UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

AGAS
Arbeitsgruppe Aktives Sehen

Implementierung und Evaluierung von SIFT auf der GPU

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von

Sven-René von der Heidt

Betreuer:	Dipl.-Inf. Tobias Feldmann, Institut für Computervisualistik, Fachbereich Informatik
Erstgutachter:	Prof. Dr.-Ing. D. Paulus, Institut für Computervisualistik, Fachbereich Informatik
Zweitgutachter:	Dipl.-Inf. Tobias Feldmann, Institut für Computervisualistik, Fachbereich Informatik

Koblenz, im Juli 2007

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien der Arbeitsgruppe für Studien- und Diplomarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. ja ☐ nein ☐

Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. ja ☐ nein ☐

Koblenz, den ..... Unterschrift

Inhaltsverzeichnis

Kapitel 1

Einleitung

In der Bildverarbeitung werden zunehmend Algorithmen unter Verwendung von prägnanten Merkmalen implementiert. Merkmale müssen in Bildern lokalisiert und in weiteren Bildern bzw. Bildfolgen wiedergefunden werden. Ein Beispiel für eine solche Anwendung ist die Zusammenführung von mehreren Bildern zu sogenannten Panoramabildern. Prägnante Merkmale können sowohl für die optische Kameraposebestimmung als auch für die Kalibrierung von Stereokamerasystemen verwendet werden. Für solche Algorithmen ist die Qualität von Merkmalen in Bildern ein entscheidender Faktor. Gemessen wird die Qualität an der Invarianz bezüglich Translationen, Rotationen, Skalierungen und Beleuchtungsunterschieden.
In den letzten Jahren hat sich an dieser Stelle ein Verfahren hervorgetan, das von D. Lowe Anfang 2004 vorgestellt wurde [Low04]. Dieses Verfahren wird als *SIFT*[1] bezeichnet. Problematisch bei der Anwendung des Verfahrens ist seine hohe Komplexität und der daraus resultierende hohe Rechenaufwand. Durch die ständig steigende Rechenleistung der *GPU*[2] und die Möglichkeit aktiv mittels Vertex- und Fragment-Shader auf diese Rechenleistung zugreifen zu können hat sich in der Bildverarbeitung die sogenannte *GPGPU*[3] etabliert. Bei dieser Technik wird in der Bildverarbeitung der Grafikprozessor für Aufgaben verwendet, wofür dieser ursprünglich nicht vorgesehen war. Die enorme Schnelligkeit heuti-

[1] SIFT ist die Abkürzung für (engl.) *scale-invariant feature transform*.
[2] GPU steht für (engl.) *graphic processing unit*.
[3] GPGPU ist die Abkürzung für (engl.) *general-purpose computation on graphics processing unit*.

ger Grafikprozessoren ermöglicht eine Beschleunigung gängiger Algorithmen bei Einsatz dieser Technik gegenüber der ausschließlichen Nutzung der CPU.
Das SIFT-Verfahren ist sehr rechenintensiv und benötigt für die Detektierung prägnanter Merkmale unter bestimmten Voraussetzungen[4] bei Bildern der Größe 320 x 240 Pixel eine Rechenzeit von ungefähr 300 Millisekunden. Einige Verfahren aus der Bildverarbeitung gewinnen an Bedeutung, sobald sie echtzeitfähig sind und entsprechend eingesetzt werden können. Diese Echtzeit[5] ist bei der Implementierung nach D. Lowe nicht gegeben. Um das Verfahren zu beschleunigen, wurden bereits mehrere Implementationen veröffentlicht, die auf der einen Seite weiterhin aussschließlich die CPU nutzen, aber auch Verfahren die neben der CPU auch die GPU zur Berechnung bestimmter Teilbereiche des SIFT verwenden [Hey05], [SFPG06]. Unter Ausnutzung der GPU wurden Steigerungen der Geschwindigkeit um bis zum Faktor 10 erreicht.
Diese Implementationen gilt es zu hinterfragen. Ebenso ist die Qualität der Merkmale zu untersuchen, um die Verwendbarkeit von SIFT-Merkmalen für andere Bereiche der Bildverarbeitung gewährleisten zu können. Zur Visualisierung der Ergebnisse wurde eigens eine *GUI*[6] erstellt. Im Rahmen dieser Arbeit wurde die originale Implementation des SIFT-Verfahrens von D. Lowe verwendet, um Ergebnisse aus dieser Implementation zu betrachten und mit dem im Rahmen dieser Arbeit implementierten Verfahren vergleichen zu können.
In Kapitel 2 wird auf die Grundlagen des verwendeten Verfahrens, Bibliotheken und relevante Themenbereiche eingegangen, bevor Erklärungen und Erläuterungen zum implementierten Ansatz in Kapitel 3 beschrieben werden. Ergebnisse, Vergleiche und Aussichten zur Umsetzung der Aufgabe, das SIFT-Verfahren zu beschleunigen, sind in Kapitel 4 zusammengefasst.

[4] Pozessor: AMD Athlon64 X2 3800 Windsor, Grafikkarte: Nvidia 7600 GT PCX retail.
[5] Ab einer Bildrate von 20 Bildern pro Sekunde wird von Echtzeit gesprochen.
[6] GUI steht für (engl.) *graphic users interface*.

Kapitel 2

Stand der Wissenschaft

2.1 Verfahren zur Merkmalsfindung

In der Bildverarbeitung werden hohe Ansprüche an zu bestimmende und verfolgbare Merkmale in Bildern gestellt. So müssen prägnante Merkmale möglichst invariant gegenüber Translationen, Rotationen, Skalierungen und wechselnden Beleuchtungsverhältnissen sein. Hinzu kommen aber auch Probleme beim Tracking vorhandener Merkmale in Bildfolgen wie z. B. die kurzzeitige Verdeckung zu trackender Schlüsselpunkte. Aufgrund dieser Anforderungen an die Verfahren zur Extraktion robuster Merkmale haben sich zwei Verfahren durchgesetzt. Auf der einen Seite steht der Ansatz des *Kanade-Lucas-Tomasi feature trackers* (KLT), beschrieben in Kapitel 2.1.1, und auf der anderen Seite das Verfahren zur Merkmalsextraktion durch *scale-invariant feature transform* (SIFT), erläutert in Kapitel 2.1.2. Während der KLT-Algorithmus in der Bildverarbeitung in den Bereichen zur Messung des optischen Flusses oder im Anwendungsfeld der Gesichtserkennung weit verbreitet ist, hat dieses Verfahren seine Schwächen eindeutige Merkmale zu detektieren. Diese Schwäche kompensiert das SIFT-Verfahren durch einen anderen Ansatz. Hier steht nicht die Methode zur Auffindung von Punktkorrespondenzen in Bildfolgen im Vordergrund, sondern die Eindeutigkeit eines Schlüsselpunktes in seiner Umgebung, der extrahiert werden.

2.1.1 Kanade-Lucas-Tomasi feature tracker

Im Jahre 1991 entwickelten Tomasi und Kanade aus einer Methode zur Detektion von Punktkorrespondenzen einen Algorithmus, um Schlüsselpunkte in Bildfolgen robust tracken zu können [TK91]. Die Methode, auf der der entwickelte Algorithmus beruht, wurde bereits 1981 von Lucas und Kanade veröffentlicht, womit sich auch der Name *Kanade-Lucas-Tomasi feature tracker* begründen lässt. Der KLT-Algorithmus läßt sich in drei Teilbereiche gliedern:

1. **Extraktion guter Merkmalspunkte**
   Der Algorithmus wird mit dem ersten Bild initialisiert und die erste Auswahl möglicher Schlüsselpunkte wird anhand der Berechnung der Gradienten in x- und y-Richtung durchgeführt. Bei allen möglichen Schlüsselpunkten werden die kleinsten Eigenwerte der 2×2 Gradientenmatrix betrachtet und auf die Qualität des Schlüsselpunktes in Form der Trackbarkeit geschlossen. Zusätzlich müssen alle Merkmale einen Mindestabstand zueinander aufweisen, damit eine gute Abdeckung der Feature Punkte über das komplette Bild gegeben ist. Dieser Vorgang wird für jedes Bild der Bildfolge durchgeführt.

2. **Findung von Punktkorrespondenzen**
   Der Tracker des KLT-Algorithmus wird mit zwei aufeinanderfolgenden Bildern und den Koordinaten der Merkmale aus dem ersten Bild initialisiert. Für die Bewegungsschätzung wird das translatorisches Bewegungsmodell[1] und die Newton-Raphson-Methode verwendet. Dabei wird der Bildausschnitt um den Schlüsselpunkt des ersten Bildes mit dem Bildausschnitt des zweiten Bildes verglichen und mit jedem weiteren Durchlauf verkleinert. Dieses iterative Verfahren wurde für den KLT-Algorithmus um eine hierarchische Bewegungsschätzung erweitert.

3. **Verwerfung und Neufindung von Merkmalspunkten**
   Merkmale können aus dem zu trackenden Bereich herauslaufen oder einfach durch

[1] Das translatorische Bewegungsmodell geht davon aus, dass sich die Kamera nicht bewegt und dass sich ein Bild in zwei Teile gliedert, in den Vordergrund und Hintergrund. Auf dem festen Hintergrund bewegen sich Objekte die vom Hintergrund und von weiteren Objekten verdeckt werden können.

andere Objekte im Bild verdeckt werden. In einem solchen Fall werden beim KLT-Verfahren die Schlüsselpunkte verworfen. Wenn in der Folge eine Mindestanzahl vorhandener Schüsselpunkte unterschritten wird, werden weitere Merkmale extrahiert und die Liste der zu verwendeten Punkte damit aufgefüllt.

Dieser Algorithmus wurde im Laufe der Zeit mehrfach modifiziert und auch erweitert. Ein Beispiel für eine Abwandlung des ursprünglichen Trackers stellt die Erweiterung von Tommasini und Fusiello dar [TRFT98]. In dieser Variante werden schlechte Merkmale direkt verworfen, ohne sie zu tracken, bevor sie verworfen werden müssen. Dadurch wird das Verfahren wesentlich robuster, da die Fehlerrate der getrackten Punkte deutlich reduziert wird. Während der ursprüngliche Algorithmus ein reines translatorisches Bewegungsmodell dem Matching zu Grunde legt, verwendet die Erweiterung ein affines Bewegungsmodell, wodurch ein besseres Ergebnis bezüglich Rotationen und Skalierungen erzielt wird.

2.1.2 Scale-invariant feature transform

In [Low04] wird das SIFT-Verfahren vorgestellt, mit dem aus Bildern Merkmale extrahiert werden, die invariant gegen Skalierungen, Translationen und Rotationen sind. Ebenso haben Beleuchtungsunterschiede und unterschiedliche Blickwinkel auf solche Merkmalen einen sehr geringen Einfluss. In dieser Veröffentlichung werden die SIFT-Merkmale im Zusammenhang mit bildorientierter Objekterkennung[2] erläutert und deren Vorteile gegenüber anderen Verfahren verdeutlicht. Dieses Verfahren ist in der Bildverarbeitung zum heutigen Zeitpunkt Standard der Technik, wenn es um Extraktion von prägnanten Merkmalen und Objekterkennung geht. Das SIFT-Verfahren lässt sich wie folgt gliedern:

1. **Aufbau einer Skalenraumpyramide**
   Als erstes wird im SIFT-Verfahren ein Skalenraum unter Verwendung einer Gaußfaltung aufgebaut. Dazu wird das Eingangsbild mit einer Gauß's'schen Unschärfe versehen und dient für das nächste Bild, das ebenfalls mit einem solchen Filter bearbeitet wird, als Ausgangsbild für die weitere Bearbeitung. Aus diesen Bildern wer-

[2] In der Bildverarbeitung unterscheidet man zwischen objektorientierter (zugrunde liegendes 3D-Modell) und bildorientierter (einfache bildliche Darstellung) Objekterkennung.

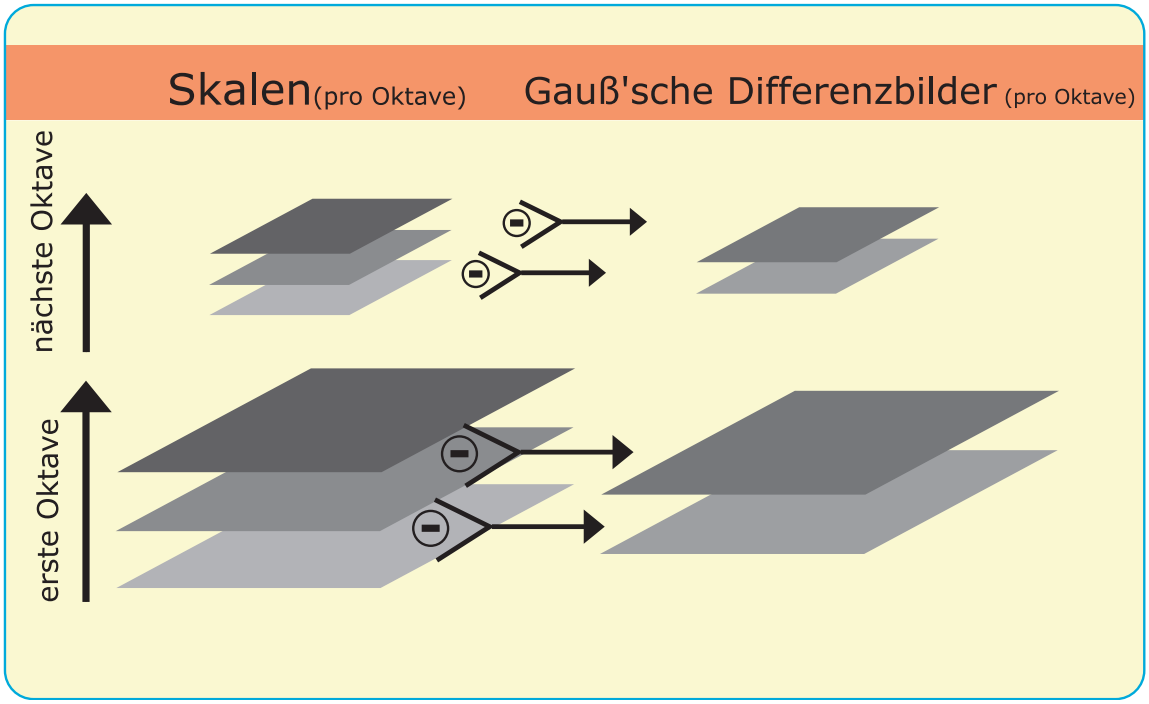


Bild 2.1: Aufbau einer Skalenraumpyramide und Erzeugung der Differenzbilder.

den die *DoGs*[3] berechnet und dienen als Grundlage für die Berechnung der Schlüsselpunkte. Nachdem die gewünschte Anzahl der DoGs[4] erzeugt wurde, wird die Bildgröße halbiert und die Bearbeitung der Bilder wiederholt. Dies passiert, bis ein Schwellwert der Bildgröße erreicht wird. Dieser Schwellwert wird in der Literatur [Low04] mit 12 Pixeln beziffert. Abbildung 2.1 veranschaulicht den Aufbau einer Skalenraumpyramide, die Erzeugung der *DoGs* und Bildung einzelner Oktaven.

