausgeführt werden.

2. **Zusammenstellung der Primitive und Rasterung:** Die Vertices werden anschließend zu ihren Primitiven gruppiert und verschiedene Sichtbarkeitsbetrachtungen pro Primitiv durchgeführt. Dazu gehört das Aussondern nicht sichtbarer Objekte (*culling*) und das Abschneiden der Objekte an den Grenzen der 2D-Darstellungsfläche (*clipping*). Diese Schritte werden beispielsweise bei der GeForce 6 Architektur auch vom Vertexprozessor ausgeführt (siehe Abbildung 4.2).

   Bei der Rasterung wird das durch die Vertexdaten beschriebene Primitiv in *Frag-*

mente, das sind potentielle Pixelkandidaten, zerlegt. Dabei werden gleichzeitig Fragmente verworfen, die von Objekten mit kleinerem Z-Wert verdeckt sind. Dies bezeichnet man als (*Z-Culling*). Der Z-Wert ist dabei der Abstand zwischen Vertex und Kamera. Durch die Rasterung wird eine projizierte Fragmentdarstellung der Primitive erzeugt.

3. **Fragmentoperationen:** Nach der Rasterung werden die Vertexdaten, wie Farbe und Texturkoordinaten, für jedes Fragment interpoliert. In der Fragment-Texturierungs-Stufe, oft auch als Pixelshader-Pipeline bezeichnet, können die interpolierten Farbwerte mit Texeln (*texture pixel*) kombiniert werden. Die Texturdaten können optional bilinear, trilinear oder anisotrophisch gefiltert werden.

   Der Fragmentprozessor kann Farbwerte aus mehreren Texturen lesen und diese auf alle erdenklichen Arten mit den Farbwerten des Fragments kombinieren. Dabei arbeitet der Fragmentprozessor laut Kilgariff und Fernando [KF05] zu einem Zeitpunkt immer auf Gruppen aus Hunderten von Pixeln in *single-instruction, multiple-data* (SIMD) Manier. Dabei wird die Latenz der Texturzugriffe durch die Berechnungen des Fragmentprozessors verdeckt [Buc05,OLG+07]. Er unterstützt auf heutigen GPUs Texturen und Berechnungen in einfacher Gleitkommagenauigkeit nach IEEE-754 (siehe Unterabschnitt 4.1.4).

4. **Finale Bildkomposition (Rasteroperationen):** Die Fragmente verlassen die Fragmentprozessor-Einheit in derselben Reihenfolge, wie sie gerastert wurden und werden zu den Tiefentest- und Blendeinheiten geschickt. Diese führen einen Tiefentest (Z-Vergleich und Aktualisierung), Schablonentest, Alpha-Blending und das finale Schreiben des Farbwerts in den Bildpuffer [KF05]. Durch *Multiple Render Targets* (MRT, Abschnitt 4.5) lassen sich auf heutigen GPUs an dieser Stelle bis zu vier Renderziele gleichzeitig beschreiben.

Jede dieser Stufen ist als eigenständiger Baustein auf der GPU in Hardware realisiert. Dies bezeichnet man laut Owens et al. [OLG+07] als aufgabenparallele Maschinenorganisation. Da Pixel in der Computergrafik in der Regel aus vier Farbkomponenten (RGBA) bestehen, sind die Vertex- und Fragmentprozessoren als 4D-Vektorprozessoren realisiert. Das bedeutet, dass sie pro Takt vier parallele Berechnungen durchführen können. Diese können je nach Architektur auch in 3+1 Berechnung pro Takt aufgeteilt werden (*co-issue*), was beispielsweise bei Berechnungen mit 3D-Koordinaten sinnvoll sein kann. Grafikkarten müssen bei heutigen Anwendungen, wie etwa Videospielen, in der Regel deutlich mehr Modulierungen an den Farbwerten als an den Vektor- und Geometriedaten vornehmen. Deswegen steht meist eine deutlich größere Anzahl an Pixelshadern als an Vertexshadern zur Verfügung. Bei der GeForce 7900 GTX sind dies

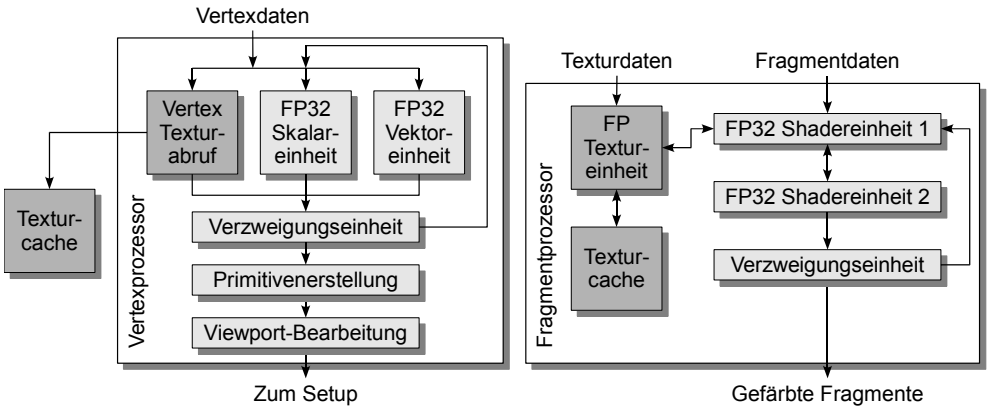


Abb. 4.2: Vertex- und Fragmentprozessor der GeForce 6 Serie [KF05]

beispielsweise 24 Pixelshader und 8 Vertexshader. Im Anhang (S. 83) ist eine Übersicht mit den technischen Daten einiger GPUs der letzten Jahre dargestellt. Diese zeigen, wie sich die Anzahl der Shadereinheiten oder der Textureinheiten (TMUs) entwickelt hat. Die Architektur der NVIDIA GeForce 6 Serie wurde von Kilgariff und Fernando beschrieben [KF05]. Aus Software-Sicht ist der *GPU Programming Guide* [NVI05b] oder der *CUDA Programming Guide* [NVI07a] von NVIDIA sowie das OpenGL Programming Guide [OSW+05] zu nennen.

4.1.1 Programmierbare Grafikhardware

Die Fixed-Function-Pipeline ist über die letzten acht Jahre zu einer flexibleren, programmierbaren Pipeline geworden. Von den Änderungen sind hauptsächlich die Vertex- und die Fragmentstufe der Pipeline betroffen: die Operationen auf den Vertices, wie etwa Beleuchtung und Transformation sowie die Operationen zur Bestimmung der Fragmentfarbwerte wurden in der programmierbaren Pipeline durch benutzerdefinierte Programme ersetzt. Mit jeder neuen GPU-Generation ist bisher die Funktionalität und die allgemeine Nutzbarkeit dieser Stufen weiter verbessert worden. Während die Programmierbarkeit dieser Stufen 1999 noch aus einer eingeschränkten Kombinierbarkeit von Textur- und interpoliertem Farbwert zur Berechnung der Fragmentfarbe bestand [OLG+07], stellen heutige GPU-Spitzenmodelle Hochleistungs-Streamprozessoren dar, auf die über spezielle APIs aus Sprachen, wie C oder C++, zugegriffen werden kann [NVI07a].

Der entscheidende Schritt für die Ermöglichung von *general-purpose* Berechnungen auf der GPU war nach Owens et al. die Einführung einer vollständig programmierbaren Hardware sowie einer Assemblersprache, mit der Programme definiert werden konnten, die auf jedem Vertex oder Fragment ausgeführt wurden [OLG+07].

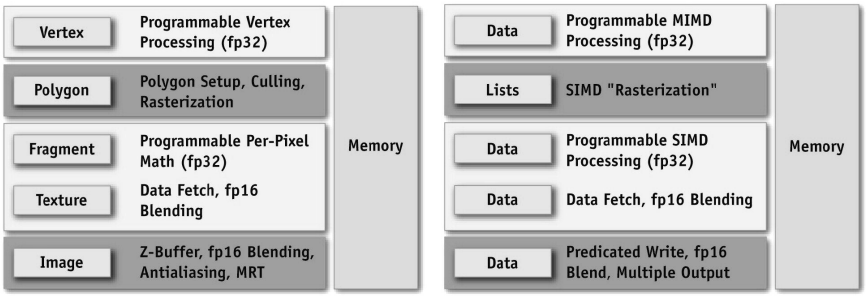


Abb. 4.3: Die Architektur der GeForce 6 Serie
Die normale Grafikpipeline (linkes Bild) bietet auch aus Sicht einer nicht-grafischen Anwendung eine interessante Plattform für die Datenverarbeitung (rechtes Bild). Grafik entnommen aus [KF05]

Diese programmierbare Shader-Hardware ist explizit dafür ausgelegt, mehrere gleichartige Primitive zur selben Zeit parallel zu bearbeiten. Obwohl 2006 die Vertex- und Pixelshader-Standards beide in Revision 3 vorliegen und offizielle OpenGL Erweiterungen für beide existieren, bieten sie im Vergleich zur CPU dennoch nur einen eingeschränkten Befehlssatz. Dieser enthält hauptsächlich grundlegende arithmetische und trigonometrische Befehle (`sin`/`cos`, Skalarprodukt, etc.) sowie grafikspezifische Befehle (Texturdaten lesen, Lichtkoeffizienten oder Brechungsvektor berechnen, etc.). Als neueste Operationen sind eingeschränkte Befehle zu Programmflusskontrolle hinzugekommen.

Wie auch bei Owens et al. [OLG+07] nachzulesen ist, haben diese programmierbaren Stufen generell eine begrenzte Anzahl an 32-bittigen Gleitkommavektoren mit jeweils vier Komponenten als Eingabe. Die Vertex-Stufe gibt dabei eine begrenzte Anzahl von vier-elementigen Gleitkommavektoren aus und lässt diese dann von der Rastereinheit interpolieren. Die Fragment-Stufe kann bis zu vier vier-elementige Gleitkommavektoren ausgeben (siehe MRTs, S. 45). Dabei handelt es sich typischerweise um Farbwerte. Des Weiteren kann in beiden programmierbaren Stufen auf allgemein gültige Konstantenregister und auf eigene Schreib- und Lese-Register zugegriffen werden.

Einschränkungen

Die Möglichkeiten der Shadereinheiten sind durch die festgelegte Anzahl der Eingabe- und Ausgabedaten, der verfügbaren Konstanten, und durch die Anzahl und Art der Register (je nach GPU gibt es verschieden viele 16 und 32-bittige) sowie dem Befehlssatz beschränkt. Diese Einschränkungen wurden allerdings mit jeder neuen Revision des Vertex- und Pixelshader-Standards verringert (siehe Unterabschnitt 4.1.3).

Gather und Scatter Fragmentprozessoren können zwar Daten von belieben Positionen aus Texturen lesen, was einer *gather*-Operation auf dem Speicher entspricht. Da die jeweilige Ausgabeadresse schon von vornherein feststeht, sind sie jedoch nicht zu *scatter*-Operationen (siehe S. 40), dem Schreiben von Daten an beliebige Positionen im Speicher, in der Lage. Im Vertexprozessor ist es jedoch, durch Verändern der Positionen der Vertices möglich, Einfluss auf die Auswahl der zu zeichnenden Bereiche des Bildes zu nehmen. Da heutige Vertexprozessoren mittlerweile auch Daten aus Texturen lesen können, lassen sich dort sowohl *gather*- als auch *scatter*-Operationen realisieren. Laut Owens et al. [OLG+07] kann *Vertex-scatter* jedoch im weiteren Verlauf der Pipeline zu Kohärenzproblemen zwischen Speicher und Rastereinheit führen.

4.1.2 Spezifikationen

Im Laufe der Entwicklung der Shadereinheiten wurden immer wieder neue Standards und Modelle eingeführt, die die Funktionalitäten der Einheiten beschreiben. Das Wichtigste davon ist das *Shader-Modell*, das inklusive der Version 3.0 für Pixel- und Vertexshader unterschiedliche Definitionen führte. Im Shader-Modell 4.0, das 2006 vorgestellt wurde, gibt es diese Trennung der beiden Shader, aufgrund der eingeführten Unified-Shader-Architektur (Unterabschnitt 4.1.3), nicht mehr. Die Entwicklung der Shadereinheiten kann in vier Stufen unterteilt werden, die in Tabelle A.2 zusammengefasst sind [Bly06].

Neben dem Shader-Modell gibt auch Microsoft mit den Versionen der Direct3D-Schnittstelle Vorgaben über die zu unterstützenden Funktionen. Für Direct3D 10 ist beispielsweise das Shader-Modell 4.0 Grundlage. Der Funktionsumfang wird dort jedoch etwas genauer spezifiziert. Die Entwicklung von OpenGL hat hingegen kaum Einfluss auf die Entwicklung der Grafikhardware. Es ist im Gegenteil so, dass der OpenGL-Standard einfach durch Erweiterungen der GPU-Hersteller ergänzt wird. Blythe fasst in seiner Veröffentlichung [Bly06] alle wesentlichen Hardware- und Software-Änderungen zusammen, die die neue GPU-Generation charakterisieren. Auf die wichtigsten Neuerungen wird in dem folgenden Unterabschnitt eingegangen.

4.1.3 Ausblick auf die Unified-Shader-Architektur

Die GPU-Architektur hat sich bei NVIDIAs Nachfolgemodellen bis vor kurzem nicht wesentlich verändert. Auch die GPUs von AMD/ATI folgten dem eingangs beschriebenen Schema. Allerdings bringen die für Direct3D 10 kürzlich eingeführten Unified-Shader-GPUs von NVIDIA (G80) und AMD (R600) ein paar Änderungen mit sich, die kurz vorgestellt werden. Die Architektur der neuen Direct3D-10-Grafikchips wird beispielsweise von Bertuch [Ber07] oder Sturm [Stu06] in Einzelheiten beschrieben.

Eine der wichtigsten Neuerungen ist die Einführung einer zusätzlichen Shadereinheit, dem Geometrieprozessor. Dieser arbeitet im Unterschied zum Vertexprozessor anstatt auf Vertices auf ganzen Primitiven. Er hat außerdem die Möglichkeit, neue Geometrieobjekte zu erzeugen und diese in die Rendering-Pipeline einzufügen. Dies kann etwa dazu benutzt werden, Dreiecksnetze feiner zu unterteilen oder gekrümmte Flächen besser anzunähern. Die Geometrieshader werden nach den Vertextransformationen und vor dem Clipping in der ursprünglichen Grafikpipeline ausgeführt. Von dieser neuen Einheit wird erwartet, dass sie beispielsweise bei der Berechnung von Schattenvolumen oder Partikelsystemen Vorteile bringt.

Ein wesentlicher Grund für die Einführung eines vielseitigen Shadertyps ist die Tatsache, dass die Pixel- und Vertexshadereinheiten nur in den seltensten Fällen vollständig ausgelastet werden: manchmal müssen mehr Vertexdaten berechnet werden, manchmal mehr Fragmentdaten. Bei der *Unified-Shader*-Architektur können alle Shadereinheiten prinzipiell dieselben Operationen ausführen. Ein so genannter *Dispatcher* kümmert sich um die Aufteilung und Zuweisung der Prozessoraufgaben. Er entscheidet, welche Shadereinheit zu welchem Zeitpunkt als Vertex-, Pixel- oder als Geometrieshader eingesetzt wird. Diese flexible Aufteilung führt zwangsweise zu einer effizienteren Leistungsausbeute als bei den bisherigen GPU-Architekturen.

Das Shader-Modell 4.0 sieht außerdem noch einen weiteren Datenpfad für die Rechenergebnisse der Shaderprozessoren vor: mittels *Stream Out* lassen sich jetzt nicht nur Pixel sondern auch Geometriedaten und Koordinaten in den Speicher schreiben (*Render to Vertex Buffer*). Dies ermöglicht Multi-Pass-Programme mit den Vertex- und Geometrieshadern, wie sie bisher den Pixelshadern vorbehalten waren.

Eine weitere wichtige Neuerung ist die Einführung von ganzzahligen Datentypen (Integer). Diese können sowohl zur Berechnung als auch in Vertexpuffern, Texturen, Stream-Outputs und Renderzielen verwendet werden. Integerwerte werden beispielsweise bei der Adressberechnung und bei indexbasierten Feldoperationen benötigt. Neben den Ganzzahlen werden auch bitweise Operationen (Shift, `AND`, `OR`, etc.) und die zugehörige Typumwandlung (`int` ↔ `float`) unterstützt.

Ferner gab es auch einige kleine Verbesserungen wie die gesteigerte Anzahl verschiedener Registertypen, die Erhöhung der handhabbaren Texturgröße, die Verdopplung der gleichzeitig möglichen Renderziele (MRT) oder die Zunahme der maximalen Anzahl Befehle in einem Shaderprogramm von mindestens 512 auf über 65536.

4.1.4 Genauigkeit und Fließkommaformate

Moderne Grafikkarten bewerkstelligen alle ihre Berechnungen neben Fixpunkt- auch mittels Fließkommaarithmetik. Während normale Desktop CPUs für gewöhnlich ein-

fach genaue Fließkommazahlen (single precision float) nach IEEE-754 [IEE85] unterstützen, haben die Grafikkartenhersteller unterschiedliche Formate eingeführt. Um zu verstehen, ob eine Anwendung exakte Ergebnisse liefern kann und um Fehler bei der Fließkommaarithmetik zu vermeiden sollte dies bedacht werden. Eine Gleitkommazahl wird im Allgemeinen durch folgenden Ausdruck repräsentiert:

$$Vorzeichen \times 1.Mantisse \times 2^{(Exponent-Bias)}.$$

Der Bias wird verwendet, da das Zweierkomplement, das normalerweise für die Darstellung negativer Zahlen verwendet wird, den Vergleich zwischen Gleitkommazahlen erschwert.

Ein Vorteil dieser dynamischen Darstellung ist der enorme Umfang des verfügbaren Zahlenraums. Aufgrund der breitwertigen Darstellungsform können allerdings nicht alle Zahlen des Gültigkeitsbereichs lückenlos repräsentiert werden. Deswegen muss speziell bei Adressberechnungen mit großen Arrays mit erhöhter Sorgfalt vorgegangen werden. Mit 16 Bit Genauigkeit lassen sich beispielsweise ±2048 Felder lückenlos adressieren. Bei 24 Bit sind es schon ±131.072 und mit 32 Bit gar ±16.777.216 [Buc05]. Sollen beispielsweise Felder eines maximal großen 1D Arrays angesprochen werden, welches in einer 4096×4096 Pixeln großen Textur liegt, so ist dies mit 24 Bit oder geringerer Genauigkeit nicht korrekt möglich. Daher sollte für Adressberechnung grundsätzlich immer Ganzzahlarithmetik verwendet werden.

Obwohl bei Echtzeitgrafik eine geringe Genauigkeit für das Shading generell ausreicht, da hier die exakte Farbe keine Rolle spielt, kann sie numerische Berechnungen in grober Weise beeinträchtigen [Buc05].

Tab. 4.1: Übersicht der verwendeten Gleitkommaformate

Hersteller	Format	Bits	Darstellung	ab Serie
Intel	LONGREAL[1]	64	s52e11	8087[3]
Intel	SHORTREAL[2]	32	s23e8	8087[3]
ATI	FP32[2]	32	s23e8	R5xx (X1xxx)
NVIDIA	FP32[2]	32	s23e8	NV30 (GF5xxx ??)
ATI	FP24	24	s16e7	R4xx (X8xx ??)
NVIDIA	FP16	16	s10e5	NV30 (GF5xxx ??)

[1]Entpricht dem C-Typ `double`.
[2]Entpricht dem C-Typ `float`.
[3]Der Intel 8087 verwendet intern sogar das 80-bittige `extended double` Format, siehe [Pod99, S. 89]. Seit dem 486 ist der Coprozessor fest in die CPU integriert.

FP16 – Half Precision Float Der `half` wie er vom OpenGL ARB und NVIDIA in [Ope04] spezifiziert wird, ist eine 16-bittige Gleitkommazahl, die aus fünf Expo-

nentenbits, einem Vorzeichenbit und zehn Mantissenbits besteht (s10e5). Sie ist nicht Bestandteil der IEEE-754 Norm, aber in deren Revision (IEEE 754r) aufgenommen. Der verwendete Bias einer FP16 Zahl ist $2^{e-1}-1=15$. Die genaueste Darstellung von beispielsweise der Kreiszahl π setzt sich in FP16 nach obiger Formel wie folgt zusammen: $\pi_{16} = s*(1+m)*2^{e-bias}$ wobei $s=(-1)^0$, $e=16$ und $m=\frac{1}{2^1}+\frac{1}{2^4}+\frac{1}{2^7}=0{,}5703125$ ist; also $\pi_{16}=3{,}140625$. In Tabelle 4.2 ist die Kreiszahl π in der genauest möglichen Darstellung im jeweiligen Gleitkommaformat dargestellt. Der von AMD übernommene Grafikchiphersteller ATI unterstützt das FP16-Format erst in seiner im Mai 2007 vorgestellten GPU-Generation (R600).

Tab. 4.2: Vergleich der Genauigkeit von Gleitkommaformaten anhand π

Format	Resultat	Fehler (10 Stellen genau)
reell	3,14159265...	–
FP32	3,1415927	0,0000000460
FP24	3,141571	0,0000216544
FP16	3,140625	0,0009676540

FP24 Dieses 24-Bit Format wurde bislang nur von ATI bis zur 1000er Serie der Radeon Produktfamilie verwendet. Es setzt sich zusammen aus einem Vorzeichenbit, 16 Mantissenbits und sieben Exponentenbits (s16e7). Ab der 1000er Serie benutzt auch ATI das FP32 Format für interne Berechnungen.

FP32 – Single Precision Float Der 32-bittige `float` des FP32 Formates entspricht der einfach genauen Gleitkommazahl nach IEEE-754, wie sie auch in den meisten CPUs verwendet wird. Sie setzt sich aus einem Vorzeichenbit, acht Exponentenbits und 23 Mantissenbits zusammen (s23e8). Der Bias zur Exponentenberechnung ist beim FP32 Format 127.

4.1.5 Speicherbandbreite

Ein wesentlicher Punkt beim Entwickeln von GPU-Anwendungen ist, den Datentransfer von und zur GPU zu berücksichtigen. Dieser wird als *Readback* und *Download* bezeichnet. Eine reine Ausführung der Berechnung auf der CPU benötigt diese zusätzlichen Schritte dagegen nicht. Daher muss Download und Readback stets beachtet werden, wenn Implementierungen auf beiden Systemen verglichen werden [Buc05]. Die behandelten Verfahren zur Merkmalsverfolgung bilden hier keine Ausnahme.

