Master Thesis zum Thema

GPU-unterstützte Bildverarbeitung und Bilderkennung im Kontext einer Multi-Touch-Anwendung

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)
in Computer Science and Media

vorgelegt dem
Fachbereich Druck und Medien
der Hochschule der Medien, Stuttgart

Patrick Bader
31. August 2010

Erstkorrektor: Prof. Dr. Jens-Uwe Hahn
Zweitkorrektor: Prof. Walter Kriha

Abstract

Diese Thesis beschreibt die Entwicklung von GPU-basierten Bildverarbeitungs- und Erkennungsalgorithmen zur Beschleunigung kamerabasierter Multi-Touch-Bildschirme. Dazu wurde der Prototyp eines Bildschirms entwickelt, der zusätzlich zur Erkennung von Berührpunkten die Detektion bestimmter Marker erlaubt. Im Gegensatz zu großen Multi-Touch-Tischen, die einen Beamer zur Darstellung nutzen, war der Bau eines kleinen Bildschirms, basierend auf einem 17 Zoll LCD, mit den Ausmaßen eines Röhrenbildschirms das Ziel.

Die Berührpunkte und Marker werden dabei mit Hilfe einer Infrarotkamera optisch erfasst. Aus den Bildern der Kamera können deren Koordinaten und im Fall der Marker zusätzlich eine ID extrahiert werden. Ziel der Arbeit war dabei, die Erkennung der Berührpunkte und Marker durch den Einsatz moderner Grafikprozessoren zu beschleunigen. Dazu wurden Algorithmen entwickelt, die auf die hochgradig parallelisierte Architektur der Grafikkarten angepasst sind und deren enorme Rechenleistung ausnutzen. Diese Algorithmen nutzen den offenen OpenCL-Standard, der die Durchführung von generellen Berechnungen auf Grafikhardware erlaubt.

Durch eine zusätzlich entwickelte Applikation können die einzelnen Berechnungsschritte der Algorithmen in Echtzeit visualisiert und nachvollzogen werden.

Danksagung

Mein herzlicher Dank geht an alle, die mich während der Erstellung dieser Arbeit unterstützt haben.

Insbesondere möchte ich meinen Eltern, Marianne und Robert Bader, meinem Bruder, Holger Bader, und meiner Freundin, Maritta Weiß, die mich stets unterstützt und die Arbeit korrekturgelesen haben, bedanken.

Für weiteres Korrekturlesen bedanke ich mich zudem bei Jessica Schön.

Zudem geht mein Dank an meinen Bruder, Ralf Bader, für das Bedrucken der beiliegenden DVD.

An dieser Stelle möchte ich mich nochmals besonders bei meinem Vater für die technische Unterstützung beim Bau des Prototyps und speziell für das Schweißen des Rahmens bedanken. Ohne seine Unterstützung wäre ein Bau in dieser Form nicht möglich gewesen.

Last but not least möchte ich mich bei Prof. Dr. Jens-Uwe Hahn für die Betreuung der Arbeit und den großen Freiraum bei ihrer Ausgestaltung, sowie bei Prof. Walter Kriha für die Zweitkorrektur der Arbeit bedanken.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Hintergrund . . . 1
- 1.2 Motivation . . . 2
- 1.3 Ziele . . . 3
- 1.4 Überblick . . . 3

2 Grundlagen **5**
- 2.1 Multi-Touch LC-Display . . . 5
  - 2.1.1 Aufbau . . . 5
  - 2.1.2 Funktionsweise . . . 8
  - 2.1.3 Erster Prototyp . . . 10
- 2.2 Bilderkennung und -verarbeitung . . . 10
  - 2.2.1 Faltung . . . 12
  - 2.2.2 Glättung . . . 15
  - 2.2.3 Kantendetektion . . . 19
- 2.3 Marker . . . 23
  - 2.3.1 Marker für Menschen . . . 24
  - 2.3.2 Maschinelle Marker . . . 24
  - 2.3.3 Strichcodes . . . 25
  - 2.3.4 AR-Marker . . . 26
  - 2.3.5 Multi-Touch-Marker . . . 27
- 2.4 CPU-Architektur vs. GPU-Architektur . . . 29
  - 2.4.1 Parallelisierung . . . 29
  - 2.4.2 Speicherzugriff . . . 30
  - 2.4.3 Speicherallokation . . . 32
  - 2.4.4 Zusammenfassung und weitere Entwicklung . . . 32

3 Implementierung **35**
- 3.1 Technologien . . . 35
  - 3.1.1 OpenCL . . . 36
  - 3.1.2 OpenGL . . . 43
  - 3.1.3 Interoperabilität . . . 44
- 3.2 GUI-Framework . . . 45
  - 3.2.1 Komponenten . . . 46

3.2.2 Ereignisse . . . 47
3.2.3 Rendern . . . 49
3.3 Vorverarbeitung . . . 51
3.4 Berührpunkterkennung . . . 55
3.4.1 Bestehende Implementierungen . . . 56
3.4.2 GPU-Algorithmus . . . 56
3.4.3 Performance . . . 60
3.5 Markererkennung . . . 61
3.5.1 Bestehende Implementierungen . . . 61
3.5.2 Marker . . . 62
3.5.3 GPU-Algorithmus . . . 62
3.5.4 Performance . . . 66
3.6 Zurücklesen der Ergebnisse . . . 67
3.7 Applikation . . . 67

4 Evaluation **71**
4.1 Performanceanalyse . . . 71
4.2 Prototyp . . . 76

5 Fazit **79**
5.1 Ausblick . . . 79

A Erklärung **81**

Abbildungsverzeichnis **83**

Listings **85**

Literaturverzeichnis **87**

Kapitel 1

Einleitung

1.1 Hintergrund

Nach vielen Jahren, in denen die Mensch-Maschine-Interaktion von der Benutzung von Maus und Tastatur geprägt war, drängen seit kurzem neue Bedienkonzepte in den Massenmarkt. Im Endkundenmarkt betrifft dieser Trend überwiegend Spielekonsolen und mobile Geräte. Im Geschäftskundensektor genießen große Geräte für mehrere simultane Benutzer, wie interaktive Tische, steigende Popularität.

Bei den Spielekonsolen begann die Suche nach neuen Bedienkonzepten mit der Wii-Konsole von Nintendo. Der Controller wurde im Wesentlichen mit einem Beschleunigungssensor und einer IR-Kamera erweitert (Seidle 2006). Mit Hilfe des Sensors lassen sich Bewegungsabläufe des Benutzers grob abschätzen[1]. In Kombination mit der auf dem Fernseher angebrachten Sensor Bar[2] wird die Kamera benützt, um mit dem Controller auf virtuelle Objekte zeigen zu können.

Diese Art der Interaktion lockt besonders Casual Gamer[3] wegen der intuitiven Bedienung an. Auch die Konkurrenten Sony und Microsoft sehen in den Casual Gamern einen großen Markt und wollen an den Erfolg von Nintendo anknüpfen (vgl. Sony 2010; Microsoft 2010). Die Antwort von Sony hört auf den Namen Move und ist dem Nintendo Controller nachempfunden. Dabei wird die Kamera jedoch am Bildschirm platziert und erkennt die Position der unterschiedlich leuchtenden Controller im Raum. Zur Erkennung der exakten Lage des Controllers kommen Beschleunigungs- und Rotationssensoren zum Einsatz (Gieselmann 2010). Microsofts Bedienkonzept, Kinekt, verzichtet komplett auf den Einsatz von Controllern. Eine Kamera filmt den Benutzer, erkennt dabei dessen Bewegungen und reagiert auf Gesten.

Auch im Bereich der mobilen Geräte, insbesondere der Smartphones, lässt sich in zweierlei Hinsicht ein Wandel in der Bedienung erkennen. Die Navigation mit den

[1] Der Sensor nimmt Beschleunigungen in drei Achsen wahr. Rotationen des Controllers können nicht erfasst werden.

[2] Die Bezeichnung Sensor Bar ist irreführend, da die Leiste keine Sensoren enthält, sondern lediglich IR-LEDs.

[3] Casual Gamer = Gelegenheitsspieler.

sehr kleinen Ziffertasten wird zunehmend schwieriger, je reichhaltiger die Benutzeroberflächen werden. Durch relativ große Bildschirme in den heutigen Smartphones erlebt die schon in die Jahre gekommene Technik der berührungsempfindlichen Displays eine Renaissance. Diese Technik wurde um die Fähigkeit mehrere Berührpunkte simultan zu erfassen erweitert und mit der Bezeichnung Multi-Touch ein Verkaufsargument.

Ein weiterer Durchbruch im Smartphonebereich ist die Augmented Reality[4]. Ermöglicht wird dies durch eine Kamera auf der Rückseite des Smartphones. Die Umgebung wird vom Gerät aufgezeichnet, ausgewertet und auf dem Display in Echtzeit angezeigt. Im angezeigten Bild können dann, in Kombination mit GPS, Zusatzinformationen eingeblendet werden. Anwendungsbeispiele, wie digitale Stadtführer, die Informationen zu lokalen Unternehmen und sehenswerten Orten anzeigen, gibt es viele. Zur Vereinfachung der Objekterkennung werden in der Augmented Reality oft Marker eingesetzt. In ihnen können Informationen, wie eine Objekt ID, hinterlegt werden und ihre Form ermöglicht die Bestimmung deren Lage im Raum.

Die Möglichkeiten für die Augmented Reality auf mobilen Geräten sind im Moment jedoch durch die geringe Leistungsfähigkeit der Geräte beschränkt. Durch die Komplexität der notwendigen Bilderkennungsalgorithmen stößt die Augmented Reality schnell an ihre Grenzen. Zudem steigt durch intensive Berechnungen auf den Geräten der Stromverbrauch worunter die Akkulaufzeit erheblich leidet.

Jenseits des Endkundenmarktes findet eine andere Entwicklung statt, die aufgrund der hohen Kosten auf Geschäftskunden ausgelegt ist. Die Idee ist Interaktion mit Computern ohne klassische Monitore mit Maus und Tastatur. Große interaktive Tische oder Wände finden zunehmend auf Messen Verbreitung. Die Bedienung erfolgt wie auch bei den Smartphones durch Multi-Touch-Eingaben, unterscheidet sich jedoch in ihrer Ausrichtung. Am Multi-Touch-Tisch sollen mehrere Benutzer gleichzeitig interagieren können, der Computer wird Multi-User fähig. Ein bereits verfügbarer Tisch wurde von Microsoft entwickelt und ist unter dem Namen Surface erhältlich (Microsoft 2009). Die Erkennung von Berührungen erfolgt optisch, im Gegensatz zu mobilen und Desktop Multi-Touch Monitoren[5]. IR-Kameras filmen dabei die Bildfläche ausgehend vom Boden des Tisches. Berührpunkte erscheinen auf dem Bild der Kamera als helle Bereiche. Mit Hilfe von Bildverarbeitungs- und Bilderkennungsalgorithmen werden aus den von den Kameras aufgenommenen Bildern Positionen extrahiert. Als Erweiterung ist bei manchen optischen Verfahren, wie der Surface, auch Marker- und Objekterkennung möglich.

1.2 Motivation

Für alle genannten Technologien spielt Bildverarbeitung und -erkennung eine entscheidende Rolle. Dabei ist die Anwendung der Algorithmen in Echtzeit eine der

[4]Augmented Reality (AR) beschreibt die Anreicherung der tatsächlich wahrgenommen Realität durch virtuelle Elemente.

[5]Dort werden induktive oder kapazitive Verfahren eingesetzt.

größten Herausforderungen dieser Systeme. Echtzeit bedeutet, es müssen mindestens 25 Bilder pro Sekunde verarbeitet werden können. Je mehr Bilder in der Sekunde verarbeitet werden, desto geringer ist die wahrgenommene Verzögerung und Bewegungen erscheinen flüssiger.

Im Bereich der Multi-Touch-Eingabegeräte hat sich in den letzten Jahren eine Community gebildet, die versucht selbst große Tische zu bauen. Dabei schränkt ein begrenztes finanzielles Budget die Möglichkeiten der Entwickler ein. Aus diesem Grund wird anstatt spezieller Hardware bestehende Low-Cost Hardware umgebaut. Beispiele sind Beamer oder auch Webcams, die zu IR-Kameras umgebaut werden. Zum Tracking von Punkten oder Markern wurden diverse Bibliotheken, in der Regel als Open-Source, programmiert. Dabei fehlt oft die Unterstützung für Markererkennung oder die Verarbeitungsgeschwindigkeit der Bildverarbeitung ist grenzwertig.

1.3 Ziele

Im Rahmen dieser Master Thesis wird evaluiert, ob die für das Tracking notwendige Bildverarbeitung und -erkennung in optischen Multi-Touch-Systemen mit Hilfe moderner Grafikkarten beschleunigt werden kann. Ziel ist, mindestens 100 Bilder in der Sekunde verarbeiten zu können und es sollen sowohl Berührpunkte als auch Marker, in denen Informationen hinterlegt sind, erkannt werden. Für die Marker muss ein geeignetes Format, das eine schnelle Erkennung ermöglicht, entworfen werden.

Zum Test des Systems wird der Prototyp eines optischen Multi-Touch-Bildschirms gebaut. Dieser soll auf einem bestehenden 17-Zoll-LCD beruhen.

Um von der Rechenleistung der Grafikkarte profitieren zu können, müssen speziell auf die Grafikkartenarchitektur zugeschnittene Algorithmen entworfen werden.

Zusätzlich wird ein GUI-Framework[6] entwickelt, um die einzelnen Schritte der Algorithmen visualisieren zu können. Dabei spielt auch das Zusammenspiel der unterschiedlichen Grafikschnittstellen eine wichtige Rolle.

1.4 Überblick

Im folgenden Kapitel werden die Grundlagen für die Entwicklung erläutert. Dazu gehört der Aufbau und die Funktionsweise des, im Rahmen der Thesis entwickelten, Multi-Touch-Bildschirmprototyps. Für das spätere Verständnis der implementierten Algorithmen werden die verwendeten Bildverarbeitungsalgorithmen vorgestellt und ein Überblick über Markerarten und deren Eigenschaften gegeben. Abschließend erfolgt ein Vergleich der GPU-Architektur mit der CPU-Rechnerarchitektur, in welchem Besonderheiten bei der Programmierung für beide Systeme hervorgehoben werden.

Kapitel 3 behandelt die Implementierung der Blob[7]- und Markererkennung auf Grafikhardware. Dabei werden die eingesetzten Technologien beschrieben. Auf den

[6] GUI = Graphical User Interface.

[7] Bezeichnung für einen Berührpunkt.

Technologien aufbauend, werden die entwickelte grafische Benutzeroberfläche und die Algorithmen vorgestellt.

Im 4. Kapitel wird eine Evaluation der Implementierung und des Prototyps durchgeführt. Diese beinhaltet Geschwindigkeitsmessungen im Vergleich zu bestehenden Trackingverfahren, sowie mögliche Erweiterungen und Verbesserungen sowohl des Prototyps als auch der Implementierung.

In einem Fazit werden abschließend in Kapitel 5 die Ergebnisse zusammengefasst und ein genereller Ausblick in die Zukunft der GPU-Programmierung gegeben.

Kapitel 2

Grundlagen

2.1 Multi-Touch LC-Display

Im Low-Cost Bereich existieren mehrere optische Multi-Touch-Verfahren, welche Infrarotkameras zum Tracking von Berührpunkten nutzen. Die Ausgabe von Bildern erfolgt in der Regel mit Hilfe eines Beamers. Dadurch können sehr große Bilddiagonalen erzielt werden. Nachteilig auf die Gesamtgröße des Bildschirms wirkt sich dabei der benötigte Mindestabstand des Beamers zur Projektionsfläche aus.

Ein Beispiel hierfür ist das, von Jefferson Han erfundene, FTIR[1]-Verfahren (Han 2005), das den optischen Effekt der Totalreflexion nützt. Mit Hilfe dieses Verfahrens können zwar Berührpunkte, jedoch keine Objekte oder Marker, die sich auf der Bildoberfläche befinden, erkannt werden.

Dieses und weitere Verfahren, wie Diffused Illumination (DI) oder Diffused Surface Illumination (DSI), welche sich auch zur Markererkennung eignen, sind im Buch "Multitouch Technologies" der NUI Group ausführlich beschrieben (NUI Group Authors 2009).

2.1.1 Aufbau

Im Gegensatz zum Großteil der bestehenden optischen Systeme, die Beamer zur Bildausgabe verwenden, basiert der im Folgenden beschriebene Prototyp auf einem LC-Bildschirm[2]. Zum Einsatz kommt aus Kostengründen der mit 17 Zoll Bilddiagonale relativ kleine Bildschirm SDM-S71R von Sony. Für die Erkennung wird die PlayStation® Eye Kamera, ebenfalls von Sony, eingesetzt. Sie ist in der Anschaffung günstig und ist in der Lage, 60 Bilder in der Sekunde, bei einer Auflösung von 640x480 Bildpunkten aufzuzeichnen.

Der schematische Aufbau des selbstgebauten Bildschirmprototyps ist in Abbildung 2.1 dargestellt. Die Hauptkomponente ist das für die Darstellung benötigte LC-Panel. Es ist zusammen mit zwei Plexiglas® EndLighten Platten am Rahmen des Bildschirms befestigt. Alle drei Schichten liegen direkt aufeinander, sind jedoch

[1] FTIR = Frustrated Total Internal Reflection = Totalreflexion.

[2] LC = Liquid Crystal = Flüssigkristall.

zur besseren Übersicht in der Abbildung mit einem Abstand zueinander dargestellt.

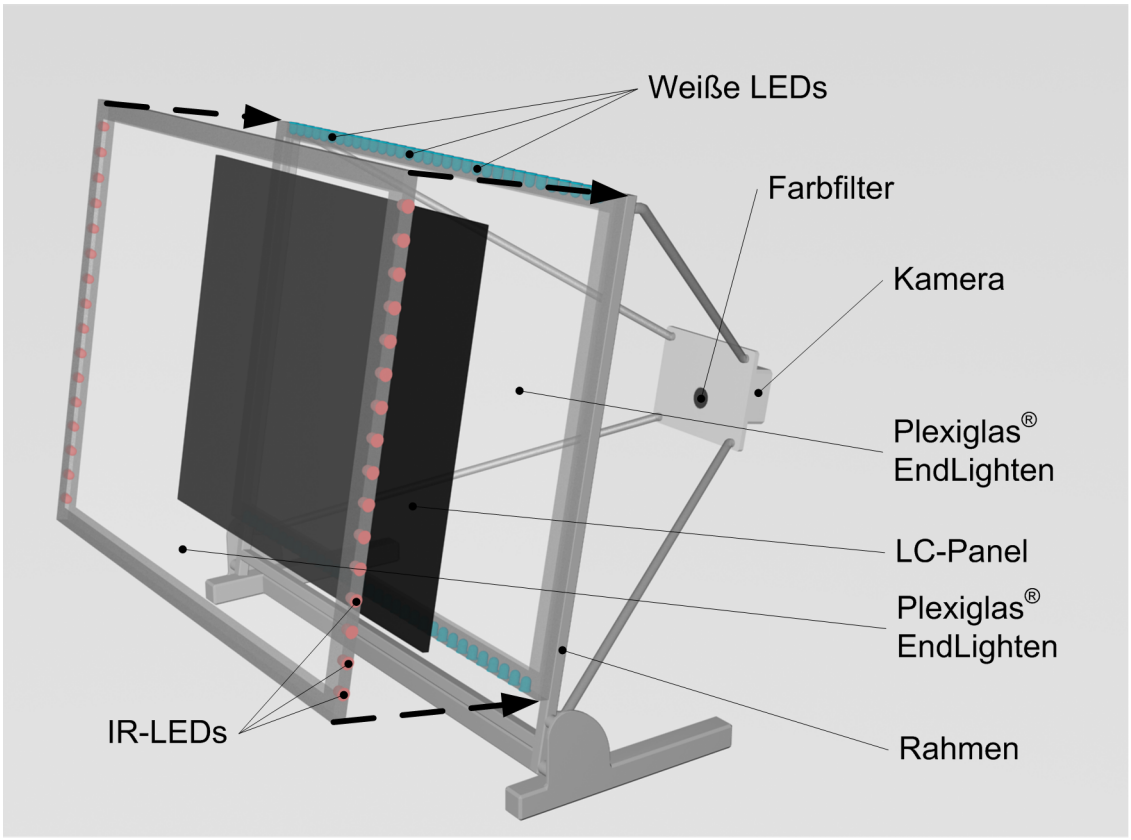


Abbildung 2.1: Schematischer Aufbau des Bildschirmprototyps.

LC-Panel

Wie bereits erwähnt ist das LC-Panel das zentrale Element für die Darstellung und wurde aus dem ursprünglichen Bildschirm ausgebaut, siehe Abbildung 2.2 a). Zur Beleuchtung wurden im ursprünglichen Aufbau Kaltlichtkathoden eingesetzt. Die Verteilung des Lichts erfolgte über ein spezielles Acrylglas mit diffus abstrahlendem Quadratmuster, das auf weißem Papier aufliegt. Zwischen Acryl und Panel befinden sich zahlreiche Diffusorfolien, die für eine gleichmäßige Ausleuchtung sorgen. Die Diffusorfolien und das Acrylglas sind in Abbildung 2.2 b) bzw. c) dargestellt.

Diese Art der Beleuchtung ist aus zwei Gründen für den Multi-Touch-Aufbau nicht geeignet. Zum einen blockiert das notwendige weiße Papier die Sicht der Kamera, zum andern würden die Diffusorfolien Berührpunkte und vor allem Marker für die Kamera unkenntlich machen. Deshalb musste die Hintergrundbeleuchtung ersetzt werden. Durch das Wegfallen der Kaltlichtkathoden der Hintergrundbeleuchtung konnte das interne Netzteil des Bildschirms durch ein gewöhnliches PC-Netzteil ersetzt werden.

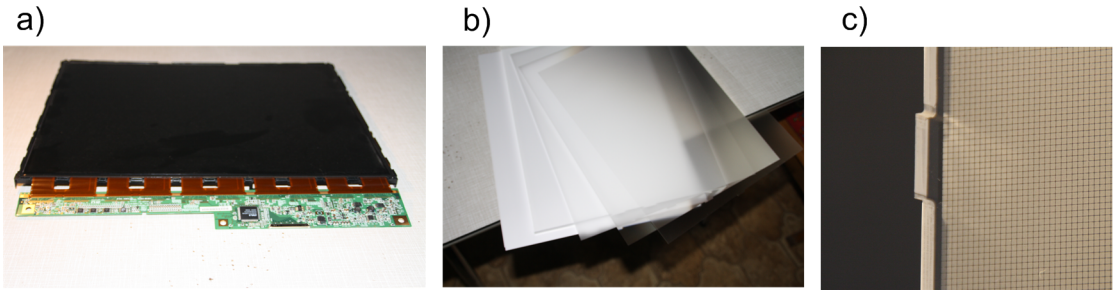


Abbildung 2.2: LC-Panel und Teile der ursprünglichen Hintergrundbeleuchtung.

Acrylglas

Plexiglas® EndLighten ist ein speziell für die Kantenbeleuchtung entwickeltes Acrylglas. Streupartikel im Material reduzieren die Totalreflexion, sodass von den Kanten eintreffendes Licht, im Gegensatz zu Glas oder gewöhnlichem Acrylglas, gleichmäßig auf der gesamten Glasfläche abgestrahlt wird. Detaillierte Informationen können der Broschüre des Herstellers (Evonik 2008) entnommen werden.

In die vordere Platte sind an beiden Seiten je 16 SFH 4350 IR-LEDs[3] (Osram 2007) mit 850 nm Wellenlänge montiert, in die hintere, oben und unten, jeweils 48 weiße[4] LEDs. Sie sind entsprechend Abbildung 2.3 zu je acht infraroten bzw. drei weißen LEDs mit einem passenden Vorwiderstand in Reihe an die 12 V Spannungsversorgung des PC-Netzteils angeschlossen. Beide Platten sind an den Rändern mit selbstklebenden Spiegelfolien ausgestattet um seitliches Austreten des Lichts zu verhindern.

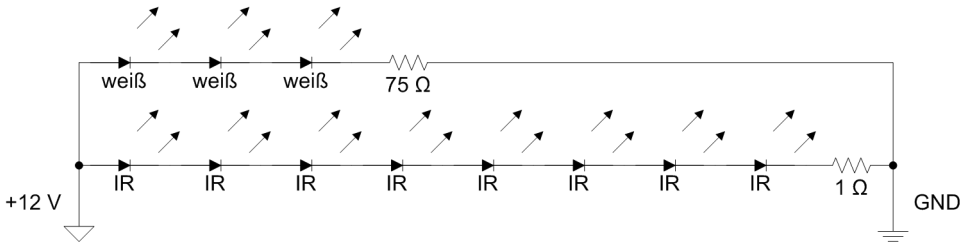


Abbildung 2.3: Schaltplan für weiße LEDs (oben) und infrarote LEDs (unten).

Die Löcher, in welche die LEDs in die Plexiglas® Platten eingesetzt sind, haben durch die Zerspahnung beim Bohren eine raue Oberfläche. Dadurch würde ohne Gegenmaßnahmen das von den LEDs kommende Licht diffus abgestrahlt. Um eine optimale Beleuchtung der gesamten Fläche zu gewährleisten, muss das Licht jedoch in einem möglichst kleinen Winkel durch die Platte strahlen. Dies kann durch Füllen der Löcher mit einem durchsichtigen Mineralöl[5] bewerkstelligt werden.

[3] IR = infrarot.
[4] Farbtemperatur laut Händler ca. 6000 K.
[5] Parfümfreies Babyöl wurde im konkreten Fall verwendet.

Kamera

Die Kamera ist durch vier bewegliche Teleskopstangen mit dem Bildschirm verbunden. Somit kann sie frei bewegt und beliebig auf das Panel ausgerichtet werden. Vor der Kamera befindet sich ein Filter, welcher das Licht in einem schmalen Band um 850 nm, was der Wellenlänge der IR-LEDs entspricht, passieren lässt.

Im Originalzustand lässt sich die verwendete Kamera nicht fest montieren, weshalb speziell für die Montage am Bildschirm ein geeignetes Gehäuse gebaut wurde. Zudem besitzen Kameras in der Regel IR-Filter, die lediglich sichtbares Licht passieren lassen. Im Fall der eingesetzten PlayStation®-Kamera kann dieser Filter, da er sich im Objektiv an letzter Stelle[6] direkt über dem Sensor befindet, entfernt werden. Obwohl der Filter selbst nicht als Linse fungiert, ändert seine Abwesenheit die Brennweite des Objektivs und wurde durch einen rundgeschliffenen Objektträger für Mikroskope ersetzt.

2.1.2 Funktionsweise

Berührungserkennung

Der gebaute Bildschirmprototyp arbeitet ähnlich zu dem bereits erwähnten DSI-Prinzip, welches sich die Eigenschaften von Plexiglas® EndLighten, Licht gleichmäßig von einer Fläche abzustrahlen, zu Nutze macht.

Die vordere Acrylscheibe strahlt infrarotes Licht in beide Richtungen ab. Die Hälfte des Lichts durchdringt das LC-Panel und wird von der auf das Panel ausgerichteten Kamera wahrgenommen. Der restliche Teil wird vom Bildschirm zum Benutzer abgestrahlt. Nähert sich ein Objekt dem Bildschirm wird ein Teil des nach vorn abgestrahlten Lichts am Objekt in Richtung Kamera reflektiert und erscheint auf der Kamera heller als die Umgebung.

Bei einer Berührung mit der Acrylplatte tritt am Berührpunkt zudem keine Totalreflexion auf und zusätzliches Licht wird in Richtung Kamera abgestrahlt. Dadurch werden Berührungen mit der Acrylplatte besonders hell von der Kamera wahrgenommen. In der Produktbeschreibung des Herstellers wird aus diesem Grund empfohlen zur gleichmäßigen Beleuchtung von Motiven einen Abstand zwischen Motiv und Plexiglas® einzuhalten (Evonik 2008).

Im Gegensatz zu der im Buch (NUI Group Authors 2009) beschriebenen Anwendung, in Kombination mit einem Beamer, muss das infrarote Licht das LC-Panel, welches sich hinter der Acrylplatte befindet, durchdringen, bevor es auf das Objektiv der Kamera trifft.

In einem Versuch vor dem Bau des Prototyps wurde untersucht, ob das auf dem Panel dargestellte Bild Einfluss auf die Transmission von Licht im infraroten Wellenlängenbereich hat. Dabei wurde das Panel mit der modifizierten Kamera unter verschiedenen Beleuchtungsbedingungen gefilmt. Abbildung 2.4 zeigt die Versuchsergebnisse. In Bild a) wurde das Panel ausschließlich mit weißem Licht beleuchtet

[6] Auf dem Markt sind verschiedene Varianten der Kamera erhältlich, in denen sich der Filter nicht immer an der gleichen Stelle im Objektiv befindet.

und mit der Kamera ohne Filter aufgenommen. Das Bild zeigt das auf dem Panel dargestellte Bild. Im Bild b) wurde zusätzlich infrarotes Licht zur Beleuchtung eingesetzt. Auch zuvor dunkle Bereiche im Bild sind durch das infrarote Licht gleichmäßig beleuchtet. In c) wurde zusätzlich der IR-Bandpass-Filter vor dem Kameraobjektiv befestigt, damit lediglich infrarotes Licht von der Kamera wahrgenommen wird. Im Bild sind keine Unterschiede in der Helligkeit, die auf den Bildinhalt schließen lassen, erkennbar. Somit konnte vorab sowie bei Tests am laufenden Prototyp nachgewiesen werden, dass das auf dem Panel dargestellte Bild kaum Einfluss auf die Transmissionfähigkeit im infraroten Bereich hat. Daraus folgt, dass der Zustand des Panels für die Berührpunkt- und Markererkennung nicht berücksichtigt werden muss.

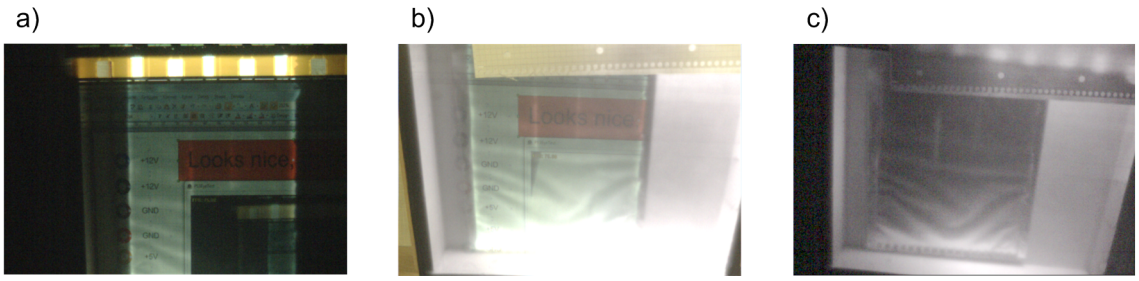


Abbildung 2.4: Aufnahmen des LC-Panels mit der modifizierten Kamera unter verschiedenen Beleuchtungsbedingungen.

Die Blickwinkelabhängigkeit des Panels ist jedoch auch gegenüber infrarotem Licht vorhanden, wie in weiteren Versuchen am Prototyp gemessen werden konnte. Da die von der Kamera stammenden Infrarotbilder die Grundlage der Berührpunkt- und Markererkennung ist, muss die Winkelabhängigkeit bei der Bildverarbeitung und -erkennung berücksichtigt werden.

Hintergrundbeleuchtung

Für die neue Hintergrundbeleuchtung wird ebenfalls das bei der Berührungserkennung beschriebene DSI-Prinzip eingesetzt. Dies garantiert eine gleichmäßige Beleuchtung über das gesamte Panel. Jedoch ist es problematisch eine ausreichende Leuchtkraft zu erzielen, ohne die Sicht der Kamera durch zusätzliche Folien einzuschränken.