2. **Detektierung von Schlüsselpunkten anhand Extremwerte im Skalenraum**
   Nach Aufbau der Skalenraumpyramide wird jeder Pixel daraufhin geprüft, ob dieser in seiner Umgebung ein Extremwert darstellt. Die Abbildung 2.2 zeigt welche Nachbarschaft zur Prüfung auf einen Extremwert jedes Pixels herangezogen wird. Stellt

[3] *DoG* steht für *difference of gaussian* und bezeichnet ein Differenzbild aus zwei Bildern, die mit einem Gaußfilter vorverarbeitet wurden.

[4] Die Anzahl der DoGs ist abhängig von der Zahl der Skalen. Aus n Skalen werden n-1 Differenzbilder erzeugt.

ein Pixel ein Extremwert dar, wird dieser als möglicher Schlüsselpunkt vorgemerkt.

3. **Entfernung instabiler Merkmale**
   Nun werden die vorgemerkten Mermale nochmals überprüft, indem Schlüsselpunkte die auf Kanten liegen als instabil bezeichnet und aus der Liste möglicher Merkmale entfernt werden.

4. **Berechnung der Orientierung**
   Für die nun ausgewählten Schlüsselpunkte werden die Richtungen berechnet, um die Rotationsinvarianz der Merkmale zu erhalten.

5. **Aufbau der Deskriptoren aller Schlüsselpunkte**
   Der Deskriptor eines Merkmals hat eine Länge von 128 Elementen. Dieser 128-elementige Deskriptor setzt sich aus einem Gradientenfenster der Größe 4x4 mit je einem Gradientenhistogramm und dessen 8 Richtungen zusammen. Die Normalisierung dieses Vektors erzeugt die Helligkeitsinvarianz, die dieses Verfahren, ebenso wie die Skalierungs- und Transformationsinvarianz, auszeichnet.

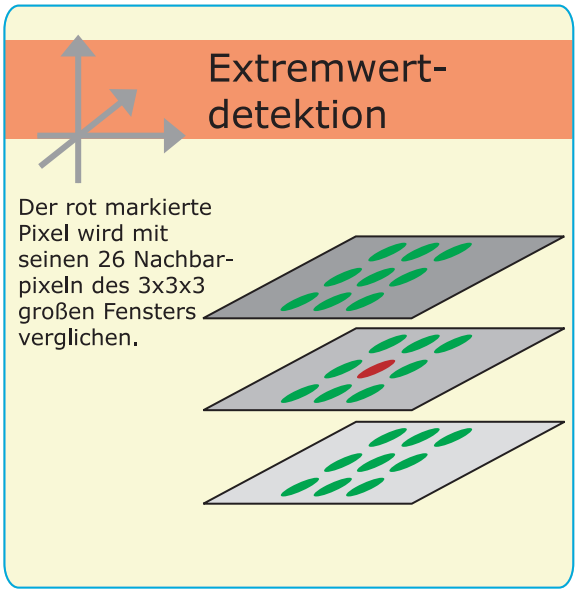


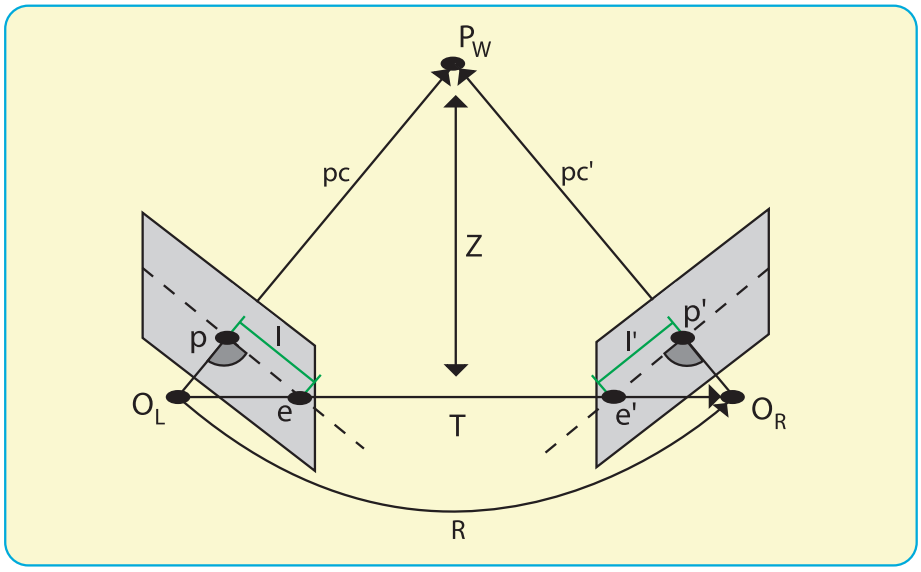
Bild 2.2: Findung der Extrema in seiner Umgebung.

2.2 Verwendung prägnanter Merkmale

2.2.1 Erstellung von Panoramabildern

Um großen ausgedehnten Motiven in ihrer Wirkung auf Bildern gerecht zu werden, hat sich das Panoramabild etabliert. In einem solchen Format können weitläufige Landschaften oder große Bauwerke in einer hohen Auflösung als ganzes Bild oder sogar als 360° Ansicht dargestellt werden. Diese Art von Foto hinterlässt oft einen weitaus imposanteren Eindruck von Landschaften, Plätzen und Bauwerken, als es herkömmliche Bildformate vermitteln können.
Um solche Panoramabilder zu erstellen, benötigt man eine große Anzahl von Bildern einer kompletten Bildserie, die bestimmten Ansprüchen genügen muss. Diese Bildserien sollten mit gleicher Belichtungszeit, möglichst zeitgleich, vom gleichen Standpunkt aus und unter Verwendung eines Stativs erstellt werden, um den Rechenaufwand zu minimieren und gleichzeitig das Ergebnis eines Panoramabildes zu verbessern. Neben den Korrekturen unterschiedlicher Helligkeiten und möglichen Anpassungen in der Positionierung, liegt auch hier das Hauptaugenmerk auf prägnanten Merkmalen, damit die Bilder an den richtigen Stellen zusammengefügen werden können. Dabei werden Bilder an Stellen eindeutig zugeordneter Merkmale zusammengefügt, um den Eindruck eines Panoramas zu erhalten. Sind die eindeutig zugeordneten Merkmale nicht identisch miteinander, werden Bilder an falschen Stellen zusammengeführt und der Übergang von Bild zu Bild ist ersichtlich. Für die Erstellung von Panoramabildern wird das SIFT-Verfahren eingesetzt, da solche Schlüsselpunkte aufgrund ihres Deskriptors eindeutig beschrieben werden können. Autopano-SIFT [Now04] von S. Nowozin ist eine Software zur Erstellung von Panoramabildern die sich dem SIFT-Verfahren bedient und mittels deren Merkmale die Zusammenführung der einzelnen Bilder berechnet. Autopano-SIFT ist eine freie Software die unter der *GNU GPL*[5] steht.

[5]GNU GPL steht für engl. *GNU General Public License*.



O_L Optisches Zentrum der Kamera L im Weltkoordinatensystem
O_R Optisches Zentrum der Kamera R im Weltkoordinatensystem
T Basisvektor (auch Translationsvektor) um O_L nach O_R zu überführen
P_W Punkt p in Weltkoordinaten
p Punkt p in Bildkoordinaten der Kamera L
p' Punkt p' in Bildkoordinaten der Kamera R
e Epipol in der Bildebene von Kamera L
e' Epipol in der Bildebene von Kamera R
l Epipolarlinie in der Bildebene von Kamera L
l' Epipolarlinie in der Bildebene von Kamera R
Z Tiefe des Punktes p
pc Punkt p im Koordinatensystem der Kamera L
pc' Punkt p im Koordinatensystem der Kamera R
R Rotationsmatrix um den Vektor pc auf den Vektor pc' zu drehen

Bild 2.3: Bildliche Darstellung geometrischer Zusammenhänge der Epipolargeometrie.

2.2.2 Kalibrierung von Stereokamerasystemen

Um eine Abbildung von 3D-Weltkoordinaten in 2D-Bildkoordinaten beschreiben zu können, benötigt man bestimmte Parameter. Die Bestimmung dieser Parameter bezeichnet man als Kamerakalibrierung. Diese Kalibrierung bezieht sich auf intrinsische und extrinsische Parameter. Intrinsische Parameter sind durch die Kamera selber gegebene Faktoren, die das entstehende Bild beeinflussen. Extrinsische Parameter sind abhängig von der Po-

sitionierung und der Blickrichtung der jeweiligen Kamera. In einem Stereokameramodell sind die intrinsischen Parameter bekannt, extrinsische Parameter müssen berechnet werden. Es gibt zwei verschiedene Stereokameramodelle. Dabei unterscheiden sie sich in der Blickrichtung beider Kameras. Im ersten Modell sind beide Kameras so angeordnet, dass sich ihre optischen Achsen im Fixationspunkt scheiden. Diese Stereokameramodell entspricht dem Aufbau des menschlichen Sehens; die Sehstrahlen beider Augen schneiden sich im sogenannten Punkt des scharfen Sehens. Das zweite Stereokameramodell zeichnet sich durch die parallel zueinander laufenden optischen Achsen beider Kameras aus. Durch diese Anordnung der Kameras wird die Geometrie vereinfacht und die Rekonstruktion von Tiefeninformationen beschleunigt. Der Nachteil des zweiten Stereokameramodell liegt in seiner fast unmöglichen Umsetzung. Dies bedeutet, dass es kaum realisierbar ist, zwei Kameras und deren optischen Achsen exakt parallel zueinander anzuordnen. Zusätzlich sind zwei Kameras in ihren Eigenschaften nie identisch, da produktionsbedingte Ungenauigkeiten zu Verschiebungen und Verzerrungen führen.
Da das zweite Stereokameramodell die Berechnungen vereinfacht und beschleunigt, wird von diesem Kameramodell bei der Berechnung Tiefeninformationen ausgegangen, obwohl Bilder des ersten Modells vorliegen. Folglich müssen Bilder erst vorverarbeitet werden, damit die Bilder auf ein Stereokameramodell ohne Fixationspunkt anwendbar sind.
Welche Geometrie zugrunde liegt, wenn eine Szene von zwei Kameras aufgenommen wurde, wird in Abildung 2.3 gezeigt. Bezeichnet wird diese Geometrie als Epipolargeometrie, deren Zusammenhänge die Grundlagen für die 3D-Rekonstruktion aufgenommenen Szenen darstellt. Abbildung 2.3 veranschaulicht die geometrischen Zusammenhänge der Epipolargeometrie. Aus korrespondierenden Punktepaaren kann mit Hilfe der Trigonometrie deren Tiefe zur Basislinie bestimmt werden. Die Verwendung von Punkten aus dem Bildkoordinatensystem für die Tiefenbestimmung lässt auf eine noch genauere Berechnung von Tiefeninformationen schließen, wird neben den Bildkoordinaten zusätzlich der Subpixelbereich von Merkmalen in die Berechnung mit einbezogen. Das im Rahmen dieser Arbeit modifizierte SIFT-Verfahren berechnet seine Merkmale mit dieser Subpixelgenauigkeit. Jedoch gilt es zu untersuchen, wie konstant dieser Bereich gegenüber möglichen Störungen im Bild ist und ob sich die Verwendung dieses Bereichs negativ auf die Genauigkeit der Berechnungen von Tiefeninformationen auswirkt.
Eine andere Anwendung für den Subpixelbereich von Merkmalen ist die Schätzung der

Kameraposition. Dabei wird anhand natürlicher Merkmale in Bildern die Schätzung der Kameraposition vorgenommen. Auch hier verspricht die Hinzunahme des Subpixelbereichs von Koordinaten eine verbesserte Genauigkeit.

2.3 Programmierung der GPU

2.3.1 Entwicklung der Grafikhardware

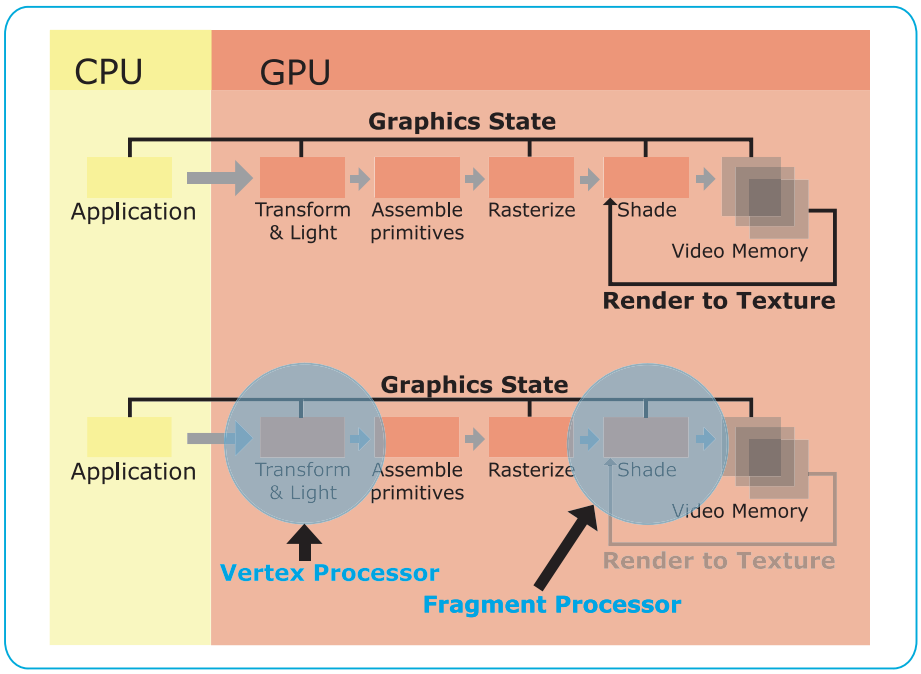


Bild 2.4: Einordnung des Vertex- und des Fragment-Prozessors in der Grafikpipeline.