Die in Tabelle 4.3 dargestellten Werte aus heutigen Computersystemen zeigen, dass das Zurücklesen und Hochladen von Daten zur Grafikkarte relativ langsam ist.Mitunter kann es geschickter sein, Berechnungen direkt auf der CPU auszuführen, selbst wenn

Tab. 4.3: Speicherbandbreite
Die Gesamtbandbreite errechnet sich aus den Werten für die Hin- und Rückrichtung. Die GPU-Werte stammen von einer GeForce 6800 (NV40) und einer GeForce 8800 Ultra (G80). Die Download- und Readback-Werte wurden mit GPUBench [Sta] ermittelt.

Komponente	Bandbreite
GPU Speicherinterface (G80)	103,5 GB/s
GPU Speicherinterface (NV40)	22,4 GB/s
PCI-Express Bus (x16)	8 GB/s
CPU Speicherinterface (PC3200)	6,4 GB/s
AGP 8x	4,1 GB/s
GPU-Readback (G80)	1 GB/s
GPU-Download (G80)	0,8 GB/s

diese hierfür deutlich länger braucht als die GPU. Die Einsparung von Download und Readback kann unterm Strich deutlich mehr ausmachen. Daher sollten sich die Datenübertragungen von und zur GPU immer lohnen, also auch mit genügend Berechnungen verbunden sein.

Eine schlechte Readback-Rate kann aber auch noch andere Ursachen haben, wie etwa eine ungünstige Wahl des Texturformats, bei der der Grafiktreiber dann eine Umwandlung während des Transfers vornehmen muss (siehe Abschnitt 5.6). Auch ein nicht optimal programmiertes System-BIOS kann zu geringen Transferleistungen führen.

4.2 Softwareschnittstellen und Shadersprachen

Im Folgenden werden die grundlegenden Softwareschnittstellen, die generell bei der GPU-Programmierung verwendet werden, erläutert.

Auf die programmierbare Pipeline wird über eine systemnahe Abstraktionsschicht, wie Direct3D (Teil von DirectX) oder OpenGL, zugegriffen. Diese Abstraktionsschicht versteckt einerseits die Unterschiede verschiedener Pipeline-Realisierungen und bietet andererseits eine angenehmere Abstraktion auf Seite der Programmierebene. Während das Direct3D-API unter Microsoft Windows Verwendung findet, ist das OpenGL-API ein plattformunabhängiger offener Standard, der vom OpenGL ARB (Architecture Review Board) gepflegt wird. Da neue OpenGL-Versionen relativ spät verabschiedet werden, entwickeln die Hardwarehersteller ständig neue proprietäre Erweiterungen des OpenGL-Standards. Finden diese eine breite Unterstützung, werden sie in einer neuen Revision aufgenommen.

Diese APIs bieten ein geräteunabhängiges Ressourcenmanagement (Allokation, Lebensdauer, Initialisierung, Virtualisierung, etc.) für Texturen, Vertexpuffer und andere Zustände [Bly06]. Die Kommunikation mit der GPU übernimmt ein gerätespezifischer

Treiber. Durch die Einführung der programmierbaren Grafikpipeline kam die Verwaltung der Shaderprogramme zum Aufgabengebiet der APIs hinzu.

Shadersprachen

Da sich die Programmierung der Shadereinheiten mittels Assembler mit zunehmendem Funktionsumfang als unkomfortabel erwiesen hat, wurden C-ähnliche Hochsprachen entwickelt, die eine angenehmere Programmierung der Shadereinheiten ermöglichen. In den letzten fünf Jahren sind zu diesem Zweck *C for graphics* (Cg) von NVIDIA, die *High Level Shading Language* (HLSL) als Schnittstelle in Microsofts Direct3D und die *OpenGL Shading Language* (GLSL), die mit OpenGL 2.0 eingeführt wurde, entstanden.

Cg, HLSL und GLSL abstrahieren dabei die Funktionen der darunterliegenden GPU und bieten dem Programmierer eine C-ähnliche Sprache zum Schreiben der Programme. Cg und HLSL beschreiben dieselbe Sprache, nur, dass HLSL ausschließlich unter DirectX funktioniert, während Cg plattformunabhängig sowohl unter Direct3D als auch unter OpenGL verwendet werden kann [FK03].

Als Einführung und Anleitung zur Cg-Programmiersprache ist das Tutorial von Fernando und Kilgard empfehlenswert [FK03]. Für GLSL ist das *Orange Book* von Rost das Standardwerk [Ros06]. Es existieren für beide Sprachen auch einfache Handbücher im Internet [NVI05a, KBR06].

Neben den Ähnlichkeiten zu C gibt es auch einige entscheidende Abweichungen. Beispielsweise wird das Hochsprachen-Shaderprogramm erst zur Laufzeit in Maschinencode übersetzt und auf die GPU geladen. Die Hochsprache wird also eher wie bei einer virtuelle Maschine interpretiert. Es ist bei Cg und HLSL jedoch auch möglich den Quell-

```
float4 rgb2gray( half2 coords : WPOS, uniform samplerRECT tex0 ) : COLOR
{
    float4 color = float4( 0.0, 0.0, 0.0, 1.0 );
    float3 p = texRECT(tex0, coords).xyz;

    // use luminance values for grayscale calulation
    float3 lum = { 0.212671, 0.715160, 0.072169 };  //luminance weight
    color.xyz = dot(p, lum);
    return color;
}
// PARAM c[1] = { { 1, 0.21264648, 0.71533203, 0.072143555 } };
// TEMP R0;
// TEX R0.xyz, fragment.position, texture[0], RECT;
// DP3 result.color.xyz, R0, c[0].yzww;
// MOV result.color.w, c[0].x;
// END
```

Quellcode 4.1: Cg-Beispielprogramm
Dieses Beispiel zeigt, wie man aus den Farbkomponenten eines Texels die Grauwerte (Helligkeit) erhält. Die kommentierten Zeilen am Ende zeigen das daraus erzeugte Assemblerprogramm.

code vorab in Maschinencode übersetzen zu lassen und diesen dann zur Laufzeit auf die GPU zu laden. Ein weiterer Unterschied ist laut Blythe auch, dass Shaderprogramme keine eigenständigen Anwendungen sind, sondern zusammen mit einem Programm ausgeführt werden, das auf der CPU gestartet wird und die GPU bedient [Bly06]. Das CPU-Programm versorgt dabei das Shaderprogramm durch Parameter, die es über Texturen oder Befüllen der Konstantenregister auf der GPU bereitstellt.

Die grafikzentrierte Natur der Shadersprachen erschwert die Nutzung der Renderpipeline für GPGPU-Programme, da diese vom Grundgedanken her nicht unbedingt etwas mit Grafik zu tun haben müssen. Aus diesem Grund wurden spezielle Programmiersysteme und Frameworks entwickelt, die dem Programmierer diese Arbeit (Initialisierung, Texturerzeugung, etc.) abnehmen. Dieser muss dann nur noch mit Speicher- und Arithmetikoperationen umgehen, um seine Algorithmen zu implementieren. Owens et al. [OLG+07] nennen hierfür verschiedene Beispiele, auf die an dieser Stelle aber nicht näher eingegangen wird.

4.3 GPU als Streamprozessor

Wie schon zu Beginn dieses Kapitels erläutert, eignen sich Streamprozessoren hervorragend zur Verarbeitung von Bilddaten. Im Folgenden wird untersucht, inwieweit sich moderne GPUs als Streamprozessoren und damit zur Bildverarbeitung eignen.

Da die in den GPUs arbeitenden Shaderprozessoren ihre Berechnungen komplett unabhängig voneinander ausführen, werden sie oft als Streamprozessoren angesehen. Diese Begriffswahl ist jedoch etwas pathetisch, da ihnen zum vollständigen Prozessor laut Bertuch [Ber07] ein eigener Befehlszähler fehlt und sie im Prinzip nur aus Rechenwerken (ALUs - arithmetische, logische Einheiten) bestehen, die ihre Befehle und Daten von übergeordneten Einheiten erhalten.

Ein Streamprozessor ist ein Prozessor, der Operationen nicht wie bei einer CPU seriell, sondern parallel auf Gruppen gleichartiger Eingabedaten, den Datenströmen, anwendet. Man unterscheidet im *Stream*-Programmiermodell zwei wesentliche Komponenten, die Datenströme, die als *Streams* bezeichnet werden und die Kernelemente, kurz *Kerne* genannt, die die Operationen auf den Streamelementen ausführen. Die Datenströme können aus einfachen Elementen wie Ganz- oder Gleitkommazahlen oder aus komplexen Elementen wie Dreiecken oder Transformationsmatrizen bestehen [Owe05]. Ein Kern arbeitet auf ganzen Datenströmen mit einem oder mehreren Streams und erzeugt dabei einen oder mehrere Streams als Ausgabe.

Um als Streamprozessor gelten zu können muss eine GPU vor allem zwei Kriterien erfüllen: die Forderung nach der Lokalität der Daten und die Parallelisierbarkeit. Die Lokalität der Daten wird gefordert, damit der Ausgabestrom eines Kerns direkt als Ein-

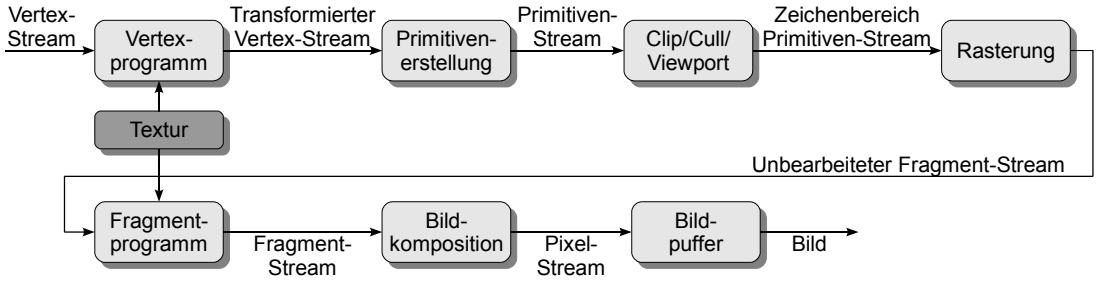


Abb. 4.4: Abbildung der Grafikpipeline auf das Stream-Modell
Wenn alle Daten als Streams und die einzelnen Stufen als Kerne gesehen werden, passt die Grafikpipeline perfekt zum Stream-Modell. Dabei können sowohl programmierbare als auch nicht programmierbare Stufen als Kerne gelten. [Owe05]

gabe für den nächsten Kern genutzt werden kann. Eine externe Speicherung der Daten entfällt somit. Die Forderung nach der Parallelisierbarkeit ergibt sich aus dem Wunsch mehrere (alle) Streamelemente gleichzeitig und ohne Abhängigkeiten bearbeiten zu können. Die zu Beginn dieses Kapitels beschriebene Grafikpipeline erfüllt diese Kriterien aufgrund ihrer pipelineartigen Struktur (Lokalität) und der Unabhängigkeit der verwendeten Datenelemente (Vertices, Fragmente; siehe Abbildung 4.4). Das Stream-Programmiermodell eignet sich daher hervorragend als Grundlage zur Programmierung der GPU für allgemeine Aufgaben.

In GPGPU-Programmen werden verschiedene grundlegende Kernoperationen, wie *map*, *reduce*, *scan*, usw., verwendet. Diese werden im Folgenden kurz erläutert. Für die Implementierungen in Kapitel 5 werden jedoch nur die Operationen *map*, *reduce* sowie *gather & scatter* benötigt. Eine etwas ausführlichere Beschreibung ist bei Owens et al. [OLG+07] zu finden.

Abbilden (map) Map wendet eine bestimmte Funktion auf jedes Element des Datenstroms an. Beispielsweise kann damit ein Datenstrom mit Werten aus dem Bereich von `[0,0..1,0)` in Werte mit dem Bereich `[0..255]` umgewandelt werden, indem jedes Element mit 256 multipliziert und abgerundet wird.

Reduzieren (reduce) Unter Reduzierung versteht man die Erzeugung eines kleinen Datenstroms oder gar eines einzelnen Wertes aus einem großen Eingabestrom. Ein Standardbeispiel für so eine Operation ist eine Maximumsberechnung oder die Summenbildung über alle Elemente eines Datenstroms.

Auf der GPU kann die Reduzierung durch alternatives Lesen und Schreiben (Rendern) auf zwei Texturen erfolgen. In jedem Durchlauf wird die Ausgabegröße, der Berechnungsbereich, halbiert. Um die Folgewerte zu erzeugen, werden vom Fragmentprogramm jeweils zwei Werte aus den korrespondierenden Stellen jeder Hälfte des Ergebnisses des vorhergehenden Durchlaufs gelesen und mit dem

Reduzierungsoperator (z.B. `add` oder `max`) verknüpft. Göddeke bietet auf seiner Webseite ein Tutorial an, dass sich nur mit der Datenreduktion auf der GPU befasst [Göd06].

Scatter und Gather Die fundamentalen Speicheroperationen sind Lesen und Schreiben. Greifen diese Operationen indirekt auf den Speicher zu, nennt man sie *gather* (engl. auflesen, einsammeln) und *scatter* (engl. ver-, zerstreuen). *Gather* entspricht dem C-Code `a=b[c]`, wobei der Wert des Feldes `b` mit Index `c` der Variablen `a` zugewiesen wird. *Scatter* ist genau die umgekehrte Operation zu *gather* und lässt sich in C durch `b[c]=a` ausdrücken.

Wie bereits in Unterabschnitt 4.1.1 erwähnt, ist die GPU Implementierung von *gather* einfach ein Zugriff auf ein Texturelement, wohingegen sich *scatter* auf der GPU so nicht realisieren lässt, da die Ausgabe der Fragmentprogramme an eine feste Zieladresse gebunden ist. Neben der dort bereits beschriebenen Methode, die Vertexprozessoren für *scatter*-Operationen zu nutzen, behelfen sich GPGPU-Programmierer mit weiteren Tricks um diese Operation zu ermöglichen. Eine Möglichkeit, ist die von Buck [Buc05] geschilderte Umschreibung des Scatterproblems in eines bestehend aus *gather*-Operationen. Eine andere wäre, die Daten während des normalen Renderdurchlaufs mit zusätzlichen Adressen zu versehen und anschließend die Daten nach den Adressen zu sortieren.

Präfixsumme (Scan) Ein einfacher und verbreiteter Baustein für parallele Algorithmen ist die *Präfixsummenbildung*, die auch unter dem Namen *Scan* bekannt ist. Dabei wird in einer Folge von Elementen, für jedes Element die Summe der vorhergehenden Elemente berechnet. Owens(u.a) [OLG+07] verweisen auf Quellen, die beschreiben, wie *Scan* auch auf der GPU in $O(n)$ Durchläufen ausgeführt werden kann.

Datenstromfilterung Viele Algorithmen müssen aus einem Datenstrom eine kleine Menge von Elementen auswählen und den Rest verwerfen, wobei deren Position im Strom und deren Anzahl in der Regel nicht bekannt ist. Als Beispiel wäre die Auswertung von Extremwerten bei einem Eckendetektor zu nennen.

Horn hat in [Hor05] eine Technik namens *stream compaction* vorgestellt, die eine Datenstromfilterung auf der GPU in $O(\log n)$ Durchgängen bewältigen soll.

Sortieren Das Sortieren ist ein klassisches algorithmisches Problem, für das es auf der CPU verschiedenste Lösungsimplementierungen gibt. Die meisten dieser Algorithmen eignen sich nicht für eine Implementierung auf der GPU, weil sie einerseits nicht datenunabhängig sind und andererseits ausgiebigen Gebrauch der Scatter-Operation machen. Effiziente GPU-Implementierungen sollten daher laut

Owens et al. nicht an den Eingabedaten hängen und die Scatteroperation nicht benötigen [OLG+07].

Wie bei Owens et al. aufgeführt ist, basieren die meisten GPU-Implementierungen auf Sortiernetzwerken. In diesen ist die Anzahl der Durchläufe fest und unabhängig von den Eingabedaten, was eine Implementierung ohne datenabhängige Verzweigungen ermöglicht. Da zusätzlich die Kommunikationsmuster starr sind, kann man die Sortierung eher mit *gather*- als mit *scatter*-Operationen umsetzen.

Govindaraju et al. [GRHM05] bieten mit *GPUsort* eine äußerst cacheeffiziente Implementierung eines verbesserten bitonischen Suchnetzwerks, das deutlich kürzere Laufzeiten als ein 3,4 GHz Pentium 4 mit Quicksort erreicht. Greß und Zachmann [GZ06] gelingt es mit *GPU-ABiSort* sogar $O(n \log n)$ als theoretische Zeitschranke zu erzielen.

Suchen Die Idee hinter streambasierten Suchalgorithmen ist es, den Durchsatz der Suche zu erhöhen, indem viele Suchoperationen parallel angestoßen werden.

Beispielsweise bei einer Binärsuche, in der das zu suchende Element zunächst mit dem mittleren verglichen wird, und dann abhängig vom Resultat in den Teilhälften rekursiv weitergesucht wird, kann das Finden eines einzelnen Elements nicht parallelisiert werden, jedoch lässt es sich nach mehreren Elementen parallel suchen [OLG+07].

4.4 Vorgehensweise bei der GPU-Programmierung

Im Folgenden werden die einzelnen Schritte, die zur Durchführung von Berechnungen auf der GPU benötigt werden, beschrieben. Dafür wird auf die in 4.3 eingeführte Terminologie des Stream-Programmiermodells zurückgegriffen.

Zur Berechnung von allgemeinen Aufgaben und grafikbasierten Aufgaben im Speziellen, wird dem Fragmentprozessor gegenüber dem Vertexprozessor der Vorzug gegeben. Dies liegt daran, dass dieser direkt auf Datenfeldern (Texturen) und nicht auf Geometriedaten arbeitet. Zudem sind auf gängigen Grafikkarten mehr Fragmentprozessoren als Vertexprozessoren verbaut, was einen höheren Durchsatz verspricht

Um einen Algorithmus auf der GPU umzusetzen, muss dieser zunächst in unabhängig voneinander arbeitende Teile, die parallel auf die Daten zugreifen können, zerlegt werden. Diese Teile können dann als eigenständige Kerne betrachtet und als einzelne Fragmentprogramme implementiert werden. Die Ein- und Ausgaben dieser Kernprogramme sind Datenfelder, die in den Texturen im Grafikspeicher abgelegt sind.

Der weitere Ablauf lässt sich folgendermaßen beschreiben:

1. Um einen Kern auszuführen, muss diesem der Bereich, auf dem die Berechnungen ausgeführt werden sollen, zugewiesen werden. Damit der Fragmentprozessor genau auf den Eingabedaten arbeitet und diese nicht interpoliert werden, erzeugt man eine `1:1`-Abbildung zwischen Texeln und Pixeln (Fragmenten). Dies geschieht, indem man ein Rechteck zeichnet, das parallel zur Bildebene ausgerichtet und genau so groß ist, dass es den rechteckigen Bereich abdeckt, der der verlangten Ausgabegröße entspricht.
2. Jedes dieser Fragmente wird dann von demselben Fragmentprogramm (Kern) bearbeitet. Das Fragmentprogramm kann dabei über Texturzugriffe von frei wählbaren Speicherstellen lesen, aber nur in die Speicherstelle schreiben, die durch das jeweilige Fragment im jeweiligen Renderziel (Framebuffer, Textur) vorgegeben ist. Der Bereich für die Berechnungen wird dem Fragmentprogramm für jede Eingabetextur (Stream) über Texturkoordinaten mitgeteilt, die dann für die jeweiligen Fragmente interpoliert werden.
3. Das Fragmentprogramm liefert einen Wert (Vektor) pro Fragment als Ausgabe. Dieser Wert kann für weitere Berechnungen verwendet werden, indem man ihn in einer Textur speichert (*Render To texture*, siehe Abschnitt 4.5).

Komplexe Anwendungen benötigen mehrere oder sogar etliche Pipelinedurchläufe (*multipass*). Mehrere Durchläufe ermöglichen eine nahezu beliebige Komplexität für eine Implementierung, da die Einschränkungen (maximale Anzahl Befehle, Ausgaben, beschränkte Kontrollmöglichkeit) wegfallen [OLG+07].

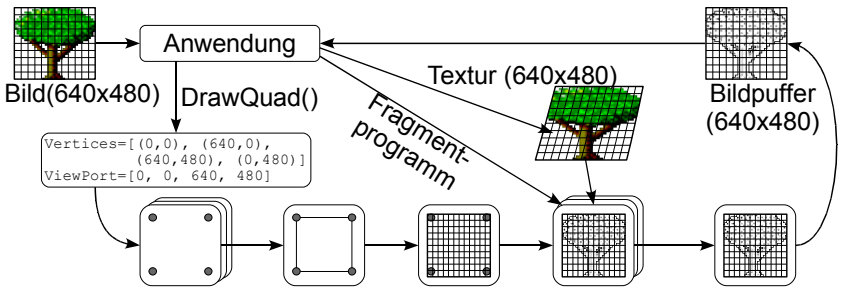


Abb. 4.5: Ablauf eines GPU-Programms
Es wird ein Rechteck (Quad) erzeugt, das dieselbe Größe wie das Eingangsbild hat. Um eine 1:1 Abbildung zwischen Texeln und Pixeln herzustellen wird der Viewport auch auf diese Dimensionen gesetzt. Die Rastereinheit erzeugt dann einen Bereich der der Bildgröße entspricht. Das Fragmentprogramm kann dann auf die Texturwerte zugreifen und z.B. eine Kantendetektion ausführen. Dabei wird die automatisch übermittelte Fensterkoordinate genutzt, um die jeweilige Texturposition zu bestimmen.

4.5 Techniken für die GPU-Bildanalyse

Zur Implementierung von Verfahren auf der GPU gibt es einige Techniken, die diesen Vorgang erheblich erleichtern. Im folgenden Abschnitt werden grundlegende Methoden und besondere Eigenschaften der Grafikschnittstelle vorgestellt, ohne die eine effektive Bildanalyse auf der GPU nicht möglich wäre.

Fragmentprogramme Fragmentprogramme sind die natürlichste Stelle, um Algorithmen für die Bildverarbeitung oder das Maschinelle Sehen auszuführen, da sie direkt auf jedem einzelnen Pixel arbeiten.

Render To Texture (RTT) Dies ist eine Methode, die es erlaubt, direkt in Texturen zu rendern. Dabei wird der für das Zeichnen zuständige Pixelpuffer direkt als Textur gebunden. Die RTT-Semantik ist vergleichbar mit dem klassischen Zeichnen in den Bildpuffer mit anschließendem Aufruf von `glCopyTexSubImage`. Der Vorteil der neuen Methode ist, dass man dasselbe Ergebnis erhält, dabei aber auf das Kopieren der Daten verzichten kann. Dies spart gerade bei Multipass-Verfahren Bandbreite und wirkt dem langsamen Bussystem entgegen.

Unter Windows wird RTT in OpenGL primär durch die *PBuffers* unterstützt. Mittlerweile existiert mit der Framebuffer-Objekt-Erweiterung hierfür jedoch eine bessere und plattformunabhängigere Methode.