Deshalb wurden für die Beleuchtung sehr viel mehr LEDs verbaut als für die Berührungserkennung nötig sind. Die Hälfte des abgestrahlten Lichts entweicht jedoch auch bei der Beleuchtung auf der falschen Seite der Acrylplatte. Mit Hilfe einer weißen, pyramidenförmigen Abdeckung zwischen Acrylglas und Kamera kann das Licht zurückreflektiert werden und somit auch zur Hintergrundbeleuchtung beitragen. Diese Art der Abdeckung kann im ersten Prototyp nicht umgesetzt werden, da die Kameraposition durch die Abdeckung nicht mehr veränderbar wäre.

Sollte die Leuchtkraft trotz Abdeckung nicht ausreichen, besteht die Möglichkeit, an den Seiten des Acrylglases weitere LEDs für die Beleuchtung einzusetzen. Problematisch wird die Beleuchtung jedoch beim Einsatz eines Bildschirms mit größeren

Bilddiagonale, da die Anzahl der LEDs die verbaut werden können proportional zum Umfang des Panels ansteigt. Der Umfang U steigt linear mit der Seitenlänge an, die zu beleuchtende Fläche A jedoch quadratisch, siehe Formeln (2.1) und (2.2).

$$U = 2 \cdot (a + b) \tag{2.1}$$

$$A = a \cdot b \tag{2.2}$$

Zudem hängt das Verhältnis zwischen Umfang und Fläche vom Seitenverhältnis des Panels ab. Wird für das Panel ein konstanter Umfang von $4a$ angenommen, können beliebige Rechtecke mit den Seitenlängen $a - c$ und $a + c$ beschrieben werden.

$$U = 2 \cdot ((a - c) + (a + c)) = 4a \tag{2.3}$$

Werden die Seitenlängen in die Formel für den Flächeninhalt eingesetzt, ergibt sich folgende Gleichung:

$$A = (a - c) \cdot (a + c) = a^2 - c^2 \tag{2.4}$$

Die Fläche wird bei gleichbleibendem Umfang folglich maximal, wenn die Abweichung c der Seitenlängen 0 beträgt. Dies entspricht einem Seitenverhältnis von 1:1. Daraus folgt, dass mit Panels im 16:9 Format größere Bilddiagonalen erzielt werden können, als mit Panels im 4:3 Format. Das Skalierungsproblem bleibt jedoch bestehen.

2.1.3 Erster Prototyp

In den Abbildungen 2.5 und 2.6 ist der im Rahmen der Arbeit entstandene Prototyp abgebildet. In der Schrägansicht ist der IR-Filter der Kamera, die mit Hilfe der Teleskopstangen hinter dem LC-Panel positioniert wurde, deutlich erkennbar. Auf das Gestänge wurde eine Abdeckplatte zur Befestigung der Ansteuerelektronik des LC-Panels gelegt. Die seitlich in das Plexiglas® scheinenden LEDs sind auf Platinen gelötet und werden über das PC-Netzteil mit Spannung versorgt. Durch die Platinen wird zugleich das Kabelaufkommen reduziert und die Befestigung mehrerer LEDs am Stück ermöglicht[7]. Das mehrfach gelochte Halteblech am Standfuß ermöglicht eine Veränderung der Bildschirmneigung.

Die Elektronik zum Ansteuern des Panels und deren Anschluss an das Netzteil sind in Abbildung 2.6 gut zu erkennen. Die ursprünglichen Bedienknöpfe des LC-Panels, über die unter anderem die Helligkeit und der Kontrast eingestellt werden können, sind seitlich an der Abdeckplatte befestigt.

2.2 Bilderkennung und -verarbeitung

Um Marker und Berührpunkte aus den von der Kamera aufgenommenen Bildern extrahieren zu können, werden diverse Algorithmen der Bildverarbeitung eingesetzt.

[7] Ohne Platinen müsste jede LED einzeln an das Plexiglas geklebt werden.

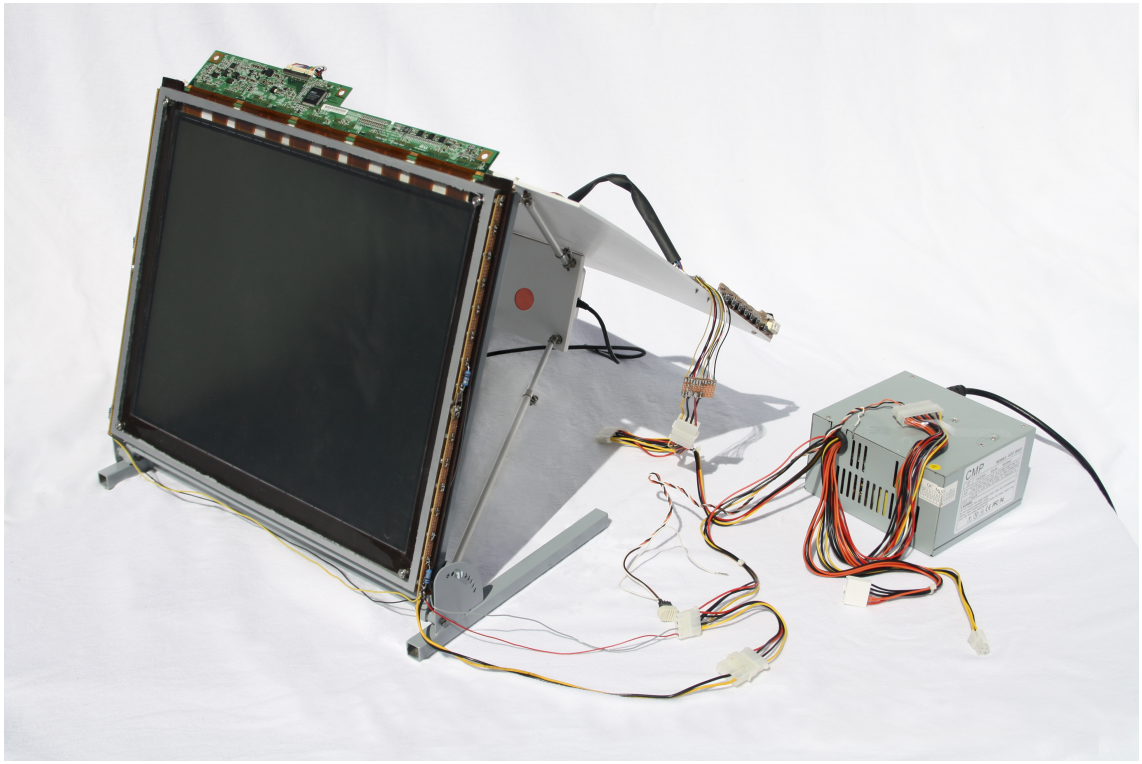


Abbildung 2.5: Schrägansicht des entwickelten Prototyps und Netzteils.

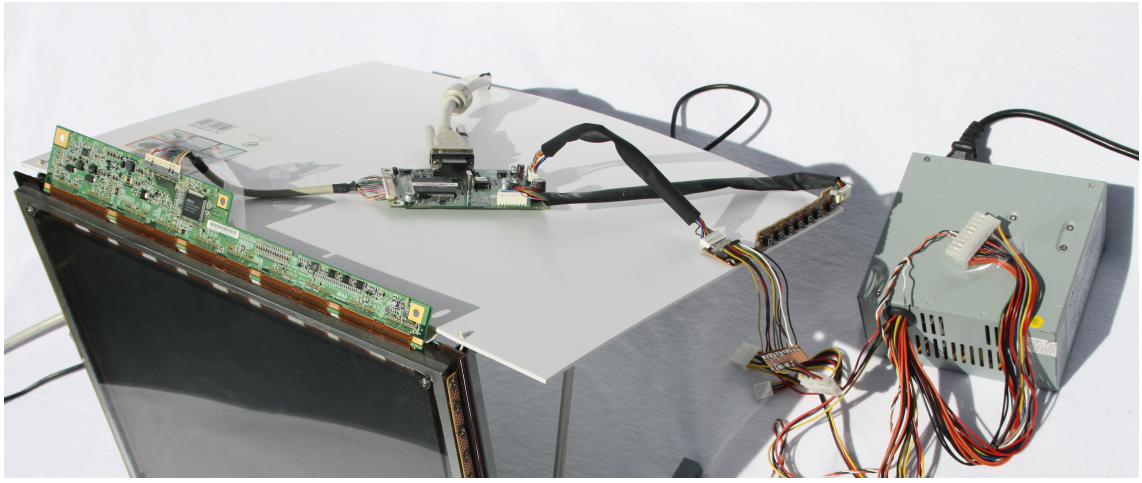


Abbildung 2.6: Elektronik zur Ansteuerung des LC-Panels.

Für ein besseres Verständnis der GPU-Implementierung werden nachfolgend die eingesetzten Algorithmen, deren mathematischen Grundlagen und verwandte Algorithmen erläutert.

2.2.1 Faltung

Die Faltung ist ein aus der Signalverarbeitung stammender mathematische Operator, der angewandt auf zwei Funktionen g und h wiederum eine Funktion liefert. Sie ist für kontinuierliche Funktionen im allgemeinen Fall definiert durch:

$$(g * h)(x) = \int_{-\infty}^{+\infty} h(w)g(x - w)d^N w \tag{2.5}$$

Anschaulich ist die Faltung für eindimensionale Funktionen in jedem Punkt die vorzeichenbehaftete und mit der Funktion h gewichtete Fläche zwischen der Funktion g und der x-Achse über deren gesamten Definitionsbereich. Abbildung 2.7 veranschaulicht dies an einem einfachen Beispiel.

Die Schaubilder a) und b) der Abbildung zeigen die Funktionen g und h. Exemplarisch wird die Faltung an der Stelle $x_1 = -0.8$ durchgeführt (in der Abbildung mit grün gekennzeichnet). In c) sind die Integranten in Abhängigkeit von w an der Stelle x_1 aufgetragen. Die Integration wird nun über dem Produkt beider in c) dargestellten Funktionen durchgeführt. Schaubild d) zeigt den Verlauf des Resultats und die daraus resultierende Fläche. Wie leicht erkennbar ist, ist das Produkt überall dort null, wo die Funktion h null wird. Das letzte Schaubild zeigt die Ergebnisse der Faltung für alle x und dem hervorgehobenen Ergebnis für x_1.

Werden, wie im vorangegangenen Beispiel, für h Funktionen gewählt, die lediglich in einem kleinen Bereich um den Ursprung ungleich null sind, spricht man von einer Faltungsmaske. Durch Anwendung der Faltungsmaske sind die Werte der resultierenden Funktion g' lediglich von den Werten in einer kleinen Umgebung um x der Funktion g abhängig. Die Faltung erzeugt in dieser Umgebung einen durch die Funktion h genau definierten gewichteten Mittelwert der Funktion g, was einer Filterung des Signals entspricht (vgl. Jähne 2005, Seiten 55ff).

Faltung diskreter Funktionen

Bilder werden in der digitalen Bildverarbeitung durch Pixelfelder repräsentiert. Ein beliebiges zweidimensionales Bild besitzt N Spalten und M Zeilen, die jeweils von null beginnend gezählt werden. Im Gegensatz zu kontinuierlichen Signalen sind diese Bildinformationen diskret: Ein Bild kann als Matrix mit der jeweiligen Anzahl Zeilen und Spalten dargestellt werden. Somit muss die Faltung ebenso diskret formuliert werden, um auf Bilddaten arbeiten zu können. Dies erfolgt analog zu (2.5), jedoch mit auf den Definitionsbereich angepassten Integrations- bzw. Summationsgrenzen.

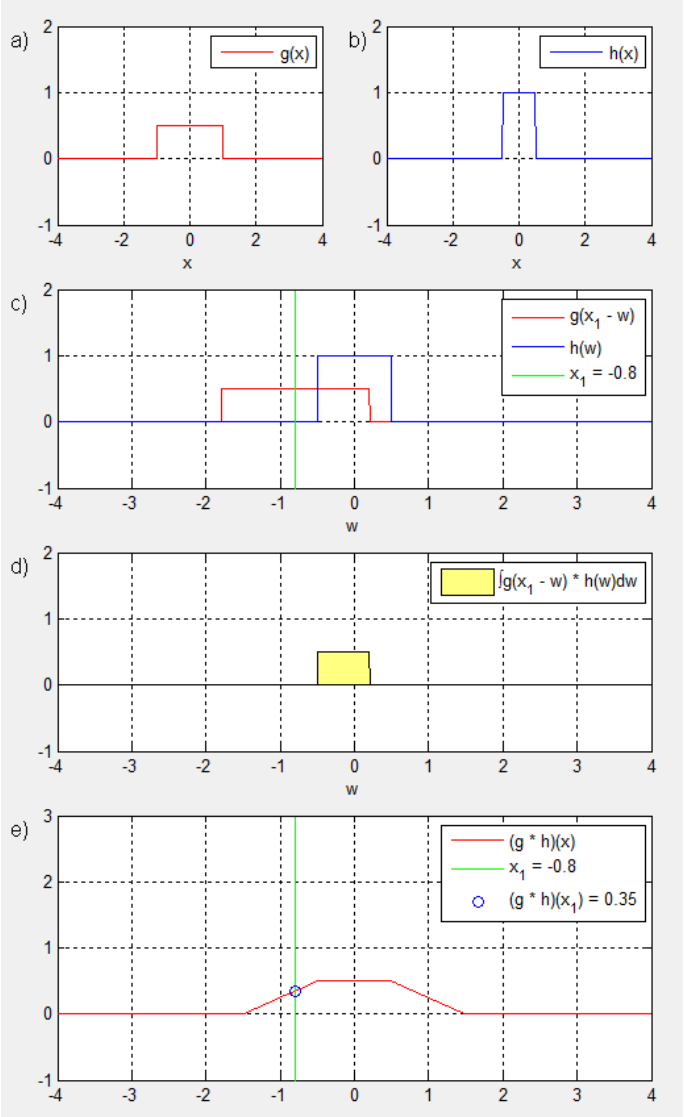


Abbildung 2.7: Darstellung einer einfachen eindimensionalen Faltung.

Die eindimensionale diskrete Faltung lautet somit:

$$g'_n = \sum_{n'=0}^{N-1} h_{n'} g_{n-n'} \tag{2.6}$$

Und die zweidimensionale diskrete Faltung:

$$g'_{m,n} = \sum_{m'=0}^{M-1} \sum_{n'=0}^{N-1} h_{m',n'} g_{m-m',n-n'} \tag{2.7}$$

Wird nun ein ein- bzw. zweidimensionales Bild als diskrete periodische Funktion betrachtet, die sich jeweils nach N, bzw. M Werten in allen Richtungen wiederholt

(vgl. Jähne 2005, Seiten 42ff), können Indizes wie folgt ersetzt werden:

$$g_{N-n} = g_{-n}, \text{bzw.} g_{N-n,M-m} = g_{-n,-m} \tag{2.8}$$

Handelt es sich bei der Funktion h um eine Faltungsmaske, die auf einen Bereich beschränkt ist[8], lässt sich daraus eine Formel für die Faltung, ausgehend von einem zentralen Pixel, herleiten:

$$g'_{m,n} = \sum_{m'=-r}^{r} \sum_{n'=-r}^{r} h_{m',n'} g_{m-m',n-n'} \tag{2.9}$$

Die Größe der Faltungsmaske h beträgt folglich $(2r+1) \times (2r+1)$. Diese Formel besitzt, vergleichbar mit der kontinuierlichen Faltungsformel, den Mittelpunkt ihres Summationsintervals im zentralen Pixel. Dadurch wird die Wirkungsweise eines Filters durch eine kompaktere Schreibweise der Filtermaske klarer.

Die Faltungsmaske lässt sich nun als Filtermaske in Matrixform darstellen. Nachfolgend wird dies am Beispiel eines 3×3 Boxfilters, der in Abschnitt 2.2.2 näher erläutert wird, gezeigt.

Die aus Gleichung (2.7) hervorgehende Funktion h kann im Falle des Boxfilters als $M \times N$ Matrix H wie nachfolgend dargestellt werden:

$$H = \frac{1}{9} \begin{bmatrix} 1 & 1 & 0 & \cdots & 0 & 1 \\ 1 & 1 & 0 & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ 1 & 1 & 0 & \cdots & 0 & 1 \end{bmatrix} \tag{2.10}$$

Auffällig ist, dass die Matrix fast ausschließlich aus Nullen besteht und abweichende Werte lediglich in den Ecken vorkommen. Dies resultiert aus der Tatsache, dass eine Filtermaske in einem Bereich um ihren Ursprung definiert ist und der Definitionsbereich des Bildes bei Null beginnt. Durch die modifizierte Formel in Gleichung (2.9) lässt sich der Filter durch Angabe der Filtermaske H_{Filt} auf ein Minimum reduzieren (Entsprechungen zwischen den Matrizen sind farblich hervorgehoben):

$$H_{Filt} = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.11}$$

Eigenschaften

Die Faltung besitzt drei wichtige Eigenschaften, die für die Erzeugung und Anwendung von Filtern von Bedeutung sind (vgl. Jähne 2005, Seite 56). Die Faltung ist:

- kommutativ $a * b = b * a$

[8] Dies bedeutet, dass alle Werte außerhalb der Faltungsmaske null sind.

- assoziativ $a * (b * c) = (a * b) * c$
- distributiv bezüglich Addition $a * (b + c) = (a * b) + (a * c)$

2.2.2 Glättung

Mit Hilfe der Konvolution können verschiedene Filter realisiert werden. Dazu gehören diverse Glättungsfilter, die nachfolgend erläutert werden. Glättungsfilter werden eingesetzt um hochfrequentes Rauschen aus Bildern entfernen zu können und wirken somit als Tiefpass auf das Bild. Dieses Rauschen tritt beispielsweise bei Kameraaufnahmen durch das zufällige Auftreffen von Photonen auf den Bildsensor und durch die Verstärkung der vom Sensor kommenden Signale in der Elektronik auf.

Boxfilter

Die einfachste Möglichkeit das Rauschen eines Bildes zu verringern ist der bereits erwähnte Boxfilter. Er bildet aus einer rechteckigen Umgebung jedes Pixels den Durchschnitt. Die Filtermaske besteht ausschließlich aus Einsen und wird durch die Anzahl ihrer Elemente geteilt, siehe (2.11). Auf diese Weise wird sichergestellt, dass die Helligkeit in homogenen Gebieten konstant bleibt.

Der Boxfilter besitzt eine weitere Eigenschaft, die bei seiner Anwendung ausgenutzt werden kann. Bei Anwendung einer zweidimensionalen Faltung müssen generell $N \cdot M$ Multiplikationen und $N \cdot M - 1$ Additionen durchgeführt werden. Folglich steigt der Rechenaufwand beim Vergrößern der Maske quadratisch. Durch die Assoziativität der Faltung können Filter zusammengefasst werden. Dies kann in manchen Fällen auch dazu genutzt werden, einen Filter in eine horizontale und eine vertikale Komponente aufzuspalten. Filter, die diese Bedingungen erfüllen, nennt man separierbare Filter. Der Boxfilter gehört zu den separierbaren Filtern und lässt sich sehr einfach aufteilen.

$$H = H_x * H_y \tag{2.12}$$

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} = \frac{1}{3}\begin{bmatrix}1 & 1 & 1\end{bmatrix} * \frac{1}{3}\begin{bmatrix}1\\ 1\\ 1\end{bmatrix} \tag{2.13}$$

Der Gesamtfilter besteht nun aus zwei eindimensionalen Filtern. Dadurch müssen zwar zwei Faltungen durchgeführt werden, die Zahl der benötigten Multiplikationen und Additionen sinkt jedoch auf $N + M$ bzw. $N + M - 2$. Somit steigt der Rechenaufwand lediglich linear mit der Größe der Filtermaske. Im speziellen Fall des Boxfilters können die Multiplikationen durch Ausklammern zudem auf zwei gesenkt werden.

Die Wirkungsweise eines 5×5 Boxfilters ist in Abbildung 2.8 b) anhand eines speziellen Testbildes dargestellt. Das Testbild besteht aus konzentrischen Ringen, die nach außen hin schmäler werden. Das Bild simuliert Bildrauschen mit zu den

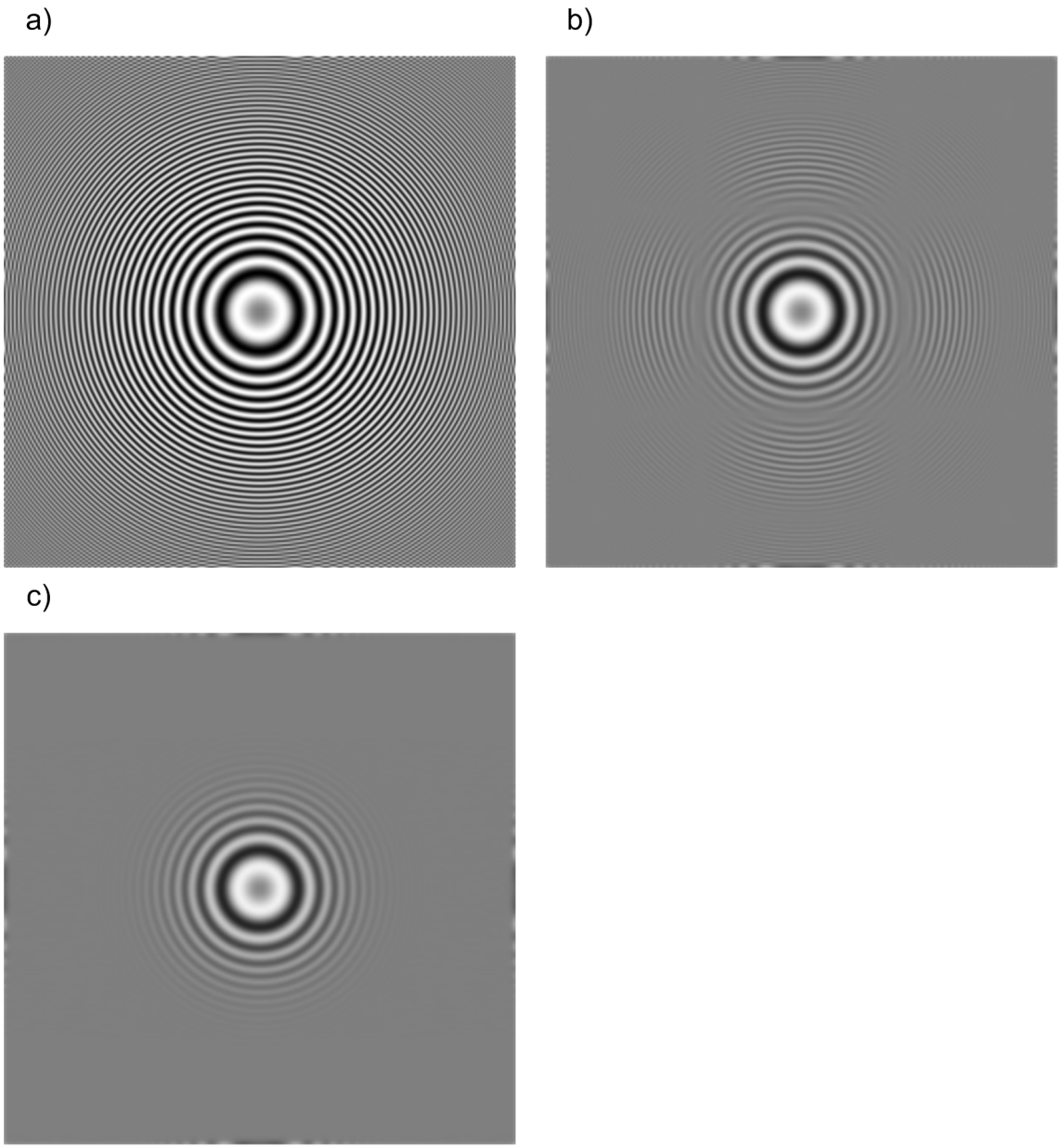


Abbildung 2.8: Glättungsfilter: a) Originalbild, b) 5×5 Boxfilter, c) 5×5 Binomialfilter.

Bildrändern steigenden Frequenzen und, aufgrund der Kreisform, unterschiedlichen Richtungen.

Im direkten Vergleich mit dem Originalbild a) wird sichtbar, dass sich der Boxfilter nicht ideal verhält. Bei genauer Betrachtung werden im gefilterten Bild horizontale und vertikale Linien, an denen das Rauschen stark unterdrückt wird, sichtbar. Der Boxfilter unterdrückt folglich das Rauschen bestimmter Frequenzen effektiv, lässt jedoch davon abweichende Frequenzen zu. Ein idealer Filter würde steigende Frequenzen in zunehmendem Maße unterdrücken.

Binomialfilter

Das Problem des Boxfilters liegt in der Gewichtung der Pixel. Da das arithmetische Mittel aller Pixel gebildet wird, tragen weitentfernte Pixel[9] genauso zum Wert des Pixels bei wie Pixel, die in direkter Nachbarschaft liegen. Wünschenswert ist ein Filter, der Pixel in der Umgebung stärker gewichtet als entfernte Pixel.

Ein Filter mit diesen Eigenschaften lässt sich aus dem Boxfilter ableiten. Dazu betrachtet man zunächst im eindimensionalen Fall den kleinsten Boxfilter mit der Filtermaske:

$$B = \frac{1}{2}\begin{bmatrix}1 & 1\end{bmatrix} \tag{2.14}$$

Dieser bildet den Mittelwert aus lediglich zwei horizontal benachbarten Pixeln. Dies erscheint durch die Tatsache logisch, dass nur zwei Pixel an der Filterung beteiligt sind und somit immer denselben Abstand zueinander haben. Nachteil dieses Filters ist die gerade Anzahl von Elementen in der Filtermaske. Man muss entscheiden, ob das linke oder das rechte Pixel durch den neuen Wert ersetzt wird. Da die korrekte Position für eine Ersetzung genau zwischen den Pixeln liegt, kommt es unweigerlich zu einer Verschiebung des Bildes um ein halbes Pixel.

Abbildung 2.9 a) und b) zeigen diese Verschiebung: Die Filtermaske wurde durch jeweils eine Null ergänzt damit das Zentrum der Faltung aus der Matrix hervorgeht. Aus den Darstellungen wird ersichtlich, dass die Verschiebung der beiden Filter gegensätzlich ist. Nun ist naheliegend, beide Filtermasken nacheinander auf das Bild anzuwenden. Wie erwartet heben sich die Verschiebungen gegenseitig auf, siehe 2.9 c). Mathematisch kann diese Operation wie folgt beschrieben werden:

$$B^2 = B_L * (B_R * A) \tag{2.15}$$

Durch die Assoziativität der Faltung ergibt sich dann:

$$B^2 = (B_L * B_R) * A \tag{2.16}$$

$$= (\frac{1}{2}\begin{bmatrix}0 & 1 & 1\end{bmatrix} * \frac{1}{2}\begin{bmatrix}1 & 1 & 0\end{bmatrix}) * A \tag{2.17}$$

$$= \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} * A \tag{2.18}$$

[9] Pixel, die sich am Rand der Filtermaske befinden.

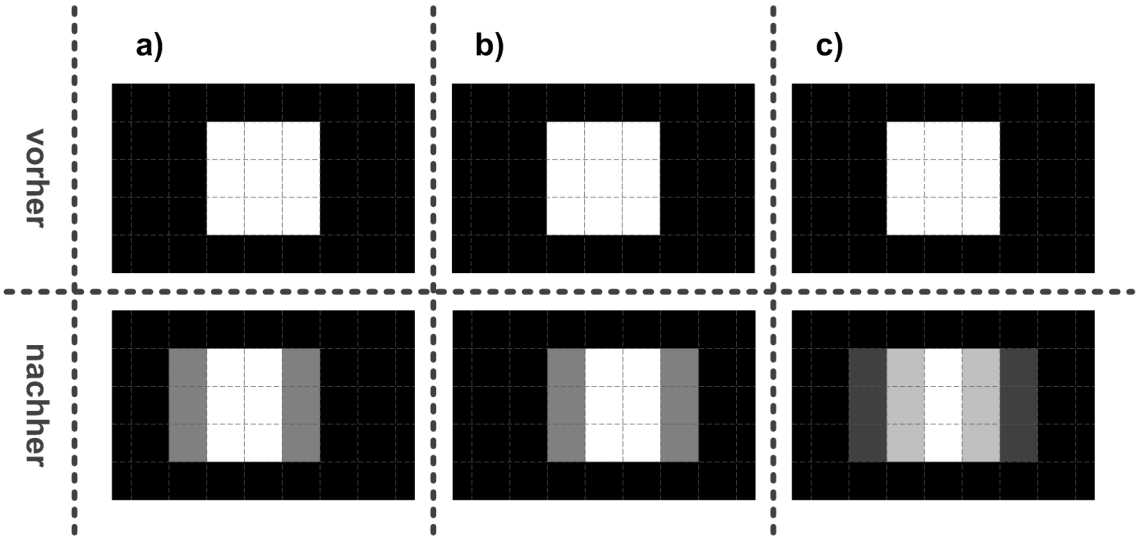


Abbildung 2.9: Anwendung von minimalen Boxfiltern (unten) auf ein Bild (oben). Filtermasken: a) $B_R = \frac{1}{2}\begin{bmatrix}1 & 1 & 0\end{bmatrix}$, b) $B_L = \frac{1}{2}\begin{bmatrix}0 & 1 & 1\end{bmatrix}$, c) beide Filter angewandt.

Beliebig große Filter lassen sich durch sukzessives Filtern mit den elementaren Boxfiltern erreichen, nachfolgend einige auf diese Weise erstellte Filtermasken:

$$\begin{aligned} B^1 &= \frac{1}{2}\begin{bmatrix}1 & 1\end{bmatrix} \\ B^2 &= \frac{1}{4}\begin{bmatrix}1 & 2 & 1\end{bmatrix} \\ B^3 &= \frac{1}{8}\begin{bmatrix}1 & 3 & 3 & 1\end{bmatrix} \\ B^4 &= \frac{1}{16}\begin{bmatrix}1 & 4 & 6 & 4 & 1\end{bmatrix} \\ B^5 &= \frac{1}{32}\begin{bmatrix}1 & 5 & 10 & 10 & 5 & 1\end{bmatrix} \end{aligned} \tag{2.19}$$

Nach kurzer Betrachtung der in Gleichung (2.19) nach Größe geordnet dargestellen Filtermasken wird ersichtlich, dass die Gewichte der einzelnen Filter binomial, Gleichung (2.20), mit Trefferwahrscheinlichkeit $p = 0,5$ verteilt sind. Aus diesem Zusammenhang kann eine allgemeine Gleichung für diese Art Filtermasken hergeleitet werden, Gleichungen (2.21) und (2.22).

$$B(k \mid n, p) = \binom{n}{k} p^k (1-p)^{n-k} \tag{2.20}$$

$$\begin{aligned} B(k \mid n, 0.5) &= \binom{n}{k} 0.5^k (1-0.5)^{n-k} \\ &= \binom{n}{k} 0.5^n = \binom{n}{k} \frac{1}{2^n} = \binom{n}{k} 2^{-n} \end{aligned} \tag{2.21}$$

$$\begin{aligned} B^n &= \begin{bmatrix} B(0 \mid n, 0.5) & B(1 \mid n, 0.5) & \cdots & B(n-1 \mid n, 0.5) & B(n \mid n, 0.5) \end{bmatrix} \\ B^n &= 2^{-n} \begin{bmatrix} \binom{n}{0} & \binom{n}{1} & \cdots & \binom{n}{n-1} & \binom{n}{n} \end{bmatrix} \end{aligned} \tag{2.22}$$

Aufgrund ihrer binomialen Verteilung werden diese Filter Binomialfilter oder auch Gauß-Filter[10] genannt. Zweidimensionale Filter lassen sich durch Kombination eines horizontalen und eines vertikalen Filters leicht erzeugen. Die Rechenkomplexität ist jedoch analog zu den Boxfiltern für mehrdimensionale Filter deutlich höher.

Binomial- oder Gauß-Filter verhalten sich im Gegensatz zu Boxfiltern bezüglich der Dämpfung hoher Frequenzen ideal, siehe hierzu Abbildung 2.8 c). Die konzentrischen Kreise werden ab einem bestimmten Radius stark gedämpft. Die Dämpfung ist im Gegensatz zum Boxfilter unabhängig von der Richtung des Rauschens, das heißt, es sind keine senkrechten oder waagrechten Dämpfungsmaxima erkennbar.