Mit Steigerung der Anforderungen an die Hardware von Computern aufgrund ihrer wachsenden Bedeutung in der Wirtschaft und im häuslichen Gebrauch entwickelte sich in den 90er Jahren eine Trennung zwischen der CPU und der reinen grafischen Darstellung. 1987 führte IBM den sogenannten Standard für die Computergrafik *Video Graphics Array* (VGA) ein. Bezeichnet wurde die Hardware als *VGA Controller* und ist in der heutigen eingesetzten Hardware die Einheit, die als *Framebuffer* betitelt wird. Ende der 90er Jahre

wurde der Begriff *VGA Controller* von der Bezeichnung *GPU* abgelöst und bezeichnet einen Teil der Hardware in einem Computer, der in den letzten acht Jahren eine enorme Entwicklung bezüglich Geschwindigkeit und Programmierbarkeit gemacht hat. Heute spricht man noch von vier Generationen der Grafikhardware, von denen jede Generation eine Weiterentwicklung der möglichen Programmierung und Schnelligkeit darstellt. Wenn man von der Entwicklung der GPU spricht, muss man auch die Schnittstellen zur Programmierung der Grafikhardware erwähnen. Die beiden Hauptakteure der 3D-Programmierung sind *OpenGL*[6] und *Direct3D*. Während OpenGL ein offener Standard für Windows, Linux, Unix und Macintosh darstellt, ist Direct3D die von Microsoft entwickelte Schnittstelle für die 3D-Programmierung, die ein Teilbereich der Multimedia-Programmierung darstellt. Der Themenkomplex Multimedia Programmierung wird von Microsoft als *DirectX* bezeichnet. Die führenden Marken, die den Markt der Grafikhardware beherrschen, sind nVIDIA [7] und ATI [8].
Abbildung 2.4 zeigt, an welchen Stellen der Grafikpipeline die Programmierung der GPU eingreift und wie sich das Thema Shader in den Prozess der bildlichen Darstellung eingliedert. In dieser Abbildung wird der Unterschied zwischen Vertex- und Fragment-Shader ersichtlich. Vertex-Shader manipulieren die Geometrie einer Szene, indem die Vertices der Oberfläche verschoben werden können. Der Vertex-Shader ersetzt hinsichtlich seines höheren Funktionsumfangs den Bereich der Grafikpipeline *transform and light*.
Der Fragment-Shader ist nach der Rasterisation angeordnet. In dem Teilbereich der Rasterisation werden die Primitive auf die Projektionsebene projiziert und die Positionen der Pixel für das entstehende Bild berechnet. Nach diesem Bereich der Grafikpipeline kann die Geometrie der Oberfläche nicht mehr verändert werden. Die einzige Komponente, die durch dem Fragment-Shader manipulierbar bleibt, ist die Farbe der Fragmente[9].

[6] OpenGL steht für Open Graphics Library

[7] nVIDIA Cooperation ist einer größten Entwicklerfirmen von Grafikprozessoren und Chipsätzen.

[8] ATI ist eine Marke der Firma AMD und wird u.a. bei Grafikprozessoren verwendet. ATI ist ein Markenname, der auf die von AMD im Jahr 2006 übernommene Firma *ATI Technologies Ltd* zurückzuführen ist.

[9] Mehr zur Entwicklung der Grafikhardware, deren Schnittstellen zur 3D-Programmierung und eine Übersicht der verschiedenen Grafikprozessorgenerationen findet man in [FK03], auf der Internetpräsenz von Microsoft zum Thema DirectX oder auch auf den Internetseiten von nVIDIA und ATI.

2.3.2 Hochsprachen zur Programmierung der Grafikhardware

Um die Grafikhardware programmieren zu können, müssen die Shadereinheiten der GPU mittels *Vertex-* und *Fragement-Shader* genutzt werden. Diese Shader-Programme werden in speziell dafür vorgesehenen Hochsprachen erstellt. Zu diesen Hochsprachen gehören *C for Graphics* (Cg) von nVIDIA, *OpenGL Shading Language* (GLSL) und die *High Level Shading Language* (HLSL). HLSL ist ein Teil von DirectX und ermöglicht die Programmierung von Shadern für der Schnittstelle *Direct3D*. GLSL ist wie Cg stark an die Syntax der Programmiersprache C angelehnt. GLSL ist entgegen Cg Teil der OpenGL-Spezifikation und wurde mit der Version OpenGL 2.0 im Jahr 2004 eingeführt. Während bei der Verwendung der Hochsprachen HLSL oder GLSL für die jeweilig andere Schnittstelle der komplette Programmcode der Shader konvertiert werden muss, ist Cg so konzipiert, dass erst der Cg-Compiler den Programmcode in Abhängigkeit der gewählten Schnittstelle (OpenGL oder Direct3D) übersetzt.

Kapitel 3

Eigener Ansatz

Ziel dieser Arbeit ist das SIFT-Verfahren unter Ausnutzung der GPU zu beschleunigen, zu untersuchen, welche Auslagerungen der einzelnen Teilbereiche dieser Merkmalsfindung sinnvoll sind und ob der Subpixelbereich der SIFT-Features zur Tiefenberechnung geeignet ist. Aus diesem Grund liegt das Hauptaugenmerk dieser Arbeit nicht nur auf der Programmierung von Fragment-Shadern, sondern auch in der Entwicklung einer grafischen Oberfläche, die die Evaluation der Ergebnisse erleichtert, und schon nach direkter Anwendung der Implementationen erste Schlüsse ermöglicht.
Für die Auslagerung bestimmter Bereiche des SIFT-Verfahrens zur Berechnung auf der GPU wurde die Implementation mit ausschließlicher Nutzung der CPU erweitert und geringfügig abgeändert. Es wurden weitestgehend alle Funktionen übernommen, damit die erzielten Ergebnisse vergleichbar bleiben und zu Evaluationszwecken genutzt werden können. Die Umsetzung der Auslagerung von Teilbereichen des SIFT-Verfahrens und den benötigten Bibliotheken finden sich im Abschnitt 3.1.
Die grafische Oberfläche wurde mittels Qt in C++ erstellt. Dabei galt es Funktionen umzusetzen, die das Verhalten der SIFT-Merkmale widerspiegeln, die über Bildfolgen eindeutig dem zugehörigem Schlüsselpunkt zugewiesen werden. Gleichzeitig werden Zeitmessungen, die Anzahl gefundener Merkmale und entsprechende Vergleiche beider Implementationen dargestellt, damit erzielte Ergebnisse leicht interpretierbar sind. Eine kurze Beschreibung der Oberfläche folgt im Abschnitt 3.2.

3.1 Auslagerung des Verfahrens auf die GPU

3.1.1 Aufbau und Struktur

Durch den großen Umfang der Aufgabenstellung war es wichtig, eine Datenstruktur aufzubauen, die es ermöglicht, Ergebnisse zugänglich und übersichtlich gestalten zu können. Aus diesem Grund wurde in dieser Arbeit darauf verzichtet, bereits vorhandene Datenstrukturen weiter zu verwenden. Stattdessen wurde eine eigene Datenstruktur implementiert, die die Darstellung in der grafischen Oberfläche und das Einfügen unterschiedlicher Varianten der Merkmalsberechnung erleichtert. Zwar verwendet die Implementation nach D. Lowe eine gute Struktur der Daten, jedoch wird im Rahmen dieser Arbeit eine eigene Klasse namens *result* verwendet, die alle relevanten Daten hält und die ursprüngliche Struktur erweitert. *Result* hält alle Attribute, die für die Darstellung der Ergebnisse und für die Evaluierung gebraucht werden. Diese Klasse stellt das Bindeglied zwischen den beiden implementierten SIFT-Verfahren, der grafischen Oberfläche und der Bedienung mittels Konsoleneingaben dar. Jedes Objekt der Klasse *result* kann eine Menge von Schlüsselpunkten haben, die auf der CPU oder auch mittels der GPU berechnet werden. Alle Zeitmessungen, die bei Anwendung der Implementationen gemacht werden, alle in der GUI angezeigten Bilder sowie Attribute, die zur Steuerung der grafischen Benutzeroberfläche dienen, werden in dieser Klasse gehalten. Die Klasse namens *keypoint* hält neben den Koordinaten der Merkmale Informationen über eindeutig zugeordente Schlüsselpunkte in einem darauffolgenden oder vorherigen Bild einer Bildfolge.

3.1.2 Initialisierung und Verwendung benötigter Bibliotheken

Für die Programmierung der Grafikhardware müssen bestimmte Bilbliotheken verwendet werden. Zu diesen Bibliotheken gehören das *Utility Toolkit* (GLUT), die *OpenGL Extensions* (GLEXT) und die *OpenGL Extension Wrangler Library* (GLEW). GLUT erzeugt für die Einbindung von Shadern einen OpenGL-Kontext, GLEXT stellt Funktionen für die Berechnung mit Floating-Points zur Verfügung und mittels GLEXT wird die Verwendung von Framebuffer-Objekten ermöglicht. Zusätzlich muss für die Verwendung der Hochsprache Cg die Headerdatei `cgGL.h` eingebunden werden.

Die Klasse *siftgpu* beinhaltet alle Attribute und Operationen, die für die Berechnung der Merkmale unter Mitbenutzung der GPU benötigt werden. Im Unterschied zu der Klasse *sift* wird der Startfunktion, die die Berechnung anstößt, eine Liste der zu verwendeten Bilder übergeben. Dies wurde auf diese Weise implementiert, da für die Nutzung der GPU bestimmte Voraussetzungen geschaffen werden müssen, die, einmal initialisiert, bis zur vollständigen Bearbeitung aller ausgewählten Bilder bestehen bleiben. Der Startfunktion der Klasse *sift* wird jeweils ein Bild zur Berechnung von Merkmalen übergeben.
Die Headerdatei `siftgpu.h` bindet bereits die benötigten Bibliotheken mit `GL/glut.h`, `GL/glext.h` und `Cg/cgGL.h`. In der Datei `siftgpu.cpp` müssen als erstes die Bibliotheken `glew.h` und `gl.h` im Quellcode eingebunden werden. Wichtig dabei ist die Einhaltung der hier verwendeten Reihenfolge, da es sonst beim Kompilieren des Quellcodes zu Konflikten kommt, die vom Compiler nicht aufgelöst werden können. Zu den Voraussetzungen, die im Rahmen der Verwendung der GPU geschaffen werden müssen, zählt die Erzeugung eines OpenGL-Kontextes, die Initialisierung der OpenGL-Erweiterungen sowie die Erzeugung eines Framebuffer-Objektes für das Offscreenrendering.
Der OpenGL-Kontext wird, wie im folgenden Abschnitt ersichtlich, mittels GLUT erzeugt.

```
> glutInit (&argc, argv);
> glutCreateWindow("GL context for GLEW");

> GLenum err = glewInit();

> //generate a framebuffer object
> glGenFramebuffersEXT(1, &fb);
> // bind offscreen framebuffer
> glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
> glMatrixMode(GL_PROJECTION);
...
> glViewport(0, 0, width, height);
> glGenTextures (4, tex);
```

Nach der Erzeugung des OpenGL-Kontextes wird GLEW initialisiert. Für die Verwendung der GPU wird im Rahmen dieser Arbeit ein Offscreenrendering in Verbindung mit sogenannten Framebuffer-Objekten angewendet. Für diese Art des Renderings wird die Pipeline der Grafikhardware so abgewandelt, dass in einem Speicherbereich der GPU gerendert wird, der von den Shadern direkt ausgelesen werden kann. Dadurch können Daten länger auf der GPU gehalten werden und müssen nicht dauernd zwischen den Speichern der CPU und GPU hin und her geschoben werden. Das Auslesen der Framebuffer kann einen Bottleneckeffekt darstellen, der für einen hohen Zeitverlust verantwortlich ist und den Performancegewinn bei Ausnutzung der GPU minimiert.
Am Anfang werden die Grauwerte aller Pixel des Bildes, das zur Merkmalsberechnung verwendet werden soll, ohne seine Headerinformationen, wie z. B. Dateiformat und Bildgröße, in ein Array geschrieben. Um ein 1:1-Mapping des Arrays und der 2D-Textur beizubehalten, wird im Anschluss an die Reservierung des Speicherbereichs für das Renderziel das Koordinatensystem der Renderumgebung gesetzt. Standardmäßig wird die z-Koordinate in Weltkoordinaten auf Null gesetzt. Dafür verwendet man die Größe der Textur, die jedoch nach Einlesen der Bilddaten eine andere Größe aufweisen kann als die eigentliche Bildgröße des Ausgangsbildes. Der genauere Zusammenhang dieser Thematik ist im Abschnitt 3.1.3 beschrieben.
Danach wird mit `glGenTextures(4, tex)` die Anzahl und der Name der zu verwendeten Texturen gesetzt. Eine Anmerkung an dieser Stelle soll sein, dass bei einigen Tests ein Geschwindigkeitsverlust auftrat, wenn die Deklaration der Textur vor der Initialisierung des Framebuffer-Objektes gemacht wird. Die letzte Codezeile schließt die Schaffung der geforderten Rahmenbedingung für die Verwendung der GPU ab.
Anschließend wird Cg initialisiert. Dazu wird als erstes ein Kontext für die Fragment-Shader erzeugt und das zu verwendende Profil der eingesetzten Hardware gesetzt. Danach werden die zu verwendenden Fragmentprogramme erzeugt und in den Speicher geladen. Dabei werden die Parameter, die den Shadern zugänglich gemacht werden sollen, an die entsprechenden Fragmentprogramme und Bezeichnungen der Parameter gebunden. Im folgenden Beispiel wird ein solcher Kontext erzeugt, das Profil der verwendeten Hardware auf *ARBFP1* gesetzt, ein Cg-Programm aus der Datei `gaussianNoiseX.cg` erzeugt und in den Speicher geladen. Abschließend wird der Parameter `textureParam` der Variablen `texture` aus dem eben erzeugten Cg-Programm zugewiesen.

```
> fragmentContext = cgCreateContext();
> cgGLSetOptimalOptions(CG_PROFILE_ARBFP1);
> gaussian_X = cgCreateProgramFromFile
>  (fragmentContext,CG_SOURCE, ".shader/gaussianNoiseX.cg",
>   fragmentProfile, "gaussianNoiseX", NULL);
> cgGLLoadProgram(gaussian_X);
> textureParam = cgGetNamedParameter (gaussian_X, "texture");
```

Durch diese grundlegende Initialisierung eines Shaders können nun die Bilddaten weiterverarbeitet werden. Dazu werden die Daten des Eingangsbildes in eine Textur geschrieben, die den Shadern zugänglich gemacht wird. Im nächsten Abschnitt wird gesondert auf die zu initialisierenden Texturen eingegangen.