Render To Texture ist für GPGPU Anwendungen von essentieller Bedeutung: Mit diesem Verfahren ist es möglich, berechnete Werte im Texturspeicher der Grafikkarte abzulegen. Dadurch kann im nächsten Bearbeitungsschritt auf diese Daten zugegriffen werden. Erst mit dieser Methode sind Multipass-Verfahren überhaupt sinnvoll umsetzbar.

Rechteckige Texturen Schon auf einigen frühen GPU Serien wird die `ARB_texture_rectangle` Erweiterung unterstützt, die schließlich in die Spezifikation von OpenGL 2.0 aufgenommen wurde. Mit dieser Erweiterung lassen sich 2D-Texturen benutzen, deren Breite und Höhe keine Zweierpotenz ist, wie dies bei `GL_TEXTURE_2D` der Fall ist. Dies ist nützlich, um etwa Videobilder zu speichern, deren Dimensionen keine Zweierpotenzen sind. Das Umskalieren des Bildes in Zweierpotenzen kann somit entfallen, was Speicher spart und eventuelle Artefakte vom Umrechnen vermeidet. Ein weiterer Vorteil ist, dass die Texturkoordinaten nicht mehr normalisiert angegeben werden, sondern in den Intervallen `[0, Breite]` und `[0, Höhe]`. `GL_TEXTURE_RECTANGLE_ARB`-Texturen unterstützen jedoch kein Mipmapping und beschränken die *Wrap*-Modi auf `CLAMP*`-Werte.

Bilineare Filterung Heutige Grafikhardware kann Texturen ohne zusätzlichen Auf-

wand bilinear filtern. Dazu muss man diese lediglich mit `GL_LINEAR` anstatt mit `GL_NEAREST` initialisieren. Dies kann genutzt werden, um eigene Interpolationsschritte einzusparen. Vorsicht ist jedoch geboten, wenn keine Farbwerte sondern Daten in den Farbkanälen der Texel gespeichert sind, da diese nicht getrennt voneinander interpoliert werden und es somit zu Wertverfälschungen kommt. Auch werden nicht auf allen GPUs alle Gleitkommaformate bei der Filterung unterstützt.

PBuffers PBuffers sind *Offscreen*-Zeichenpuffer für OpenGL-Renderer [GPGb]. Sie können außerdem, im Gegensatz zum Standard-Framebuffer, Gleitkommawerte speichern. Aufgrund ihrer Komplexität und Ineffizienz verwendet man heute stattdessen meist die einfacheren und flexibleren Framebuffer-Objekte.

Framebuffer-Objekte (FBOs) Die `GL_EXT_framebuffer_object` Erweiterung stellt eine Schnittstelle bereit, die es ermöglicht in Ziele zu zeichnen, die nicht an Fenster gebunden sind [Ope05]. Im Gegensatz zu den PBuffern sind bei Framebuffer-Objekten keine plattformabhängigen und langsamen WGL/GLX-Befehlen nötig. Auch die zeitraubenden Umschaltungen des Render-Kontexts entfallen.

Diese neuen Renderziele sind so genannte Framebuffer-Objekte. Sie selbst stellen aber noch keine Puffer bereit, daher muss man Zeichenpuffer und Texturen an sie binden. Dies ist bis jetzt für alle bisherigen logischen Standard-GL Puffertypen möglich: Farb-, Tiefen- und Schablonenpuffer. Es lassen sich, je nach GPU, bis zu 16 gleichformatige Texturen an ein FBO binden. Die Umschaltung der Renderziele erfolgt mit `glDrawBuffers`.

Nachdem man ein Framebuffer-Objekt gebunden hat, kann man direkt in die verknüpften Texturen mittels Fragmentprogrammen zeichnen. Die Texturdaten lassen sich dann in weiteren Fragmentprogrammen verwenden oder durch `glReadPixels` auslesen.

Durch die Einführung von Bildpuffern als Renderziel wird das Zeichnen in nicht sichtbare Bereiche (*offscreen rendering*) ermöglicht.

Tips & Tricks beim Umgang mit Framebuffer-Objekten:

- Man sollte `glViewport` nutzen, um sicherzustellen, dass der gezeichnete Bereich auch nicht größer als im FBO ist.
- Die Texturen bleiben solange dem FBO zugewiesen, bis sie manuell entfernt werden, unabhängig davon, ob das FBO gebunden ist oder nicht.
- Das Umschalten zwischen den Render-Targets geht schneller als das Hin- und Herschalten zwischen FBOs (was aber immer noch deutlich schneller als der Kontextwechsel mit `wglMakeCurrent` ist).

- Für jede Texturgröße/Format sollte ein eigenes FBO angelegt werden. Dies ist zwar nicht direkt Teil der Spezifikation, ist aber durch die Beschränkungen der Treiber gegeben [Ope05].

Pingpong-Verfahren Das *Pingpong-Verfahren* ist eine Technik, die oft mit Framebuffer-Objekten und RTT verwendet wird. Da man in einem Fragmentprogramm nicht gleichzeitig lesend und schreibend auf denselben Puffer zugreifen kann, verwendet man hier ein Pufferpaar, auf das abwechselnd zugegriffen wird. Dies kann im Bereich der Merkmalsverfolgung beispielsweise sinnvoll zur Ausführung einer zweistufigen Gaußfaltung eingesetzt werden. Dabei werden die Werte im ersten Durchlauf in x- und beim zweiten in y-Richtung geglättet.

Multiple Render Targets (MRTs) Mit OpenGL Version 2.0 ist die nützliche Möglichkeit in mehrere Farb-Bildpuffer gleichzeitig zu zeichnen hinzugekommen. Mit der GPU-Generation ab ATI Radeon 9800 (R350) und NVIDIA GeForce 6 (NV40) lassen sich hiermit Daten in bis zu vier Zielpuffer gleichzeitig schreiben. Dies ist besonders dann nützlich, wenn auf denselben Eingangsdaten verschiedene oder sogar ähnliche Berechnungen durchgeführt werden müssen, wie beispielsweise zur gleichzeitigen Berechnung der Größe und der Orientierung der Gradienten eines Bildes. Durch den Einsatz von *Multiple Render Targets* lassen sich komplexe Multipass-Algorithmen oftmals leichter umsetzen.

4.6 Zusammenfassung

In diesem Kapitel wurde ein Einblick in die Funktionsweise der parallelen Architektur programmierbarer Grafikhardware gegeben. Es wurde auf Besonderheiten hingewiesen, die bei der Programmierung beachtet werden sollten, wie beispielsweise die unterschiedliche Genauigkeit auf manchen GPUs und der Datentransfer von und zur GPU. Ferner wurden verschiedene Techniken vorgestellt, die eine sinnvolle Programmierung mit der Grafikpipeline überhaupt erst ermöglichen. Des Weiteren wurde gezeigt, dass sich die parallele Architektur heutiger GPUs hervorragend auf das für die Bildverarbeitung vorgeschlagene Modell eines Streamprozessors abbilden lässt. Speziell die auf Texturdaten arbeitenden Fragmentshader sind ideal zur Implementierung von Filteroperationen geeignet, die wesentlicher Bestandteil der Verfahren aus Kapitel 3 sind.

5 Umsetzung

In den Verfahren zur natürlichen Merkmalsverfolgung im Maschinellen Sehen oder der Erweiterten Realität müssen eine Unmenge an Bilddaten verarbeitet werden. Da Grafikkarten an sich schon für die Verarbeitung von Bilddaten ausgelegt sind, soll untersucht werden, inwiefern sich diese Methoden auf ihren parallelen Architekturen umsetzen lassen.

In diesem Kapitel werden GPU-Implementierungen solcher Verfahren betrachtet und ihre Vor- und Nachteile gegenüber CPU-Implementierungen abgewogen.

Die größte Hürde bei der Umsetzung auf die GPU bestand darin, die ungewohnte Architektur der programmierbaren GPUs und die überaus trickreichen Methoden ihrer Programmierung zu verstehen. Hierzu wurden herstellerspezifische Erweiterungen benötigt, die teilweise noch nicht Teil des OpenGL-Standards sind und das Repertoire der normalen Grafikprogrammierung weit übersteigen. Deswegen musste sehr viel experimentiert werden und es hat einige Zeit gedauert bis auf ein funktionierendes Basissystem zurückgegriffen werden konnte, das die wesentlichsten Funktionen für eine erfolgreiche Umsetzung bereitstellte. Ferner erforderte das parallele Programmierkonzept zuweilen einiges Umdenken. Man kann daher den Aufwand einer effizienten GPU-Implementierung mindestens mit dem einer handoptimierten CPU-Version vergleichen. In Abschnitt 5.5 wird auf einige Hürden, die es bei der Umsetzung zu bewältigen galt, hingewiesen. Die Optimierungsstrategien (Abschnitt 5.6) spiegeln einen Teil der Erfahrungen wieder, die bei den Implementierungen gesammelt wurden.

Aus eben genannten Gründen konnte nur ein Teil der vorgestellten Verfahren selbst implementiert werden. Der Autor griff daher für weitere Untersuchungen auch auf bereits existierende Anwendungen aus diesem Themenbereich zurück. Diese sind: der KLT-Tracker von Sinha (Abschnitt 5.2), der Feature-Tracker des OpenVIDIA Projekts (Abschnitt 5.3) und der SIFT-Merkmalsextraktor von Sinha (Unterabschnitt 5.4.2). Da die Quellen des OpenVIDIA-Trackers frei verfügbar sind, dieser aber unter Linux entwickelt wurde, mussten erst die wesendlichen Teile nach Windows portiert[1] werden. Als Eigenimplementierungen wird eine Umsetzung des Harris-Eckendetektors (Abschnitt 5.1) und der SIFT-Merkmalsextraktor (Unterabschnitt 5.4.1) betrachtet.

[1] Es existiert zwar ein Demoprogramm für Windows, dieses verwendet allerdings eine fertig compilierte Bibliothek, die nicht für die geplanten Zwecke verwendet werden konnte.

Letzterer konnte jedoch im Rahmen dieser Arbeit nicht vollständig fertiggestellt werden. Dies lag einerseits an der hohen Komplexität des SIFT-Algorithmus und andererseits an technischer Schwierigkeiten, die in Abschnitt 5.4 erläutert werden.

Diese Implementierungen müssen ihre Leistungsfähigkeit im Vergleich mit CPU-Versionen unter Beweis stellen. Hierfür wurden mit allen Implementierungen verschiedene Leistungsmessungen durchgeführt, deren Ergebnisse in den einzelnen Abschnitten aufgeführt werden. Das verwendete Testsystem ist im Anhang (S. 82) genauer spezifiziert.

Da keine freien Implementierungen für SIFT auf der GPU verfügbaren sind, und die eigene Implementierung des Autors nicht voll funktionsfähig ist, konnte hierzu keine Leistungsmessung durchgeführt werden. Die Einschätzung der Leistung beschränkt sich daher auf die Auswertung der bei Sinha [SFPG06] und Heymann [Hey05] angegebenen Daten sowie auf die gesammelten Erfahrungen der eigenen Implementierung.

Zur Programmierung der GPUs wurde bei allen Implementierungen sowohl bei den Eigenentwicklungen als auch bei den bestehenden Implementierungen, auf die Kombination aus C++, OpenGL und Cg (siehe Abschnitt 4.2) gesetzt. Damit wird die größte Plattformunabhängigkeit erreicht und sichergestellt, dass die Anwendungen prinzipiell sowohl unter Windows als auch unter Linux lauffähig sind. Für das Erzeugen der OpenGL-Kontexte sowie für die Benutzerinteraktion wurde GLUT[2] verwendet.

5.1 GPU-Harris-Corner Implementierung

Die GPU-Implementierung des Eckendetektors von Harris und Stephens aus Unterabschnitt 3.3.4 besteht im Wesentlichen aus vier Fragmentprogrammen, die den einzelnen Schritten des Ablaufs entsprechen. Das jeweilige Videobild wird als Textur in den Grafikspeicher geladen und dort, wie in Abbildung 5.1 dargestellt, bearbeitet. Die gefundenen Eckpunkte werden anschließend auf die CPU übertragen und ausgewertet. Hierbei werden zwei alternative Lösungswege untersucht und gezeigt, dass man durch Reduktion der zurückzulesenden Daten in diesem Fall bis zu 35 Prozent Leistungszuwachs erreichen kann.

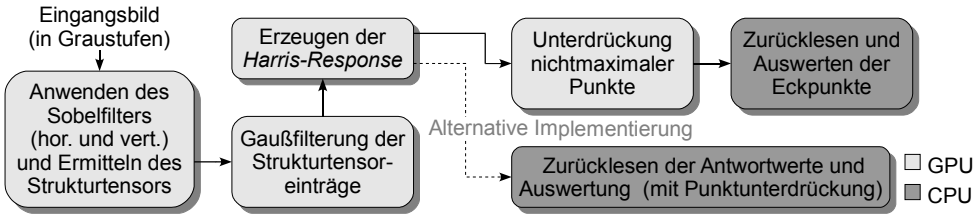


Abb. 5.1: Ablauf des Harris-Eckendetektors auf der GPU

[2] OpenGL Utility Toolkit

5.1.1 Implementierungsdetails

Zuerst wird das Eingangsbild in Graustufen auf die Grafikkarte geladen. Dazu definiert man die Textur in OpenGL explizit mit dem Wert `GL_LUMINANCE`. Dann werden die Grauwerte direkt vom Grafikkartentreiber entsprechend den Helligkeitswerten vor der Übertragung zur Grafikkarte erzeugt. Dadurch können bei der Übertragung zur GPU zwei Drittel der Daten eingespart werden. Wird das farbige Bild jedoch zur Anzeige benötigt, kann man die Grauwerte auch durch ein einfaches Fragmentprogramm wie beispielsweise Quellcode 4.1 (S. 37) erzeugen. Hierfür müssen dann die kompletten `RGB`-Daten zur GPU übertragen werden. Um stets auf den exakten Werten zu arbeiten, wird als Texturformat `GL_NEAREST` und nicht `GL_LINEAR` gewählt. Denn bilineare Filterung kann laut Warden [War05], aufgrund interner Optimierungen der Sample-Positionen der Textur, zu einem weichzeichnerischen Effekt führen. Diese Interpolation wäre aber, gerade bei der Verwendung von anderen Daten als Farben in den einzelnen Farbkanälen, unerwünscht, da diese verfälscht würden.

Anwenden des Sobelfilters Im nächsten Schritt wird auf dem Graustufenbild ein horizontaler und vertikaler Sobelfilter angewendet, um die Gradienten und daraus den Strukturtensor zu bestimmen. Dazu wird der *Viewport* so gesetzt, dass die Texel der Eingabetextur 1:1 auf die Bildschirmpunkte abgebildet werden. Um die berechneten Ergebnisse im nächsten Schritt wiederverwenden zu können, wird, wie in Abschnitt 4.5 beschrieben, mittels eines Framebuffer-Objekts abwechselnd in zwei Texturen gezeichnet (*Pingpong*-Verfahren). Bei der Bearbeitung entsprechen die Texturwerte dann den einzelnen Fragmenten. Die Sobelfilterung wird durch ein Fragmentprogramm (siehe Quellcode 5.1) umgesetzt, das die aus Unterabschnitt 3.3.1 bekannte Faltungsmaske auf die Bildpunkte anwendet. Um Texturzugriffe zu sparen, werden die beiden Komponenten des Sobelfilters zusammen in einem Programm berechnet. Die Einträge der Strukturtensormatrix werden dann in den einzelnen `RGB`-Farbkanälen des aktuellen Fragments gespeichert.

Gaußfilterung Das Ergebnis wird anschließend, wie in Unterabschnitt 3.3.4 beschrieben, durch einen 3×3 Gaußfilter geglättet. Wegen der vektorbasierten Architektur der GPU kann dieser auf alle drei Farbkanäle simultan angewendet werden. Durch das Aufteilen der Gaußfaltung in x- und y-Richtung, die in zwei Zeichendurchgängen ausgeführt werden, könnenn pro Fragment drei Texturzugriffe und fünf Additionen eingespart werden. Es wird lediglich mit einer zusätzlichen Multiplikation und einem Kopierbefehl (`MOV`) dafür bezahlt. Unterm Strich werden so fast ein Drittel der nötigen Befehle eingespart. Dies wirkt sich in der Gesamtlaufzeit jedoch nur minimal aus,

```
// Sobel values (first derivatives) are stored in rgb components:
//   red = hor*hor, green = vert*vert, blue = hor*vert
void main( half2 coords : WPOS, uniform samplerRECT tex0,
           out half4 OUT : COLOR )
{
    half tl = texRECT( tex0, coords + half2(-1.0,  1.0) ).x;
    ...
    half br = texRECT( tex0, coords + half2( 1.0, -1.0) ).x;
    half sobelV = bl - tl + 2*(bc - tc) + br - tr;
    half sobelH = tr - tl + 2*(cr - cl) + br - bl;
    OUT.x = sobelH * sobelH;
    OUT.y = sobelV * sobelV;
    OUT.z = sobelH * sobelV;
    OUT.w = 1;
}
```

Quellcode 5.1: Cg-Programm für den horizontalen und vertikalen Sobel-Filter

da es andere Programmteile, wie die Datenübertragung gibt, die deutlich mehr Zeit benötigen.

Erzeugen der Harris-Response Der Antwortwert des Harris-Eckendetektors aus Gleichung 3.2 (S. 19) wird im nächsten Fragmentprogramm aus den geglätteten Einträgen des Strukturtensors berechnet.

Unterdrückung nichtmaximaler Punkte Um alle schwachen Antwortwerte zu entfernen, müssen diese zunächst eine Schwellwertabfrage passieren und sich als lokale Maximalwerte behaupten. Quellcode 5.2 zeigt einen Ausschnitt des hierfür verwendeten Fragmentprogramms. Darin werden auch die subpixel-genauen Koordinaten der Eckpunkte berechnet und in der y- und z-Komponente gespeichert.

Zurücklesen und Auswerten der Eckpunkte In diesem Schritt werden die Ergebniswerte des letzten Fragmentprogramms als `RGB`-Daten in den Hauptspeicher gelesen und ausgewertet. Dabei werden die Eckpunkte mit ihren Koordinaten und dem zugehörigen Antwortwert in einem Vektor gespeichert.

Alternative Implementierung Ein alternativer Lösungsweg für diesen Algorithmus ist, die Daten direkt nach der Berechnung des Harris-Antwortwertes zurückzulesen, da diese dann lediglich aus einem Wert pro Bildpunkt bestehen (Harris-Antwort) und nicht aus drei wie bisher (Harris-Antwort und subpixelgenaue x sowie y Koordinate). Dies bedeutet allerdings, dass die Unterdrückung der nichtmaximalen Punkte und die subpixelgenaue Koordinatenbestimmung auf der CPU durchgeführt werden muss. Da sich durch die Reduzierung der Farbkanäle auch die Indexoperationen beim Zugriff auf dem Ergebnispuffer deutlich vereinfachen, ist die Berechnung auf der CPU sogar

```
void main( half2 coords : WPOS, uniform samplerRECT tex0,
           uniform half thresh, out half4 OUT : COLOR )
{
    OUT.xyz = 0;
    OUT.w   = 1;
    half3 x = texRECT( tex0, coords ).xyz;
    if ( x.x < thresh || x.x == 0 ) return;
    ...
    // Use only the first component (harris response value)
    if ( x.x>tl && x.x>tc && x.x>tr && x.x>cl && x.x>cr &&
         x.x>bl && x.x>bc && x.x>br )
    {
        // Calculates the subpixel position from the local neighbourhood
        x.y = (cl - cr) / (2*(cr - 2*x.x + cl));
        x.z = (tc - bc) / (2*(bc - 2*x.x + tc));
        OUT.xyz = x;
    }
}
```

Quellcode 5.2: Ausschnitt aus dem Cg-Programm der 'non-maximum suppression'

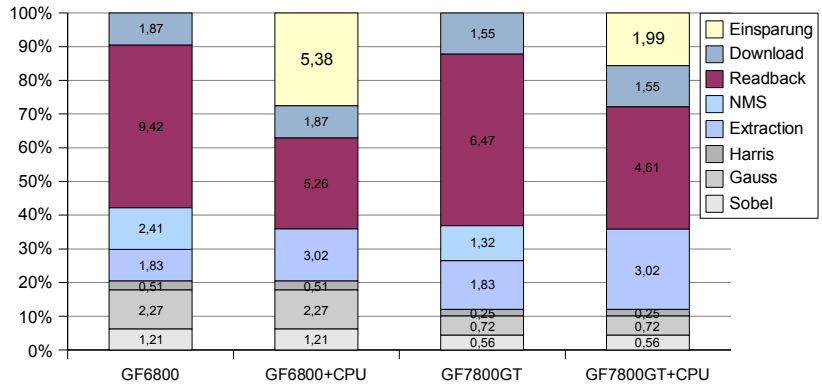


Abb. 5.2: Komponentenanalyse der Harris-Implementierung
Die Grafik zeigt, wie hoch das Einsparpotential ist, wenn man die Non-Maximum-Suppression in den Extraction-Vorgang auf der CPU integriert und deswegen `GL_RED` anstatt `GL_RGB`-Daten verwenden kann. Dadurch verringert sich der Readback drastisch. Die Werte in den Feldern sind die jeweiligen Ausführungszeiten in ms bei einem 640×480 Video.

geringfügig schneller als auf der GPU. Die größte Einsparung erzielt man jedoch durch das reduzierte Datenvolumen beim Zurücklesen, das immer noch die meiste Zeit bei der Programmausführung in Anspruch nimmt. Diese könnte allerdings auch erreicht werden, indem auf die subpixelgenaue Koordinatenbestimmung verzichtet wird.

Auf einer GeForce 6800 AGP wurde bei 1024×768 eine Leistungszunahme von 35 Prozent gemessen. Da die Formatumwandlung durch den Grafiktreiber jedoch Zeit kostet, verkürzt sich die Readbackdauer nicht wie erwartet auf ein Drittel (siehe Abbildung 5.2). Auch wenn die Einsparung nicht auf allen GPUs gleich groß ist, ist sie dennoch beträchtlich.

5.1.2 Ergebnisse

Der Eckendetektor von Harris und Stephens lässt sich ideal auf der GPU abbilden, da er sich so aufteilen lässt, dass jeder Schritt parallel auf den Eingabedaten operieren kann. Die Schwachstelle der Umsetzung ist das Zurücklesen der erkannten Eckpunkte in den Hauptspeicher. Dies ist jedoch für die Extraktion der Eckpunkte zwingend erforderlich. Eine weitere Bearbeitung auf den kompletten Daten wäre höchstgradig ineffizient. Meist werten weiterführende Verfahren diese Daten mittels pointerlastigen Vergleichen aus, wie beispielsweise zur Merkmalsverfolgung oder Posenbestimmung. Diese lassen sich in der Regel sowieso auf der CPU effizienter implementieren.