Rekursive Gauß-Filter

Zur Approximation von Gauß- bzw. Binomialfiltern existieren verschiedene rekursive Implementierungen. Diese haben den Vorteil, dass deren Rechenzeit konstant und somit unabhängig von der Standardabweichung des Filters ist. Da in rekursiven Implementierungen keine Filtermasken eingesetzt werden, wird die Größe des Filters durch die Standardabweichung der jeweiligen Gauß'schen Glockenkurve angegeben. Untersuchungen verschiedener Implementierungen (Hale 2006) haben gezeigt, dass diese erst bei Standardabweichungen über $\sigma = 3$ effizienter als entsprechende Faltungsmasken sind.

Die Varianz der Binomialverteilung ist $\sigma^2 = \frac{n}{4}$, die Standardabweichung folglich $\sigma = \frac{\sqrt{n}}{2}$. Rekursive Filter sind somit erst bei Filtermasken mit $n \geq 36$ effizienter. Da derartig große Filter zur Unterdrückung des Bildrauschens der Kamera nicht notwendig sind, wurden rekursive Filterimplementierungen im Rahmen dieser Arbeit nicht weiter berücksichtigt.

2.2.3 Kantendetektion

Eine weitere Anwendung der Faltung findet sich in der Kantendetektion. Kanten zeichnen sich in Graustufen-Bilder, die im Folgenden behandelt werden, durch rasche Übergänge von Hell nach Dunkel oder umgekehrt aus. Betrachtet man ein Bild als diskrete Funktion, die Pixelkoordinaten auf Helligkeitswerte abbildet, ist an Kanten die Steigung besonders groß. Dadurch bietet es sich an, den Funktionsverlauf durch Ableitung zu analysieren um Kanten zu finden.

Pixelkoordinaten sind in der Regel zweidimensional, eine Kante kann folglich als Vektor aufgefasst werden. Dieser Vektor, der im Betrag die Steilheit der Kante angibt und der in Richtung der größten Steigung zeigt, nennt man im allgemeinen Fall Gradient ∇ der Funktion. Man erhält ihn durch partielles Ableiten einer Funktion

[10] Die diskrete Variante der Gauß- bzw. Normalverteilung ist die Binomialverteilung.

nach ihren Argumenten, siehe Gleichung (2.23).

$$\nabla f(x_1, \ldots, x_n) = \begin{pmatrix} \frac{\delta f}{\delta x_1} \\ \vdots \\ \frac{\delta f}{\delta x_n} \end{pmatrix} \tag{2.23}$$

Die partielle Ableitung einer Funktion ist definiert durch:

$$\frac{\delta f(x_1, \ldots, x_n)}{\delta x_i} = f_{x_i}((x_1, \ldots, x_n) = \lim_{h \to 0} \frac{f(\ldots, x_i + h, \ldots) - f(\ldots, x_i, \ldots)}{h} \tag{2.24}$$

Für den diskreten Fall ergibt sich für die partielle Ableitung nach x_i:

$$\frac{\delta f(x_1, \ldots, x_n)}{\delta x_i} \approx \frac{f(\ldots, x_i + \Delta x, \ldots) - f(\ldots, x_i, \ldots)}{\Delta x} \tag{2.25}$$

Im einfachsten Fall setzt man $\Delta x = 1$ bzw. $\Delta y = 1$. Dadurch ist die partielle Ableitung die Differenz benachbarter Pixel in horizontaler bzw. vertikaler Richtung. Die entsprechenden Filtermasken sind:

$$D_x = \begin{bmatrix} 1 & -1 \end{bmatrix} \qquad D_y = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{2.26}$$

Abbildung 2.10 a) und b) zeigt ein Beispielbild vor und nach der Ableitung. Das Beispielbild ist im oberen Schaubild von a) dargestellt, die Achsennummerierung bezeichnet die einzelnen Pixel[11]. Das untere Schaubild zeigt den Funktionsverlauf für die Pixelreihe mit $y = 2$, also der mittleren Pixelreihe. Analog dazu stellt das untere Schaubild von b) die Ableitung der in a) abgebildeten Funktion dar. Zu beachten ist, dass die Funktion aufgrund der negativen Steigung negativ wird. Um auch negative Werte darstellen zu können, wurde das entsprechende Bild mit grün für positive Werte und rot für negative Werte kodiert.

Wie erwartet sind die Kanten durch Maxima bzw. Minima in der Ableitung erkennbar. Es zeigt sich jedoch derselbe Nachteil wie schon bei den minimalen Boxfiltern: Erkannte Kanten sind um ein halbes Pixel verschoben.

Laplace-Filter

Erneutes Anwenden des Filters behebt hier zwar, analog zu Gleichung (2.16), die Verschiebung, führt jedoch zu einer weiteren partiellen Ableitung des Bildes, siehe Abbildung 2.10 c). Der daraus resultierende Filter ist ein Maß für die Krümmung der Funktion:

$$D_x^2 = D_{Rx} * D_{Lx} = \begin{bmatrix} 1 & -1 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} \tag{2.27}$$

$$D_y^2 = \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} \tag{2.28}$$

[11] Die y-Achse ist aus Darstellungsgründen gedehnt, dies hat jedoch keinen Einfluss auf das Ergebnis.

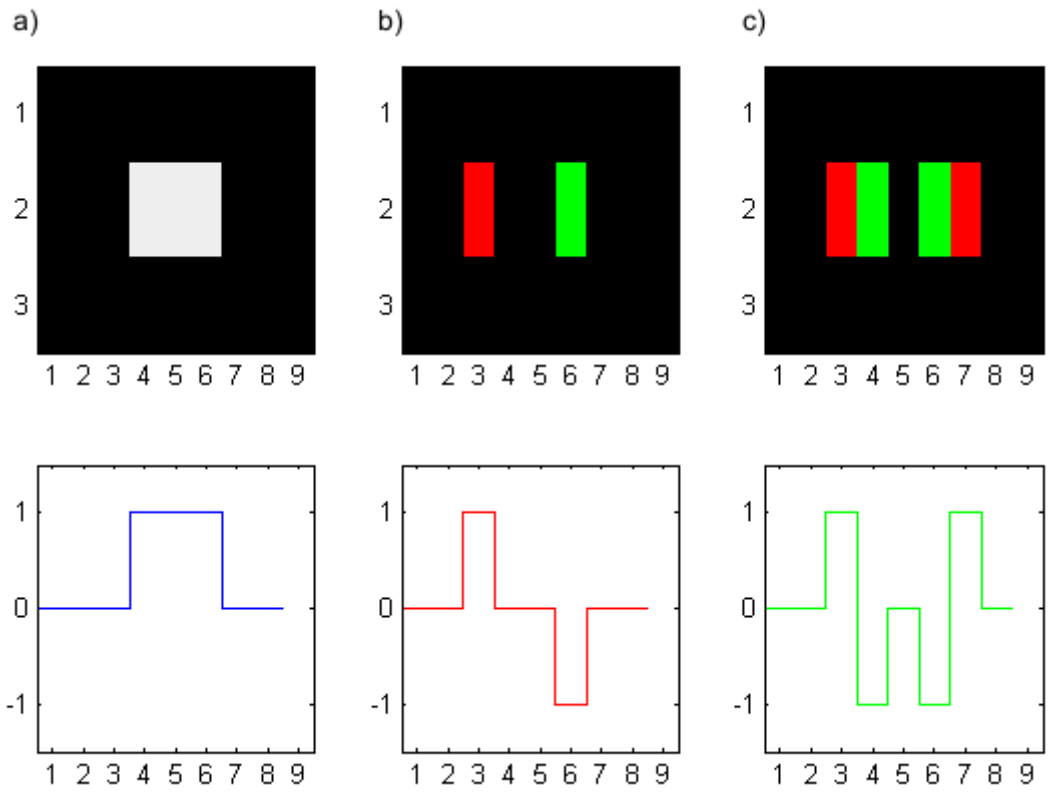


Abbildung 2.10: Einfache Ableitungsfilter in horizontaler Richtung.

Aus Abbildung 2.10 c) wird ersichtlich, dass auch die zweite partielle Ableitung zur Lagebestimmung von Kanten genutzt werden kann, jedoch muss dabei nach Nulldurchgängen gesucht werden. Nachteilig ist, dass die zweite Ableitung auch in homogenen Flächen und Bereichen mit gleichmäßigen Farbverläufen null ist, und somit die Erkennung von Nulldurchgängen erschwert wird. Zudem kann auch Bildrauschen zu Nulldurchgängen führen, was diesen Filter besonders rauschanfällig macht.

Bildet man die Summe der in den Gleichungen (2.27) und (2.28) dargestellten zweiten Ableitungen, erhält man den zweidimensionale Laplace-Filter:

$$\begin{aligned} L = D_x^2 + D_y^2 &= \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} + \begin{bmatrix} 1 \\ -2 \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{aligned} \tag{2.29}$$

Dieser Filter erkennt zwar Kanten aller Richtungen, ist jedoch, analog zu den eindimensionalen Filtern, sehr rauschanfällig.

Sobel-Operator

Um die durch Anwendung des Filters enstehende Verschiebung ohne erneute partielle Ableitung zu beheben, kann der Ableitungsfilter gestreckt werden, indem, zur Berechnung der Ableitung an einem Pixel die Differenz von Vorgänger- und Nachfolgerpixel gebildet wird[12]. Dadurch vergrößert sich Δx auf zwei, somit lauten die Filtermasken:

$$D_{2x} = \frac{1}{2}\begin{bmatrix}-1 & 0 & 1\end{bmatrix} \qquad D_{2y} = \frac{1}{2}\begin{bmatrix}-1\\0\\1\end{bmatrix} \tag{2.30}$$

Der Betrag $|D|$ und die Richtung θ des in Gleichung (2.23) definierten Gradientenvektors, im Folgenden mit D und den Elementen D_x und D_y bezeichnet, sind definiert durch:

$$|D| = \sqrt{D_x^2 + D_y^2} \tag{2.31}$$

$$\theta = \begin{cases} \arctan\left(\dfrac{D_y}{D_x}\right) & \mid \quad D_x \neq 0 \\ \dfrac{\pi}{2} & \mid \quad D_x = 0, D_y > 0 \\ -\dfrac{\pi}{2} & \mid \quad D_x = 0, D_y \leq 0 \end{cases} \tag{2.32}$$

Um die Kantenrichtung bestimmen zu können, müssen die Gradienten möglichst genau berechnet werden. Nachteilig ist, dass auch die erste Ableitung durch die kleine Filtermaske rauschanfällig ist. Eine Möglichkeit das Bildrauschen zu unterdrücken, ohne die Kante in Ableitungsrichtung zu beeinflussen, bietet die Mittelung senkrecht zur Ableitungsrichtung. Wird sie mit dem Binomialfilter B^2 berechnet, erhält man den Sobel-Operator:

$$S_x = D_{2x} * B_y^2 = \frac{1}{2}\begin{bmatrix}1 & 0 & -1\end{bmatrix} * \frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix} = \frac{1}{8}\begin{bmatrix}1 & 0 & -1\\2 & 0 & -2\\1 & 0 & -1\end{bmatrix} \tag{2.33}$$

$$S_y = D_{2y} * B_x^2 = \frac{1}{8}\begin{bmatrix}1 & 2 & 1\\0 & 0 & 0\\-1 & -2 & -1\end{bmatrix} \tag{2.34}$$

Abbildung 2.11 zeigt den Betrag der durch den Sobel-Operator gefundenen Kanten. Das Originalbild ist in a) dargestellt. b) und c) zeigen die Filterung in horizontaler, bzw. vertikaler Richtung. In d) ist der Betrag beider Filterungen dargestellt. Aus dem Bild ist ersichtlich, dass der Sobel-Operator Kanten in allen Richtungen erkennt.

[12]Die Streckung entspricht der Anwendung eines minimalen Boxfilters in Ableitungsrichtung.

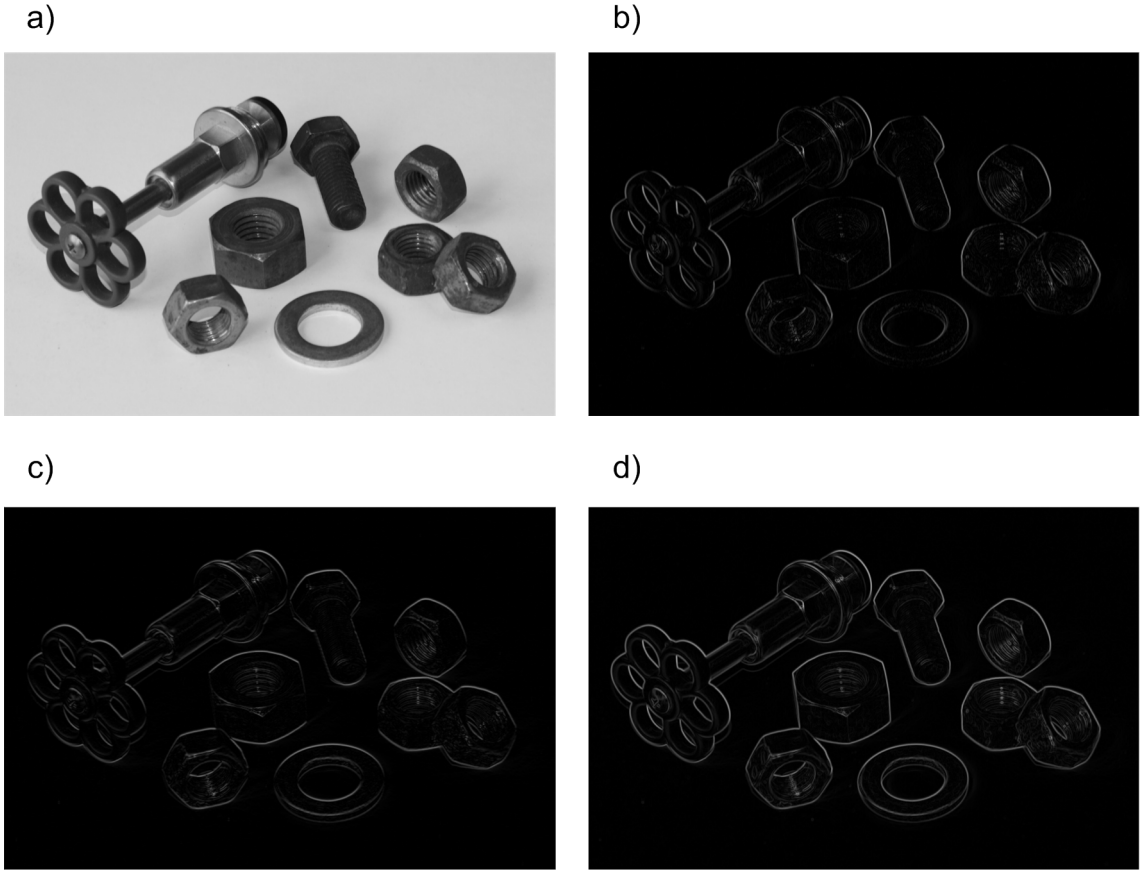


Abbildung 2.11: Sobel-Operator: a) Original, b) $|S_x|$, c) $|S_y|$, d) $|S|$.

Zwar mindert die Mittelung das Rauschen des Signals, jedoch verursacht sie mit den obigen Filtermasken einen systematischen Winkelfehler. Untersuchungen von Scharr (Scharr 2000) haben ergeben, dass dieser Winkelfehler, auf Kosten der Genauigkeit des Betrags, durch nichtlineare Optimierung der Filterkoeffizienten minimiert werden kann. Die resultierenden optimalen Operatoren (siehe Scharr 2000, Gleichung 9.14) bezüglich des Winkelfehlers sind:

$$S_{x_{opt}} = \frac{1}{32} \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix} \tag{2.35}$$

$$S_{y_{opt}} = \frac{1}{32} \begin{bmatrix} 3 & 10 & 3 \\ 0 & 0 & 0 \\ -3 & -10 & -3 \end{bmatrix} \tag{2.36}$$

2.3 Marker

Der Multi-Touch-Bildschirm soll in der Lage sein, sowohl einfache Berührpunkte als auch Marker zu erkennen. Marker erweitern die Anwendungsmöglichkeiten des

Bildschirms, indem neben der Berührung Informationen zu den jeweiligen Berührpunkten übertragen werden können. Im Alltag trifft man auf eine Vielzahl verschiedener Arten, von denen manche für die menschliche und andere für die maschinelle Erkennung ausgelegt sind.

Ziel der Marker ist, durch ihr genau spezifiziertes Aussehen, die Erkennung von Objekten zu vereinfachen und zu beschleunigen. Nachfolgend werden grundlegende Eigenschaften am Beispiel von für den Menschen gedachten Markern erläutert und anschließend verschiedene maschinelle Markertypen vorgestellt.

2.3.1 Marker für Menschen

Eine weltweit verbreitete Art von Markern, die von Menschen interpretiert werden, sind Verkehrsschilder. In der Bundesrepublik Deutschland ist deren Aussehen und Bedeutung in den Paragraphen 39 - 41 der Straßenverkehrs-Ordnung (StVO 2009) und deren Anlagen geregelt. Durch farbliche Kennzeichnung und durch die Form des Schildes ist dessen Art ohne näheres Betrachten erkennbar. Gefahrenzeichen sind beispielsweise dreieckig, zeigen mit der Spitze nach oben und besitzen einen roten Rand.

Aufgrund ihrer Form und Farbgebung heben sich Verkehrsschilder deutlich von ihrer Umgebung ab. Somit kann aus weiter Entfernung innerhalb kürzester Zeit erkannt werden um welche Art Schild es sich handelt. Anschließend muss lediglich die genaue Bedeutung des Schilds anhand des abgebildeten Symbols oder der abgebildeten Zahl erkannt werden. Wären Straßenschilder rein textuell beschrieben, könnte deren Bedeutung nicht in ausreichender Geschwindigkeit und aus dem nötigen Abstand erschlossen werden.

Gute Marker, wie die im Straßenverkehr eingesetzten Verkehrsschilder, zeichnen sich maßgeblich durch die folgenden Faktoren aus:

- Eindeutigkeit: In der StVO ist die Bedeutung der im Straßenverkehr vorkommenden Schilder eindeutig beschrieben.
- Erkennbarkeit: Straßenschilder heben sich stark von ihrer Umgebung ab und sind deshalb gut erkennbar.
- Codierung: Informationen werden im Schild durch Farbgebung, Form und Symbole codiert.

2.3.2 Maschinelle Marker

Marker, die von Maschinen erkannt werden sollen, zeichnen sich ebenfalls durch die oben genannten Eigenschaften aus. Im Gegensatz zu den für Menschen geeigneten Markern sind diese jedoch auf die Eigenschaften des maschinellen Sehens ausgerichtet. Menschen sind in der Regel nicht in der Lage, die Bedeutung eines derartigen Markers mit geringem Zeitaufwand zu interpretieren.

Im vorangegangenen Abschnitt wurden einige Bilderkennungsmechanismen beschrieben, die großen Einfluss auf das Design von Markern haben. Während die Erkennung von komplexen Formen für derzeitige Computer nur bedingt in Echtzeit realisierbar ist (Bayazit u. a. 2009; Heymann 2005), können Kanten beispielsweise sehr schnell erkannt werden. Aus diesem Grund beruhen viele der nachfolgend besprochenen Verfahren in irgendeiner Weise auf der Erkennung von großen Helligkeitsunterschieden.

2.3.3 Strichcodes

Eine der verbreitetsten Markerarten ist der eindimensionale Strichcode, mit dessen Hilfe beispielsweise Produkte im Handel oder auch Bücher identifiziert werden. Eine für jedes Produkt eindeutige Identifikationsnummer wird durch unterschiedlich dicke schwarze Linien auf weißem Hintergrund auf dem Produkt abgedruckt. Spezielle Laser-Strichcodescanner sind dann in der Lage, die Nummer des aufgedruckten Codes auszulesen. Vergleichbar mit Verkehrsschildern werden Nummernbereiche für einzelne Hersteller zentral vergeben. Ein typischer Strichcode ist in Abbildung 2.12 dargestellt.

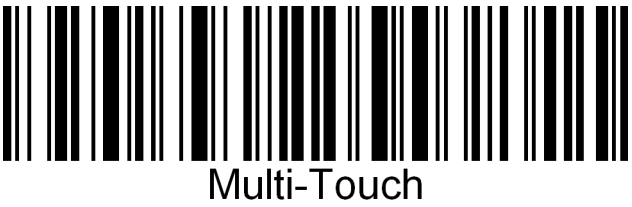


Abbildung 2.12: Code-128 codierter Strichcode des Worts Multi-Touch. Barcode generiert mit TEC-IT Barcode Software, `http://www.tec-it.com/`.

Neben eindimensionalen sind auch mehrdimensionale Strichcodes möglich. Zweidimensionale Codes wie zum Beispiel der Aztec-Code (Longacre u. Hussey 1997; Merki 2003), den die Deutsche Bahn in einer abgewandelten Form für Online-Tickets einsetzt, sind ebenfalls weit verbreitet. Meist sind diese Codes jedoch keine klassischen Strichcodes, da die Informationen nicht in unterschiedlich dicke Balken, sondern in Binärmatrizen codiert werden. Zum Auslesen dieser Marker sind daher im Gegensatz zu eindimensionalen Strichcodes Kameras notwendig, die den gesamten Marker auf einmal erfassen können.

Der Aufbau mehrdimensionaler Strichcodes unterteilt sich in der Regel in zwei Bereiche. Ein Teil des Strichcodes ist für die Erkennung der Lage und Größe erforderlich. Im Fall des Aztec-Strichcodes gehören dazu die an einen Aztekentempel erinnernden verschachtelten Quadrate in der Mitte, siehe Abbildung 2.13. Dieser Teil des Codes ist unabhängig von der gespeicherten Information immer identisch. Der zweite Teil beinhaltet die eigentlichen Daten, die in den restlichen Bereichen untergebracht und binär in das Bild codiert sind. Um Fehlerkennungen zu verhindern werden die Daten zusätzlich redundant im Strichcode hinterlegt. Weiterführende In-

formationen zu diesen und weiteren Strichcodes finden sich in (Bender u. Wagner 2007; Kato u. a. 2010).

Abbildung 2.13: Aztec-Strichcode.

2.3.4 AR-Marker

Auch im Bereich der Augmented Reality wurden verschiedene Systeme zur Erkennung von zweidimensionalen Markern entwickelt, dazu gehören unter anderen ARTag, ARToolKit und ARToolKitPlus (Kato u. Billinghurst 1999; Hirzer 2008; Wagner 2007). Primäres Ziel der Augmented Reality ist, die Lage von Markern im dreidimensionalen Raum zu erkennen, anstatt möglichst viele Informationen in ihnen zu hinterlegen.

Bei den AR-Markern hat sich bezüglich der Markerform ein gewisser Standard entwickelt. Sie sind meist quadratisch und besitzen einen relativ breiten schwarzen Rahmen. Dadurch heben sie sich deutlich von den meisten anderen sich im Raum befindenden Objekten ab. Zur Markererkennung wird nach schwarzen Vierecken, beispielsweise durch das Zusammenfassen von Kantenlinien (Hirzer 2008), im Bild gesucht.

Wird ein Marker erkannt, muss dessen Lage im Raum ermittelt werden. Das Problem ist unter dem Namen "Camera Pose Estimation" bekannt und ist für die Computer Vision fundamental. Aus diesem Grund existiert eine Vielzahl von analytischen und Optimierungsverfahren (Didier u. a. 2008), auf die im Rahmen dieser Thesis jedoch nicht weiter eingegangen werden kann.

Ist die Lage bekannt, kann der Inhalt des Markers analysiert und Informationen extrahiert werden. Dazu wird in einem ersten Schritt der Marker auf eine zweidimensionale Ebene projiziert. Dabei wird bei entsprechender Lage des Markers ein sehr kleiner Bereich im ursprünglichen Bild auf eine große Fläche abgebildet. Da die Auflösung von Kameras beschränkt ist, kann dies dazu führen, dass der Inhalt des Markers nicht korrekt ausgelesen werden kann.

In der Art und Weise, wie Daten im Marker hinterlegt sind, unterscheiden sich die verschiedenen Systeme voneinander. In ARToolKit können beliebige Schwarzweißbilder als Markerinhalt genutzt werden, siehe Abbildung 2.14 a). Diese müssen jedoch im Voraus im System hinterlegt werden. Dieser ästhetische Vorteil schlägt sich jedoch beim Vorhandensein vieler verschiedener Marker negativ auf die Geschwindigkeit aus. Alle Marker müssen mit den gefundenen verglichen werden. Zudem steigt die Chance, dass sich Marker sehr ähnlich sind und deshalb falsch erkannt werden.

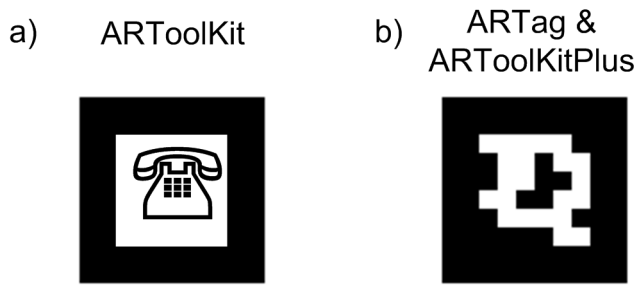


Abbildung 2.14: Unterschiedliche AR-Marker.

Einen anderen Weg gehen ARTag und ARToolKitPlus, eine Weiterentwicklung von ARToolKit, mit Unterstützung für mobile Geräte. Die Informationen werden, wie auch bei zweidimensionalen Strichcodes als Binärmatrix hinterlegt, siehe Abbildung 2.14 b). Ein Vergleich verschiedener Marker entfällt, da eine eindeutige ID direkt aus dem erkannten Marker errechnet werden kann.

2.3.5 Multi-Touch-Marker

Auch in optischen Multi-Touch-Bildschirmen können Marker eingesetzt werden, um Informationen an Berührpunkte zu knüpfen. Auf dem Bildschirm können dann direkt neben einem Marker kontextbezogene Informationen angezeigt werden.

Grundsätzlich lassen sich die bestehenden AR-Algorithmen auch für die Markererkennung in Multi-Touch-Geräten einsetzen. Die Beschränkung auf die Bildschirmoberfläche lässt jedoch Vereinfachungen zu, die zur Beschleunigung der Verfahren genutzt werden können.

Marker müssen lediglich von gewöhnlichen Berührpunkten unterschieden werden, dadurch kann die Erkennung vereinfacht werden. Zudem beschränkt sich die Lage des Markers auf eine zweidimensionale Ebene, weshalb keine komplexen Algorithmen zur Lagebestimmung eingesetzt werden müssen. Da die Marker direkt auf der Bildschirmoberfläche aufliegen, werden sie in der Regel nicht von anderen Objekten verdeckt. Aus diesem Grund muss eine partielle Verdeckung bei der Erkennung nicht berücksichtigt werden.

Durch die geänderten Rahmenbedingungen werden gänzlich andere Markerarten möglich. Als Beispiel dient hier das reacTIVision System (Bencina u. a. 2005). Dessen Marker unterscheiden sich, wie in Abbildung 2.15 a) ersichtlich, in ihrem Aussehen

massiv von den zuvor vorgestellten AR-Markern. Sie basieren auf dem in (Costanza u. Robinson 2003) vorgestellten d-touch System.

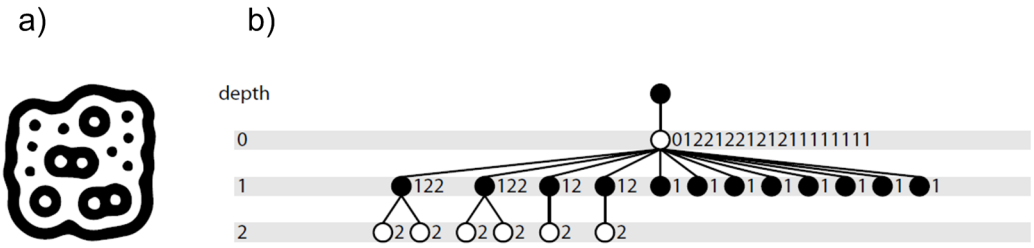


Abbildung 2.15: Marker des reacTIVision Systems mit zugehörigem Nachbarschaftsgraph, entnommen aus (Bencina u. a. 2005).

Beide Verfahren erkennen Marker anhand von Nachbarschaftsgraphen[13]. Hierzu wird das Bild segmentiert, das heißt zusammenhängende Bereiche mit sehr ähnlicher Helligkeit werden zusammengefasst. Wie in Abbildung 2.15 a) erkennbar, besteht ein Marker aus mehreren sich umschließenden Regionen. Die Segmente werden anschließend in eine Baumstruktur übernommen, dabei sind die umschlossenen Segmente jeweils Kinder ihrer umschließenden Segmente. In Abbildung 2.15 b) ist der Graph des in a) dargestellten Markers abgebildet. Werden die Knoten anhand ihres Gewichts[14] geordnet, wird der Aufbau des Baumes unabhängig von der Reihenfolge seiner Erstellung. Im System sind nun verschiedene Marker, also bestimmte Baumstrukturen, hinterlegt mit denen die jeweis extrahierten Bäume verglichen werden.

Auch bei dieser Art Marker können Position und Richtung festgelegt werden, dabei beschränkt sich die Richtungserkennung auf eine zweidimensionale Ebene. Im Multi-Touch-Bereich ist dies jedoch ausreichend. Die Position eines Markers berechnet sich mit Hilfe des Mittelpunkts aller Blätter des Baumes, also aller Knoten, die keine Kinder besitzen. Zur Richtungserkennung wird der Mittelpunkt aller schwarzen Blätter berechnet. Der Vektor zwischen der Markerposition und dem Mittelpunkt der schwarzen Blätter gibt die Richtung des Markers an.

Vorteil rein topologisch basierter Marker ist deren Unabhängigkeit von der eigentlichen Markerform. Dadurch können für menschliche Betrachter schönere und der Ergonomie des zugehörigen Objekts angepasste Marker erzeugt werden. Die einzige Beschränkung an die Form ergibt sich aus der Richtungserkennung, die nicht möglich ist, wenn die Mittelpunkte auf derselben Position liegen. Nachteil der topologischen Marker ist ein deutlich größerer Aufwand bei deren Markergenerierung, wenn die benötigte Fläche minimiert und der Marker dennoch robust erkennbar bleiben soll.

[13]Im Original: Region Adjacency Graph.
[14]Das Gewicht eines Knotens ist die Anzahl aller untergeordneter Knoten.

2.4 CPU-Architektur vs. GPU-Architektur

Ziel dieser Arbeit ist die GPU-basierte Erkennung von Berührpunkten und Markern in Multi-Touch-Anwendungen. Da sich die Architektur modernen Grafikkarten sehr stark von CPU-Architektur unterscheidet und dies gravierende Auswirkungen auf deren Programmierung hat, werden im Folgenden die wichtigsten Unterschiede beider beschrieben. Dazu gehören sowohl die Art der Parallelisierung als auch des Speicherzugriffs und dessen Allokation.

Dabei beschränkt sich der Vergleich auf moderne x86 Multi-Core Prozessoren (wie beispielsweise Casazza 2009), da diese im Verbrauchersektor die größte Verbreitung finden und andere CPU-Architekturen ähnliche Architekturen besitzen. Auf Seite der Grafikkarten wird im Wesentlichen auf die G80-Architektur von NVIDIA (NVIDIA 2008), die zur Implementierung im Rahmen dieser Arbeit eingesetzt wurde, näher eingegangen. Im Folgenden werden die wesentlichen Unterschiede, die Auswirkungen auf die Implementierung der GPU-Algorithmen haben, erläutert.

2.4.1 Parallelisierung

Im Gegensatz zu aktuellen CPUs besitzen Grafikprozessoren eine hochgradig parallelisierte Architektur. Statt vier oder sechs Rechenkernen sind in Grafikkarten mehrere dutzend oder bis zu hunderte im Einsatz[15], siehe hierzu Abbildung 2.16, grüne Rechtecke. Da GPUs eigentlich zum Echtzeit-Rendern von 3D-Szenen gedacht sind, ist deren Architektur auf Streaming ausgelegt, das heißt, auf eine große Datenmenge werden immer dieselben, voneinander unabhängigen, Berechnungen parallel durchgeführt. Deshalb bestehen GPUs aus SIMD-Einheiten[16], in denen mehrere Kerne zusammengefasst sind (in der Abbildung durch acht große Blöcke gekennzeichnet). SIMD bedeutet, dass alle Kerne innerhalb einer Einheit immer dieselbe Instruktion ausführen müssen, dabei können lediglich die Daten, auf denen diese Instruktion ausgeführt wird, variieren.