3.1.3 Generierung der Texturen

Als erstes müssen die Parameter der Texturen, die OpenGL zur Verfügung stehen sollen, richtig gesetzt werden. Um diesen Sachverhalt verständlicher zu machen, ist der folgende Ausschnitt des Quellcodes gedacht:

```
> //activate and bind texture
> glBindTexture(GL_TEXTURE_RECTANGLE_ARB,textureID);
> //turn off filtering and wrap modes
> glTexParameteri
>  (GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
...
> glTexParameteri
>  (GL_TEXTURE_RECTANGLE_ARB, GL_TEXTURE_WRAP_T, GL_CLAMP);
> //define texture with floating point format
> glTexImage2D
>  (GL_TEXTURE_RECTANGLE_ARB, 0, GL_RGBA32F_ARB, width, height,
>   0, GL_RGBA,GL_FLOAT, 0);
```

Zu Beginn muss eine bestimmte Textur ausgewählt werden. Wie im Abschnitt 3.1.2 beschrieben, wurden dort mittels *glGenTextures (4, tex);* vier Speicherbereiche für Texturen unter dem Namen *tex* erzeugt. Als Texturformat wird die OpenGL-Erweiterung *ARB Texture Rectangle* verwendet. Dieses Format ermöglicht Bildbreiten oder -höhen, die nicht durch eine 2er-Potenz darstellbar sind. Mit `glTexImage2D()` der Initialisierung wird das interne Format der zu speichernden Werte auf Floating-Points gesetzt. Ebenso wird hier die Größe der Textur angegeben. Diese wird auf die Hälfte der Breite und der Höhe des Ausgangsbildes gesetzt, da die Bilddaten mittels einer bestimmten Sortierung in ein anderes Format mit vier Speicherbereichen pro Pixel abgelegt werden.

Abbildung 3.1 zeigt das Verhalten von unterschiedlich gepackten Texturen. Durch die

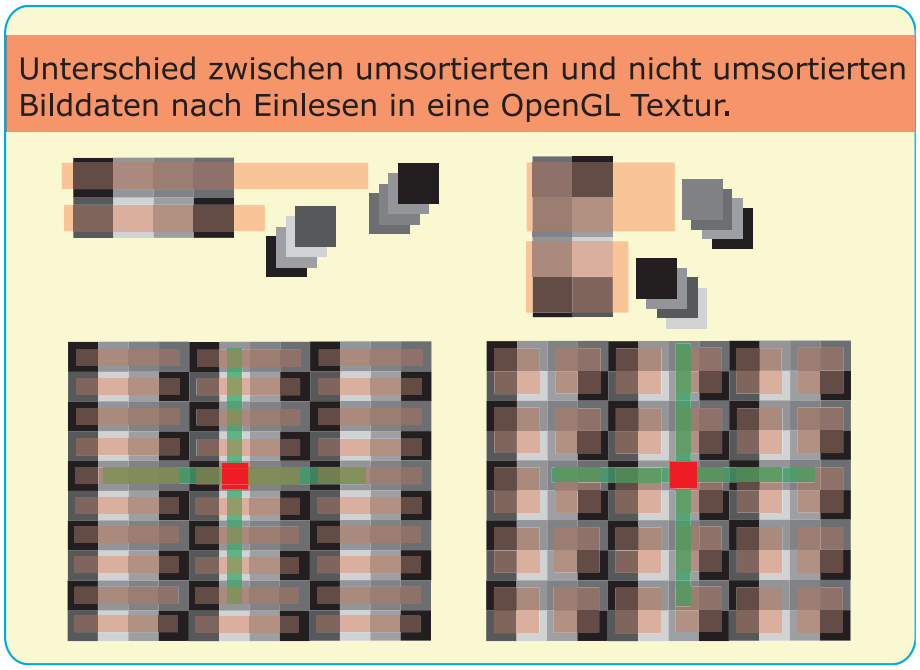


Bild 3.1: Sortierung der Bilddaten für die zu verwendenden Texturen.

Verwendung von Texturen zur Weiterverarbeitung auf der GPU mit der Formatierung der Bilddaten nach RGBA ist es möglich, die Grauwerte so umzuordnen, dass sich daraus ein Vorteil für die Gaußfilterung ergibt. Beim Schreiben des mit Grauwerten gefüllten Arrays mittels der zugehörigen OpenGL-Funktion in das Texturformat werden automatisch die Werte nacheinander in die Speicherbereiche für die RGBA-Werte überführt. Wie in Abbil-

dung 3.1 ersichtlich, wird durch diese einfache Umsortierung der Bilddaten und Speicherung der Daten in das Texturformat die Texturgröße lediglich in x-Richtung um den Faktor vier minimiert. Sortiert man die Daten für die Textur um, kann die Texturgröße gegenüber der anfänglichen Bildgröße in beide Richtungen um den Faktor zwei minimiert werden. Bei der Bildfilterung mittels eines Gaußkerns der Größe neun, ist eine Einsparung von zwei Texturzugriffen pro Fragment möglich. In Zahlen ausgedrückt bedeutet dies, dass bei einer Bildgröße von 800×600 Pixeln für die Gaußfilterung 960.000 Texturzugriffe weniger benötigt werden. Bei Zeitmessungen wurde ein Geschwindigkeitsgewinn in der Größenordnung von 10 Millisekunden im Vergleich zur Gesamtdauer verzeichnet.

```
glFramebufferTexture2DEXT
(GL_FRAMEBUFFER_EXT, AttachmentID[i-1],
 GL_TEXTURE_RECTANGLE_ARB, tex[i], 0);
```

Nachdem die Parameter der Texturen gesetzt und diese erzeugt sind, müssen die Texturen nun an das erzeugte Framebuffer-Objekt gebunden werden. Dazu werden drei Texturen als Renderziele dem Framebuffer-Objekt mittels der Bezeichnung `AttachmentID` eindeutig zugeordnet. Diese Variable hält die Adressen der Speicherbereiche auf der GPU, in die gerendert wird, und wird als *Attatchment Point* bezeichnet.
Das *Ping-Pong-Verfahren* betitelt in der GPGPU das Speichern von Daten in zwei Texturen, die abwechselnd als Renderziel und als Ausgangsdatensatz für Berechnungen fungieren. Dabei wird eine Textur dem Fragment-Shader im *read-only*-Modus zur Verfügung gestellt und eine zweite Textur im Modus *write-only* als Renderziel gebunden. Nachdem das Renderziel im Modus write-only verwendet wurde, wird dieses für die nächste Anwendung im read-only-Modus verwendet. Dadurch wechselt der Modus zwischen den einzelnen Attatchment Points.

3.1.4 Berechnungen von Teilbereichen des SIFT-Verfahrens auf der GPU

Nachdem alle geforderten Elemenete initialisiert sind, kann mit den Auslagerungen begonnen werden. Dazu wird das Eingangsbild als Grauwertbild eingelesen und schließlich

in das geforderte Texturformat geschrieben. Dadurch minimiert sich die Texturgröße in x- und y-Richtung um die Hälfte im Vergleich zur ursprünglichen Bildgröße. Diese Eingangstextur wird in den Speicher der GPU geladen und dort als Eingangsparameter behandelt. Für weitere Erklärungen im Zusammenhang mit dem eigenen Ansatz ist in der folgenden Gliederung das SIFT-Verfahren nochmals in seine Bereiche zerlegt. Die dabei fettmarkierten Punkte sind die Bereiche, deren Berechnung auf der GPU durchgeführt werden.

- **Aufbau der Skalenraumpyramide (Verwendung einer Gaußfilterung),**
- **Berechnung der Orientierung und der Gradienten,**
- *Detektierung von Schlüsselpunkten anhand von Extremwerten im Skalenraum,*
- *Entferung instabiler Merkmale (Merkmale dürfen nicht auf Kanten liegen),*
- *Aufbau der Deskriptoren für jedes Merkmal.*

Der Aufbau der Skalenraumpyramide mittels gaußgefilterten Bildern, die Erzeugung der DoGs, wie auch die Berechnungen aller Orientierungen von Punkten und deren Gradienten finden auf der GPU statt. Der gewählte Ansatz nur Teilbereiche des Verfahrens auszulagern, wird in der Programmierung der GPU deutlich. Dabei wird nicht versucht Texturen so lange wie möglich im Speicher der Grafikhardware zu halten, sondern es werden lediglich Teile des Verfahrens auf der GPU berechnet, als Textur in den Speicher der GPU geschrieben und schließlich der CPU zur weiteren Verwendung wieder zur Verfügung gestellt.
Nachdem das Eingangsbild als Textur in den Speicher der GPU geladen wurde, werden auf diese Bilddaten die Fragment-Shader für die Gaußfilterung angewendet. Dabei wird die Gaußfilterung erst in x-Richtung mit dem Kern der Größe neun angewendet. Die Ergebnisse werden danach in einen bestimmten Speicherbereich der GPU geschrieben, der sich im Modus write-only befindet. Dieser Speicherbereich wird in Folge dessen im read-only-Modus dem Fragment-Shader für die Filterung in y-Richtung übergeben, dessen Ergebnisse wiederum in einen anderen Bereich des Speichers geschrieben werden. Nachdem die Filterung in x- und in y-Richtung durchgeführt wurde, muss dieses Ergebnis im Speicher gehalten werden. Diese erzeugte Textur dient wiederum als Ausgangstextur für die

nächste Gaußfilterung der jeweiligen Oktave. Jeweils zwei gefilterte Texturen werden im Speicher gehalten, damit daraus das Differenzbild berechnet werden kann. Das gaußgefilterte Differenzbild wird für die Detektion und Filterung von Extremwerten zurück in den Speicher der CPU geschrieben. Aus den berechneten, gaußgefilterten Bildern werden im Anschluss zwei weitere Texturen berechnet, die jeweils alle Orientierungen der Punkte und die Gradienten beinhalten. Diese beiden Texturen werden aus dem Speicher der GPU in den Speicher der CPU zurückgelesen. Diese Texturen dienen später dazu, die Schlüsselpunkte zu interpretieren und die Deskriptoren zu erzeugen.
Damit eine Bildpyramide aufgebaut werden kann, müssen die Bilddaten jeweils halbiert werden. OpenGL stellt für einen solchen Zweck die sogenannten MipMaps zur Verfügung. Dadurch kann im Vorfeld der verwendeten Texturen zwischen den bereitgestellten Texturgrößen gewählt werden und es müssen nicht extra die Buffer ausgelesen werden, um diese Berechnung durchzuführen. Jedoch sind solche MipMaps in ihren möglichen Größenverhältnissen eingeschränkt, da sie Größen aufweisen müssen, die durch eine 2er-Potenz darstellbar sind. Aus diesem Grund wurde auf die Verwendung von MipMaps verzichtet, die Bilder nach jeder Oktave halbiert und wieder in das Texturformat überführt. Diese umständliche Handhabung bei der Erzeugung einer Skalenraumpyramide hat jedoch nur sehr minimale Auswirkungen auf die Performance dieser Implemenation und wurde bis zum Ende hin beibehalten. Nach der Berechnung sämtlicher Differenzbilder, Orientierungs- und Gradientenhistogramme werden alle weiteren Berechnungen wie in der originalen Implementation nach David Lowe durchgeführt.
Abbildung 3.2 stellt den Ablauf des hier beschriebenen Ansatzes nochmals grafisch dar. Dabei wird deutlich, welche Menge an Daten in Form von Texturen zwischen den Buffern ausgetauscht werden müssen, um auf diese Art und Weise die Merkmale zu berechnen. Um weitere Berechnungen des SIFT-Verfahrens auf der GPU realisieren zu können, sind weitaus komplexere Shader zu realisieren.

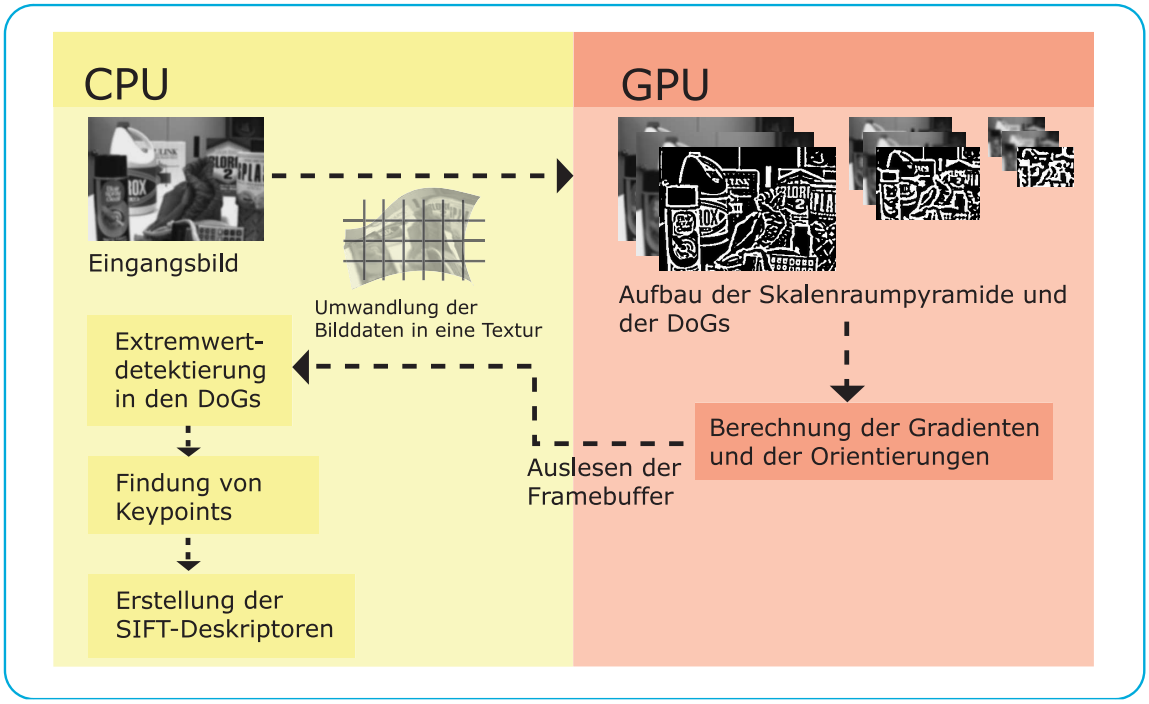


Bild 3.2: Auslagerung der Bereiche des SIFT-Verfahrens auf die GPU.