Um den Datentransfer so klein wie möglich zu halten, kann man neben der bereits besprochenen Integration der Unterdrückung nichtmaximaler Punkte in den Extraktionsvorgang auf der CPU, das `half`-Format und 16-bittige Texturen anstatt `float` und 32Bit-Texturen verwenden. Dies führt zwar, aufgrund der geringeren Genauigkeit, zu ungefähr drei Prozent weniger Merkmalspunkten. Dafür ist allerdings ein Geschwindigkeitszuwachs von circa 60 Prozent zu verzeichnen.

Auflösung	GeForce 7900 GT	GeForce 7800 GT	GeForce 6800	Athlon64 3800+
320x240	338	300	244	17
512x384	151	127	99	9
640x480	98	80	62	8
800x600	62	55	39	6
1024x768	36	33	23	4

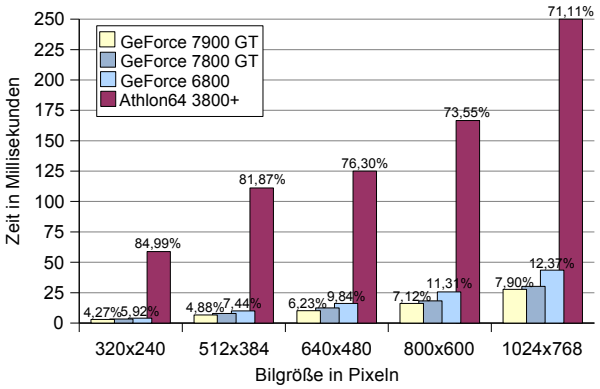


Abb. 5.3: Performanceauswertung der Harris-Implementierung
Die Tabelle gibt die Anzahl Bilder pro Sekunde bei verschiedenen Auflösungen wieder. Bei den verwendeten Szenen wurden zwischen 1000 und 3000 Merkmale erkannt. Die Grafik zeigt die mittlere Zeit eines Durchlaufs. Für echtzeitnahe Geschwindigkeit sind höchstens 50ms (entspricht mindestens 20fps) erforderlich. Die Prozentwerte geben das jeweilige Verhältnis der Implementierungen zueinander wieder.

Trotz der vorgenommenen Optimierungen nimmt der *Readback*-Vorgang immer noch circa 60 Prozent der gesamten Laufzeit in Anspruch. Selbst mit diesem Manko erreicht die GPU-Implementierung auf einer NVIDIA GeForce 7900 GT die neun- bis neunzehnfache Leistung der CPU-Implementierung[3] auf einem AMD Athlon 64 3800+. Abbil-

[3]Für die CPU-Implementierung wurde auf Teile eines Programms zurückgegriffen, das von Dr. K. Dorfmüller-Ulhaas, E. Saumweber und F. Mücke im Zeitraum des Multimedia-Praktikums im WS06/07 an der Universität Augsburg erstellt wurde.

dung 5.3 zeigt die Bildwiederholraten sowie die entsprechenden Ausführungszeiten bei verschiedenen Auflösungen auf unterschiedlichen GPUs. Dabei wird der Leistungsunterschied offensichtlich: selbst bei einer Auflösung von 1024×768 Bildpunkten läuft die GPU-Anwendung mit über 30 Bildern pro Sekunde noch in Echtzeit ab, während die CPU-Version die Echtzeitanforderung selbst in der niedrigsten Auflösung von 320×240 Bildpunkten verfehlt. Bei den höheren Auflösungen wirkt sich der Speichertransfer von der GPU zur CPU jedoch merklich als Störfaktor aus, weswegen sich das Verhältnis zwischen CPU und GPU etwas angleicht. Der einzige auszumachende Kritikpunkt an der GPU-Version ist die minimal geringere Genauigkeit gegenüber der CPU-Implementierung.

5.2 GPU-KLT Implementierung

Sinha veröffentlichte 2006 eine Implementierung des in Unterabschnitt 3.3.5 beschriebenen KLT-Trackers, welche vollständig auf der GPU abläuft [SFPG06]. Diese Implementierung wird im Folgenden untersucht, Methoden ihrer Umsetzung herausgestellt und ihre Effizienz eingeschätzt. Die verschiedenen Schritte des Tracking-Algorithmus werden durch jeweils verschiedene Fragmentprogramme abgebildet. Diese Schritte sind in Abbildung 5.4 dargestellt.

Bei der GPU-Implementierung wird jedes Einzelbild in den Grafikspeicher geladen und dort geglättet. Daraus wird dan die mehrstufig aufgelöste Pyramide der Helligkeitswerte und der Gradienten erstellt. Das Tracking wird für jedes Einzelbild angewandt und dabei die Bildpyramiden, die zu dem aktuellen sowie den vorhergehenden Bildern gehören, verwendet. Um die Anzahl der Merkmalspunkte möglichst konstant zu halten, wird alle k Einzelbilder eine Wiederauswahl durchgeführt. Der Wert von k hängt dabei von der Kamerabewegung und der daraus resultierenden Anzahl verlorener Merkmalspunkte ab. Sinha ermittelte in seinen Experimenten einen Wert von $k = 5$ [SFPG06].

5.2.1 Implementierungsdetails

Aufbau der Bildpyramide Im ersten Schritt wird die Bildpyramide erstellt, die aus mehreren Auflösungsstufen der Bildintensitäten sowie den zugehörigen Bildgradienten besteht. Diese wird durch eine aufteilbare Zwei-Wege-Gaußfaltung in zwei Fragmentprogrammen berechnet. Sinha definiert in seinem Programm die Texturkoordinaten direkt in OpenGL, sodass er diese nicht im Fragmentprogramm berechnen muss. Die Größe des 1D-Faltungskerns ist dadurch zwar auf sieben beschränkt, dies reicht aber für die meisten Werte von σ aus. Durch die vektorbasierte Architektur können die ge-

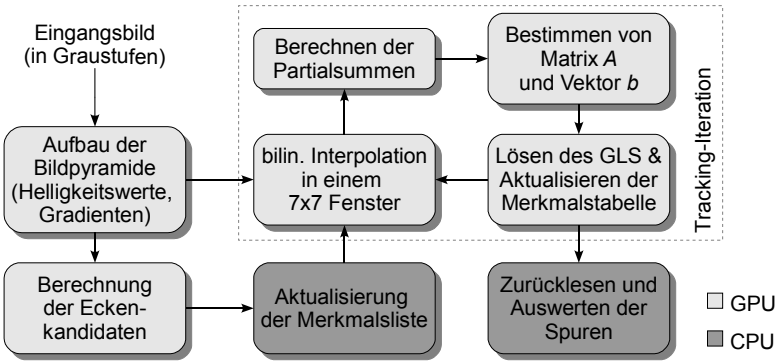


Abb. 5.4: Ablauf der GPU-KLT-Implementierung

glätteten Pixel und die Beträge der Gradienten simultan berechnet werden. Dabei wird, wie auch schon bei der Harris-Implementierung, ein Texturenpaar verwendet, um im zweiten Schritt auf das Ergebnis des ersten Schritts zugreifen zu können (*Pingpong*-Verfahren, S. 45). Da der Tracker nur Informationen aus jeweils zwei Einzelbildern gleichzeitig benötigt, reicht es aus, nur Texturen für zwei Bildpyramiden anzulegen. Ein zwischen 1 und 2 alternierender Zeiger gibt dann das jeweilige Einzelbild an. Beim KLT-Tracking wird eine feste Anzahl von Tracking-Iterationen für jede Auflösungsstufe der Bildpyramide, beginnend bei der gröbsten, durchgeführt. In jeder dieser Iterationen wird für jeden Merkmalspunkt ein lineares Gleichungssystem mit zwei Unbekannten ($Ad = b$, siehe Gleichung 3.3 in Unterabschnitt 3.3.5) aufgestellt und gelöst, um die geschätzte Verschiebung des Merkmalspunktes zu aktualisieren [SFPG06]. Dies wiederum wird in vier Teilschritten, durch verschiedene Fragmentprogramme, erledigt.

Tracking-Iteration Zunächst werden die Intensitäts- und Gradientenwerte, in einem 7×7-Feld um jedes KLT-Merkmal herum, bilinear interpoliert. Dies funktioniert auf NVIDIA GPUs laut Sinha in Hardware, wohingegen AMD(ATI)-Karten keine bilineare Filterung von Gleitkommatexturen unterstützen und deswegen ein zusätzliches Fragmentprogramm benötigen. Sinha übersieht hier jedoch, dass auch GeForce-Architekturen nur eine Filterung auf 16-Bit-Gleitkommatexturen in Hardware ermöglichen. Für höhere Genauigkeit wird auch hier ein separates Fragmentprogramm zur Filterung benötigt.

Die in den 7×7 großen Bildblöcken bestimmten Werte werden anschließend in zwei Schritten hinzugefügt: Erst werden die Partialsummen über die Zeilen und dann die Summe über deren Ergebnisse, die Spaltensumme, berechnet.

Die Auswertung aller sechs Elemente der Matrix A und des Vektors b geschieht in zwei weiteren Fragmentprogrammen, die ihre Ergebnisse für den nächsten Teilschritt

in eine weitere Textur schreiben.

Das nächste Fragmentprogramm löst die KLT-Gleichung (Gleichung 3.3) und schreibt dann die aktuell verfolgte Position in die nächste Zeile der Merkmalstabelle, die durch eine weitere Textur repräsentiert ist. Diese Teilschritte bilden genau eine Tracking-Iteration des originalen Algorithmus (siehe S. 21) ab.

Zurücklesen und Auswerten der Spuren Nach $m \times n$ Iterationen wird die endgültige Position des Merkmalspunktes sowie Δd, der letzte Aktualisierungswert des Trackers, und res, das SSD-Residuum (Summe der Differenzquadrate) zwischen dem ursprünglichen und dem verfolgten Bildstück, auf die CPU zurück gelesen. Sinha verwendet an dieser Stelle einen Schwellwert für Δd und res, um ungenaue Verfolgungsvorschläge zurückzuweisen. Im originalen KLT Algorithmus wurden diese Tests ursprünglich nach jedem Durchlauf gemacht. Sinha lässt diese jedoch aus, um bedingte Sprünge in den Fragmentprogrammen einzusparen und somit einen Geschwindigkeitsvorteil zu erzielen. Daher müssen in seiner GPU-Implementierung immer die maximale Anzahl Merkmale N verfolgt werden. Die Laufzeit hängt dann direkt von N und nicht von der Anzahl gültiger Merkmalspunkte ab.

Berechnung der Eckenkandidaten Die Karte, die die Eckigkeitswerte enthält, wird in zwei Schritten berechnet. Im ersten wird für jeden Pixel ein lokaler Strukturtensor bestimmt. Im zweiten Schritt wird der kleinste Eigenwert dieser Matrix in einer Textur, der Eckigkeitskarte (*cornerness map*), gespeichert. Dabei wird nur der erste Farbkanal verwendet, um später beim Zurücklesen Bandbreite zu sparen.

Aktualisierung der Merkmalsliste Um stets eine konstante Anzahl an Merkmalen zu verfolgen, ist nach einer bestimmten Zeit eine Erneuerung der Referenzmerkmale notwendig. Während der Wiederauswahl der Merkmale werden die Nachbarwerte bestehender Merkmalspunkte bei Sinhas Implementierung von den weiteren Berechnungsschritten ausgeschlossen. Dies wird erreicht, indem man sie bei der Verdeckungsabfrage durch den Tiefentest fallen lässt. Dadurch werden die zu den ausgeschlossenen Punkten gehörenden Fragmente bei der Berechnung der Eckigkeitskarte nicht mehr erzeugt. Das Ausschließen der Bildbereiche vor der Berechnung der Eckigkeitskarte macht sich nach Sinha bezahlt, da diese dadurch erheblich schneller wird [SFPG06]. Schließlich wird die Eckigkeitskarte auf die CPU zurückgelesen und alle nichtmaximalen Werte in einer Umgebung um den Kandidaten herum unterdrückt, um die zusätzlichen Merkmalspunkte herauszufiltern. Diese werden dann in der Liste gespeichert und wieder auf die GPU geladen.

5.2.2 Ergebnisse

Zur Überprüfung der Leistungsfähigkeit der GPU-Implementierung wurden in [SFPG06] Tests auf verschiedenen ATI und NVIDIA Grafikkarten durchgeführt. Diese Tests zeigten, dass die GPU-Version des KLT-Algorithmus gegenüber der CPU-Implementierung etwa 20-mal so schnell arbeitet. Sie erreicht bei einem Ausgangsvideo mit 1024×768 Bildpunkten und 1000 verfolgten Merkmalspunkten 30 Bilder pro Sekunde. Diese Ergebnisse konnten durch eigene Testergebnisse in etwa bestätigt werden. Die Abbildung 5.5 zeigt eine Auswertung der Laufzeiten der KLT-Implementierungen von Sinha (GPU) und Birchfield [Bir] (CPU).

Auflösung	GeForce 7900 GT	GeForce 7800 GT	GeForce 6800	Athlon64 3800+
320x240	48	44	38	9
512x384	36	34	24	6
640x480	29	27	18	4
800x600	22	20	14	3
1024x768	16	15	8	1

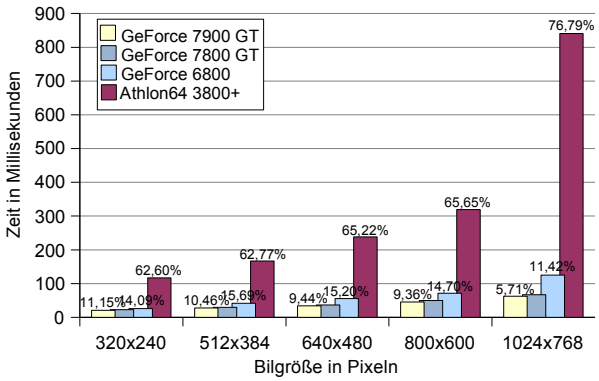


Abb. 5.5: Performanceauswertung der KLT-Implementierung von Sinha
Die Tabelle gibt die Anzahl Bilder pro Sekunde bei verschiedenen Auflösungen wieder. Bei den verwendeten Szenen wurden maximal 1000 Merkmale erkannt. Die Grafik zeigt die mittlere Zeit eines Durchlaufs. Für echtzeitnahe Geschwindigkeit sind höchstens 50ms (entspricht mindestens 20fps) erforderlich. Die Prozentwerte geben das jeweilige Verhältnis der Implementierungen zueinander wieder.

Die NVIDIA Karten schnitten bei Sinhas Tests etwas schlechter ab als die von AMD(ATI), was laut Sinha daran liegt, dass bei AMD(ATI) die Berechnungen immer in 32-Bit Gleitkommagenauigkeit durchgeführt werden, selbst wenn die Texturen nur in 16-Bit vorliegen. Dadurch wird nur die Hälfte der Bandbreite für den Transfer der Texturdaten benötigt. Dies liegt wiederum an der Tatsache, dass AMD(ATI) bis vor kurzem gar keine hardwareseitige Unterstützung von 16-bittiger Genauigkeit vorsieht und daher Gleitkommawerte intern immer als 32-Bit betrachtet, wie schon in Unterabschnitt 4.1.4 ausführlich erläutert wurde. Da aber beim Lösen der Gleichung 3.3 möglichst mit voller Genauigkeit gerechnet werden soll, müssen bei NVIDIA-GPUs die Texturen auch in 32-Bit vorliegen.

Sinhas Tests zeigen außerdem, dass die GPU-Implementierung des KLT-Algorithmus auf allen Grafikkarten lediglich durch die verfügbare Bandbreite beschränkt ist. Die Komplexität der durchzuführenden Berechnungen hingegen hängt nur linear von

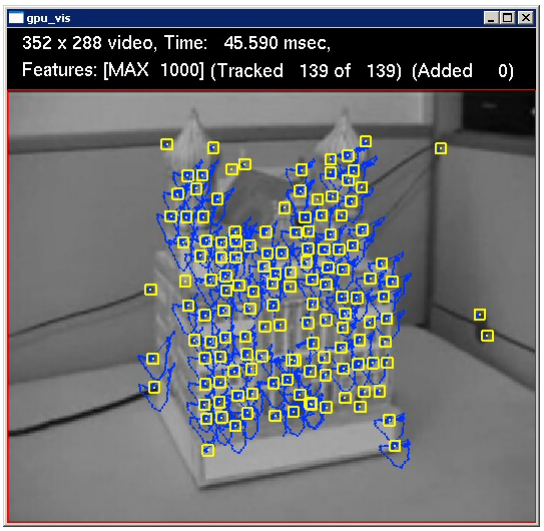


Abb. 5.6: GPU-KLT Implementierung in Aktion
Die gelben Markierungen kennzeichnen die aktuell verfolgten Merkmale, die blauen Linien zeigen die Bewegung der Kamera an.

der Anzahl der Merkmalspunkte sowie der Anzahl Pixel im Bild ab [SFPG06]. Sinha überprüft in seiner Veröffentlichung die Genauigkeit seiner GPU-Implementierung beim Tracking innerhalb einer Anwendung zur Posenbestimmung der Kamera. Die Qualität der geschätzten Kameraposen diente dabei als Kriterium für die Genauigkeit der Merkmalsverfolgung. Dabei waren laut Sinha keine Unterschiede zur CPU-Implementierung auszumachen außer, dass die GPU-Version deutlich schneller war [SFPG06].

5.3 OpenVIDIA Feature-Tracker

Das OpenVIDIA Projekt [Ope], das von James Fung [FM04, FM05] ins Leben gerufen wurde, liefert eine quelloffene Bibliothek, die sich der Beschleunigung der Bildanalyse und des Maschinellen Sehens, mittels einer oder mehrerer Grafikkarten, verschrieben hat. Es stellt ein einfaches API zur Verfügung, das einige gebräuchliche Algorithmen des Maschinellen Sehens implementiert. Darunter sind Bildverarbeitungstechniken (Canny Kantendetektor, Bildfilterung), der Umgang mit Bildmerkmalen (Merkmale identifizieren und vergleichen) und Bildregistrierung zu finden [FM05]. Darüber hinaus untersucht das OpenVIDIA Projekt eine parallele *Graphics-for-vision*-Architektur, bei der mehrere Grafikkarten eines Rechners zusammengefasst werden. So lässt sich eine preisgünstige Architektur für hardwarebeschleunigtes Maschinelles Sehen und Signalverarbeitung realisieren [FM04].

Der verwendete *OpenVIDIA Feature-Tracker* basiert auf den frei verfügbaren Quellen von `libopenvidia-0.8.3`. Das Beispielprogramm zur Merkmalsverfolgung wurde

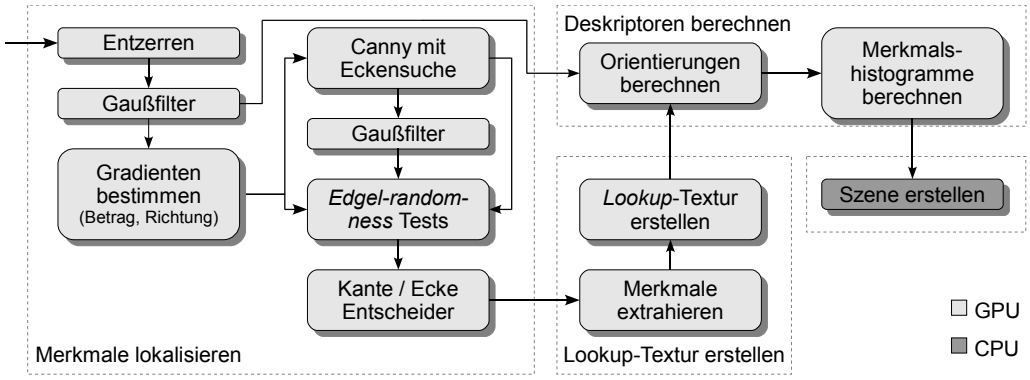


Abb. 5.7: Ablauf des OpenVIDIA Merkmalsdetektors

vollständig nach Windows portiert und die Schnittstelle der Firewire-Kamera durch einen DirectShow-Video-Grabber[4] ersetzt, der, statt des Kamerabildes, die Bilder aus einer beliebigen DirectShow-Quelle wie etwa den Testvideos lesen kann.

Das Feature-Tracker-Programm erzeugt für jedes Merkmal einen 128-elementigen Deskriptor, der denen des SIFT-Algorithmus aus Unterabschnitt 3.3.6 nachempfunden ist. Aufgrund etlicher Vereinfachungen sind diese jedoch nicht so robust, wie die in [Low04] von Lowe vorgestellten. Zur Erkennung der Merkmale wird ein modifizierter Canny-Kantendetektor verwendet, der gleichzeitig Informationen über Ecken liefern kann. Die eigentliche Merkmalsverfolgung des OpenVIDIA Feature-Trackers ist durch ein RANSAC-Verfahren auf der CPU implementiert. Daher liegt im Folgenden das Hauptaugenmerk auf der Merkmalsextraktion, die auf der GPU umgesetzt wurde.

5.3.1 Implementierungsdetails

Die Merkmalsextraktion des OpenVIDIA Feature-Trackers ist, wie in Abbildung 5.7 dargestellt, in vier wesentliche Schritte unterteilt: Merkmale lokalisieren, die Lookup-Textur erstellen, Deskriptoren berechnen und die Szenenbeschreibung, bestehend aus den Deskriptoren, erstellen.