Daraus resultiert, dass dynamisches Verzweigen (Dynamic Branching) auf Grafikkarten sehr teuer ist, wenn Rechenkerne innerhalb einer SIMD-Einheit unterschiedliche Pfade nehmen. In diesem Fall müssen alle anderen Recheneinheiten jeden eingeschlagenen Pfad sequenziell evaluieren. Bestehen unterschiedliche Pfade aus Zugriffen auf den globalen Speicher, siehe 2.4.2, verlängert sich die Laufzeit aufgrund dessen Latenz erheblich. Somit benötigt eine SIMD-Einheit immer so lange für eine Berechnung, wie deren langsamster Pfad.

In der Architektur von ATI findet Parallelisierung selbst auf Instruktionsebene statt. Die Grafikarchitektur besteht dabei aus Vektoreinheiten, die bis zu fünf skalare Rechenoperationen in einem einzigen Taktzyklus durchführen können, (vgl. ATI 2009), da diese beim Rendern von 3D-Grafiken sehr häufig benötigt werden. Der Geschwindigkeitsgewinn bei Vektoroperationen bringt jedoch auch Nachteile

[15] Während der G80-Grafikchip 96 Rechenkerne besitzt (NVIDIA 2008), besteht die aktuelle Fermi Architektur aus 512 Rechenkernen (NVIDIA 2010).

[16] SIMD = Single Instruction Multiple Data.

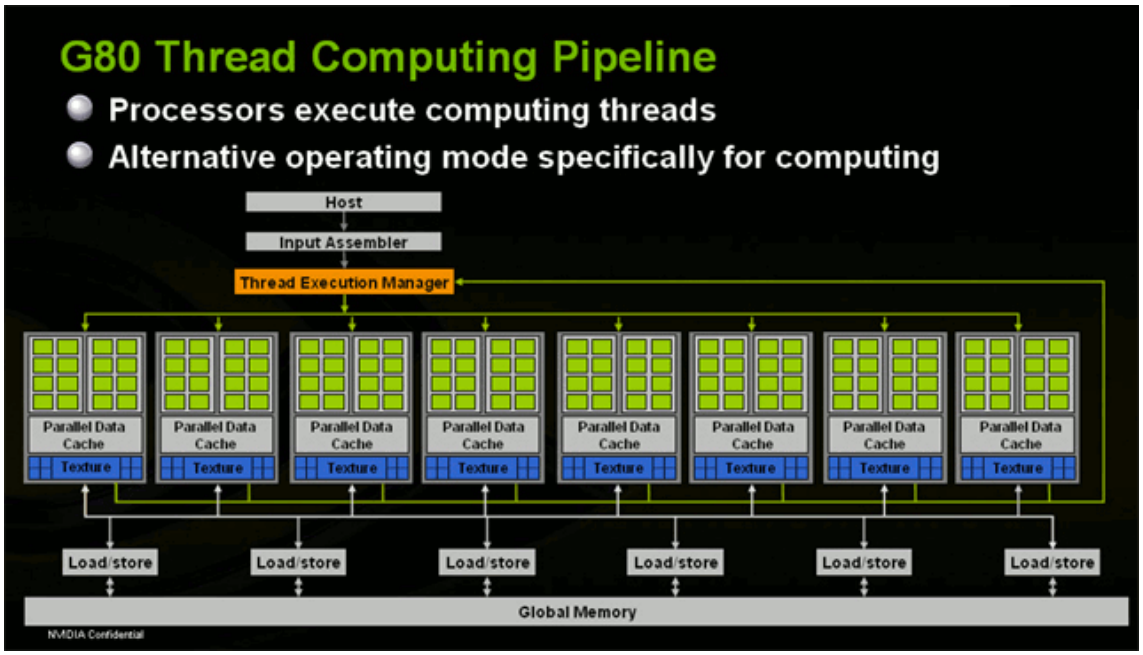


Abbildung 2.16: CUDA-Architektur des G80-Chips von NVIDIA. Quelle: (NVIDIA 2008, Abbildung 9).

mit sich. Werden skalare Operationen durchgeführt, wird nur ein Teil der vorhandenen Rechenleistung genutzt. Aus diesem Grund besitzen NVIDIA Grafikkarten, beginnend mit der G80-Architektur, skalare Recheneinheiten, die vergleichbar mit regulären CPU-Kernen sind. Die Vektorisierung von Operationen bringt dann zwar keinen Geschwindigkeitsgewinn, jedoch werden auch bei skalaren Operationen alle verfügbaren Recheneinheiten genutzt.

2.4.2 Speicherzugriff

Die Art des Speicherzugriffs unterscheidet sich ebenfalls erheblich zwischen den Architekturen. CPUs besitzen eine große Caching-Infrastruktur, die zur Kompensation der Latenz dient (vgl. Casazza 2009). Dazu existieren unterschiedlich große Caches, die entweder pro Rechenkern[17] vorhanden sind oder zwischen den Kernen geteilt werden. Die Befüllung der Caches sowie die Garantie diese konsistent zu halten erfolgt automatisch von der CPU (siehe hierzu auch Sutter 2007; Meyers 2010).

Im Unterschied dazu besitzen GPUs grundsätzlich keine derartige Caching-Architektur. Im Fall des Grafikrenderns sind die Arten des Speicherzugriffs bekannt und wurden von den Herstellern optimiert. Wenn generelle Berechnungen auf der Grafikkarte durchgeführt werden sollen, ist dies jedoch nicht gewährleistet. Zwar besitzen Grafikkarten eine Speicherhierarchie, jedoch muss diese manuell verwaltet werden. Einzige Ausnahme davon sind Texturzugriffe, die separat gecached werden und über separate Textureinheiten, in Abbildung 2.16 blau dargestellt, angespro-

[17] Meist sind dies die L1 und L2 Caches, auf die sehr schnell zugegriffen werden kann.

chen werden. Grund hierfür ist, dass Texturzugriffe in einer anwendungsabhängigen Reihenfolge ablaufen, dabei können lediglich Lokalitäten[18] ausgenutzt werden.

Die Speicherhierarchie teilt sich in drei Teile mit unterschiedlichen Zugriffszeiten auf. Zum einen der Arbeitsspeicher der am meisten Speicherplatz bietet, jedoch benötigen Zugriffe darauf sehr lange, in Abbildung 2.16 mit "Global Memory" bezeichnet. Jede SIMD-Einheit besitzt einen weiteren, für sie lokalen Speicher, auf den deutlich schneller zugegriffen werden kann. In der Abbildung wird er "Parallel Data Cache" genannt, ist jedoch kein echter Cache, da dessen Daten manuell verwaltet werden müssen. Am schnellsten sind die Register, deren Anzahl stark von der Grafikhardware abhängig ist, die jedoch generell deutlich weniger Speicherplatz als der lokale Speicher bieten. Jeder Rechenkern, in der Abbildung als grünes Kästchen dargestellt, besitzt eigene Register und kann nicht auf die der anderen Kerne zugreifen.

Die Architekturunterschiede führen zu ganz unterschiedlichen Bedingungen an den Speicherzugriff. Gleichzeitige Zugriffe mehrerer Kerne auf eng zusammenliegende Bereiche können in der CPU-Architektur zu Geschwindigkeitseinbußen führen (siehe Toub u. a. 2008). Grund hierfür ist, dass Cache-Lines[19] von unterschiedlichen Kernen durch den Prozessor untereinander konsistent gehalten werden müssen. Deshalb sollten verschiedene CPU-Kerne auf im Speicher voneinander getrennten Daten arbeiten.

Im Gegensatz dazu steht der globale Speicherzugriff der GPU. Dabei muss darauf geachtet werden, dass alle Kerne einer SIMD-Einheit gleichzeitig auf benachbarte Speicherbereiche zugreifen. Zudem sollten diese an bestimmten Adressgrenzen ausgerichtet sein. Dadurch können mehrere Speicherzugriffe zu einem einzigen zusammengefasst und, da nur eine Übertragung stattfindet, die Latenz des Zugriffs deutlich reduziert werden. Die Bedingungen für das Zusammenfassen von Speicherzugriffen sind stark von der Hardware abhängig und wurden beispielsweise vom G80 zu neueren Chipsätzen deutlich gelockert (siehe NVIDIA 2009, Abschnitt 3.2.1).

Unabhängig von internen Architekturunterschieden muss ein weiterer wichtiger Faktor berücksichtigt werden. Von Seiten der Grafikhardware kann nicht direkt auf den Arbeitsspeicher oder andere Speichermedien zugegriffen werden. Die Daten müssen deshalb von einem auf der CPU laufenden Programm eingelesen und anschließend von diesem auf die Grafikkarte übertragen werden. Dieses Programm verwaltet die Ausführung der GPU-Programme und muss ebenso Ergebnisse vom Grafikspeicher zurücklesen. Diese Übertragung ist sowohl in ihrer Bandbreite beschränkt als auch mit relativ großer Latenz verbunden. Die für die Übertragung benötigte Zeit muss folglich auf die Dauer der auf der Grafikkarte stattfindenden Berechnungen aufgeschlagen werden.

[18] Wird beim Rendern eines Pixels auf eine bestimmte Stelle in einer Textur zugegriffen, wird der Nachbarpixel mit hoher Wahrscheinlichkeit auf eine nahegelegene Stelle derselben Textur zugreifen.

[19] Cache-Line = kleinste vom Cache verwaltete Einheit.

2.4.3 Speicherallokation

In vielen Algorithmen steht zu Beginn die Größe des Ergebnisses einer Berechnung nicht fest. In diesem Fall wird in der Regel zur Laufzeit Speicher allokiert, der, nachdem er nicht mehr benötigt wird, freigegeben werden kann. Dieser Mechanismus existiert auf Grafikkarten nicht. Der Speicher wird, wie auch die auszuführenden Programme, vom Host[20] verwaltet. Reservierter Speicher wird durch Übergabe eines Zeigers auf dessen Anfang an das entsprechende GPU-Programm übergeben.

Um diese Einschränkung umgehen zu können, sind mehrere Möglichkeiten denkbar:

1. Wenn der maximal benötigte Speicher bekannt und klein genug ist, kann dieser immer im Voraus reserviert werden.

2. Programme, die ihren Speicherverbrauch nicht im Voraus kennen, werden zweimal ausgeführt. Beim ersten Durchlauf wird lediglich der erwartete Speicherverbrauch berechnet. Danach wird dieser Speicherbereich allokiert und kann im zweiten Durchlauf verwendet werden.

3. Eine Mischung der zuvor genannten Verfahren unter Annahme eines bestimmten Speicherverbrauchs. Dieser wird vorab reserviert. Wird der verfügbare Speicher im Programm überschritten, kann für den Rest der Berechnung analog zur zweiten Lösung vorgegangen werden.

Alle genannten Methoden haben Vor- und Nachteile. Vorteil des ersten Verfahrens ist dessen Einfachheit, der Nachteil liegt im Speicherverbrauch. Das mittlere Verfahren benötigt exakt so viel Speicher, wie erforderlich ist. Nachteil ist, dass alle Berechnungen doppelt durchgeführt werden müssen. Zudem müssen zwei Programme ausgeführt und eine Rückmeldung an den Host zur Speicherreservierung gegeben werden. Dadurch steigt die Gesamtdauer weiter. Das letztgenannte Verfahren versucht die Vorteile beider Verfahren auszunutzen. Der Nachteil hierbei ist die steigende Komplexität der Programme. Diese müssen sowohl die Ausgabe erzeugen als auch den benötigten Speicher berechnen können. Zudem ist wünschenswert, dass der Algorithmus an einer zuvor hinterlegten Stelle fortgesetzt werden kann. Die letzten beiden Möglichkeiten können zudem nur bei deterministischen Algorithmen eingesetzt werden, außer es wird gewährleistet, dass der Algorithmus in beiden Durchläufen dieselben Entscheidungen trifft[21].

2.4.4 Zusammenfassung und weitere Entwicklung

Zusammenfassend kann gesagt werden, dass sich die Entwicklung für Grafikkarten stark von der klassischen CPU-Programmierung unterscheidet. Zudem unterscheiden sich die Architekturen verschiedener Hersteller stärker. Dadurch muss bei der Optimierung immer die sinkende Portierbarkeit des Codes berücksichtigt werden.

[20] Als Host wird der Rechner an den die Grafikkarte angeschlossen ist bezeichnet.

[21] Dies ist möglich, wenn Pseudozufallszahlen eingesetzt werden und für beide Durchläufe dieselbe Seed verwendet wird.

Dieser Architekturunterschied wird mit der aktuellen Generation von Grafikkarten noch stärker. Während ATI mit der RV870-Architektur nahe an vorangegangenen Architekturen bleibt und somit beispielsweise weiter auf Vektoreinheiten setzt, übernimmt die Fermi-Architektur Technologien aus dem CPU-Umfeld. Beispielsweise wurden Caches für den generellen Speicherzugriff eingeführt. Dadurch sollen zufällige Zugriffe auf den globalen Speicher deutlich beschleunigt werden, was sich wiederum auf eine Vielzahl von Algorithmen auswirkt. Zudem wird der Laufzeitunterschied zwischen Algorithmen, deren Speicherzugriff von Hand optimiert wurde, und trivialen Implementierungen deutlich kleiner.

Um die ohnehin schon skalaren Recheneinheiten noch besser nutzen zu können, wurde von NVIDIA ein Mechanismus, ähnlich dem von Intel bekannten Hyper-Threading (Casazza 2009), implementiert. Er erlaubt Recheneinheiten, die quasi simultane Ausführung zweier Threads auf einem Rechenkern.

Dieser Trend zeigt, dass der Schwerpunkt auf Seite von NVIDIA auf dem Ausbau der Grafikkarten in Richtung generelle Berechnungen liegt. Insbesondere die hauseigene CUDA-Technologie, die für generelle Berechnungen auf Grafikkarten ausgelegt ist, soll gestärkt werden. Dazu gehört auch die Unterstützung von C++ auf Grafikhardware, weshalb nicht zuletzt die zuvor existierenden drei Adressräume, siehe Abschnitt 2.4.2 Speicherhierarchie, zu einem einzigen zusammengeführt wurden.

Kapitel 3

Implementierung

In diesem Kapitel wird die Implementierung der Berührpunkt- und Markererkennung ausführlich erläutert. Dabei wird zunächst auf die eingesetzten Technologien eingegangen. Für die Visualisierung von Ergebnissen und einzelnen Berechnungsschritten wurde ein minimales GUI[1]-Framework entwickelt. Dadurch konnte auch das Zusammenspiel verschiedener Technologien untersucht werden. Bei den Algorithmen zur Berührpunkt- und Mustererkennung werden Unterschiede zu bestehenden CPU-Implementierungen erklärt.

3.1 Technologien

Ziel der Arbeit ist die Implementierung der Erkennungsalgorithmen auf modernen Grafikprozessoren. Für die Programmierung von generellen Programmen auf der Grafikkarte existieren verschiedene Schnittstellen. Im Rahmen dieser Arbeit wurde die Schnittstelle OpenCL[2] eingesetzt. Sie wird in Abschnitt 3.1.1 kurz vorgestellt und es werden Gründe für deren Einsatz erläutert.

Zur GPU-basierten Berechnung müssen die Daten der Kamera auf die Grafikkarte übertragen werden. Ebenso werden Zwischen- und Endergebnisse der Algorithmen im Grafikkartenspeicher abgelegt. Somit ist die möglichst direkte Visualisierung dieser Daten, ohne sie in den Hauptspeicher des Rechners kopieren zu müssen, naheliegend. Daher wird für die Darstellung OpenGL[3], eine plattformunabhängige Grafikschnittstelle, eingesetzt.

Als Programmiersprache wurde C++ gewählt. Durch die Kompatibilität mit C mussten keine zusätzlichen Frameworks für die Nutzung der als C-Bibliotheken vorliegenden Schnittstellen genutzt werden und es konnte objektorientiert entwickelt werden. Besonders für die Programmierung der graphischen Benutzterschnittstelle war dies von Vorteil.

[1] GUI = Graphical User Interface = Benutzerschnittstelle.
[2] OpenCL = Open Computing Language.
[3] OpenGL = Open Graphics Library.

3.1.1 OpenCL

OpenCL wurde ursprünglich von Apple Inc. ins Leben gerufen und steht unter der Schirmherrschaft der Khronos Group, einer Arbeitsgemeinschaft von über 100 Firmen, die im Medienbereich tätig sind. Sie erstellt offene Standards für eine Vielzahl von Medienplattformen und -geräten. OpenCL gehört zu den neusten Standards und wurde im Dezember 2008 in Version 1.0, welche die Grundlage der nachfolgenden Erläuterungen ist, veröffentlicht (Khronos Group 2009).

Die Schnittstelle ist nicht explizit für Berechnungen auf Grafikkarten ausgelegt, sondern ist als generelle Programmierplattform für verschiedene Gerätetypen, darunter auch mobile Geräte, gedacht. Der Bezug zu Grafikkarten zeigt sich insbesondere durch Erweiterungen für Texturzugriffe oder das Zusammenspiel mit dem in Abschnitt 3.1.2 beschriebenen OpenGL. OpenCL soll somit die Programmierung für verschiedenste Plattformen vereinheitlichen und dennoch eine hardwarenahe Schnittstelle anbieten. Dass dies in einigen Fällen gegensätzliche Anforderungen sind, wird in nachfolgenden Abschnitten dieser Arbeit deutlich.

Im GPGPU[4]-Bereich konkurriert der Standard mit anderen Technologien. Dazu gehört das von NVIDIA stammende CUDA, das jedoch auf Grafikkarten desselben Herstellers beschränkt ist und deshalb bei der Auswahl der Technologie ausscheidet.

Auch das von Microsoft stammende Direct Compute bietet die Möglichkeit generelle Berechnungen auf Grafikhardware durchzuführen und gehört zum DirectX 11 API (Boyd u. a. 2010). Es besitzt eine starke Bindung zur Grafikschnittstelle Direct3D und wurde dort mittels sogenannter Compute Shader implementiert. Primäres Ziel von Direct Compute ist folglich die Anreicherung von 3D-Applikationen durch generelle Berechnungen auf der Grafikkarte. Zwar ist die Visualisierung von Zwischenergebnissen auch im Rahmen dieser Arbeit von Bedeutung, jedoch sollten keine Abhängigkeiten von der Visualisierung entstehen. Aus diesem Grund wurde OpenCL als Technologie zur Berechnung auf Grafikhardware gewählt.

OpenCL beruht auf vier verschiedenen Modellen, welche die Gesamtarchitektur beschreiben. Diese Modelle sind hierarchisch geordnet und werden im Folgenden kurz vorgestellt.

Plattformmodell

Die Architektur von OpenCL besteht auf oberster Ebene aus einem Host und einem oder mehreren Geräten (Devices). Der Host ist für die Verteilung der Arbeit an die verschiedenen Geräte zuständig. Ein Device kann beispielsweise sowohl eine GPU als auch eine CPU sein. Jedes Gerät besteht aus Recheneinheiten (Compute Units), die eine bestimmte Anzahl von Verarbeitungselementen (Processing Elements) zusammenfassen, siehe Abbildung 3.1 a). Vergleicht man die Aufteilung der OpenCL Geräte mit der Architektur moderner Grafikkarten, wie in Abbildung 2.16 dargestellt, fällt deren Ähnlichkeit sofort auf.

[4]GPGPU = General Purose computations on Graphics Processing Units.

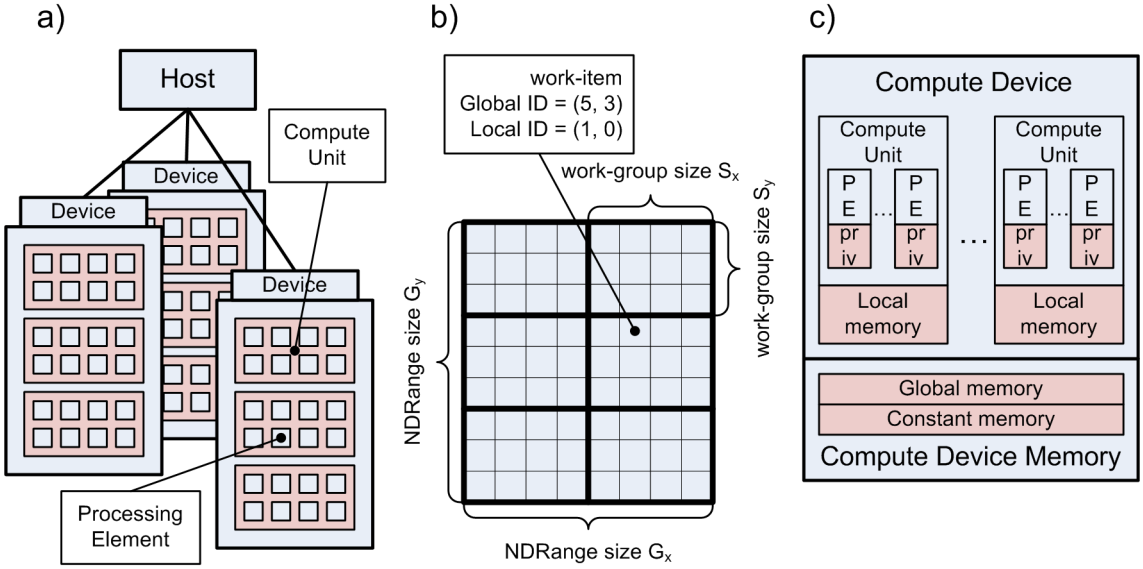


Abbildung 3.1: OpenCL Architekturmodelle: a) Plattformmodell, b) Ausführungsmodell, c) Speichermodell.

Ausführungsmodell

OpenCL Berechnungen werden in sogenannten Kerneln durchgeführt, die innerhalb eines Contexts definiert werden. Er dient als Verwaltungsstruktur und beinhaltet außer den Kerneln Geräte, die für die OpenCL Berechnungen genutzt werden, allokierten Speicher und Programmobjekte, die den Quellcode der Kernel enthalten. Ein Kernel ist eine Funktion mit einer bestimmten Anzahl von Übergabe- und Rückgabeparametern. Kernel werden asynchron zur laufenden Anwendung gestartet. Dies geschieht über eine oder mehrere Befehlswarteschlangen (Command Queues), die Kernel nacheinander oder anhand von Abhängigkeiten untereinander ausführen. Mit Hilfe von Ereignissen (Events) kann auf die Fertigstellung von Kerneln gewartet werden. Falls diese nicht in der Reihenfolge ausgeführt werden sollen, in der sie zur Warteschlange hinzugefügt wurden, können mit Hilfe der Events Abhängigkeiten zwischen ihnen definiert werden.

Ein Kernel wird bei der Ausführung durch die Command Queue auf eine zuvor festgelegte Anzahl von Arbeitsgruppen (work-groups) aufgeteilt. Jede Arbeitsgruppe besteht wiederum aus einer, ebenfalls zuvor festgelegten, Anzahl von Arbeitselementen (work-items). Die Arbeitsgruppen und -elemente bilden dabei jeweils einen Indexraum, der in einen globalen, NDRange[5] genannten, eingebettet ist, siehe hierzu Abbildung 3.1 b). Für jedes Arbeitselement wird dieselbe Kernel-Funktion aufgerufen und führt folglich denselben Code aus. Dieses Ausführungsmodell spiegelt somit die SIMD-Charakteristik von Grafikkarten wider. Innerhalb eines Kernels können mit speziellen Funktionen beispielsweise die Arbeitsgruppe, das Arbeitselement oder der globale Index des Elements erfragt werden. Abhängig von diesen Informationen

[5]NDRange steht für N-dimensionaler Bereich, welcher bis zu dreidimensional sein kann.

kann dann auf entsprechende Speicherbereiche zugegriffen und damit auf separaten Daten gearbeitet werden.

Speichermodell

Der Speicherzugriff ist hierarchisch aufgebaut und besteht aus globalem, lokalem, privatem und konstantem Speicher, wie in Abbildung 3.1 c) dargestellt. Jedes Arbeitselement (PE) besitzt einen privaten Speicher (priv), der von keinem anderen Arbeitselement gelesen oder beschrieben werden kann. Innerhalb einer Arbeitsgruppe kann auf einen geteilten, lokalen Speicher zugegriffen werden. Er ermöglicht somit, Daten zwischen den Arbeitselementen innerhalb einer Arbeitsgruppe auszutauschen. Speicherbarrieren im Kernelcode können für die Synchronisierung dieses Speicherzugriffs genutzt werden.

Der globale Speicher ist von jeder Arbeitsgruppe erreichbar. Auf ihn werden Daten vom Host übertragen und er wird zum Speichern von Berechnungsergebnissen genutzt. Globaler Speicher kann nicht zum Austausch von Daten unter verschiedenen Arbeitsgruppen genutzt werden, da zwischen den Arbeitsgruppen keine Möglichkeit der Synchronisierung besteht. Schreibende Zugriffe auf den globalen Speicher müssen somit zwischen den Arbeitsgruppen in getrennten Bereichen ablaufen. Konstanter Speicher ist eine spezielle Art von globalem Speicher, der allein vom Host beschrieben werden kann.

Die Speicherverwaltung gehört zu den Aufgaben des Hosts. Er muss vor der Ausführung eines Kernels den Speicher auf der Grafikkarte reservieren. Über Zeiger auf diesen Speicher wird den Kerneln der Zugriff auf selbigen ermöglicht. Dynamische Speicherallokation ist in den Kerneln folglich nicht möglich und muss beim Design von Algorithmen beachtet werden.

Auch das Speichermodell von OpenCL deckt sich, durch dessen hierarchischen Aufbau, mit dem moderner Grafikhardware. Dies wird beim direkten Vergleich beider deutlich, siehe Abbildungen 2.16 und 3.1 c).

Programmiermodell

Die Programmierung von OpenCL teilt sich in zwei Bereiche auf:

1. Erstellung des Hostprogramms, das den Speicher und die Ausführung der Kernel verwaltet

2. Programmierung der Kernel

Zur Programmierung des Hostprogramms bietet OpenCL eine C-Bibliothek. Dadurch können Bindings[6] in diverse andere Programmiersprachen erstellt werden. C bietet sich insbesondere durch die Hardwarenähe als Programmiersprache für geschwindigkeitskritische Anwendungen an. Zur Unterstützung von C++ Programmierern ist ebenso ein offizielles Binding für C++ verfügbar. Es bildet eine dünne

[6]Binding = Portierungen von einer ausgehenden Programmiersprache in eine meist höhere Programmiersprache durch Ausnutzung von Interoperabilitätsmechanismen der höheren Sprache.

Abstraktionsschicht über OpenCL und bringt keine Geschwindigkeitseinbußen mit sich. Da der im Rahmen dieser Arbeit entstandene Code auf C++ basiert, wurde dieses Binding genutzt.

Für die Programmierung der Kernel beinhaltet OpenCL eine eigene Programmiersprache. Sie ist eine partielle Erweiterung, aber auch Einschränkung der Programmiersprache C. Sie wurde beispielsweise um native Vektordatentypen erweitert, jedoch werden weder Rekursion noch Funktionszeiger unterstützt. Zeiger sind aufgrund der Speicherhierarchie für eine bestimmte Speicherart definiert und gelten nur in dieser. Kernel-Code wird im Host zur Laufzeit geladen und kompiliert. Somit kann dynamisch Code erzeugt und geändert werden. Zur Kompilierung des Codes liefert jede OpenCL-Implementierung einen Compiler mit, der speziell für die jeweilige Hardware des Geräts zugeschnittenen Maschinencode erzeugt.

Beispiel

An einem typischen Beispiel, der Transposition einer Matrix, wird nun ein Einblick in die Funktionsweise von OpenCL gegeben. Der Code teilt sich dabei in ein Host-Programm und einen Kernel, der vom Host aufgerufen wird. Ein Großteil des OpenCL-Initialisierungscodes im Host wurde zur besseren Übersichtlichkeit weggelassen. Listing 3.1 zeigt die wichtigsten Teile des Host-Codes, die im Anschluss näher erläutert werden.

```
1  void transpose(
2    cl::CommandQueue& queue,
3    cl::Context& context,
4    cl::Kernel& kernel)
5  {
6    // Matrixinitialisierung
7    float* matrix = new float[2048 * 2048];
8    const size_t MatrixSize = 2048 * 2048 * sizeof(float);
9    initMatrix(matrix);
10
11   // Reservierung von globalem Grafikspeicher
12   cl::Buffer in(context, CL_MEM_READ_ONLY, MatrixSize);
13   cl::Buffer out(context, CL_MEM_WRITE_ONLY, MatrixSize);
14
15   // Übertragung der Matrix
16   queue.enqueueWriteBuffer(
17     in,          // zu beschreibender Buffer
18     true,        // auf Übertragungsende warten
19     0,           // an Anfangsposition im Buffer schreiben
20     MatrixSize, // Anzahl der zu schreibenden Bytes
21     matrix       // Zeiger auf Host-Speicherbereich
22     );
23
24   // Übergabeparameter für Kernel setzen
```

```
25   kernel.setArg(0, in);
26   kernel.setArg(1, out);
27   // Größe des lokalen Speichers festlegen
28   kernel.setArg(2, cl::__local(16 * 16 * sizeof(float)));
29   kernel.setArg(3, 16);
30
31   queue.enqueueNDRangeKernel(
32     kernel,
33     NullRange,            // kein Versatz
34     NDRange(2048, 2048), // globale Größe = Matrixgröße
35     NDRange(16, 16)       // Größe der Arbeitsgruppen
36     );
37
38   // Zurücklesen der transponierten Matrix
39   queue.enqueueReadBuffer(
40     out,
41     true,
42     0,
43     MatrixSize,
44     matrix);
45
46   doSomethingWith(matrix);
47
48   delete[] matrix;
49 }
```

Listing 3.1: Codeausschnitt des OpenCL Host-Programms.

Im Hostprogramm wird zu Beginn eine 2048x2048 Matrix erstellt. Nach deren Initialisierung wird Speicher auf der Grafikkarte für die Matrix und deren Transponierte reserviert. Wenn dieser Schritt abgeschlossen ist, werden die Daten der, im Arbeitsspeicher des Rechners liegenden, Matrix in den Grafikspeicher übertragen. Danach können die Übergabeparameter des Kernels festgelegt und dieser ausgeführt werden. Anschließend werden die Ergebnisse zurück in den Arbeitsspeicher des Rechners kopiert.

Der in Listing 3.2 nachfolgend gezeigte Quellcode des Transpositionskernels ist ein einfaches Beispiel, enthält jedoch typische Merkmale von Kernel, wie die Einteilung in vier Bereiche:

1. Abruf der Indizes des aktuellen Elements
2. Einlesen der Daten in den lokalen Speicher
3. Verarbeitung der Daten
4. Schreiben der Resultate in den globalen Speicher

```
1  /* Kerneldefinition
2   * Parameter:
3   *   in             - Eingabematrix
4   *   intermed       - quadratische Matrix für
5   *                    Zwischenergebnisse
6   *   intermed_size  - Größe der Matrix
7   *   out            - Ausgabematrix
8   */
9  __kernel void transpose(__global const float* in,
10   __global float* out,
11   __local float* block,
12   const uint block_size)
13 {
14   /* Abrufen lokaler Informationen */
15   uint2 global_loc
16     = (uint2)(get_global_id(0), get_global_id(1));
17
18   uint2 dimensions
19     = (uint2)(get_global_size(0), get_global_size(1));
20
21   uint2 local_loc
22     = (uint2)(get_local_id(0), get_local_id(1));
23   uint2 group_loc
24     = (uint2)(get_group_id(0), get_group_id(1));
25
26
27   /* Lesen und Transponieren eines Blocks
28     in den lokalen Speicher */
29   block[local_loc.y * block_size + local_loc.x]
30     = in[global_loc.y * dimensions.x + global_loc.x];
31
32   /* Berechnung der transponierten Positionen des Blocks
33     und dessen Inhalt */
34   int2 trans_global_loc
35     = group_loc.yx * block_size + local_loc;
36
37   uint out_global_loc = trans_global_loc.y * dimensions.y
38               + trans_global_loc.x;
39
40   uint out_local_loc
41     = local_loc.x * block_size + local_loc.y;
42
43   /* Warten bis alle Daten in den lokalen Speicher
44     geschrieben sind */
45   barrier(CLK_LOCAL_MEM_FENCE);
46
47   /* Schreiben der Werte vom lokalen Speicher
```

```
48      an die neue Position */
49    out[out_global_loc] = block[out_local_loc];
50  }
```

Listing 3.2: OpenCL Kernel zur Transposition einer Matrix.