3.2 Beschreibung der erstellten GUI

Abbildung 3.3 zeigt die GUI mit ihrem Funktionsumfang. Die Nummerierung zeigt die Einteilung der Oberfläche in drei Bereiche. Über den Menüpunkt *File* lassen sich ganze Bildfolgen oder einzelne Bilder auswählen, die zur weiteren Bearbeitung angezeigt werden. Der Menüpunkt *Options* ermöglicht dem Benutzer Voreinstellungen des SIFT-Verfahrens unter ausschließlicher Nutzung der CPU zu tätigen. Da die Implementation unter Nutzung der GPU eine feste Kerngröße verwendet, kann man diesen Parameter auch für diese Implementation auswählen. Bei der Berechnung der Merkmale auf der CPU mit Hinzunahme der GPU werden die Parameter der Kerngröße und der Skalenanzahl standardmäßig auf den gleichen Wert gesetzt, damit sich die Ergebnisse zur Evaluation eignen. Zusätzlich kann der Benutzer die Anzahl der Skalen bei Berechnungen mit der originalen Implementation variieren, um so die Anzahl der zu findenen Merkmale zu erhöhen oder zu verringern. Abbildung 3.3 zeigt die Gliederung der grafischen Oberfläche in seine drei

Teilbereiche:

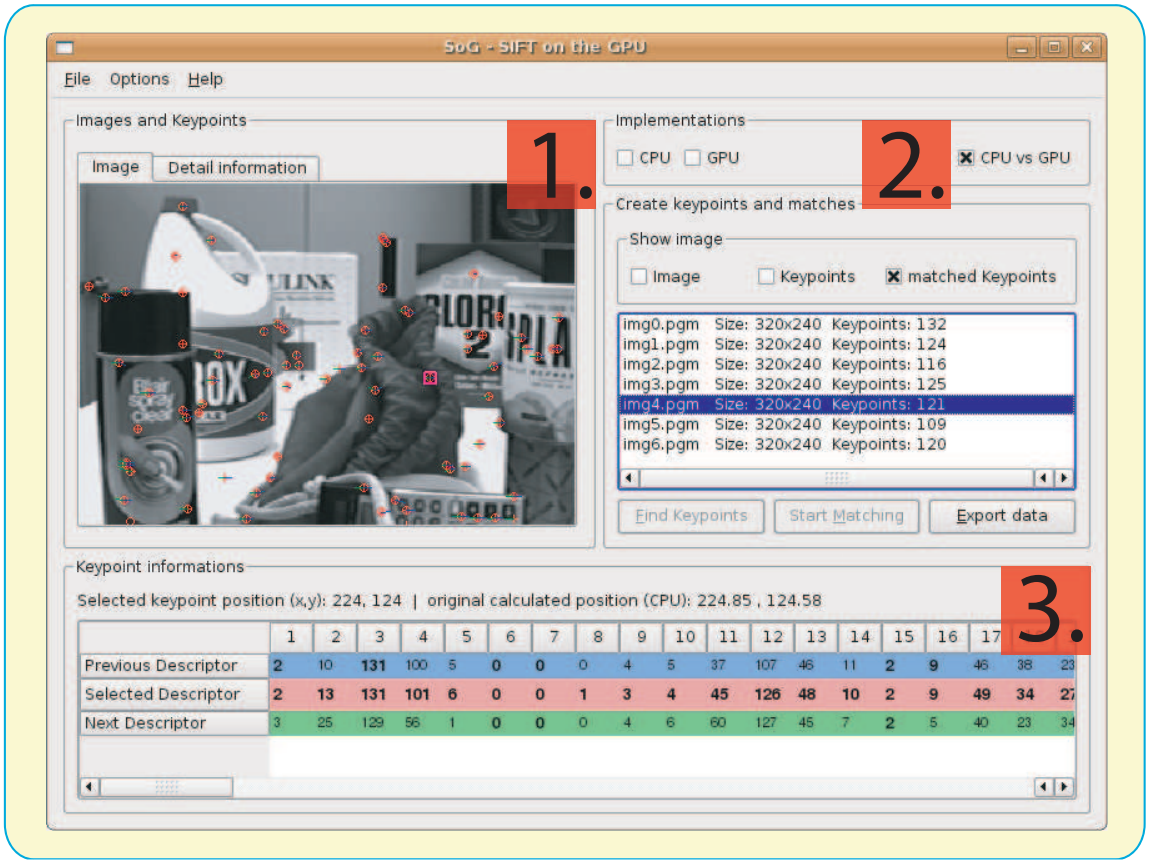


Bild 3.3: Darstellung der grafischen Oberfläche für die Visualisierung der Ergebnisse.

1. **Darstellung der Bilder und weiterer Informationen**
   Dieser Bereich zeigt das aktuell ausgewählte Bild. Wurden Merkmale berechnet, können diese hier angezeigt werden und die erzielten Ergebnisse, wie Anzahl der Merkmale und einzelne Zeitmessungen, werden im Reiter *Detail information* ersichtlich. Bei gewählter Funktion werden die Ergebnisse für Vergleichszwecke gegenübergestellt. Im Anzeigefenster der Bilder kann der Benutzer bei detektierten Merkmalen diese seperat per Mausklick auswählen. Nach getätigter Auswahl eines Merkmals werden Position, subpixelgenaue Koordinaten und der zugehörige Deskriptor im unteren dritten Bereich angezeigt.

2. **Auswahl der Implementation und Verwendung der Merkmale**
   In diesem Teilbereich der GUI kann der Benutzer die jeweilige Implementation wählen. Entweder verwendet er für die Berechnung der Merkmale die Implementation, die ausschließlich die CPU beansprucht, die Implementation die Teilbereiche des SIFT-Verfahrens auf der GPU auslagert oder er wählt *CPU vs. GPU*. Bei letzterem werden die Merkmale unter Verwendung beider Implementationen berechnet und deren Ergebnisse werden für weitere Funktionen zur Verfügung gestellt. Im unteren Teil des Bereichs kann der Benutzer Schlüsselpunkte mittels Klick des Buttons *Start matching* über eine gewählte Bildfolge zuordnen und die daraus resultierenden Deskriptoren bei Auswahl im Bildbereich anzeigen lassen. Zusätzlich kann der Benutzer die erzielten Ergebnisse in eine Textdatei exportieren, die für weitere Validierungszwecke verwendet werden kann.

3. **Darstellung der Deskriptoren und der Koordinaten**
   Dieser Teilbereich des Fensters zeigt den Deskriptor des im ersten Bereich ausgewählten Merkmals an. Bei erfolgreichem *Matching*[1] werden zusätzlich der vorherige und nachfolgende Deskriptor des eindeutig zugeordneten Merkmals angezeigt. Dadurch kann der Benutzer die Veränderungen im Deskriptor über eine Bildfolge beobachten. Ebenso werden in der Titelleiste des Bereichs die Bildkoordinaten des selektierten Merkmals, sowie die Koordinaten mit dem zugehörigem Subpixelbereich angezeigt.

[1] *Matching* bezeichnet die eindeutige Zuordnung von Schlüsselpunkten.

Kapitel 4

Experimente und Ergebnisse

4.1 Versuchsaufbau

Die Implementierung der Shader-Programme zur Berechnung bestimmter Bereiche des SIFT-Verfahrens auf der GPU erfolgt im Programmcode an den gleichen Stellen wie in der zu vergleichenden originalen Implementation, die ausschließlich die CPU zur Berechnung von Merkmalen benutzt. Die beiden Implementationen unterscheiden sich von den umgesetzten Algorithmen lediglich an den Stellen, an denen Bereiche komplett auf der GPU berechnet werden. Dadurch wird gewährleistet, dass die Ergebnisse konsistent validierbar bleiben und sich die Unterschiede beim Faktor Zeit und berechneter Merkmale nicht auf eine grundsätzlich andere Umsetzung im Programmcode zurückführen lassen. Für den Versuchsaufbau wurden verschiedene Sets von Bildern zusammengestellt, die sich pro Set weder in der Größe, noch im Bildinhalt unterscheiden. Die Verwendung solcher Bilder erlaubt es, definitive Aussagen über den zeitlichen Vergleich tätigen zu können, da sich unterschiedliche Motive in Bildern auf die Anzahl der berechneten Merkmale und somit auf die benötigte Zeit zur Berechnung der Schlüsselpunkte auswirken. Über solche Bildfolgen wird der Vergleich zwischen den verwendeten Implementationen hergestellt. Um den Subpixelbereich berechneter Merkmale zu untersuchen, sind eigene Bildersets erstellt worden, die genau bekannte Verschiebungen, Rotationen und Helligkeitsunterschiede aufweisen. Die Zeitmessungen werden an möglichst identischen Stellen im Programmcode

vorgenommen, damit die benötigte Zeit für die Messung und mögliche Rundungsfehler bei der Gegenüberstellung erzielter Ergebnisse keinen Einfluss haben.
Desweiteren wurde für den Versuchsaufbau ein Personal Computer mit folgenden Merkmalen verwendet:

- Prozessor: Athlon64 /Opteron
  - Speicher: 1GB RAM
- Grafikprozessor: GeForce 7600 GT
  - Bus Typ: PCI Express 16X
  - Speicher: 256 MB
  - Verwendete Treiberversion: 1.0-8776

4.2 Versuchsdurchführung

Bei der Zeitmessung verschiedener Bildgrößen mussten die jeweiligen Bilder einzeln zur Berechnung ausgewählt werden, da bei der Verwendung der GPU die benötigte Zeit für die Initialisierung der Buffer und Shader nicht in den Vergleich mit einfließen darf. Erwartet wird ein Bildstrom identischer Größe, damit die Ergebnisse für die Evaluation verwendet werden können. Eine unerwartete Erkenntnis führte bei der Versuchsdurchführung dazu, dass bei der Zeitmessung jedes Bild einer Größe zweimal zur Berechnung der Merkmale angewendet werden musste. Dies begründet sich darin, dass die verwendete Grafikhardware erst nach erstmaliger Verwendung die maximale Geschwindigkeit für Berechnungen erreicht. Diese Beobachtung ist in Abbildung 4.1 dargestellt. Aus diesem Grund wird für die Evaluation die Zeitmessung des jeweils ersten Bildes nicht berücksichtigt.
Die Untersuchung gefundener Merkmale in ihrem Subpixelbereich wird anhand des implementierten Matchings durchgeführt. Dabei werden Bilder, die Verschiebungen, bestimmte Rotationen und eingefügte Helligkeitsunterschiede beinhalten, für die Berechnung von Merkmalen verwendet. Daraufhin werden stichprobenartig Merkmale untersucht, die über die kompletten Bilder eindeutig ihren korrespondierenden Merkmalen zugeordnet werden

können. Diese Merkmale werden daraufhin in ihrem Subpixelbereich auf Konstanz und Invarianz gegenüber Störfaktoren wie Helligkeitsunterschiede, Rotationen und Translationen untersucht.

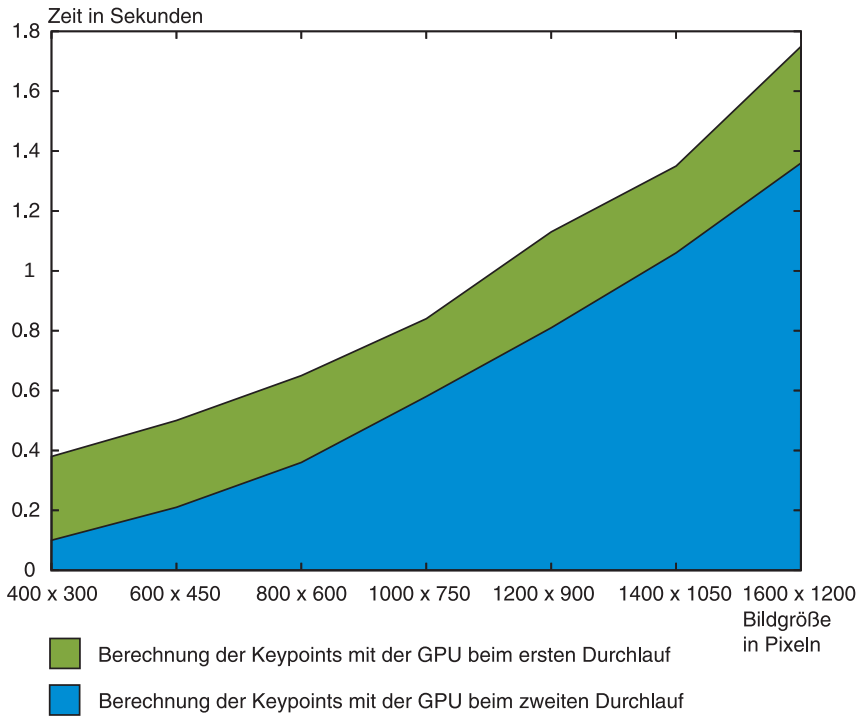


Bild 4.1: Zeitunterschiede in der Berechnung von Merkmalen auf der GPU für zwei identische Bilder.

4.3 Ergebnisse

4.3.1 Beschleunigung des SIFT-Verfahrens durch Nutzung der GPU

Abbildung 4.2 zeigt die zunehmende Zeitspanne für die Berechnung von Merkmalen mit der originalen Implementierung des SIFT-Verfahrens nach D. Lowe hinsichtlich steigender Bildgröße. Bei der vorgestellten Variante unter Verwendung der GPU für den Aufbau der Skalenraumpyramide, die Erzeugung der DoGs und die Berechnung der Orientierung

und der Gradienten der einzelnen Merkmale ist ebenso ein deutlicher Anstieg der benötigten Zeit zur Berechnung der Merkmale zu beobachten. Die in Abbildung 4.3 dargestellte Entwicklung der Anzahl von Merkmalen, bezogen auf die Größe der verwendeten Bilder, weist einen stetigen Anstieg auf. Beide Kurven weichen nur wenig voneinander ab. Dieser minimale Unterschied der berechneten Merkmale lässt sich durch die, im Gegensatz zur Originalimplementierung, nicht vorhandene Randbehandlung bei der Berechnung der Gaußbilder, der Orientierung und der Gradienten erklären. Aufgrund der Performance kann an dieser Stelle auf eine Behandlung von Sonderfällen verzichtet werden, da die erzielten Ergebnisse in der Berechnung der Merkmale dennoch ausreichend deckungsgleich und stabil sind. Setzt man die Abbildungen 4.2 und 4.3 in Verbindung, erkennt man, dass der Rechenaufwand bei ausschließlicher Nutzung der CPU genauso stetig ansteigt wie die Anzahl berechneter Merkmale. Durch die Verwendung der GPU ist dieser stetige Anstieg, im Gegensatz zur Menge berechneter Merkmale, nicht vorhanden. Hier zeigt sich, dass nur bestimmte Bereiche des SIFT-Verfahrens auf der GPU implementiert sind. Daraus lässt sich der deutlich geringere Anstieg erklären, da mit der steigenden Anzahl von Merkmalen der Aufwand für die Berechnung der Deskriptoren und die Detektierung vorhandener Extremwerte für beide Implementationen gleich bleibt, jedoch steigen die Berechnungen auf der GPU nicht so schnell in ihrem Aufwand.

In Abbildung 4.4 werden die prozentualen Anteile der einzelnen Bereiche des SIFT-Verfahrens dargestellt. Die y-Achse zeigt den prozentualen Anteil der vier Teilbereiche des SIFT-Verfahrens an, die jeweils für die Berechnung benötigt werden. 100% entsprechen der Zeit, die zur Berechnung aller Merkmaler benötigt wird. In der Abbildung 4.4 werden die prozentualen Anteile der Teilbereiche gemessen an der Gesamtzeit beider Implementationen für die drei Bildgrößen 400×300, 600×450 und 800×600 Pixel dargestellt. Diese Darstellungsform zeigt die zeitliche Entwicklung der Teilbereiche bei steigenden Bildgrößen im Verhältnis zueinander und wie hoch der Rechenaufwand für bestimmte Berechnungen auf welcher Prozessoreinheit ist.