Merkmale lokalisieren Zunächst werden in OpenGL die Texturen für die spätere Verwendung generiert. Dabei wird stets `GL_LINEAR` als Filteroperation gewählt, um durch die hardwareseitige, bilineare Interpolation die Genauigkeit beim Zugriff auf Texturwerte zu erhöhen. Das eingehende Videobild wird dann mit einem Entzerrungsfilter bearbeitet, um der Linsenkrümmung der Kamera entgegenzuwirken. Die Parameter der verwendeten Kamera können dabei einfach dem Fragmentprogramm übergeben

[4] Dieser basiert auf einer Bibliothek von V. Wiendl, die während des Multimedia Praktikums an der Universität Augsburg im WS06/07 verwendet wurde.

oder dort direkt geändert werden. Das entzerrte Eingangsbild wird im nächsten Schritt durch eine zweistufige 7×7-Gaußfaltung geglättet. Durch die Aufteilung der Gaußfaltung in zwei Schritte spart man, auf Kosten eines zweiten Durchlaufs, etliche Texturzugriffe. Die Offsets der Texturkoordinaten für die umliegenden Pixel werden dabei direkt in OpenGL mittels `glMultiTexCoord2fARB` gesetzt. Dies wird im OpenVIDIA Feature-Tracker generell bei allen Fragmentprogrammen für alle Werte, wie Koordinaten, Offsets oder Glättungsfaktoren, die vorberechnet werden können, gemacht, sodass diese nicht in jedem Fragment erneut berechnet werden müssen. Als Nächstes werden aus dem geglätteten Bild die Bildgradienten bestimmt. Dabei werden ihre Steigungen sowie der Betrag und ihre Richtung in den `RGBA`-Komponenten kodiert. Aus den Gradienten werden dann mit einem modifizierten Canny-Kantendetektor so genannte *Edgel* (*edge pixel*) ermittelt. Der zugehörige Code des Detektors ist im folgenden Codelisting dargestellt:

```
void CannySearchWithCorners( const uniform float4 thresh,
    in float2 fptexCoord[0..7] : TEXCOORD[0..7],
    out float4 color0 : COLOR0,
    out float4 color1 : COLOR1,
    in float4 wpos : WPOS,
    const uniform samplerRECT FPE1 : TEXUNIT0 )
{
    // magdir holds { dx, dy, mag, dir }
    float4 magdir = texRECT( FPE1, fptexCoord0 );
    float  angle  = 0.5/sin( 3.14159/8.0 );  // eight dirs on grid
    float2 offset = round( angle.xx * magdir.xy / magdir.zz );

    float4 fwdneighbour, backneighbour;
    fwdneighbour  = texRECT( FPE1, fptexCoord0 + offset );
    backneighbour = texRECT( FPE1, fptexCoord0 - offset );

    if ( magdir.z < thresh.x )
        color0 = float4( 0.0, 0.0, magdir.w, magdir.z );
    else
    {
        // not an edgel
        if( fwdneighbour.z > magdir.z || backneighbour.z > magdir.z )
            color0 = float4( 0.0, 1.0, magdir.w, magdir.z );
        // is an edgel
        else
            color0 = float4( 1.0, 1.0, magdir.w, magdir.z );
    }
    // For corners, its gradient magnitude has to be a local max
    // => suppress non corners
    if( magdir.z <= texRECT(FPE1, fptexCoord1).z || ...
        magdir.z <= texRECT(FPE1, fptexCoord7).z ||
        magdir.z <= texRECT(FPE1,-fptexCoord7).z )
        color0.y = 0.0; // This is not a corner max

    // magdir gives us: float4(dx, dy, mag, dir) when edge present,
    // or (0.0, ... ) if not an edge
```

```
    float4 vec1 = { magdir.x, magdir.x, magdir.y, 1.0 };
    float4 vec2 = { magdir.x, magdir.y, magdir.y, 1.0 };
    color1 = vec1 * vec2;
}
```

Quellcode 5.3: Cg-Programm des Canny-Kantendetektors mit Eckensuche
Das Programm überprüft die aktuelle Richtung und bestimmt dann, welche Pixel in Richtung des Gradienten (d.h. senkrecht zu einer Linie) liegen. Diese Information wird weitergegeben, falls ein Maximum in Richtung des Gradienten gefunden wird, andernfalls wird die Ausgabe unterdrückt. Das Programm sucht außerdem nach Ecken, die Maxima der lokalen Gradienten sind. Diese werden durch Ausgabe von `color0.y = 1.0` gekennzeichnet.

Hierbei werden *Multiple Render Targets* (siehe Abschnitt 4.5) benutzt, um zusätzlich zur Orientierung und dem Betrag der Eckenkandidaten, auch eine Art Strukturtensormatrix aus den Tangentenwerten auszugeben. Diese wird anschließend mit einem zweistufigen 19×19-Gaußfilter geglättet. Im nächsten Schritt werden aus den geglätteten Einträgen die beiden Eigenwerte λ_1 und λ_2 der Matrix S mittels Eigenwertzerlegung bestimmt, wobei λ_2 der kleinere von beiden ist. Diese werden dazu benutzt, um, ähnlich wie beim Harris-Eckendetektor, ein Maß für die Eckigkeit des Punktes zu bekommen. Als weiteres Kriterium wird zusätzlich noch ein Maß für die Zufälligkeit, die *randomness*, eingeführt,

$$\text{cornerness} = \frac{\det(S)}{\text{Spur}(S) + \epsilon}, \quad \text{randomness} = \frac{\lambda_2}{1 + \frac{1}{2}(\lambda_1 + \lambda_2)},$$

wobei $\epsilon = 1{,}0 \cdot 10^{-10}$ gewählt wird. Ist der Zufälligkeitswert des Punktes klein und ist dieser ein Edgel, so wird dieser in der x-Komponente des Ausgabewertes mit dem Wert 0,5 als Linienpunkt markiert. Als Schwellwert ist hierfür ein Wert von 0,5 gesetzt. Ist der Zufälligkeitswert im Punkt jedoch hoch und dieser ist ein Edgel, so ist dieser eine Ecke (Merkmalspunkt). Der Wert für die Eckigkeit wird in der w-Komponente zur Auswertung im nächsten Schritt übergeben; in der y- und z-Komponente wird der Betrag und der auf das Intervall [0,1] skalierte Richtungswert gespeichert. Im letzten Schritt der Merkmalslokalisierung wird überprüft, ob die Eckigkeit im jeweiligen Punkt größer oder gleich 11 ist, und ob es sich dabei um ein lokales Maximum handelt. Ist dies der Fall, so wird die x-Komponente mit dem Wert 1,0 kodiert. Als Ergebnis der Merkmalslokalisierung erhält man eine Textur, in der sämtliche Merkmalskandidaten wie folgt in den `RGBA`-Komponenten kodiert sind: in der x-Komponente steht 0,0 für kein Merkmal, 0,5 für eine Canny-Kante und 1,0 für eine Ecke; in der y- und z-Komponente sind immer noch Betrag und Richtung des Gradienten zu finden und in der w-Komponente der Wert für die Eckigkeit.

Lookup-Textur erstellen Um die Merkmalspunkte effizient verarbeiten zu können, wird eine *Lookup-Textur* erzeugt. Dies ist eine eindimensionale Textur, die lediglich die x- und y-Koordinaten in ihren Rot- und Grünkanälen gespeichert hat. Um diese zu erzeugen, wird zunächst für alle roten Punkte, das sind diejenigen, die in der x-Komponente eine 1 haben, also die Eckpunkte, im Schablonenpuffer eine 1 erzeugt. Statt der kompletten `RGBA`-Tupel wird lediglich der Schablonenpuffer ausgelesen und dabei wertvolle Bandbreite gespart. Aus den ausgelesenen Daten werden anschließend die Positionen der Merkmalspunkte berechnet und der Reihe nach in einer Textur gespeichert. Dabei wird die Breite der Textur auf die Fensterbreite beschränkt. In einem Video mit 640×480 Bildpunkten werden also maximal 640 Merkmalspunkte verarbeitet. Dies führt leider dazu, dass in höheren Auflösungen unter Umständen nur die Merkmale im oberen Teil des Bildes ausgewertet werden, wenn viele Merkmale im Bild vorhanden sind.

Deskriptoren berechnen Das Problem beim Erstellen der Deskriptoren ist, dass für jeden Eingabewert mehrere, nämlich genau 128, Ausgabewerte erzeugt werden müssen. Daher wird für jeden Merkmalspunkt eine Linie, bestehend aus 128 Punkten, gezeichnet und diese so mit der Lookup-Textur texturiert, dass ihr in jedem Punkt die korrekten Koordinaten zugewiesen werden. Den Linien werden außerdem mittels `glMultiTexCoord2fARB` die entsprechenden Nachbaroffsets zugewiesen. Um die Deskriptoren zu bestimmen, wird zunächst die Hauptorientierung des Merkmalspunktes, wie bei SIFT in Unterabschnitt 3.3.6, bestimmt. Anschließend werden Orientierungshistogramme erstellt.

Zur Bestimmung der Hauptorientierung des Merkmalspunktes, wird in einem 16×16-Feld, welches über dem Merkmalspunkt zentriert ist, jeweils eine Summe aus den Werten der x- und y-Komponenten der ersten Gaußfaltung, die bereits im Lokalisierungsschritt berechnet wurde, gebildet. Diese Faltungswerte nähern die erste Ableitung in x- und y-Richtung an. Dabei werden die Summanden mittels einer vorberechneten Gaußmaske geglättet. Der Effizienz halber, wird dieser Schritt in vier 8×8 große Teilbereiche zerlegt, und deren Summen in einem zusätzlichen Schritt addiert.

Im nächsten Schritt wird eine 16-zeilige Textur erzeugt, deren Breite einer Fensterbreite entspricht. Diese Ergebnis-Textur enthält pro Eintrag acht gepackte 16-Bit Histogrammwerte. In einer Spalte der Textur erhält man somit den kompletten 128-elementigen Deskriptorvektor des Merkmalspunktes.

Histogrammbildung: Um Histogramme zu erstellen, fehlen der GPU effiziente, indexbasierte Arrayoperationen und Zeigeroperationen, wie sie auf der CPU zur Histogrammberechnung verwendet werden. Durch ausgiebigen Gebrauch von `if` und `else` kann zwar auch der jeweilige Platz eines Wertes im Histogramm bestimmt werden,

dies führt aber nicht zwingend zu einer effizienten Berechnung, da Verzweigungen auf heutigen GPUs immer noch recht teuer sind. Im OpenVIDIA Feature-Tracker wird daher versucht, die Histogrammbildung über eine Vektoroperation zu lösen [FM05]. Um dies zu erreichen wird die Kosinusfunktion benutzt, die von Fragmentprozessoren zur Verfügung gestellt wird. Das Ziel ist die Histogrammbildung der Gradientenrichtungen in einer kleinen Umgebung um den Merkmalspunkt über acht Histogrammbehältern. Um dies zu bewerkstelligen, wird jede mögliche Gradientenrichtung, deren Werte aus dem Intervall `[0, 7]` stammen, von einem Vektor abgezogen. Dieser Vektor beinhaltet hierbei eine Folge ganzzahliger Werte von 0 bis 7. Wendet man nun den Kosinus auf den Differenzvektor an, so erhält man Kosinuswerte, die an der gewünschten Position maximal sind. Addiert man nun einen bestimmten Offsetwert und quadriert das Ergebnis, erhält man eine Spitze, die in der Behälterposition des Histogramms zentriert ist. Die Kosinusfunktion kann mehrmals quadriert werden, um die Werte stärker zu zentrieren. Die Werte in den übrigen Behältern sind zudem auch noch nützlich, da sie dem Rauschen, das durch die Quantisierung entsteht, entgegenwirken. Ein eventuelles Glätten des Histogramms entfällt also. Zum Erhalt der Rotationsinvarianz werden bei der Histogrammbildung alle Orientierungen relativ zur zuvor berechneten Hauptorientierung bestimmt. Vor der Ausgabe werden dann jeweils zwei `float`-Werte mittels der Cg-Funktion `pack_2half` in zwei 16-Bit-Werte gepackt, die wiederum durch einen `float`-Wert repräsentiert sind. Durch diese Aufteilung können pro Ausgabe acht Werte mit einem einzigen `RGBA`-Tupel geschrieben werden. Die Endtextur kann daher in zwei Zeichendurchgängen mit acht-zeiligen Bereichen erzeugt werden. Durch diese Reduktion lässt sich die benötigte Bandbreite beim Zurücklesen auf die CPU um die Hälfte reduzieren. Die Daten werden für die Szenenbeschreibung in einen Puffer gelesen.

Szenenbeschreibung erstellen Im letzten Schritt wird aus den Merkmalsdaten die Szenenbeschreibung erstellt. Diese besteht aus einem Vektor, der für alle gefundenen Merkmalspunkte folgende Daten enthält: den 128-elementigen Deskriptorvektor, die Hauptorientierung, die Ableitungen in x- und y-Richtung in einer 16×16-Umgebung, den Betrag des Gradienten und eine eindeutige Identifikationsnummer.

Merkmalsverfolgung Das Tracking der Merkmale läuft vollständig auf der CPU ab und verwendet dazu eine projektive Koordinatentransformation (PCT), die auch beim *VideoOrbits Head-Tracker* von Steve Mann zum Einsatz kommt und Teil der Bibliothek von OpenVIDIA ist [FM04]. Diese wird mittels RANSAC-Algorithmus aus vier zufälligen Übereinstimmungen ermittelt. Dabei werden offensichtlich falsche Transformationen verworfen.

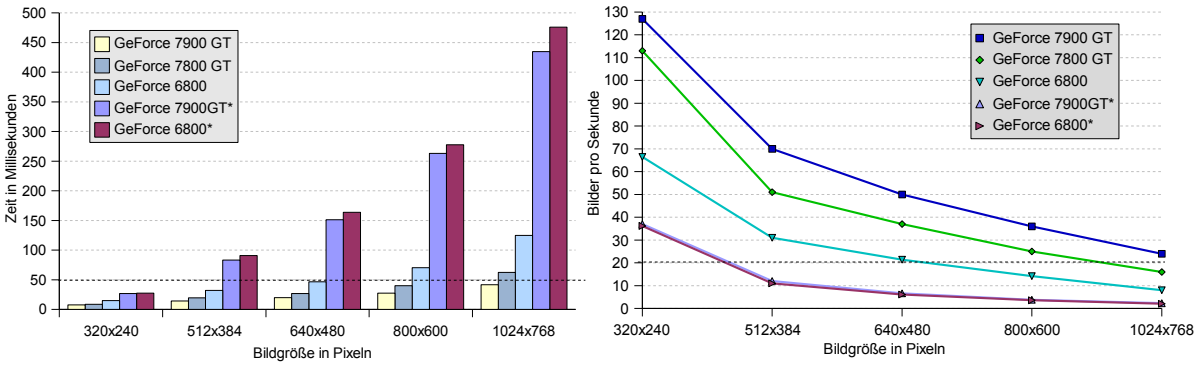


Abb. 5.8: Performanceauswertung des OpenVIDIA Feature-Trackers
*Bei den mit * gekennzeichneten Systemen wurde neben der Merkmalsextraktion auf der GPU auch das Tracking auf der CPU berücksichtigt. Die CPU-Limitierung bei aktivierter Merkmalsverfolgung ist deutlich erkennbar. Der Extraktionsvorgang läuft dagegen selbst bei 1024x768 Bildpunkten mit 24 fps noch in Echtzeit ab.*

Tab. 5.1: Zeitmessung beim OpenVIDIA Feature-Tracker
*Die Zeit für einen Durchlauf auf verschiedenen GeForce-Karten ist in Millisekunden angegeben. Dabei wurden die Werte auf Karten, die mit * gekennzeichnet sind, mit eingeschaltetem Tracking ermittelt.*

Auflösung	7900 GT	7800 GT	6800	7900GT*	6800*
320x240	8	9	15	27	28
512x384	14	20	32	83	91
640x480	20	27	47	152	164
800x600	28	40	70	263	278
1024x768	42	63	125	435	476

5.3.2 Ergebnisse

Aufgrund einiger Fehler im originalen Code, berechnete der OpenVIDIA Merkmalsextraktor nur einen 64-elementigen Deskriptorvektor. Dafür erreichte er beim Extraktionsvorgang in den Messungen annähernd die doppelte Bildwiederholrate.

Die Leistungsmessungen des OpenVIDIA Feature-Trackers sind in Abbildung 5.8 dargestellt. Diese wurden allerdings mit der korrigierten Version, welche den vollen 128-elementigen Deskriptorvektor erzeugt, durchgeführt. Diese zeigen, dass der Extraktionsprozess selbst in einer 1024er-Auflösung noch in Echtzeit abläuft. Ein Durchlauf benötigte auf einer NVIDIA GeForce 7900 GT gerade einmal 42 Millisekunden, was 24 Bildern pro Sekunde entspricht. Anders sieht es aus, wenn man die Merkmalsverfolgung einschaltet: diese ist stark durch die vorhandene CPU-Leistung beschränkt und bremst daher gerade modernere GPUs aus. Dies kann man leicht aus den Werten in Tabelle 5.1 ablesen: mit eingeschaltetem Tracking braucht eine aktuelle GeForce 7900 GT mit 435ms in etwa so lange für einen Durchlauf wie eine nicht mehr zeitgemäße GeForce 6800 (476ms).

Wie robust die Merkmalsbeschreibungen sind und wie hoch die erzielte Genauigkeit im Vergleich zum originalen SIFT-Algorithmus ist, müsste in weiteren Tests überprüft werden. Bei den Invarianzeigenschaften müssen auf jeden Fall Abstriche gemacht werden, da beispielsweise auf die Umsetzung des Skalenraums, der zur Umsetzung der Skalierungsinvarianz nötig ist, verzichtet wird.

Das OpenVIDIA Projekt bietet mit seinem Feature-Tracker eine ziemlich hoch optimierte Lösung zur Merkmalsverfolgung auf der GPU. Die Leistung des Tracking-Algorithmus reicht jedoch nicht für höher aufgelöste Echtzeitanwendungen aus. Für einfache VGA-Auflösung mit 640×480 Bildpunkten hätte eine moderne GPU genügend Leistungsreserven, um auch Berechnungen für das Tracking zu übernehmen. Hier gilt es noch effizientere Verfahren zu entwickeln.

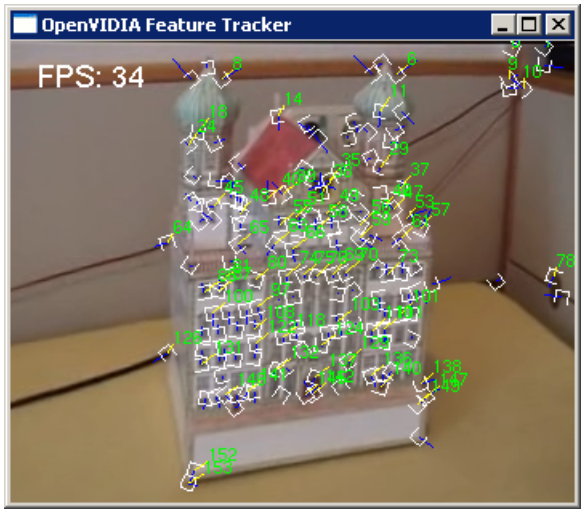


Abb. 5.9: Modifizierter OpenVIDIA Feature-Tracker in Aktion
Die offenen Quadrate zeigen die Hauptrichtung des Merkmalspunktes, die blauen Linien die Richtung des Gradienten an, die gelben Linien zeigen zur ursprünglichen Position des Punktes und die grünen Nummern entsprechen den eindeutigen IDs der Merkmalspunkte.

5.4 GPU-SIFT Implementierungen

Neben dem ursprünglichen Programm von David A. Lowe, existieren mittlerweile verschiedene andere Implementierungen sowohl für die CPU als auch für die GPU. Die Originalversion ist jedoch nur als fertig übersetzte Binärdatei verfügbar und bietet keinerlei Parameter, mit denen sich Änderungen vornehmen lassen. Nach dem starken Interesse vieler Forscher an SIFT, entstanden diverse Implementierungen in verschiedenen Programmiersprachen, darunter Matlab, C++ und C#.

Sebastian Heymann [Hey05] war der Erste, der SIFT im Rahmen seiner Diplom-

arbeit auf der GPU umsetzte. Anfang 2006 veröffentlichte Sudipta Sinha ein Dokument, das eine GPU-Implementierung der *Scale Invariant Feature Transform* beschrieb [SFPG06]. Darin stellte er auch seine GPU-KLT Implementierung, die in Abschnitt 5.2 behandelt wird, vor. Leider entstanden beide SIFT-Implementierungen als Projektarbeiten in Firmen, so dass weder Quellcode noch ausführbare Programme verfügbar sind, die für Leistungsmessungen oder die Analyse genutzt werden konnten. Die Ergebnisse beider Arbeiten sind daher mit Vorsicht zu genießen, da sie nicht verifiziert werden können. Auch bleiben einige Aspekte der Implementierung, wie etwa deren Flexibilität oder Genauigkeit, im Dunkeln.

Im Folgenden werden diese Implementierungen betrachtet und, soweit möglich, im Bezug auf ihre Leistungsfähigkeit eingeschätzt.

Zunächst wird eine eigene Implementierung analysiert, die mit Heymanns Beschreibung als Grundlage entwickelt wurde. Im Anschluss daran wird die GPU-SIFT-Umsetzung von Sinha untersucht.

5.4.1 Heymanns/eigene GPU-SIFT Implementierung

Heymann beschreibt in [Hey05] eine Umsetzung von SIFT auf der GPU. Diese Beschreibung sowie die CPU-Implementierung von Vedaldi [Ved] dienten als Grundlage für eine eigene Implementierung, die im Folgenden I_3 genannt wird.

Implementierungsdetails

Der SIFT-Algorithmus (Unterabschnitt 3.3.6) wird bei dieser Implementierung durch die folgenden Schritte umgesetzt: Zunächst wird die gaußsche Skalenraumpyramide und die entsprechende DoG-Pyramide erstellt. Dann werden die Extrema in den DoG-Ebenen bestimmt und gefiltert. Diese werden dann auf die CPU übertragen und dort als Merkmalspunkte gespeichert. Für jeden Merkmalspunkt werden dann die Gradientenrichtungen in einer lokalen Umgebung bestimmt und ausgewertet. Die dadurch gewonnenen rotationsinvarianten Gradientenrichtungen werden in den Merkmalsdeskriptoren gespeichert.

Erstellen des gaußschen Skalenraums Der Aufbau der Skalenraumpyramide ist ein rechenintensiver Teil, der auf der CPU den größten Teil der Rechenzeit in Anspruch nimmt. Durch den Einsatz der GPU lässt sich dieser Aufbau jedoch entscheidend beschleunigen. Um die GPU möglichst effizient bei der Berechnung der Gaußebenen des Skalenraums einzusetzen, schlägt Heymann vor, die Helligkeitswerte der Bilder in den vier Farbkanälen gepackt abzulegen. Dies ist möglich, da der SIFT-Algorithmus ohnehin nicht auf die Bearbeitung farbiger Bilder ausgelegt ist. Heymann sieht hier vor,

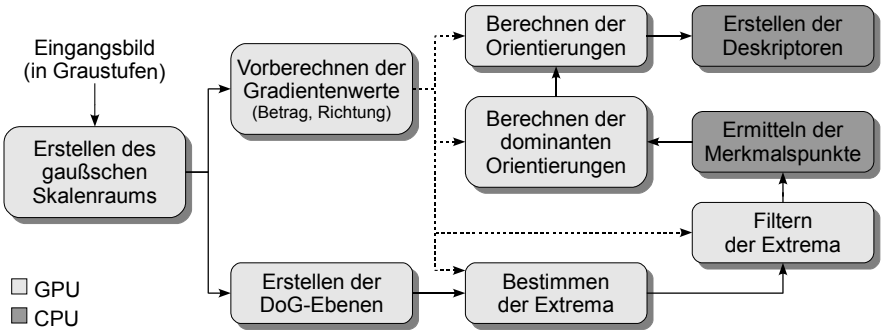


Abb. 5.10: Ablauf der GPU-SIFT Implementierung

jeweils die umliegenden Grauwerte in den `RGBA`-Kanälen so zu speichern, dass in jedem `RGBA`-Farbwert vier Grauwerte gespeichert werden und sich so die Texturgröße auf ein viertel reduziert. Er berücksichtigt in seiner Implementierung allerdings nicht die von Lowe in [Low04] vorgeschlagene Verdopplung der Eingangsbildgröße. Diese trägt laut Lowe jedoch maßgeblich zur Stabilität der SIFT-Merkmale bei. In I_3 werden diese beiden Schritte effizient in einem Fragmentprogramm kombiniert: durch den Einsatz von bilinearer Filterung lässt sich die Bildgrößenverdopplung einfach durch Abtasten der Textur in den Sub-Texeln umsetzen. Die Sub-Texel werden dann in den jeweiligen Farbkanälen gespeichert. Auf eine komplizierte Berechnung der Texturkoordinaten kann in diesem Fall verzichtet werden, da die Koordinaten für die Sub-Texel in OpenGL definiert werden können und automatisch von der Rastereinheit korrekt interpoliert werden.