Zu Beginn werden die Positionen des zu verarbeitenden Elements im Indexraum erfragt, um Speicherzugriffe entsprechend dieser Positionen durchführen zu können. Anschließend werden Daten vom globalen Speicher in den lokalen Speicher eingelesen. Dies geschieht möglichst so, dass Zugriffe zusammengefasst werden können. Sind die Daten in den lokalen Speicher eingelesen, kann die eigentliche Berechnung, in diesem Fall die Transposition, durchgeführt werden. Beim anschließenden Speichern der Ergebnisse ist darauf zu achten, dass die Zugriffe auf den globalen Speicher zusammengefasst werden können, um auch hier Latenz zu reduzieren. Für den lokalen Speicher gilt diese Einschränkung nicht, weshalb die eigentliche Transposition innerhalb eines Speicherblocks lediglich beim Zugriff auf den lokalen Speicher geschieht.

Das obige Beispiel zeigt, dass die Programmierung für Grafikprozessoren selbst für einfache Aufgaben deutlich mehr Aufwand ist als eine einfache CPU-Implementierung, wie in Listing 3.3 dargestellt. Für die Entscheidung, ob Berechnungen auf der Grafikkarte durchgeführt werden sollen, ist folglich auch der enorm erhöhte Entwicklungsaufwand zu berücksichtigen. Für die im Rahmen der Arbeit erstellte Implementierung der Berührungs- und Markererkennung mussten zur Steigerung der Produktivität verschiedene Abstraktionen entwickelt werden, die beispielsweise Kernel bei Bedarf automatisch laden und kompilieren oder mit gewöhnlicher Funktionssyntax aufrufbar machen.

```
1  void CPUtranspose(void)
2  {
3    // Matrixinitialisierung
4    float* matrix = new float[2048 * 2048];
5    float* matrixTransposed = new float[2048 * 2048];
6
7    initMatrix(matrix);
8
9    // Transposition
10   for(int x = 0; x < 2048; ++x)
11     for(int y = 0; y < 2048; ++y)
12     {
13       matrixTransposed[y * 2048 + x]
14         = matrix[x * 2048 + y];
15     }
16
17   // weitere Berechnungen
18
19   doSomethingWith(matrixTransposed);
20
21   // Speicherfreigabe
```

```
22
23    delete[] matrix;
24    delete[] matrixTransposed;
25 }
```

Listing 3.3: Einfacher sequenzieller Transpositionsalgorithmus für CPUs.

3.1.2 OpenGL

Für die Visualisierung von Ergebnissen wurde OpenGL als Schnittstelle zur Grafikhardware genutzt. OpenGL wird ebenfalls von der Khronos Group spezifiziert und steht in Konkurrenz zu dem von Microsoft stammenden Direct3D. Die Spezifikation der Schnittstelle ist momentan in Version 4.1 verfügbar (Segal u. Akeley 2010). Da die für die Programmierung eingesetzte NVIDIA GeForce 8800 GTS Grafikkarte diese Version nicht unterstützt, basiert die Entwicklung auf der älteren Version 3.2 vom Dezember 2009 (Segal u. Akeley 2009), welche die Basis der nachfolgenden Erläuterungen bildet. Wie auch OpenCL ist OpenGL, im Gegensatz zu Direct3D, ein plattformunabhängiger, offener Standard.

Die Spezifikation teilt sich in zwei Profile auf, da die Funktionsweise von OpenGL im Laufe der Entwicklung massiv überarbeitet wurde. Zum einen existiert das sogenannte "Core Profile", es beinhaltet ausschließlich aktuelle Funktionalität und wurde für die Implementierung verwendet. Zudem gibt es ein "Compatibility Profile", welches alle, auch als veraltet eingestufte, Funktionen enthält. Grund hierfür ist die große Masse an bestehender Software, die nicht ohne weiteres vollständig auf neue Versionen umgeschrieben werden kann. Diese Anwendungen sollen durch das "Compatibility Profile" somit weiterhin von neuen Entwicklungen profitieren.

OpenGL wurde zum Echtzeit-Rendern von 2D- und 3D- Computergrafiken entwickelt. Es bietet Funktionen zum Rendern von Punkten, Linien und Polygonen auf dem Bildschirm. Dazu werden Eckpunkte (Vertices) von Primitiven an die Grafikkarte übertragen. Durch Programme, die auf der Grafikkarte ablaufen, werden diese Primitiven verändert und schließlich als farbige Pixel auf dem Bildschirm angezeigt.

In Abbildung 3.2 ist ein stark vereinfachter Ablauf des Rendervorgangs dargestellt. Der Datenfluss ist von links nach rechts dargestellt. Vertices werden durch Zeichenaufrufe aus der Applikation an die Grafikkarte übertragen. Dort läuft der Vertex-Shader ab, eine Funktion, die für jeden Eckpunkt einmal aufgerufen wird. In dieser Funktion wird beim dreidimensionalen Rendern in der Regel eine perspektivische Transformation durchgeführt. Dabei werden die 3D-Koordinaten durch eine Matrixmultiplikation in 2D-Koordinaten umgerechnet. Danach werden die eigentlichen Primitiven aus den projizierten Vertices generiert und diese rasterisiert. Dies bedeutet, dass aus jeder Primitive ein Pixelbild erzeugt wird. Für jedes Pixel in den generierten Pixelbildern wird anschließend der Fragment-Shader aufgerufen. Er ist unter anderem typischerweise für die Lichtberechnung und Texturierung zuständig. Die Ergebnisse des Fragment-Shaders werden letztlich in den Framebuffer geschrieben, der auf dem Bildschirm dargestellt wird.

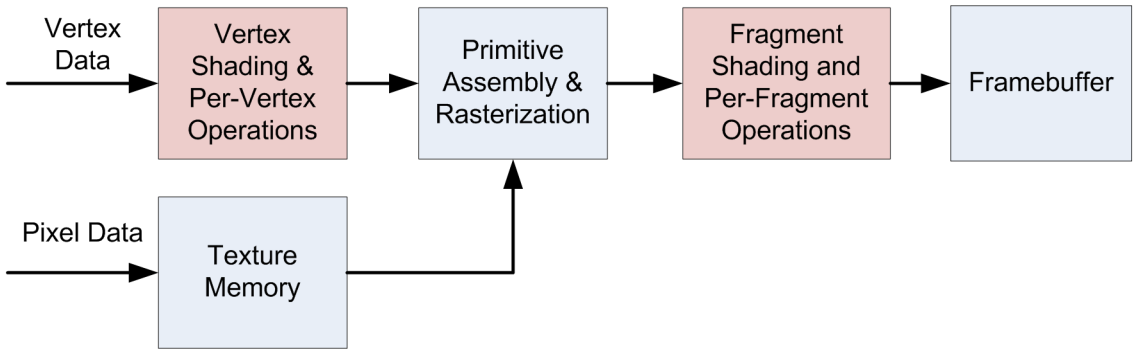


Abbildung 3.2: Stark vereinfachte OpenGL Architektur (basierend auf Segal u. Akeley 2010, Abbildung 2.1).

Da OpenGL als C-Bibliothek zur Verfügung steht, zur Implementierung jedoch C++ zum Einsatz kam, wurde ein C++ Wrapper, im Folgenden EGL genannt, für OpenGL programmiert. Hauptaufgabe dieses Wrappers ist die Objektstruktur der OpenGL 3.2 Schnittstelle als C++ Objekte verfügbar zu machen, dabei wurden jedoch Teile der Bibliothek, die im Rahmen der Thesis nicht benötigt wurden, nicht implementiert. EGL unterstützt zusätzlich einige Komfortfunktionen, welche die Entwicklung beschleunigen:

- Durch Überladen von Methoden sind Suffixe in Funktionsnamen unnötig.
- Methoden zum Schreiben in und Kopieren und Lesen von Puffer und Texturen erkennen Datentypen zur Kompilierzeit und reichen diese Informationen korrekt an die entsprechenden OpenGL Funktionen weiter[7].
- Durch Nutzung von STL[8]-Containern muss die Größe von dynamisch allokiertem Speicher nicht explizit angegeben werden.
- Vektor- und Matrixdatentypen, die verschiedene mathematische Operatoren unterstützen, wurden implementiert und können in EGL-Methoden verwendet werden.
- 100%ige Kompatibilität zu nativen OpenGL-Funktionen.

3.1.3 Interoperabilität

Um Zwischenergebnisse für die Visualisierung nicht unnötig in den Hauptspeicher übertragen zu müssen, ist die Interoperabilität beider Grafikschnittstellen wichtig. Zwar wird von OpenCL der Zugriff auf Ressourcen von OpenGL nicht zwingend

[7]OpenGL-Funktionen, wie beispielsweise "glTexImage2D", nehmen sowohl einen void-Zeiger als Parameter für die zu kopierenden Daten als auch einen Integer Parameter für deren Datentyp entgegen. Somit muss der Programmierer den Datentyp immer explizit angeben.

[8]STL = Standard Template Library = Standardbibliothek der C++ Programmiersprache.

unterstützt, er ist jedoch über Erweiterungen (Khronos Group 2009, Abschnitte 9.11 und 9.12), die sowohl von NVIDIA als auch von ATI unterstützt werden, möglich.

Dazu wird ein OpenCL Context aus einem OpenGL Context erzeugt. Anschließend können OpenCL Buffer und Images aus OpenGL-Objekten erzeugt werden. Die Verwaltung dieser Objekte unterliegt jedoch weiterhin OpenGL. Wird beispielsweise der Speicher einer Textur in OpenGL freigegeben, wird das zugehörige OpenCL Image-Objekt ungültig. Um Inkonsistenzen zwischen den Schnittstellen beim Zugriff auf geteilte Objekte zu vermeiden, müssen diese vor ihrer Verwendung in OpenCL erst akquiriert und anschließend wieder freigegeben werden. In dieser Phase dürfen laut Standard keine OpenGL-Befehle auf die von OpenCL genutzten Daten stattfinden[9].

Die Synchronisierung beider Schnittstellen gestaltet sich in der Praxis problematisch. OpenGL führt Befehle asynchron aus. Aufgrund des Fehlens von Ereignissen kann nicht sicher festgestellt werden, wann ein Befehl vollständig ausgeführt wurde. Die einzige Möglichkeit, die Fertigstellung zu garantieren, ist mit Hilfe der `glFinish` Funktion. Diese wartet jedoch auf alle ausstehenden Befehle. Da OpenGL zudem nativ keine Nebenläufigkeit unterstützt und OpenCL nur sehr schwache Garantien macht, ist extensives Locking erforderlich.

Dieses Synchronisationsproblem wurde inzwischen vom Standardisierungskomitee erkannt und in den neuen Versionen der Schnittstellen behoben. Die Version 1.1 der OpenCL-Spezifikation (Munshi 2010) erweitert die Garantien für Nebenläufigkeit. Somit ist beispielsweise auch die Command Queue threadsicher. Des weiteren erlaubt OpenCL 1.1 das Erzeugen von Event-Objekten aus OpenGL-Synchronisationsobjekten. Eine mit OpenGL 4.1 erschienene Erweiterung (Leech 2010), die kompatibel zu älteren OpenGL-Versionen ist, bietet analog dazu die Möglichkeit aus OpenCL-Events Synchronisationsobjekte zu erzeugen. Eine Portierung des Codes war jedoch im Rahmen dieser Arbeit mangels Treiber bzw. Unterstützung der Grafikhardware nicht möglich.

3.2 GUI-Framework

Basierend auf EGL und OpenGL wurde ein minimalistisches GUI-Framework entwickelt, das die Darstellung von Zwischenergebnissen und eine interaktive Bedienung erlaubt. Ziel ist es, die Erweiterbarkeit der Architektur zu gewährleisten, um langfristig eine vollwertige Umgebung zur Erstellung von Multi-Touch-Anwendungen bereitzustellen. Zur Umsetzung wurden bewusst neue Technologien des Renderns eingesetzt und auf Altlasten, wie beispielsweise die OpenGL-Funktionen des Compatibility Profiles, verzichtet. Wie auch die restliche Implementierung wurde das Framework in C++ geschrieben.

[9] Bei der Implementierung zeigte sich jedoch, dass jegliche OpenGL Aufrufe in dieser Phase zu Inkonsistenzen und letztlich zum Programmabsturz führen können.

3.2.1 Komponenten

Die Architektur der GUI-Komponenten bilden, wie in Abbildung 3.3 dargestellt, eine Klassenhierarchie. Die Komponenten sind ereignisbasiert, es wird lediglich gerendert, wenn Änderungen an der graphischen Oberfläche stattgefunden haben. Dabei werden möglichst nur Bereiche aktualisiert, die sich tatsächlich geändert haben. Die Aufgaben der dargestellten Komponenten werden nachfolgend erläutert.

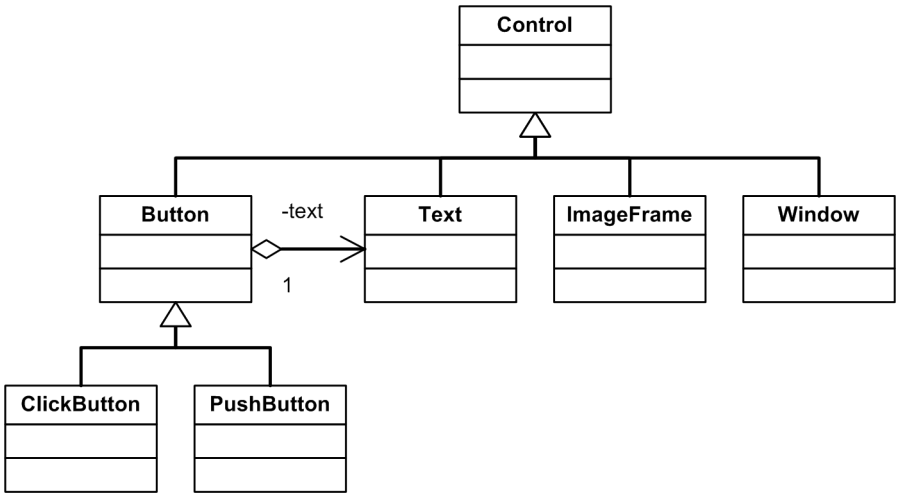


Abbildung 3.3: UML-Klassendiagramm der Komponentenhierarchie des GUI-Frameworks.

Control

Die Control-Klasse bildet das Grundgerüst des Frameworks ab. Dazu gehören standard Event-Handler und die Abbildung der Komponentenstruktur zur Laufzeit. Für letzteres besitzt sie Methoden zum Hinzufügen und Entfernen von Kindkomponenten. Somit ist jede Komponente, die von Control erbt, ein Container für weitere Komponenten. Ein Fenster kann beispielsweise mehrere Buttons und diese wiederum Symbolgrafiken und Text enthalten.

Button

Der Button dient als abstrakte Basisklasse für zwei verschiedene Arten von Schaltflächen, Push- und ClickButtons. Diese unterscheiden sich in der Art, wie sie auf Klicks reagieren: Während der PushButton durch Klicken aktiviert und durch erneutes Klicken deaktiviert wird, bleibt der ClickButton lediglich, während die Maustaste gedrückt ist, aktiv und verhält sich sonst gleich wie gewöhnliche Schaltflächen.

Im deaktivierten Zustand sind Schaltflächen durch einen roten Farbverlauf gekennzeichnet. Im aktivierten Zustand ändert sich die Farbe zu grün. Zur Unterscheidung verschiedener Buttons kann für jede Schaltfläche ein Text definiert werden, der automatisch anhand der Höhe der Schaltfläche skaliert wird und weiß ist.

Text

Diese Komponente ist für die Darstellung von Text verantwortlich. Der Hintergrund ist immer transparent. Die Textfarbe kann durch Aufruf entsprechender Funktionen geändert werden. Die Anzahl der schreibbaren Zeichen pro Textkomponente ist auf 500 begrenzt und eine mehrzeilige Ausgabe ist nicht möglich. Für den praktischen Einsatz in Schaltflächen und für die Ausgabe von kurzen Informationen stellt dies jedoch keine Einschränkung dar und ermöglicht eine vereinfachte Implementierung, siehe Abschnitt 3.2.3.

ImageFrame

Zum Debuggen und zur Visualisierung von Zwischenergebnissen der Bildverarbeitung wurde die ImageFrame-Komponente entwickelt. Sie ermöglicht die Darstellung von Texturen, die im Grafikspeicher abgelegt sind. Auf diese Texturen kann von OpenCL, wie in Abschnitt 3.1.3 beschrieben, mit Hilfe einer Erweiterung zugegriffen werden.

Window

Die Window-Klasse stellt die Verknüpfung zum nativen Fenstersystem her. Sie ist für die OpenGL Kontexterzeugung zuständig, verwaltet die vom Betriebssystem kommenden Ereignisse und veranlasst das Neuzeichnen der Bedienoberfläche.

Das Fenster ist zur Laufzeit die Wurzel des gesamten Komponentenbaums und für die Funktion des Systems essenziell. Aufgrund der Verbindung zum Fenstersystem ist diese Komponente nicht plattformunabhängig und wurde im Rahmen dieser Arbeit für Microsoft Windows® implementiert.

3.2.2 Ereignisse

Wie bereits erwähnt ist das Framework ereignisbasiert. Ereignisse können in drei Arten unterteilt werden:

1. Ereignisse für das Rendern
2. Mausereignisse
3. Tastaturereignisse

Da die Fensterklasse die Verbindung zwischen Betriebssystem und GUI-Framework herstellt, gelangen alle Ereignisse über diese in das Framework.

Tastaturereignisse werden an die in der Fensterklasse als aktuell fokusiert hinterlegte Komponente direkt weitergeleitet.

Im Fall der Mausereignisse ist das Verteilen komplizierter. Da nicht nur einzelne Mausklicks, sondern auch Drag & Drop unterstützt werden soll, muss die Mausbewegung innerhalb des Fensters verfolgt und analysiert werden. Dies geschieht über die Objekthierarchie. Dazu wird für jede Komponente im Fenster überprüft, ob sie

sich unter dem Mauszeiger befindet: Jede Komponente in der derzeitigen Implementierung ist rechteckig und kann nicht rotiert werden. Dadurch ist die Überprüfung durch einen Vergleich der Zeigerposition mit der linken oberen und der rechten unteren Ecke der Komponente sehr einfach. Liegt eine Komponente unter dem Zeiger, werden die Mauszeigerkoordinaten in das Koordinatensystem der Komponente umgerechnet und ein Vergleich mit all ihren Kindern durchgeführt. Befindet sich keine Kindkomponente unter dem Mauszeiger, so befindet sich die aktuelle Komponente unter dem Zeiger. Bei der Iteration über die Kinder muss darauf geachtet werden, dass sie entgegengesetzt der Darstellungsreihenfolge durchlaufen werden, damit Komponenten, die sich im Vordergrund befinden, zuerst ausgewählt werden. Wird eine Komponente mit der Maus beispielsweise durch Klicken ausgewählt, wird das Ereignis mit den Koordinaten an die Event-Handler der entsprechenden Komponente weitergeleitet. Außerdem wird die Zielkomponente für Tastaturereignisse entsprechend angepasst.

Ein Event-Handler ist eine Funktion, die auf bestimmte Ereignisse reagiert. Dabei wird zwischen zwei Arten unterschieden:

Alle Komponenten erben für die Ereignisbehandlung vorgesehene Methoden von der Control Klasse, die je nach Bedarf überschrieben werden können. Diese Art der Ereignisbehandlung ist zur Implementierung der Basisfunktionalität einer Komponente gedacht. Im Fall der Schaltfläche ändert sich beispielsweise deren Farbe beim Anklicken. Diese Methoden werden beim Auftreten eines Ereignisses zuerst aufgerufen.

Die zweite Art sind Event-Handler, die vom Benutzer zur Laufzeit einer Komponente hinzugefügt werden können. Sie haben generell keinen Einfluss auf die Basisfunktionalität. Um das Hinzufügen und Entfernen der Event-Handler möglichst flexibel zu gestalten, nutzt das Framework die Signals and Slots Implementierung von Boost: `Boost.Signals2`[10]. Signals und Slots ist eine Generalisierung des Observer Design Patterns (Gamma u. a. 1995).

Durch diese Aufteilung wird die Basisfunktionalität einer Komponente strikt vom Benutzercode getrennt und kann nicht versehentlich manipuliert oder überschrieben werden.

Auch Renderereignisse werden durch die Objekthierarchie propagiert. In der Regel ändern sich nur Teile des Fensterinhalts, die dann neu gezeichnet werden müssen. Deshalb wird vor dem Rendern einer Komponente zuerst geprüft, ob diese überhaupt neu gezeichnet werden muss. Ist dies der Fall, zeichnet sie sich erst selbst und lässt ihre Kinder gegebenenfalls auch neu zeichnen. Dazu ist das Rendern in zwei Abschnitte unterteilt: Die `onDraw` Methode, die wie bereits beschrieben rekursiv den kompletten Objektbaum durchläuft und dabei den Zeichenbereich auf die Größe der jeweiligen Komponente beschränkt[11]. Sie ruft für jedes Objekt, das neu gezeichnet werden muss, die `onPaint` Methode und anschließend die entsprechenden Slots auf. Beide bilden den zweiten Teil des Renderns, nämlich das eigentliche Darstellen der Komponenten, der im folgenden Abschnitt näher erläutert wird.

[10] `http://www.boost.org/doc/libs/1_42_0/doc/html/signals2.html`.

[11] Dies geschieht mit Hilfe des Scissor-Tests von OpenGL.

3.2.3 Rendern

Komponenten, die auf dem Bildschirm dargestellt werden sollen, werden in der onPaint Methode gerendert. Hierzu werden Vertex-Buffer, also ein Speicherbereich auf der Grafikkarte zum Ablegen von Eckpunkten, allokiert und diese mit entsprechenden Daten gefüllt. Zusätzlich wird das Aussehen einer Komponente in einer Textur ebenfalls auf der Grafikkarte gespeichert. Im Fall der Schaltflächen genügen ein Vertex-Buffer und zwei Texturen pro Klasse, da alle Instanzen dieser Komponenten identisch aussehen und entweder aktiviert oder deaktiviert sein können.

Die Komponenten werden durch Aufruf einer OpenGL-Funktion gezeichnet, die festlegt, wie die Vertexdaten interpretiert werden, und die Daten zusammen mit der Position und der Größe der Komponente an die Shaderprogramme weiterleitet. Diese Shader sind sehr einfach aufgebaut: Der Vertex-Shader transformiert die Vertices entsprechend der Positions- und Größenangabe. Der Fragment-Shader färbt die aus der Rasterisierung kommenden Bildpunkte entsprechend ihrer Texturen ein.

Damit der Zeichenvorgang für den Benutzer nicht sichtbar ist und er immer nur das vollständig gerenderte Bild sieht, wird Double-Buffering eingesetzt. Dies bedeutet, dass immer zwei Framebuffer existieren. Einer, der gezeichnet wird, und ein weiterer der auf dem Bildschirm angezeigt wird. Ist das Bild vollständig gezeichnet, werden die Buffer vertauscht.

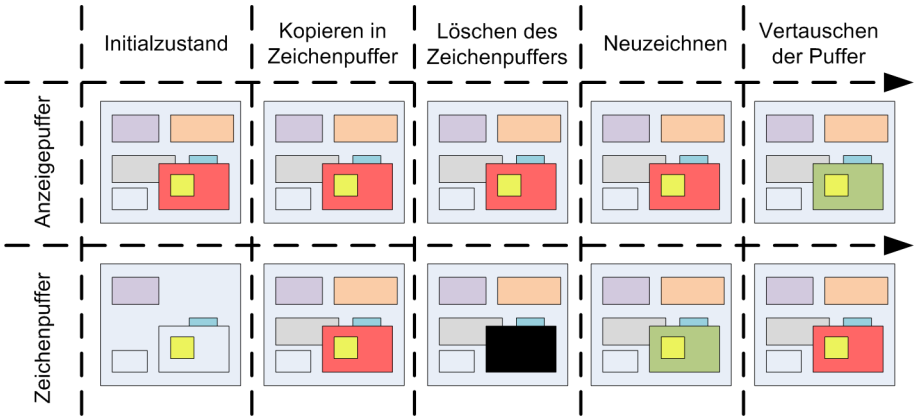


Abbildung 3.4: Double Buffering: Ablauf beim Aktualisieren der rot dargestellten Komponente.

Ändert sich ein Teil des Fensters, wird zunächst das aktuell dargestellte Bild in den für das Zeichnen zuständigen Buffer kopiert. Anschließend wird der zu aktualisierende Zeichenbereich gelöscht[12] und auf den gelöschten Bereich beschränkt. Danach werden alle Komponenten, die sich im Sichtbereich befinden, ihrer Reihenfolge entsprechend gezeichnet. Im letzten Schritt werden die Buffer vertauscht und das neue Bild angezeigt. Auf diese Weise wird immer ein konsistentes Bild gerendert und dennoch der Zeichenaufwand minimiert. Der Vorgang ist in Abbildung 3.4 für die rote Komponente dargestellt.

[12] Alle Pixel in diesem Bereich werden auf schwarz gesetzt.

Rendern von Text

Aufgrund der fehlenden Unterstützung von OpenGL zum Rendern von Schriften musste hierfür ein Mechanismus entwickelt werden. Die Text-Komponente besitzt eine Textur, die alle Schriftzeichen, die dargestellt werden können beinhaltet, sie wird Bitmap-Font genannt. Jedes Textobjekt besitzt einen eigenen Speicherbereich auf der Grafikkarte. In diesem Bereich wird der darzustellende Text abgelegt.

Mit Geometry Instancing wird für jeden Buchstaben ein eigenes Rechteck gerendert. In einem speziellen Vertex-Shader werden die Rechtecke aufsteigend nach Instanz ID[13] bündig nebeneinander angeordnet. Die Breite der Rechtecke berechnet sich dabei über ihre Höhe. Die Beschränkung des Zeichenbereichs garantiert, dass der Text nicht über die Komponentenbreite hinaus geschrieben wird. Jedes Rechteck muss nun mit dem richtigen Schriftzeichen, das in der Textur abgelegt ist, gerendert werden. Dazu greift der Shader mit der Instanz ID als Index auf den Speicherbereich, der den Text enthält, zu und erhält somit den aktuell zu rendernden Buchstaben. Aus dessen Wert werden die Texturkoordinaten des zu rendernden Zeichens berechnet. Für die in Abbildung 3.5 dargestellte Textur ergibt sich dann als Formel für die linke obere Ecke eines Buchstabens:

$$P_x = \frac{Buchstabe \bmod 16}{16} \tag{3.1}$$

$$P_y = \frac{floor(\frac{Buchstabe}{16})}{16} \tag{3.2}$$

Die `floor` Funktion schneidet Nachkommastellen ab. Die Texturkoordinaten sind normiert, das heißt sie sind immer in einem Wertebereich zwischen null und eins. Dies hat den Vorteil, dass die tatsächliche Auflösung des Bildes keine Rolle spielt, solange dessen Aufbau mit 16 Zeichen pro Zeile und Spalte gleich bleibt.

Im Fragment-Shader wird für jedes angezeigte Pixel der Farbwert an den entsprechenden Texturkoordinaten abgefragt und mit einem dem Shader übergebenen Farbwert multipliziert. Somit kann auch farbiger Text angezeigt werden. Zu beachten ist, dass der Hintergrund im Originalbild nicht wie in Abbildung 3.5 schwarz, sondern transparent ist.

Dieses Renderverfahren für Schriften kann sehr effizient implementiert werden, hat jedoch den Nachteil, dass für jede Schrift eine Textur im Grafikspeicher abgelegt werden muss. Wird die Schrift sehr groß angezeigt, muss die Bitmap-Font auch eine entsprechende Auflösung besitzen, da sonst Pixel in der Schrift erkennbar werden. Durch eine höher aufgelöste Schrift steigt jedoch auch der benötigte Speicherplatz für die Textur.

[13] OpenGL übergibt dem Shader beim instanzierten Rendern eine Id, die mit jeder gerenderten Instanz inkrementiert wird.

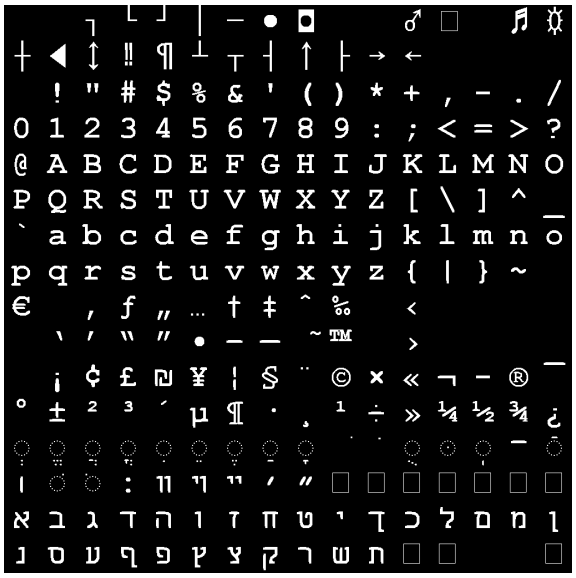


Abbildung 3.5: Bitmap-Font für die Darstellung des gesamten ASCII Zeichensatzes. Erstellt mit Bitmap Font Builder: `http://www.lmnopc.com/bitmapfontbuilder`.

3.3 Vorverarbeitung

Sowohl die Berührpunkterkennung als auch die Markererkennung nutzen das von der IR-Kamera aufgenommene Bild, siehe Abbildung 3.6, als Grundlage. Dieses ist jedoch im initialen Zustand nicht sehr gut für die weitere Verarbeitung geeignet.

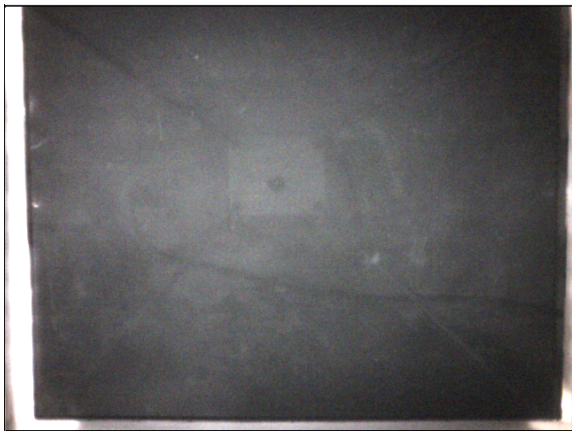


Abbildung 3.6: Bild der Infrarotkamera.

Im Bild sind drei unterschiedliche Eigenschaften erkennbar, die sich negativ auswirken:

1. Im Bild ist viel Rauschen sichtbar. Die Ursache hierfür liegt am kleinen Bildsensor der Kamera und der notwendigen Verstärkung des Signals.

2. Die seitlich abstrahlenden LEDs sind als helle vertikale Balken erkennbar. Zudem ist der Hintergrund, also das generell Richtung Kamera austretende IR-Licht erkennbar.

3. Die Helligkeit ist mit steigendem Abstand zum Bildzentrum reduziert. Gründe hierfür sind die Winkelabhängigkeit des LC-Panels gegenüber infrarotem Licht, der vor der Kamera angebrachte Filter, sowie ein Randlichtabfall (Kerr 2007).