Dabei werden die Verschiebungen der Anteile bei steigender Bildgröße unter Verwedung der GPU sowie ausschließlicher Verwendung der CPU deutlich. Unter Berücksichtigung der Tatsache, dass lediglich zwei Bereiche der hier dargestellten vier Teilbereiche des Verfahrens auf der GPU berechnet werden, wird deutlich, dass die Berechnungen der Orientierung und der Gradienten von Merkmalen auf der GPU den größten Geschwindigkeits-

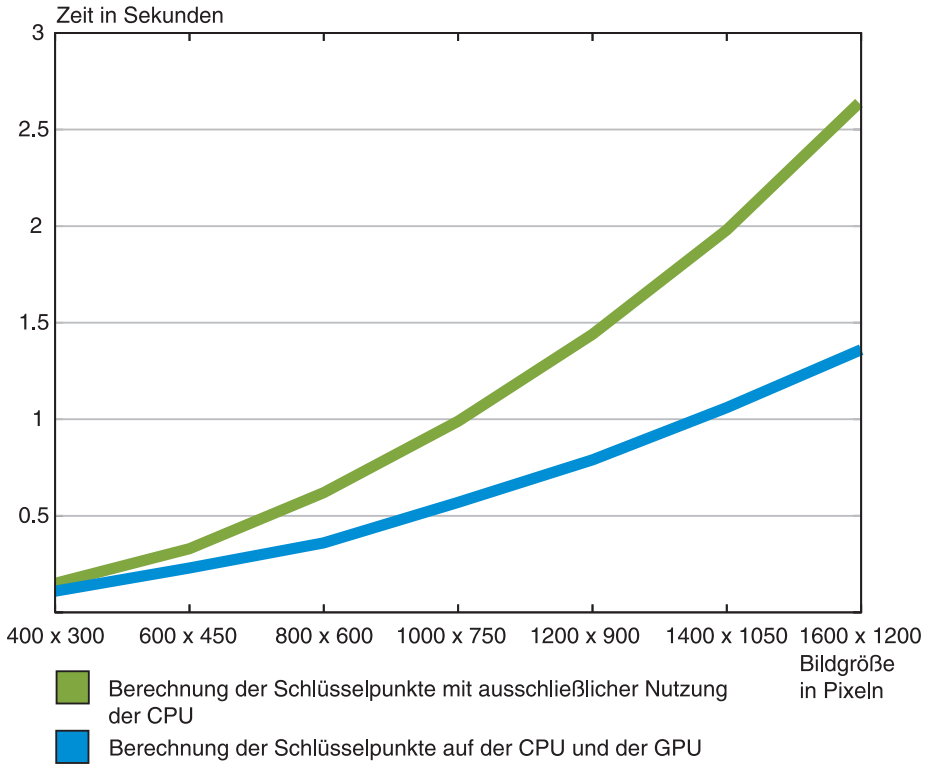


Bild 4.2: Benötigte Zeit zur Berechnung von Merkmalen in unterschiedlichen Bildgrößen.

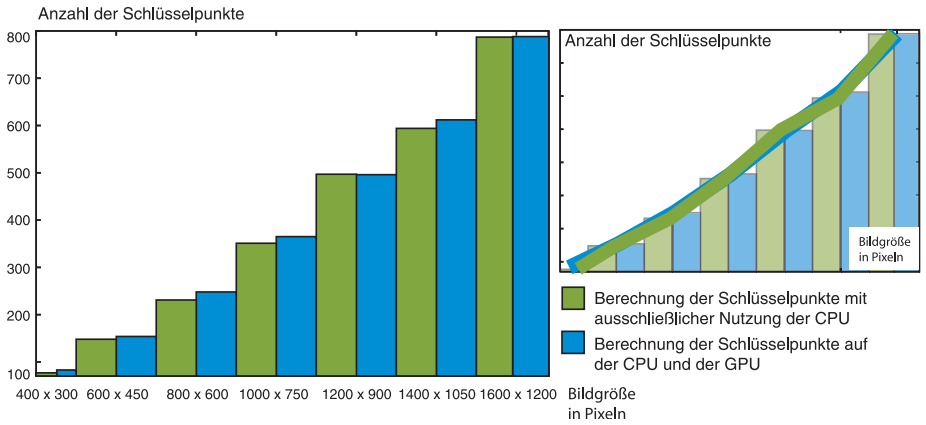


Bild 4.3: Anzahl der gefundenen Merkmale in unterschiedlichen Bildgrößen. Die rechte Grafik verdeutlicht den konstanten Anstieg der Anzahl berechneter Merkmale.

gewinn ausmachen. Mit steigender Bildgröße und steigender Anzahl der Merkmale bleibt der höchste Aufwand für die Implementation mit Nutzung der GPU bei der Berechnung der Skalenraumpyramide mit gaussgefilterten Bildern und bei der Erzeugung der DoGs. Die Implementation nach den Vorgaben von D. Lowe, beschrieben in [Low04], verzeichnet minimale Schwankungen in den Teilbereichen des SIFT-Verfahrens, wobei die Interpretation eines Schlüsselpunktes im Zusammenhang mit der Bestimmung von Extremwerten im Skalenraum und die Generierung der Deskriptoren einen deutlich geringeren Anteil an der aufzubringenden Zeit hat. Bei der Verwendung der GPU schwankt dieses Verhältnis wesentlich stärker, pendelt sich jedoch mit steigender Bildgöße bei einem Verhältnis von 1:1 ein. Diese Erkenntnis bestätigt das Ergebnis aus Abbildung 4.2, in der ein Geschwindigkeitszuwachs um 100 Prozent ersichtlich ist.

Die Grafik 4.5 zeigt die komplett benötigten Zeiten, die für die Berechnungen der augelagerten Bereiche auf der GPU, die Zeiten, die nur für das Zurücklesen der Buffer gebraucht werden, sowie den reinen Rechenaufwand ohne dass Zeit für das Zurücklesen der Daten verbraucht wird, um diese der CPU zur Verfügung zu stellen. Dabei ist der enorme Zeitaufwand für das Zurücklesen der Buffer auffällig. Obwohl die verwendete Grafikhardware an dem *PCIe*[1] Bus angekoppelt ist, geht sehr viel Zeit mit diesem Prozess verloren. Dieser enorme Zeitaufwand im vorhandenen Versuchsaufbau für das Auslesen der Framebuffer minimiert den Geschwindigkeitsgewinn, der sich von der Verwendung der GPU versprochen wird. Erkennbar ist auch der Zeitaufwand der reinen Berechnung auf der GPU gegenüber dem Zurücklesen der Buffer. Während der Zeitaufwand bei steigender Bildgröße für das Auslesen der Buffer stark ansteigt, steigt die benötigte Zeit der Berechnung lediglich im einstelligen Millisekundenbereich. Dieser Bottleneckeffekt ist in dieser Hardwareaustattung trotz PCIe vorhanden und minimiert stark die Geschwindigkeit bei einer solchen Implementation.

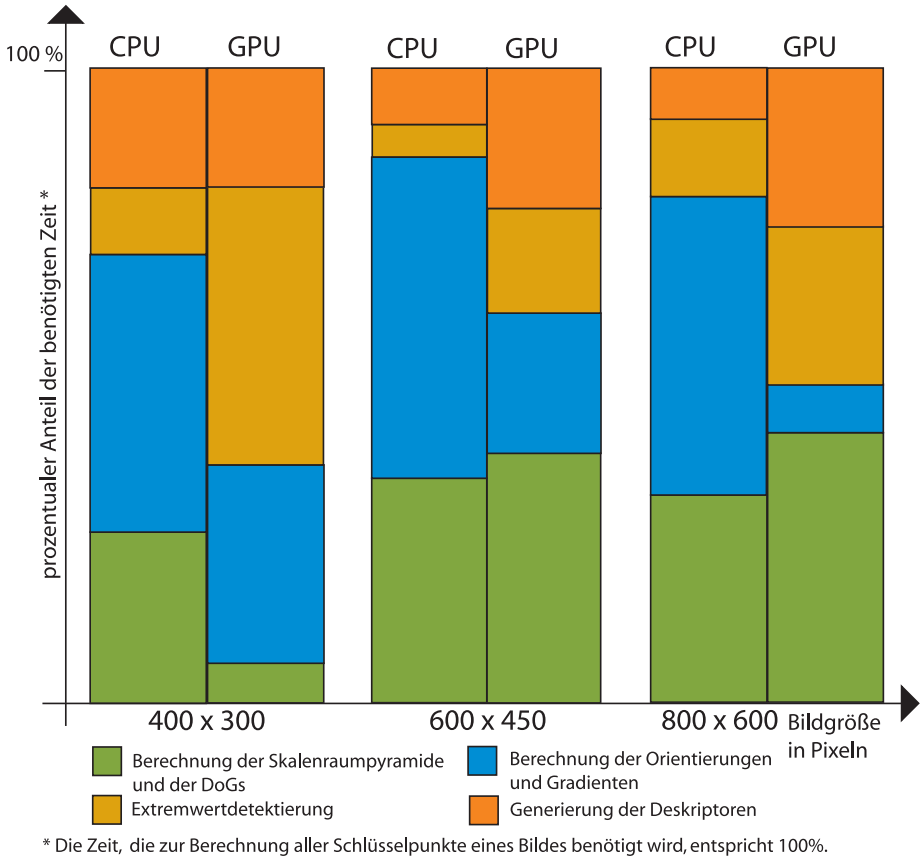


Bild 4.4: Vergleich der prozentualen Anteile der benötigten Zeit zur Berechnung verschiedener Teilbereiche des SIFT-Verfahrens.

4.3.2 Verhalten des Subpixelbereichs bei Störfaktoren

Dieser Unterabschnitt bezieht sich auf Genauigkeitsmessungen des Subpixelbereichs von berechneten Merkmalen. Die hier angeführten Ergebnisse sollen zeigen, welche Faktoren in Bildern den Subpixelbereich beeinflussen und als Störfaktoren gelten, die bei Verwendung des Subpixelbereichs für die Bestimmung von Tiefeninformationen und Kalibrierung von Stereokameras berücksichtigt werden müssen. Tabelle 4.6 zeigt einen Teil der Ergebnisse, die für die Evaluation des Subpixelbereichs herangezogen wurden. Die in der

[1]PCIe steht für *Peripheral Component Interconnect Express* und ist die Erweiterung des Standards zur Anbindung von Peripheriegeräten an den verwendeten Chipsatz.

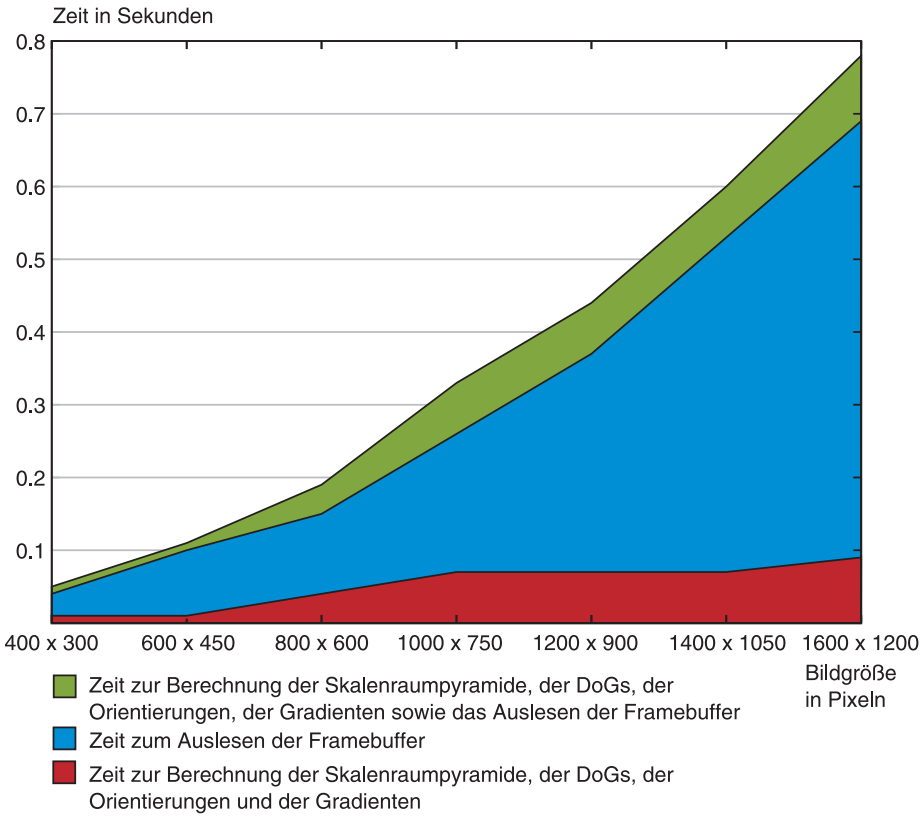


Bild 4.5: Benötigte Zeit die Daten von der GPU der CPU zur Verfügung zu stellen.

	Verschiebung der Bilder um jeweils 8 Pixel.		Konstante Aufhellung der Bilder um 10%.		Rotation des Bildes um den Punkt an der Stelle 641, 349.	
	x-Position	y-Position	x-Position	y-Position	x-Position	y-Position
Bild 1 - 4 ↓	777.775146	302.038574	656.900940	445.834839	641.888550	349.495300
	769.775146	302.038574	656.773743	445.835876	641.652527	350.374481
	761.775146	302.038574	656.988342	446.222656	641.422791	350.544281
	753.775146	302.038574	657.283020	446.905060	641.209656	350.692535

Bild 4.6: Verhalten im Subpixelbereich bei Störung der Pixelumgebung von Merkmale.

Tabelle 4.6 dargestellten Ergebnisse beziehen sich auf drei Störfaktoren, die sich auf den Subpixelbereich auswirken können. Die Störfaktoren sind Verschiebungen der Bildinhal-

te um den konstanten Faktor von acht Pixeln, Aufhellungen der Bilder um zehn Prozent, sowie Rotationen der Bilder um einen bestimmten Punkt, der gleichzeitig in den Bildern als Merkmal berechnet wird. Abbildung 4.7 veranschaulicht die eingefügten Störfaktoren, auf die sich Tabelle 4.6 bezieht. a) zeigt Bildausschnitte von Verschiebungen des Bildes in Richtung negativer x-Achse um jeweils 8 Pixel, b) veranschaulicht das Verhalten von Merkmalen bei konstanten Aufhellungen der Bilder um jeweils 10% und c) zeigt die Rotation um ein Merkmal.