Der Aufbau der Skalenraumpyramide wird durch eine Aneinanderreihung von Fragmentprogrammen realisiert. Auf dem gepackten und vergrößerten Eingangsbild wird anschließend eine Gaußfaltung mit verschieden großen Gaußkernen angewandt. Wie auch schon bei den bisherigen Implementierungen, wird diese in jeweils zwei Durchgängen durchgeführt. Hier macht sich das Packen der Bildpunkte bezahlt, da so zusätzlich Texturzugriffe gespart werden können. Für die Berechnung der nächsten Skalierung wird das Bild mit zweifachem Glättungsfaktor (2σ) halbiert und dann genauso verfahren, wie mit dem Eingangsbild. Die Halbierung erreicht man einfach durch Weglassen jeder zweiten Zeile und Spalte, was durch die ursprüngliche Packfunktion von Heymann erreicht werden kann.

Der gesamte Vorgang wird solange wiederholt, bis die Grenzgröße der Skalenraumpyramide erreicht ist.

Erstellen der DoG-Ebenen Die gepackten DoG-Ebenen des zugehörigen Skalenraums werden einfach durch Subtraktion der einzelnen Gaußfaltungsstufen in einem Frag-

mentprogramm erzeugt.

Vorberechnen der Gradientenwerte Für die späteren Schritte wird die Richtung und der Betrag (Größe) der Gradienten benötigt. Deswegen werden diese im Voraus berechnet. Die Berechnung erfolgt, wie in Abbildung 5.11 illustriert, unter Verwendung der gepackten Daten. Das Fragmentprogramm schreibt die Ergebnisse dann mittels MRTs (siehe Abschnitt 4.5) in zwei verschiedene Texturen, eine für die Richtung der Gradienten, die andere für deren Beträge.

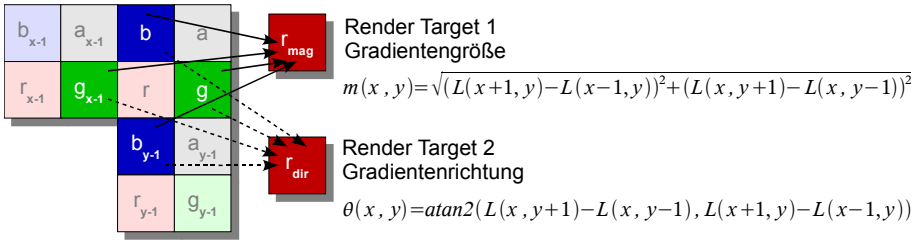


Abb. 5.11: Berechnung der Gradientenwerte
Für den Rotkanal wird die Gradientengröße und dessen Richtung bestimmt. Dies wird auf allen RGBA-Kanälen parallel durchgeführt. Hierfür sind pro Bildpunkt fünf Texturzugriffe nötig.

Bestimmen der Extrema Im nächsten Schritt werden die Extremwerte aus den DoG-Ebenen in einem weiteren Fragmentprogramm bestimmt. Dieses wird für drei benachbarte DoG-Bilder der jeweiligen Skalenebene ausgeführt. Jeder Bildpunkt muss darin entweder einen festgelegten oberen oder unteren Grenzwert überschreiten sowie lokales Maximum oder Minimum der umliegenden 26 Bildpunkte (je neun aus den direkt benachbarten DoG-Bildern) sein, um als Kandidat für einen Merkmalspunkt markiert zu werden. Dadurch entstehen pro Oktave des Skalenraums mehrere Texturen, die diese Extremwerte enthalten. Diese Texturen werden in I_3 durch ein weiteres Fragmentprogramm zu einer einzigen Textur zusammengefügt, wobei ihre ursprünglichen Skalierungen in den Farbwerten kodiert werden. Ein Extremwert aus der ersten Textur bekommt 0,5 zugewiesen, der in der zweiten 0,25 und so weiter.

Filtern der Extrema Bevor die Merkmalskandidaten zur CPU zurückgelesen werden, wird überprüft, dass sie nicht auf einer Kante liegen, und ihre Positionen werden auf Subpixel genau bestimmt. Dieser Schritt wird bei Heymann jedoch nicht beschrieben, weswegen hierzu keine Aussagen bezüglich der Umsetzung getroffen werden können. Er ist jedoch zwingend erforderlich, da die DoG-Funktion stark auf Kanten reagiert, selbst wenn diese schwach sind. In I_3 wird der Kantentest realisiert, indem für jeden Merkmalskandidaten, wie in [Low04] beschrieben, die Hessematrix bestimmt und das

Verhältnis ihrer Eigenwerte beurteilt wird. Die 2×2 Hessematrix kann relativ einfach durch Differenzenquotienten angenähert werden. Der Versuch, die Verfeinerung auf die Subpixel, wie in der Implementierung von Vedaldi, durch Newton-Iterationen zu lösen, führte jedoch zu ziemlich langen Shaderprogrammen und scheiterte letzten Endes wegen zu langer Ausführungszeiten.

Ermitteln der Merkmalspunkte Aus den vorhergehenden Schritten erhält man in I_3 pro Oktave der Skalenraumpyramide eine Textur, in der die Merkmalspunkte markiert sind. Diese werden zur CPU zurückgelesen und daraus die Merkmalspunkte extrahiert. Dabei wird deren Position sowie die Nummer ihres ursprünglichen Skalenbilds gespeichert. Für das Zurücklesen werden hier jedoch noch die gepackten `RGBA`-Daten verwendet. Hier könnte beispielsweise durch eine Binarisierung der sehr dünn besetzten Merkmalstextur die zu übertragende Datenmenge erheblich reduziert und so einer der zeitraubendsten Vorgänge deutlich verkürzt werden. Dies wird im folgenden Abschnitt (5.4.2) bei der Implementierung von Sinha erläutert.

Die Merkmalspunkte werden dann lückenlos in einer Textur gespeichert und wieder zur Grafikkarte übertragen. Dabei werden in den *RGBA*-Komponenten die zum Eingangsbild relativen Koordinaten sowie die Information über die zugehörige Oktave und das Skalenbild kodiert. Die nächsten Schritte müssen dann nur noch mit dieser deutlich kleineren Textur arbeiten anstatt auf den dünn besetzten Merkmalstexturen. Dies erhöht den Durchsatz dieser Schritte deutlich.

Berechnen der dominanten Orientierungen Für die Bestimmung der Deskriptoren werden zunächst die Hauptrichtungen der Gradienten für jeden Merkmalspunkt bestimmt. Diese werden dazu benötigt, um im darauffolgenden Schritt die Richtungen der Gradienten in der näheren Umgebung relativ zu diesen speichern zu können. Dadurch lassen sich die Merkmale später auch unter Rotation wiedererkennen.

Die Hauptrichtungen werden mittels Histogrammbildung über die Gradientenrichtung in einer kleinen Umgebung um den Merkmalspunkt herum bestimmt. Vedaldi übernimmt in seiner CPU-Implementierung die von Lowe vorgeschlagene Aufteilung in 36 Histogrammbehälter, was einem Winkel von zehn Grad entspricht.

Eine effiziente Implementierung dieses Schrittes schien zum Zeitpunkt der Implementierung von I_3 nicht möglich. Auch Heymann scheint diesen Schritt bei seiner Implementierung nicht weiter zu berücksichtigen. Selbst bei Verwendung von nur zwölf Histogrammbehältern, die über konditionale Abfragen ausgewertet werden, werden auf der GPU immer noch fast 700 Befehle dafür benötigt. Dies führt unweigerlich zu Laufzeiten, die eine Ausführung in Echtzeit verhindern.

Berechnen der Orientierungen Bei der Berechnung der Deskriptoren verwendet Heymann zwei Renderziele (MRTs, siehe Abschnitt 4.5), um die gewichteten Histogramme der 4×4 Umgebung um jeden Merkmalspunkt, wie in Abbildung 5.12 illustriert, zu erzeugen. Jedem dieser 16 Bereiche werden acht Richtungen zugewiesen, die zusammen den 128-elementigen Deskriptorvektor ergeben.

Diese Operation unterscheidet sich von den bisher implementierten, da hier zu einem Eingabeelement, dem Merkmalspunkt, 128 Ausgabeelemente erzeugt werden. Dies lässt sich selbst mit den maximal möglichen vier MRTs nicht in einem Schritt realisieren. Heymann benutzt daher ein Gitter aus Vertices, die die 128 Elemente des Deskriptors für einen Merkmalspunkt repräsentieren. Diese werden mit den entsprechenden Koordinaten der einzelnen Bereiche belegt und von einem Fragmentprogramm abgearbeitet. Diese Schritte sind bei Heymann ziemlich undurchsichtig dargestellt und lassen sich

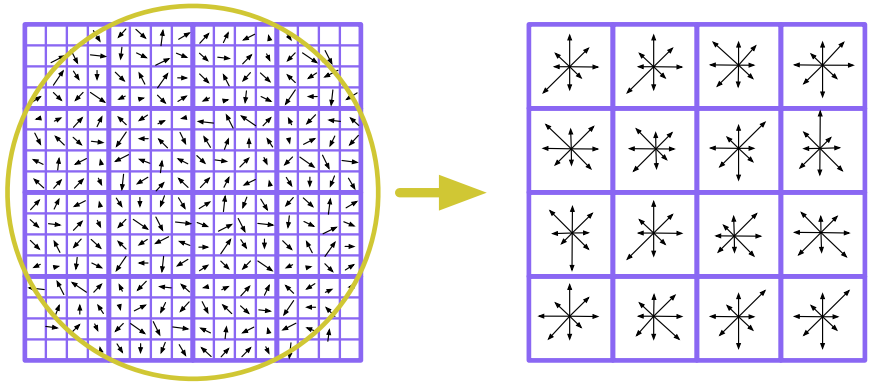


Abb. 5.12: Bestimmung des Merkmalsdeskriptors
Der Deskriptor wird aus den lokalen Gradienten um den Merkmalspunkt berechnet (links). Dazu wird der Gradientenbereich in 4×4 Unterbereiche aufgeteilt, für die dann je ein Histogramm mit acht Hauptrichtungen berechnet wird. Dies ergibt einen Deskriptor mit $4 \times 4 \times 8 = 128$ *Elementen. Quelle: [Hey05]*

daher nicht im Detail nachvollziehen.

Erstellen der Deskriptoren Der vorhergehende Schritt erzeugt im Bildpuffer einen kompletten Merkmalsdeskriptor. Für die spätere Verwendung wird dieser ausgelesen und gespeichert. Dieser Vorgang wird für alle weiteren Merkmalspunkte wiederholt.

Die gewonnenen Merkmalsdeskriptoren können jetzt zum Bestimmen von Übereinstimmungen in unterschiedlichen Bildern verwendet werden. Hierzu wird der euklidische Abstand $D = \sqrt{\sum_{i=1}^{N} (V_{1i} - V_{2i})^2}$ zweier Merkmalsdeskriptoren berechnet. Dies kann laut Heymann effizient auf der GPU durch parallele Subtraktion realisiert werden. Dessen ungeachtet erzielt Heymann jedoch bei einem Leistungsvergleich mit dem von Lowe vorgeschlagenen *Best-Bin-First*-Verfahren (siehe Unterabschnitt 3.3.6) auf

der CPU schnellere Ergebnisse (gemessen am Vergleich zweier 2500-elementiger Vektormengen), selbst wenn man den Datentransfer berücksichtigt.

Ergebnisse

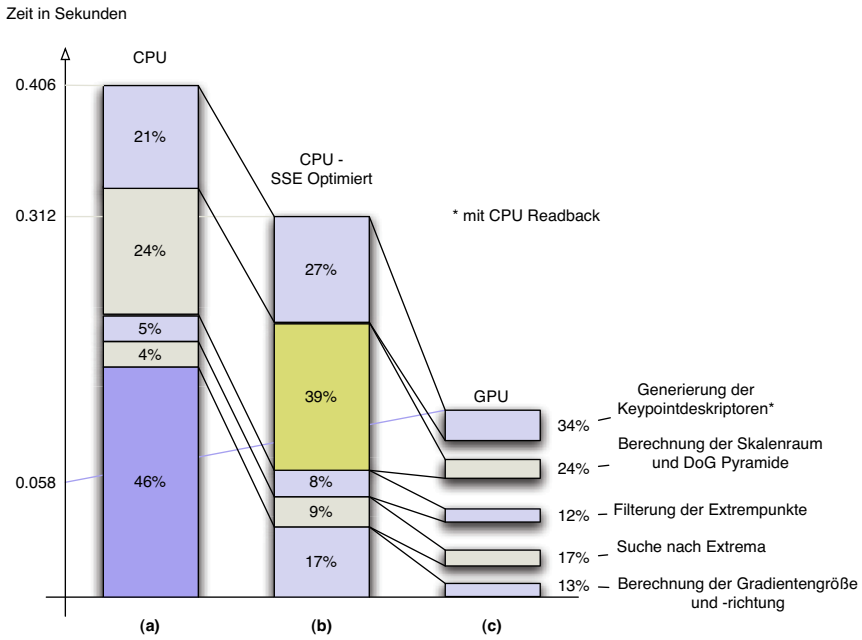


Abb. 5.13: Gegenüberstellung der Berechnungszeiten von SIFT-Implementierungen;
Die Zeiten wurden bei einem Eingangsbild mit 320×240 Bildpunkten ermittelt. Grafik entnommen aus [Hey05]

Die GPU-Implementierung wurde von Heymann getestet und mit der originalen sowie einer handoptimierten CPU-Version verglichen. Abbildung 5.13 zeigt die Ergebnisse dieses Vergleichs. Laut Heymann benötigt die optimierte CPU-Version 0,312 Sekunden und die originale 0,406 Sekunden für einen Durchgang bei einer Auflösung von 320×240 Bildpunkten. Im Vergleich dazu konnte der SIFT-Algorithmus durch Ausnutzen der parallelen Verarbeitung auf der GPU drastisch beschleunigt werden: ein Durchlauf dauert hier gerade einmal 0,058 Sekunden. Die SIFT-Merkmalsextraktion kann mit dieser Implementierung mit knapp 20 Bildern pro Sekunde fast in Echtzeit ablaufen. Da die Tests nur auf einer GeForce6-Serie durchgeführt wurden, ist mit aktuellerer Grafikhardware eine Echtzeitfähigkeit selbst in höheren Auflösungen denkbar. Über die Anzahl der dabei erkannten Merkmale wird bei Heymann jedoch keine Aussage getroffen. Die Vergleiche der Merkmalsdeskriptoren lassen sich laut Heymann bei ca. 450 Merkmalen in einer Zeit von 0,019 Sekunden erreichen.

In der Implementierung I_3 wurden alle Schritte bis zum Berechnen der Deskriptoren

umgesetzt. Die unvollendete und nicht optimierte Version erreicht bei einem Video mit einer Auflösung von 320×240 Pixeln auf einer GeForce 6800 zwischen sieben und acht Bildern pro Sekunde.

Beruft man sich lediglich auf die Informationen, die in Heymanns Arbeit stehen, kann seine GPU-Implementierung im Vergleich zu Referenzimplementierungen auf der CPU, im Bezug auf die Robustheit der gefundenen Merkmale, sicherlich nicht mithalten. Allein durch das Weglassen der Verdopplung des Eingangsbildes wird nur ein Viertel der eigentlich verfügbaren robusten Merkmale genutzt wird. Des Weiteren wird nur eine Hauptrichtung zu jedem Merkmalspunkt bestimmt. Mehrere Hauptrichtungen tragen aber laut Lowe [Low04] maßgeblich zur Stabilität der SIFT-Merkmale bei. Auch über die verwendeten Texturformate wird bei Heymann keine Aussage getroffen.

Bei der I_3 Implementierung wurde dagegen darauf geachtet, dass sie prinzipiell dieselben Ergebnisse liefert wie die Referenzimplementierung von Vedaldi. Aufgrund des Fehlens einer effizienten Histogrammberechnung sind jedoch kaum bessere Laufzeiten als bei einer CPU-Version für die fertig implementierte Umsetzung zu erwarten. Auch geht I_3 nicht gerade sparsam mit dem vorhandenen Texturspeicher um, sodass mit 128 MB Grafikspeicher eine Auflösung von 800×600 Bildpunkten nicht mehr verarbeitet werden kann. Da aber aktuellen Grafikkarten bereits standardmäßig mit 256 MB und mehr Speicher erhältlich sind, fällt dieses Manko nicht allzu stark ins Gewicht.

Ob die Genauigkeit von Heymanns GPU-Version beim Bestimmen der SIFT-Merkmale in Wirklichkeit mit der CPU-Version vergleichbar ist, oder ob die erzielte Laufzeit auf das Weglassen einiger Schritte zurückzuführen ist, bleibt hingegen offen.

5.4.2 GPU-SIFT Implementierung von Sinha

Die folgende Beschreibung basiert größtenteils auf den Angaben, die Sudipta Sinha in [SFPG06] zu seiner Implementierung gemacht hat. Hier wird nur ein kurzer Einblick in diese Implementierung gegeben und für Details auf die Veröffentlichung von Sinha verwiesen.

Implementierungsdetails

Erstellen des gaußschen Skalenraums Die Erstellung der gaußschen Skalenraumpyramide wird durch Fragmentprogramme auf der GPU beschleunigt. Es wird dabei wieder eine zweistufige Gaußfaltung verwendet, um Texturzugriffe zu sparen. Die Helligkeitswerte des Eingangsbildes sowie die Gradienten und die DoG-Werte werden zusammen von einem Fragmentprogramm erzeugt und in einer `RGBA`-Textur gespeichert, wobei jeder Wert in einem eigenen Farbkanal abgelegt wird.

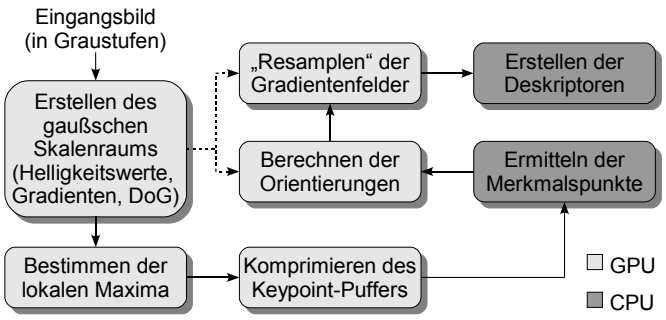


Abb. 5.14: Ablauf der GPU-SIFT Implementierung von Sinha

Bestimmen der lokalen Maxima Die Blendoperationen der Grafikhardware werden dafür eingesetzt, um die lokalen Extrema in den jeweiligen DoG-Pyramidenebenen gleichzeitig zu bestimmen. Durch den Tiefen- und Alphatest kann eine Schwellwertabfrage auf den zuvor bestimmten Extremwerten durchgeführt werden. Um das Krümmungsverhältnis in den Extrempunkten zu untersuchen, wird das Verhältnis der Eigenvektoren der 2×2 Strukturtensor-Matrix in diesem Punkt berechnet.

Komprimieren des Keypoint-Puffers und Ermitteln der Merkmalspunkte Die Positionen der einzelnen Schlüsselpunkte werden implizit in binären Puffern berechnet, welche dieselbe Größe wie das jeweilige Bild der Skalenebene haben. Diese Binärbilder werden dann mit Hilfe eines Fragmentprogrammes auf $\frac{1}{32}$ ihrer Größe in `RGBA`-Daten gepackt. Die gepackten Daten werden dann auf die CPU zurückgelesen und dort dekodiert. Aus den entpackten Daten wird dann eine Liste mit den Merkmalspunkten sowie ihren Skalenwerten erstellt.

Berechnen der Orientierungen Da das Zurücklesen der Gradientendaten von der GPU zur CPU teuer ist, werden die folgenden Schritte laut Sinha auch auf der GPU ausgeführt. Als erstes werden die Gradientenvektoren um die Schlüsselstelle herum durch eine Gaußkurve gewichtet und anschließend durch ein anderes Fragmentprogramm in Orientierungshistogrammen ausgewertet. Nachdem das Histogramm zurückgelesen wurde, können seine Spitzen auf der CPU bestimmt werden. Da, wie schon beim OpenVIDIA Feature-Tracker auf Seite 60 erklärt, die Histogrammberechnung auf der GPU ziemlich aufwendig ist, entscheidet sich Sinha, die Histogrammdaten zurückzulesen und auf der CPU auszuwerten. Dies scheint in seinem Fall etwas schneller zu sein, muss aber nicht für neuere Grafikhardware gelten, da die Leistung der Shaderprozessoren im Vergleich zur reinen Transferleistung unverhältnismäßig stark zunimmt (siehe Kapitel 2, S. 4 sowie Unterabschnitt 4.1.5, S. 35).

Resampeln der Gradientenfelder und Erstellen der Deskriptoren Im letzten Schritt werden die 128-elementigen SIFT-Deskriptoren der Merkmalspunkte berechnet. Die SIFT-Deskriptoren können nach [SFPG06] nicht effizient vollständig auf der GPU berechnet werden, da die Histogrammeinträge noch angeglichen werden müssen, um das Quantisierungsrauschen, das beim Aufteilen auf die diskreten Werte entsteht, zu entfernen. Eine Möglichkeit dies zu tun ist allerdings im OpenVIDIA Feature-Tracker umgesetzt (Abschnitt 5.3). Sinha entschied sich, diesen Schritt unter CPU und GPU aufzuteilen. Dafür wird für das Gradientenfeld um jeden Merkmalspunkt herum „resampelt“ und anschließend, unter Einsatz der Blendeinheit der GPU, mit einer Gaußmaske geglättet.

Die neu abgetasteten und gewichteten Gradientenvektoren werden in einem gekachelten Texturblock gesammelt und im Folgenden zurück auf die CPU übertragen, wo sie zum Ermitteln der Deskriptoren genutzt werden [SFPG06]. Diese GPU-CPU-Aufteilung wurde vorgenommen, um die Datenmenge beim Zurücklesen möglichst gering zu halten und außerdem weil ein kompletter Transfer der Gradientenpyramide zur CPU unpraktikabel wäre [SFPG06]. Textur-Resampling und Blending dagegen, sind auf der GPU effizient implementiert. Ein weiterer Vorteil besteht darin, dass man so einen kompakten gekachelten Texturblock erhält, der auf einmal zur CPU zurückgelesen werden kann.