In einem Vorverarbeitungsschritt wird nun versucht, diese Eigenschaften zu korrigieren, um ein geeignetes Bild für die Berührpunkt- und Markererkennung zu erhalten. Die Vorverarbeitung läuft ausschließlich auf der Grafikkarte ab. Dazu wurden für die einzelnen Korrekturen OpenCL Kernel implementiert. Sie lesen und schreiben Daten vom Typ Image anstatt regulärer Speicherpuffer. Dies hat mehrere Vorteile in der Vorverarbeitung oder auch in weiteren Verarbeitungsschritten:

- Texturkoordinaten, die sich außerhalb der Textur befinden, können automatisch in Zugriffe auf den nächstliegenden Randpixel umgerechnet werden.

- Zugriffe auf den Texturspeicher können bilinear gefiltert werden. Dies bedeutet, dass die Koordinaten als Gleitkommazahlen angegeben werden können und der entsprechende Pixelwert von der Grafikhardware durch lineare Interpolation der umliegenden Pixelwerte errechnet wird.

- Der Texturspeicher ist gecached, somit können die Zugriffszeiten für zufällige Zugriffsmuster reduziert werden.

Für die Vorverarbeitung ist lediglich der erste Punkt von Bedeutung, da die Zugriffsmuster der Filter bekannt sind und nicht zwischen den Pixeln gelesen wird. Durch Einsatz der Bilddatentypen auch bei der Vorverarbeitung mussten die Daten jedoch nicht für die folgenden Schritte umgewandelt werden. Durch die erwartet hohe Hit Rate[14] von ca. 90 %[15] wurde somit ohne umfangreiche manuelle Optimierungen eine gute Performance erreicht.

Rauschunterdrückung

Das Rauschen im Bild stört durch dessen hohe Frequenz besonders die Kantendetektionsfilter, kann aber auch zu Fehlerkennungen von Berührpunkten führen und sollte deshalb möglichst aus dem Bild entfernt werden. Dazu wurde ein separierter Binomialfilter eingesetzt, dessen Filtergröße nicht zu groß gewählt werden durfte, da sonst zu viele Details, die für die weitere Verarbeitung benötigt werden, verloren gehen würden. Als optimal stellte sich experimentell die Binomialfiltermaske B^8, siehe Gleichung (2.22), heraus, die eine Standardabweichung von $\sigma \approx 1,414$ besitzt.

[14]Die Hit Rate gibt das Verhältnis von Speicherzugriffen, die vom Cache erfüllt werden können zu den gesamten Speicherzugriffen an.

[15]Die Hit Rate des Caches wurde mit dem Profiling Tool *openclprof* von NVIDA ermittelt. Es ist ein Bestandteil des CUDA SDKs und unter `http://developer.nvidia.com/object/cuda_3_1_downloads.html` erhältlich.

Ein Vergleich vor und nach der Filterung in horizontaler und vertikaler Richtung ist in Abbildung 3.7 dargestellt.

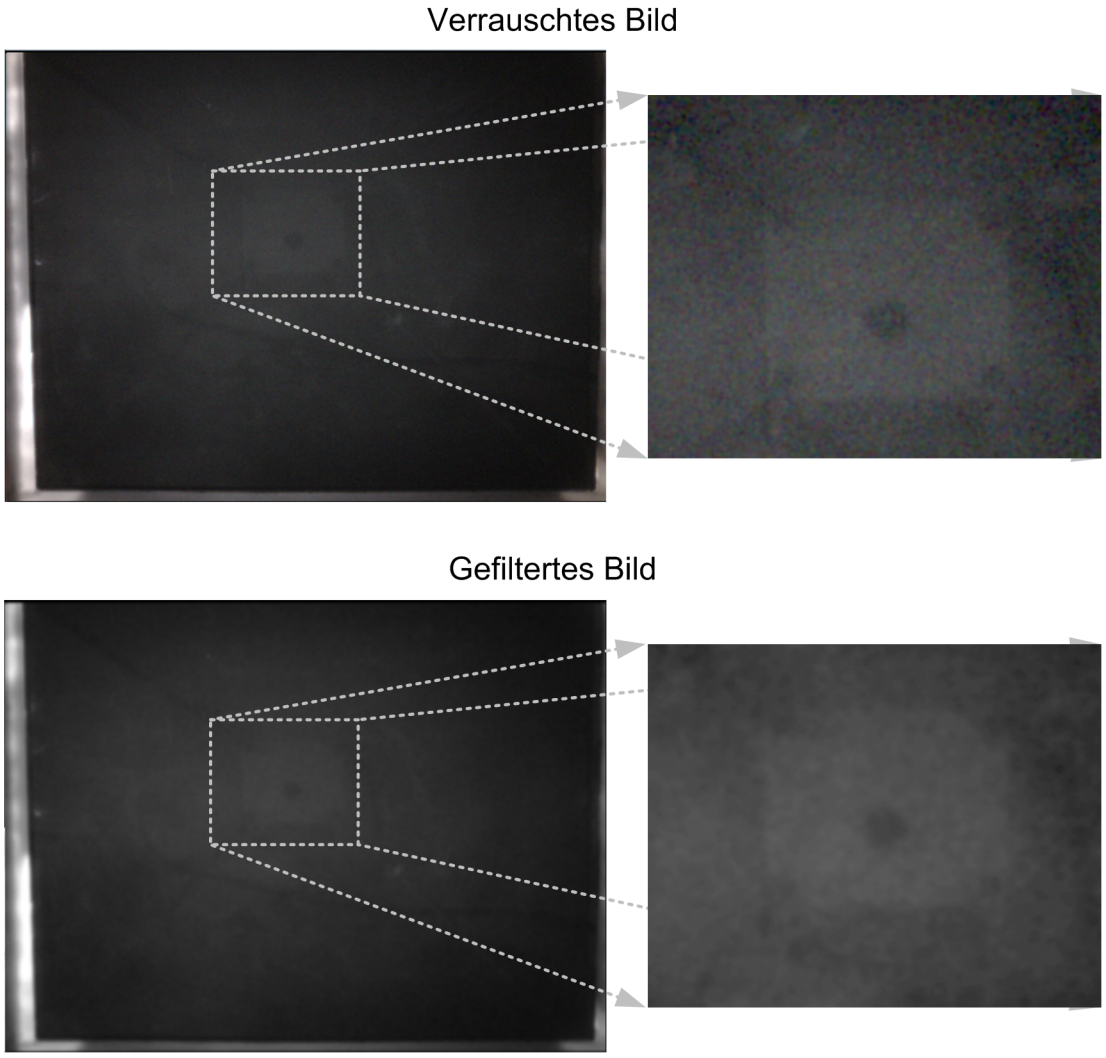


Abbildung 3.7: Anwendung des Binomialfilters auf das Kamerabild.

Hintergrundentfernung

Wie bereits in Abschnitt 2.1.2 erwähnt wurde, strahlt ein Teil des Infrarotlichts direkt in Richtung Kamera. Zudem nimmt die Kamera sich selbst durch Reflexionen an der glatten Oberfläche der Plexiglas®-Platten wahr. Diese Störungen des Bildes sind statisch und werden deshalb als Hintergrund bezeichnet. Da sich dieser Hintergrund in der Regel nicht oder nur sehr selten ändert, kann er als statisches Bild angenommen werden. Veränderungen treten beispielsweise durch wechselnde Sonneneinstrahlung oder Glühbirnen auf, die sich im Sichtbereich befinden und ein- oder ausgeschaltet werden. Adaptive Implementierungen der Hintergrundentfernung sind zwar denkbar, jedoch bezüglich dieses Prototyps zweitrangig.

Durch Annahme eines sich nicht verändernden Hintergrunds kann die Hintergrundentfernung relativ einfach implementiert werden. Dazu wird ein Bild der Kamera, das lediglich den Hintergrund enthält, separat abgespeichert und von jedem neuen Kamerabild subtrahiert. Um die Qualität der Hintergrundentfernung zu verbessern wird das Hintergrundbild doppelt binomial gefiltert und enthält somit deutlich weniger Rauschen als die Vordergrundbilder. Das resultierende Bild ist, das Rauschen ausgenommen, schwarz, siehe Abbildung 3.8 b).

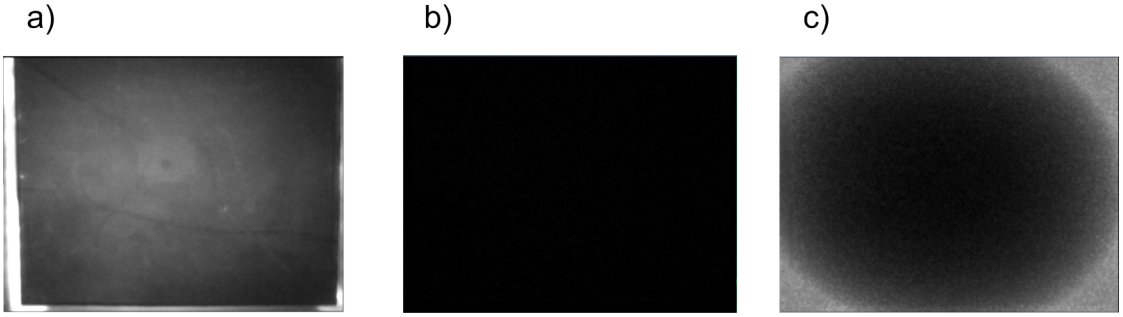


Abbildung 3.8: Bildvorverarbeitung: a) geglättetes Bild, b) nach Hintergrundentfernung, c) nach radialer Verstärkung.

Radiale Verstärkung

Während die ersten beiden negativen Eigenschaften relativ leicht beseitigt werden können, ist dies bei der dritten schwieriger. Das von der Kamera wahrgenommene Bild wird radial vom Zentrum weg schwächer. Ein Grund dafür ist der unterschiedliche Blickwinkel der Kamera auf die einzelnen Pixel des LC-Panels. Im Zentrum des Panels kann das IR-Licht senkrecht durch das Panel gelangen, an den Rändern wird nur diagonal zur Kamera strahlendes Licht wahrgenommen. Dieses Licht passiert zudem den IR-Bandpassfilter diagonal. Dieser dunkelt das Bild selbst bei senkrechter Durchdringung mit Licht im Durchlasswellenlängenbereich[16] ab. Hinzu kommt ein weiterer Faktor für die Abdunklung in Richtung der Bildrändern, der in (Kerr 2007) hergeleitete Randlichtabfall. Er ist bedingt durch den Aufbau des Kameraobjektivs.

Ziel ist, dass das Licht an den Rändern genauso hell im Bild zu sehen ist, wie im Zentrum. Dazu müssen Bereiche im Bild abhängig von deren Position verstärkt werden. Hierfür wurde zuerst gemessen, wie groß die Helligkeitsunterschiede über dem gesamten Bildbereich sind. Basierend auf diesen Angaben wurde eine zweidimensionale Polynomfunktion[17] zweiten Grades auf den Verlauf durch Minimierung der Fehlerquadrate angepasst. Hierfür wurde die Funktion `polyfitweighted2` von Salman Rogers[18] verwendet und anschließend ein OpenCL-Kernel entwickelt, der auf

[16] Der Filter lässt 90 % des Lichts mit einer Wellenlänge von 850 nm passieren.

[17] Die Funktionswerte dieser Funktion lassen sich in einem Kernel schnell berechnen.

[18] `http://www.mathworks.com/matlabcentral/fileexchange/13719-2d-weighted-polynomial-fitting-and-evaluation`.

Basis der aus der Optimierung gewonnenen Funktionsparameter das Kamerabild abhängig vom Verlauf der Funktion verstärkt.

Abbildung 3.8c) zeigt das subtrahierte Bild nach Anwendung des Kernels. Aus der Abbildung wird ersichtlich, dass sich durch die Verstärkung des Bildes auch das Rauschen verstärkt. Für die Verstärkung wurde ein Maximalwert festgelegt. Deshalb nimmt das Rauschen in den Ecken wieder ab. Ohne dieses Maximum wären die Eckbereiche allein durch das auftretende Rauschen vollständig weiß, was für die weitere Bildverarbeitung unvorteilhaft wäre.

Die radiale Verstärkung des Bildes kann folglich keine perfekte Negierung der Blickwinkelabhängigkeit erreichen, da durch die Verstärkung auch das Rauschen zunimmt. In den Randgebieten des Panels ist somit der Signal-Rauschabstand nach wie vor sehr klein oder gar null. Das heißt, dass das eigentliche Bild in diesen Bereichen kaum oder gar nicht mehr vom Rauschen unterscheidbar ist. Durch die Verstärkung kann dieses physikalische Gesetz zwar nicht gebrochen werden, jedoch gewährleistet sie in Bereichen, in denen der Signal-Rauschabstand ausreichend ist, bei Ignorierung des Rauschens eine gleichmäßige Helligkeitsverteilung.

3.4 Berührpunkterkennung

Auf Basis der zuvor beschriebenen Vorverarbeitung können nun Berührpunkte, die Inhalt dieses Kapitels sind, und Marker erkannt werden. Dazu muss das korrigierte Bild weiterverarbeitet werden. Abbildung 3.9 zeigt ein Bild der IR-Kamera nach der Vorverarbeitung und bei Berührung mit einer Hand. Die Berührpunkte sind im Kamerabild als helle Flächen erkennbar. Ziel ist nun, die Mittelpunkte dieser Flächen zu finden und deren Koordinaten zu ermitteln.

Abbildung 3.9: Kamerabild nach der Vorverarbeitung bei Berührung mit der Hand.

3.4.1 Bestehende Implementierungen

Hierfür existieren bereits mehrere CPU-basierte Implementierungen. Ein Beispiel ist die Touchlib (vgl. Muller 2008, Kapitel 3). Dort werden Berührpunkte sowie Marker durch Extraktion von Konturen erkannt, aus denen die Mittelpunkte errechnet werden können. Dies geschieht durch Schwellwertberechnung und sequenzielles Umrunden von Objektkanten. Auf demselben Prinzip beruht das, als Open Source verfügbare, Community Core Vision System der NuiGroup, [19].

Eine weitere Möglichkeit besteht darin, das Bild zu segmentieren, das heißt, in Bereiche einzuteilen. Auf diesem Prinzip beruht das bereits vorgestellte reacTIVision System (Bencina u. a. 2005). Die Segmentierung kann beispielsweise mit einem Scanline[20] Algorithmus durchgeführt werden. Die Idee hierbei ist, zunächst in jeder Zeile Liniensegmente, die entweder durch Startpunkt und Länge oder durch Start- und Endpunkt beschrieben werden, zu finden. Diese Liniensegmente werden, falls sie sich in benachbarten Zeilen befinden und sich gegenseitig überlappen, zusammengefasst. Wird dies für alle Zeilen durchgeführt, erhält man für jeden Berührpunkt eine bestimmte Anzahl von Liniensegmenten. Aus diesen kann anschließend der Mittelpunkt des gesamten Segments berechnet werden. Der Ablauf des Algorithmus ist in Abbildung 3.10 veranschaulicht. Dabei können beim sequenziellen Durchlaufen des Bildes beide Verarbeitungsschritte zusammengefasst werden. Die Beschreibung von Liniensegmenten durch Anfangspunkt und Länge wird Lauflängenkodierung (RLE[21]) genannt. Für lange Liniensegmente spart diese Repräsentation als positiver Nebeneffekt Speicherplatz.

Die Bildsegmentierung zur Erkennung von Berührpunkten oder hellen Bereichen, auch Blobs genannt, wurde bereits auf verschiedenen Plattformen parallelisiert. In (Bochem u. a. 2009) geschah dies durch spezielle Hardware, sogenannten FPGAs[22]. Eine weitere Implementierung wurde auf dem, auch in der PlayStation® 3 von Sony eingebauten, Cell Prozessor erfolgreich durchgeführt (Kumar u. a. 2009). Letztere teilt das gesamte Bild in so viele Bereiche auf, wie Prozessorkerne zur Verfügung stehen. Diese müssen in einem Folgeschritt zusammengefasst werden.

Für den Spezialfall eines einzigen Blobs, der zudem kreisförmig ist, existiert eine Implementierung auf Basis der Bildglättung und Differenzbildung (Glebov 2008). Zwar konnte dies von Glebov sehr effizient implementiert werden, jedoch ist die Implementierung für Multi-Touch-Systeme aufgrund ihrer Einschränkungen nicht geeignet.

3.4.2 GPU-Algorithmus

Zur Implementierung der Berührpunkterkennung ist die Bildsegmentierung naheliegend. Dies gestaltet sich jedoch aufgrund der hochgradig parallelisierten Architektur

[19] `http://ccv.nuigroup.com/,Stand:August2010`.
[20] Scanline = Das Bild wird Zeile für Zeile abgearbeitet.
[21] RLE = Run-Length Encoding.
[22] FPGA = Field Programmable Gate Array.

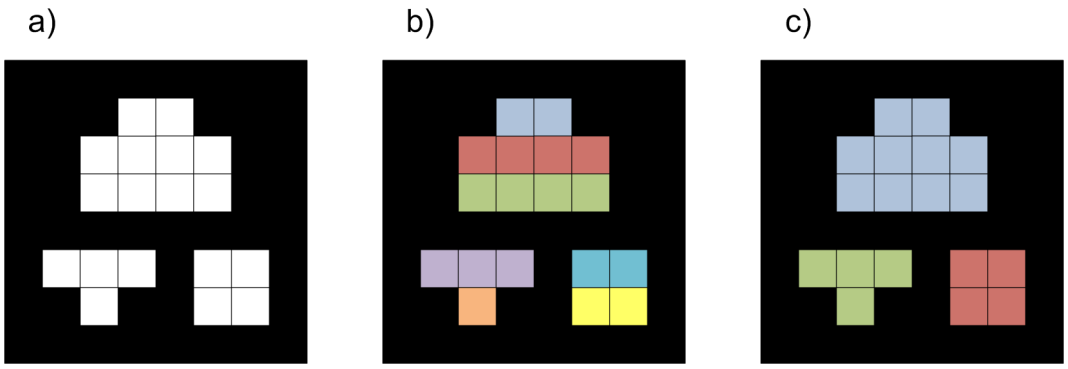


Abbildung 3.10: Berührpunkterkennung durch Segmentierung. a) Ursprungsbild mit drei Berührpunkten. b) Extrahierte Liniensegmente sind farblich gekennzeichnet. c) Resultierende Segmente nach dem Zusammenfassen der Liniensegmente.

der Grafikhardware problematisch. Im Fall des Cell Prozessors genügt die Aufteilung des Bildes in weniger als zehn Bereiche, die zusammengefasst werden müssen. Durch Einsatz von Grafikhardware sind jedoch, je nach Modell, mehrere hundert Rechenkerne verfügbar. Je mehr Rechenkerne eingesetzt werden sollen, desto kleiner werden die Bereiche, die ein einzelner Kern verarbeitet. Somit steigt der Aufwand für die Zusammenfügung dieser Bereiche drastisch an.

Aus diesem Grund wurde ein anderer Algorithmus zur GPU-basierten Erkennung entwickelt. Er nutzt die typisch ovale Form der Berührpunkte aus. Mit Hilfe einer Schwellwertberechnung wird für jeden Punkt überprüft, ob dieser zu einem Berührpunkt gehört. Ist dies der Fall, wird in horizontaler und vertikaler Richtung jeweils auf beiden Seiten entlang einer Geraden gezählt, wie viele Pixel ebenfalls über dem Schwellwert liegen. Die Zählung in eine Richtung wird unterbrochen, sobald ein Pixel unterhalb des Schwellwerts liegt. Ist die Anzahl jeweils in beiden Richtungen gleich oder weicht sie lediglich um einen Pixel ab, so wird angenommen, dass das Pixel im Zentrum der Fläche liegt. Abbildung 3.12 veranschaulicht den Algorithmus anhand zweier Pixel. Das grün umrahmte Pixel liegt im Gegensatz zum rot umrahmten im Zentrum, da dessen Differenzen der Längen in beiden Richtungen null sind.

Ist die Anzahl der Pixel in eine Richtung der Fläche gerade, so liegt in dieser Richtung das Zentrum zwischen zwei Pixeln, die beide als Mitte erkannt werden. Die errechnete Differenz beider Pixel ist somit eins. Tritt dieser Fall auf, so muss sich der Algorithmus für eines der beiden Pixel entscheiden. In der Implementierung wird immer das linke bzw. obere Pixel bevorzugt.

Der implementierte Algorithmus erkennt für die ovalen Berührpunkte den Mittelpunkt korrekt, siehe Abbildung 3.11 a) und c). Die Färbung der Berührpunkte in c) stellt die Nähe zum Mittelpunkt in der jeweiligen Richtung dar. Je intensiver die Farbe, desto näher befindet sich der Punkt am errechneten Mittelpunkt, der selbst gelb dargestellt ist. Die rote Farbkomponente repräsentiert dabei die Nähe in horizontaler Richtung und die grüne Farbkomponente die Nähe in vertikaler Richtung.

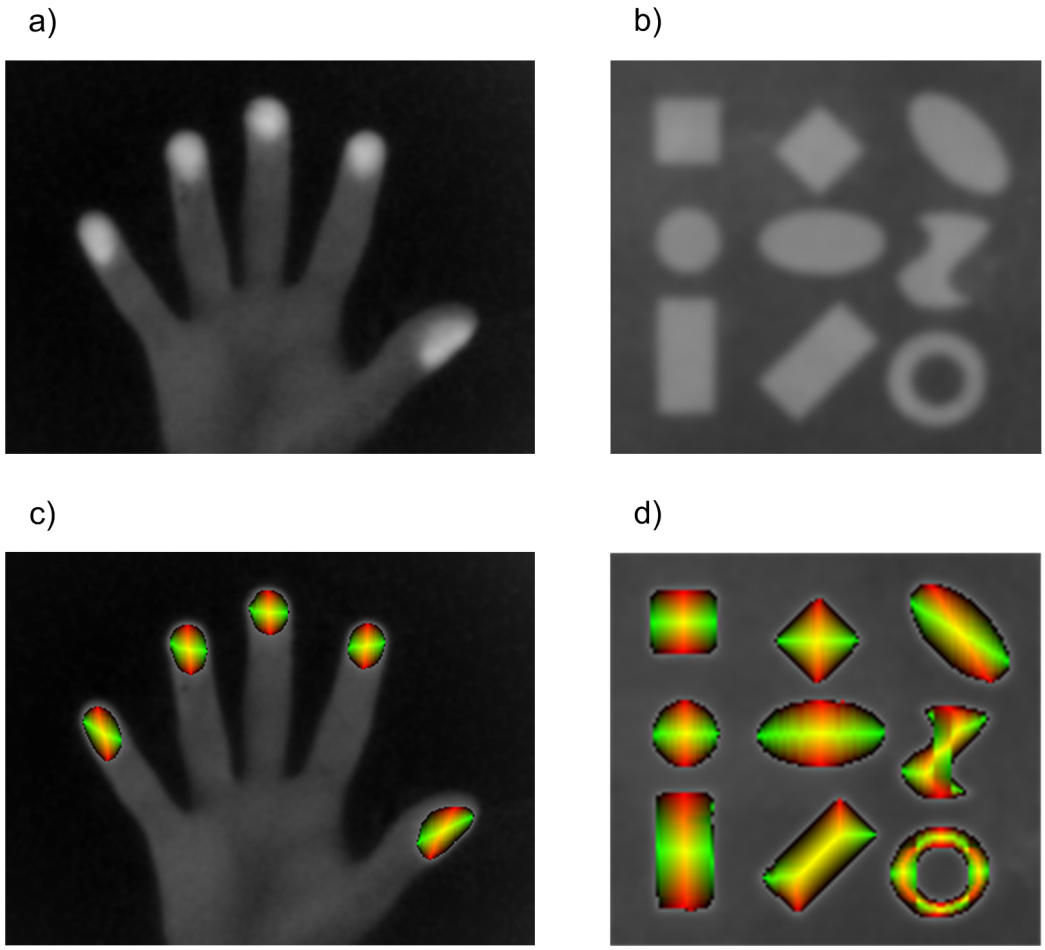
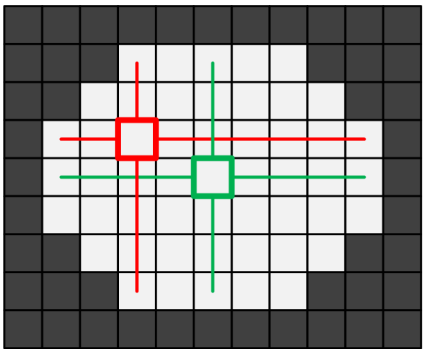


Abbildung 3.11: Berührpunkterkennung am Beispiel einer Hand und eines Testmusters.



	Nachbarpixel		Differenz
	links	rechts	
	2	6	4
	oben	unten	
	2	4	2
	links	rechts	
	4	4	0
	oben	unten	
	3	3	0

Abbildung 3.12: Visualisierung des Erkennungsalgorithmus anhand zweier Punkte.

Die Berechnung der Farbe C erfolgt anhand nachfolgender Gleichung:

$$
\begin{aligned}
C &= \begin{pmatrix} C_{rot} \\ C_{gruen} \\ C_{blau} \end{pmatrix} \\
C_{rot} &= 1 - \frac{|links - rechts|}{links + rechts + 1} \\
C_{gruen} &= 1 - \frac{|oben - unten|}{oben + unten + 1} \\
C_{blau} &= 0
\end{aligned}
\tag{3.3}
$$

Der Algorithmus erkennt die Mittelpunkte ovaler Formen korrekt, was für die Berührpunkterkennung ausreichend ist, jedoch besitzt er Einschränkungen für eine Anwendung im Allgemeinen. In Abbildung 3.11 b) und d) sind verschiedene Formen und deren erkannte Mittelpunkte dargestellt. Darunter befinden sich auch Formen, die nicht korrekt erkannt werden können.

Im Fall der Quadrate, Kreise und Ellipsen funktioniert der Algorithmus und es ist immer ein einziger gelber Mittelpunkt erkennbar. Ist die Form jedoch rechteckig, können Probleme bei der Erkennung auftreten. Rechtecke, die senkrecht oder waagrecht im Bild positioniert sind, werden zwar richtig erkannt, liegt ein Rechteck jedoch diagonal im Bild, bildet eine diagonal verlaufende Linie die erkannte Mitte. Durch Erweiterung des Algorithmus um die Suche in diagonalen Richtungen lassen sich auch beliebig rotierte Rechtecke erkennen. Dies geht jedoch auf Kosten des Rechenaufwands.

Konkave Formen sowie Formen mit Löchern können im Allgemeinen nicht vom Algorithmus erkannt werden. Zudem können Mittelpunkte, die nicht innerhalb der Form liegen, generell nicht erkannt werden.

Die Ergebnisse der Berührpunkterkennung werden in einem Puffer gespeichert. Dazu wird die Koordinate des Mittelpunkts linearisiert und als Index für den Pufferzugriff genutzt. Die Ergebnisse werden dann in Form einer bestimmten Struktur im Puffer abgelegt. Der Grund hierfür ist, dass OpenCL keine Möglichkeit zur Synchronisierung zwischen verschiedenen Arbeitsgruppen vorsieht und durch die Indizierung jedes Ergebnis einen eigenen Bereich im Puffer zugewiesen bekommt. Der

Puffer muss folglich so viele Elemente besitzen, wie das Bild Pixel hat. Derselbe Puffer dient später auch zur Speicherung der Ergebnisse der Markererkennung, da sich in jedem Pixel entweder der Mittelpunkt eines Berührpunkts oder eines Markers befinden kann, jedoch nie beides gleichzeitig.

3.4.3 Performance

Die Mittelpunktsuche lässt sich, im Gegensatz zu den in der Vorverarbeitung angewandten Algorithmen, nicht intuitiv auf SIMD Recheneinheiten abbilden. In der Vorverarbeitung wurden immer dieselben Berechnungen pro Pixel durchgeführt, bei der Mittelpunktsuche wird jedoch für jede Fläche derselbe Algorithmus durchlaufen. Die Berechnung kann jedoch dahingehend nicht parallelisiert werden, da hierfür die einzelnen Flächen durch Segmentierung bekannt sein müssten, um korrekt aufteilen zu können. Da die Segmentierung selbst ein Ansatz zur Mittelpunktsuche ist, bringt dieser Ansatz keinen Vorteil.

Der zuvor beschriebene Algorithmus wurde deshalb pro Pixel parallelisiert. Zugrunde liegt die Annahme, dass ein Großteil des Bildes nicht aus Berührpunkten besteht. Das gesamte Bild wird dabei in Bereiche aufgeteilt, die jeweils immer von einer Arbeitsgruppe, also von einer SIMD-Einheit, verarbeitet wird. Sind alle Pixel in einem Bereich kein Bestandteil einer zu erkennenden Fläche, so bricht der Algorithmus bei der ersten Überprüfung sofort ab. Die komplexe Mittelpunktsuche wird folglich nur in Regionen durchgeführt, die einen Teil einer Fläche beinhalten. Hier dauert die Berechnung für alle Pixel so lange, wie der längste Ausführungspfad benötigt, da alle Rechenkerne einer SIMD-Einheit dieselben Instruktionen ausführen müssen.

Die benötigte Zeit für die Mittelpunktsuche hängt somit maßgeblich von der Größe der Berührpunkte ab, da für deren Pixel jeweils sequenziell überprüft wird, ob sie sich im Mittelpunkt befinden. Treten im Bild sehr große Flächen auf, steigt die Berechnungszeit stark an und kann zum Flaschenhals werden. Zum Finden des Mittelpunkts des Berührpunkts einer Fingerspitze, die eine runde Fläche von 25×25 Pixel einnimmt, benötigt der Algorithmus 0,05 ms. Die Zeit, die zur Überprüfung aller Bereiche, die keine Flächen enthalten, benötigt wird, beträgt 0,07 ms. Eine senkrecht stehende ovale Fläche mit einer Breite von 30 Pixeln und einer Höhe von 90 Pixeln benötigt hingegen 0,5 ms.

Um den Berechnungsaufwand für große Flächen zu reduzieren, ist eine alternative Implementierung möglich. Sie beruht auf der Separierung des Problems. Zuerst werden für jede Zeile die Mittelpunkte horizontaler Liniensegmente heller Pixel ermittelt. Anschließend werden analog dazu die Mittelpunkte vertikaler Liniensegmente berechnet. Liegen die Mittelpunkte jeweils eines horizontalen und vertikalen Liniensegments auf derselben Position, befindet sich dort ein Flächenmittelpunkt. Die Laufzeit der alternativen Implementierung beruht dabei lediglich auf der Größe des Bildes und ist grundsätzlich unabhängig von der Anzahl und Größe der Berührpunkte. Ohne spezielle Optimierungen wurden im Versuch 0,8 ms für die Berechnung der Mittelpunkte gemessen. Damit liegt die Rechenzeit für kleine Berührpunkte deutlich

über der der ursprünglichen Implementierung.

Da die Flächen der Berührpunkte in Multi-Touch-Anwendungen in der Regel relativ klein sind und häufig gar keine Berührungen stattfinden, wurde die erste Implementierung genutzt. Denkbar wäre auch, den Algorithmus anwendungs- und situationsabhängig zu wechseln.

3.5 Markererkennung

Zusätzlich zur Erkennung von Berührpunkten sollten Marker erkannt werden. Die Marker sollen die Möglichkeit bieten, Informationen in ihnen zu speichern, die automatisch ausgelesen werden können. Aus Sicht der Bildverarbeitung existieren auch für die Markererkennung verschiedene Verfahren, die im Folgenden kurz erläutert werden. Anschließend wird der Aufbau der im Rahmen dieser Arbeit eingesetzten Marker und deren GPU-basierte Erkennung erklärt.

3.5.1 Bestehende Implementierungen

Für die Markererkennung bestehen, wie in Abschnitt 2.3 bereits erläutert, verschiedene Systeme. Das reacTIVision System (Bencina u. a. 2005) nutzt die bereits bei der Berührpunkterkennung extrahierten Konturen und erzeugt daraus den Region Adjacency Graph. Dazu sind für die in diesem System eingesetzten Markerarten keine weiteren Bildverarbeitungsschritte notwendig.

Ebenfalls basierend auf der Erkennung von Konturen können auch quadratische Marker erkannt werden (Kato u. Billinghurst 1999). Dazu wird versucht, in jede Kontur vier Liniensegmente einzupassen. Gelingt dies mit ausreichender Genauigkeit, handelt es sich mit hoher Wahrscheinlichkeit um einen Marker. Aus den vier Liniensegmenten werden anschließend die 3D-Koordinaten eines Quadrats mit der Größe des Markers errechnet. Ist die Lage des Markers erkannt, kann er perspektivisch auf eine quadratische Fläche transformiert und sein Inhalt extrahiert werden.