1. **Verschiebung der Bilder um jeweils acht Pixel in negativer x-Richtung**
   Vergleicht man die x- und y-Werte nach jeder Verschiebung, zeigt sich die Stabilität der Positionswerte des beobachteten Merkmals in der Bildfolge. Es ändern sich lediglich die x-Werte, der Subpixelbereich bleibt identisch. Auf den ersten Blick scheinen die Merkmale in ihrem Subpixelbereich für eine weitere Verwendung der Kalibrierung eines Stereokamerasystems geeignet zu sein.

2. **Konstante Aufhellung der Bilder um zehn Prozent**
   Jedoch ist bereits in der Aufhellung der Bilddaten ein möglicher Störfaktor gefunden. Die Aufhellung der Bilddaten wird über die Bildfolge konstant bei 10 Prozent gehalten. Hier zeigt sich schnell, dass sich der Subpixelbereich nicht wie die Aufhellung in eine Richtung ändert, sondert dass hier die x-Werte doch deutlich schwanken. Ebenso verhalten sich die y-Werte, die auch keinen konstanten Faktor in ihrer Änderung aufweisen.

3. **Rotation der Bilder um ein berechnetes Merkmal**
   Die letzte Spalte der Tabelle 4.6 zeigt das Verhalten eines Merkmals, bei dem um die eigene Position des Merkmals im Ausgangsbild gedreht wird. Dabei scheint sich der Subpixelbereich der x-Werte um einen konstanten Faktor zu verändern, der sich auf eine Ungenauigkeit in der Positionierung des Rotationspunktes zurückführen lässt. Jedoch ist in den Änderungen der y-Werte im Subpixelbereich kein konstanter Faktor erkennbar.

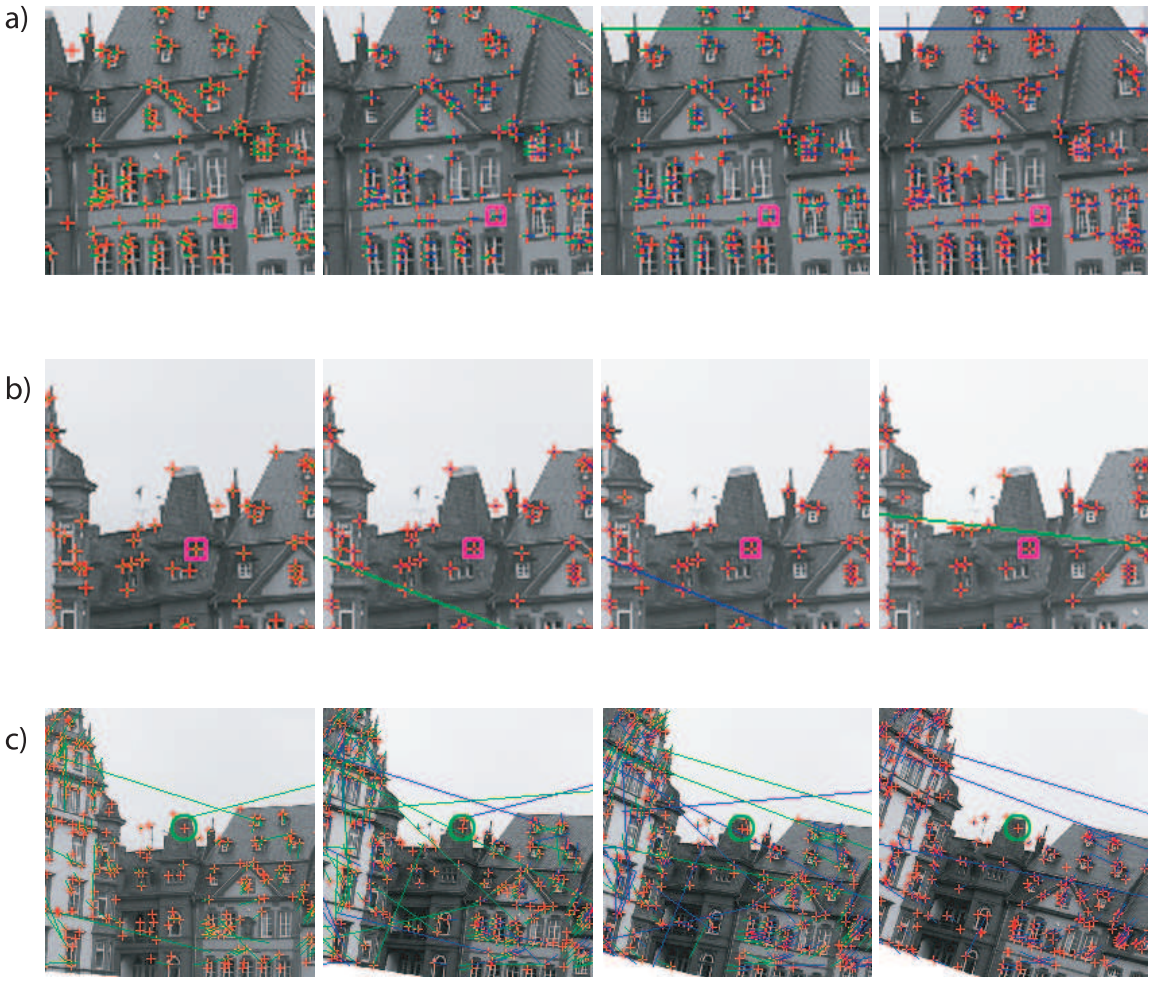


Bild 4.7: Beispielbilder für die Verschiebungen, Aufhellungen und Rotationen der in Tabelle 4.6 dargestellten Ergebnisse.

Während Verschiebungen der Bilder keinen Einfluss auf den Subpixelbereich haben, beeinflussen konstante Rotationen und Änderungen in der Helligkeit die Werte der Subpixel. Jedoch sind bei konstanten Rotationen und Helligkeitsveränderungen keine konstanten Veränderungen des Subpixelbereichs erkennbar. Erklären läßt sich dieses Ergebnis in der Tatsache, dass sich der Subpixelbereich aus den Positionen der Extremwerte in den jeweiligen Skalen berechnet, wobei leichte Änderungen im Ausgangsbild beim Aufbau der Skalenpyramide und den DoGs weitaus größere Auswirkungen haben, als es auf den er-

sten Blick erkennbar ist.
Tabelle 4.6 zeigt Entwicklungen im Subpixelbereich bei Störfaktoren wie Verschiebungen, Helligkeitsveränderungen und Rotationen. Diese Faktoren entstehen durch die Veränderungen des kompletten Bildes, bzw. durch Bewegen der Kamera, die die Bilder zur Berechnung von Merkmalen liefert. Dem gegenüber zeigt Tabelle 4.8 die Entwicklung der Koordinaten von Merkmalen, die sich durch Änderung ihrer Umgebung verändern. Im Abschnitt 4.1 wird kurz auf Störfaktoren in Bildern eingegangen. Störfaktoren können durch Änderungen der Kameraposition entstehen, aber auch durch die Szene selber. Aufgenommene Szenen können eigene, sich ändernde Objekte beinhalten. Ein Beispiel für ein solches Objekt, das die aufgenommene Szene beeinflusst, ist ein Fussgänger, der bei grüner Ampel über die Straße geht. Merkmale können sich auf dem Objekt befinden, dass sich durch die Szene bewegt, oder ein Objekt beeinflusst die Umgebung eines Merkmals, indem sich das Objekt in dessen Nähe bewegt. Zeigt Tabelle 4.6 noch Änderungen im Subpixelbereich der Koordinaten bei einfachen Verschiebungen, konstanten Aufhellungen der Bilddaten und Rotationen um einen bestimmten Punkt, werden die in dieser Tabelle dargestellten Koordinaten durch ein Objekt beeinflusst, das sich auf einem Hintergrund in x- und y-Richtung verschiebt. Das erste Merkmal befindet sich auf dem Objekt, verschiebt sich somit über den Hintergrund. Bei einfachen Verschiebungen des Hintergrundes ohne ein Objekt erzeugt dies keine Änderung des Subpixelbereichs, lediglich die Vorkommastellen, die absoluten Bildkoordinaten, ändern sich je nach Verschiebung. Die in der Tabelle dargestellten Werte zeigen diese Koordinaten und den dazugehörigen Subpixelbereich bei einfachen Verschiebungen bei zusätzlichen Änderungen des Pixelbereichs um das Merkmal herum. Dabei ändern sich die Koordinaten wie erwartet gemäß der Verschiebung, jedoch nimmt die Änderung der Umgebung ebenso Einfluss auf den Subpixelbereich. Da man die Änderung des Hintergrundes nicht vorhersagen kann, lassen sich auch keine konstanten Werteänderungen im Subpixelbereich erkennen.
Im zweiten dargestellten Merkmal handelt es sich um einen Punkt auf dem Hintergrund. Dieses Merkmal bleibt fest an seiner Position gebunden, jedoch wird dessen Umgebung durch das Objekt beeinflusst, welches durch das Bild, über den Hintergrund, geschoben wird. Auffällig hier sind die minimalen Änderungen im Subpixelbereich ab der Stelle 10^{-3}. Dies läßt sich jedoch mit dem Abstand des Objektes zum Merkmal erklären, da der Abstand auf eine geringere Änderung der Umgebung schließen läßt, die für die Be-

	Das Merkmal befindet sich auf dem zu verschiebenen Objekt.		Das Merkmal befindet sich fest im Hintergrund.	
Koordinaten des Merkmals in den jeweiligen Bildern der Bildfolge.	x-Position	y-Position	x-Position	y-Position
	698,458069	420,689941	778,178162	168,189758
	689,470276	420,986969	778,176208	168,186203
	679,482483	418,215210	778,175964	168,185806
	667,482666	416,166229	778,174988	168,184082
	661,566711	416,848297	778,174683	168,183441
	655,507446	416,190369	778,174622	168,183289
	642,301270	414,025665	778,174622	168,183273
	618,345215	416,905518	778,174622	168,183273
	605,235107	417,963074	778,174622	168,183273

Bild 4.8: Entwicklung des Subpixelbereichs bei provozierten Störungen im Bild.

rechnung des Subpixelbereichs verwendet wird. Trotzdem schwanken auch diese minimalen Änderungen. Allein in der x-Position ändern sich die Werte bis zum fünften Bild um 0,0034, während in den letzten vier Bildern keine Änderung der Werte zu verzeichnen ist. Dass sich der Wertebereich in den letzten vier Bildern nicht ändert, liegt am Objekt selber. Wie in Abbildung 4.9 ersichtlich ist, ist der Bereich des Objektes, der Einfluss auf diesen Bildbereich nimmt, ein konstanter, dunkler Bereich. In den ersten Schritten der Verschiebung ändert sich der Bereich, der für die Berechnung des Subpixelbereichs verwendet wird, stark, bleibt jedoch dann über eine bestimmte Anzahl von Verschiebungen konstant.

Neben dem untersuchten Verhalten des Subpixelbereichs bei Kamerabewegungen und Störungen der Pixelumgebung von Merkmalen wird das Verhalten des Wertebereichs bei Verschiebungen um einen halben Pixel untersucht. Dabei wird das Bild in negativer x-Richtung um einen Pixel verschoben und gleichzeitig die Auflösung halbiert.

Tabelle 4.10 zeigt die Änderungen der Koordinaten und des Subpixelbereichs anhand zwei ausgewählter Merkmale im Verlauf einer Bildfolge, die jeweils bei Halbierung der Auflösung um einen Pixel nach links verschoben werden. Die Verschiebung spiegelt sich in den Bildkoordinaten durch eine konstante Verschiebung um jeweils einen Pixel in den

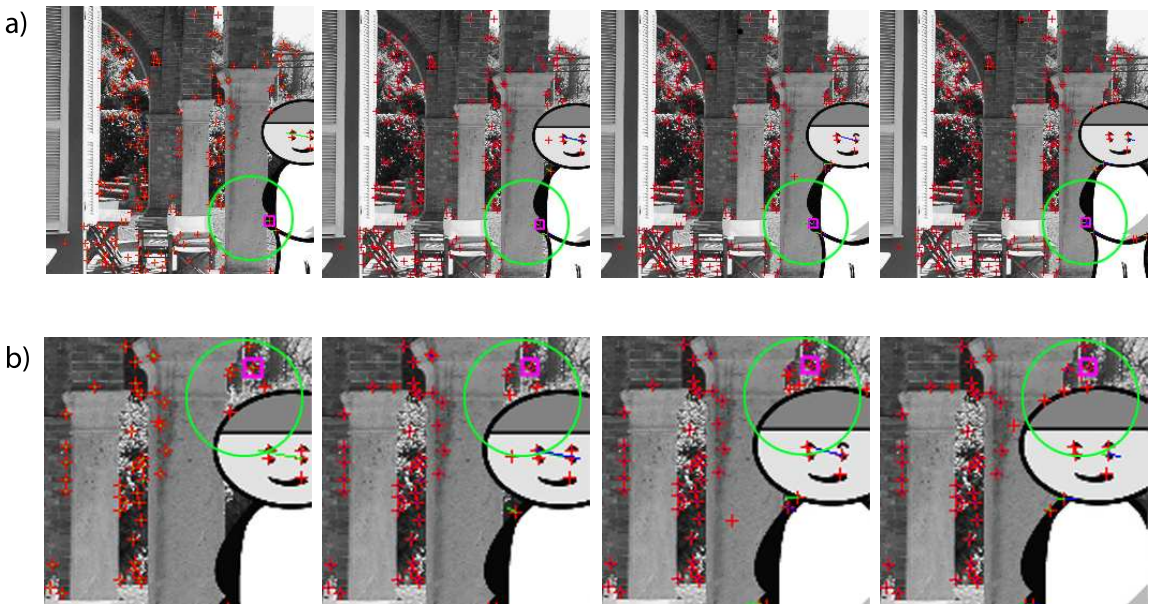


Bild 4.9: Beispielbilder für das Verhalten des Subpixelbereichs bei bewegten Objekten. Die Ergebnisse sind in Tabelle 4.8 dargestellt.

x-Werten wider. Gleichzeitig schwankt jedoch der Subpixelbereich in den x- und in den y-Werten deutlich. Folglich ist eine Änderung der Auflösung ein weiterer Störfaktor, der den Subpixelbereich beeinflusst.
Diese drei Tabellen untermauern die Aussage, dass der Subpixelbereich durch verschiedenste Störfaktoren beinflussbar ist, kein durchgehend konstantes Verhalten aufweisen kann und nicht für die Verwendung zur Tiefenberechnung geeignet ist. Jedoch können einige Störfaktoren bei der Kamerakalibrierung und Tiefenbestimmung ausgeschlossen werden, da nicht alle hier untersuchten Faktoren unter realen Bedingungen auftreten. Veränderungen von Bildern in ihrer Helligkeit und sich bewegende Objekte in einer Szene können den Subpixelbereich für die Verwendung der Kalibrierung und der Tiefenbestimmung negativ beeinflussen, indem sie die Ergebnisse der Berechnungen verfälschen. Für die Stereokamerakalibrierung kann mit der Forderung einer konstanten Helligkeit und Schaffung von Rahmenbedingungen, so dass keine sich bewegenden Objekte Einfluss auf die Szene nehmen dürfen, der Subpixelbereich die Genauigkeit erhöhen.