Ergebnisse

Laut Sinha erzielt seine GPU-SIFT Implementierung einen großen Geschwindigkeitsvorteil beim Aufbau der gaußschen Skalenpyramide und bei den Schritten zur Bestimmung der Merkmalspunkte. Das Zurücklesen mit komprimierten Daten reduziert die erforderliche Datenmenge bei der Übertragung dabei um den Faktor 32. Die Berechnung der Merkmalsorientierung sowie der Deskriptoren ist so zwischen CPU und GPU aufgeteilt, dass der Datentransfer minimal ist. Alles in allem erreicht Sinhas GPU-SIFT Implementierung, verglichen mit der CPU-Version, einen etwa 10-fachen Geschwindigkeitszuwachs auf einer GeForce 7900 GTX [SFPG06, S. 11].

GPU-SIFT wurde in OpenGL/Cg implementiert und dabei PBuffer für das Off-Screen-Rendering benutzt. Framebuffer-Objekte (siehe Abschnitt 4.5, S. 44) wären hier jedoch eine effizientere, einfachere und vor allem wesentlich flexiblere Alternative gewesen. Sinha vergleicht in [SFPG06] seine Implementierung in verschiedenen Tests auf einer NVIDIA Geforce 7900GTX mit einer CPU-Version. Aus einem Video mit einer Auflösung von 640×480 Bildpunkten kann er, bei durchschnittlich 10 Bildern pro Sekunde, etwa 1000 SIFT-Merkmale extrahieren. Sinha weist darauf hin, dass bei seiner Implementierung die meiste Zeit für den Aufbau der Skalenraumpyramide (GPU) und

für die Berechnung der Deskriptoren (CPU) verloren geht. Bei großen Auflösungen des Eingangsmaterials ($\geq$1024) benötigt daher die GPU die meiste Zeit; werden hingegen viele Merkmale ($\geq$200) betrachtet, benötigt die CPU die meiste Zeit.

Wie Sinha in einer E-Mail-Antwort [SM] an den Autor einräumte, liefere seine GPU-Implementierung etwas ungenauere Ergebnisse als die Referenzimplementierung von Lowe. Dies solle zum einen an der verwendeten näherungsweisen Gaußfaltung und zum anderen daran liegen, dass einige Schritte zur Verfeinerung der Resultate in der Implementierung übersprungen würden. Darunter fallen beispielsweise die subpixelgenaue Bestimmung der Extremwerte der DoG-Ebenen und die Verfeinerung der Orientierungshistogramme durch quadratische Interpolation der diskreten diskreten Histogrammwerte.

Sobald geeignete, effiziente Verfahren zur Berechnung von Histogrammen auf der GPU entwickelt worden sind, werden sich einige wesentliche Berechnungsschritte vereinfachen und dadurch Laufzeit einsparen. Dann werden auch die Datenübertragungen zwischen CPU und GPU wegfallen, die einen nicht unerheblichen Teil der Zeit benötigen. Auch durch den Einsatz aktueller Erweiterungen lassen sich bestimmte Teile des Codes mit Sicherheit optimieren.

5.4.3 Ergebnisse der SIFT-Implementierungen

In diesem Abschnitt wurden verschiedene Implementierungen des SIFT-Algorithmus untersucht. Es wurde gezeigt, dass unter Ausnutzung von programmierbarer Grafikhardware die Berechnung von SIFT-Merkmalen in niedrigen auflösungen fast echtzeitnah ausgeführt werden kann. Dies entspricht einem Leistungsunterschied gegenüber der CPU-Implementierung um ca. das Siebenfache. Bei den Umsetzungen wird eine mit CPU-Implementierungen vergleichbar hohe Genauigkeit angestrebt, aber noch nicht ganz erreicht. Das liegt unter anderem an einigen Vereinfachungen, die die Umsetzung auf der GPU erleichtern oder aber aufgrund von Hardwarebeschränkungen erforderlich waren. Ziel für zukünftige Versionen sollte daher auf alle Fälle auch eine gleichwertige Genauigkeit bei der Berechnung der Merkmalsbeschreibungen sein. Probleme bei der Umsetzung bereitet vor allem die Histogrammbildung zur Auswertung der lokalen Gradientenrichtungen. Hierfür gilt, es noch effiziente Methoden zu finden, da auch das Auslagern des Problems auf die CPU, wie dies in Sinhas Implementierung der Fall ist, durch den zusätzlichen Datentransfer nicht wirklich effizient ist.

Die betrachteten Implementierungen liefern nur Aussagen bezüglich der Extraktion von Merkmalen, das Tracking hingegen, wird nur am Rand erwähnt. Der Grund hierfür ist, dass bei den untersuchten Verfahren die Vergleiche zwischen den Merkmalsdeskriptoren auf der CPU deutlich schneller ausgeführt werden können als auf

der GPU [Hey05]. Hier wäre es beispielsweise wünschenswert, auf eine GPU-Implementierung des von Lowe vorgeschlagenen *Best-Bin-First*-Verfahrens zurückgreifen zu können, um die Deskriptoren direkt auf der GPU effizient vergleichen zu können.

5.5 Hürden bei der Umsetzung

Wie bereits eingangs erwähnt, lassen sich Algorithmen des Maschinellen Sehens aufgrund ihres grafiklastigen Kontextes prinzipiell gut auf der GPU umsetzen. Dabei gibt es jedoch einige Problemstellen, die eine effiziente Implementierung erschweren und daher kurz erläutert werden sollten. Hierfür wird im Folgenden eine Trennung zwischen allgemeinen Hürden bei der Programmierung der GPU und Hürden, die bei der Portierung von Algorithmen der Merkmalsverfolgung auf die GPU zu bewältigen sind, vorgenommen.

5.5.1 Hürden bei der GPU-Programmierung

Die Entwicklung von GPU-Anwendungen gestaltet sich aufgrund mehrerer Punkte schwierig. So existieren beispielsweise kaum Debugging-Werkzeuge, die Funktionen bieten, wie man sie von den interaktiven CPU-Debuggern kennt (Breakpoints, schrittweise Ausführung, Variablenüberwachung, etc.). Zwar stellen die GPU-Hersteller Werkzeuge zur Performanceanalyse bereit, diese erfordern allerdings ein hohes Maß an Einarbeitung und eignen sich nur bedingt für das Debugging von GPGPU-Anwendungen.

Eine andere Hürde auf dem Weg zur effizienten Implementierung ist das im Vergleich zu CPU deutlich komplexere parallele Programmiermodell und die zugrunde liegende Grafikarchitektur, auf der sich einige Schritte nur mit beträchtlichem Aufwand realisiert lassen. Neben den programmierbaren Shadereinheiten lassen sich auch die Stufen der *Fixed-Function*-Pipeline zur Umsetzung von Teilschritten nutzen. Dadurch ergeben sich für die Realisierung auf der GPU mehrere Möglichkeiten, die berücksichtigt werden sollten. Ferner kann es passieren, dass sich eine ursprünglich effiziente Lösung, im Zusammenspiel mit den folgenden Schritten als ungünstig erweist. Daher ist es unbedingt erforderlich die Anwendungsumsetzung von Anfang an sinnvoll zu planen und sich genau zu überlegen, zu welchem Zeitpunkt welche Daten benötigt werden und in welchem Format diese vorliegen müssen. Weiterhin hat sich bewährt, die benötigte Zeit der jeweiligen Teilschritte von Beginn an genau zu messen und auszuwerten. Manchmal kann es zudem günstiger sein, gewisse Teile auf der CPU zu berechnen. Dabei muss jedoch der Transfer zwischen GPU und CPU berücksichtigt werden. Welche Umsetzung für einen Teilschritt am effizientesten ist, lässt sich aufgrund mannigfaltiger Faktoren, wie GPU-Serie, Kern- und Speichertakt der GPU, Version des Grafiktreibers, hardwa-

reinterne Umsetzung einer OpenGL-Erweiterung, Speicheranbindung, etc. nur selten korrekt abschätzen. Deswegen müssen oft mehrere Lösungen ausprobiert und deren Ergebnisse gegeneinander abgewogen werden.

5.5.2 Algorithmische Hürden

Nicht alle Teile eines Algorithmus aus der Bildverarbeitung lassen sich ohne Weiteres auf der GPU umsetzen. Dies liegt zum einen daran, dass diese Algorithmen primär für eine serielle Abarbeitung der Daten auf der CPU entworfen wurden und daher das parallele Programmiermodell der GPU nicht berücksichtigen. Andererseits ergeben sich durch dieses parallele Programmiermodell sowie durch die Grafikarchitektur selbst Einschränkungen, die zu Problemen bei der Implementierung führen. Die Hauptprobleme bei der Umsetzung der vorgestellten Algorithmen waren:

- **1:n-Abbildungen:** Das Erzeugen von mehreren Elementen aus einem, wie es beispielsweise bei der Berechnung der Merkmalsdeskriptoren bei den SIFT-Implementierungen erforderlich ist, ist in der Grafikpipeline nicht direkt vorgesehen. Dies kann allerdings erreicht werden, indem beispielsweise der Punkt durch Zeichnen einer Linie aus n Punkten „ver-n-facht“ wird. Diese muss dann so texturiert werden, dass jeder Punkt auf ihr denselben Wert, nämlich den Eingabewert, zugewiesen bekommt.

- **Histogrammbildung:** Die Erstellung von Histogrammen gestaltet sich auf der GPU äußerst schwierig, da die Eingabedaten passend in den Histogrammbehältern verteilt werden müssen und dies mit konditionaler Auswertung praktisch nicht möglich ist. Auch eine modulo-basierte Berechnung des Behälterindexes, wie man sie bei CPU-Implementierungen häufig verwendet, lässt sich aufgrund fehlender Ganzzahltypen nicht realisieren. Eine mögliche Umsetzung lässt sich, wie in Abschnitt 5.3 beschrieben, durch den Einsatz der Kosinusfunktion realisieren (siehe S. 60).

- **Reduktion m → n:** Eine Reduktion von m nach n Elementen, wobei n deutlich kleiner als m ist, wie beispielsweise die ausgewerteten Extremstellen eines Eckendetektors, ist auf der GPU mit bekannten Mitteln nahezu nicht umzusetzen. Dies liegt daran, dass die Extrempunkte nicht wie bei einer normalen Reduktion (siehe Abschnitt 4.3) geordnet, sondern zufällig angeordnet sind. Da die Zieladresse eines Fragmentprogrammes fix ist, kann auch kein Ergebnispuffer in Abhängigkeit von den Extremstellen gefüllt werden. Da es allerdings nicht ratsam ist, mit dem vollständigen Bild weiter zu arbeiten, werden die Extremwerte auf der CPU aus der Textur extrahiert. Um den Datentransfer hierbei möglichst klein zu halten,

können die Daten beispielsweise durch Binarisierung wie bei Sinha (Unterabschnitt 5.4.2) oder durch geschicktes Ausnutzen des Schablonenpuffers, wie im OpenVIDIA Projekt (Abschnitt 5.3) reduziert werden.

5.6 Optimierungsstrategien

Neben der Implementierung eines Algorithmus ist die Optimierung ein entscheidender Teil der Anwendungsentwicklung, unabhängig davon, ob für die CPU oder für die GPU.

Im Folgenden werden daher einige Methoden vorgestellt, die für eine effiziente Umsetzung eines Verfahrens auf der GPU berücksichtigt werden sollten. Weiterführende detaillierte Ausführungen sind im *GPU Programming Guide* von NVIDIA zu finden [NVI05b].

Engpässe ausfindig machen

Um ein Programm sinnvoll zu optimieren, gilt es als Erstes die Programmteile zu bestimmen, die am meisten Zeit in Anspruch nehmen, da hier der Gewinn in der Regel am höchsten ausfällt. Ein Engpass kann jedoch viele Ursachen haben:

- GPU (Kerntakt, Speichertakt, Speicherzugriffszeit, Anzahl Shadereinheiten, Genauigkeit, Caches)
- CPU (Takt, Anzahl Kerne, Caches, FPU)
- Zugriffszeit und Transferleistung des Hauptspeichers
- Transferleistung des Bussystems (PCI, AGP-X4/X8, PCIe-X8/X16)
- Betriebssystem (Hintergrundprozesse)
- Grafikschnittstelle (Direct3D, OpenGL, Cg, ...)
- Grafikkartentreiber
- Programmcode/Progammdesign

Ein gängiger Weg den „Flaschenhals“ zu finden, ist beispielsweise die Auslastung der GPU zu betrachten. Dies kann relativ einfach mit Werkzeugen wie dem NVPerfKit von NVIDIA [NVI07b] und dem gDEBugger von Graphic Remedy [Gra06] ausgewertet werden. Ist die GPU häufig nicht voll ausgelastet, spricht dies oft für eine Beschränkung durch die CPU. Die Leistung der GPU zu verbessern nützt dann nichts. Ein weiteres probates Mittel um Schwachstellen ausfindig zu machen, ist die Ausführung des Programms auf verschieden starken CPUs bzw. GPUs.

Methoden zur Performancesteigerung

Neben Standardoptimierungen, wie man sie auch von der CPU kennt (algebraische Umformungen, Konstantenfaltung, etc.), gilt es bei der Programmierung der GPU vor allem, die technischen Möglichkeiten der zugrunde liegenden Hardware möglichst gut auszuschöpfen. Moderne GPUs verfügen über eine Vielzahl von spezialisierten Einheiten (Vertex-/Pixelshader, Blendeinheit, Rastereinheit, Vektoreinheiten, etc.) sowie über etliche in Hardware realisierte Operationen (bilinerare Filterung, Mipmapping, Unterstützung bestimmter Texturformate, Rechenoperationen, Antialiasing, etc.). Viele dieser Operationen und Einheiten kosten keine zusätzliche Rechenleistung, da sie ohnehin in der Renderpipeline vorgesehen sind. Sie werden deswegen auch als atomare Operationen bezeichnet.

Im Folgenden werden einige grundsätzliche Methoden zur Geschwindigkeitssteigerung von GPU-Programmen aufgelistet, die man bereits bei der Implementierung beachten sollte:

Half statt Float Eine einfache Methode zum Performancegewinn ist der Einsatz von 16-bittigen Fließkommawerten, anstatt der vollen einfachen Genauigkeit nach IEEE-754. Dies bringt zum einen einen Vorteil bei der Berechnung, sofern der `half`-Typ überhaupt von der Grafikhardware unterstützt wird, und zum anderen reduziert es die Datenmenge für die Übertragung der Texturen beim Zurücklesen (Readback) auf die Hälfte. Es muss selbstverständlich abgewogen werden, ob die verbleibende Genauigkeit für die jeweiligen Zwecke ausreicht (siehe Unterabschnitt 4.1.4).

Wahl des korrekten Texturformats Bilder mit 8-Bit Farbinformation pro Kanal sind unter Windows standardmäßig im `BGRA`-Format (GDI[5] Pixellayout) im Speicher abzulegen. Da Grafikhardware diesen Umstand berücksichtigt, muss der Treiber gegebenenfalls die Kanäle der Pixeldaten beim Transfer auf die Grafikkarte vertauschen (`RGBA`→`BGRA`). Der Tabelle in [Elh05] nach, kann das die Transferrate auf einigen Modellen um bis zu über 70 Prozent reduzieren. Bei 16 oder 32-Bit Fließkommaformaten (z.B. `GL_RGBA16F_ARB`) ist dies allerdings nicht der Fall.

Früher Tiefentest (Z-Cull , Early-Z) Durch den Tiefentest können verdeckte Objekte vom Zeichenprozess ausgeschlossen werden. Dies kann dazu benutzt werden, um etwa Berechnungen auf bestimmte Bereiche einer Textur einzuschränken, beziehungsweise, um bestimmte Bereiche von Berechnungen auszuschließen. Ein ähnliches Verfahren kann auch mit dem Schablonenpuffer angewendet werden.

Nutzen der Verdeckungsabfrage Eine andere Technik um das Zeichnen von nicht

[5] Micosoft Graphics Device Interface, http://en.wikipedia.org/wiki/Graphics_Device_Interface

Sichtbarem zu vermeiden, ist die Hardware-Verdeckungsabfrage (*occlusion query*). Diese Technik ermöglicht die Abfrage der Anzahl der Bildpunkte, die bei einem Renderaufruf aktualisiert werden. Diese Abfragen sind in der Rendering-Pipeline vorgesehen, was bedeutet, dass sie eine eingeschränkte Möglichkeit bieten, einen Wert von der GPU zu erhalten, ohne die Pipeline zu blockieren (was durch Zurücklesen der eigentlichen Bildpunkte passieren würde) [OLG+07].

Z-Only und Stencil Rendering GeForce Grafikkarten arbeiten doppelt so schnell, wenn die zu zeichnenden Daten auf den Schablonen- oder Tiefenpuffer beschränkt sind. Nähere Ausführungen hierzu sind in [NVI05b, S. 29] zu finden.

Vorziehen von linearen Berechnungen aus dem Fragmentshader Redundante lineare Berechnungen, wie etwa die Berechnung von Texturkoordinaten, sollten nicht für jedes Fragment ausgeführt werden, sofern dies nicht notwendig ist. Effizienter ist es, die Berechnungen nur für die Eckpunkte durchzuführen und diese von der Rastereinheit interpolieren zu lassen. In OpenGL lassen sich mit `glMultiTexCoord` beispielsweise acht Koordinatenpaare pro Eckpunkt für Texturen definieren, auf die dann direkt im Fragmentprogramm zugegriffen werden kann. Kompliziertere Berechnungen lassen sich auch direkt im Vertexshader durchführen.

Verwenden von Display-Listen Durch OpenGLs Display-Listen werden OpenGL-Aufrufe in der Grafikkarte gespeichert. Dies kann bei größeren Szenen helfen, den CPU-Overhead zu verringern.

Frühe Flusskontrolle Explizite Sprünge auf der GPU können die Performance beeinträchtigen. Daher ist es sinnvoll, Entscheidungen über die Flusskontrolle in der Pipeline nach oben (frühere Stufe) zu schieben, um eine effizientere Auswertung machen zu können. Dies kann beispielsweise durch einen frühen Tiefentest oder durch Vorberechnen gleichbleibender Werte erfolgen.

5.7 Zusammenfassung

In diesem Kapitel wurde gezeigt, dass sich gängige Verfahren zur Verfolgung von Merkmalen auf der GPU effizient implementieren lassen. Durch Ausnutzen der parallelen Grafikarchitektur zur gleichzeitigen Verarbeitung der Bilddaten erzielten die GPU-Umsetzungen dabei nicht nur fast dieselbe Genauigkeit wie ihre CPU-Pendants, sondern waren obendrein auch noch um eine Größenordnung schneller (bis zu Faktor zehn und mehr). Dabei wurden sowohl die programmierbaren als auch die feststehenden Stufen der Grafikpipeline genutzt, um einzelne Teilschritte zu realisieren.

Neben der, im Detail analysierten, effizienten Implementierung des Eckendetektors von Harris und Stephens (Unterabschnitt 3.3.4), wurde eine Umsetzung des KLT-Trackers (Unterabschnitt 3.3.5), der OpenVIDIA Feature-Tracker sowie verschiedene SIFT-Implementierungen (Unterabschnitt 3.3.6) untersucht. Hierbei wurden verschiedene Verfahren zur Umsetzung der Erkennung und Extraktion von Merkmalspunkten auf der GPU ausführlich dargestellt und auf Problemstellen, wie etwa der effizienten Gewinnung der über eine Textur zufällig verteilten Extremwerte, hingewiesen und Lösungen angeboten. Mit dem vollständig auf der GPU ablaufenden KLT-Tracker und dem OpenVIDIA Feature-Tracker wurden zudem zwei, vom Ansatz her verschiedene, Tracking-Methoden vorgestellt, die auch auf unterschiedlichen Verfahren zur Merkmalsgewinnung aufbauen. Des Weiteren wurden verschiedene Strategien diskutiert, die zeigen, wie sich für GPU-Programme eine möglichst optimale Performance erreichen lässt.

Die durchgeführten Leistungsmessungen haben gezeigt, dass sich mit dem KLT-Tracker eine Merkmalsverfolgung auf der GPU realisieren lässt, die bei Standard-SVGA-Auflösung (800×600 Pixel) mit 1000 erkannten Merkmalen in Echtzeit abläuft. Die Implementierung des Harris-Eckendetektors erreicht sogar auf älteren GPUs bei einer Auflösung von 1024×768 echtzeitfähige Bildwiederholraten. Der SIFT-ähnliche OpenVIDIA Feature-Tracker verfügt zwar über eine optimierte und schnelle Merkmalsextraktion, durch seinen langsamen Trackingvorgang wird jedoch die Gesamtleistung so stark gedrosselt, dass selbst bei 512×384 Bildpunkten keine Echtzeitfähigkeit erreicht wird. Die SIFT-Verfahren eignen sich aufgrund ihrer Komplexität lediglich bei niedrigen Auflösungen zur Merkmalsextraktion in Echtzeit. Dies ist jedoch immer noch ein gewaltiger Fortschritt gegenüber den CPU-Versionen, die ein Vielfaches der Zeit für die Berechnungen benötigen.

Die GPU-Implementierungen ermöglichen zwar das Erkennen oder Verfolgen von Merkmalen in hochaufgelöstem Bildmaterial in Echtzeit, sind aber in manchen Punkten noch verbesserungswürdig: einerseits erreicht deren Genauigkeit nicht ganz das Niveau der CPU-Versionen und andererseits fehlen noch effiziente Verfahren zur Umsetzung einiger Teilschritte, wie etwa die Histogrammbildung bei den SIFT-Verfahren. Auch die Tracking-Methoden der SIFT-Verfahren lassen sich auf höheren Auflösungen noch nicht in Echtzeit nutzen.

6 Schlussbemerkungen und Ausblick

In der vorliegenden Arbeit wurde untersucht, wie gut sich Verfahren zur Echtzeit-Bildverarbeitung und der Merkmalsverfolgung im Speziellen auf Graphics Processing Units umsetzen lassen. Eine weitere Fragestellung war, ob die für Anwendungen der Erweiterten Realität geforderte Echtzeit in diesen Verfahren erreicht werden kann. Hierzu wurden verschiedene Methoden zur Extraktion und zum Verfolgen von Merkmalen in Videobildern erläutert und die technischen Hintergründe der GPU-Programmierung ausführlich dargestellt. Die anschließend untersuchten Implementierungen erreichten im Schnitt etwa die zehnfache Leistung vergleichbarer CPU-Implementierungen, bei nur geringen Einbußen im Bezug auf die Genauigkeit. Fast alle beschriebenen Umsetzungen ermöglichen den Einsatz in Echtzeit-Anwendungen und können dabei sogar mit der Verarbeitung von hochaufgelöstem Bildmaterial punkten. Lediglich die Implementierungen des SIFT-Merkmalsextraktors erreichten infolge der Komplexität des Verfahrens noch keine echtzeitfähigen Ausführungszeiten. Die erreichten Werte sind jedoch immer noch um ein Vielfaches besser als selbst handoptimierte CPU-Versionen.