Eine weitere Möglichkeit quadratische Marker zu erkennen beschreibt Hirzer in (Hirzer 2008). Dabei wird ein Bild in Bereiche aufgeteilt und Pixel, die zu Kanten gehören, mit ihren Kantenrichtungen durch Anwendung eines Kantendetektionsfilters erkannt. Anschließend wird versucht innerhalb dieser Bereiche durch zufällige Auswahl zweier Kantenpixel Linien zu finden. Die Kantenrichtung wird in die Suche mit einbezogen. Zusammenhängende Liniensegmente werden dann zu Linien zusammengefasst. Danach werden die Linien entlang ihrer Kantenrichtung auf ihre tatsächliche Länge verlängert. Grund für die Verlängerung ist, dass durch die Aufteilung in Bereiche Linien abgeschnitten werden und sehr kurze Linienenden, die in einem anderen Bereich liegen, durch zufälliges Auswählen von Pixeln, mit geringer Wahrscheinlichkeit erkannt werden.

3.5.2 Marker

Für die Implementierung der GPU-basierten Markererkennung kamen ebenfalls quadratische Marker mit einem schwarzen Rand auf weißem Untergrund zum Einsatz. Jeder Marker beinhaltet eine ID, die als helle und dunkle Quadrate in den Marker binärkodiert wird. Wie auch bei den in den Grundlagen besprochenen Markern teilen sich die entwickelten in einen Bereich zur Feststellung der Drehrichtung und einen Bereich, der die Informationen enthält ein.

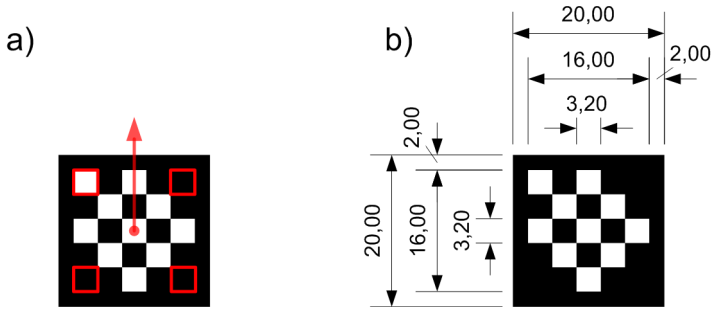


Abbildung 3.13: Entwickelte Marker mit hervorgehobenem Bereich zur Rotationserkennung in a) und Bemaßung in b).

Die für die Erkennung der Drehrichtung zuständigen Bereiche sind in Abbildung 3.13 a) rot umrahmt. Lediglich einer dieser Bereiche darf weiß und die anderen müssen schwarz sein. Der rote Pfeil gibt die Richtung des Markers an und ist immer um 45 Grad im Uhrzeigersinn zur Richtung vom Mittelpunkt zum Zentrum des weißen Bereichs verschoben.

Die restlichen 21 Felder des Markers stellen die binärkodierte Marker-ID dar. Auf diese Weise können insgesamt über zwei Millionen[23] unterschiedliche Marker codiert werden.

Die genauen Ausmaße der Marker sind in Abbildung 3.13 b) dargestellt. Sie sind 20 Millimeter groß und besitzen einen zwei Millimeter dicken, schwarzen Rand. Die Felder im Marker sind jeweils 3,2 Millimeter groß. Werden mehr unterschiedliche Marker benötigt lässt sich dies durch Vergrößerung und eine höhere Anzahl von Feldern erreichen. Die Markergröße und die Anzahl der Informationsfelder können im nachfolgend beschriebenen Erkennungsalgorithmus angegeben werden.

3.5.3 GPU-Algorithmus

Der GPU-Algorithmus setzt wie die Berührpunkterkennung die in Abschnitt 3.3 beschriebene Vorverarbeitung voraus. Die Idee hinter dem Algorithmus beruht auf den folgenden Vereinfachungen:

- Marker werden nur auf einer Ebene, der Bildschirmebene, erkannt.

[23] Die genaue Zahl beträgt: $2^2 1 = 2.097.152$.

- Die Markergröße ist dem Erkennungsalgorithmus bekannt.
- Die schwarzen Ränder der Marker müssen von weißen Flächen umgeben sein. Dies kann auch ein weiterer weißer Rahmen sein.

Durch diese Vereinfachungen erscheinen Marker immer in der gleichen Größe auf dem Bildschirm und werden lediglich auf der Bildebene verschoben oder rotiert. Dadurch betragen die Winkel der Innenkanten immer 90 Grad, folglich stehen die Kantenrichtungen senkrecht aufeinander. Es wird nun versucht mit möglichst wenigen Überprüfungen der Kanten entscheiden zu können, ob sich an der gesuchten Position ein Marker befindet.

Die Markererkennung ist komplizierter als die Berührpunkterkennung und läuft in vier Schritten ab:

1. Kantenerkennung mit Richtungsberechnung.
2. Suche von Markermittelpunkten durch intelligentes Sampling der Ränder.
3. Ermittlung der Markerrichtung und Abtastung des Markerinhalts.
4. Berechnung der Marker-ID.

Bis auf den ersten erfordern alle Schritte einen lokal begrenzten zufälligen Zugriff auf die Bilddaten, weshalb auf sie über den Image-Datentyp zugegriffen wird um den Texturcache zu nutzen. Im Verlauf des Algorithmus werden häufig Bildkoordinaten berechnet, die als Fließkommazahlen vorliegen. Durch Nutzung des Image-Datentyps können diese durch die bereits zuvor erwähnte bilineare Filterung direkt für den Speicherzugriff verwendet werden. Nachfolgend sind die einzelnen Schritte des Algorithmus erläutert.

Kantenerkennung

Die Kantendetektion wird in Kombination mit einer Schwellwertoperation durchgeführt. Dabei spielt die Richtung der Kanten im Gegensatz zu deren Betrag eine entscheidende Rolle. Aus diesem Grund wurde der in Abschnitt 2.2.3 beschriebene von Scharr optimierte Sobel Operator genutzt. Durch die Schwellwertberechnung werden dunkle Pixel, die lediglich Rauschen enthalten, von der Kantenberechnung ausgeschlossen. In Abbildung 3.14 b) ist das Ergebnis der Kantendetektion anhand des Bildes in a) dargestellt. Die Kanten sind dabei farblich kodiert. Zu beachten ist, dass Kanten immer von einem hellen Bildbereich in Richtung eines dunklen Bereichs zeigen.

Sampling

Durch Sampling der Markerränder werden Marker erkannt und deren Mittelpunkte berechnet. Grundsätzlich kommt jedes als Kante erkannte Pixel als Teil eines Markers in Frage. Deshalb wird für jedes Pixel im Kantenbild ein Sampling über die

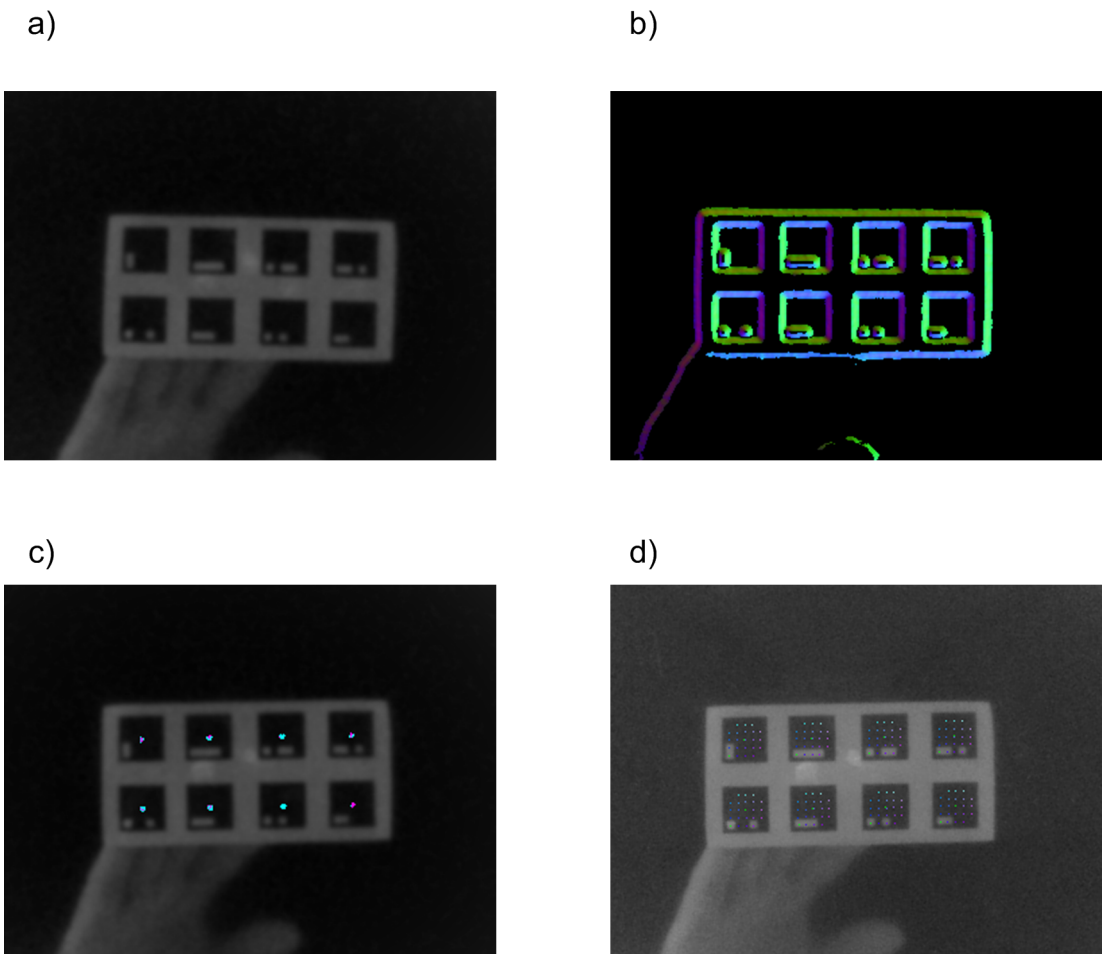


Abbildung 3.14: Ablauf der Markererkennung: a) Ursprungsbild, b) Kantendetektion, c) Mittelpunktsuche, d) Abtasten des Markerinhalts.

restlichen Markerkanten durchgeführt, anstatt in Kantenrichtung Liniensegmente zu erzeugen. Das Sampling geschieht in vier Schritten, deren Ablauf in Abbildung 3.15 a) verdeutlicht wird.

Im ersten Schritt wird in einem kleinen Bereich um das aktuelle Pixel die exakte Kantenrichtung bestimmt. Dazu werden vier Samples entlang der Kantenrichtung ausgewertet. In der Abbildung ist dieses Sampling als rotes Oval gekennzeichnet. Sind alle Samples Kanten, wird das arithmetische Mittel aus der ursprünglichen Kantenrichtung und der Richtung der einzelnen Samples gebildet. Ist dies nicht der Fall, bricht der Algorithmus ab.

Im zweiten und dritten Schritt werden jeweils zwei weitere Samples anhand der Kantenrichtung und deren Normalen genommen. Sie prüfen die an die aktuelle Kantenlinie im rechten Winkel angrenzenden Kantenlinien. Dabei wird zuerst geprüft, ob es sich um eine Kante handelt, und anschließend, ob deren Richtung im rechten Winkel zur momentanen Kantenrichtung steht. Trifft eine dieser Bedingungen für ein Sample nicht zu, wird der Algorithmus für das aktuelle Pixel abgebrochen.

Im letzten Schritt werden erneut zwei Samples genommen, deren Kantenrichtungen müssen jedoch entgegengesetzt zur momentanen Kantenrichtung sein. Erst wenn auch diese Bedingungen erfüllt sind wird davon ausgegangen, dass es sich um einen Marker handelt und das aktuelle Pixel sich in der Mitte einer Randlinie befindet. Über die Position des Randpixels und der exakten Kantenrichtung wird dann der Mittelpunkt des Markers ermittelt.

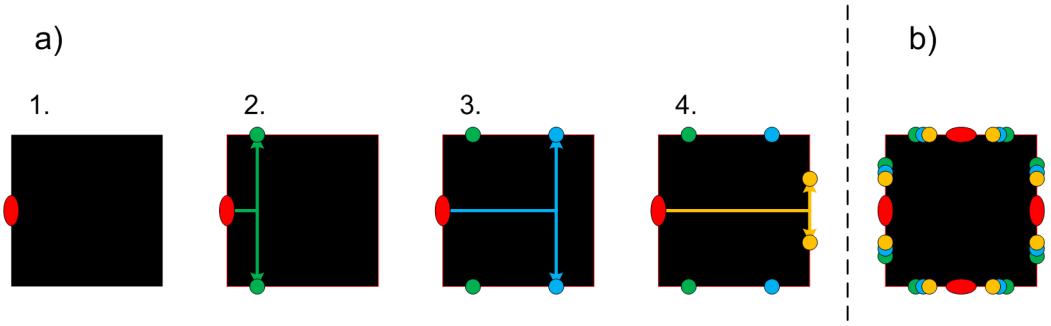


Abbildung 3.15: Sampling der Markerkanten.

Für das gesamte Sampling werden in jedem Schritt Samples in unterschiedlichen Abständen genommen. Dadurch wird, unter Beachtung, dass ein Marker vier Randlinien besitzt und für jede das Sampling durchgeführt wird, ein großes Gebiet des Randes mit Samples abgedeckt, siehe Abbildung 3.15 b). Somit wird auch bei Verdeckung kleiner Bereiche des Randes eine Markererkennung möglich.

In der Regel werden mehrere nahe beieinander liegende Pixel als Mittelpunkte eines Markers erkannt. Dies liegt daran, dass die Kanten breiter als ein Pixel sind. Grund hierfür ist die Rauschunterdrückung und der damit verbundene Weichzeichner bei der Vorverarbeitung, sowie Berechnungsungenauigkeiten. Der Weichzeichner verwischt die Kanten zu einem weichen Übergang über mehrere Pixel hinweg. Dieses Verhalten ist jedoch nicht von Nachteil. Der tatsächliche Mittelpunkt und dessen vorläufige Markerrichtung kann durch Bildung des Durchschnitts aller erkannten Markermittelpunkte in einem kleinen Bereich genau bestimmt werden.

Dabei muss jedoch beachtet werden, dass die Richtungsvektoren einzelner Mittelpunkte senkrecht oder in gegensätzlicher Richtung zueinander stehen können. Vor der Mittelung werden sie deshalb durch Drehung um 90, bzw. 180 Grad vereinheitlicht. Nach der Drehung darf jeder Richtungsvektor lediglich in den ersten Quadranten eines kartesischen Koordinatensystems zeigen. Liegen die Markerrichtungen fast senkrecht oder waagrecht in der Bildebene, können einzelne Richtungsvektoren unterschiedlicher Kanten immer noch beinahe senkrecht zueinander stehen. In diesem Fall ist eine erneute Drehung einzelner Vektoren notwendig. Die Markermittelpunkte und deren vereinheitlichte Kantenrichtungen sind in Abbildung 3.14 c) zu sehen, die Mittelpunkte sind dabei abhängig von der Kantenrichtung eingefärbt. Die zweite Richtungskorrektur für horizontale und vertikale Richtungsvektoren wurde noch nicht durchgeführt.

Markerrichtung und Markerinhalt

Ist die exakte Mitte eines Markers gefunden, wird die tatsächliche Rotationsrichtung des Markers ermittelt. Dazu werden die Koordinaten der für die Rotationserkennung zuständigen Bereiche bestimmt. Anschließend wird die Rotationsrichtung des Markers anhand des weißen Bereichs eindeutig ermittelt. Ist diese Richtung bekannt, werden die Markerinformationen der Reihe nach ausgelesen, dabei wird der restliche Markerinhalt in einem Raster abgetastet, siehe Abbildung 3.14 d). Der Mittelwert zwischen dem weißen Bereich der Rotationserkennung und den schwarzen Bereichen wird als Schwellwert für die Entscheidung, ob ein Bereich innerhalb des Markers hell oder dunkel ist, genutzt. Somit ist das Auslesen der Markerinformationen im Vergleich zur Nutzung eines konstanten Schwellwerts weniger anfällig gegenüber Helligkeitsschwankungen.

Marker-ID

In der derzeitigen Implementierung sind die Markerwerte binärkodiert, das bedeutet, dass jeder Bildpunkt im abgetasteten Raster genau ein Bit der Marker-ID repräsentiert. Da die Reihenfolge der Abtastung immer dieselbe ist, können die Bits der Marker-ID nach aufsteigender Signifikanz gesetzt werden. Die Position und Richtung werden anschließend zusammen mit der berechneten Marker-ID in demselben Puffer gespeichert wie zuvor bei der Berührpunkterkennung.

3.5.4 Performance

Auch die Markererkennung wurde auf der Grafikkarte pro Pixel parallelisiert. Da Kanten nur in bestimmten Bereichen im Bild auftreten und nur dort das Sampling durchgeführt wird, kann ein großer Bereich vorab von der Berechnung ausgeschlossen werden. Auch in Bereichen, in denen Kanten vorhanden sind, führen die ersten Samples in der Regel zum Abbruch der weiteren Erkennung. Zu diesen Bereichen gehören Berührpunkte, deren Umrisse als Kanten erkannt werden. Hier spielt die Überprüfung der Kantenrichtung eine entscheidende Rolle. Da die Marker schwarz sind, zeigen deren Kanten zum Markermittelpunkt. Die Kanten der Berührpunkte hingegen zeigen, da sie heller als ihre Umgebung sind, vom Berührpunkt weg. Viele Sampleauswertungen sind durch die zusätzliche Überprüfung der Kantenrichtungen folglich nur in den vier Bereichen, in denen sich die Kantenmittelpunkte der Marker befinden, notwendig. Somit haben Berührpunkte lediglich einen kleinen Einfluss auf die Markererkennung. Siehe hierzu Abbildung 4.2 am Beispiel von fünf Berührpunkten.

Die Zeit für die Überprüfung aller Pixel ohne das Vorhandensein von Kanten beträgt wie auch die Überprüfung von Berührpunkten 0,07 ms. Die benötigte Zeit pro Marker hängt von dessen Ausrichtung ab und beträgt zwischen 0,05 und 0,1 ms. Der Grund für diese Abhängigkeit ist der konstante Schwellwert der Kantendetektion. Durch die optimierten Filterparameter werden zwar korrekte Richtungen errechnet, die Intensität der Kanten weicht jedoch vom genauen Wert, abhängig

von der Kantenrichtung, ab. Somit werden für Kanten, die diagonal verlaufen, mehr Kantenpixel, die über dem Schwellwert liegen, erkannt. Ein weiterer Einflussfaktor ist das Zusammenfassen von gefundenen Mittelpunkten. Da für jeden Berührpunkt ein Bereich um jenen untersucht werden muss, ist diese Operation zeitaufwändig.

3.6 Zurücklesen der Ergebnisse

Nachdem sowohl die Berührpunkte als auch die Marker vom System erkannt wurden, müssen die im Puffer abgelegten Ergebnisse zurück in den Arbeitsspeicher des Hosts übertragen werden. Da der Puffer sehr groß ist und die Informationen zudem auf den gesamten Puffer verteilt sind, ist das Kopieren des Puffers nicht zielführend. Aus diesem Grund werden die gespeicherten Ergebnisse zuerst zusammengefasst. Hierfür werden, mangels Synchronisierungsmöglichkeiten zwischen den Arbeitsgruppen, insgesamt drei Kernel benötigt.

Der erste Kernel fasst die verstreuten Ergebnisse jeder Spalte in einen Bereich am Anfang der Spalte zusammen und zählt dabei, wie viele Ergebnisse sich in der Spalte befinden. Der zweite Kernel berechnet nun über die Anzahl aller sich in den einzelnen Spalten befindenden Ergebnisse die Präfixsumme[24] (Blelloch 1993). Diese liefert für jede Spalte einen eindeutigen Index für deren erstes Element, sodass sich die Ergebnisse nicht mit denen anderer Spalten überlappen und dennoch kein Zwischenraum zwischen ihnen frei bleibt. Der dritte Kernel kopiert anschließend die Ergebnisse der Spalten mit Hilfe der Präfixsummen in einen weiteren Speicherpuffer. Dieser enthält nun alle Ergebnisse in kompakter Form und kann in den Arbeitsspeicher des Hosts übertragen werden.

3.7 Applikation

Zur Visualisierung und zum Debugging der zuvor beschriebenen Algorithmen wurde die in Abbildung 3.16 dargestellte Applikation entwickelt. Sie ist sehr einfach aufgebaut und teilt sich in drei Bereiche auf:

1. Steuerung der Applikation und Ergebnisvisualisierung.
2. Ausgabe von Geschwindigkeitsmessungen.
3. Debugausgabe mit Darstellung der Zwischenergebnisse.

Steuerung und Ergebnisvisualisierung

Die Steuerung der Applikation, in der Abbildung mit Nummer eins gekennzeichnet, besteht lediglich aus drei Schaltflächen. Mit Hilfe der Start-Schaltfläche wird die Berührpunkterkennung aktiviert. Erkannte Berührpunkte werden im darüberliegenden

[24] Die Präfixsumme akkumuliert für jedes Element einer geordneten Liste die Werte aller Vorgänger.

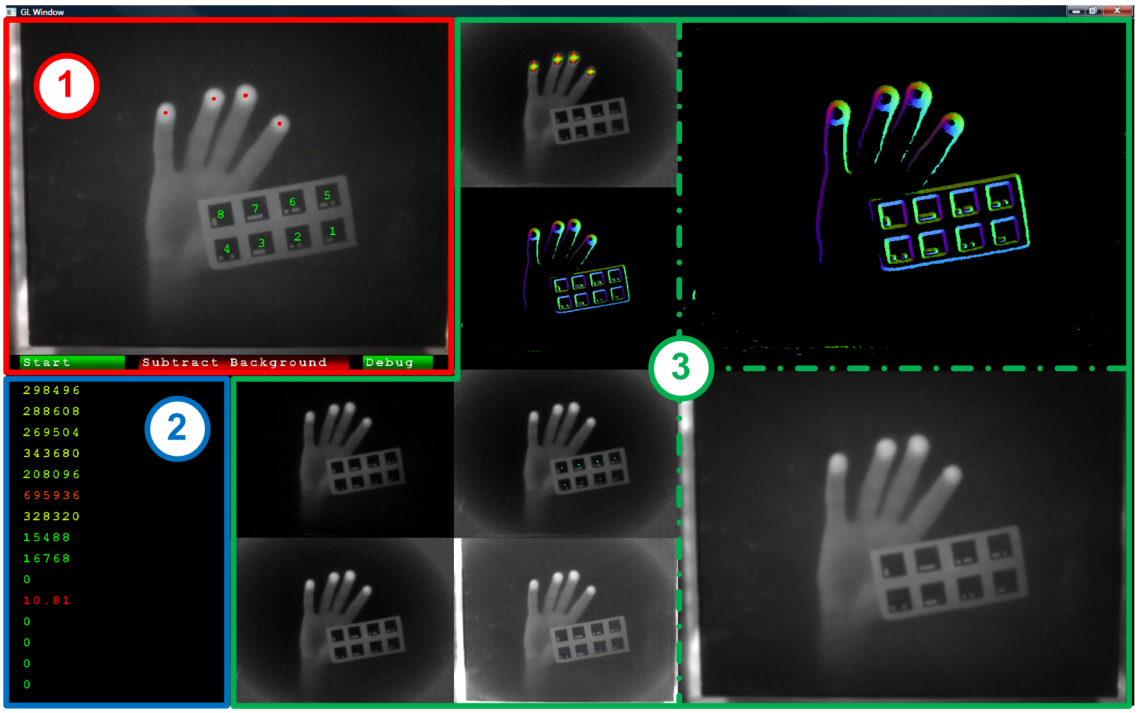


Abbildung 3.16: Testapplikation zur Visualisierung der Algorithmen. Die drei Hauptbereiche sind farblich gekennzeichnet und nummeriert.

Fenster als rote Punkte und Marker durch Darstellung der Marker-ID gekennzeichnet. Die nächste Schaltfläche setzt das Bild für die Hintergrundentfernung auf das zum Zeitpunkt des Klicks aktuell dargestellte Bild. Zum Zeitpunkt des Klicks sollten dabei keine Berührungen mit dem Bildschirm stattfinden, da die Erkennung sonst in diesen Bereichen gestört würde. Die letzte Schaltfläche aktiviert oder deaktiviert den Debugbereich, der noch genauer beschrieben wird.

Ausgabe von Geschwindigkeitsmessungen

Der in der Abbildung mit Nummer zwei gekennzeichnete Bereich ist für Geschwindigkeitsmessungen relevant. Die oberen Zahlen geben die Zeit an, die für die Ausführung verschiedener Kernel benötigt wird. Die Zeiten im oberen Bereich sind in Nanosekunden angegeben. Die Namen der Kernel werden momentan nicht angezeigt, da sie sich während der Entwicklung häufig änderten. Die Zeiten sind jedoch nach der Reihenfolge der entsprechenden Kernelausführung geordnet. Der mit etwas Abstand weiter unten dargestellte Wert ist die Mittelung der gesamten Marker- und Berührpunkterkennung inklusive der Übertragung der Videodaten zu und der Ergebnisse von der Grafikkarte. Angegeben ist dieser Wert in Millisekunden.

Alle Ausgaben sind zudem mit einem linear interpolierten Farbbereich von grün nach rot codiert. Der Bereich für die Kernelzeiten erstreckt sich von 0 ns (grün) bis 800.000 ns (rot). Für die Farbcodierung der Gesamtdauer erstreckt er sich von 1 ms bis 10 ms.

Debugausgabe

In der Debugausgabe, die durch die Nummer drei gekennzeichnet ist, können Zwischenergebnisse angezeigt werden. Alle verfügbaren Zwischenergebnisse sind im linken Teil des Bereichs als Vorschau dargestellt. Durch Klick auf eine Vorschau wird das entsprechende Zwischenergebnis im rechten oberen Bereich in voller Auflösung dargestellt. Zum Vergleich ist im rechten unteren Bereich das Eingangsbild nach der Binomialfilterung dargestellt. So kann beispielsweise überprüft werden, ob errechnete Koordinaten der Visualisierung plausibel sind.

Für das Finden von Fehlern ist dieser Bereich besonders wichtig, da die Ergebnisse der Kernel nicht in einem Debugger angezeigt werden konnten.

Kapitel 4

Evaluation

Zum Vergleich der GPU-basierten Implementierung mit verschiedenen bestehenden CPU-basierten Implementierungen wurden Geschwindigkeitsmessungen durchgeführt und ausgewertet. Im Anschluss wird der im Rahmen der Arbeit entwickelte Prototyp analysiert. Sowohl für die Software als auch für die Hardware werden Möglichkeiten zur Optimierung und Erweiterung diskutiert.

4.1 Performanceanalyse

Die Messungen für die Geschwindigkeitsanalyse erfolgten mit verschiedenen Anzahlen von Berührpunkten oder Markern. Dazu wurden für jede Anzahl 21 Messungen, die jeweils das Mittel über 100 Durchläufe bilden, durchgeführt.

Berührpunkterkennung

In Abbildung 4.1 sind die Ergebnisse dieser Messungen sowie zum Vergleich Ergebnisse von Messungen bestehender Systeme (Muller 2008; Kaltenbrunner 2009) dargestellt. Die Ergebnisse der im Rahmen dieser Arbeit entstandenen Implementierung sind mit GPU gekennzeichnet.

Aus der Grafik ist ersichtlich, dass die Berechnungen der GPU-Implementierung auch ohne erkannte Berührpunkte circa 5 ms benötigen, jedoch mit Zunahme der Berührpunkte kaum ansteigen. Zu beobachten ist eine Rückläufigkeit der Berechnungszeit von fünf zu zehn Berührpunkten. Bei genauerer Analyse der Messwerte wurde festgestellt, dass die Berechnungszeit der einzelnen Kernel jedoch stetig ansteigt. Deshalb besteht Grund zur Annahme, dass die Schwankungen der Gesamtberechnungszeit auf unterschiedliche Scheduling-Entscheidungen des Betriebssystems zurückzuführen sind.

Vergleicht man die Ergebnisse mit denen der Touchlib, die häufig eingesetzt wird, ist eine deutliche Geschwindigkeitssteigerung durch die Implementierung auf der GPU erkennbar. Einen Großteil der Berechnungszeit der Touchlib, ca. 20 ms, benötigt die Korrektur der Bildverzerrung, die durch das Objektiv der Kamera auftritt

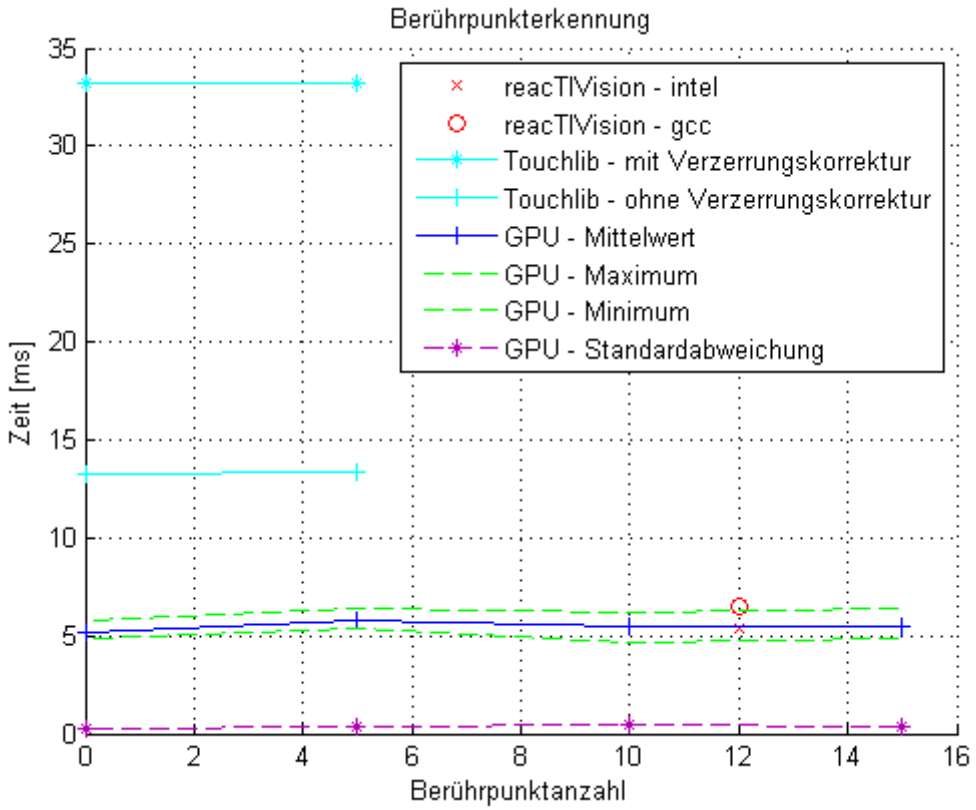


Abbildung 4.1: Geschwindigkeitsvergleich der Berührpunkterkennung.

(siehe hierzu Bourke 2002). Aus diesem Grund wurden die Messwerte der Touchlib einmal mit Korrektur und einmal ohne dargestellt.

Die Verzerrung tritt insbesondere bei Weitwinkelobjektiven auf. Da die im Prototyp eingebaute Kamera keines besitzt, wurde für die Messungen auch keine Korrektur durchgeführt. Eine einfache Implementierung dieser Korrektur benötigt auf der eingesetzten Grafikkarte 0,3 ms und ist somit kein entscheidender Faktor für die Gesamtgeschwindigkeit des Systems.