	Zwei Merkmale einer Bildserie mit jeweiliger Verschiebung in negativer x-Achse und Halbierung der Auflösung.			
Verschiebung um 1 Pixel ↓	x-Position	y-Position	x-Position	y-Position
2000 pixel / Inch	399,740082	243,400146	553,819885	423,400391
1000 pixel / Inch	398,848969	243,236313	552,871765	423,409851
500 pixel / Inch	397,756409	243,427200	551,916138	423,394775
250 pixel / Inch	396,828033	243,213959	550,947205	423,394928
125 pixel / Inch	395,753021	243,462479	549,965088	423,380707

Bild 4.10: Entwicklung des Subpixelbereichs bei Verschiebung und Änderung der Auflösung.

4.4 Validierung / Verifikation

Für die Validierung werden die Ergebnisse der Zeitmessungen der in dieser Arbeit vorgestellten Implementation mit den Ergebnissen der Implementation nach S. Heymann verglichen. Abbildung 4.11 zeigt die Gegenüberstellung der verglichenen Ergebnisse aus dieser Arbeit und aus der Arbeit von S. Heymann. Dargestellt werden die prozentualen Anteile der einzelnen Bereiche des Verfahrens gemessen an der Gesamtzeit zur Berechnung der Merkmale. Die Implementation mit ausschließlicher Nutzung der CPU, die in der Arbeit von S.Heymann für einen Vergleich verwendet wird, benötigt 400 msec zur Berechnung der Merkmale. Im Gegensatz dazu benötigt die Implementation, die im Rahmen dieser Arbeit für die Evaluation verwendet wird, lediglich 320 msec. Zu erklären ist dieser Unterschied in der Anwendung der Implementation nach D. Lowe, auf die sich die Vergleiche in dieser Arbeit stützen. Die Berechnung der gaußgefilterten Bilder wird mit einer festen Kerngröße von neun Gewichtungen in horizontaler und vertikaler Richtung in der Implementation dieser Arbeit durchgeführt. Aus diesem Grund wird für die Berechnung in der Implementation, die ausschließlich die CPU beansprucht, für Vergleichszwecke ebenfalls eine feste Kerngröße von neun verwendet. Durch die Anwendung des Verfahrens werden weniger Merkmale gefunden. Dies spiegelt auch die grafische Darstellung wider. Die

Anteile der Berechnungen für die Skalenraumpyramide, der Orientierungen und der Gradienten sind in Zeit umgerechnet bei gleicher Bildgröße fast identisch. Kleine Abweichungen im Millisekundenbereich lassen sich durch die unterschiedliche verwendete Hardware erklären. Lediglich die Bereiche, die durch die Anzahl möglicher Merkmale im Rechenaufwand stark beeinflusst werden, weichen deutlich voneinander ab. Trotzdem eignen sich die dargestellten Ergebnisse aus der Arbeit von S. Heymann, da diese Unterschiede nachvollziehbar und erklärbar sind.
Abbildung 4.11 zeigt zu genaueren Vergleichszwecken eine Skalierung der erzielten Ergebnisse von S. Heymann. Die skalierten Ergebnisse von S. Heymann sind in der Abbildung 4.11 transparent dargestellt. In der Skalierung werden die 100 Prozent der Zeitmessung der 60 msec auf die 100 Prozent der 210 msec skaliert, um die prozentualen Anteile der Gesamtzeit grafisch vergleichbarer darzustellen. Mittels dieser Gegenüberstellung wird ersichtlich, dass sich die Bereiche, die in dieser Implementation auf der GPU berechnet werden, einen weitaus größeren Anteil am Rechenaufwand einnehmen. Dies ist durch das Auslesen der Buffer zu erklären, dass einen großen Anteil in dieser gemessenen Zeit einnimmt. In diesem Zusammenhang zeigte die Abbildung 4.5 bereits, wieviel Zeit benötigt wird, um die Ergebnisse der Berechnung auf der GPU als Textur aus den Framebuffern zu lesen. Da bei einer Implementation, die so viele Berechnungen wie möglich auf der GPU durchführt, nur ein Bruchteil an Daten aus dem Framebuffer der CPU für eine Weiterverarbeitung zur Verfügung gestellt werden muss, minimiert sich dieser Anteil an Zeit und beschleunigt das Verfahren deutlich. Ersichtlich ist auch die Beschleunigung der Generierung der Deskriptoren, sowie die Detektion und Filterung der Extremwerte. Allein aus dem Grund das Auslesen der Framebuffer zu minimieren, müssen alle Bereiche des SIFT-Verfahrens ausgelagert werden, damit so wenig Daten wie möglich aus den Buffern der GPU der CPU zur Verfügung gestellt werden müssen. Auffällig in den Ergebnissen von S. Heymann ist die Verschiebung der prozentualen Anteile. Nehmen Aufbau der Skalenraumpyramide, Berechnung der Orientierungen und der Gradienten auf der CPU noch den größten Anteil des Rechenaufwandes ein, verschiebt sich dieses Verhältnis zur Extremwertdetektierung und -filterung und der Deskriptorgenerierung sehr stark. Auf der CPU kann man von einem Verhältnis 3:1 sprechen, wohingegen sich das Verhältnis bei der Implementierung auf der GPU in Richtung 1:3 bewegt. Die Berechnungen der Pyramide mit ihren gaußgefilterten Bildern, sowie die Berechnungen der Gradienten und Orientierungen

sind gut für einen Streamingprozessor geeignet. Bei diesen Bereichen des SIFT-Verfahrens lassen sich die Vorteile der GPU weitaus besser nutzen als bei der Detektierung und Filterung von Extremwerten. Was auf der CPU sehr geschickt und effizient programmiert werden kann, ist auf der GPU oftmals nicht so effizient umsetzbar. Dennoch ist es sehr effizient im Gesamtergebnis, wenn alle Bereiche auf die GPU ausgelagert werden, da mittels einer solchen Implementation der Rechenaufwand auf 15 Prozent reduziert werden kann.

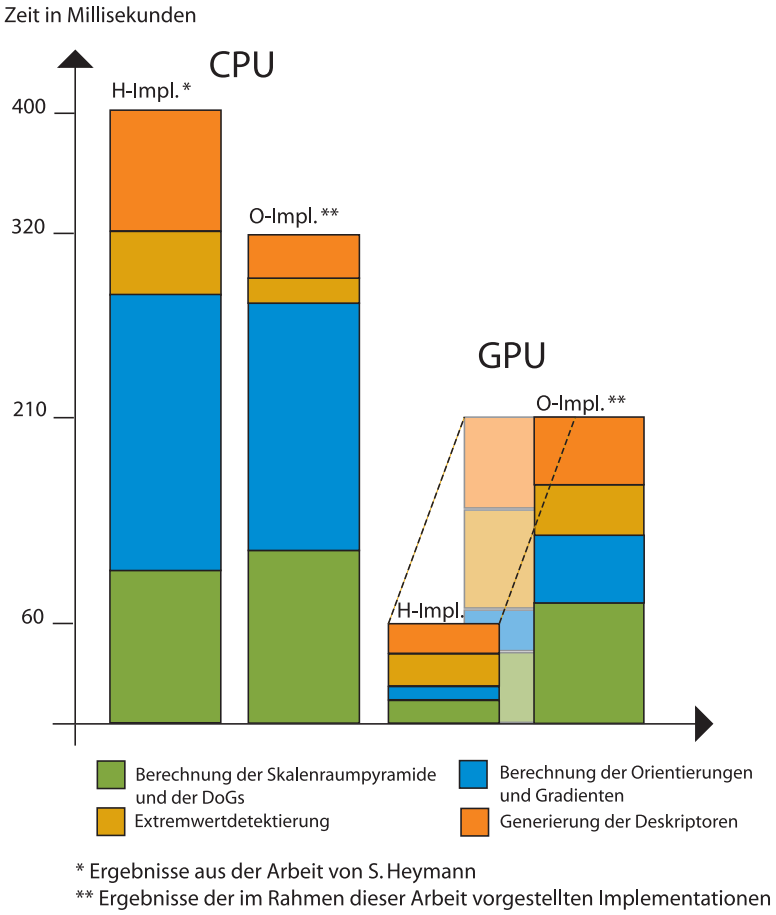


Bild 4.11: Gegenüberstellung der erzielten Ergebnisse mit den Ergebnissen von S. Heymann

Kapitel 5

Zusammenfassung

Die in Kapitel 4 vorgestellten Ergebnisse und deren Validierung zeigen die möglichen Unterschiede in der Performancegewinnung. Der im Rahmen dieser Arbeit vorgestellte Ansatz für mögliche Auslagerungen bestimmter Bereiche des SIFT-Verfahrens eignet sich nur bedingt für die Beschleunigung des Verfahrens, da die gewünschte Echtzeitanwendung mit einer solchen Implementation nicht erreicht wird. Damit dieses Verfahren in Echtzeit Merkmale in Bildern berechnet, müssen diese in einer Zeit von 50 Millisekunden berechnet werden, um ein Bildrate von 20 Bildern pro Sekunde zu erreichen. Diese Schnelligkeit ist durch die hier vorgestellte Implementierung nicht gegeben. Wie in Kapitel 4 beschrieben, liegt die Zeit zur Berechnung von Merkmalen bei Bildgrößen von 600 x 450 Pixeln mit benötigten 200 Millisekunden um das Vierfache darüber. Die Berechnung der ausgelagerten Bereiche dauert nur einen Bruchteil der Zeit, die auf der GPU dazu benötigt wird, die Ergebnisse der CPU zur Verfügung zu stellen. In diesem Zusammenhang wird deutlich, dass die Busanbindung mit dem verwendeten Treiber eine Schwachstelle dieser Implementation bildet. Bei Verwendung einer Bildgröße von 600 x 450 Pixeln liegt die benötigte Zeit für die Berechnung bei einem Zehntel der aufzubringenden Zeit, die Buffer auszulesen. Betrachtet man den Verlauf der Kurven bei steigenden Bildgrößen, ist ersichtlich, dass der Anstieg der benötigten Zeit für das Auslesen der Buffer gegenüber der reinen Rechenzeit wesentlich steiler verläuft. Diese Schwachstelle der Grafikhardware in Verbindung mit dem verwendeten Treiber minimiert den möglichen Geschwindigkeits-

zuwachs enorm. Die Implementationen nach Sinha [SFPG06] und S. Heymann [Hey05] umgehen diesen Bottleneckeffekt, indem alle Bereiche des SIFT-Verfahrens auf die GPU ausgelagert werden. Lediglich kleine Datenmengen werden schließlich der CPU zur weiteren Verarbeitung überlassen. Schaut man sich die Ergebnisse aus Abbildung 4.11 genauer an, ist bei den Ergebnissen von S. Heymann eine deutliche Wendung der Verhältnisse erkennbar. Gelten auf der CPU die Bereiche zur Berechnung der DoGs, der Orientierungs- und Gradientenhistogramme noch als die rechenintensivsten Teile des SIFT-Verfahrens, sind auf der GPU die Bereiche zur Detektierung und Filterung der Extremwerte, sowie die Generierung der Deskriptoren der Merkmale die rechenintensivsten Elemente. Folglich macht die Auslagerung aller Bereiche des SIFT-Verfahrens weitaus mehr Sinn, möchte man den vollen möglichen Geschwindigkeitszuwachs durch Verwendung der GPU erreichen. Bei Auslagerungen von ausgewählten Bereichen müssen zu viele Daten zwischen den verschiedenen Buffern hin und her geschoben werden. Diese Menge an Daten, in dieser Arbeit als Texturen deklariert, sollte bei der GPGPU so gering wie möglich gehalten werden, solange dieser Bottleneck bei der Anbindung von Grafikhardware an den Bus besteht.

Die Untersuchung der Merkmale in ihrem Subpixelbereich weist keine Konstanz bei Störungen der Bilddaten auf. Dabei ist natürlich zu beachten, dass es künstlich provozierte Störungen sind, die in Kapitel 4 durch die präsentierten Ergebnisse widergespiegelt werden. Aus diesem Grund sollte das Verhalten im Subpixelbreich von SIFT-Merkmalen unter realen Bedingungen untersucht werden. Dazu gehört unter anderem die Kalibrierung eines Stereokamerasystems mittels SIFT-Merkmalen und Verwendung ihres Subpixelbreichs. Jedoch wird aufgrund der in dieser Arbeit erzielten Beobachtungen davon ausgegangen, dass die Verwendung des Subpixelbereichs zu anfällig für Störungen ist, als dass Berechnungen unter Verwendung des Subpixelbereichs von SIFT-Merkmalen genauere Ergebnisse erzielen.

Literaturverzeichnis

[FK03] FERNANDO, Randima ; KILGARD, Mark J.: *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 0321194969

[Hey05] HEYMANN, Sebastian: *Implementierung und Evaluierung von Video Feature Tracking auf moderner Grafikhardware*, Bauhaus Universitaet Weimar, http://www.uni-weimar.de/, Diplomarbeit, 8 2005. http://www.uni-weimar.de/cms/Thesis_Implementierung_und_Ev.466.0.html

[Low04] LOWE, David G.: Distinctive Image Features from Scale-Invariant Keypoints. In: *International Journal of Computer Vision* 60 (2004), Nr. 2, 91-110. http://www.cs.ubc.ca/~lowe/papers/ijcv04.pdf

[Now04] NOWOZIN, Sebastian: *Autopano-SIFT (making panoramas fun)*. http://user.cs.tu-berlin.de/~nowozin/autopano-sift/. Version: 2004

[SFPG06] SINHA, Sudipta N. ; FRAHM, Jan-Michael ; POLLEFEYS, Marc ; GENC, Yakup: GPU-based Video Feature Tracking And Matching / UNC Chapel Hill Computer Science. Version: 5 2006. fish://serres/lab/as/Docs/Protected/Sinha2006GVF.pdf. Department of Computer Science, CB 3175 Sitterson Hall, University of North Carolina at Chapel Hill, NC 27599, 5 2006. – Forschungsbericht

[TK91] TOMASI, Carlo ; KANADE, T.: Detection and Tracking of Point Features / Carnegie Mellon University. 1991 (CMU-CS-91-132). – Forschungsbericht

[TRFT98] TRUCCO, E. ; ROBERTO, Vito ; FUSIELLO, Andrea ; TOMMASINI, Tiziano: Making Good Features Track Better. In: *IEEE Conf. Computer Vision and Pattern Recognition* (1998), 178-183. citeseer.ist.psu.edu/article/tommasini98making.html