Diese Arbeit zeigt, dass moderne GPUs auch für nichtgrafische Anwendungen ein enormes Leistungspotential bieten, aber deren Programmierung immer noch ein detailliertes Wissen der zugrundeliegenden parallelen Architektur erfordert. Eigene Compiler und Laufzeitumgebungen, wie beispielsweise Brook [Sta06], erlauben es, die Grafikkarte wie einen Streamprozessor zu nutzen, ohne dabei tieferes Wissen der Grafikarchitektur zu erfordern. Auch die GPU-Hersteller bieten mittlerweile mit CUDA (*Computer Unified Device Architecture*, NVIDIA [NVI07a]) und CTM (*Close To Metal*, AMD [Adv06]) Techniken, mit denen ihre neuesten GPU-Modelle direkt als Streamprozessor verwendet werden können.

Da eine quantitative Auswertung der umgesetzten Verfahren in dieser Arbeit aufgrund der unerwartet hohen Komplexität der Umsetzungen nicht möglich war, wäre es wünschenswert diesen Punkt in einer weiteren Arbeit zu untersuchen. Neben allgemeinen Verbesserungen im Bezug auf Genauigkeit und Geschwindigkeit, sind ferner folgende Fragestellungen für zusätzliche Untersuchungen interessant:

1. Inwieweit lassen sich die beschriebenen Implementierungen durch Techniken wie *Pixelbuffer-Objekte* [GPGb], die *stream compaction* von Horn [Hor05] oder die

Histogramm-Pyramiden von Ziegler [ZTTS06] weiter beschleunigen bzw. vorhandene Problemstellen beseitigen?

2. Wie wirkt sich die etwas geringere Genauigkeit der Implementierungen auf die Robustheit der erkannten Merkmale aus?
3. Inwiefern lassen sich die GPU-Implementierungen von der CPU abkapseln, so dass diese wieder vollständig für andere Aufgaben zur Verfügung steht?
4. Welche Vorteile ergeben sich für die Implementierung durch neue Techniken, wie CUDA und der Unified-Shader-Architektur? Sind damit automatisch ähnlich effiziente Umsetzungen wie von Hand möglich?
5. Aufgrund der Ausführung von Berechnungen auf der GPU kann diese nicht mehr zum Darstellen von 3D-Objekten verwendet werden. Diese sind jedoch wesentlicher Bestandteil von Anwendungen der Erweiterten Realität. Kann dieses Problem sinnvoll durch den Einsatz mehrerer Grafikkarten gelöst werden?
6. Quantitative Analyse: Wie verhalten sich einzelne Teile der Verfahren auf der GPU? Wie können diese sinnvoll optimiert werden? Lassen sich diese zweckmäßig mit anderen Teilen zu effizienten Lösungen kombinieren?

Alles in allem haben sich GPUs als ideale Plattform zur parallelen Verarbeitung von Bilddaten in Echtzeit und als günstige Alternative zu teurer Spezialhardware erwiesen.

Aktuelle GPUs bieten mit ihren parallelen Einheiten enorme Rechenleistung, die für immer mehr Anwendungsgebiete genutzt werden kann. Aktuelle Entwicklungen zeigen, dass in Zukunft mehrfädige Prozessoren an Bedeutung gewinnen: schon heute sind die Hersteller von CPUs durch physikalische Grenzen dazu gezwungen, statt immer höheren Taktraten auf mehrere Prozessorkerne auszuweichen, um eine höhere Leistung zu erzielen. Daher wird die Parallelisierung von Anwendungen auch in anderen Bereichen eine wichtige Rolle spielen. Dazu ist ein generelles Umdenken bei der Programmierung und Entwicklung von Algorithmen und Anwendungen erforderlich, da neben dem bisherigen seriellen Programmiermodell zukünftig auch die parallelen Ausführungseinheiten moderner Architekturen berücksichtigt werden müssen.

A Anhang

A.1 Testsystem

Bei den Leistungsmessungen wurde auf folgendes System zurückgegriffen:

CPU	GPUs
Athlon64 3800+	NVIDIA GeForce 6800 (AGP) mit 128MB,
und 2GB RAM	NVIDIA GeForce 7800 GT (PCIe) mit 256MB,
	NVIDIA GeForce 7900 GT (PCIe) mit 256MB

Alle Implementierungen mussten Videos in verschiedenen Auflösungsstufen (320×240 bis 1024×768) durchlaufen. Hierfür wurde ein Testvideo (Gartenhaus mit Sträuchern) aufgenommen, das enorm viele markante Punkte enthält (Blätter des Busches). Die jeweiligen Programme wurden dabei um eine eigens programmierte Routine ergänzt, die eine effiziente Zeitmessung ermöglichte.

Abb. A.1: Testszene

A.2 Technische Daten heutiger GPUs

Tab. A.1: Technische Daten heutiger GPUs; Quellen: [Stu06], [And07]

Typ	GeForce 6800	Radeon X850 XT	GeForce 7900 GTX
Hersteller	NVIDIA	AMD (ATI)	NVIDIA
GPU-Kern	NV40	R480	G71
Transistoren	222 Mio.	160 Mio.	278 Mio.
Fertigungsprozess	130 nm	130 nm	90 nm
Chiptakt	325 Mhz	540 Mhz	650 MHz
Speichertakt	350 Mhz	590 MHz	800 MHz
Speicherausbau	128 MB DDR	256 MB GDDR3	512 MB GDDR3
Speicherinterface	256-Bit	256-Bit	256-Bit
Speicherbandbreite	21,9 GB/s	37,8 GB/s	51,2 GB/s
Theoret. Leistung	33 GFLOPS	66 GFLOPS	250 GFLOPS
Shader-Einheiten	12 (4D)	16 (4D)	48 (4D)
Pixel-Pipelines	12	16	24
Vertex-Pipelines	6	6	8
ROPs	12	16	16
TMUs	12	16	24
Shaderversion	PS 3.0/VS 3.0	PS 2.0b/VS 2.0	PS 3.0/VS 3.0
Genauigkeit	FP16/FP32	FP24	FP16/FP32

Typ	Radeon X1900 XTX	GeForce 8800 GTX	Radeon HD 2900 XT
Hersteller	AMD (ATI)	NVIDIA	AMD (ATI)
GPU-Kern	R580	G80	R600
Transistoren	384 Mio.	681 Mio.	720 Mio.
Fertigungsprozess	90 nm	90 nm	80 nm
Chiptakt	650 MHz	575 Mhz	740 Mhz
Speichertakt	775 MHz	900 MHz	825 Mhz
Speicherausbau	512 MB GDDR3	768 MB GDDR3	512 MB GDDR3
Speicherinterface	256-Bit	384-Bit	512-Bit
Speicherbandbreite	49,6 GB/s	84,4 GB/s	103,5 GB/s
Theoret. Leistung	374 GFLOPS	518 GFLOPS	475 GFLOPS
Shader-Einheiten	48 (4D)	128 (1D)	64 (5D)
Pixel-Pipelines	16	-	-
Vertex-Pipelines	8	-	-
ROPs	16	24	16
TMUs	16	64 (32 TAUs)	16 (32 TAUs)
Shaderversion	PS 3.0/VS 3.0	SM 4.0	SM 4.0
Genauigkeit	FP32	FP16/FP32/Int32	FP16/FP32/Int32

A.3 Shader-Modell Funktionsübersicht und -vergleich

Die folgende Tabelle zeigt die Entwicklung und den Funktionsstand der Shadereinheiten bis zum heutigen Datum.

Tab. A.2: Shader-Modell Funktionsübersicht und -vergleich
Falls zwei unterschiedliche Werte angegeben sind, ist der erste für den Vertexshader und der zweite für den Fragmentshader; Quelle: [Bly06]

Version	1.1 (2001)	2.0 (2002)	3.0 (2004)	4.0 (2006)
Anzahl Befehle	128/4+8[1]	256/32+64[1]	≥512	≥64k
Konstantenregister	≥96/8	≥256/32	≥96/224	16×4096
temporäre Register	12/2	12	32	4096
Eingaberegister	16/4+2[2]	16/8+2[2]	16/10	16/32
Renderziele	1	4	4	8
Sampler	8	16	16	16
Texturen	-/8	-/16	4/16	128
2D Texturgröße	-	-	2k×2k	8k×8k
Integeroperationen	-	-	-	√
Lade-Operation	-	-	-	√
Sample-Offset	-	-	-	√
transzendente Ops.	√/-	√	√	√
Ableitungsoperatoren	-	-	√	√
Flusskontrolle	-	statisch/-	stat,dyn	dynamisch

[1]Texture-load + arithmetische Operation
[2]Textur- + Farbregister

Literaturverzeichnis

[ABB+01] Azuma, Ronald, Yohan Baillot, Reinhold Behringer, Steven Feiner, Simon Julier und Blair MacIntyre: *Recent Advances in Augmented Reality*. In: *IEEE Computer Graphics and Applications*, Band 21, Seiten 34–47, Dezember 2001.

[Adv06] Advanced Micro Devices: *AMD "Close to Metal"Technology Unleashes the Power of Stream Computing*, November 2006. http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~114147,00.html, Abruf: 13.02.2007.

[And07] Andermahr, Wolfgang: *Archiv Grafikkarten*. computerbase.de, 2006/2007. Diverse Artikel, http://www.computerbase.de/artikel/hardware/grafikkarten/.

[ath06] aths: *Ein erster Blick auf die G80-Technologie*. 3dcenter.org, Dezember 2006. http://www.3dcenter.org/artikel/2006/12-28_a.php. Abruf: 30.01.2007.

[Azu95] Azuma, Ronald T.: *A survey of augmented reality*. In: *ACM SIGGRAPH '95 Course Notes #9 – Developing Advanced Virtual Reality Applications*, Seiten 1–38, August 1995.

[BB03] Bender, Michael und Manfred Brill: *Computergrafik*. Carl Hanser Verlag, München, August 2003.

[Ber07] Bertuch, Manfred: *Architektur aktueller Direct3D-10-Grafikchips im Detail*. c't, 2007(13):176–181, Juni 2007.

[Bir] Birchfield, Stan: *KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker*. http://www.ces.clemson.edu/~stb/klt/, Abruf: 04.02.2007.

[BL97] Beis, Jeffrey S. und David G. Lowe: *Shape indexing using approximate nearest-neighbour search in high-dimensional spaces*. In: *Conference on Computer Vision and Pattern Recognition*, Seiten 1000–1006, Puerto Rico, Juni 1997.

[Bly06] Blythe, David: *The Direct3D 10 system*. In: *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, Seiten 724–734, New York, NY, USA, 2006. ACM Press.

[BTG06] Bay, Herbert, Tinne Tuytelaars und Luc Van Gool: *SURF: Speeded Up Robust Features*. In: *Proceedings of the ninth European Conference on Computer Vision*, Mai 2006. http://www.vision.ee.ethz.ch/~surf/index.html, Abruf: 03.02.2007.

[Buc05] Buck, Ian: *Taking the Plunge into GPU Computing*. In: Pharr, Matt (Herausgeber): *GPU Gems 2*, Kapitel 32, Seiten 509–519. Addison Wesley, April 2005.

[CBdR03] Colantoni, Philippe, Nabil Boukala und Jérôme da Rugna: *Fast and Accurate Color Image Processing Using 3D Graphics Cards*. In: *8th International Fall Workshop on Vision, Modeling and Visualization (VMV 2003)*, November 2003.

[Elh05] Elhassan, Irkima: *Fast Texture Downloads and Readbacks using Pixel Buffer Objects in OpenGL*. Technical Brief, NVIDIA Corporation, August 2005.

[FB81] Fischler, Martin A. und Robert C. Bolles: *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography*. Commun. ACM, 24(6):381–395, 1981.

[FK03] Fernando, Randima und Mark J. Kilgard: *The Cg Tutorial. The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley Professional, April 2003.

[FM04] Fung, James und Steve Mann: *Computer Vision Signal Processing on Graphics Processing Units*. In: *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2004)*, Seiten V–93–V–96, Montreal, Quebec, Canada, Mai 2004.

[FM05] Fung, James und Steve Mann: *OpenVIDIA: parallel GPU computer vision*. In: *MULTIMEDIA '05: Proceedings of the 13th annual ACM international conference on Multimedia*, Seiten 849–852, New York, NY, USA, 2005. ACM Press.

[FPWW03] Fisher, Robert, Simon Perkins, Ashley Walker und Erik Wolfart: *Canny Edge Detector*, 2003. Image Processing Learning Resources. http://homepages.inf.ed.ac.uk/rbf/HIPR2/canny.htm, Abruf: 12.03.2007.

[Göd06] Göddeke, Dominik: *GPGPU–Reduction Tutorial*. Technischer Bericht, Fachbereich Mathematik, Universität Dortmund, Juni 2006. http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu.

[GPGa] GPGPU: *General-Purpose Computation on GPUs*. http://gpgpu.org.

[GPGb] GPGPU.ORG: *Wiki Glossar*. http://www.gpgpu.org/w/index.php/Glossary.

[Gra06] Graphic Remedy: *gDEBugger*, 2006. http://www.gremedy.com/, Abruf: 09.06.2007.

[GRHM05] Govindaraju, Naga K., Nikunj Raghuvanshi, Michael Henson und Dinesh Manocha: *A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors*. Technischer Bericht, University of North Carolina at Chapel Hill, 2005. GPUsort library für nichtkommerziellen Einsatz frei verfügbar: http://gamma.cs.unc.edu/GPUSORT/.

[GZ06] Gress, Alexander und Gabriel Zachmann: *GPU-ABiSort: Optimal parallel sorting on stream architectures.* In: *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS '06)*, Seite 45, April 2006.

[Hey05] Heymann, Sebastian: *Implementierung und Evaluierung von Video Feature Tracking auf moderner Grafikhardware*. Diplomarbeit, Bauhaus Universität Weimar, August 2005.

[Hor05] Horn, Daniel: *Stream Reduction Operations for GPGPU Applications*. In: Pharr, Matt (Herausgeber): *GPU Gems 2*, Kapitel 36, Seiten 573–589. Addison Wesley, April 2005.

[HS88] Harris, Chris und Mike Stephens: *A combined corner and edge detector*. In: *ALVEY Vision Conference*, Seiten 147–151, 1988.

[IEE85] IEEE Standards Committee 754: *IEEE Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Std 754–1985. http://754r.ucbtest.org/standards/754.pdf.

[KBR06] Kessenich, John, Dave Baldwin und Randi Rost: *OpenGL Shading Language v1.20*. 3Dlabs, v1.20, 8. Auflage, September 2006. http://www.opengl.org/documentation/glsl/.

[KF05] Kilgariff, Emmett und Randima Fernando: *The GeForce 6 Series GPU Architekture*. In: Pharr, Matt (Herausgeber): *GPU Gems 2*, Kapitel 30, Seiten 471–492. Addison Wesley, April 2005.

[KM06] Klein, Georg und David Murray: *Full-3D Edge Tracking with a Particle Filter*. In: *British Machine Vision Conference*, Edinburgh, September 2006.

[KW03] Krüger, Jens und Rüdiger Westermann: *Linear algebra operators for GPU implementation of numerical algorithms*. ACM Transactions on Graphics (TOG), 22(3):908–916, 2003.

[LFWK05] Li, Wei, Zhe Fan, Xiaoming Wei und Arie Kaufman: *Flow Simulation with Complex Boundaries*. In: Pharr, Matt (Herausgeber): *GPU Gems 2*, Kapitel 47, Seiten 747–764. Addison Wesley, April 2005.

[Lin96] Lindeberg, Tony: *Scale-space: A framework for handling image structures at multiple scales*. In: *CERN School of Computing*, Egmond aan Zee, Niederlande, September 1996. http://www.nada.kth.se/~tony/cern-review/cern-html/cern-html.html, Abruf: 13.04.2007.

[Low04] Lowe, David G.: *Distinctive image features from scale-invariant keypoints*. In: *International Journal of Computer Vision*, Band 60, Seiten 91–110, 2004. http://www.cs.ubc.ca/~lowe/keypoints/, Abruf: 12.02.2007.

[Möl05] Möller, Bernhard: *Vorlesung zur Grafikprogrammierung*, 2005. Universität Augsburg.

[MTUK94] Milgram, Paul, Haruo Takemura, Akira Utsumi und Fumio Kishino: *Augmented Reality: A class of displays on the reality-virtuality continuum*. In: *Proceedings of Telemanipulator and Telepresence Technologies*, 1994. SPIE Vol. 2351-34.

[NH05] Neoh, Hong Shan und Asher Hazanchuk: *Adaptive Edge Detection for Real-Time Video Processing using FPGAs*. Altera Corporation, 2005.

[NVI05a] NVIDIA Corporation: *Cg Toolkit User's Manual. A Developers Guide to Programmable Graphics*, September 2005. v1.40, http://developer.nvidia.com/Cg.

[NVI05b] NVIDIA Corporation: *NVIDIA GPU Programming Guide*, Juli 2005. Version 2.4.0.

[NVI07a] NVIDIA Corporation: *CUDA Programming Guide Version 0.8.2*, April 2007. http://developer.nvidia.com/object/cuda.html.

[NVI07b] NVIDIA Corporation: *NVPerfKit*, 2007. http://developer.nvidia.com/object/nvperfkit_home.html, Abruf: 09.06.2007.

[OLG+07] Owens, John D., David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn und Timothy J. Purcell: *A Survey of General-Purpose Computation on Graphics Hardware*. Computer Graphics Forum, 26(1):80–113, 2007. http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x. Noch nicht erschienen.

[Ope] OpenVIDIA Project: *Projektseite im Internet*. http://openvidia.sourceforge.net, Abruf: 17.12.2006.

[Ope04] OpenGL ARB: *ARB_half_float_pixel specification*. OpenGL Architecture Review Board Working Group, Oktober 2004. Datum der letzten Änderung: October 1, 2004, Revision: 6, http://www.opengl.org/registry/specs/EXT/ARB/half_float_pixel.txt.

[Ope05] OpenGL ARB: *EXT_framebuffer_object specification*. OpenGL Architecture Review Board Working Group, Januar 2005. Datum der letzten Änderung: April 5, 2006, Revision: 118, http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt.

[OSW+05] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider und Tom Davis: *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley Professional, 5. Auflage, August 2005.

[Owe05] Owens, John: *Streaming Architectures and Technology Trends*. In: Pharr, Matt (Herausgeber): *GPU Gems 2*, Kapitel 29, Seiten 457–470. Addison Wesley, April 2005.

[Pod99] Podschun, Trutz Eyke: *Das Assembler-Buch – Grundlagen und Hochsprachenoptimierung*. Addison-Wesley-Longman, Bonn, 4. Auflage, 1999.

[RD05] Rosten, Edward und Tom Drummond: *Fusing points and lines for high performance tracking*. In: *IEEE International Conference on Computer Vision*, Band 2, Seiten 1508–1511, Oktober 2005.

[RD06] Rosten, Edward und Tom Drummond: *Machine learning for high-speed corner detection*. In: *European Conference on Computer Vision*, Band 1, Seiten 430–443, Mai 2006. http://mi.eng.cam.ac.uk/~er258/work/fast.html, Abruf: 11.06.2007.

[RH05] Rao, Srinivasa G. und Larry F. Hodges: *Interactive marker-less tracking of human limbs*, 2005.

[Ros06] Rost, Randi J.: *OpenGL Shading Language*. Addison-Wesley Professional, 2. Auflage, Januar 2006.

[RT07] Ready, Jason M. und Clark N. Taylor: *GPU Acceleration of Real-time Feature Based Algorithms*. wmvc, 0:8, 2007.

[SB95] Smith, S. M. und J. M. Brady: *SUSAN – A new approach to low level image processing*. Technischer Bericht TR95SMS1c, Oxford University, Chertsey, Surrey, UK, 1995.

[SDR03] Strzodka, Robert, Marc Droske und Martin Rumpf: *Fast image registration in DX9 graphics hardware*. Medical Informatics and Technologies, 6:43–49, November 2003.

[SFPG06] Sinha, Sudipta N., Jan-Michael Frahm, Marc Pollefeys und Yakup Gen: *GPU-based Video Feature Tracking And Matching*. Technischer Bericht, Department of Computer Science, UNC Chapel Hill, Department of Computer Science, CB 3175 Sitterson Hall, University of North Carolina at Chapel Hill, NC 27599, Mai 2006. http://www.cs.unc.edu/~ssinha/Research/GPU_KLT/.

[SM] Sinha, Sudipta N. und Florian E. Mücke: *GPU-SIFT*. E-Mail Korrespondenz vom 15. April 2007.

[ST94] Shi, Jianbo und Carlo Tomasi: *Good Features to Track*. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR94)*, Seattle, Juni 1994.

[Sta] Stanford University Graphics Lab: *GPUBench*. http://graphics.stanford.edu/projects/gpubench/, Abruf: 09.06.2007.

[Sta06] Stanford University Graphics Lab: *BrookGPU*, 2006. http://graphics.stanford.edu/projects/brookgpu/, Abruf: 18.06.2007.

[Ste06] Steudten, Thomas: *Alleskönner oder Hype – der Cell-BE-Prozessor*. tecchannel.de, Juli 2006. http://www.tecchannel.de/technologie/prozessoren/444414/. Abruf: 30.01.2007.

[Stu06] Sturm, Leander: *NVIDIA G80 – GeForce 8800 im Test*. hardtecs4u.com, Seiten 1–4, November 2006. http://www.hardtecs4u.com/reviews/2006/nvidia_g80/. Abruf: 30.01.2007.

[Ved] Vedaldi, Andrea: *SIFT++: A lightweight C++ implementation of SIFT*. http://vision.ucla.edu/~vedaldi/code/siftpp/siftpp.html, Abruf: 30.01.2007.

[War05] Warden, Pete: *GPU Image Processing in Apple's Motion*. In: Pharr, Matt (Herausgeber): *GPU Gems 2*, Kapitel 25, Seiten 393–408. Addison Wesley, April 2005.

[Wei91] Weiser, Mark: *The Computer for the 21st Century*. Scientific American, September 1991. http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html, Abruf: 30.04.2007.

[ZTTS06] Ziegler, Gernot, Art Tevs, Christian Theobalt und Hans-Peter Seidel: *On-the-fly Point Clouds through Histogram Pyramids*. In: Kobbelt, Leif, Torsten Kuhlen, Til Aach und Rüdiger Westermann (Herausgeber): *11th International Fall Workshop on Vision, Modeling and Visualization 2006 (VMV2006)*, Seiten 137–144, Aachen, Germany, 2006. European Association for Computer Graphics (Eurographics), Aka.