Vergleicht man die GPU-Implementierung mit der reacTIVision Implementierung, ist kein wesentlicher Geschwindigkeitsunterschied erkennbar. Abhängig vom eingesetzten Compiler, in der Darstellung mit "intel" bzw. "gcc" gekennzeichnet, ist die reactiVISion Implementierung gleich schnell oder minimal langsamer. Grund hierfür ist deren Implementierung. Im Wesentlichen besteht sie lediglich aus einer Schwellwertberechnung und anschließender Segmentierung des daraus resultierenden Binärbilds. Somit werden keine zeitaufwändigen Filteroperationen auf den Bilddaten durchgeführt. Besitzen die Bilder der Kamera einen hohen Rauschanteil oder muss eine Bildentzerrung durchgeführt werden, so steigt der Rechenaufwand der reactiVISion Implementierung enorm.

Zusammenfassend lässt sich sagen, dass die Implementierung der Berührpunkterkennung primär von der Beschleunigung der Vorverarbeitungsschritte profitiert

und auch bei vielen Berührpunkten kaum Geschwindigkeit einbüßt.

Markererkennung

Für die Markererkennung wurden ebenfalls Messungen durchgeführt, welche mit den Ergebnissen verschiedener Systeme verglichen wurden (Hirzer 2008; Wagner 2007; Kaltenbrunner 2009). Die Ergebnisse sind in Abbildung 4.2 dargestellt.

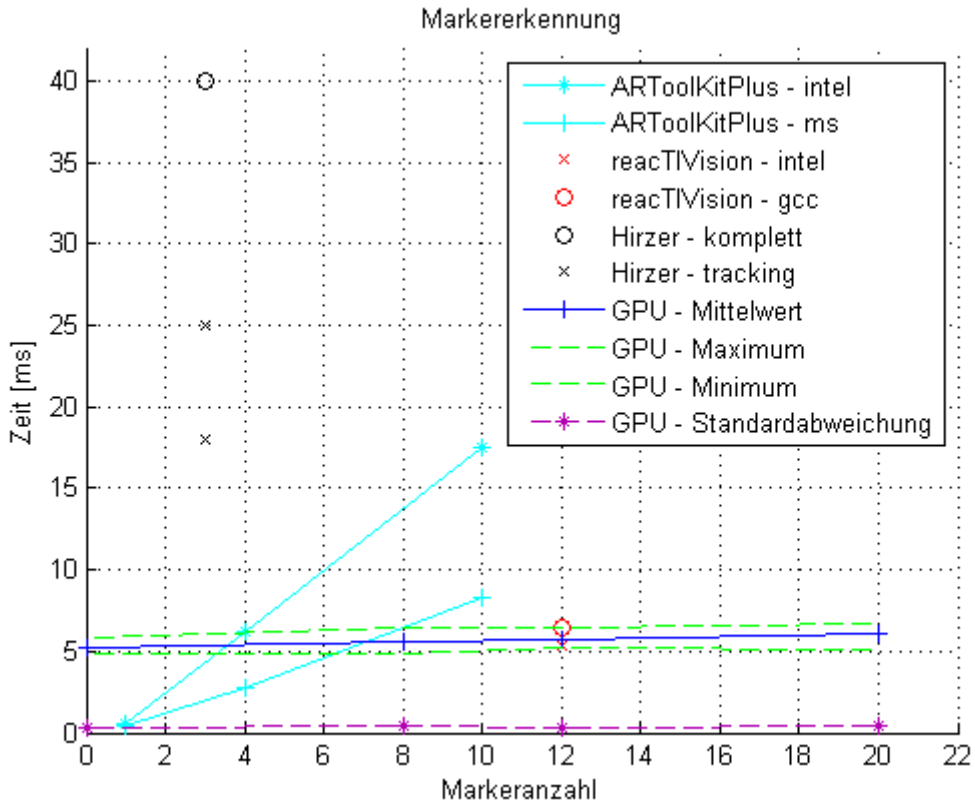


Abbildung 4.2: Geschwindigkeitsvergleich der Markererkennung.

Für die GPU-Implementierung zeigt sich ein ähnliches Bild zur Berührpunkterkennung. Da die Vorverarbeitungsschritte gleich geblieben sind, werden auch hier 5 ms ohne erkannten Marker benötigt. Der Anstieg der Berechnungszeit verläuft ebenfalls ähnlich. Auch hier wurde ein Vergleich mit dem reacTIVision System durchgeführt. Erwartungsgemäß mit demselben Ergebnis. Anders verhält sich dies im Vergleich zu Implementierungen, die quadratische Marker nutzen.

Die Implementierung von Hirzer liegt schon bei drei Markern deutlich hinter der GPU-basierten. Für die Erkennung eines kompletten Bildes mit drei Markern werden 40 ms benötigt. Zur Reduzierung des Rechenaufwands nutzt diese Implementierung Ergebnisse vergangener Bilder und beschränkt die Suche auf Bereiche in der Umgebung zuvor erkannter Marker. Auf diese Weise sinkt der Rechenaufwand auf durchschnittlich zwischen 18 ms und 25 ms, abhängig von der Markerlage. Dies

ist im Vergleich zur GPU-basierten Implementierung dennoch deutlich langsamer. Ein weiteres Problem der Hirzer-Implementierung ist die Erkennung der Marker aus einzelnen Linien. Dabei wird jede Linie mit jeder anderen Linie auf Zusammengehörigkeit überprüft. Für viele Marker bedeutet dies, dass eine starke Zunahme des Rechenaufwands zu erwarten ist.

Als weitere Vergleichsimplementierung dient das ARToolKitPlus. Dieses System wurde für den mobilen Einsatz konzipiert, ist jedoch auch für PCs implementiert. Beim Tracking eines einzigen Markers ist das System sehr schnell. Die Software wurde in diesem Fall auf die Erkennung lediglich eines Markers eingestellt. Ebenfalls abhängig vom eingesetzten Compiler steigt die Berechnungszeit drastisch an und ist bei neun Markern langsamer als die GPU-Implementierung. Zudem wurden die Messungen mit einer Auflösung von 320×240 Pixeln durchgeführt. Dies entspricht einem viertel der Bildfläche, die die PlayStation® Eye-Kamera liefert.

Die Resultate des Vergleichs bei der Markererkennung sind, dass die GPU-Implementierung insbesondere bei steigender Markerzahl deutliche Geschwindigkeitsvorteile bringt. Zudem ist die Beschränkung der Markererkennung auf die Bildebene ein weiterer Vorteil für die Implementierung.

Rechenzeitverteilung

Um mögliche Flaschenhälse in der Rechenzeit der Implementierung erkennen zu können, wurde die Aufteilung der Gesamtzeit der Erkennung auf die einzelnen OpenCL-Kernel ermittelt. Abbildung 4.3 zeigt drei typische Verteilungen. Dargestellt sind lediglich reine Kernelzeiten, ohne die für die Übertragung der Daten benötigte Zeit.

Die ersten vier Kernel, `binomial_v`, `binomial_h`, `subtract` und `amplify`, sind für die Vorverarbeitung zuständig. Da sie immer auf dem kompletten Bild arbeiten, benötigen sie stets die gleiche Rechenzeit. Der Kernel `detect` dient zur Erkennung der Berührpunkte. Seine Ausführungszeit verlängert sich, sobald Berührpunkte im Bild sichtbar werden. Angezeigte Marker haben keinen Einfluss auf diesen Kernel. Anschließend folgt die Markererkennung. Sie unterteilt sich im Wesentlichen in die Kantenerkennung, `scharr`, das Finden der Markermittelpunkte, `fiducial`, und die Zusammenfassung der Mittelpunkte mit anschließender Extraktion der Marker-ID, `merge`. Die Verstärkung wird hier nochmals aufgerufen um das Originalbild zu verstärken. Dies ist für das Auslesen der Marker-ID notwendig, da der Binomialfilter auch den Inhalt des Markers verwischt. Der `clearImg` Kernel dient lediglich zum Löschen des Ergebnisses des `fiducial` Kernel. Aus der Grafik ist erkennbar, dass der Großteil der Zeit der Markererkennung für das Zusammenfassen der Mittelpunkte im `merge` Kernel benötigt wird. Im letzten Schritt werden die Ergebnisse beider Algorithmen zusammengfasst. Hierfür sind die Kernel `compact1`, `compact2` und `compact3` zuständig. Sie benötigen ebenfalls nahezu konstante Zeit.

Ein Großteil der Gesamtzeit der Erkennung, circa 2 ms, wird für die Vorverarbeitung und Kantendetektion benötigt. Da die Zugriffsmuster für diese Kernel bekannt sind, kann durch manuelle Optimierung des Speicherzugriffs die Ausführungszeit

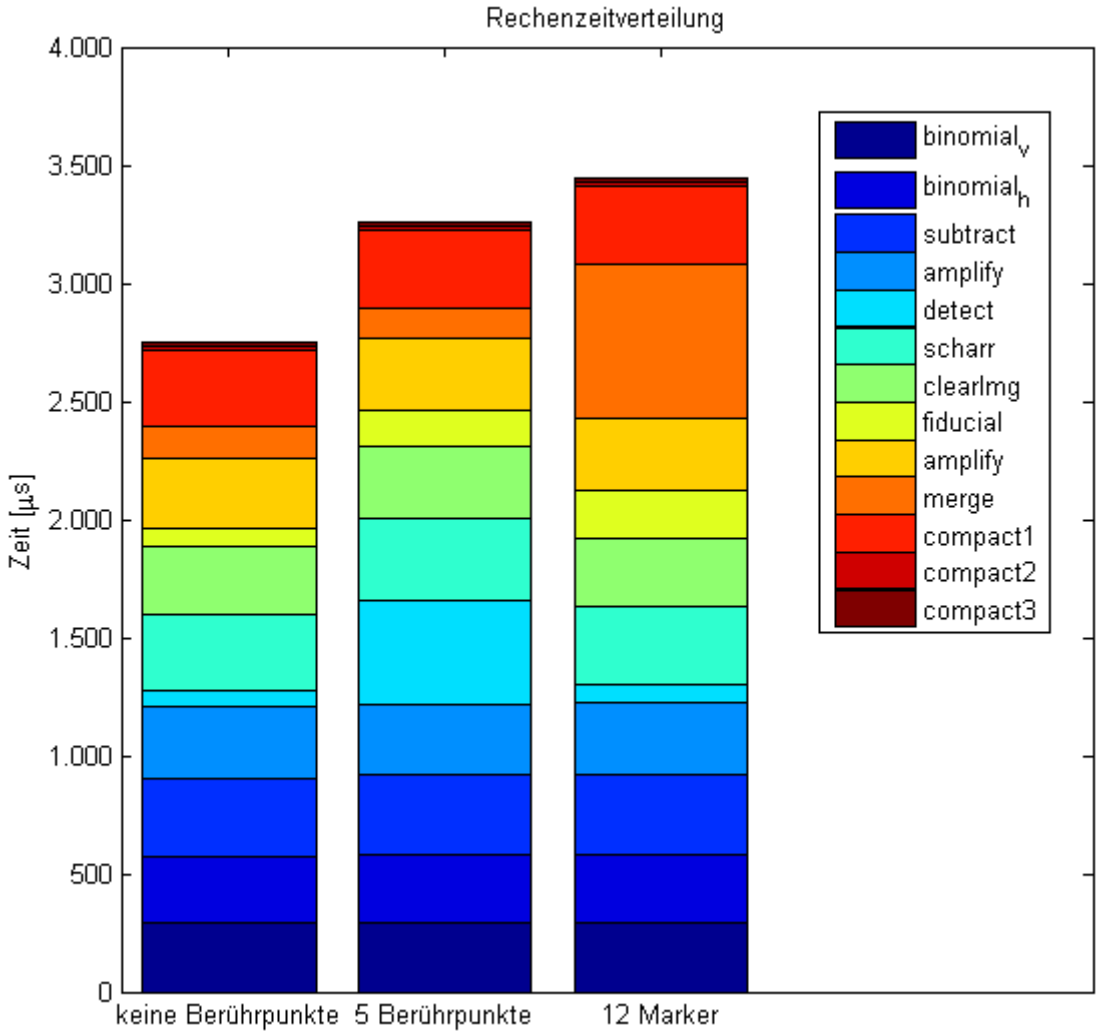


Abbildung 4.3: Rechenzeitverteilung der OpenCL Kernel in drei verschiedenen Situationen.

verkürzt werden. Dazu müssen die derzeit verwendeten Images durch gewöhnliche OpenCL Buffer ersetzt werden. Hierbei kann auch überprüft werden, inwiefern die alternative Implementierung der Berührpunkterkennung mit konstanter Laufzeit beschleunigt werden kann.

Die Optimierung der Kernel erfordert zwar weitere Entwicklungszeit, ermöglicht jedoch auch eine Optimierung der Datenstrukturen, um beispielsweise Grafikspeicher einzusparen oder die Genauigkeit mancher Berechnungen zu erhöhen[1].

4.2 Prototyp

Der Fokus bei der Entwicklung des Prototyps wurde auf die Möglichkeit, die Implementierung der Berührpunkt- und Markererkennung zu testen, gelegt. Zudem wurde untersucht, ob optische Multi-Touch-Systeme auch in einem kleinen Maßstab in Kombination mit LC-Panels gebaut werden können. Dies ist durch den Prototyp sehr gut gelungen.

Durch den geringen Abstand der Kamera vom LC-Panel gleicht die Bauweise einem gewöhnlichen Röhrenmonitor. Somit ist der Monitor transportabel und benötigt wenig Platz. Zudem besitzt der Bildschirm im Verhältnis zu seiner Größe eine hohe Auflösung. Hauptvorteil ist dabei die gut lesbare Darstellung von kleinem Text. Dies ist bei großen Multi-Touch-Tischen meist nicht der Fall.

Die Möglichkeit, sowohl Berührpunkte als auch Marker zu erkennen, bietet ein großes Spektrum unterschiedlicher Anwendungen. Sind die Marker an Objekten angebracht, können Zusatzinformationen zu den Objekten am Bildschirm angezeigt werden. In einer Bar kann beispielsweise durch Marker am Boden des Glases eines Drinks ein Video passend zur Herkunft des Rezepts eingeblendet werden. Vorausgesetzt der Bildschirm wird waagerecht in die Tischplatte eingelassen.

In weiteren Versuchen muss geklärt werden, ob die Hintergrundbeleuchtung zu dunkel ist. Hierzu muss der Bereich zwischen den Teleskopstangen verschlossen werden, um ein Austreten des Lichts hinter der Bildfläche zu vermeiden. Dies hat zudem den Vorteil, dass kein infrarotes Licht seitlich durch den Bildschirm in die Kamera strahlen kann. Die Infrarotbeleuchtung kann zudem durch weitere IR-LEDs verstärkt werden. Dadurch kann die Erkennung der Marker in Randbereichen verbessert werden.

Im derzeitigen Aufbau werden Berührpunkte hauptsächlich durch Ausnutzung des FTIR-Effekts erkannt. Das Licht, das bei der Berührung mit den Fingern zur Kamera strahlt, wird dadurch zusätzlich verstärkt. Dies ist insbesondere der Fall, wenn die Fingerkuppen feucht sind. Durch die Flüssigkeit entsteht eine gute optische Verbindung zwischen dem Plexiglas® und der Fingerkuppe. Dieser Effekt wird in der derzeitigen Implementierung zur einfachen Unterscheidung von Berührpunkten und Markern genutzt. Nach mehreren Versuchen mit unterschiedlichen Testpersonen

[1] Durch die Speicherung von Fließkommazahlen in Farbkomponenten können Genauigkeitsverluste auftreten.

wurde jedoch festgestellt, dass sich der für eine Erkennung benötigte Druck zwischen den Testpersonen stark unterscheidet.

Zur Verbesserung der Berührungserkennung besteht die Möglichkeit, die Reihenfolge der Plexiglas®-Platten zu ändern. In Versuchen muss geklärt werden, ob durch die Platzierung des LC-Panels an vorderster Stelle eine Verbesserung erzielt werden kann. Dann befindet sich die Plexiglas®-Platte, die mit den IR-LEDs beleuchtet wird, an zweiter Stelle. Zwischen ihr und dem LC-Panel muss dann eine Diffusorfolie platziert werden. Dies wäre dann ein DSI-Aufbau (siehe NUI Group Authors 2009, Seite 17), mit dem Unterschied, dass sich das LC-Panel zwischen dem Benutzer und der Plexiglas®-Platte befindet. Zu untersuchen ist, ob in diesem Aufbau trotz Diffusorfolie Marker erkannt werden können.

Kapitel 5

Fazit

In dieser Arbeit konnte gezeigt werden, dass Bilderkennungs- und Verarbeitungsalgorithmen, die für optische Multi-Touch-Verfahren eingesetzt werden, durch ihre Berechnung auf moderner Grafikhardware beschleunigt werden können.

Hierfür wurde der Prototyp eines Multi-Touch-Bildschirms auf Basis eines 17 Zoll LCDs und einer Kamera entwickelt, welcher sowohl für die Erkennung von Berührpunkten als auch von Markern eingesetzt werden kann. Er zeichnet sich durch eine im Vergleich zu großen Multi-Touch-Tischen kompakte Bauform und die daraus resultierende Transportabilität aus. Der Prototyp bildet eine solide Grundlage für zukünftige Projekte, da sein Aufbau modifiziert und erweitert werden kann.

Mit Hilfe von OpenCL wurde eine portable Implementierung für die Berührpunkt- und Markererkennung entwickelt. Dabei führten die SIMD-Einheiten moderner Grafikkarten besonders in der Vorverarbeitung zu einer Geschwindigkeitssteigerung gegenüber vergleichbaren CPU-Implementierungen. Durch geschicktes Algorithmendesign und diverse Vereinfachungen konnte auch die Erkennung der Berührpunkte und Marker effizient implementiert werden. Somit kann die Bildverarbeitung von der Bildvorverarbeitung bis zur Extraktion der Daten auf Grafikhardware durchgeführt werden.

Das Ziel, nicht länger als zehn Millisekunden sowohl für die Berührpunkterkennung als auch für die Markererkennung zu benötigen, wurde mit einer durchschnittlichen Verarbeitungszeit unter sechs Millisekunden übertroffen, was einer Erkennungsrate von 167 Bildern pro Sekunde entspricht.

5.1 Ausblick

Zum Zeitpunkt dieser Arbeit gestaltete sich die Entwicklung für Grafikhardware im Vergleich zur Entwicklung für gewöhnliche CPUs deutlich schwieriger. Die Hauptgründe hierfür waren der Mangel einer Entwicklungsumgebung, die OpenCL unterstützt, und die architekturbedingten Einschränkungen der Grafikhardware.

Ersteres wird sich in naher Zukunft durch das Erscheinen verschiedener Entwicklungsumgebungen mit Syntaxunterstützung, integriertem Profiling und der Möglichkeit, laufende Kernel zu debuggen, beheben. NVIDIA nimmt in diesem Feld mit der

kürzlich erschienenen Nexus Entwicklungsumgebung eine Vorreiterrolle ein.

Für die weitere Entwicklung der Grafikkartenarchitekturen ist ein klarer Trend in Richtung universeller Rechenkarten erkennbar. Bereits aktuelle Grafikkarten besitzen deutlich weniger Einschränkungen, insbesondere beim Speicherzugriff, als dies noch in der für die Implementierung genutzten G80-Architektur der Fall war.

Abschließend kann gesagt werden, dass die derzeitige GPU-basierte Implementierung bereits eine Geschwindigkeitssteigerung gegenüber CPU-basierten Verfahren bringt, jedoch insbesondere durch das Erscheinen verbesserter GPU-Architekturen und Entwicklungswerkzeuge viel Raum für Optimierungen und Erweiterungen geschaffen wird.

Anhang A

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Master Thesis selbständig angefertigt habe. Es wurden nur die in der Arbeit ausdrücklich benannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut habe ich als solches kenntlich gemacht.

Ort,Datum	Unterschrift

Abbildungsverzeichnis

2.1 Schematischer Aufbau des Bildschirmprototyps. 6
2.2 LC-Panel und Teile der ursprünglichen Hintergrundbeleuchtung. . . . 7
2.3 Schaltplan für weiße LEDs und infrarote LEDs. 7
2.4 Aufnahmen des LC-Panels mit der modifizierten Kamera. 9
2.5 Schrägansicht des entwickelten Prototyps und Netzteils. 11
2.6 Elektronik zur Ansteuerung des LC-Panels. 11
2.7 Darstellung einer einfachen eindimensionalen Faltung. 13
2.8 Glättungsfilter 16
2.9 Minimale Boxfilter 18
2.10 Einfache Ableitungsfilter in horizontaler Richtung. 21
2.11 Sobel-Operator 23
2.12 Code-128-codierter Strichcode des Worts Multi-Touch 25
2.13 Aztec-Strichcode. 26
2.14 Unterschiedliche AR-Marker. 27
2.15 Marker des reacTIVision Systems 28
2.16 CUDA Architektur des G80 Chips von NVIDIA 30

3.1 OpenCL Architekturmodelle 37
3.2 Stark vereinfachte OpenGL Architektur. 44
3.3 UML-Komponentenhierarchie 46
3.4 Double Buffering 49
3.5 Bitmap-Font für die Darstellung des gesamten ASCII Zeichensatzes. . 51
3.6 Bild der Infrarotkamera. 51
3.7 Anwendung des Binomialfilters auf das Kamerabild. 53
3.8 Bildvorverarbeitung 54
3.9 Kamerabild nach der Vorverarbeitung bei Berührung mit der Hand. . 55
3.10 Berührpunkterkennung durch Segmentierung 57
3.11 Berührpunkterkennung am Beispiel einer Hand und eines Testmusters. 58
3.12 Visualisierung des Erkennungsalgorithmus anhand zweier Punkte. . . 59
3.13 Entwickelte Marker 62
3.14 Ablauf der Markererkennung 64
3.15 Sampling der Markerkanten. 65
3.16 Testapplikation zur Visualisierung der Algorithmen. 68

4.1 Geschwindigkeitsvergleich der Berührpunkterkennung. 72
4.2 Geschwindigkeitsvergleich der Markererkennung. 73
4.3 Rechenzeitverteilung der OpenCL Kernel 75

Listings

3.1 Codeausschnitt des OpenCL Host-Programms. 39
3.2 OpenCL Kernel zur Transposition einer Matrix. 41
3.3 Einfacher sequenzieller Transpositionsalgorithmus für CPUs. 42

Literaturverzeichnis

[ATI 2009] ATI: ATI Stream Computing / Advanced Micro Devices, Inc. 2009

[Bayazit u. a. 2009] Bayazit, Mark ; Couture-Beil, Alex ; Mori, Greg: Real-time Motion-based Gesture Recognition using the GPU / School of Computing Science Simon Fraser University Burnaby, BC, Canada. 2009

[Bencina u. a. 2005] Bencina, Ross ; Kaltenbrunner, Martin ; Jorda, Sergi: Improved Topological Fiducial Tracking in the reacTIVision System. In: *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Workshops*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2372–2–3, S. 99

[Bender u. Wagner 2007] Bender, Pascal ; Wagner, Philippe O.: Advanced BarCode Information / Fachhochschule Nordwestschweiz (FHNW). Version: 07 2007. `http://web.fhnw.ch/projekte/computervision/fhbb/studierendenprojekte/2007/ABCInfo_Dokumentation.pdf`. 2007

[Blelloch 1993] Blelloch, Guy E.: Prefix Sums and Their Applications / School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213-3890. 1993

[Bochem u. a. 2009] Bochem, Alexander ; Herpers, Rainer ; Kent, Kenneth B.: *Acceleration of Blob Detection Within Images in Hardware*. 2009

[Bourke 2002] Bourke, Paul: *Lens Correction And Distortion*. `http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/lenscorrection/`. Version: 04 2002

[Boyd u. a. 2010] Boyd, Chas ; Huang, Xin ; Pritchard, Cody ; Gee, Kev: DirectX 11 DirectCompute: A Teraflop for Everyone. In: *Gamefest - Microsoft Game Technology Conference*, 2010

[Casazza 2009] Casazza, Jeff: First the Tick, Now the Tock: Intel® Microarchitecture (Nehalem) / Intel Corporation. 2009

[Costanza u. Robinson 2003] Costanza, E. ; Robinson, J.: *A Region Adjacency Tree Approach to the Detection and Design of Fiducials*. `http://eprints.ecs.soton.ac.uk/20958/1/vvg.pdf`. Version: 2003

[Didier u. a. 2008] Didier, Jean-Yves ; Ababsa, Fakhr eddine ; Mallem, Malik: Hybrid camera pose estimation combining square fiducials localisation technique and orthogonal iteration algorithm. In: *International Journal of Image and Graphics (IJIG) 8*, 2008, 169–188

[Evonik 2008] Evonik: *PLEXIGLAS® EndLighten: Licht in neuer Dimension.* Version: 01 2008. `http://www.plexiglas-shop.com/pdfs/de/232-19_P_EndLighten_de.pdf`, Abruf: 01. 10. 2010

[Gamma u. a. 1995] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995. – 293–303 S.

[Gieselmann 2010] Gieselmann, Hartmut: *GDC: Bewegungssteuerung mit Playstation Move.* Version: 03 2010. `http://www.heise.de/newsticker/meldung/GDC-Bewegungssteuerung-mit-Playstation-Move-951886.html`, Abruf: 01. 10. 2010

[Glebov 2008] Glebov, Roman: Verfahren zum effzienten Tracking schneller Objekte - implementiert mit CUDA am Beispiel Tischtennis spielender Roboter / Bachelorarbeit im Studiengang Informatik, Hochschule für Technik Stuttgart. 2008

[Hale 2006] Hale, Dave: Recursive Gaussian filters / Center for Wave Phenomena, Colorado School of Mines, Golden CO 80401, USA. Version: 2006. `http://www.cwp.mines.edu/Meetings/Project06/cwp546.pdf`. 2006

[Han 2005] Han, Jefferson Y.: Low-cost multi-touch sensing through frustrated total internal reflection. In: *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology.* New York, NY, USA : ACM, 2005. – ISBN 1–59593–271–2, S. 115–118

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

[Hirzer 2008] Hirzer, Martin: Marker Detection for Augmented Reality Applications / Inst. for Computer Graphics and Vision Graz University of Technology, Austria. 2008

[Jähne 2005] Jähne, Bernd: *Digitale Bildverarbeitung.* 6. Berlin : Springer, 2005. – ISBN 978–3–540–24999–3

[Kaltenbrunner 2009] Kaltenbrunner, Martin: reacTIVision and TUIO: A Tangible Tabletop Toolkit / Universitat Pompeu Fabra 08018 Barcelona, Spain. 2009

[Kato u. Billinghurst 1999] Kato, Hirokazu ; Billinghurst, Mark: Marker Tracking and HMD Calibration for a Video-based Augmented Reality Conferencing System / Faculty of Information Sciences, Hiroshima City University; Human

Interface Technology Laboratory, University of Washington. Version: 1999. http://www.hitl.washington.edu/artoolkit/Papers/IWAR99.kato.pdf. 1999

[Kato u. a. 2010] Kato, Hiroko ; Tan, Keng T. ; Chai, Douglas: *Barcodes for Mobile Devices*. Cambridge University Press, New York, 2010

[Kerr 2007] Kerr, Douglas A.: *Derivation of the "Cosine Fourth" Law for Falloff of Illuminance Across a Camera Image*. http://www.dougkerr.net/pumpkin/articles/Cosine_Fourth_Falloff.pdf. Version: 2007

[Khronos Group 2009] Khronos Group: *The OpenCL Specification*. http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf. Version: 10 2009

[Kumar u. a. 2009] Kumar, Praveen ; Palaniappan, Kannappan ; Mittal, Ankush ; Seetharaman, Guna: Parallel Blob Extraction Using the Multi-core Cell Processor / Dept. of Computer Science, Univ. of Missouri, Columbia, MO 65211, USA. 2009

[Leech 2010] Leech, Jon: *ARB_cl_event*. http://www.opengl.org/registry/specs/ARB/cl_event.txt. Version: 07 2010

[Longacre u. Hussey 1997] Longacre, Andrew Jr. ; Hussey, Rob: *Two Dimensional Data Encoding Structure And Symbology For Use With Optical Readers*. http://www.freepatentsonline.com/5591956.pdf. Version: 5 1997

[Merki 2003] Merki, Heinrich G.: *Informationen zum Aztec Code*. Version: 03 2003. http://www.ades.ch/aaaa/pdf/Informationen%20zum%20Aztec%20Code.pdf

[Meyers 2010] Meyers, Scott: *CPU Caches and Why You Care*. http://aristeia.com/TalkNotes/PDXCodeCamp2010.pdf. Version: 2010

[Microsoft 2009] Microsoft: *Microsoft Launches New Product Category: Surface Computing Comes to Life in Restaurants, Hotels, Retail Locations and Casino Resorts*. Version: 05 2009. http://www.microsoft.com/presspass/press/2007/may07/05-29MSSurfacePR.mspx, Abruf: 01. 10. 2010

[Microsoft 2010] Microsoft: *Offizielle Kinect Homepage*. Version: 08 2010. http://www.xbox.com/de-DE/kinect, Abruf: 01. 10. 2010

[Muller 2008] Muller, L.Y.L.: Multi-touch displays: design, applications and performance evaluation / Universiteit van Amsterdam. 2008

[Munshi 2010] Munshi, Aaftab: *The OpenCL Specification*. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf. Version: 06 2010

[NUI Group Authors 2009] NUI Group Authors: *Multitouch Technologies*. 1.01. NUI Group, 2009 http://nuicode.com/attachments/download/115/Multi-Touch_Technologies_v1.01.pdf

[NVIDIA 2008] NVIDIA: NVIDIA GeForce 8800 GPU Architecture Overview / NVIDIA Corporation. Version: 2008. www.nvidia.com/object/IO_37100.html. 2008

[NVIDIA 2009] NVIDIA: NVIDIA CUDA C Programming Best Practices Guide / NVIDIA Corporation. Version: 2009. http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_BestPracticesGuide_2.3.pdf. 2009

[NVIDIA 2010] NVIDIA: NVIDIA GF100 World's Fastest GPU Delivering Great Gaming Performance with True Geometric Realism / NVIDIA Corporation. 2010

[Osram 2007] OSRAM: *IR-Lumineszenzdiode (850 nm) mit hoher Ausgangsleistung: SFH 4350*. http://catalog.osram-os.com/media/_en/Graphics/00042965_0.pdf. Version: 03 2007

[Scharr 2000] SCHARR, Dipl. Phys. H.: *Optimale Operatoren in der Digitalen Bildverarbeitung*, Naturwissenschaftlich-Mathematischen Gesamtfakultät der Ruprecht-Karls-Universität Heidelberg, Diss., 2000

[Segal u. Akeley 2009] SEGAL, Mark ; AKELEY, Kurt: *The OpenGL Graphics System: A Specification Version 3.2 (Core Profile)*. http://www.opengl.org/registry/doc/glspec32.core.20091207.pdf. Version: 12 2009

[Segal u. Akeley 2010] SEGAL, Mark ; AKELEY, Kurt: *The OpenGL Graphics System: A Specification Version 4.1 (Core Profile)*. http://www.opengl.org/registry/doc/glspec41.core.20100725.pdf. Version: 07 2010

[Seidle 2006] SEIDLE, Nathan: *Wii-mote guts*. Version: 12 2006. http://www.sparkfun.com/commerce/tutorial_info.php?tutorials_id=43, Abruf: 01. 10. 2010

[Sony 2010] SONY: *Offizielle PlayStation Move Homepage*. Version: 10 2010. http://us.playstation.com/ps3/playstation-move/, Abruf: 01. 10. 2010

[StVO 2009] STVO: *Straßenverkehrs-Ordnung*. http://www.bmvbs.de/Anlage/original_1094697/Strassenverkehrs-Ordnung-Stand-01.09.2009.pdf. Version: 09 2009

[Sutter 2007] SUTTER, Herb: *Machine Architecture(Things Your Programming Language Never Told You)*. http://www.nwcpp.org/Downloads/2007/Machine_Architecture_-_NWCPP.pdf. Version: 2007

[Toub u. a. 2008] TOUB, Stephen ; OSTROVSKY, Igor ; YILDIZ, Huseyin: False Sharing / Microsoft Corporation. Version: 08 2008. http://msdn.microsoft.com/en-us/magazine/cc872851.aspx. 2008

[Wagner 2007] Wagner, Daniel: ARToolKitPlus for Pose Tracking on Mobile Devices / Institute for Computer Graphics and Vision, Graz University of Technology. Version: 02 2007. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.157.1879&rep=rep1&type=pdf. 2007